F.S

Questions principales

Décrivez la structure d'une partition formatée en F.A.T Le schéma ci-dessous représente une partition formatée en F.A.T

```
-----------------------------------------------
| BA | FAT | Copie FAT | Tableau de clusters |
-----------------------------------------------
```

- BA = Boot Area[1]
- FAT = Index

Le tableau de clusters est constitué de clusters de même taille. Un cluster ne peut contenir qu'un seul fichier.

1 cluster == n * secteur défini[2]

La FAT est l'index du tableau de clusters, il est chargé en RAM au démarrage pour des raisons de performance. La FAT contient autant d'entrée que de clusters présent dans le tableau de clusters. Cependant une entrée n'occupe pas nécessairement un cluster. La FAT contient l'adresse des clusters. Elle fait le lien entre les différents clusters d'un fichier.

Une entrée dans la fat peut avoir plusieurs valeurs :

- 0 si le cluster est libre [3]
- Une valeur positive pour désigner l'adresse du cluster suivant.
- -1 si c'est le dernier cluster.
- -2 si le cluster est défectueux. Comment savoir si un cluster est défectueux ? On écrit dedans, on lit ce qu'on a écrit, si les valeurs sont différentes, alors le cluster est défectueux.

Le répertoire racine se trouve trouve toujours au cluster 0 dans une partition formatée en FAT16. En FAT32, le cluster est donné par le MBR.

Les **répertoires en FAT sont des fichiers** particuliers contenant les métadonnées des fichiers présent dans le répertoire. Pour chaque fichier, le répertoire possède un descripteur de 32 bytes.

Les champs d'un descripteur :

[1] BA == BR == Boot Record
[2] cluster == bloc
[3] ne veut pas dire que le cluster est vide, seulement qu'il est libre

- Nom : permet de connaître le nom du fichier.
- Cluster : l'adresse du premier cluster du fichier.
- Dates : dates de modification, création, dernier accès.
- Extension : l'extension du fichier
- Taille : la taille du fichier.[4]

Un répertoire a donc toujours au moins deux descripteurs : le descripteur du répertoire courant et le parent (sauf la racine qui n'a pas de parent).

Avantages de la FAT :

- Très simple à intégrer.

Inconvénients de la FAT :

- Lourd en mémoire si la partition est trop grande.[5]
- Si le nom est trop long, il peut être étalé sur plusieurs entrées.
- Pas de droits d'accès.
- Fragmentation externe, les clusters ne sont pas toujours contigus lorsqu'on écrit et efface beaucoup.
- Fragmentation interne, un cluster ne peut contenir qu'un seul fichier, si celui-ci n'est pas remplit, on perd de la place.

Voir les questions secondaires FAT.

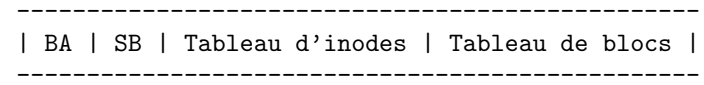
Décrivez la structure d'une partition formatée en EXT/EXT2 `Voir question ext/ext2 (dernière) pour les différences`

EXT :

En EXT, un mini-disque (une partition) est composée de quatres zones :

- Boot Area : Contient les informations sur le démarrage
- Super-bloc : Contient les informations sur le mini-disque
- Tableau d'inodes : contient les inodes (métadonnées des fichiers)
- Tableau de blocs : contient les blocs (les données)

[4] 32Bits !, raison de la taille max d'un fichier de 2^{32}
[5] #bloc * taille entrée FAT



Le tableau d'inode contient un inode par fichier. Pour accéder à un fichier, il faut connaître l'inode, cet inode comprend :

- Nom du propriétaire
- Droits d'accès
- Dates
- Taille
- Type (fichier / dossier)
- Liste de tous les blocs du fichiers

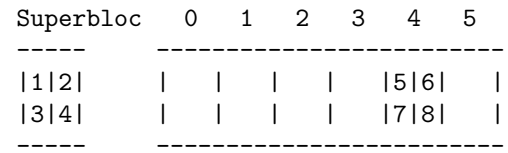
Le chaînage est donc fait dans l'inode lui même. La liste de tous les blocs contient 10 pointeurs (12 en ext2) de blocs direct, et 3 pointeurs avec indirection.

- Le 11ème pointeur contient un seul niveau d'indirection, il pointe vers un tableau de blocs qui pointe vers les blocs de données.
- Le 12ème pointeur contient deux niveaux d'indirection, il pointe vers un tableau de blocs qui pointent eux même sur un tableau de blocs qui pointent sur les blocs de données.
- Le 13ème pointeur contient trois niveaux d'indirection. il pointe vers un tableau de blocs qui pointent eux même sur un tableau de blocs qui pointent eux même sur un tableau de blocs qui pointent vres les blocs de données.

Comme en FAT, un répertoire est un fichier dont les données forment la liste des fichiers qu'il contient. Chaque fichier est représenté par deux champs, son nom et son numéro d'inode. Le répertoire racine en EXT a toujours l'inode numéro 2. Les blocs défectueux ont l'inode 1.

Le superbloc est spécifique à EXT, il contient des informations sur les tableaux d'inodes et de blocs. Il permet de connaître leur position, le nombre d'inodes, le nombre de blocs, la taille d'un bloc.

En EXT : Le superbloc gère également les blocs libre. Il contient un bloc dont les données sont des adresses de blocs libres. La dernière adresse du bloc pointe vers un autre bloc contenant aussi des adresses de blocs libres et ainsi de suite. L'avantage de ce système est qu'il ne prend aucune place sur le disque.



En EXT2 : Le superbloc ne gère plus la gestion des blocs libres, ce sont les blocs bitmap et inodes bitmaps pour la gestion des inodes libres.

Dans l'exemple ci-dessus, les blocs 1, 2, 3, et 4 sont libres dans le superbloc. La dernière adresse est 4. On va donc voir dans le bloc 4 pour les prochains blocs libres, celui-ci nous dit que les blocs 5, 6, 7 et 8 sont libres et qu'il faut aller dans le bloc 8 pour connaître la suite des blocs libres.

Voir questions liens soft/hard pour les liens.

Détaillez comment l'OS permet de découper un disque en plusieurs partitions (primaire, logique, étendue). Quelles sont les limites de cette technique ? Comment évolue-t-elle ? Quels sont les outils utilisés pour gérer ces partition (mkfs, fdisk, mount, unmount, df, /dev, ...) A l'époque, on avait 4 partitions maximum. Le disque a un MBR et chaque partition à un BR.

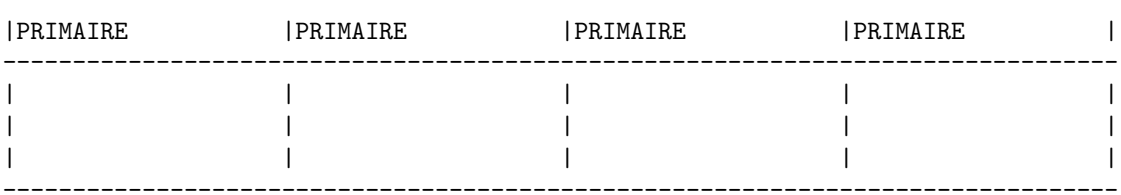
MBR = programme d'amorce + table des partitions.

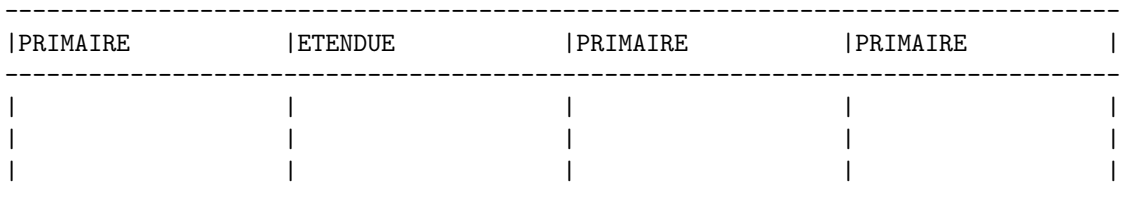
Table des partitions :

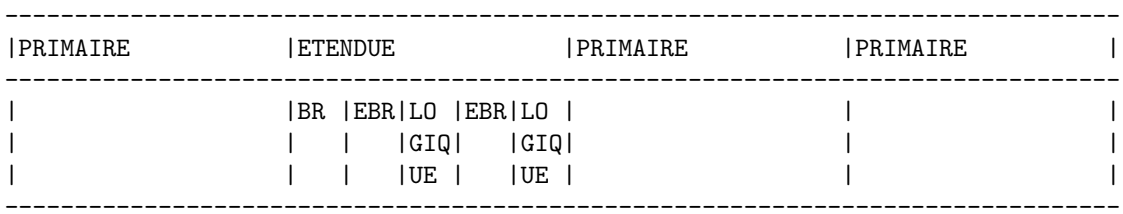
- 4 entrées (16 bytes)
  - status 1 byte
  - CHS 3 bytes (obsolete)
  - ..
  - type 1 byte
  - numéro preimer secteur partition, 4 bytes
  - numéro dernier secteur partition, 4 bytes

Comment avoir plus de partions ?

Partition primaire > partition étendue > partition logique







Chaque partition logique a un EBR == programme d'amorce + table de partitions avec des entrées de type "numéro de secteur + taille + numéro section.

Questions secondaires

(FAT) Détaillez comment l'OS retrouve un fichier, ajoute des données à ce fichier, efface ce fichier Exemple pour retrouver un fichier : `int h = open("/home/user/f1", O_RDONLY);`

- En FAT16 : Le cluster 0 contient la racine. Dans la liste des descripteurs présent dans le répertoire racine, l'OS cherche un descripteur avec le nom "user". Si il le trouve, il cherche le descripteur avec le nom "f1" dans user. Dans le descripteur de f1, on a le premier cluster du fichier ainsi que sa taille. Les clusters suivant sont chainés et sont retrouvés grâce à la FAT.
- En FAT32 : Même principe qu'en FAT16, sauf que le répertoire racine est retrouvé grâce au MBR.

Supprimer un fichier : On met 0 dans la FAT aux clusters correspondant au fichier.

Ecrire dans un fichier : Il est très simple d'écrire dans un fichier, il suffit de changer le chainage en fat si on utilise de nouveaux clusters.

(FAT) Détaillez les situations d'incohérence et montrer comment l'OS récupère cette situation. Lors d'un arrêt brutal du système, la FAT peut comporter des erreurs par rapport au contenu des clusters. Le système analyse le disque :

- Si la suite d'un fichier est un cluster vide, on remplace l'adresse par -1.
- Si on trouve un cluster qui contient des données mais ne semble appartenir à aucun fichier, et qu'il n'est pas libre.

(EXT2) Détaillez comment l'OS retrouve un fichier, ajoute des données à ce fichier, efface ce fichier. handle = open("/usr/home/d", O_RDONLY);

- / (root) == inode 2 => Numero de bloc 28
- bloc 28 => contient l'entrée "usr, inode 35"
- usr/ == inode 35 => Numero de bloc 64
- bloc 64 => contient l'entrée "home, inode 11"
- home/ == inode 11 => Numero de bloc 77
- bloc 77 => contient l'entrée "d, inode 45"
- d == inode 45, l'ensemble des blocs appartiennent au fichier "d".

(EXT2) Détaillez les situations d'incohérence et montrer comment l'OS récupère cette situation Intégrité des liens hard : On parcourt tout le F.S, on regarde le nombre de liens dans l'inode d'un fichier et on compare avec le nombre de fichiers ayant cet inode. Si le compteur d'inode est incohérent, on le corrige.

Fichier fantôme : fichier référencé dans un inode mais aucun fichier avec cet inode n'est référencé dans un répertoire.

(EXT2) Détaillez le contenu d'un super-bloc et l'utilité des champs qui s'y trouvent La réponse est présente dans la question principale sur EXT. + Dernière question sur ext/ext2

(EXT2) Détaillez le contenu d'un inode et l'utilité des champs qui s'y trouvent La réponse est présente dans la question principale sur EXT. + Dernière question sur ext/ext2 ## Questions secondaires

(FAT) Détaillez comment l'OS retrouve un fichier, ajoute des données à ce fichier, efface ce fichier Exemple pour retrouver un fichier : `int h = open("/home/user/f1", O_RDONLY);`

- En FAT16 : Le cluster 0 contient la racine. Dans la liste des descripteurs présent dans le répertoire racine, l'OS cherche un descripteur avec le nom "user". Si il le trouve, il cherche le descripteur avec le nom "f1" dans user. Dans le descripteur de f1, on a le premier cluster du fichier ainsi que sa taille. Les clusters suivant sont chainés et sont retrouvés grâce à la FAT.
- En FAT32 : Même principe qu'en FAT16, sauf que le répertoire racine est retrouvé grâce au MBR.

Supprimer un fichier : On met 0 dans la FAT aux clusters correspondant au fichier.

Ecrire dans un fichier : Il est très simple d'écrire dans un fichier, il suffit de changer le chainage en fat si on utilise de nouveaux clusters.

(FAT) Détaillez les situations d'incohérence et montrer comment l'OS récupère cette situation. Lors d'un arrêt brutal du système, la FAT peut comporter des erreurs par rapport au contenu des clusters. Le système analyse le disque :

- Si la suite d'un fichier est un cluster vide, on remplace l'adresse par -1.
- Si on trouve un cluster qui contient des données mais ne semble appartenir à aucun fichier, et qu'il n'est pas libre.

(EXT2) Détaillez comment l'OS retrouve un fichier, ajoute des données à ce fichier, efface ce fichier. handle = open("/usr/home/d", O_RDONLY);

- / (root) == inode 2 => Numero de bloc 28
- bloc 28 => contient l'entrée "usr, inode 35"
- usr/ == inode 35 => Numero de bloc 64
- bloc 64 => contient l'entrée "home, inode 11"
- home/ == inode 11 => Numero de bloc 77
- bloc 77 => contient l'entrée "d, inode 45"
- d == inode 45, l'ensemble des blocs appartiennent au fichier "d".

(EXT2) Détaillez les situations d'incohérence et montrer comment l'OS récupère cette situation Intégrité des liens hard : On parcourt tout le F.S, on regarde le nombre de liens dans l'inode d'un fichier et on compare avec le nombre de fichiers ayant cet inode. Si le compteur d'inode est incohérent, on le corrige.

Fichier fantôme : fichier référencé dans un inode mais aucun fichier avec cet inode n'est référencé dans un répertoire.

(EXT2) Détaillez le contenu d'un super-bloc et l'utilité des champs qui s'y trouvent La réponse est présente dans la question principale sur EXT. + Dernière question sur ext/ext2

(EXT2) Détaillez le contenu d'un inode et l'utilité des champs qui s'y trouvent La réponse est présente dans la question principale sur EXT. + Dernière question sur ext/ext2

(EXT2) Détaillez les appels système qui permettent d'utiliser le système de fichier (open, read, write, close, dir, dup, lseek, stat, ...) et comment l'OS implémente ces appels système (handle, TDFO, ...)

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

`open` retourne un descripteur de fichier, c'est-à-dire une entrée dans la table des handle du process pointant vers la table des fichiers ouverts (`TDFO`). Un descripteur de fichier reste ouvert par défaut après une fonction `exec` car elle n'écrase pas la table des handle du process.

`creat` retourne également un descripteur de fichier. `open` peut s'occuper de la création si le flag `O_CREAT` est spécifié.

`close` ferme un descripteur de fichier et supprime son entrée de la table des handle. Si le descripteur spécifié en paramètre est le dernier descripteur correspondant à un fichier ouvert dans la `TDFO`, alors les ressources sont libérées dans cette dernière. `close` retourne `0` si il réussit, `-1` sinon.

`open` crée une nouvelle entrée dans la table des descripteurs des fichiers ouverts (`TDFO`) s'il n'est pas déjà ouvert, sinon il partage cette entrée. Cette entrée comprend l'offset et le statut du fichier.

Flags de open :

- `O_RDONLY` : lecture seulement
- `O_WRONLY` : écriture seulement
- `O_CREAT` : création
- `O_RDWR` : lecture et écriture
- `O_APPEND` : offset positionné à la fin du fichier comme si on utilisait un lseek
- `O_TRUNC` : Le fichier est "tronqué" (le contenu est supprimé)

Il est évident que pour `O_APPEND` et `O_TRUNC`, il faut un flag en écriture. L'effet de `O_RDONLY | O_TRUNC` est indéfini et varie selon les implémentations, mais a de grande chance d'être tronqué.

On peut utiliser plusieurs flags en les séparant avec le séparateur de flags |, le `ou binaire`.

Dans le cas d'une création, il faut spécifier un mode. Le mode est ignoré si `O_CREAT` n'est pas présent. Le mode correspond aux droits du fichier une fois créé.

```
#include <unistd.h>
ssize_t read(int fd, void * buffer, size_t buffer_size);
```

- `fd` correspond au descripteur de fichier retourné par `open/creat`, dans la table des handles.
- `buffer` correspond à la zone mémoire où iront les données lues.
- `buffer_size` correspond au nombre d'octet qui doivent être lus à partir de l'offset du fichier spécifié dans la `TDFO`. Si l'offset vaut `E.O.F.`, rien n'est lu et `read()` retourne `0`.
- `read` retourne le nombre de bytes lus ou `0` si la fin du fichier est atteinte. L'offset du fichier est avancé par ce nombre.

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

- `fd` correspond au descripteur retourné par `open/creat`.
- `buffer` correspond à la zone mémoire contenant les données à écrire dans le fichier.
- `buffer_size` correspond au nombre d'octet à écrire.

- `write` retourne le nombre de bytes écrit dans le fichier. Ce nombre peut être plus petit que `buffer_size` si la place est insuffisante sur le disque dur. `write` retourne `-1` si erreur.
- `write` écrit à l'offset du fichier. Cet offset est incrémenté par le nombre de bytes écrit.

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

`lseek` a pour but de repositionner l'offset du fichier ouvert associé au descripteur de fichier `fd` spécifié en paramètre.

- `offset` correspond au décalage par rapport à l'offset courant.
- `whence` correspond à la position de départ du décalage.
  - `SEEK_SET` : correspond au début du fichier.
  - `SEEK_CUR` : correspond à la position actuelle de l'offset.
  - `SEEK_END` : corresond à la fin du fichier.
- lseek retourne la nouvelle position de l'offset par rapport au début du fichier. Si il y a erreur, `-1` est retourné.

```
int position_actuelle = lseek(fd, 0, SEEK_CUR); // Connaitre l'offset de la TDFO
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char * path, struct stat * buf);
int fstat(int fd, struct stat * buf);
int lstat(const char * path, struct stat * buf);
```

`stat` permet d'obtenir des informations contenues dans l'inode d'un fichier. L'utilisateur doit avoir le droit d'accéder au dossier parent de ce fichier mais ne doit pas nécessairement avoir de droits pour ce fichier.

Quelques champs utiles de la structure stat (`man 2 stat` pour voir la structure complète).

- `stat / lstat` :
  - `path` correspond au nom du fichier.

  - `buf` correspond à l'adresse d'une zone où seront stockées les informations correspondantes au fichier (adresse d'une structure stat, voir ci-dessous).

- `fstat`
  - `fd` correspond au descripteur de fichier.
  - `buf` correspond à l'adresse d'une zone où seront stockées les informations correspondantes au fichier (adresse d'une structure stat, voir ci-dessous).

`lstat` est identique à `stat`, sauf que si le fichier est un lien soft, alors les informations obtenues concernent ce lien. Pour `stat`, si le fichier est un lien soft, alors les informations obtenues concernent le fichier sur lequel ce lien pointe.

```
struct stat
{
    ino_t st_ino;           // numéro de l'inode
    mode_t st_mode;         // les permissions
    nlink_t st_nlink;       // Nombre de liens hard
    uid_t st_uid;           // user ID du propriétaire
    gid_t st_gid;           // group ID du propriétaire
    off_t st_size;          // la taille en bytes
    blksize_t st_blksize;   // La taille des blocs
    blkcnt_t st_blocks;     // Nombre de blocs de 512 bytes alloués
    time_t st_atime;        // Date/heure du dernier accès
    time_t st_mtime;        // Date/heure de la dernière modification
};
```

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

`dup` et `dup2` créent une copie de l'entrée du descripteur `oldfd`.

- `dup` crée une copie de oldfd et retourne le plus petit descripteur de fichier non utilisé comme nouveau descripteur.
- `dup2` fait de `newfd` la copie de `oldfd` et ferme `newfd` si celui-ci est ouvert.

Exemple : rediriger la sortie standard vers un fichier :

```
close(1);
int nouveau_h = dup(h_fichier);
close(h_fichier);
```

```
dup2(h_fichier, 1);
close(h_fichier);
```

Ces deux codes ci-dessus font exactement la même chose.

- Dans le cas du `dup` : On ferme `1` qui correspond à la sortie standard, ce qui le rend non utilisé. On fait ensuite un `dup(h_fichier)` qui va copier le descripteur `h_fichier` dans l'entrée la plus petite dans la table des handle, ici `1` étant donné qu'on vient de le fermer. L'entrée `1` et `h_fichier` pointe donc vers le fichier ouvert de `h_fichier` dans la `TDFO`. On ferme ensuite `h_fichier` car on en a plus besoin. L'entrée de la table des handle `1` pointe donc vers le fichier ouvert qui correspondait à `h_fichier` dans la `TDFO`. Quand on va écrire sur la sortie standard (`1`), on va donc écrire sur le fichier.
- Dans le cas de `dup2`, c'est exactement la même chose, sauf qu'il ferme `1` lui-même.

```
#include <sys/types.h>
#include <dirent.h>
DIR * opendir(const char * name);
```

Ouvre un flux sur le répertoire correspondant au nom passé en paramètre.

```
#include <dirent.h>
struct direct * readdir(DIR * dir);
```

- Retourne un pointeur vers une structure dirent qui représente la prochaine entrée dans le flux pointé par `dir`. Retourne NULL si la fin du répertoire est atteinte ou si une erreur survient.
- La structure dirent est détaillée ci-dessous :

```
struct dirent
{
    ino_t d_ino;      // Numéro d'inode
    char d_name[256]; // Le nom du fichier
};
```

Seuls les deux fichiers ci-dessus font partie du standard. Les autres champs (visibles grâce à man 3 readdir) ne sont pas implémentés dans tous les systèmes.

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR * dir);
```

`closedir` ferme le flux associé à `dir`. Retourne -1 si une erreur survient, 0 sinon.

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

`pipe` créé un pipe (canal unidirectionnel qui est utilisé dans la communication inter-process). Le tableau `pipefd` est utilisé pour retourné deux descripteurs de fichiers où :

- pipefd[0] correspond à la sortie du pipe, "la source" du flux qui peut être lu.
- pipefd[1] correspond à l'entrée du pipe, "l'entonnoir" du flux qui peut être rempli.

Exemple : ls | wc -c

```
int p[2];

pipe(p);

if(fork() == 0)
{
    close(p[0]); // Ne lit pas dans le pipe
    dup2(p[1], 1); // La sortie standard devient l'entrée du pipe
    close(p[1]);
    execlp("ls", "ls", NULL);
}

if(fork() == 0)
{
    close(p[1]); // N'écrit pas dans le pipe
    dup2(p[0], 0); // L'entrée standard devient la sortie du pipe
    close(p[0]);
    execlp("wc", "wc", "-c", NULL);
}

close(p[0]);
close(p[1]);
```

(EXT2) Détaillez comment l'OS mémorise les liens à l'aide d'exemple (soft, hard)

- Les liens hards : on souhaite que plusieurs utilisateurs puissent accéder au même fichier mais sous plusieurs noms différents à des path différents (le nom peut être le même car le nom est renseigné dans le répertoire parent, non dans l'inode). Ces deux fichiers partagent le même inode, ce qui signifie que les deux fichiers ne sont en fait qu'un seul et unique fichier. Cependant ces deux fichiers doivent se trouver sur le même mini-disque, car il n'est pas possible de savoir de quel inode on parle sinon. Si un des deux fichiers est modifié, le 2e le sera également étant donné qu'il s'agit du même fichier. Dans l'inode du fichier, un compteur de lien hard est présent, ce compteur est décrémenté quand un des deux fichiers est supprimé et est incrémenté quand un nouveau lien hard est créé sur ce fichier. Les blocs du fichier de base sont libérés seulement quand le compteur passe à 0 (`ls -l` permet de voir le nombre de liens pour chaque fichier).
- Les liens softs : Le lien soft, contrairement au lien hard, est un autre fichier. Les données de ce fichier sont le chemin du fichier pointé par le lien. Dans l'inode de ce fichier, le type est un lien. Un lien soft peut référencer un fichier sur un autre mini-disque. Quand on efface le lien, on efface seulement le lien, pas le fichier. Si on efface le fichier de base, le lien soft existe toujours mais n'est plus valable.

Pour créer des liens sur GNU/Linux :

- Liens hard : `ln fichier nom_lien`
- Liens soft : `ln -s fichier nom_lien`

(EXT2) Détaillez la notion de fichier creux à l'aide d'un exemple (création, taille, occupation du disque) Exemple : On écrit dans un fichier "1" à la position 1 `C int handle = open("file", O_WRONLY); write(handle, "1", 1);`

et "2" à la position 100000.

```
lseek(handle, 100000, SEEK_CUR);
write(handle, "2", 1);
```

Lors d'une lecture, le système retournera des 0 binaires, mais il ne stocke pas sur le disque. Les blocs qui contiennent des 0 **ont l'adresse 0 dans l'inode (aucun bloc réservé)**. Dans ce cas le fichier n'occuperait que 3 blocs sur le disque : celui qui contient le "1", celui qui contient le "2" et le bloc pointé par la 11ème adresse. Alors que normalement un tel fichier devrait occuper 100 blocs.

- Sa taille **est bien de 100000 * 4 = 400000 Kio**.

- Son disk usage **est de 3**.

PS : On se déplace dans un fichier avec un lseek(), expliqué dans une autre question.

(EXT-EXT2) Détaillez la structure d'une partition formatée en EXT et EXT2. Détaillez les avantages d'EXT2 et l'implémentation de ces avantages. Comment EXT2 a t-il évolué ensuite ? \

Défauts de l'EXT :

- Le nom d'un fichier/dossier ne peut pas dépasser 14 caractères.
- Les inodes et les blocs ne se trouvent pas au même endroit ce qui demande beaucoup de déplacement au niveau des têtes de lecture.
- Le superbloc contient des informations vitales pour le système et il n'en existe aucune copie.
- Problèmes de fragmentation : les blocs d'un fichier sont alloués aléatoirement.

Améliorations apportées par EXT2 :

Structure d'un groupe :

```
-------------------------
| 1 | 2 | 3 | 4 | 5 | 6 |
-------------------------
```

- On peut améliorer les performances (moins de déplacement pour la tête de lecture) en regroupant les blocs d'un fichier dans une même zone.
- Le mini-disque est maintenant divisé en groupes (4 par exemple). Chaque groupe possède une structure similaire au mini-disque.
  - 1 : Copie du super bloc du mini-disque. Contient :
    - nombre de blocs et d'inodes
    - nombre de blocs et d'inodes libres (pour ne pas devoir recompter)
    - la taille d'un bloc
    - le nombre de blocs et d'inodes par groupe
    - D'autres champs comme le nombre de montage, la date de dernière vérification sont présents.
  - 2 : description du groupe

  - 3 : Blocs bitmap : liste de bits où chaque bit représente un bloc. Si ce bite vaut 1, le blot est utilisé, sinon il est libre.
  - 4 : Inode bitmap : liste de bits où chaque bit représente un inode. Si ce bit vaut 1, l'inode est utilisé, sinon il est libre.
  - 5 : Inodes : ils contiennent maintenant 12 adresses directes et 3 indirections.
  - 6 : Blocs

- Pour un répertoire, on essaye de regrouper les inodes de ses fichiers dans un même groupe mais de mettre ce répertoire dans un autre groupe que son parent.

- Lorsqu'on veut allouer un bloc à un fichire, on fait la démarche suivante :
  - On recherche un bloc voisin au dernier bloc alloué. On essaye qu'ils se trouvent sur la même piste à environ 16 blocs d'écart.
  - Sinon on réserve 8 blocs libres (overbooking = réserver plus que ce qu'il ne faut), ce qui correspond à un byte dans le bloc bitmap.
  - Sinon on recherche un bloc isolé dans le même groupe.
  - Sinon on réserve 8 blocs dans un autre groupe.

- Les noms de fichiers et répertoires ne sont plus limité à 14 bytes, possibilités d'avoir des noms longs. Une entrée d'un répertoire est maintenant structurée de la manière suivante :
  - Numéro d'inode sur 4 bytes
  - Longueur de l'entrée sur 2 bytes
  - Longueur du nom de fichier sur 2 bytes
  - Nom de fichier sur x bytes où x est un multiple de 4.

- Si on doit effacer une entrée, la taille de la précédente augmente pour occuper l'espace libre. Cet espace pourra être utilisé pour une nouvelle entrée.

- Si on renomme un fichier et que ce nouveau nom dépasse l'espace réservé, on copie l'entrée à la fin du dossier, on renomme le fichier et on efface l'ancienne entrée comme décrit ci-dessus.

- Les liens symboliques sont spécifiques à EXT2.