# Programmation système

## Ressource R3.05 - Système de fichiers

monnerat@u-pec.fr

IUT de Fontainebleau

# Sommaire

# Le système de fichier Linux Ext2(3-4)

# Le système de fichier Linux Ext2(3-4)

## Introduction

# Système de fichiers : pourquoi ?

- Pour l'os, le périphérique de stockage est un (gros) tableau d'octets.
- secteur = unité minimum de données qu'un périphérique peut lire ou écrire.

**Besoin d'une abstraction et d'une api**

- Organisation et nommage des fichiers (notion de path),

  `/IUT/C/prog.c` vs les octets du secteurs 1500 à 1503
- Gestion de la (dés)allocation des secteurs pour les fichiers,
- Accélaration des accès (cache),
- Récupération en cas de problème (journalisation).

# Layout

Plusieurs possibilités pour allouer les blocs de données à un fichier :

- Allocation contigüe. CD, DVD.
  Fragmentation.
- Allocation chaînée (liste chaînée). On stocke (au moins) dans les entrées du répertoire le premier bloc du fichier. Adpaté au pattern d'accès séquentiel, mais pas pour l'accès direct (pourquoi ?).
- Variante : on stocke les numéros de bloc dans une table séparée : FAT
- Allocation indéxée : chaque fichier possède sa propre table d'allocation de blocs. La table est généralement sur plusieurs niveaux. (Ext2/3/4)

# Le système de fichier Linux Ext2(3-4)

## Principes

## inode

Chaque fichier est représenté par une structure `inode`. Le numéro de l'inode identifie de manière unique le fichier dans le SGF.

En particulier, chaque inode contient la liste des blocs de données pour le contenu du fichier.

## directory

La correspondance entre inode et nom symbolique se fait grâce aux `directories`. fichier contenant une liste de correspondances :

`(#inode,nom).`

Remarque :

Un même inode peut être présent dans plusieurs directory $\Rightarrow$ hard link

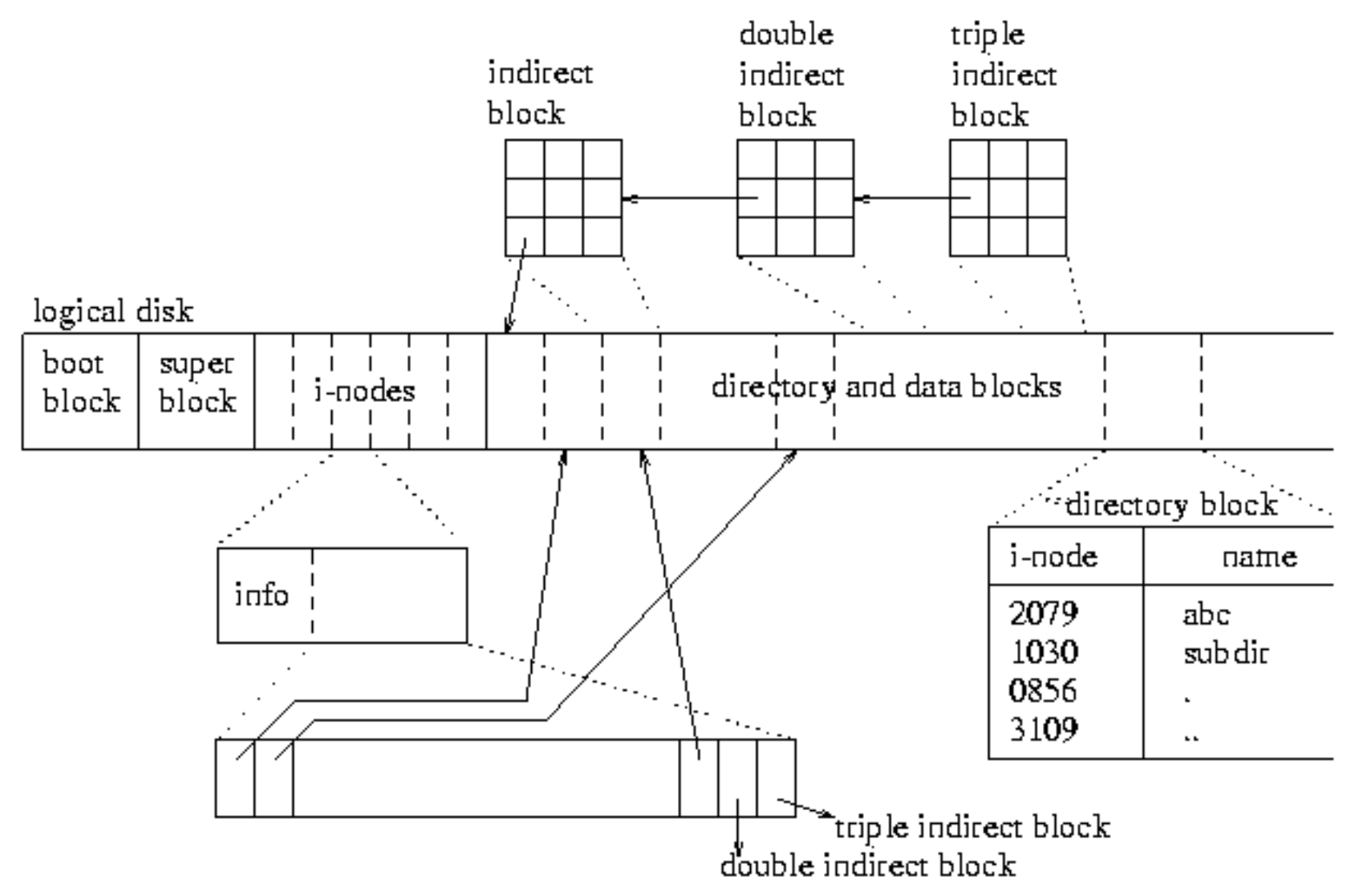


**Figure 1:** Apreçu du SGF

# Le système de fichiers

Organisation logique du périphérique de stockage (disque, clé usb, etc.) avec des données "administratives" utilisés par le SE.

Pour linux, ext2/3/4.

Partionnement en groupes de blocs (bloc = unité élémentaire)

Chaque groupe de blocs possède :

- une réplique du superbloc (redondance)
- `bg_block_bitmap` : table (1 bloc) d'occupation (bitmap) des blocs,
- `bg_inode_bitmap` : table (1 bloc) d'occupation des inodes,
- `bg_inode_table` : table des inodes.
- les blocs de données pour les fichiers.

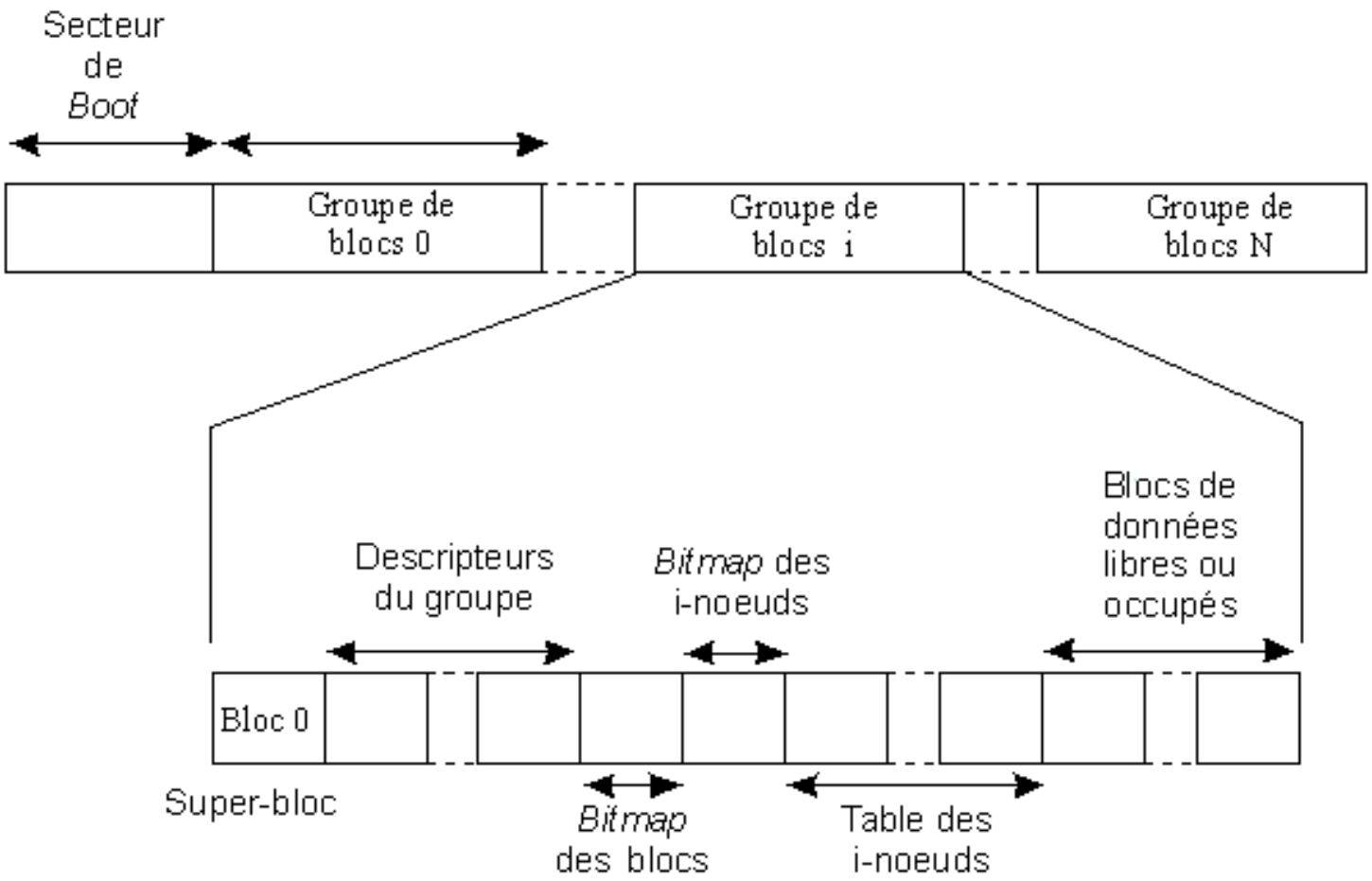


**Figure 2:** Découpage ext2

- Les fichiers sont représentés par des inodes.
- Les répertoire sont simplement des fichiers contenant une liste d'entrées.
- L'interaction avec les équipements (device) se fait par l'intermédiaire de requêtes E/S (I/O) sur des fichiers spéciaux.

# Le système de fichier Linux Ext2(3-4)

## Inode

# Chaque fichier est représenté par une structure : inode.

```
struct ext2_inode {
  /*00*/  __u16   i_mode;     /* File mode */
  __u16   i_uid;      /* Low 16 bits of Owner Uid */
  __u32   i_size;     /* Size in bytes */
  __u32   i_atime;    /* Access time */
  __u32   i_ctime;    /* Inode change time */
  /*10*/  __u32   i_mtime;    /* Modification time */
  __u32   i_dtime;    /* Deletion Time */
  __u16   i_gid;      /* Low 16 bits of Group Id */
  __u16   i_links_count;  /* Links count */
  __u32   i_blocks;   /* Blocks count */
  /*20*/  __u32   i_flags;    /* File flags */
  struct {
    __u32   l_i_version; /* was l_i_reserved1 */
  } linux1;
  /*28*/  __u32   i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
  /*64*/  __u32   i_generation;   /* File version (for NFS) */
  __u32   i_file_acl; /* File ACL */
  __u32   i_size_high;
  /*70*/  __u32   i_faddr;    /* Fragment address */
  struct {
    __u16   l_i_blocks_hi;
    __u16   l_i_file_acl_high;
    __u16   l_i_uid_high;   /* these 2 fields    */
    __u16   l_i_gid_high;   /* were reserved2[0] */
    __u16   l_i_checksum_lo; /* crc32c(uuid+inum+inode) */
    __u16   l_i_reserved;
  } linux2;
};
```

Cette structure inode contient la description complète d'un fichier :

- le type de fichiers et droits associés (`i_mode`)
- la taille (`i_size`) pour un fichier régulier
- l'identité du propriétaire et du groupe (`i_uid` et `i_gid`)
- nombre de références (`i_links_count`)
- le nombre de blocs et les numéros des blocs (`i_blocks` et `i_block[]`)
- des temps (création, changement, etc.)

Les blocs alloués au fichier sont indéxés par le tableau `i_block[]` :

- 0-11 sont directs
- 12 est indirect
- 13 est doublement indirect
- 14 est triplement indirect

Exemple avec des blocs de `1ko` et des numéros de blocs sur 4 `octets`.

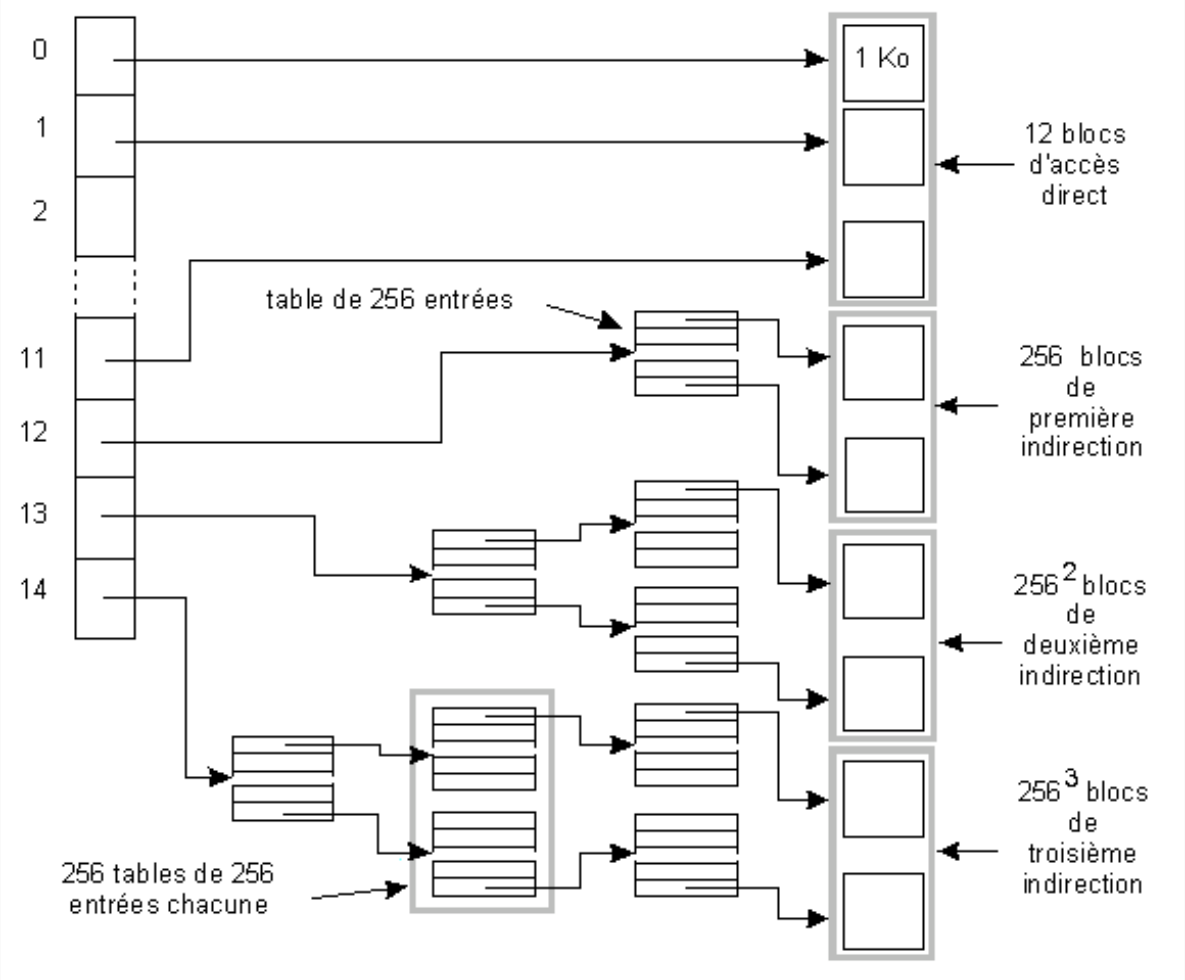


**Figure 3:** blocs d'un fichier

A chaque fichier correspond des blocs de données pour stocker son contenu.

Les numéros des blocs sont dans l'inode.

Lors d'une requête E/S sur un fichier, le noyau :

- convertit l'offset en un numéro de bloc.
- ce numéro est utilisé comme index dans la table des adresses de blocs.
- lit/écrit les blocs physiques correspondant.

Il y a en plus un cache (`buffer cache`) en mémoire centrale (blocs de données et inodes).

# Le système de fichier Linux Ext2(3-4)

## Type de fichiers

# Fichier régulier

Tableau d'octets.

# Les répertoires

Un répertoire est un fichier qui contient des entrées de taille variable :

- Chaque entrée est un couple (inode,nom).
- Les deux premières sont toujours . et ..
- Les répertoires sont le seul moyen d'établir une correspondance entre un fichier et son (ses !) nom(s).
- La taille d'une entrée étant arrondie au multiple de 4 supérieur, les octets de remplissage sont initialisés à 0.
- Lors de la suppression d'un élément dans la table, les octets qui le composent sont ajoutés à la taille de l'élément qui le précède.

```
struct ext2_dir_entry_2 {
  __u32  inode;      /* Inode number */
  __u16  rec_len;    /* Directory entry length */
  __u8  name_len;    /* Name length */
  __u8  file_type;
  char  name[EXT2_NAME_LEN];  /* File name */
};
```

Accès à un fichier avec le nom

Comment "accède-t-on au fichier" à partir d'un path, par exemple `/usr/bin/ls` ?

- On connaît l'inode racine, toujours 2.
- Grâce à cette inode, on a accès au contenu du repertoire racine.
- Dans le repertoire racine, on cherche l'inode correspondant à l'entrée `usr`.
- On recommence.

# Lien

Chaque inode possède un compteur de réference. (nombre de fois où l'inode est réferencé dans un repertoire) On parle de lien dur : hard link.

Ajouter un lien :

- Ajout d'une entrée corresponante à l'inode.
- Incrémentation du compteur de référence dans l'inode.

Effacer un lien

- Supprimer l'entrée du repertoire (`rm` par exemple)
- Décrementer le compteur de réference.
- S'il est nul, l'inode et les blocs de données associés sont libérés.

Restriction

- Uniquement sur des fichiers
- Dans le même système de fichier.

# Lien soft

Fichier dont le contenu est un nom de fichier.

# Fichier spécial périphérique (device)

Un fichier special de device est un point d'accès vers le driver du périphérique correspondant. Il n'occupe pas de place mémoire sur le disque.

2 types :

- character : accès en mode caractère. chaque octet est manipulé individuellement.
- bloc : random accès (bloc). accès par bloc, cache E/S.

Identification : couple

- Numéro majeur identifie le device.
- Numéro mineur identifie l'unité.

Exemple :

```
>ls -l /dev/sda
brw-rw---- 1 root disk 8, 0 22 sept. 14:18 /dev/sda
>ls -l /dev/tty
crw-rw-rw- 1 root tty 5, 0 23 sept. 07:44 /dev/tty
```

Il existe deux autres types de fichiers : tubes només (FIFO) et sockets locales.

# Commande et fonction `stat`

La fonction

```
int stat(const char *restrict pathname,
                 struct stat *restrict statbuf);
```

retourne les informations du fichier `pathname` avec la structure

```
struct stat {
  dev_t      st_dev;      /* ID of device containing file */
  ino_t      st_ino;      /* Inode number */
  mode_t     st_mode;     /* File type and mode */
  nlink_t    st_nlink;    /* Number of hard links */
  uid_t      st_uid;      /* User ID of owner */
  gid_t      st_gid;      /* Group ID of owner */
  dev_t      st_rdev;     /* Device ID (if special file) */
  off_t      st_size;     /* Total size, in bytes */
  blksize_t  st_blksize;  /* Block size for filesystem I/O */
  blkcnt_t   st_blocks;   /* Number of 512 B blocks allocated */


  struct timespec  st_atim;  /* Time of last access */
  struct timespec  st_mtim;  /* Time of last modification */
  struct timespec  st_ctim;  /* Time of last status change */

#define st_atime  st_atim.tv_sec  /* Backward compatibility */
#define st_mtime  st_mtim.tv_sec
#define st_ctime  st_ctim.tv_sec
};
```

## Dans le header `sys/stat.h`

### Macros

```
int S_ISDIR (mode_t m)
int S_ISCHR (mode_t m)
int S_ISBLK (mode_t m)
int S_ISREG (mode_t m)
int S_ISFIFO (mode_t m)
int S_ISLNK (mode_t m)
int S_ISSOCK (mode_t m)
```

### Masques

```
int S_IFMT : This is a bit mask used to extract
             the file type code from a mode value.
int S_IFDIR
int S_IFCHR
int S_IFBLK
int S_IFREG
int S_IFLNK
int S_IFSOCK
int S_IFIFO

S_ISCHR (mode) is equivalent to:
((mode & S_IFMT) == S_IFCHR)
```

# Le système de fichier Linux Ext2(3-4)

## Un systeme de fichier ext2 sur une clé usb

# Création

```
[root@portabledenis ~]$ mke2fs -t ext2 -L "test" /dev/sda1
```

```
[root@portabledenis ~]$ dumpe2fs /dev/sda1
dumpe2fs 1.46.5 (30-Dec-2021)
Filesystem volume name:   test
...
Filesystem OS type:       Linux
Inode count:              121680
Block count:              486392
Reserved block count:     24319
Overhead clusters:        8355
Free blocks:              478031
Free inodes:              121669
First block:              0
Block size:               4096
Fragment size:            4096
Reserved GDT blocks:      118
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         8112
Inode blocks per group:   507
Filesystem created:       Mon Sep 12 11:07:13 2022
Last mount time:          n/a
Last write time:          Mon Sep 12 11:07:30 2022
Mount count:              0
Maximum mount count:      -1
Last checked:             Mon Sep 12 11:07:13 2022
...
First inode:              11
Inode size:           256
Required extra isize:     32
Desired extra isize:      32
Default directory hash:   half_md4
Directory Hash Seed:      5b01648b-bd30-469d-b1eb-b26ea2a0c5ce
```

# Description des groupes de blocs

```
Groupe 0 : (Blocs 0-32767)
 superbloc Primaire à 0, Descripteurs de groupes à 1-1
  Blocs réservés GDT à 2-119
 Bitmap de blocs à 120 (+120)
  Bitmap d'i-noeuds à 121 (+121)
 Table d'i-noeuds à 122-628 (+122)
  32133 blocs libres, 8101 i-noeuds libres, 2 répertoires
 Blocs libres : 635-32767
 I-noeuds libres : 12-8112
...
Groupe 14 : (Blocs 458752-486391)
 Bitmap de blocs à 458752 (+0)
  Bitmap d'i-noeuds à 458753 (+1)
 Table d'i-noeuds à 458754-459260 (+2)
  27131 blocs libres, 8112 i-noeuds libres, 0 répertoires
 Blocs libres : 459261-486391
 I-noeuds libres : 113569-121680
```

Où se trouve la carte des inodes du groupe 0 dans le système crée précédemment ? C'est le bloc 121.

```
[root@portabledenis ~]$ dd if=/dev/sda1 bs=4096c skip=121 count=1 |hexdump -n 16 -C
00000000  ff 07 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000010
1+0 enregistrements lus
1+0 enregistrements écrits
4096 octets (4,1 kB, 4,0 KiB) copiés, 0,000864748 s, 4,7 MB/s
```

On créé un fichier à la racine du système

```
[root@portabledenis test]$ echo "je suis un fichier regulier" > fichier.txt
```

Quel est son numéro d'inode ?

```
[root@portabledenis ~]$ dd if=/dev/sda1 bs=4096c skip=121 count=1 |hexdump -n 16 -C
00000000  ff 0f 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000010
```

## Confirmation

```
[root@portabledenis test]$ stat fichier.txt
  Fichier : fichier.txt
   Taille : 28          Blocs : 8          Blocs d'E/S : 4096   fichier
Périphérique : 8/1  Inœud : 12          Liens : 1
Accès : (0644/-rw-r--r--)  UID : (    0/    root)   GID : (    0/    root)
 Accès : 2022-09-12 13:08:32.209996598 +0200
Modif. : 2022-09-12 13:08:32.209996598 +0200
Changt : 2022-09-12 13:08:32.209996598 +0200
  Créé : 2022-09-12 13:08:32.209996598 +0200
```

- Où se trouve l'inode 12 ? dans la table d'inodes !
- Où est la table d'inodes ? Blocs 122-628 !
- Dans quel bloc ? 4096/256 = 16 inodes par bloc. 12 est dans le bloc 122. (le premier est 1)

```
$ dd if=/dev/sda1 bs=4K skip=122 count=1 | dd bs=256c skip=11 count=1 of=inode12
$ hexdump -C inode12
00000000  a4 81 00 00 1c 00 00 00  30 13 1f 63 30 13 1f 63  |........0..c0..c|
00000010  30 13 1f 63 00 00 00 00  00 00 01 00 08 00 00 00  |0..c............|
00000020  00 00 00 00 01 00 00 00  00 04 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000060  00 00 00 00 9e 85 b0 91  00 00 00 00 00 00 00 00  |................|
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000080  20 00 00 00 d8 2c 11 32  d8 2c 11 32 d8 2c 11 32  | ....,.2.,.2.,.2|
00000090  30 13 1f 63 d8 2c 11 32  00 00 00 00 00 00 00 00  |0..c.,.2........|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000100
```

Où trouve-t-on le nombre et les numéros de blocs de données ? Il faut connaître la structure d'inode !

On voit (expliquez) :

- la taille : 28 octets ;
- le nombre de blocs : 1 bloc ;
- Les numéros des blocs : le bloc 1024.

Comment lire ce bloc ?

```
$ dd if=/dev/sdb1 bs=4096c skip=1024 count=1 | hexdump -C
00000000  6a 65 20 73 75 69 73 20  75 6e 20 66 69 63 68 69  |je suis un fichi|
00000010  65 72 20 72 65 67 75 6c  69 65 72 0a 00 00 00 00  |er regulier.....|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001000
```

# Contenu du repertoire racine

Il faut les blocs de données correspondant à l'inode 2 : 629

```
$ dd if=/dev/sdb1 bs=4K skip=122 count=1 | dd bs=256c skip=1 count=1 of=inode2
$ hexdump -C inode2
00000000  ed 41 00 00 00 10 00 00  af 14 1f 63 30 13 1f 63  |.A.........c0..c|
00000010  30 13 1f 63 00 00 00 00  00 00 03 00 08 00 00 00  |0..c............|
00000020  00 00 00 00 01 00 00 00  75 02 00 00 00 00 00 00  |........u.......|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000080  20 00 00 00 d8 2c 11 32  d8 2c 11 32 58 41 9b b3  | ....,.2.,.2XA..|
00000090  d2 f6 1e 63 00 00 00 00  00 00 00 00 00 00 00 00  |...c............|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000100
$ dd if=/dev/sdb1 bs=4096c skip=629 count=1 | hexdump -C
00000000  02 00 00 00 0c 00 01 02  2e 00 00 00 02 00 00 00  |................|
00000010  0c 00 02 02 2e 2e 00 00  0b 00 00 00 14 00 0a 02  |................|
00000020  6c 6f 73 74 2b 66 6f 75  6e 64 00 00 0c 00 00 00  |lost+found......|
00000030  d4 0f 0b 01 66 69 63 68  69 65 72 2e 74 78 74 00  |....fichier.txt.|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001000
```

# VFS

# VFS

Virtual File System est une interface et des structures de données génériques qui permettent à Linux de pouvoir utiliser des systèmes de fichiers hétérogènes dans la même arborescence.

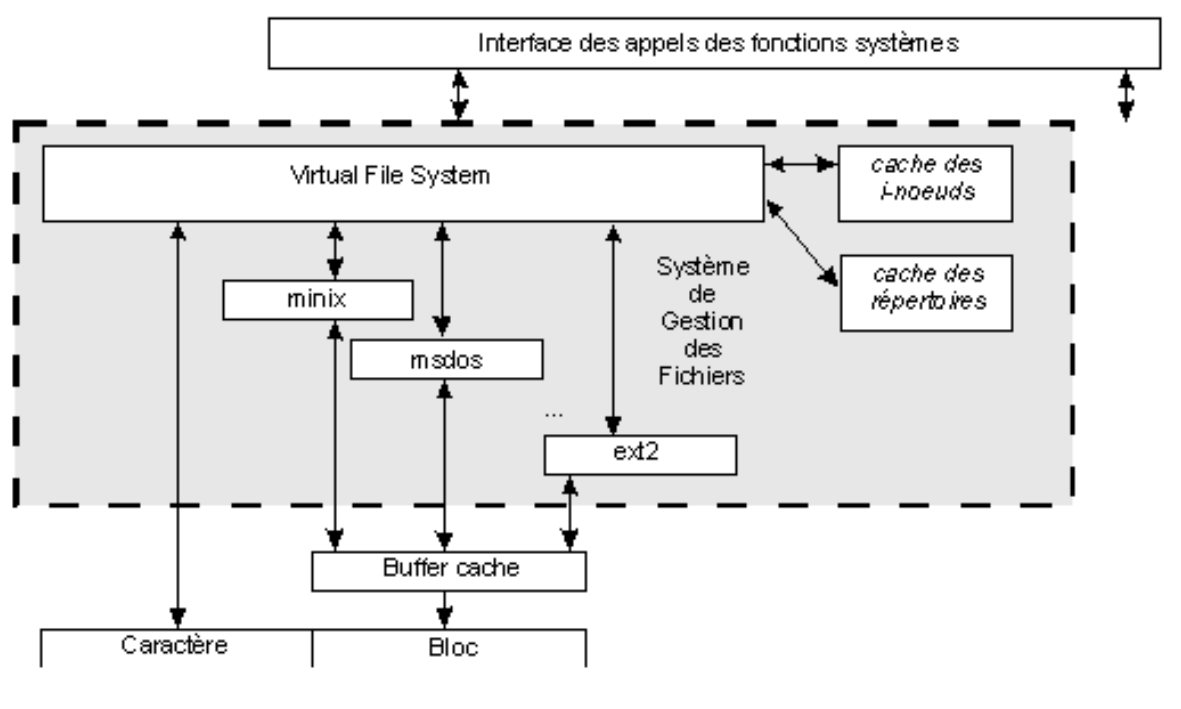


**Figure 4:** VFS

VFS redirige les appels E/S vers le code correspondant des FS,

Un module spécifique à chaque FS installé effectue réellement les instructions qui lui sont transmises par VFS.

VFS supporte des types de systèmes de fichiers très divers :

- "physique" : ext3/4, vfat, iso9660, etc.
- "réseau" : nfs, samba, etc.
- pseudo système de fichiers : proc, sysfs, etc.
- autres : tmpfs, ramfs, rootfs, etc.

VFS gère une table de FS installés, qu'il reconnaît. On retrouve les notions de superbloc et d'inode au niveau de VFS (en s'inspirant largement du cas ext2).
VFS suppose de tout FS :

- qu'il est organisé sous forme d'une arborescence,
- qu'il possède un superbloc, contenant des informations caractéristiques du FS,
- qu'il supporte la notion de i-nœud (i-node). A chaque fichier correspond un inode, structure qui contient la description du fichier.

Le rôle du module correspondant au FS est de simuler ces propriétés s'il ne les possède pas réellement : c'est le cas de `FAT` (msdos) par exemple.
VFS utilise également un cache d'inodes et de répertoires.

# VFS : généricité

VFS définit des structures (et des opérations) génériques

- `inode` : represente un inode,
- `dentry` : represente une entrée dans un répertoire,
- `file` : représente un fichier ouvert,
- `super_block` : represente une instance de filesystem,
- `file_system_type` : represente un type de filesystem.

# Journalisation

# Pourquoi ?

**Rôle**

Le problème que résout la journalisation est celui du maintien de la cohérence du système de fichiers.

Examinons le scénario suivant : on écrit un bloc supplémentaire de données dans un fichier (open, lseek, write, close). Au niveau du FS, cela correspond à

- mise à jour (écriture) de la table d'inode ($e_{ti}$),
- mise à jour de la table de blocs ($e_{tb}$),
- écriture du bloc de données ($e_b$).

Avec le cache, les écritures physiques sont (souvent) différées, et des problèmes peuvent survenir.

Chaque ligne représente un scénario (une croix indique que l'écriture a eu lieu)

| $e_{ti}$ | $e_{tb}$ | $e_b$ | état |
|---|---|---|---|
| | | | ok |
| × | | | nok |
| | × | | nok |
| | | × | ok |
| × | × | | ok |
| | × | × | nok |
| × | | × | nok |

- Dans la plupart des cas, le FS est incohérent.
- Même les cas cohérents pour le FS sont problématiques pour l'utilisateur (pourquoi ?).

Solutions ?

# fsck

**fsck**

```
fsck - check and repair a Linux filesystem
```

- lecture et vérification du superbloc,
- vérification des blocs libres (calcul à partir des inodes, et comparaison avec la table des blocs),
- vérification de l'état des inodes (en particulier, le compteur de liens),
- vérification des pointeurs de blocs : sur-allocations, mauvaises valeurs,
- vérification des repertoires.

Très long !

# Journalisation Ext3

## Journal

Ext3 : il y a juste après le superbloc un journal. (des blocs reservés à cet usage). L'idée est d'écrire dans le journal (log) des transactions.

- un premier bloc TxB (transaction begin), avec un numéro de séquence, et une description de la transaction.
- les blocs de métadonnées (et de données suivant le mode de journalisation)
- Quand les écritures sont terminées, on ferme (commit) la transaction avec l'écriture d'un bloc de fin TxE (avec le même numéro de séquence). Cette dernière écriture est atomique (un seul secteur).
- Ecriture ensuite dans le FS lui-même.

- En cas de crash, on rejoue les transactions valides,
- Le journal est une buffer circulaire (fini !).

Le mode de journalisation le plus courant (compromis) est `ordered`.
Seuls les blocs de metadonnées sont ecrits dans les transactions.

Un extrait avec la commande `jls /dev/sda4` sur mon FS :

```
26414:  Allocated Descriptor Block (seq: 38998013)
26415:  Allocated FS Block 47192132
26416:  Allocated FS Block 47200254
26417:  Allocated FS Block 47185965
26418:  Allocated FS Block 47185975
26419:  Allocated FS Block 0
26420:  Allocated FS Block 8388609
26421:  Allocated FS Block 3
26422:  Allocated FS Block 12
26423:  Allocated FS Block 47185936
26424:  Allocated FS Block 29360132
26425:  Allocated FS Block 8
26426:  Allocated FS Block 29361269
26427:  Allocated FS Block 29360129
26428:  Allocated FS Block 29360131
26429:  Allocated FS Block 29360130
26430:  Allocated FS Block 29361270
26431:  Allocated FS Block 47185959
26432:  Allocated FS Block 27262981
26433:  Allocated FS Block 7
26434:  Allocated Commit Block (seq: 38998013, sec: 1546937637.598224384)
```

# Récupération d'un fichier effacé ?

**Problème**

Récupération d'un fichier effacé avec je journal

- on peut savoir qu'un inode a été effacé (grâce au champs `d_time` deletion time)
- problème : le tableau de blocs est zéroifié !

**Solution**

Retrouver dans le journal une version antérieure de l'inode.

En tp, outre les outils "classiques", vous disposez de la suite

`http://www.sleuthkit.org/`

# API fichiers - primitives

- `open()` : crée une nouvelle description de fichier ouvert dans la table globale des fichiers ouverts. Réserve un nouveau numéro descripteur de fichier pour le processus (et le lui renvoie).
- `read()` : transfère des données depuis le fichier vers la mémoire. Avance la position dans la description de fichier ouvert.
- `write()` transfère des données depuis la mémoire vers le fichier. Avance la position dans la description de fichier ouvert.
- `lseek()` modifie la position dans la description de fichier ouvert.
- `close()` libère le numéro descripteur et, éventuellement, détruit la description de fichier ouvert.
- `fcntl()` modifie les attributs d'un fichier ouvert.

Toutes les primitives travaillent avec des descripteurs de fichiers.

Un descripteur de fichier pointe vers une description d'un fichier ouvert.

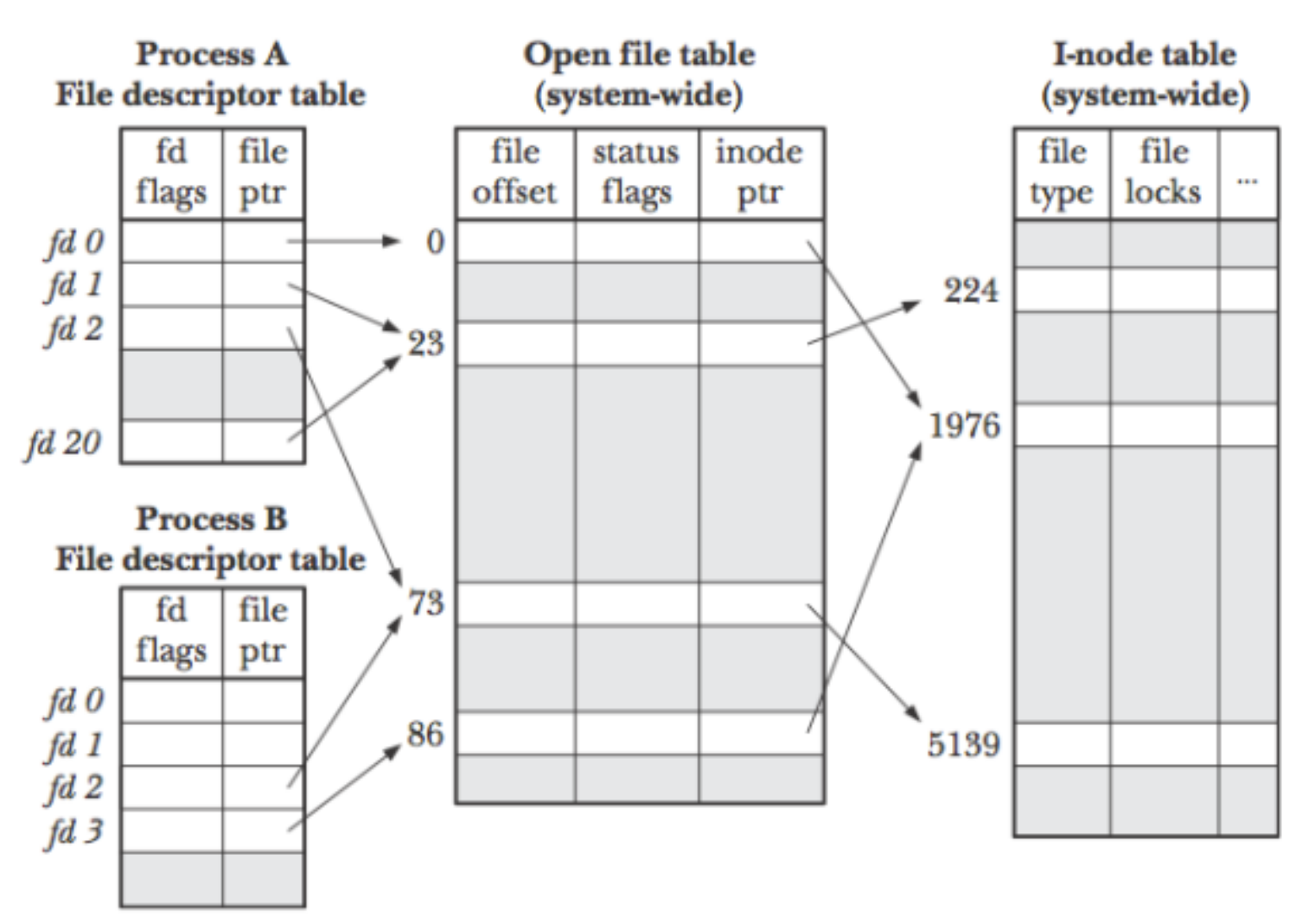


**Figure 5:** descripteur de fichier

Plusieurs cas sont possibles :

- Dans un même processus, 2 descripteurs pointent sur la même entrée dans la table des fichiers ouverts :
  `dup` et `dup2`
- Dans 2 processus distincts, 1 descripteur dans chaque processus pointe sur la même entrée :
  `fork`
- Deux entrées différentes de la table des fichiers ouverts font réference au même inode :
  2 processus ont fait `open` sur le même fichier

# Ouverture

```c
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int oflag, ...);
int open(const char *path, int oflag, mode_t mode);
```

- `path` : le fichier à ouvrir.
- `oflag` : soit `O_RDONLY`,`O_WRONLY`,`O_RDWR`.
  On peut ajouter (ou bit à bit) les masques `O_APPEND`, `O_CREAT`,`O_EXCL`,`O_DIRECT`,`O_SYNC`,`O_NONBLOCK`,etc .
- `mode` : avec `O_CREAT` pour la création, précise les droits d'accès en octal.
- `O_CREAT|O_EXCL` : test et création atomique.

Renvoie le numéro de descriteur attribué au fichier ouvert ( ou -1 si erreur).

1. Le noyau retrouve l'inœud du fichier indiqué par le chemin
    - `O_CREAT` est parmi les drapeaux et le fichier n'existe pas : un nouveau inœud est créé avec les permissions dans mode
    - `O_CREAT` et `O_EXCL` sont parmi les drapeaux et le fichier existe : open() échoue
    - `O_WRONLY / O_RDWR` et `O_TRUNC` sont parmi les drapeaux : le fichier est remis à vide
2. Une nouvelle description de fichier ouvert est créée :
    - les attributs d'état sont les drapeaux (modifiables par `fcntl()`).
    - la position (le numéro du prochain octet à lire ou à modifier) est mise à 0
    - pointe sur l'inœud du fichier.
3. Le premier descripteur libre dans la table de descripteurs du processus pointe sur la nouvelle description.

# Fermeture

```c
#include <unistd.h>

int close(int fd);
```

1. Libère le descripteur
2. S'il n'y a plus de pointeurs sur la description de fichier ouvert, alors l'entrée est detruite.
3. Met à jour les informations dans l'inœud
   - si le nombre de références (hard links) dans l'inœud est 0 et il n'y a plus de descriptions de fichier ouvert pour cet inœud, alors l'inœud est détruit et le fichier n'est plus accessible
   - si close() renvoie -1, alors l'état du fichier est inconnu, perte de données possible

# Lecture

```c
#include <unistd.h>

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

Lit des données à partir de la position courante.

- `fd` : descripteur de fichier.
- `buf` : tampon ou seront stockées les données lues.
- `count` : nombre d'octets à lire.

Renvoie le nombre d'octets lus ($\leq$ *count*), et avance la position associé à `fd`.

- 0 indique la fin du fichier.
- -1 indique une erreur.

# Écriture

```c
#include <unistd.h>

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

Si `O_APPEND` est parmi les attributs d'état, met la position à la fin du fichier. Écrit des données à partir de la position courante.

- `fd` : descripteur de fichier
- `buf` : adresse des données à écrire.
- `count` : le nombe d'octets à écrire.

- Renvoie le nombre d'octets écrits ( $\leq$ *count* ).
- Augmente la valeur de position du même nombre.
- Renvoie -1 en cas d'erreur.

# Déplacement

```c
#include <unistd.h>

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

Modifie la position courante dans la descrition du fichier.

- `fildes` : descripteur de fichier.
- `offset` : déplacement d'octets.
- `whence` :
  - `SEEK_SET` : par rapport au début.
  - `SEEK_CUR` : par rapport à la position courante.
  - `SEEK_END` : par rapport à la fin.

Renvoie la nouvelle position, ou -1 en cas d'erreur.

Attention : tous les fichiers ne sont pas seekables (tube ou socket par exemple)

# mmap

projeter un fichier en mémoire

```
void *mmap(void *addr, size_t length, int prot, int flags,
                    int fd, off_t offset);
```

On ne passe plus par le cache du système.

# Commande cat

```c
int cat(int fd){
  char buf[256];
  int nb;
  while(1){
    nb=read(fd,buf,256);
    if (nb <=0) return nb;
    write(STDOUT_FILENO,buf,nb);/*on suppose que tout
                    est ecrit(sinon coder un full_write)*/
  }
}
int main(int argc,char ** argv){
  int n,fd,ret,i;
  if (argc==1) cat(STDIN_FILENO);
  for(i=1;i<argc;i++){
    fd=open(argv[i],O_RDONLY);
    if (fd == -1) { perror("open error()");
    }else{
      ret = cat(fd);
      if (ret == -1) perror("read() error");
      close(fd);
    }
  }
}
```