1 Le système de fichiers

Il existe différents types de fichiers :

— Fichiers ordinaires : Les fichiers utilisateurs (ASCII ou binaire)
— Répertoires : des fichiers systèmes indiquant la structure du système de fichiers
— Fichiers spéciaux caractère : modélise les E/S (terminaux, imprimante . . .)
— Fichiers spéciaux bloc : modélise les disques.

Voici les caractéristiques des systèmes de fichiers qui existent dans les deux mondes Microsoft :
— Windows : FAT-16 ou FAT-32
  — Aucune différence entre JEAN et Jean.
  — Utilisation d'extensions (.exe, .jpg, . . .).
— Windows : NTFS (64bits = 264octets)
  — Différence entre JEAN et Jean dans la notation.
  — Utilisation d'extensions.
— Linux (Unix) : EXT/EXT2/EXT3/EXT4
  — 2go/2To/2To/16To)
  — Différence entre JEAN et Jean.
  — Extensions superflues.

1.1 La structure arborescente sous UNIX.

Les fichiers sont organisés en répertoires et sous-répertoires, formant une arborescence. La figure suivante montre une exemple d'arborescence sous UNIX :

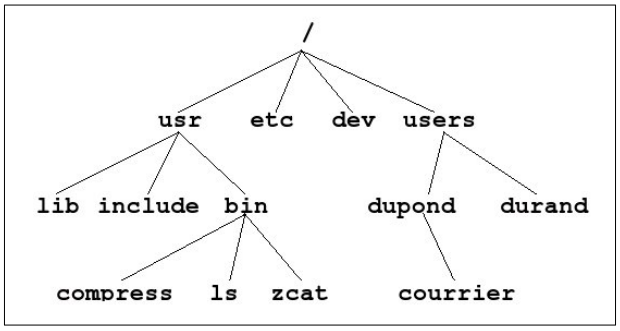

Figure 1 – Exemple d'arborescence sous UNIX.

La racine de l'arbre est le répertoire / (en haut). Ce répertoire contient ici 4 sous-répertoires.

Dans chaque répertoire, on trouve au moins deux fichiers, nommés . (point) et .. (point point). Le premier (.) permet de référencer le répertoire lui même, et le second (..) d'accéder au répertoire parent (du dessus).

1.2 Chemins absolus et relatifs.

Pour désigner un fichier quelconque, on peut utiliser soit un chemin absolu, soit un chemin relatif. Un chemin absolu spécifie la suite des répertoires à traverser en partant de la racine, séparés par des caractères / (et non \ comme sous DOS). Par exemple, le chemin :

/usr/bin/compress

désigne le fichier *compress*, qui se trouve dans le répertoire *bin*, lui même dans le répertoire *usr* de la racine. Le premier caractère / indique qu'il s'agit d'un chemin absolu.

Il est souvent pratique d'utiliser un chemin relatif, à partir du répertoire courant. Par exemple, si l'on travaille dans le répertoire *dupond* de la figure 1, on peut accéder au fichier durand en spécifiant le chemin :

../durand

Du même endroit, on peut accéder au fichier *compress* via le chemin :

../../usr/bin/compress

Dans ce cas précis, il est bien sûr plus simple d'utiliser le chemin absolu. Tout chemin qui ne commence pas

par un caractère / (prononcé slash) est interprété comme un chemin relatif au répertoire courant. On peut ainsi accéder aux fichiers du répertoire courant en donnant simplement leur nom.

1.3 Types de fichiers.

Nous appelons fichier tout point dans l'arborescence des fichiers. Tous ne correspondent donc pas à des fichiers de données ordinaires. On distingue 5 types de fichiers :

- les fichiers ordinaires, qui contiennent des données. UNIX ne fait aucune différence entre les fichiers de texte et les fichiers binaires. Dans un fichier texte, les lignes consécutives sont séparées par un seul caractère '\n'.
- les répertoires, qui contiennent une liste de références à d'autres fichiers UNIX ;
- les fichiers spéciaux, associés par exemple à des pilotes de périphériques ;
- les tubes et sockets, utilisés pour la communication entre processus ;
- les liens symboliques (fichiers "pointant" sur un autre fichier).

1.3.1 La commande ls

La commande ls permet de lister des informations sur les fichiers et répertoires. la syntaxe est :

ls [-options] chemin$_1$ chemin$_2$ ...chemin$_n$

chemin$_i$ est un nom de fichier ou de répertoire. Si c'est un fichier, affiche sa description ; si c'est un répertoire, affiche la description de son contenu. Principales options :
-a liste tous les fichiers (y compris les .* normalement cachés).
-l format long (taille, date, droits, etc).
-d décrit le répertoire et non son contenu.
-i affiche les numéros d'inode des fichiers.

1.4 Les droits des fichiers.

Justement en effectuant la commande ls -l, on obtient les informations sur les droits des fichiers. Par exemple :

Chaque fichier a plusieurs propriétés associées : le propriétaire, le groupe propriétaire, la date de dernière modification, et les droits d'accès. Dans cet exemple, nous voyons les permissions standard d'un répertoire et de deux fichiers :

```
sas ~/DEA $ ls -l
total 205
drwxr-xr-x    2 toto phy03        512 Jan 16 10:02 fiches/
-rw-r--r--    1 toto phy03      72008 Oct  2  2003 article.dvi
-rw-r--r--    1 toto phy03     145905 Oct  2  2003 article.pdf
```

FIGURE 2 – Listage des détails de fichiers et répertoires.

L'utilisateur propriétaire est toto, le groupe propriétaire est phys3. Les 10 premiers caractères représentent le bloc de permissions, c'est à dire le type et les droits. Voici la signification :

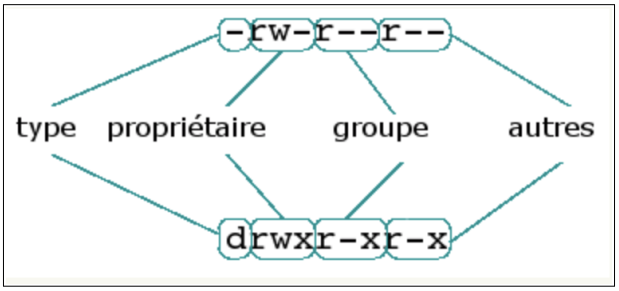

FIGURE 3 – Signification des 10 premiers caractères.

Type : le premier caractère du bloc de permissions indique le type du fichier : - pour un fichier normal, d pour un répertoire (voir plus haut).

Ensuite chaque groupe de 3 signes représente les droits sucessivement du propriétaire, des utilisateurs du groupe propriétaire et enfin des autres utilisateurs. La signification pour les fichiers est la suivante :

— r ou - : droit de lire (r pour read) le fichier (r pour oui, - pour non)
— w ou - : droit d'écrire (w pour write) dans le fichier (c'est à dire modifier son contenu)
— x ou - : droit d'exécuter (x pour execute) le fichier.

Pour les dossiers la signification est la suivante :

— r ou - : droit de lister (r pour read) le contenu du répertoire (r pour oui, - pour non)
— w ou - : droit de créer, renommer ou supprimer des fichiers ou répertoires (w pour write) dans le répertoire.
— x ou - : droit de traverser (x pour cross) le répertoire (en faire son répertoire courant).

1.5 La commande chown

Cette commande sert à changer le propriétaire et/ou le groupe propriétaire d'un fichier ou d'un répertoire. Seul l'administrateur peut changer le propriétaire d'un fichier ou répertoire.

chown user1 rep1

Cette commande désigne user1 comme propriétaire pour rep1.

chown user1 :group1 rep1

Cette commande désigne user1 comme propriétaire pour rep1 et group1 comme groupe propriétaire .

chown :group1 rep1

Cette commande ne change que le groupe.

chown -R user1 rep1

Cette fois, la commande est récursive et user1 devient propriétaire de rep1 et tout ce qu'il contient.

1.6 La commande umask

Lorsqu'un nouveau fichier est créé, celui-ci obtient automatiquement certains paramètres :
— **Propriétaires** : Par défaut, le propriétaire d'un nouveau fichier est son créateur et le groupe propriétaire, le groupe principal de son créateur. Par exemple, si l'utilisateur toto, dont le groupe principal est utilisateurs, crée un nouveau fichier ou dossier, celui-ci appartient à toto :utilisateurs ;
— **Permissions** : Les permissions accordées par défaut sont celles déterminées par un paramètre particulier appelé le **masque utilisateur** (ou **user mask**). Dans Ubuntu, le masque utilisateur par défaut accorde les permissions rwxr-xr-x correspondant à un umask = 022 :
  — le propriétaire du fichier dispose des permissions de lecture, d'écriture et d'exécution ;
  — le groupe propriétaire dispose des droits de lecture et d'exécution, mais pas d'écriture ;
  — les autres disposent des droits de lecture et d'exécution, mais pas d'écriture.

On peut modifier le masque par défaut avec la commande umask :

umask 0xyz

xyz étant le nouveau masque en octal. Ses bits à 1 représentent les droits otés lors de la création d'un nouveu fichier ou répertoire.

1.7 Les liens.

Sous Unix, un même fichier peut avoir plusieurs noms : c'est ce qu'on appelle un **lien** On les crée avec la commande **ln**. Sous Unix, il existe deux types de **liens** entre fichiers, que l'on nomme généralement liens matériels (ou physiques ou dur) et liens symboliques (ou logiques).

1.8 Les liens physiques.

Par exemple, on a déjà un fichier fichier1, et si on exécute la commande :

ln fichier1 fichier2

On obtient fichier2 qui est un nouveau lien dur sur le fichier fichier1. fichier1 et fichier2 représentent le même fichier : si on modifie fichier2, les changements apparaîtront aussi dans fichier1. En particulier, il existe un seul exemplaire du fichier sur le disque : la création d'un lien dur ne prend pas de place.

Si maintenant on supprime fcichier 1, le fichier existe encore sous le nom de fichier2 (et réciproquement !). Un fichier est définitivement supprimé quand son dernier lien est effacé.

1.8.1 Les liens symboliques.

Comme un lien physique, un lien symbolique est un alias d'un fichier. Mais contrairement au lien physique, le lien symbolique n'est qu'un alias : on distingue bien le fichier lui-même et les liens, qui ne veulent plus rien dire si le fichier est supprimé.

Par exemple, si on a toujours fichier1 et si on veut créer un lien symbolique fichier2 on a la commande :.

ln -s fichier1 fichier2

La encore, si on modifie fichier1, fichier2 sera également affecté, et vice-versa. Par contre, si on supprime fichier1 et qu'on essaye de modifier fichier2, le système va afficher une erreur : fichier2 est un lien symbolique invalide, puisque le fichier qu'il désignait a disparu.

Remarques :

— on peut créer plusieur liens (avec une liste de fichiers) si le dernier nom est un répertoire.
— Si dans la commande ln, le dernier nom est un répertoire, les lien sont créés dans ce répertoire en gardant les noms des fichiers.

2 Les bibliothèques de fonctions d'entrées/sorties

Les programmeurs C sous GNU/Linux ont deux jeux de fonctions d'entrées/sorties à leur disposition. La bibliothèque C standard fournit des fonctions d'E/S : printf, fopen, etc. Le noyau Linux fournit un autre ensemble d'opérations d'E/S qui opèrent à un niveau inférieur à celui des fonctions de la bibliothèque C. C'est cet ensemble qu'on va étudier. Il faut savoir que l'utilisation ce ces fonctions peut s'avérer être moins portable que l'utilisation des fonctions standards.

3 Création de fichiers et répertoires

Pour créer un fichier il y a deux fonctions utilisables :
— creat()
— open()
Voici le prototype de creat() :

```
int creat(const char *pathname, mode_t mode);
```

— pathname est le chemin absolu ou relatif du fichier à créer.
— mode indique les permissions à utiliser si un nouveau fichier est créé. Cette valeur est modifiée par le umask du processus : la véritable valeur utilisée est : (mode & ∼ umask).
— la valeur de retour représente **le descripteur de fichier** si creat réussit, ou -1 en cas d'échec (voir la suite)

Les valeurs qu'on peut affecter avec mode sont des combinaisons valides des constantes suivantes :

— S_IRWXU 00700 L'utilisateur (propriétaire du fichier) a les autorisations de lecture, écriture, exécution.
— S_IRUSR 00400 : L'utilisateur a l'autorisation de lecture.
— S_IWUSR 00200 : L'utilisateur a l'autorisation d'écriture.
— S_IXUSR 00100 : L'utilisateur a l'autorisation d'exécution.
— S_IRWXG 00070 : Le groupe a les autorisations de lecture, écriture, exécution.
— S_IRGRP 00040 : Le groupe a l'autorisation de lecture.
— S_IWGRP 00020 : Le groupe a l'autorisation d'écriture.
— S_IXGRP 00010 : Le groupe a l'autorisation d'exécution.
— S_IRWXO 00007 : Tout le monde a les autorisations de lecture, écriture, exécution.
— S_IROTH 00004 : Tout le monde a l'autorisation de lecture.
— S_IWOTH 00002 : Tout le monde a l'autorisation d'écriture.
— S_IXOTH 00001 : Tout le monde a l'autorisation d'exécution.

4 ouverture d'un fichier

La fonction open() est utiliser pour ouvrir un fichier pour lire ou écrire dans ce fichier. Il est nécessaire nécessaire de d'abord ouvrir le fichier avant de pouvoir écrire dedans ou de lire son contenu.

Voici les prototypes de la fonction open() :

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- pathname est le chemin absolu ou relatif du fichier à créer.
- flags doit comporter une des options suivantes :
  - O_RDONLY : on ouvre en lecture,
  - O_WRONLY : on ouvre en écriture,
  - O_RDWR : on ouvre en lecture/écriture

  On peut encore combiner la valeur en utilisant un OU logique (|) avec les options ou attributs de fichier, dont les principaux sont :
  - O_APPEND : on ouvre en écriture à la fin du fichier (mode ajout),
  - O_CREAT : si le fichier n'existe pas on le crée,
- mode indique les permissions à utiliser si un nouveau fichier est créé. C'est identique à la fonction creat() .
- La valeur de retour est le descripteur de fichier (entier) qui permettra de réaliser des lectures ou écritures du fichier.

Remarque : la fonction creat() est équivalente à la fonction open avec flags égal à O_CREAT | O_WRONLY | O_TRUNC.

4.1 La fonction read()

C'est la fonction qui lit des données depuis un descripteur de fichier. Le prototype en est :

```
int read(int fd, void *buf, int count);
```

Description :

read() lit jusqu'à count octets depuis le descripteur de fichier fd dans le tampon pointé par buf. Si count vaut zéro, read() renvoie zéro et n'a pas d'autres effets.

Paramètres :

- **fd** : c'est le descripteur du fichier à lire (obtenu avec open)
- **buf** : adresse du buffer dans lequel lire les octets.
- **count** : c'est le nombre d'octets maximum à lire.

Valeur de retour : le nombre d'octets effectivement lus. Ou -1 si la lecture échoue, auquel cas errno contient le code d'erreur, et la position de la tête de lecture est indéfinie.

4.2 La fonction write()

C'est la fonction qui écrit des données dans un descripteur de fichier. Le prototype en est :

```
int write(int fd, void *buf, int count);
```

Description :

write() écrit jusqu'à count octets vers le descripteur de fichier fd depuis le tampon pointé par buf.

Paramètres :

- **fd** : c'est le descripteur du fichier dans lequel écrire (obtenu avec open)
- **buf** : adresse du buffer qui contient les octets à écrire.
- **count** : c'est le nombre d'octets à écrire.

Valeur de retour : le nombre d'octets effectivement écrits (il peut y avoir insuffisamment de place sur le support d'écriture). Ou -1 si l'écriture échoue, auquel cas errno contient le code d'erreur, et la position de la tête de lecture est indéfinie.

4.3 La fonction lseek()

La fonction **lseek()** positionne la tête de lecture/écriture dans un fichier. Voici le prototype :

```
int lseek(int fd, int offset, int whence);
```

Description : La fonction lseek() place la tête de lecture/écriture à la position offset dans le fichier ouvert associé au descripteur fd en suivant la directive whence.

Paramètres :

- **fd** : c'est le descripteur du fichier dans lequel on veut déplacer la tête de lecture/écriture (obtenu avec open)
- **offset** : valeur du déplacement à effectuer (en nombre octets)
- **whence** : constante donnant la nature du déplacement :
  - SEEK_SET : La tête est placée à offset octets depuis le début du fichier.
  - SEEK_CUR : La tête de lecture/écriture est avancée de offset octets.
  - SEEK_END : La tête est placée à la fin du fichier plus offset octets.

Valeur de retour : lseek(), s'il réussit, renvoie le nouvel emplacement, mesuré en nombre d'octets depuis le début du fichier. En cas d'échec, la valeur -1 est renvoyée, et errno contient le code d'erreur.

4.4 La fonction close()

Cette fonction permet de fermer un fichier ouvert (on rend le descripteur de fichier). Son prototype :

```
int close(int fd, int offset);
```

Paramètre :

- **fd** : c'est le descripteur du fichier qu'on veut fermer (obtenu avec open)

Valeur de retour : close renvoie 0 s'il réussit, ou -1 en cas d'échec, auquel cas errno contient le code d'erreur.

4.5 La fonction rename()

La fonction **rename()** renomme un fichier, en le déplacant vers un autre répertoire si besoin est. Son prototype :

```
int rename(const char *oldpath, const char *newpath);
```

Paramètres :

- **oldpath** : chemin du fichier à renommer (relatif ou absolu)
- **newpath** : nouveau chemin du fichier à renommer (relatif ou absolu)

Valeur de retour : rename renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.

4.6 La fonction mkdir()

La fonction **mkdir()** crée un nouveau répertoire. Le prototype est :

```
int mkdir(const char *pathname, mode_t mode);
```

Paramètres :

- **pathname** : nom du répertoire à créer (relatif ou absolu)
- **mode** : droits (mode & ~umask)

Valeur de retour : mkdir renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.

4.7 La fonction rmdir()

La fonction **rmdir()** supprime un répertoire (celui-ci doit être vide). Le prototype est :

```
int rmdir(const char *pathname);
```

Paramètre :

- **pathname** : nom du répertoire à supprimer (relatif ou absolu)

Valeur de retour : rmdir renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.

4.8 La fonction chdir()

La fonction **chdir()** change le répertoire courant . Le prototype est :

```
int chdir(const char *path);
```

Paramètre :
— **path** : nom du répertoire à atteindre (relatif ou absolu)
Valeur de retour : chdir renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.

4.9 La fonction link()

La fonction **link()** crée un nouveau nom pour un fichier . Le prototype est :

```
int link (const char *oldpath, const char *newpath);
```

Paramètres :
— **oldpath** : nom du fichier pour lequel créer le lien (relatif ou absolu)
— **newpath** : nom du fichier lien à créer(relatif ou absolu)
Valeur de retour : link renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.

4.10 La fonction unlink()

La fonction **unlink()** détruit un nom et éventuellement le fichier associé. Le prototype est :

```
int unlink(const char *pathname);
```

Paramètre :
— **pathname** : nom à supprimer (relatif ou absolu)
Description : unlink() détruit un nom dans le système de fichiers. Si ce nom était le dernier lien sur un fichier, et si aucun processus n'a ouvert ce fichier, ce dernier est effacé, et l'espace qu'il utilisait est rendu disponible. Si le nom était le dernier lien sur un fichier, mais qu'un processus conserve encore le fichier ouvert, celui-ci continue d'exister jusqu'à ce que le dernier descripteur le référençant soit fermé.

Si le nom correspond à un lien symbolique, le lien est supprimé.

Si le nom correspond à une socket, une FIFO, le nom est supprimé mais les processus qui ont ouvert l'objet peuvent continuer à l'utiliser.

Valeur de retour : unlink renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.

4.11 La fonction opendir()

Cette fonction ouvre un répertoire. Le prototype :

```
DIR* opendir(const char *name);
```

Paramètre :
— **name** : nom du répertoire à ouvrir (relatif ou absolu)
Description :

La fonction opendir() ouvre un flux répertoire correspondant au répertoire name, et renvoie un pointeur sur ce flux. Le flux est positionné sur la première entrée du répertoire.

Valeur de retour : opendir() renvoit un pointeur sur le flux répertoire ou NULL si une erreur se produit, auquel cas errno contient le code d'erreur.

4.12 La fonction readdir()

Cette fonction permet d'obtenir une élément d'un répertoire. Le prototype est :

```
struct dirent *readdir(DIR *dir);
```

Paramètre :

— **dir** : pointeur de flux dont on veut lister les entrées.
Valeur de retour : La fonction readdir() renvoie un pointeur sur une structure dirent représentant l'entrée suivante du flux répertoire pointé par dir. Elle renvoie NULL à la fin du répertoire, ou en cas d'erreur. Si une erreur se produit, errno contient le code d'erreur.

Voici la description de la structure **dirent** :

```
struct dirent {
    ino_t          d_ino;       /* numéro d'inoeud */
    off_t          d_off;       /* décalage jusqu'à la dirent suivante */
    unsigned short d_reclen;    /* longueur de cet enregistrement */
    unsigned char  d_type;      /* type du fichier */
    char           d_name[256]; /* nom du fichier */
};
```

4.13 La fonction rewinddir()

Cette fonction réinitialise un flux répertoire , c'est à dire repositionne le flux au début de la liste. Le prototype est :

```
void rewinddir(DIR *dir);
```

Paramètre :
— **dir** : pointeur de flux dont on veut lister les entrées.
Valeur de retour : Aucune.

4.14 La fonction closedir()

Cette fonction ferme un flux répertoire. Le prototype est :

```
void rewinddir(DIR *dir);
```

Paramètre :
— **dir** : pointeur de flux qu'on veut fermer.
Valeur de retour : La fonction closedir() renvoie 0 si elle réussit, ou -1 si elle échoue, auquel cas errno contient le code d'erreur.