Système de Fichiers

Aloÿs Dufour

ATER, LIPN équipe LoCal
Université Paris-Nord XIII

20 janvier 2026

Partie 1 : Généralités

Concept de Fichier

Un fichier est une unité de stockage logique de l'information

- Abstraction des propriétés physiques des dispositifs de stockage
- La correspondance est établie par l'OS

Concept de Fichier

Un fichier est une unité de stockage logique de l'information

- Abstraction des propriétés physiques des dispositifs de stockage
- La correspondance est établie par l'OS

Attributs d'un fichier

- Nom, taille, type, protection, date, propriétaire, ...

Concept de Fichier

Un fichier est une unité de stockage logique de l'information

- Abstraction des propriétés physiques des dispositifs de stockage
- La correspondance est établie par l'OS

Attributs d'un fichier

- Nom, taille, type, protection, date, propriétaire, ...

Opérations sur les fichiers

- Création, Écriture/Lecture, Exécution, Suppression, Concatenation ...

Type et Structure des Fichiers

Manières de stocker les fichiers

- suite d'enregistrements (CP/M, VSAM)
- suite d'octets (Unix, MS-DOS)

Type et Structure des Fichiers

Manières de stocker les fichiers

- suite d'enregistrements (CP/M, VSAM)
- suite d'octets (Unix, MS-DOS)

"Nature" des fichiers

- régulier, dossier, tube, lien, bloc device, char device, socket

Type et Structure des Fichiers

Manières de stocker les fichiers

- suite d'enregistrements (CP/M, VSAM)
- suite d'octets (Unix, MS-DOS)

"Nature" des fichiers

- régulier, dossier, tube, lien, bloc device, char device, socket

"Type" de fichiers

- *Magic number* & structure interne des fichiers
- Par les suffixes des noms de fichiers : .txt, pdf, png, jpg, mp4, c, h, cpp, aux, log, tex, mkv, ogg, toc, out, ...
- `OPENER` : type $\mapsto$ application, classification MIME

Méthodes d'Accès aux Fichiers

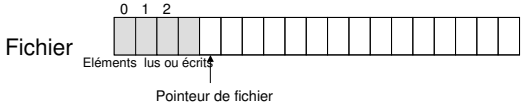
Dépendant du matériel

- Accès séquentiel
- Accès direct (ou aléatoire)
- Accès indexé

Méthodes d'Accès aux Fichiers

Accès séquentiel

- Les éléments sont lus ou écrits dans l'ordre

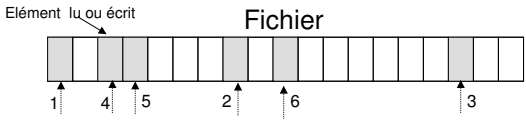


- Méthode adaptée aux supports de stockage séquentiels : Bandes magnétiques

Méthodes d'Accès aux Fichiers

Accès direct

- L'ordre d'accès aux éléments est quelconque



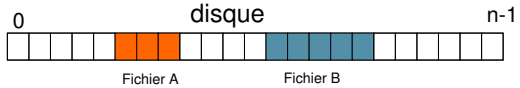
- Méthode adaptée aux supports de stockage direct : disques

- Accès à partir d'une clé

Méthodes d'Allocation des Fichiers

Allocation contiguë

Chaque fichier occupe un nombre de blocs contigus sur le disque

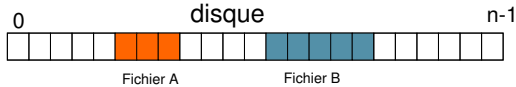


Accès à un bloc : nv = num bloc logique + num premier bloc

Méthodes d'Allocation des Fichiers

Allocation contiguë

Chaque fichier occupe un nombre de blocs contigus sur le disque



Accès à un bloc : nv = num bloc logique + num premier bloc

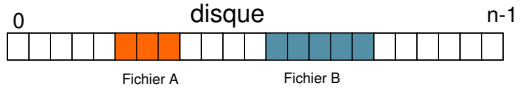
Avantages

- Simple à implémenter
- Accès direct aux blocs en temps constant
- Adapté aux supports "Write Once" (CDRom, CDRW)

Méthodes d'Allocation des Fichiers

Allocation contiguë

Chaque fichier occupe un nombre de blocs contigus sur le disque



Accès à un bloc : nv = num bloc logique + num premier bloc

Avantages

- Simple à implémenter
- Accès direct aux blocs en temps constant
- Adapté aux supports "Write Once" (CDRom, CDRW)

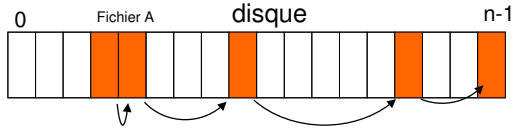
Inconvénients

- Problème : Extension d'un fichier
  - Nécessite de connaître à l'avance de la taille du fichier
  - Déplacement possible du fichier lors de son extension
- Problème : Fragmentation
  - L'espace libre peut être fragmenté en plusieurs trous et aucun n'est pas suffisant pour stocker un fichier
  - $\rightarrow$ Défragmantation

Méthodes d'Allocation des Fichiers

Allocation chaînée

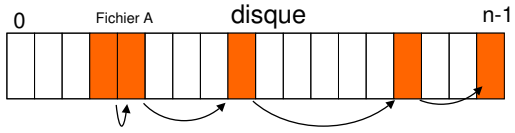
- Un fichier occupe une liste chaînée de blocs sur le disque
- Chaque bloc contient une partie des données et un pointeur sur le bloc suivant



Méthodes d'Allocation des Fichiers

Allocation chaînée

- Un fichier occupe une liste chaînée de blocs sur le disque
- Chaque bloc contient une partie des données et un pointeur sur le bloc suivant



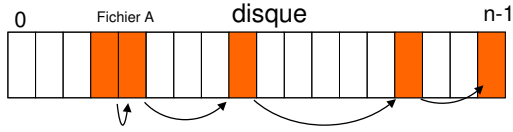
Avantages

- Possibilité d'étendre un fichier
- Allocation par bloc individuel : Tout bloc libre peut être utilisé pour satisfaire une requête d'allocation

Méthodes d'Allocation des Fichiers

Allocation chaînée

- Un fichier occupe une liste chaînée de blocs sur le disque
- Chaque bloc contient une partie des données et un pointeur sur le bloc suivant



Avantages

- Possibilité d'étendre un fichier
- Allocation par bloc individuel : Tout bloc libre peut être utilisé pour satisfaire une requête d'allocation

Inconvénients

- Solution non adaptée à l'accès direct
  - L'accès à un bloc quelconque nécessite l'accès à tous les blocs qui le précèdent
- Les pointeurs sont stockés sur disque

Méthodes d'Allocation des Fichiers

Allocation chaînée et indexée

Séparer les pointeurs et les données

Méthodes d'Allocation des Fichiers

Allocation chaînée et indexée

Séparer les pointeurs et les données
Technique :

- Utilisation d'une table d'allocation de fichier (FAT : File Allocation Table)
- À chaque bloc est associée une entrée dans la FAT qui contient le numéro du bloc suivant (Méthode utilisée dans MS-DOS et OS/2)

Méthodes d'Allocation des Fichiers

Allocation chaînée et indexée

Séparer les pointeurs et les données
Technique :

- Utilisation d'une table d'allocation de fichier (FAT : File Allocation Table)
- À chaque bloc est associée une entrée dans la FAT qui contient le numéro du bloc suivant (Méthode utilisée dans MS-DOS et OS/2)

Avantages

- Extension des fichiers
- Les blocs de données ne contiennent pas les pointeurs
- Accès direct facile

Méthodes d'Allocation des Fichiers

Allocation chaînée et indexée

Séparer les pointeurs et les données
Technique :

- Utilisation d'une table d'allocation de fichier (FAT : File Allocation Table)
- À chaque bloc est associée une entrée dans la FAT qui contient le numéro du bloc suivant (Méthode utilisée dans MS-DOS et OS/2)

Avantages

- Extension des fichiers
- Les blocs de données ne contiennent pas les pointeurs
- Accès direct facile

Inconvénients

- Occupation de la mémoire centrale par la FAT
- Problème des disques de grande capacité

Méthodes d'Allocation des Fichiers

Allocation par nœud d'information

- Éclater la FAT en plusieurs petites tables appelées *nœuds d'informations (i-node)*
- À chaque fichier est associé un nœud d'information
- Chaque table contient les attributs et les adresses sur le disque des blocs du fichier

Méthodes d'Allocation des Fichiers

Allocation par nœud d'information

- Éclater la FAT en plusieurs petites tables appelées *nœuds d'informations (i-node)*
- À chaque fichier est associé un nœud d'information
- Chaque table contient les attributs et les adresses sur le disque des blocs du fichier

Unix

- La table est hiérarchisé sur Unix
- FS System V
  - 10 direct, 1 simple indirection, 1double indirection, 1 triple indirection
- BSD Fast FS / UFS
  - 12 direct, 1 simple indirection, 2 double indirection

Méthodes d'Allocation des Fichiers

Allocation par nœud d'information

- Éclater la FAT en plusieurs petites tables appelées *nœuds d'informations (i-node)*
- À chaque fichier est associé un nœud d'information
- Chaque table contient les attributs et les adresses sur le disque des blocs du fichier

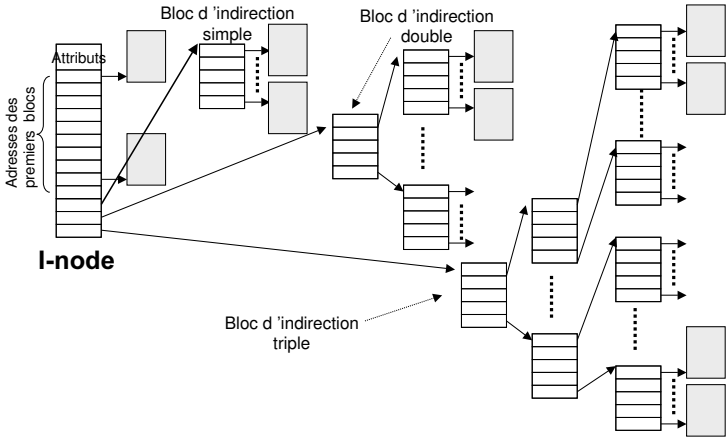
Unix

- La table est hiérarchisé sur Unix
- FS System V
  - 10 direct, 1 simple indirection, 1double indirection, 1 triple indirection
- BSD Fast FS / UFS
  - 12 direct, 1 simple indirection, 2 double indirection

Que des avantages

- Seuls les nœuds d'information des fichiers ouverts sont chargés en mémoire centrale
- Allocation par bloc individuel
- Accès direct facile (nécessite au maximum 4 accès disque)
- Adaptée aux disques de très grande capacité

Structure d'un nœud d'information



Les systèmes de fichiers existant

- `ext2` : historique, GNU+Linux (extent)
- `ext4` : son successeur encore très utilisé
- `btrfs` : en fin de développement (arbres-b)
- `zfs` : SunMicrosystems, puis BSD
- `nfs` : en réseau
- `fat32` : historique, encore utilisé, clefs USB
- `ntfs` : windows
- `apfs` : apple

Support intégrés dans les (bons) noyaux monolithiques, module pour les micronoyaux ou noyaux hybrides.

Propriétés additionnelles

- journalisation
- snapshots
- support réseau
- compression
- RAID
- sous-volumes
- redimensionnement à chaud
- copy-on-write (CoW)
- chiffrement intégré…

En pratique

- Les volumes de stockages sont représentés par des fichiers de type “block”, dans `/dev`

En pratique

- Les volumes de stockages sont représentés par des fichiers de type “block”, dans `/dev`
- Au début d’un *block device* se trouve une table de partition (outils : `gparted`, `fdisk`, ...)
  Exemples : `/dev/sda1`, `/dev/sdb2`, `/dev/sdb3`, `/dev/mmcblk0p1`, `/dev/sr0`

En pratique

- Les volumes de stockages sont représentés par des fichiers de type “block”, dans `/dev`
- Au début d’un *block device* se trouve une table de partition (outils : `gparted`, `fdisk`, ...)
  Exemples : `/dev/sda1`, `/dev/sdb2`, `/dev/sdb3`, `/dev/mmcblk0p1`, `/dev/sr0`
- Chaque partition est formatée avec un système de fichier :
  `mkfs.ext4 /dev/sdb1`

En pratique

- Les volumes de stockages sont représentés par des fichiers de type “block”, dans `/dev`
- Au début d’un *block device* se trouve une table de partition (outils : `gparted`, `fdisk`, ...)
  Exemples : `/dev/sda1`, `/dev/sdb2`, `/dev/sdb3`, `/dev/mmcblk0p1`, `/dev/sr0`
- Chaque partition est formatée avec un système de fichier :
  `mkfs.ext4 /dev/sdb1`
- Un système de fichiers est *monté* sur un répertoire :
  `mount /dev/sdb1 /media/flims`
  `lsblk, df -h, findmnt`

Exercices

- Prenons une variante d'i-nœud d'une taille de 16 octets, représenté comme suit :
  - Trois champs de 8 bits chacun pour l'information sur le fichier,
  - Onze pointeurs de blocs primaires,
  - Un pointeur pour une indirection simple sur un bloc dont le dernier pointeur fait une seconde indirection simple.

  Sachant que les blocs sur ce système sont de 1 Ko et que les adesses (pointeurs) sont codées sur 16 bits,

Exercices

- Prenons une variante d'i-nœud d'une taille de 16 octets, représenté comme suit :
  - Trois champs de 8 bits chacun pour l'information sur le fichier,
  - Onze pointeurs de blocs primaires,
  - Un pointeur pour une indirection simple sur un bloc dont le dernier pointeur fait une seconde indirection simple.

  Sachant que les blocs sur ce système sont de 1 Ko et que les adesses (pointeurs) sont codées sur 16 bits,

  1. Quelle sera la taille maximale d'un fichier (en octets) que ce système de fichier peut supporter ?
  2. Dans quel bloc du disque dur retrouve-t-on le 209921ème octet du fichier représenté par le nœud ?

Solution

- Un bloc peut contenir au maximum 512 adresses (pointeurs). La taille est de $(11 + 511 + 512) = 1034$ blocs. Ceci correspond à 1034 *Ko*, donc environ 1 Mo.

Solution

- Un bloc peut contenir au maximum 512 adresses (pointeurs). La taille est de $(11 + 511 + 512) = 1034$ blocs. Ceci correspond à 1034 *Ko*, donc environ 1 Mo.
- Il s'agit du premier octet du bloc numéro 205. Le pointeur à retrouver est donc à la position 194 du premier bloc d'indirection simple.

Exercices

On considère un système de fichiers tel que l'information concernant les blocs de données de chaque fichier est accessible à partir du i-nœud de celui-ci (comme dans UNIX). On supposera que :

- Le système de fichiers utilise des blocs de données de taille fixe 1 Ko ;
- L'i-nœud de chaque fichier (ou répertoire) contient 12 pointeurs directs sur des blocs de données, 1 pointeur indirect simple, 1 pointeur indirect double et 1 pointeur indirect triple.
- Chaque pointeur (numéro de bloc) est représenté sur 4 octets.

Exercices

On considère un système de fichiers tel que l'information concernant les blocs de données de chaque fichier est accessible à partir du i-nœud de celui-ci (comme dans UNIX). On supposera que :

- Le système de fichiers utilise des blocs de données de taille fixe 1 Ko ;
- L'i-nœud de chaque fichier (ou répertoire) contient 12 pointeurs directs sur des blocs de données, 1 pointeur indirect simple, 1 pointeur indirect double et 1 pointeur indirect triple.
- Chaque pointeur (numéro de bloc) est représenté sur 4 octets.

1. Quelle est la plus grande taille de fichier que ce système de fichiers peut supporter ?
2. On considère un fichier contenant 10^5 octets. Combien de blocs de données sont-ils nécessaires (au total) pour représenter ce fichier sur disque ?

Solution

1. Chaque bloc de données peut contenir 256 pointeurs.

$$\text{Taille}_{\text{en Ko}} = 12 + 256 + 256^2 + 256^3 = 16'843'020 \simeq 16,06 Go.$$

Solution

1. Chaque bloc de données peut contenir 256 pointeurs.

$$\text{Taille}_{\text{en Ko}} = 12 + 256 + 256^2 + 256^3 = 16'843'020 \simeq 16,06 Go.$$

2. Soit un fichier de 10^5 octets,
   $10^5 = 97 \times 1024 + 672$. Il faudra donc 98 blocs pour conserver les données de ce fichier.

Solution

1. Chaque bloc de données peut contenir 256 pointeurs.

$$\text{Taille}_{\text{en Ko}} = 12 + 256 + 256^2 + 256^3 = 16'843'020 \simeq 16,06 Go.$$

2. Soit un fichier de 10^5 octets,
$10^5 = 97 \times 1024 + 672$. Il faudra donc 98 blocs pour conserver les données de ce fichier. L'i-nœud ne dispose que de 12 pointeurs directs, il va donc falloir utiliser des blocs supplémentaires $(98 - 12 = 86)$ pour conserver le reste des données. Comme $86 < 256$, ces blocs de données seront accessibles à partir du pointeur indirect simple du i-nœud.

Solution

1. Chaque bloc de données peut contenir 256 pointeurs.

$$\text{Taille}_{\text{en Ko}} = 12 + 256 + 256^2 + 256^3 = 16'843'020 \simeq 16,06 Go.$$

2. Soit un fichier de 10^5 octets,
   $10^5 = 97 \times 1024 + 672$. Il faudra donc 98 blocs pour conserver les données de ce fichier.
   L'i-nœud ne dispose que de 12 pointeurs directs, il va donc falloir utiliser des blocs supplémentaires $(98 - 12 = 86)$ pour conserver le reste des données. Comme $86 < 256$, ces blocs de données seront accessibles à partir du pointeur indirect simple du i-nœud.
   Une partie d'un bloc sera nécessaire pour conserver les 86 pointeurs sur des blocs de données. Le nombre total de blocs utilisés est donc $98 + 1 = 99$.

Exercices

On considère un système disposant d'un système de fichiers similaire à celui d'UNIX avec une taille de blocs de données de 4Ko et des pointeurs (numéros de blocs) définies sur 4 octets. On supposera que le i-nœud de chaque fichier compte 12 pointeurs directs, 1 pointeur indirect simple, 1 pointeur indirect double et 1 pointeur indirect triple. On désire créer un fichier contenant un total de 2×10^7 de caractères ASCII.

Exercices

On considère un système disposant d'un système de fichiers similaire à celui d'UNIX avec une taille de blocs de données de 4Ko et des pointeurs (numéros de blocs) définies sur 4 octets. On supposera que le i-nœud de chaque fichier compte 12 pointeurs directs, 1 pointeur indirect simple, 1 pointeur indirect double et 1 pointeur indirect triple. On désire créer un fichier contenant un total de 2×10^7 de caractères ASCII.
Quelle est la fragmentation interne totale sur le disque résultant de la création de ce fichier ?

Solution

Notre fichier compte $2 \times 10^7 = 4882 \times 4096 + 3328$ o, donc 4883 blocs nécessaires. Il faudra aussi leur ajouter des blocs qui vont être utilisés pour stocker des pointeurs vers ces blocs de données. On effectue donc le calcul suivant (sur les blocs) :

- Le fichier compte 4883 blocs de données.
- Les pointeurs directs de l'i-nœud permettent d'accéder à 12 de ces blocs. Il reste donc 4871 blocs de données pour lesquels l'accès se fera à travers l'un des liens indirects.
- Le pointeur de lien indirect simple pointe sur un bloc qui contient 1024 numéros de blocs (pointeurs vers des blocs de données). Nous avons donc ajouté 1 bloc de pointeurs, et il reste $4871 - 1024 = 3847$ blocs à traiter.
- Le pointeur de lien indirect double permet d'accéder à 1024×1024 blocs, ce qui est plus que suffisant. Il suffit d'utiliser 4 blocs de données pour stocker les 3847 pointeurs de blocs permettant d'accéder aux données restantes.

Solution

Notre fichier compte $2 \times 10^7 = 4882 \times 4096 + 3328$ o, donc 4883 blocs nécessaires. Il faudra aussi leur ajouter des blocs qui vont être utilisés pour stocker des pointeurs vers ces blocs de données. On effectue donc le calcul suivant (sur les blocs) :

- Le fichier compte 4883 blocs de données.
- Les pointeurs directs de l'i-nœud permettent d'accéder à 12 de ces blocs. Il reste donc 4871 blocs de données pour lesquels l'accès se fera à travers l'un des liens indirects.
- Le pointeur de lien indirect simple pointe sur un bloc qui contient 1024 numéros de blocs (pointeurs vers des blocs de données). Nous avons donc ajouté 1 bloc de pointeurs, et il reste $4871 - 1024 = 3847$ blocs à traiter.
- Le pointeur de lien indirect double permet d'accéder à 1024×1024 blocs, ce qui est plus que suffisant. Il suffit d'utiliser 4 blocs de données pour stocker les 3847 pointeurs de blocs permettant d'accéder aux données restantes.

Fragmentation interne (espace alloué mais non utilisé) :

- 4 octets sur l'i-nœud (indirection triple non utilisé)
- $(1024 - 4) \times 4 = 4080$ octets dans le bloc sur lequel pointe le pointeur indirect double.
- $(4096 - 3847) \times 4 = 996$ octets dans le dernier bloc de pointeurs alloué.
- Finalement, 768 octets dans le dernier bloc de données.

La fragmentation interne totale sur le disque est donc de 5848 octets.

Partie 2 : C

Structure physique d'un i-nœud

Fichier `/usr/include/sys/stat.h`.

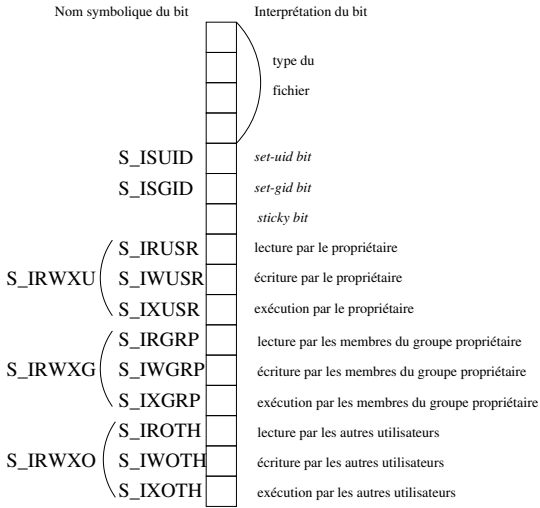
```
#include<sys/types.h>
#include<sys/stat.h>
struct stat{
    dev_t st_dev;     /*id du disque logique du fichier*/
    ino_t st_ino;     /*numero i-node*/
    mode_t st_mode;   /*type et droits d'acces*/
    nlink_t st_nlink; /*nombre de liens physiques*/
    uid_t st_uid;     /*proprietaire du fichier*/
    gid_t st_gid;     /*groupe proprietaire*/
    off_t st_size;    /*taille du fichier en octets*/
    time_t st_atime;  /*date dernier acces*/
    time_t st_mtime;  /*date derniere modif.*/
    time_t st_ctime;  /*date derniere modif. du nœud*/
}
```

`st_mode` : type du fichier (4 bits) + droits d'accès (12 bits).

Structure physique d'un i-nœud

- Les fonctions `stat` et `fstat` permettent d'obtenir dans un objet de structure `stat` les informations relatives à un fichier donné : le fichier est identifié par une référence dans la primitive `stat` et par un descripteur dans la primitive `fstat`. Les prototypes des deux fonctions sont :
- ```
  #include<sys/types.h>
  #include<sys/stat.h>
  int stat(const char *ref, struct stat *ptr_stat);
  int fstat(const int desc, struct stat *ptr_stat);
  ```
- Les deux fonctions retournent 0 si tout se passe bien et -1 en cas d'échec.
  ```

Le champs st_mode



Le champs st_mode : Types de fichiers

st_mode & S_IFMT	type du fichier
S_IFREG	fichier régulier
S_IFBLK	fichier périphérique bloc
S_IFCHR	fichier périphérique caractère
S_IFDIR	répertoire
S_IFIFO	tube nommé
S_IFLNK	lien symbolique
S_IFSOCK	socket

Types de fichier : exemple

```c
struct stat stat_fic;
...
if (stat(argv[1], &stat_fic) != 0) {
    fprintf(stderr, "Erreur a la lecture des informations \
                     relatives au fichier %s\n", argv[1]);
    exit(1);
}
switch (stat_fic.st_mode & S_IFMT) {
    case S_IFREG : /* argv[1] est un fichier régulier */
    case S_IFIFO : /* argv[1] est un tube nommé */
    case S_IFLNK : /* argv[1] est un lien symbolique */
    case S_IFSOCK: /* argv[1] est une socket */
    case S_IFBLK : /* argv[1] est un fichier périphérique bloc */
    case S_IFCHR : /* argv[1] est un fichier périphérique caractère */
    case S_IFDIR : /* argv[1] est un répertoire */
    default      : printf("Type de fichier inconnu\n");
}
```

Un système de fichiers, mais pour quoi faire ?

```
/
|-- bin/ -> /usr/bin/     # Leg historique, voir /usr
|
|-- boot/                 # utilisé lors du démarrage, contient les noyaux,
|                         # initramfs et un bout du système d'amorçage
|
|-- dev/                  # contient les (interfaces vers les) périphériques,
|                         # disques, partitions, souris & clavier, terminaux…
|
|-- etc/                  # "editable text configuration"
|                         # fichiers de configuration du système au format texte
|
|-- home/                 # répertoires personnels (HOME) des users
|
|-- media/                # contient les répertoires servant de point de montage
|                         # pour systèmes de fichiers des périphériques amovibles
|                         # (créés lors du branchement de ces derniers)
|
|-- mnt/                  # idem mais pour les systèmes de fichiers temporaires,
|                         # peuplé à la main
|
```

```
|-- opt/                # logiciels optionnels, qui sont installés manuellement
|                       # et utilisables par tous les users, vide par défaut
|
|-- proc/               # point de montage pour le système de fichiers procfs
|                       # infos sur les processsus, autres infos du noyau
|
|-- root/               # HOME du superuser
|
|-- run/                # "runtime", contient les fichiers temporaires des logiciels,
|                       # en cours d'exécution (ex: fichiers <démon>.pid,
|                       # la majorité des sockets et tubes nommés se trouvent ici
|
|-- srv/                # fichiers varibles utilisés par les services (sites web, …)
|
|-- sys/                # point de montage pour sysfs, donnant info sur les périphériques,
|                       # les drivers, d'autres infos sur le noyau absentes de /proc
|
|-- tmp/                # fichiers temporaires de tout le monde, vidé sur poweroff
|
|-- usr/                # "Unix system ressources" : exécutables, bibliothèques (lib, .so)
|                       # includes (.h), icônes, fichiers .desktop, …
|
`-- var/                # fichiers variables utilisés par le système (logs,
                        # mails, cache gestionnaire de paquets, bdd, etc…)
```

Droits d'accès

- Le système Linux est un système multi utilisateurs où l'accès aux fichiers est contrôlé par des permissions d'accès.
- Tout fichier possède des droits. La commande `ls -l` permet de les afficher.
- Il existe quatre types de droits :
  - r $\Longrightarrow$ **r**ead droit en lecture
  - w $\Longrightarrow$ **w**rite droit en écriture
  - x $\Longrightarrow$ e**x**ecute droit d'exécution
  - - $\Longrightarrow$ aucun droit

Droits d'accès

- Droits associés aux fichiers.

  (r)ead : permet la lecture d'un fichier, ce qui autorise par exemple la copie du fichier (`cat`, `less`, `cp`, ...)

  (w)rite : permet de modifier le contenu d'un fichier (`cat >>`, `vi`, ...)

  e(x)ecute : permet de considérer le fichier comme une commande (fichier binaire ou script)

  (-) : aucune permission

Droits d'accès

- Droits associés aux fichiers.

  (r)ead : permet la lecture d'un fichier, ce qui autorise par exemple la copie du fichier (`cat`, `less`, `cp`, ...)

  (w)rite : permet de modifier le contenu d'un fichier (`cat >>`, `vi`, ...)

  e(x)ecute : permet de considérer le fichier comme une commande (fichier binaire ou script)

  (-) : aucune permission

- Droits associés aux répertoires

  (r)ead : permet de lire le contenu du répertoire (`ls`)

  (w)rite : permet de modifier le contenu du répertoire. Autorise donc la création et la suppression de fichiers dans le répertoire à condition que la permission 'x' soit activée (`touch`, `cat >`, `vi`);

  e(x)ecute : permet d'entrer dans le répertoire (`cd`)

  (-) : aucune permission

Droits d'accès

- la commande `chmod` permet de changer les droits d'accès associés à un fichier/répertoire.
- seul le propriétaire du fichier ou l'administrateur peuvent effectuer cette opération.
- Syntaxe
  - `chmod [OPTIONS]... MODE[,MODE] FICHIER...`
  - deux syntaxes sont possibles
    - la méthode octale (numérique)
    - la méthode symbolique (litérale)

Droits d'accès

Principe de la méthode octale

- `r` est représenté par 4
- `w` est représenté par 2
- `x` est représenté par 1

Droits d'accès

Principe de la méthode octale

- `r` est représenté par 4
- `w` est représenté par 2
- `x` est représenté par 1

```
$ chmod 741 fichier
```

u			g			o		
r	w	x	r	-	-	-	-	x
4	2	1	4	0	0	0	0	1
7			4			1		

Droits d'accès

Principe de la méthode symbolique

```
[ ugoa ] [ +-= ] [ rwx ]
```

Droits d'accès

Principe de la méthode symbolique

```
[ ugoa ] [ +-= ] [ rwx ]
```

Exemples :

- `chmod go+r fichier`
- `chmod a-x fichier`
- `chmod u=rwx,g=r,o=x fichier`

Droits d'accès

Principe de la méthode symbolique

```
[ ugoa ] [ +-= ] [ rwx ]
```

Exemples :

- `chmod go+r fichier`
- `chmod a-x fichier`
- `chmod u=rwx,g=r,o=x fichier`
- `find . -type d -exec chmod 750 {} \;`
- `find . -type f -exec chmod ugo-x {} \;`

Droits d'accès

1. À quels droits correspondent les entiers suivants ?
   751
   521
   214
   150
2. Par quels entiers sont codés les droits `rw-r--r--` et `rwxr-xr-x` ?

Droits d'accès

1. À quels droits correspondent les entiers suivants ?
   - 751 `rwx r-x --x`
   - 521
   - 214
   - 150
2. Par quels entiers sont codés les droits `rw-r--r--` et `rwxr-xr-x` ?

Droits d'accès

1. À quels droits correspondent les entiers suivants ?
   - 751 `rwx r-x --x`
   - 521 `r-x -w- --x`
   - 214
   - 150
2. Par quels entiers sont codés les droits `rw-r--r--` et `rwxr-xr-x` ?

Droits d'accès

1. À quels droits correspondent les entiers suivants ?
   751 `rwx r-x --x`
   521 `r-x -w- --x`
   214 `-w- --x r--`
   150
2. Par quels entiers sont codés les droits `rw-r--r--` et `rwxr-xr-x` ?

Droits d'accès

1. À quels droits correspondent les entiers suivants ?
   - 751 `rwx r-x --x`
   - 521 `r-x -w- --x`
   - 214 `-w- --x r--`
   - 150 `--x r-x ---`
2. Par quels entiers sont codés les droits `rw-r--r--` et `rwxr-xr-x` ?

Droits d'accès

1. À quels droits correspondent les entiers suivants ?
   - 751 `rwx r-x --x`
   - 521 `r-x -w- --x`
   - 214 `-w- --x r--`
   - 150 `--x r-x ---`
2. Par quels entiers sont codés les droits `rw-r--r--` et `rwxr-xr-x` ?
   644, 755.

Droits d'accès

Aux droits fondamentaux (`rwx`) s'ajoutent 3 attributs spéciaux :

- le sticky-bit
- le set-user-ID (suid)
- le set-group-ID (sgid)

L'administrateur doit impérativement en connaître la signification car ils sont fondamentaux pour la sécurité du système.

Droits d'accès

- Le sticky-bit
  - **sur un fichier** : lorsque le sticky-bit est positionné sur un fichier exécutable, le code du programme reste résident en mémoire après qu'il ait été exécuté.

Droits d'accès

- Le sticky-bit
  - **sur un fichier** : lorsque le sticky-bit est positionné sur un fichier exécutable, le code du programme reste résident en mémoire après qu'il ait été exécuté.
  - **sur un répertoire** : un utilisateur qui a le droit d'écrire dans un répertoire, peut également supprimer tous les fichiers qui s'y trouvent. Le sticky-bit positionné sur ce répertoire va y remédier, ainsi un utilisateur ne pourra effacer que les fichiers qui lui appartiennent.

Droits d'accès

- Le sticky-bit
  - **sur un fichier** : lorsque le sticky-bit est positionné sur un fichier exécutable, le code du programme reste résident en mémoire après qu'il ait été exécuté.
  - **sur un répertoire** : un utilisateur qui a le droit d'écrire dans un répertoire, peut également supprimer tous les fichiers qui s'y trouvent. Le sticky-bit positionné sur ce répertoire va y remédier, ainsi un utilisateur ne pourra effacer que les fichiers qui lui appartiennent.
  - `tugo` : `chmod 1777 rep`, `chmod o+t rep`.

Droits d'accès

- Le set uid et set gid (les droits d'endossement)
  - Une commande avec set uid/et gid s'exécute avec l'identité du propriétaire (set uid) ou du groupe propriétaire (set gid),

Droits d'accès

- Le set uid et set gid (les droits d'endossement)
  - Une commande avec set uid/et gid s'exécute avec l'identité du propriétaire (set uid) ou du groupe propriétaire (set gid),
  - Au lieu de donner le droit d'accès à un fichier à une catégorie d'utilisateurs, on donne le droit d'accès à une commande (fichier compilé)

Droits d'accès

- Le set uid et set gid (les droits d'endossement)
  - Une commande avec set uid/et gid s'exécute avec l'identité du propriétaire (set uid) ou du groupe propriétaire (set gid),
  - Au lieu de donner le droit d'accès à un fichier à une catégorie d'utilisateurs, on donne le droit d'accès à une commande (fichier compilé)
  - Quand un utilisateur se connecte sur un système GNU/Linux, il détient 2 UID (UserIDentity) et 2 GID (GroupIDentity) : le réel et l'effectif

Droits d'accès

- Le set uid et set gid (les droits d'endossement)
  - Une commande avec set uid/et gid s'exécute avec l'identité du propriétaire (set uid) ou du groupe propriétaire (set gid),
  - Au lieu de donner le droit d'accès à un fichier à une catégorie d'utilisateurs, on donne le droit d'accès à une commande (fichier compilé)
  - Quand un utilisateur se connecte sur un système GNU/Linux, il détient 2 UID (UserIDentity) et 2 GID (GroupIDentity) : le réel et l'effectif
  - Quand les droits d'endossement ne sont pas positionnés, alors les UID et GID effectifs (ceux de la commande) sont identiques aux UID et GID réels

Droits d'accès

- Le set uid et set gid (les droits d'endossement)
  - Une commande avec set uid/et gid s'exécute avec l'identité du propriétaire (set uid) ou du groupe propriétaire (set gid),
  - Au lieu de donner le droit d'accès à un fichier à une catégorie d'utilisateurs, on donne le droit d'accès à une commande (fichier compilé)
  - Quand un utilisateur se connecte sur un système GNU/Linux, il détient 2 UID (UserIDentity) et 2 GID (GroupIDentity) : le réel et l'effectif
  - Quand les droits d'endossement ne sont pas positionnés, alors les UID et GID effectifs (ceux de la commande) sont identiques aux UID et GID réels
  - Significatifs pour la sécurité du système

Droits d'accès

- Le set uid et set gid

suid	sgid	sticky	u	g	o
s	s	t	rwx	r-x	r--
4	0	0	421	401	400
4	0	0	7	5	4

```
[root]#  chmod  4754  cde
                (2754 pour le guid)
                (u+s / g+s)
```

Droits d'accès

Droits par défaut

- un nouveau fichier possède des "droits par défaut" (ex. "`rwx r-x r-x`" pour un répertoire et "`rw- r-- r--`" pour un fichier)
- Pour cela, le système retire les droits `x` et l'administrateur retire ceux qu'il définit via un masque,
- La commande `umask` permet de connaître la valeur de ce masque. Cette valeur est définie dans `/etc/bashrc` (0022). Le masque est mis en place à la connexion et il reste actif jusqu'à la déconnexion.

Manipulations de fichiers

Les appels systèmes utiles :

`OPEN` pour ouvrir un fichier (création d'une entrée dans la table des fichiers ouverts du processus),

`WRITE` pour écrire (des octets) dans le fichier,

`READ` pour lire (des octets) dans le fichier,

`CLOSE` pour le fermer.

Manipulations de fichiers

Les appels systèmes utiles :

`OPEN` pour ouvrir un fichier (création d'une entrée dans la table des fichiers ouverts du processus),

`WRITE` pour écrire (des octets) dans le fichier,

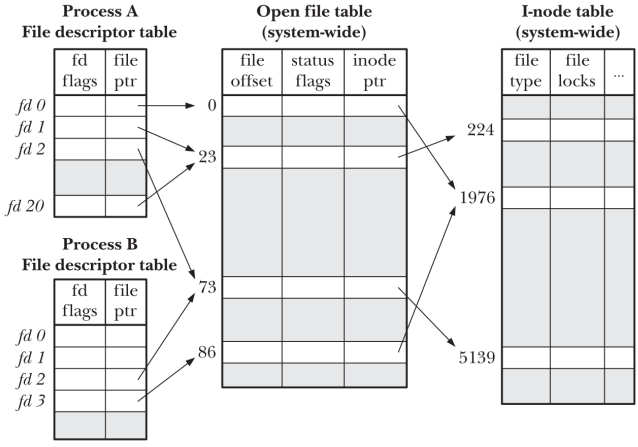
`READ` pour lire (des octets) dans le fichier,

`CLOSE` pour le fermer.

De plus :

- Le système maintient pour chaque processus une *table des fichiers ouverts*.
- À chaque fichier ouvert par le processus est associé un petit entier (son indice dans cette table) appelé *descripteur de fichier*.
- Ce descripteur de fichier est ensuite donné en argument à tous les appels systèmes qui permettent au processus d'interagir avec lui.
- Le système maintient également la position de *la tête de lecture ou écriture* dans le fichier.

Table des fichiers



OPEN

```c
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

`PATHNAME` chemin (relatif ou absolu) dans l'arborescence des fichiers ;

`FLAGS`

- `O_RDONLY` pour lecture seule
- `O_WRONLY` pour écriture seule
- `O_RDWR` pour lecture et écriture (plus rare)

Ces `flags` peuvent être « ou-bit-à-bit-és » avec des attributs de création ou d'états (voir plus loin).

`MODE` dans le cas où il faut créer le fichier, donne les permissions (avant application du masque utilisateur `umask`) associées au fichier à créer.

Exemple : ouverture d'un fichier en lecture

```c
#include <fcntl.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int fd;
    fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        perror("open");
        return 1;
    } else {
        printf("fichier %s ouvert avec succès, descripteur %d\n",
                argv[1], fd);
    }
    return 0;
}
```

Exemple : Ouverture d'un fichier en lecture

Toujours tester la valeur de retour d'`open`.

Exemple : Ouverture d'un fichier en lecture

Toujours tester la valeur de retour d'open.

```
$ ./a.out open_exemple1.c
fichier open_exemple1.c ouvert avec succès, descripteur 3
$ ./a.out open_exemple1.c
open: No such file or directory
$ chmod u-r open_exemple1.c
$ ./a.out open_exemple1.c
open: Permission denied
$ ./a.out
open: Bad address
```

Exemple : Ouverture d'un fichier en écriture

```c
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int fd;
    /* O_CREAT : créer le fichier s'il n'existe pas
     * O_TRUNC : si le fichier existe, le vider
     * 0666 : rw-rw-rw- mais enlever les bits de permissions de l'umask */
    fd = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (fd < 0) {
        perror("open");
        return 1;
    } else {
        printf("fichier %s ouvert avec succès, descripteur %d\n",
                argv[1], fd);
    }
    return 0;
}
```

Quelques états pour une ouverture O_WRONLY ou O_RDWR

- `O_TRUNC` : si le fichier existe, le vider
- `O_APPEND` : toutes les écritures se font à la fin du fichier
- `O_CREAT` : si le fichier n'existe pas, le créer (et utiliser le 3ème argument pour les permissions)
- `O_EXCL` : avec `O_CREAT`, si le fichier existe, échouer

READ

```c
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

`SSIZE_T` *signed size type*, un type entier assez grand, signé pour retourner des valeurs négatives (erreurs)

`FD` le descripteur de fichier du fichier dans lequel on lit des octets

`BUF` adresse dans la mémoire du processus où l'on veut copier ces octets

`COUNT` nombre maximal d'octets que l'on veut lire

- Valeur de retour : nombre d'octets effectivement lus ($\leqslant$ `count`)

READ : EXEMPLE

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    int fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        perror("open");
        return 1;
    }
    ssize_t n;
    char buf[4];
    n = read(fd, buf, 4);
    printf("n = %ld\n", n);
    n = read(fd, buf, 4);
    printf("n = %ld\n", n);
    n = read(fd, buf, 4);
    printf("n = %ld\n", n);
    return 0;
}
```

WRITE

```c
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

- `FD` descripteur de fichier du fichier dans lequel on écrit des octets
- `BUF` adresse dans la mémoire du processus du premier octet à écrire dans le fichier
- `COUNT` nombre maximal d'octets à écrire dans le fichier
- Valeur de retour : nombre d'octets effectivement écrits dans le fichier ($\leqslant$ `count`)

WRITE, EXEMPLE PÉDAGOGIQUE

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int fd = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (fd < 0) {
        perror("open");
        return 1;
    }
    ssize_t n;
    char buf[] = "Lol !!\n";
    /* On ne veut pas écrire l'octet nul final dans le fichier. */
    n = write(fd, buf, strlen(buf));
    printf("n = %ld\n", n);
    close(fd);

    return 0;
}
```

READ BLOQUANT

- Dans certaines circonstances, `read` est bloquant : il n'y a pas encore d'octets à lire dans le fichier
- le processus est endormi
- réveillé par le système lorsque des octets sont arrivés.
- C'est le cas pour la lecture sur le terminal ou dans un socket.

Un éditeur de texte

```c
/* super_text_editor.c */
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#define BUFSZ 256
int main(int argc, char *argv[])
{
    int fd = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (fd < 0) {
        perror("open");
        return 1;
    }
    ssize_t n;
    char buf[BUFSZ];
    while ((n = read(0, buf, 256)) > 0) {
        printf("%ld octets écrits dans %s\n", n, argv[1]);
        write(fd, buf, n);
    }
    close(fd);
    return 0;
}
```