Chapitre 4

Le système de fichiers

La mémoire **secondaire** d'un ordinateur comprend différentes formes de supports : disques, disquettes, CD-ROM etc. . .dont les caractéristiques communes les plus essentielles sont qu'ils :

- permettent de stocker durablement l'information ;
- sont plus longs d'accès que la mémoire principale ;
- permettent un accès séquentiel plus rapide que l'accès direct.

L'interêt de cette mémoire vient donc de la possibilité d'y stocker en permanence de gros volumes d'informations. La gestion de ces informations pour les mettre à la disposition des processus du système doit prendre en compte les contraintes liées au temps d'accès. Cette gestion est à la charge du *système de fichiers* qui constitue l'objet de ce chapitre.

4.1 Fichiers

Les informations sont stockées en mémoire secondaire sous forme de **fichiers**. Le fichier est la plus petite unité d'information accessible à l'utilisateur en mémoire secondaire.

4.1.1 Noms et types

Pour permettre à l'utilisateur de distinguer les fichiers, le système permet d'associer un *nom* à chaque fichier. Sous Unix ainsi que sous Windows les noms de fichiers comportent des *extensions* qui donne une idée de leur fonction. Un fichier source en langage C aura par exemple pour extension `.c` ou `.h`.

Deux types de fichiers peuvent être présents dans un système :

1. **fichiers de donnés** : contiennent uniquement des donnés (textes, images, sons etc. . .).
2. **fichiers binaires de code** : permettent de stocker des programmes dans un langage directement compréhensible par le processeur.

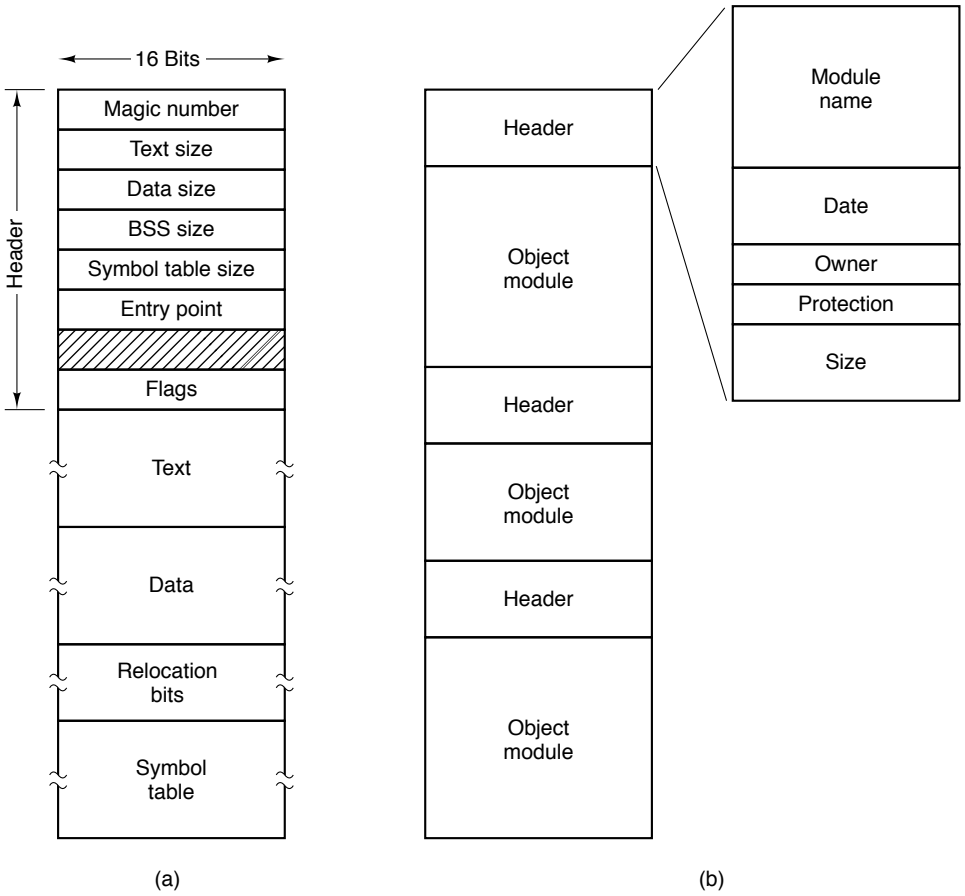

FIG. 4.1 – Exemples de fichiers binaires de code

La figure 4.1 montre un exemple de stucture interne possible pour deux types de fichiers binaires contenant du code :

(a) un *fichier exécutable* qui correspond à du code pouvant être chargé en mémoire principale sous forme de processus. Ce fichier comprend : un nombre magique permettant de l'identifier comme un fichier exécutable, le texte (code) et les données du programme, ainsi que d'autres informations qui permettrons au systeme de le charger en mémoire principale.

(b) un *fichier d'archive* constitué d'une suite de modules (procédures) compilés, mais non liés.

4.1.2 Attributs

Du point de vue du système un fichier se caractèrise par différents attributs, par exemple :

- sa taille ;
- son type ;
- son propriétaire ;
- les droits d'accès ;
- dates : création, dernier accès, dernière modification ;
- etc...

4.1.3 Attributs sous Unix

Lecture des attributs d'un fichier

Afin d'obtenir les attributs d'un fichier sous Unix, on utilise l'appel système suivant :

```
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *file_name, struct stat *buf);
```

`stat` recupère les attributs du fichier `file_name` dans la structure `buf` déclarée comme suit :

```
struct stat
 {
     dev_t         st_dev;      /* Périphérique                */
     ino_t         st_ino;      /* Numéro i-noeud              */
     umode_t       st_mode;     /* Protection                  */
     nlink_t       st_nlink;    /* Nb liens matériels          */
     uid_t         st_uid;      /* UID propriétaire            */
     gid_t         st_gid;      /* GID propriétaire            */
     dev_t         st_rdev;     /* Type périphérique           */
     off_t         st_size;     /* Taille totale en octets     */
     unsigned long st_blksize;  /* Taille de bloc pour E/S     */
     unsigned long st_blocks;   /* Nombre de blocs alloués     */
     time_t        st_atime;    /* Heure dernier accès         */
     time_t        st_mtime;    /* Heure dernière modification */
     time_t        st_ctime;    /* Heure dernier changement    */
 };
```

Modification des droits d'accès sur un fichier

Certains appels système permettent de modifier les attributs d'un fichier. La fonction suivante permet de modifier le mode d'accès d'un fichier :

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *file_name, mode_t mode);
```

Le mode est spécifié par un OU binaire (|) entre les éléments suivants :
- `S_ISUID` : modification du numéro d'utilisateur (UID) à l'exécution,
- `S_ISGID` : modification du numéro de groupe (GID) à l'exécution,
- `S_ISVTX` : positionner le sticky bit pour conserver le code du programme en mémoire après exécution,

– `S_IRUSR (S_IREAD)` : accès en lecture pour le propriétaire,
– `S_IWUSR (S_IWRITE)` : accès en écriture pour le propriétaire,
– `S_IXUSR (S_IEXEC)` : accès en exécution/parcours par le propriétaire,
– `S_IRGRP` : accès en lecture pour le groupe,
– `S_IWGRP` : accès en écriture pour le groupe,
– `S_IXGRP` : accès en exécution/parcours pour le groupe,
– `S_IROTH` : accès en lecture pour les autres,
– `S_IWOTH` : accès en écriture pour les autres,
– `S_IXOTH` : accès en exécution/parcours pour les autres.

Modification de l'utilisateur propriétaire d'un fichier

La fonction qui suit :

```
#include <sys/types.h>
#include <unistd.h>

int chown(const char *file_name, uid_t owner, gid_t group);
```

permet de modifier le propriétaire et le groupe du fichier désigné par le chemin `file_name`. Seul le Super-utilisateur peut changer le propriétaire d'un fichier. Le propriétaire peut changer le groupe du fichier pour n'importe quel groupe auquel il appartient. Le Super-Utilisateur peut changer le groupe arbitrairement.

4.1.4 Opérations

Le système de fichiers doit permettre de faire les opérations suivantes sur chaque fichier :

1. créer,
2. supprimer,
3. ouvrir,
4. fermer,
5. lire,
6. écrire,
7. ajouter,
8. modifier la position courante dans le fichier,
9. lire les attributs,
10. fixer les attributs,
11. renomer le fichier.

4.1.5 Opérations sur les fichiers sous Unix

Ouverture et Fermeture

L'appel système `open` essaye d'ouvrir un fichier et retourne un descripteur de fichier qui est un entier à utiliser pour désigner le fichier lorsqu'on effectue d'autres opérations sur un fichier ouvert (lecture, écriture etc...).

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

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

`flags` est l'un des éléments `O_RDONLY`, `O_WRONLY` ou `O_RDWR`. ces drapeaux réclament respectivement l'ouverture du fichier en lecture seule, écriture seule, ou lecture/écriture.

`flags` peut aussi être un OU binaire (|) avec un ou plusieurs des éléments suivants :

- `O_CREAT` : créer le fichier s'il n'existe pas,
- `O_APPEND` : le fichier est ouvert en mode ajout. Initialement, et avant chaque write, la tête de lecture/écriture est placée a la fin du fichier,
- `O_NONBLOCK` ou `O_NDELAY` : le fichier est ouvert en mode non-bloquant. Ni la fonction `open` ni aucune autre opération ultérieure sur ce fichier ne laissera le processus appelant en attente,
- `O_SYNC` : Le fichier est ouvert en écriture synchronisée. Chaque appel à `write` sur le fichier bloquera le processus appelant jusqu'à ce que les données aient été écrites physiquement en mémoire secondaire.

`mode` spécifie les permissions à appliquer au fichier. Cette valeur est modifiée par le `umask` du processus : la véritable valeur utilisée est `(mode & ~umask)`.

La fonction :

```
#include <unistd.h>

int close(int fd);
```

ferme le descripteur `fd`, de manière à ce qu'il ne référence plus aucun fichier pour pouvoir être réutilisé.

Lecture et écriture

La fonction :

```
#include <unistd.h>

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

lit jusqu'à `count` octets depuis le descripteur de fichier `fd` dans le tampon pointé par `buf`. La fonction :

```
#include <unistd.h>

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

écrit jusqu'a `count` octets dans le fichier associé au descripteur `fd` depuis le tampon pointé par `buf`. La norme POSIX réclame qu'une lecture avec `read()` effectuée après le retour d'une écriture avec `write()`, renvoie les nouvelles données. Notez que tous les systèmes de fichiers Unix ne sont pas compatibles avec POSIX.

Positionnement

La fonction :

```
#include <sys/types.h>
#include <unistd.h>

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

place la tête de lecture/écriture à la position `offset` dans le fichier associé au descripteur `fd` en suivant la directive `whence` ainsi :

- `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.

La fonction `lseek` permet de placer la tête au-delà de la fin actuelle du fichier. Si des données sont écrites à cet emplacement, une lecture ultérieure de l'espace intermediaire retournera des zéros (jusqu'à ce que d'autres données y soient écrites).

Sauvegarde des données modifiées

Afin de réduire les accès (lents) en mémoire secondaire, les systemes de fichiers d'Unix utilisent un cache en mémoire principale permettant d'effectuer les opérations sur ce cache plutot que sur le disque. Il est nécessaire parfois de synchroniser les fichiers, en sauvegardant les données modifiées du cache dans le disque. Les fonctions suivantes concernent ce type d'opérations.

```
#include <unistd.h>

int sync(void);
int fsync(int fd);
```

La fonction `sync` permet de vider le cache concernant tous les fichiers sur le disque. La fonction `fsync` copie du cache vers le disque toutes les données d'un fichier dont le descriteur est `fd`.

Duplication des descripteurs

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

`dup` et `dup2` créent une copie du descripteur de fichier `oldfd`.

L'ancien descripteur et le nouveau descripteur peuvent être utilisés de manière interchangeable. Ils partagent les verrous, les pointeurs de position et les drapeaux. Par exemple si le pointeur de position est modifié en utilisant `lseek` sur l'un des descripteurs, la position est également changée pour l'autre.

`dup` utilise le plus petit numéro inutilisé pour le nouveau descripteur.

`dup2` transforme `newfd` en une copie de `oldfd`, fermant auparavant `newfd` si besoin est.

4.1.6 Fichiers et communication sous Unix

Verrouillage d'un fichier

L'accès simultané d'un même fichier par plusieurs processus peut poser des problèmes de synchronisation. La fonction :

```
#include <sys/file.h>

int flock(int fd, int operation)
```

permet de placer ou d'enlever un verrou sur un fichier ouvert dont le descripteur est `fd`. Les opérations possibles sont :

- `LOCK_SH` : Verrouillage partagé. Plusieurs processus peuvent disposer d'un verrouillage partagé simultanément sur un même fichier.
- `LOCK_EX` : Verrouillage exclusif. Un seul processus dispose d'un verrouillage exclusif sur un fichier, à un moment donné.
- `LOCK_UN` : Déverrouillage.
- `LOCK_NB` : Ne pas bloquer lors d'un verrouillage. Cette option est utilisée conjointement (OU binaire |) avec les autres opérations.

Un fichier donné ne peut pas posséder simultanément des verrous partagés et des verrous exclusifs.

Verrouillage d'une section d'un fichier

Il est parfois inutile de verouiller tout le fichier lors d'un accès concurrent. Lorsque l'opération d'écriture porte uniquement sur une section du fichier on peut utiliser la fonction :

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, struct flock * lock);
```

`fcntl` permet de réaliser diverses opérations sur le descripteur de fichier `fd`. L'opération en question est déterminée par la valeur de l'argument `cmd`. Pour verrouiller une section d'un fichier il faut utiliser un argument `cmd` égal à `F_SETLK` et transmettre une structure `flock` définie dans `<fcntl.h>` :

```
struct flock
    {
    short int l_type; /* Type de verrou  */
    short int l_whence; /* Type d'emplacement ( SEEK_SET,SEEK_CUR ou SEEK_END) *
    __off_t l_start; /* Emplacement du verrrou dans le fichier  */
    __off_t l_len; /* longueur de la zone verrouillée  */
    __pid_t l_pid; /* Numéro du processus qui possède le verrou  */
    };
```

La fonction `fcntl` :
- active le verrou si `l_type` vaut `F_RDLCK` (verrou partagé) ou `F_WRLCK` (verrou exclusif).
- efface le verrou si `l_type` vaut `F_UNLCK`.

4.2 Répertoires

Le nombre de fichiers présents dans un système peut être très important. Afin de s'y retrouver, on range les fichiers dans des **répertoires** correspondant à leur fonction.

4.2.1 Système de répertoires hiérarchique

Dans le modèle d'organisation de fichiers le plus courant, l'imbrication des répertoires décrit une **arborescence** qui part du **répertoire racine** (qui contient tout) et se termine par les fichiers (qui ne peuvent inclure ni fichier ni répertoire). La figure 4.2 montre l'exemple d'une arborescence.

Sous Unix, le répertoire racine est désigné par le symbole `/`. Si on veut désigner sans ambiguité un fichier on doit indiquer le **chemin complet** permettant de l'atteindre à partir du répertoire racine. Si on considère l'arborescence de la figure 4.3, le chemin complet d'un fichier `toto` qui se trouve dans le répertoire de `jim` sera `/usr/jim/toto`.

4.2.2 Opérations

Le système de fichiers doit permettre de faire les opérations suivantes sur un répertoire :

1. créer le répertoire,
2. supprimer le répertoire,

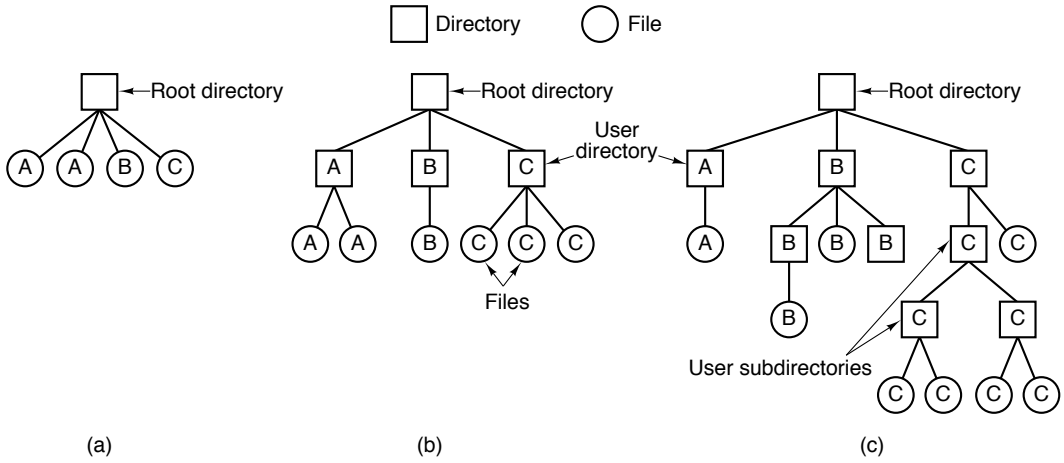

FIG. 4.2 – Exemples d'arborescences

3. ouvrir le répertoire pour y lire les noms des fichiers et répertoires qu'il contient,
4. fermer le répertoire,
5. lire un nom de fichier ou répertoire,
6. renomer le répertoire,
7. lier (ajouter) un nom de fichier ou de répertoire à l'interieur du répertoire,
8. délier (supprimer) un nom de fichier ou de répertoire du répertoire.

4.2.3 Opérations sur les répertoires sous Unix

Création de répertoires

La fonction :

```
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

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

crée un nouveau répertoire nommé `pathname`. `mode` spécifie les permissions à appliquer au répertoire. Cette valeur est modifiée par le `umask` du processus : la véritable valeur utilisée est `(mode & ~umask)`.

Suppression de répertoires

La fonction :

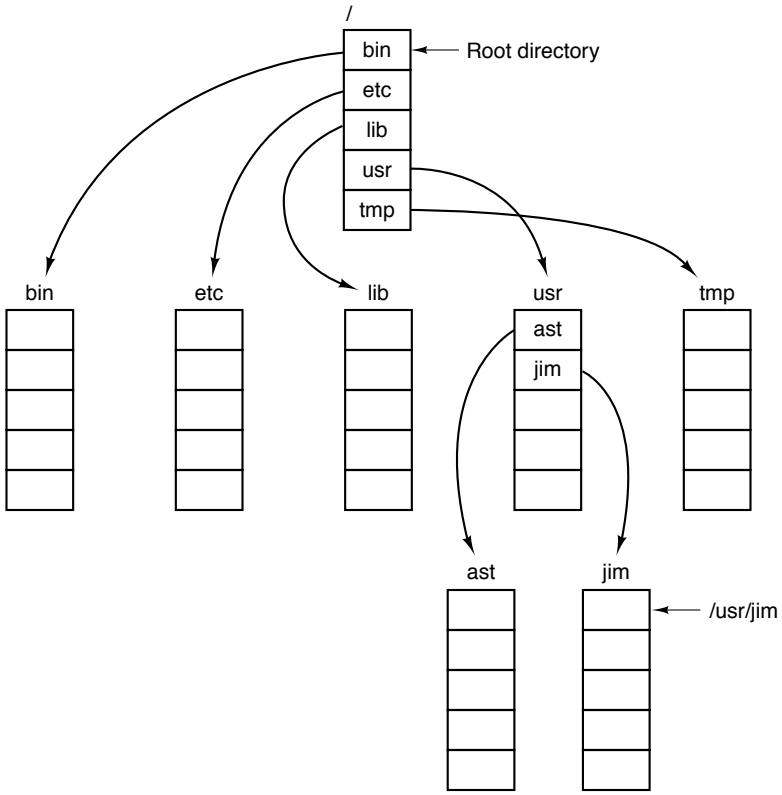

FIG. 4.3 – Arborescence Unix

```
#include <unistd.h>

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

supprime le répertoire `pathname` qui doit être vide.

Répertoire courant

La fonction :

```
#include <unistd.h>

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

permet de changer de répertoire courant.

Répertoire racine local

La fonction :

```
#include <unistd.h>

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

permet de modifier le répertoire racine. Elle remplace le répertoire racine du processus en cours par celui spécifié par le chemin `path`. Ce répertoire sera utilisé comme origine des chemins commençant par `/`. Le répertoire racine est hérité par tous les enfants du processus ayant fait le changement. Seul le Super-utilisateur peut effectuer un changement de répertoire racine.

Exploration des répertoires

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
```

La fonction `opendir` ouvre le répertoire `name`, et renvoie un pointeur sur une structure `DIR`. La fonction `readdir` permet de lire séquentiellement les entrées du répertoire correpondant à la structure `DIR`. `readdir` renvoie un pointeur sur une structure `dirent`. D'après POSIX, la structure `dirent` contient un champ `char d_name[]` de taille non spécifiée, avec au plus `NAME_MAX` caractères avant le caractère nul final. L'utilisation des autres champs de cette structure compromet la portabilité du programme.

Changement de nom

La fonction :

```
#include <stdio.h>

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

renomme un fichier ou un répertoire, en le déplacant vers un autre répertoire si `oldpath` est différent de `newpath`.

Création de liens

Sous Unix un fichier ou un répertoire peut avoir plusieurs noms et se trouver dans différents répertoires, la fonction

```
#include <unistd.h>

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

crée un nouveau lien (aussi appelé lien physique ou lien matériel) sur un fichier existant. Ce nouveau nom pourra être utilisé exactement comme l'ancien quelque soit l'opération. Les deux noms se réfèrent au même fichier (et ont donc les mêmes permissions et propriétaire) et il est impossible de déterminer quel nom était l'original. Le système maintient le nombre de liens physiques qui se réfèrent à un même fichier ou répertoire (cf. attribut `st_nlink` dans la structure `stat` présentée en section 4.1.3).

Suppression de fichiers

La fonction :

```
#include <unistd.h>

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

détruit un nom dans le système de fichiers et décrémente le nombre de liens qui s'y réfèrent. Si ce nom était le dernier lien sur un fichier, ce dernier est effacé et l'espace qu'il utilisait est rendu disponible.

Liens symboliques

Un lien symbolique consiste en un nom d'un chemin de fichier ou répertoire. Les liens symboliques différent donc des liens physiques et leurs créations n'augmentent pas le nombre de liens qui se réfèrent aux fichiers ou répertoires qu'ils désignent. Un lien symbolique peut pointer vers un fichier existant ou sur un fichier non-existant. Les permissions d'accès à un lien symbolique sont sans importance, le propriétaire est ignoré lorsque l'on suit le lien, il n'est vérifié que pour supprimer ou renommer le lien si celui ci se trouve dans un répertoire avec le Sticky-Bit positionné.

```
#include <unistd.h>

int symlink(const char *oldpath, const char *newpath);
int readlink(const char *path, char *buf, size_t bufsiz);
```

La fonction `symlink` permet de créer un lien symbolique sur un fichier ou répetoire. La fonction `readlink` lit le contenu d'un lien symbolique, pour cela elle place le contenu du lien symbolique `path` dans le tampon `buf`, lequel doit contenir au moins `bufsiz` octets. Attention : la fonction `readlink` n'ajoute pas de caractère `'\0'` dans `buf`. Il tronquera le contenu (a la longueur `bufsiz`) si le buffer est trop petit pour recevoir les données.

4.3 Mise en œuvre du système de fichiers

4.3.1 Mise en œuvre des fichiers

Un fichier est stocké en mémoire secondaire (disque) sous la forme d'une *suite de blocs*. Le bloc représente l'unité minimale d'information manipulée par le système de fichier en mémoire

secondaire. Sa taille varie selon les machines et les systèmes, elle doit être suffisament grande afin de minimiser le nombre d'accès en mémoire secondaire (qui est beaucoup plus lente que la mémoire principale), et suffisament petite pour éviter le ralentissement par le chargement de données inutiles. Sur les systèmes actuelles la taille d'un bloc est de l'ordre d'un kilo-octets.

Il existe différentes méthodes permettant de représenter un fichier sous forme d'une suite de blocs, nous allons en présenter quatre.

Représentation par blocs contigus

La méthode la plus simple consiste à allouer une suite de blocs consécutifs pour chaque fichier. Cette méthode présente les avantages et inconvenients suivants :

Avantages

- simple à mettre en œuvre.
- bonnes performances d'accès puisque la tête de lecture du disque ne fait qu'un seul déplacement, les blocs sont ensuite lus séquentiellement en une seule opération sur le disque.

Inconvénients

- on doit connaître la taille du fichier à sa création afin de lui réserver suffisament de blocs.
- fragmentation du disque, donc gaspillage d'espace disque. On peut y remedier en compactant les fichiers, mais c'est très couteux en temps.

Représentation par liste chaînée de blocs

On peut représenter un fichier par des blocs non contigüs en chainant les blocs. Chaque bloc contient, en plus des données du fichier, un pointeur sur le numéro du bloc suivant sur le disque.

Avantage

- tous les blocs du disque peuvent être utilisés

Inconvénient

- l'accés aléatoire (directe) est extrêmement long, puisqu'il faut parcourir séquentiellement le chainage pour trouver le bloc correspondant aux données auxquelles on veut directement accéder.

Représentation par liste chaînée indexée

Plutôt que de représenter le chaînage des blocs en mémoire secondaire, on peut le représenter en mémoire centrale à l'aide d'une table (cf. figure 4.4). MS-DOS utilise cette méthode.

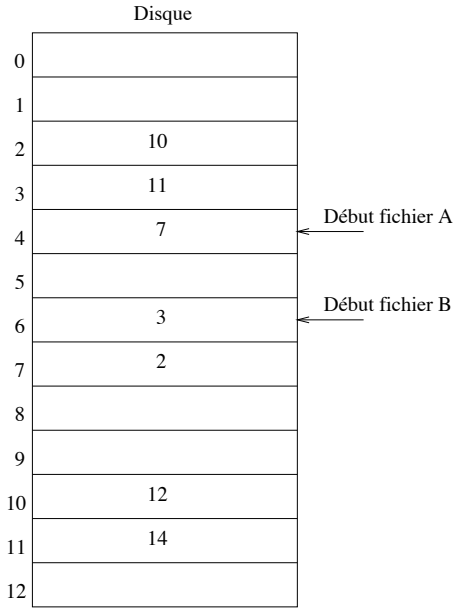

FIG. 4.4 –

Avantage
– tous les blocs du disque peuvent être utilisés
– l'accès aléatoire est rapide puisque le parcours séquentiel du chainage se fait en mémoire principale

Inconvénient
– la table doit être entièrement présente en mémoire centrale

Représentation par des nœuds d'information

Sous Unix chaque fichier est associé à une petite table appelée *nœud d'information* (*i-node*). Cette table contient les attributs et les adresses des blocs du fichier. Si la taille de la table ne suffit pas à stocker les adresses de tous les blocs du fichier, une indirection (simple) renvoie sur une autre table dite bloc d'indirection simple qui contient les blocs suivants. Si le bloc d'indirection simple ne suffit pas, une autre indirection (double) renvoie sur une table d'indirections sur des blocs d'indirection simple. Dans le cas des fichiers de très grande taille une indirection triple est prévue dans certaines implémentations de systèmes de fichiers Unix (cf. figure 4.5).

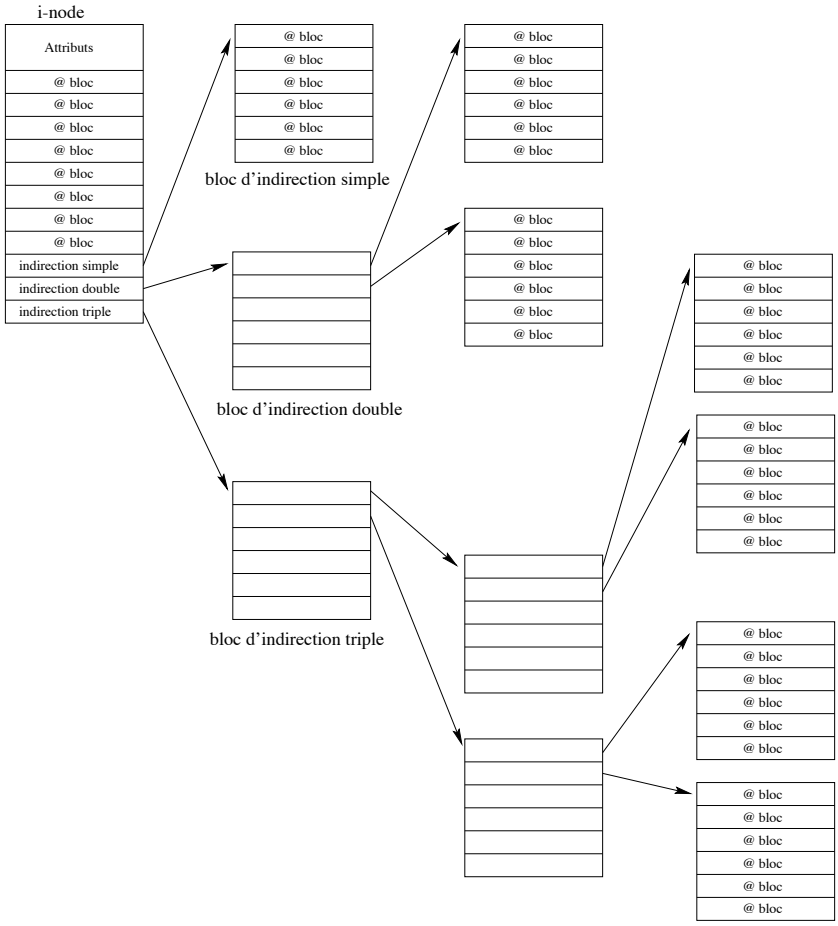

FIG. 4.5 – Structure d'un nœud d'information

Avantages
- tous les blocs du disque peuvent être utilisés
- L'accès aux fichiers courts est très rapide puisque l'*i-node* est chargé en mémoire centrale
- les adresses des blocs n'occupent pas beaucoup de place en mémoire centrale, seuls les blocs d'indirection nécessaires à un moment donné sont chargés en mémoire.

Inconvénient
- mise en œuvre plus difficile que les méthodes précédentes

4.3.2 Mise en œuvre des répertoires

Les entrées d'un répertoire doivent contenir au moins deux informations (cf. figure 4.6) :
- le nom du fichier ou répertoire (i-nœud) que l'entrée désigne ;
- un numéro qui permet de retrouver les blocs du fichier (le numéro d'i-nœud dans le cas d'Unix).

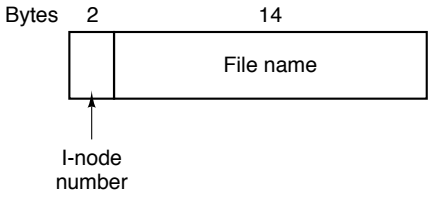

FIG. 4.6 – Entrée dans un répertoire Unix

La recherche d'un fichier à partir de son chemin complet s'effectue de manière similaire dans tous les systèmes de fichiers hiérarchisés. Dans le cas d'Unix si le système recherche par exemple un fichier `/usr/ast/mbox`, il procède de la façon suivante : il commence dabors par chercher l'i-nœud du répertoire racine (qui est situé à un endroit fixe du disque). Puis il cherche `usr` dans les entrées du répertoire racine, pour trouver l'i-nœud du répertoire `/usr`. Il fait la même chose dans `/usr` pour trouver l'i-nœud du répertoire `/usr/ast`, et de même pour trouver l'i-nœud du fichier `/usr/ast/toto` (cf. figure 4.6).

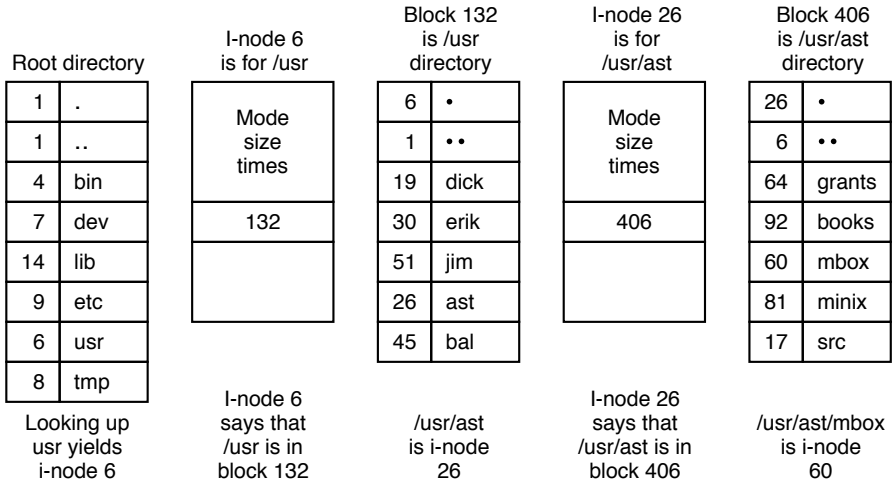

FIG. 4.7 – Etapes pour la recherche de /usr/ast/mbox

4.3.3 Gestion de l'espace disque

Tailles des blocs

Le choix de la taille d'un bloc a des répercussions sur :

- la vitesse de transfert : plus la taille d'un bloc est importante, plus le transfert est rapide ; car moins il y a de blocs dans un fichier, moins il y a de temps perdu à charger les blocs (délais de recherche de blocs et rotation du disque),
- le taux de remplisage du disque : plus la taille d'un bloc est importante, plus il y a de gaspillage d'espace disque à cause des petits fichiers dont la taille est inférieur à la taille d'un

bloc. Ainsi si par exemple la taille moyenne d'un fichier est de 1 Ko, en allouant 32Ko à un bloc, on pert 31/32 soit 97% de l'espace du disque !

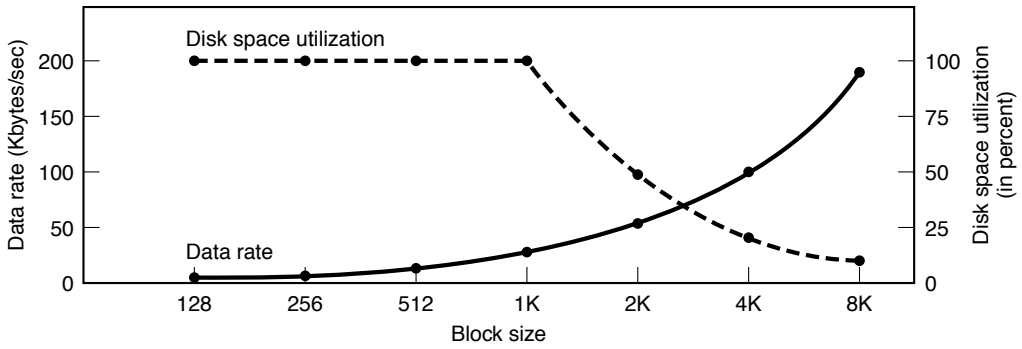

FIG. 4.8 – La courbe en trait plein donne la vitesse de transfert des données. La courbe en pointillé donne le taux de remplissage du disque.

On voit d'après la figure 4.8 qu'il est délicat de trouver le juste compromis pour trouver la taille qui convient le mieux des deux points de vue : vitesse et remplissage.

Mémorisation des blocs libres

Afin de trouver les blocs libres, le système peut :
- soit chaîner les blocs libres (cf. figure 4.9 (a)),
- soit utiliser une table de bits : à chaque bloc on fait correspondre dans une table de bits 1 si le bloc est occupée, 0 si le bloc est libre ou vice versa (cf. figure 4.9 (b)).

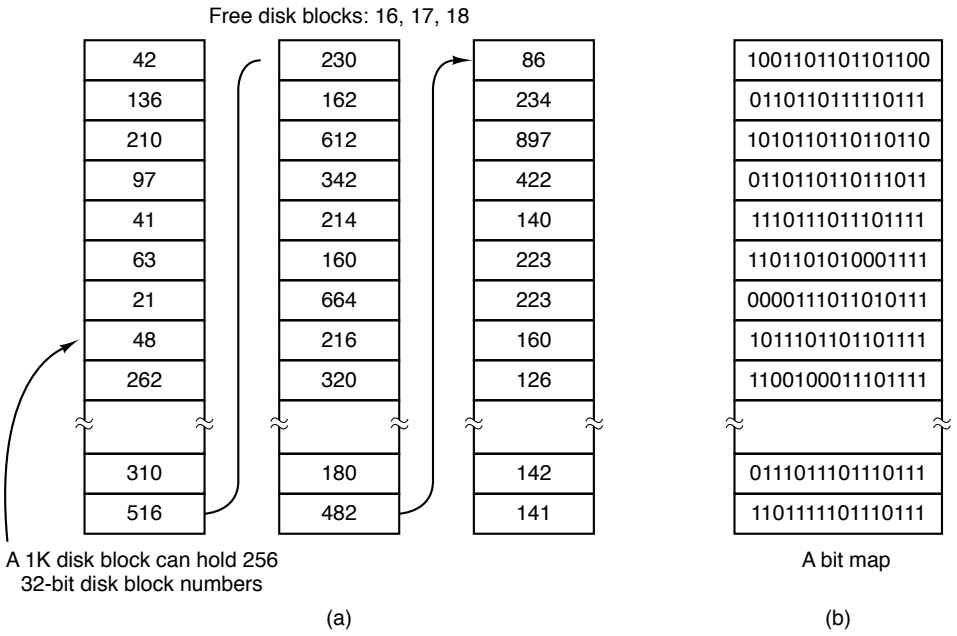

FIG. 4.9 –

4.3.4 Fiabilité

Le systeme de fichiers doit pouvoir réagir correctement en cas de problèmes qui peuvent survenir en mémoire secondaire : panne de courant (donc perte du cache), disques endommagés etc. . .

Blocs endommagés

Il est courant qu'un disque puisse avoir certain de ses blocs définitivement endommagés. Ces blocs ne doivent pas être présents dans la liste des blocs libres. On maintient pour cela une liste des blocs endommagés. Certains disques intègrent la gestion de cette liste : si le système demande un bloc endommagé, le controleur du disque lui envoie un bloc remplaçant, situé à une autre adresse que celle demandée sans que le système puisse s'en rendre compte.

Cohérence

Certaines incohérences peuvent apparaitre dans les structures de données du système de fichiers, suite à des pannes de courant par exemple. Afin de vérifier la cohérence du système de fichiers, le système peut lancer periodiquement, ou à la demande, des procédures de vérification.

On peut vérifier la cohérence à deux niveaux : bloc ou fichier.

Sous Unix la procédure de vérification au niveau bloc consiste à construire deux tables qui contiennent chacune un compteur pour chaque bloc. Dans la première table on compte le nombre de fois où chaque bloc est référencé dans un fichier, dans la seconde le nombre de fois où chaque bloc apparaît dans la liste chaînée des blocs libres (cf. figure 4.10).

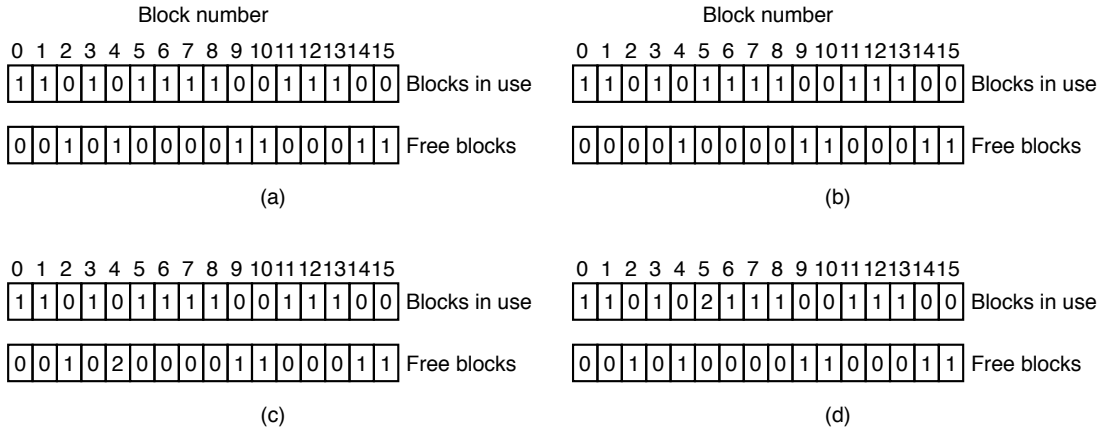

FIG. 4.10 – Différents états d'un système de fichiers. (a) Cohérent. (b) Blocs manquants. (c) Blocs dupliqués dans la liste des blocs libres. (d) Blocs de données référencé deux fois.

Pour construire les deux tables la procédure :

- parcourt tous les i-nodes, pour chaque i-node elle incrémente le nombre d'utilisation de chaque bloc de l'i-node ;
- parcourt la liste des blocs libres, pour chaque bloc trouvé, incrémente son compteur de blocs libres.

Les deux tables ainsi construites permettent de détecter et de réparer les incohérences de quatre types :

- un bloc est partagé par plusieurs fichiers (nombre d'utilisation est supérieur à 1) : dans ce cas le système duplique le bloc sur des blocs libres, et affecte chaque bloc à un des fichiers ;
- un bloc est à la fois libre et utilisé pas un fichier : il est supprimé des blocs libres ;
- un bloc est présent plusieurs fois dans la liste des blocs libres, la procédure supprime les occurrences de ce bloc dans la liste sauf une.
- un bloc est ni libre, ni utilisé : il est ajouté dans la liste des blocs libres.

4.3.5 Performances

Afin d'améliorer les performances d'un système de fichier et de pallier à la lenteur de la mémoire secondaire, on utilise un *buffer cache* en mémoire principale. Le cache contient un ensemble de blocs chargé en mémoire, à chaque accès à un bloc, le système vérifie si le bloc est dans le cache, si tel est le cas la demande est satisfaite immédiatement, dans le cas contraire, le bloc est chargé dans le cache et est fourni au programme le demandant.

La gestion du cache peut se faire en utilisant une liste FIFO. Dans le cas ou le cache est plein, le système évince le bloc le plus anciennement chargé. Cette solution n'est pas adaptée car certains blocs sont plus importants que d'autres. Deux solutions qui peuvent être combinées sont envisageables :

- répartir les blocs en plusieurs catégories : blocs des i-nodes, blocs d'indirections, blocs de répertoires, blocs de données etc. . .et mettre les blocs par ordre de priorité dans le cache ; les blocs les moins importants sont évincés en premier.
- si un bloc est important pour la cohérence du système il est écrit dans le disque dès qu'il est modifié.

4.3.6 Implantation du système de fichiers sous Linux

Le système de fichiers virtuel

Linux supporte différents types de systèmes de fichiers. Pour permettre aux processus et aux autres modules du noyau d'accéder au système de fichiers de façon uniforme, Linux possède un système fichiers virtuel dont le but est d'assurer l'interface entre les appels système et les fichiers auquels ils accèdent (cf. figure 4.11). Le buffer cache est commun à tous les systèmes de fichiers simultanément utilisés.

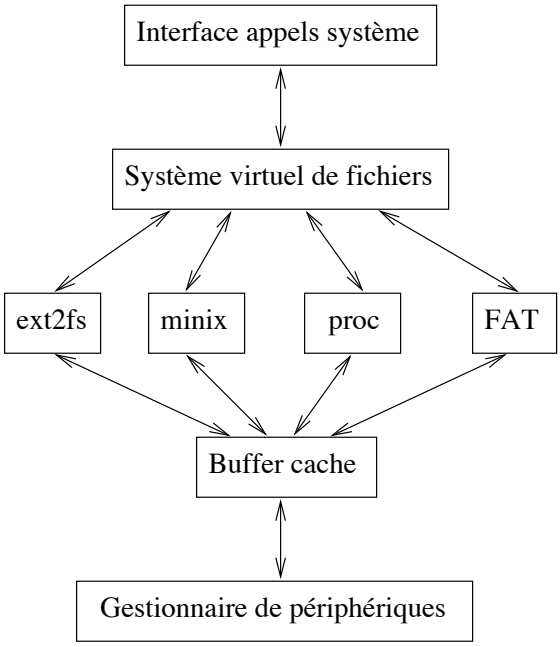

FIG. 4.11 – Le système de fichier virtuel de Linux

Le buffer cache est commun à tous les systèmes de fichiers simultanément utilisés. Le cache gère les tampons mémoire associés aux blocs sur le disque. La structure `buffer_head` déclarée dans le fichier `<linux/fs.h>` décrit le format d'un tampon mémoire associé à un bloc du disque :

```
struct buffer_head {
/* First cache line: */
unsigned long b_blocknr; /* block number */
kdev_t b_dev; /* device (B_FREE = free) */
kdev_t b_rdev; /* Real device */
unsigned long b_rsector; /* Real buffer location on disk */
struct buffer_head * b_next; /* Hash queue list */
struct buffer_head * b_this_page; /* circular list of buffers in one page */

/* Second cache line: */
unsigned long b_state; /* buffer state bitmap (see above) */
struct buffer_head * b_next_free;
unsigned int b_count; /* users using this block */
unsigned long b_size; /* block size */

/* Non-performance-critical data follows. */
char * b_data; /* pointer to data block (1024 bytes) */
unsigned int b_list; /* List that this buffer appears */
unsigned long b_flushtime;      /* Time when this (dirty) buffer
 * should be written */
unsigned long b_lru_time;       /* Time when this buffer was
 * last used. */
```

```
struct wait_queue * b_wait;
struct buffer_head * b_prev; /* doubly linked list of hash-queue */
struct buffer_head * b_prev_free; /* doubly linked list of buffers */
struct buffer_head * b_reqnext; /* request queue */

/*
 * Some MD stuff like RAID5 needs special event handlers and
 * special private buffer_head fields:
 */
void * personality;
void * private_bh;
};
```

La structure d'i-node est aussi définie dans `<linux/fs.h>` :

```
struct inode {
kdev_t i_dev;
unsigned long i_ino;
umode_t i_mode;
nlink_t i_nlink;
uid_t i_uid;
gid_t i_gid;
kdev_t i_rdev;
off_t i_size;
time_t i_atime;
time_t i_mtime;
time_t i_ctime;
unsigned long i_blksize;
unsigned long i_blocks;
unsigned long i_version;
unsigned long i_nrpages;
struct semaphore i_sem;
struct inode_operations *i_op;
struct super_block *i_sb;
struct wait_queue *i_wait;
struct file_lock *i_flock;
struct vm_area_struct *i_mmap;
struct page *i_pages;
struct dquot *i_dquot[MAXQUOTAS];
struct inode *i_next, *i_prev;
struct inode *i_hash_next, *i_hash_prev;
struct inode *i_bound_to, *i_bound_by;
struct inode *i_mount;
unsigned long i_count; /* needs to be > (address_space * tasks)>>pagebits */
unsigned short i_flags;
unsigned short i_writecount;
unsigned char i_lock;
unsigned char i_dirt;
unsigned char i_pipe;
unsigned char i_sock;
unsigned char i_seek;
```

```
unsigned char i_update;
  unsigned char i_condemned;
union {
struct pipe_inode_info pipe_i;
struct minix_inode_info minix_i;
struct ext_inode_info ext_i;
struct ext2_inode_info ext2_i;
struct hpfs_inode_info hpfs_i;
struct msdos_inode_info msdos_i;
struct umsdos_inode_info umsdos_i;
struct iso_inode_info isofs_i;
struct nfs_inode_info nfs_i;
struct xiafs_inode_info xiafs_i;
struct sysv_inode_info sysv_i;
struct affs_inode_info affs_i;
struct ufs_inode_info ufs_i;
struct socket socket_i;
void * generic_ip;
} u;
};
```

Le système de fichiers Ext2

Le système de fichiers Ext2 décompose le périphérique physique en plusieurs ensembles de blocs. Chaque ensemble de blocs contient les éléments suivants :

- Une copie du superbloc : contient les informations de contôle du système de fichiers. Chaque ensemble de blocs contient une copie du superbloc pour palier aux problèmes de corruption du système.
- Une table de descripteurs : contient les adresses de blocs contenant les informations cruciales, comme les blocs des tables de bits, de la table des i-nodes etc.... Ces informations sont aussi dupliquées dans chaque ensemble de blocs.
- Un bloc contenant la table des bits d'occupation des blocs de l'ensemble (1 alloué/O non alloué).
- Un bloc contenant la table des bits d'occupation des i-nodes de l'ensemble (1 alloué/O non alloué).
- Une table des i-nodes.
- Les blocs de données.

Un i-node Linux permet d'avoir une indirection simple ou double pour les grands fichiers. Lorsqu'un i-node à besoin d'un bloc, le sytème cherche en priorité un bloc libre dans le même ensemble de blocs, ce qui assure que les blocs d'un même fichier sont généralement situés à proximité et évite de longs déplacements de la tête de lecture du disque. La structure d'un i-node Ext2 est définie dans `<linux/ext2_fs.h>` :

```
/*
 * Structure of an inode on the disk
 */
struct ext2_inode {
```

```
        __u16   i_mode;         /* File mode */
        __u16   i_uid;          /* Owner Uid */
        __u32   i_size;         /* Size in bytes */
        __u32   i_atime;        /* Access time */
        __u32   i_ctime;        /* Creation time */
        __u32   i_mtime;        /* Modification time */
        __u32   i_dtime;        /* Deletion Time */
        __u16   i_gid;          /* Group Id */
        __u16   i_links_count;  /* Links count */
        __u32   i_blocks;       /* Blocks count */
        __u32   i_flags;        /* File flags */
        union {
                struct {
                        __u32  l_i_reserved1;
                } linux1;
                struct {
                        __u32  h_i_translator;
                } hurd1;
                struct {
                        __u32  m_i_reserved1;
                } masix1;
        } osd1;                         /* OS dependent 1 */
        __u32   i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
        __u32   i_version;      /* File version (for NFS) */
        __u32   i_file_acl;     /* File ACL */
        __u32   i_dir_acl;      /* Directory ACL */
        __u32   i_faddr;        /* Fragment address */
        union {
                struct {
                        __u8    l_i_frag;       /* Fragment number */
                        __u8    l_i_fsize;      /* Fragment size */
                        __u16   i_pad1;
                        __u32   l_i_reserved2[2];
                } linux2;
                struct {
                        __u8    h_i_frag;       /* Fragment number */
                        __u8    h_i_fsize;      /* Fragment size */
                        __u16   h_i_mode_high;
                        __u16   h_i_uid_high;
                        __u16   h_i_gid_high;
                        __u32   h_i_author;
                } hurd2;
                struct {
                        __u8    m_i_frag;       /* Fragment number */
                        __u8    m_i_fsize;      /* Fragment size */
                        __u16   m_pad1;
                        __u32   m_i_reserved2[2];
                } masix2;
        } osd2;                         /* OS dependent 2 */
};
```