# Systèmes de fichiers

Marc Pouzet

Cours 2

# Le système de fichiers

Il permet de stocker les données de façon persistante. Trois contraintes :

- Enregistrer une très grande quantité d'informations ($\geq$ taille mémoire).
- Conserver les informations après la fin du processus qui les utilise.
- Plusieurs processus doivent pouvoir accéder simultanément à une information.

Un programme ne voit pas directement les informations telles qu'elles sont stockées sur le disque. Le système de fichiers en donne une vue abstraite :

- Les détails d'implantation sont cachés ;
- la présentation à l'utilisateur est indépendante de leur représentation sur machine ;
- le système peut préserver des invariants (car l'utilisateur n'a pas accès aux détails de la représentation).

# Pluralité des systèmes de fichiers

- Différents types de systèmes de fichiers
  - exemple : fat (ms-dos), ufs (Unix), ext2 (Linux), NTFS (Windows NT), HFS (MacOs), etc.
- Découplage système d'exploitation / système de fichiers
  - Un système d'exploitation peut fonctionner avec des systèmes de fichiers différents.
- Système de fichiers = abstraction d'un disque
  - plusieurs disques
  - Plusieurs systèmes de fichiers
  - éventuellement différents
  - vue unifiée des systèmes de fichiers présents (une seule hiérarchie)

# Le système de fichiers

**Point de vue externe :**

- Le système de fichiers présente une hiérarchie.
  - Les répertoires « contiennent » des fichiers et des répertoires.
- La notion de fichier ne se limite pas à la notion de fichier sur disque.
  - Les périphériques (e.g., `/dev/tty...`) sont aussi vus comme des fichiers.
- À chaque fichier est associé l'ensemble de ses attributs
  - propriétaire
  - droits d'accès
  - etc.

# Le système de fichiers

## Point de vue interne :

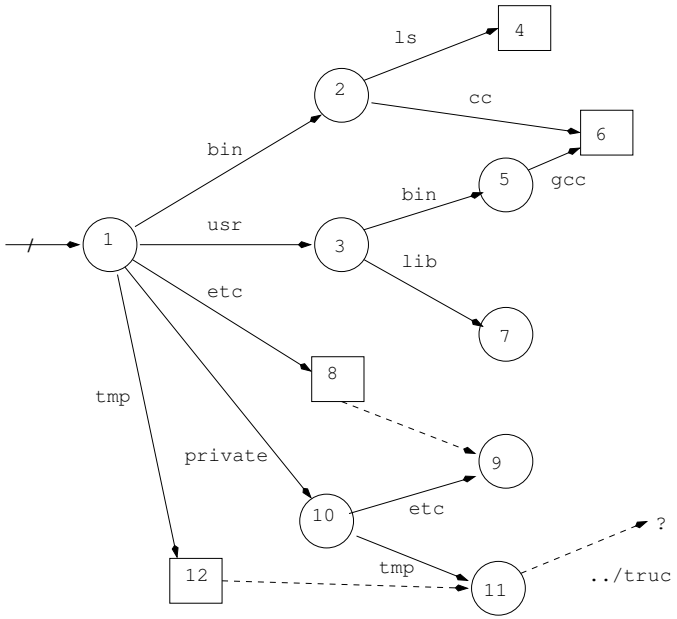
- Le système de fichier n'est pas une hiérarchie : c'est un (énorme) tableau de bits (cf. TP3).
- À chaque fichier correspond un *i-nœud* (*i-node* ou *index node*).
- Un fichier est identifié par une paire :
  - identification de la table qui contient son i-nœud
  - index dans cette table de son i-nœud (`ls -i`)
- Un i-nœud contient :
  - les attributs du fichier
  - et le moyen d'accéder à son contenu.
- Le système maintient en mémoire une table des numéros (inodes) des fichiers ouverts (`lsof`) dont les éléments pointent sur les fichiers (données).

```
% lsof | less
COMMAND    PID   USER   FD  TYPE  DEVICE  SIZE/OFF NODE NAME
ocamlrun  3133 pouzet  cwd  DIR   1,2     3264     525825 /Users/pouzet
ocamlrun  3133 pouzet  txt  REG   1,2     189744   26457446 /Users/pouzet/.opam/4.02.1
                                                             /bin/ocamlrun
...
```

# Hiérarchie

— Hiérarchie = arbre + les répertoires `.` et `..` + liens

— Pas de lien dur entre répertoire (création de cycles, sinon)




— Lien inverse (`..`) et lien sur le repertoire courant (`.`) masqués.

— Liens durs : 6 a deux antécédents.

— Liens symboliques : 8 désigne 9 ; 12 désigne 11. Chemins équivalents de 7 a 6 : `../bin/gcc`, `./../bin/gcc`

— La création d'un lien symbolique ne garantit pas que le fichier existe.

# Système de fichiers : un graphe

— Le système de fichiers est un graphe :
  — Un seul point d'entrée appelé la racine, désigné par `/`
  — Les arcs sont étiquetés par des noms
  — Deux arcs issus d'un même nœud ont des étiquettes différentes.
— Chemins :
  — chaque nœud peut être désigné de façon univoque par un chemin à partir de la racine
  — un chemin est une chaîne de caractères où les mots sont séparés par `/`.
  — Chemin relatif, exemples : `bin/gcc` et `../tmp/foo`
  — Chemin absolu (relatif à partir de `/`), exemples : `/bin/ls` et `/tmp/../usr/lib`

  Pour interpréter un chemin relatif, il faut préciser un nœud de départ (déterminé par le contexte).

# Position dans la hiérarchie

— Chaque processus Unix a une position dans la hiérarchie.
— Les chemins relatifs sont interprétés à partir de cette position.
— Connaître sa position :
  — En shell : `pwd`
  — En C :

```
#include <unistd.h>
char *getcwd(char *tampon, size_t taille);
```

— Changer sa position :
  — En shell : `cd`
  — En C :

```
#include <unistd.h>
int chdir(const char *reference);
```

En OCaml : [a] `Unix.getcwd: unit -> string` et `Unix.chdir: unit -> unit`

a. Compiler avec `ocamlc unix.cma f1.ml ... fn.ml` ou `ocamlopt unix.cmxa f1.ml ... fn.ml`

# Position dans la hiérarchie : Exemple (mpwd)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

#define BUF_SIZE 1024
char buf[BUF_SIZE];
int main () {
  if (NULL == getcwd(buf, BUF_SIZE)) {
    perror("getcwd");
    exit(EXIT_FAILURE);
  }
  printf("%s", buf);
  exit(EXIT_SUCCESS);
}
```

**Exercice :** réécrire `mpwd` en OCaml.

# Création de fichiers

Numéro d'inode d'un fichier.

```
% ls -ia
  53026 .  1123984 ..
% touch a
% touch b
% ls -ia
  53026 .  1123984 ..    53027 a    53028 b
```

# Copie et déplacement de fichiers

```
% ls -ia
  53026 .  1123984 ..    53027 a    53028 b
% cp b c
% ls -ia
  53026 .  1123984 ..    53027 a    53028 b    53029 c
% mv c d
% ls -ia
  53026 .  1123984 ..    53027 a    53028 b    53029 d
```

# Les primitives de base

# Les primitives de base : `open`

— Ouverture d'un fichier :

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

  — `pathname` désigne le fichier à ouvrir
  — `flags` spécifie les opérations que l'on désire réaliser :
    — lecture seule (`O_RDONLY`), écriture seule (`O_WRONLY`), lecture et écriture (`O_RDWR`)
    — créer le fichier s'il n'existe pas (`O_CREAT`), écriture à la fin du fichier (`O_APPEND`), etc.
  — `mode` est utile que lors de la création de fichier
  — La fonction retourne un entier que nous appellerons *descripteur*.
  — Ce descripteur est ensuite utilisé pour effectuer des lectures/écriture du fichier.

Rappel : pour chercher la librairie C d'un appel système : `man 2 open`.

Cette primitive est "relevée" en OCaml (cf. `Unix.openfile`)

## Les primitives de base : `create`

— Création d'un fichier :

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

— Cette fonction est équivalente à :

```
open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode);
```

En OCaml : `Unix.openfile(pathname, [O_WRONLY; O_CREAT; O_TRUNC], mode)`

# Les primitives de base : `read`

— Lecture d'un fichier :

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

  — `fd` : descripteur correspondant au fichier que l'on veut lire.
  — `buf` : adresse de la mémoire où les caractères lus sont écrits.
  — `count` : nombre maximum de caractères lus.
  — valeur de retour :
    — nombre de caractères effectivement lus.
    — `-1` en cas d'erreur. Dans ce cas, la variable `errno` contient le code d'erreur

Cette primitive est "relevée" en OCaml (cf. `Unix.read`)

## Les primitives de base : `read`

— La fonction **`read`** recopie dans **`buf`** les $n$ premiers caractères à partir de la position courante (**`offset`**) où

$$n = min(\texttt{count}, \texttt{longueur du ficher} - \texttt{offset})$$

— Elle incrémente l'**`offset`** de $n$.

Remarque : il n'y a pas de marqueur de fin de fichier.

# Les primitives de base : read (lire-fichier.c)

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

#define TAILLE_TAMPON 256
char tampon[TAILLE_TAMPON];

int main(int argc, char *argv[]) {
  int desc, lus;
  if (1 == argc) { exit(EXIT_FAILURE); }
  if (-1 == (desc = open(argv[1], O_RDWR))) {
    perror("open");
    exit(EXIT_FAILURE);
  }
  do {
    if (-1 == (lus = read(desc, tampon, TAILLE_TAMPON))) {
      perror("read");
      exit(EXIT_FAILURE);
    }
    printf("lecture de %d caracteres\n", lus);
  } while (lus != 0);
  exit(EXIT_SUCCESS);
}
```

# Les primitives de base : `write`

— Écriture d'un fichier :

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

  — `fd` : descripteur correspondant au fichier où l'on veut écrire.
  — `buf` : adresse de la mémoire où les caractères à écrire sont lus.
  — `count` : nombre maximum de caractères à écrire.
  — valeur de retour :
    — nombre de caractères effectivement écrits.
    — `-1` en cas d'erreur. Dans ce cas, la variable `errno` contient le code d'erreur
  — l'écriture change l'`offset` et éventuellement la taille du fichier

Cette primitive est "relevée" en OCaml (cf. `Unix.single_write`). La fonction `Unix.write` répète l'opération d'écriture.

# Les primitives de base : `lseek`

— Déplacement de la position courante :

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

  — `offset` : déplacement de `delta` octets (peut être négatif).
  — `whence` : position à partir de laquelle le déplacement est fait.
    — `SEEK_SET` : début du fichier
    — `SEEK_CUR` : position courante
    — `SEEK_END` : fin du fichier
  — retourne la nouvelle position absolue

— Positionnement possible au delà de la fin du fichier
  — ne change pas la taille
  — écriture à cette position change la taille
  — le « trou » est rempli de zéros

Cette primitive est “relevée” en OCaml (cf. `Unix.lseek`)

## Les primitives de base : `close`

— Fermeture d'un fichier :

```
int close(int fd);
```

— Libération des ressources.

— Bonne pratique : ouvrir et fermer les fichiers dans la même fonction.

Cette primitive est "relevée" en OCaml (cf. `Unix.close`)

# Attributs des fichiers

# Les i-nœuds

- Un i-nœud contient les attributs suivants :
  - Le type du fichier et des droits d'accès des différents utilisateurs
  - L'identification du propriétaire du fichier
  - L'identification du groupe propriétaire du fichier
  - La taille du fichier (en nombre de caractères)
  - Le nombre de liens physiques sur le fichier
  - La date de dernière modification
  - La date de dernière consultation
  - Pour les fichiers sur disque : l'adresse des blocs utilisés sur le disque
  - Pour les fichiers spéciaux : l'identification de la ressource associée
  - etc.
- Un i-nœud contient les adresses des blocs de données du fichier.
- Le système de fichier conserve sur disque une table des i-nœuds.

# Les i-nœuds : les attributs

```
struct stat {
  dev_t   st_dev;    // identificateur du disque logique du fichier
  ino_t   st_ino;    // numero du fichier sur le disque
  mode_t  st_mode;   // type du fichier et droits des utilisateurs
  nlink_t st_nlink;  // nombre de liens physiques
  uid_t   st_uid;    // proprietaire du fichier
  gid_t   st_gid;    // groupe proprietaire du fichier
  off_t   st_size;   // taille du fichier (si cela a un sens)
  quad_t  st_blocks; // blocs alloues
  time_t  st_atime;  // date de dernier acces
  time_t  st_mtime;  // date de dernier modification du contenu
  time_t  st_ctime;  // date de dernier modification du noeud
  ...
};
```

# Librairie Unix d'OCaml

Essentiellement le même type (mais en utilisant un type énuméré pour le type de fichier)

```
type file_kind =
| S_REG  (* Regular file *)
| S_DIR  (* Directory *)
| S_CHR  (* Character device *)
| S_BLK  (* Block device *)
| S_LNK  (* Symbolic link *)
| S_FIFO (* Named pipe *)
| S_SOCK (* Socket *)


type stats = {
   st_dev : int; (* Device number *)
   st_ino : int; (* Inode number *)
   st_kind : file_kind; (* Kind of the file *)
   st_perm : file_perm; (* Access rights *)
   st_nlink : int; (* Number of links *)
   ...
}
```

# Les i-nœuds (`stat.c`)

— Pour accéder aux informations sur les i-nœuds :

  — En shell : `ls -il` ou `stat`

  — En C :

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

  Utilisation typique :

```
struct stat statut;
if (-1 == stat("/tmp", &statut)) {
  perror("stat()"); exit(EXIT_FAILURE);
} ...
```

Cette primitive est “relevée” en OCaml (cf. `Unix.stat "mon_fichier"`)

# Types de fichiers

- Les fichiers réguliers ou ordinaires
  - Suite de caractères caractérisée par sa longueur.
  - Ces fichiers peuvent être des programmes ou des données.
- Les répertoires
- Les fichiers spéciaux
  - Associé à une ressource physique ou logique (disques, imprimantes, terminaux physique ou virtuels, etc.).
  - Mode bloc (comme les disques) : les entrées/sorties transitent par des caches ou *buffer caches* (tampons de la zone de données du noyau).
  - Mode caractère (comme les terminaux) : les entrées/sorties sont réalisées caractère par caractère et ne passent pas par les caches.
- Les liens symboliques
- Les tubes nommés et les sockets

# Types de fichiers : `ls`

— `ls -l` : le premier caractère indique le type de fichier

```
% ls -ld /dev/ /dev/cdrom /dev/sda1 /dev/tty /etc/passwd
drwxr-xr-x 13 root root 14400 2008-02-04 17:43 /dev/
lrwxrwxrwx  1 root root     4 2008-02-04 16:42 /dev/cdrom -> scd0
brw-rw----  1 root disk  8, 1 2008-02-04 17:42 /dev/sda1
crw-rw-rw-  1 root root  5, 0 2008-02-04 16:50 /dev/tty
-rw-r--r--  1 root root  1554 2007-12-21 15:36 /etc/passwd
```

## Types de fichiers : le champ `st_mode`

— Les types de fichier peuvent être testés par les fonctions suivantes :
  — `S_ISREG(mode)` : fichier régulier
  — `S_ISBLK(mode)` : fichier spécial bloc
  — `S_ISCHR(mode)` : fichier spécial caractère
  — `S_ISDIR(mode)` : répertoire
  — `S_ISLNK(mode)` : lien symbolique
  — `S_ISFIFO(mode)` : tube nommé
  — `S_ISSOCK(mode)` : socket

— Le champ `st_mode` contient aussi les droits d'accès.

# Les droits d'accès aux fichiers en Unix

— Trois types d'utilisateurs :
  — Propriétaire (`u`)
  — Membres du groupe propriétaire sauf le propriétaire (`g`)
  — Les autres (`o`)
— Un utilisateur privilégié : numéro de compte `0` (en général `root`)
— Trois types d'accès à un fichier :
  — lecture (`r`)
  — écriture (`w`)
  — exécution (`x`)
— Exemple :

```
% ls -l aucun-droit tous-les-droits
---------- 1 pouzet parkas 0 23 fev 11:46 aucun-droit
-rwxrwxrwx 1 pouzet parkas 0 23 fev 11:47 tous-les-droits*
```

# Les droits d'accès : codage

- Les droits pour chaque type d'utilisateurs sont codés sur trois bits.
  - Exemple : `r-x` = `101`, `rw-` = `110`
- Les droits pour un fichier sont codés sur neuf bits.
  - Exemple : `rw-r-xr--` = `110101100`
- Chaque droit est défini dans dans une constante.
  - droits du propriétaire (position `S_IRWXU`) :
    - `S_IRUSR` : lecture
    - `S_IWUSR` : écriture
    - `S_IXUSR` : exécution

**En OCaml :** ((0oxxx) désigne un nombre écrit en base 8)

```
%ocaml unix.cma
        OCaml version 4.03.0
# Unix.chmod "tous-les-droits" 0o777;;
# Unix.stat "aucun-droit";
- : Unix.stats = {...; Unix.st_perm = 511; ...}
# 0o777;;
- : int = 511
```

## Attributs par défaut à la création du fichier

`umask` : (user file creation mode mask)

Définit les permissions par défaut d'un fichier ou d'un répertoire. Si `m` est la valeur courante du masque (exprimé en base 8), la permission est :

- `NOT(m) AND 0666` pour les fichiers ;
- `NOT(m) AND 0777` pour les répertoires

Le masque le plus courant est `0022`, correspondant à `rw-r--r--` pour les fichiers et `rwxr-xr-x` pour les répertoires.

```
%mkdir toto
%ls -l
=> drwxr-xr-x  2 pouzet  wheel    68 10 mar 09:45 toto/

%umask 0777 -- aucun droit
touch a
%ls -l
=> ----------  1 wheel     0 10 mar 09:51 a
%umask 0077
touch b
%ls -l
=> -rw-------  1 wheel     0 10 mar 09:52 b

%mkdir toto3
%ls -l
=> drwx------  2 wheel    68 10 mar 09:53 toto3/
%umask 0000    -- tous les droits
%touch e
%ls -l
=> -rw-rw-rw-  1 wheel     0 10 mar 09:58 e
```

# Les droits d'accès : exemple (est-executable.c)

```
int main (int argc, char **argv) {
  struct stat statut;

  if (1 == argc) { exit(EXIT_FAILURE); }
  if (-1 == stat(argv[1], &statut)) {
    perror("stat()"); exit(EXIT_FAILURE);
  }
  if (S_ISREG(statut.st_mode)) {
    printf("Fichier ordinaire");
    if (statut.st_mode & S_IXUSR)
      printf(", executable par son proprietaire");
  }
  putchar('\n');
  exit(EXIT_SUCCESS);
}
```

## Les droits d'accès : trois bits supplémentaires

— Le set-uid bit (`s`) :
  — Modifie le comportement des fichiers binaires exécutables.
  — À l'exécution, le processus a les droits du propriétaire du fichier (qui est le propriétaire effectif du processus) et non de l'utilisateur qui le lance (qui en est le propriétaire réel).
  — Exemple :

```
% ls -l /etc/passwd /usr/bin/passwd
-rw-r--r-- 1 root root  1554 2007-12-21 15:36 /etc/passwd
-rwsr-xr-x 1 root root 29104 2007-05-18 11:59 /usr/bin/passwd
```

— Le set-gid bit (`s`) : idem pour le groupe

## Les droits d'accès : trois bits supplémentaires

— Le sticky bit (`t`) :
  — Modifie le comportement des répertoires
  — Interdit la suppression du répertoire et des fichiers contenu dans le répertoire à tout utilisateur autre que le propriétaire (même s'il a les droits en écriture)
  — Exemple :

```
% ls -ld /tmp
drwxrwxrwt 16 root root 36864 2008-02-05 18:34 /tmp
```

# Les droits d'accès : `chmod`

- Changer les droits
  - En shell : `chmod a+rwxt dir`
  - En C :

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

- Tester les droits d'accès
  - En shell : `ls -l`
  - En C :

```
int access(const char *path, int mode);
```

    - `mode` est défini par combinaison de `R_OK`, `W_OK`, `X_OK` et `F_OK`.

Toutes ces fonctions sont relevées en OCaml (cf. `Unix.chmod`, `Unix.fchmod`, `Unix.access`, etc.).

# Le propriétaire : le champ `uid_t`

— Informations sur le propriétaire

```
struct passwd *getpwuid(uid_t uid);

struct passwd {
  char *pw_name   // Nom de login
  uid_t pw_uid    // Numero d'utilisateur
  gid_t pw_gid    // Numero de groupe
  char *pw_dir    // Repertoire initial
  char *pw_shell  // Shell a utiliser
}
```

En OCaml, types `Unix.passwd_entry` et `Unix.group_entry`.

## Le propriétaire : les champs `uid_t` et `gid_t`

— Changement de propriétaire (utilisateur et groupe)
  — En shell : `chown`
  — En C :

```
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Ces fonctions sont relevées en OCaml : `Unix.chown`, `Unix.fchown`.

## Changement d'autres attributs d'un i-nœud

— Changement des dates de dernière modification
  — En C :

```
int utime(const char *filename, const struct utimbuf *buf);
```

— changement de la longueur :
  — En C :

```
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

Ces fonctions sont relevées en OCaml :
`Unix.utimes`, `Unix.truncate` et `Unix.ftruncate`.

# Les différents types de fichiers

# Les répertoires

— Le contenu d'un répertoire est un catalogue de paires :
  — Numéro d'i-nœud
  — Nom de fichier

— Les droits d'accès pour un répertoire correspondent à :
  — lecture : lister le contenu d'un répertoire
  — écriture : création et suppression de fichiers dans le repertoire
  — exécution : se positionner dans le répertoire ou faire figurer ce répertoire dans une référence

# Les répertoires

— Création d'un répertoire
  — En shell : `mkdir a a/b`
  — En C :

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

  — En OCaml :

```
val mkdir: string -> file_perm -> unit
```

— Destruction d'un répertoire
  — En shell : `rmdir a/b a`
  — En C :

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

  — En OCaml :

```
val rmdir : string -> unit
```

  — Peut seulement supprimer un répertoire vide.

# Les répertoires

— Ouverture :

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

— Fermeture :

```
int closedir(DIR *dir);
```

— Lecture :

```
struct dirent {
    ino_t d_ino;        /* numero de l'inode */
    char  d_name[];     /* nom du fichier */
}
struct dirent *readdir(DIR *dir);
```

— Lit une entrée du répertoire

— Rembobinage :

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

Ces fonctions sont relevées en OCaml (mais la structure `dirent` reste abstraite).

# Les répertoires : Exemple (`mls.c`)

```
int main(int argc, char **argv) {
  int ind;
  struct stat s;
  DIR *dir;
  char ref[1024];
  struct dirent *entree;
  for (ind = 1; ind < argc; ind++) {
    if ( -1 == stat(argv[ind], &s) ) {
      fprintf(stderr, "%s fichier inconnu\n", argv[ind]); continue;
    }
    if ( !S_ISDIR(s.st_mode) ) { printf("%s\n", argv[ind]); continue; }
    printf("%s : repertoire\n", argv[ind]);
    if ( NULL == (dir = opendir(argv[ind])) ) {
      fprintf(stderr, "%s repertoire : ouverture impossible", argv[ind]); continue;
    }
    while ( NULL != (entree = readdir(dir)) ) {
      sprintf(ref, "%s/%s", argv[ind], entree->d_name);
      if ( -1 == stat(ref, &s) ) { perror("stat"); continue; }
      if ( S_ISDIR(s.st_mode) ) { printf("r : %s\n", entree->d_name); }
      else { printf("    %s\n", entree->d_name); }
    }
  }
  exit(EXIT_SUCCESS);
}
```

# Les liens physiques

— Les liens physiques permettent de donner plusieurs noms à un même fichier
— Création :
  — En shell : `ln cible nom_du_lien`
  — En C :

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
```

— Renommage :
  — En shell : `mv old new`
  — En C :

```
int rename(const char *from, const char *to);
```

# Les liens physiques

— Exemple :

```
% ls -il
total 0
7466269 -rw-r--r--  1 pouzet  wheel  0 23 fev 12:13 a
% ln a b
% ls -il
total 0
7466269 -rw-r--r--  2 pouzet  wheel  0 23 fev 12:13 a
7466269 -rw-r--r--  2 pouzet  wheel  0 23 fev 12:13 b
```

  — `a` et `b` ont le même i-nœud (première colonne)

  — Le nombre de liens a augmenté (troisième colonne)

— Un répertoire a toujours au moins deux liens vers son i-nœud.

## Les liens physiques : Restrictions

— Il est impossible de créer des liens sur des répertoires
  — cela permettrait de transformer « l'arborescence » des fichiers en un graphe avec cycles
— Il est impossible de créer des liens entre des fichiers résidant sur des systèmes de fichiers différents.

# Les liens symboliques

— Les liens symboliques n'ont pas les contraintes précédentes.
  — Création de cycles.
  — Liens vers fichiers ou répertoires (même inexistants).
— Contenu d'un chemin = lien
  — relatif
  — absolu
— Interprétation du nom
  — le lien symbolique lui-même
  — le fichier qu'il désigne
  — dépend du contexte d'utilisation

```
% rm symlink
% cat symlink
```

# Les liens symboliques

— Créer un lien symbolique
  — En shell : `ln -s cible lien`
  — En C : `int symlink(const char *cible, const char *lien);`
  — En OCaml : `val symlink : string -> string -> unit`
— Lecture
  — En shell : `readlink`
  — En C : `int readlink(const char *path, char *buf, int bufsiz);`
  — En OCaml : `val readlink : string -> string`
  — Utilisation typique :

```
char buf[PATH_MAX+1];
ssizet_t len;
if (-1 != (len = readlink(path, buf, PATH_MAX))) {
  buf[len] = '\0';
} else { perror("error reading link"); }
```

— Consultation des attributs (de la cible) :

```
int lstat(const char *path, struct stat *pstat);
```

En OCaml : `val lstat : string -> stats`

# Les liens symboliques : exemple

```
% ls -il
total 0
7466269 -rw-r--r--  2 pouzet  wheel  0 23 fev 12:13 a
% cat a
aaaa
% ln -s a b
% ls -il
total 16
7466269 -rw-r--r--  1 pouzet  wheel  5 23 fev 12:20 a
7466351 lrwxr-xr-x  1 pouzet  wheel  1 23 fev 12:20 b@ -> a
% cat b
aaaa
```

# Les liens symboliques

— Lien symbolique invalide

```
% ls -il
total 16
7466269 -rw-r--r--  1 pouzet  wheel  5 23 fev 12:20 a
7466351 lrwxr-xr-x  1 pouzet  wheel  1 23 fev 12:20 b@ -> a
% rm a
% cat b
cat: b: No such file or directory
```

— Boucle

```
% ln -s . loop
% cd loop/loop/loop
```

**Remarque :**

— Les droits d'un fichier "lien symbolique" sont calculés à partir du masque (cf. `umask`) avec un droit en exécution, par défaut.

— La commande `chmod` ne change pas les droits du fichier.

# Parcours d'une hiérarchie

— Un fichier est caractérisé par une partition et un i-noeud : deux fichiers différents dans des partitions différentes peuvent avoir le même i-noeud.
— La racine (répertoire `/`) pointe sur elle-même : `ls` ai /-
— Traitement récursif
  — commandes `find`, `du`, etc.
— Traitement selon le type de fichier
  — Fichier ordinaire ⇒ traitement standard
  — Répertoire ⇒ itérer sur toutes les entrées sauf `.` et `..`
  — Lien symbolique : plusieurs choix sont possibles
    — mémoriser les nœuds visités
    — limiter le nombre de liens symboliques traversés

# Suppression

— Suppression de fichier = suppression du lien
  — En shell : `rm`
  — En C :

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

  — En OCaml :

```
val unlink : string -> unit
```

```
% ls -il
% rm a
% ls -il
% rm b
% ls -il
```

## Suppression

— Un fichier peut être supprimé physiquement lorsqu'il n'est plus pointé par aucun lien ou descripteur.
— Système de fichier = « garbage collector » par comptage de références.
— Garantie d'absence de cycles
— Maintien d'une liste des places libres. L'espace occupé par un fichier sur lequel personne ne pointe est libre.

## Suppression : Conditions de suppressions

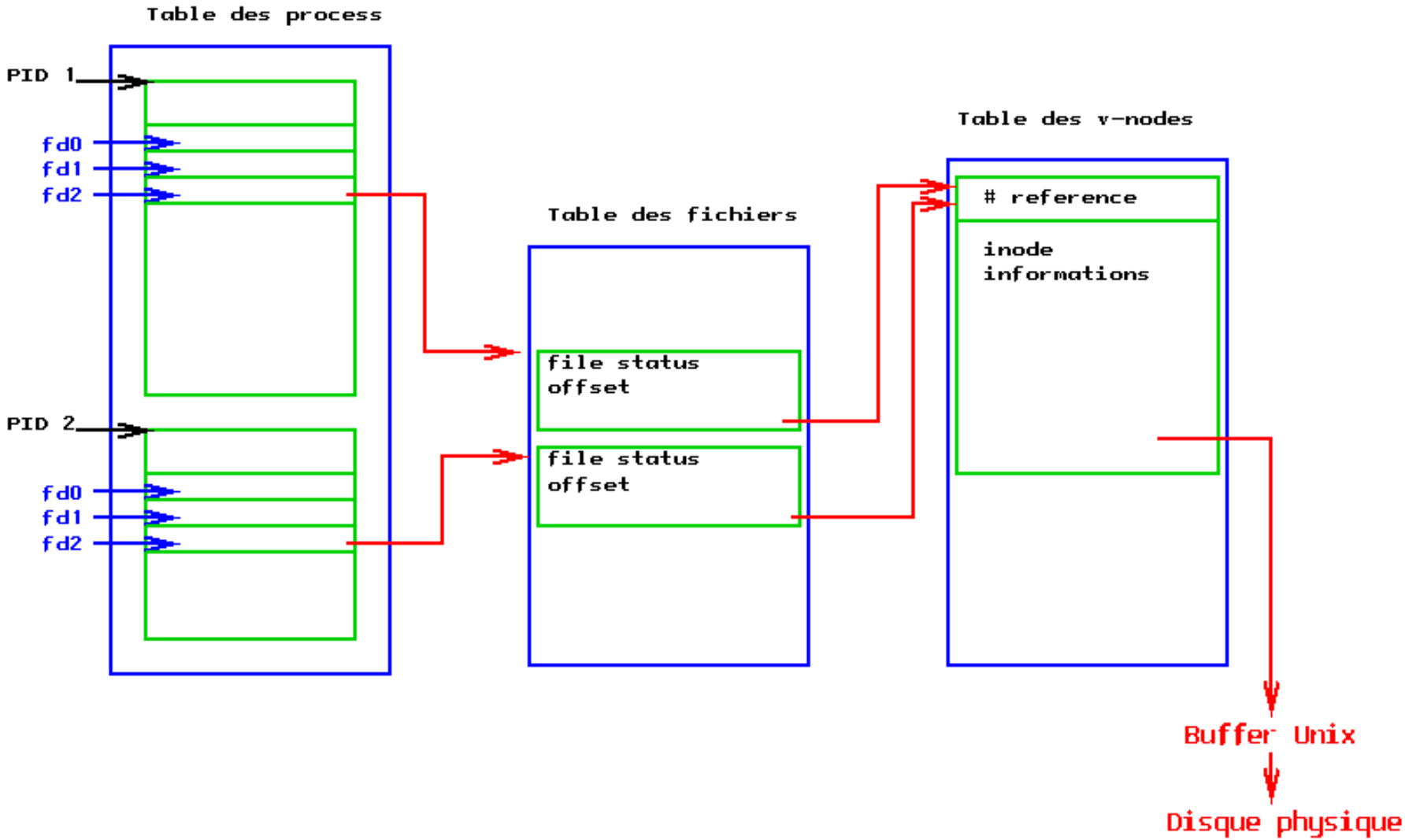
— `rm` (sans option) ne peut pas supprimer de répertoires.
— Il faut (et il suffit de) avoir les droits en écriture sur un répertoire pour supprimer un fichier.
  — Rôle du sticky-bit.

## Vision des fichiers depuis un processus

— Un processus a une table de fichiers ouverts (descripteurs).
— Elle pointe vers une table des fichiers ouverts.
— Celle-ci qui pointe vers la table des v-nodes, interface avec le disque physique.

Plusieurs processus peuvent donc référencer le même fichier ; et même un processus peut référencer le même fichier plusieurs fois avec des positions différentes. On peut aussi sous plusieurs descripteurs de fichiers référencer le même fichier avec la même position (cf. appels système `dup` et `dup2`).

Un descripteur de fichier est un entier positif (0 pour stdin ; 1 pour stdout ; 2 pour stderr).

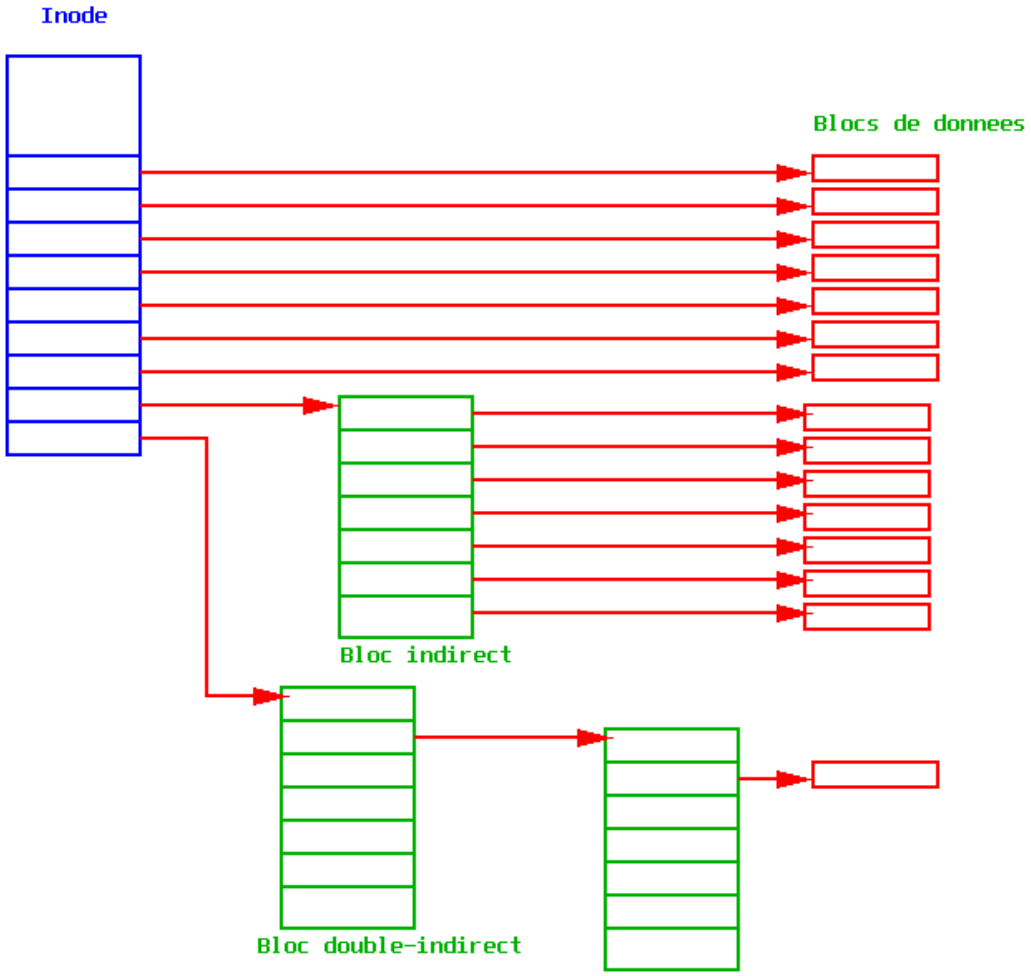



— Pour chaque fichier ouvert : mode d'ouverture, position dans le fichier, etc.

— `% lsof` donne la liste des fichiers ouverts.

# Représentation interne des donnés d'un fichier

Ce qui est stocké dans la table des v-nodes.



## Redirections et tubes

Expliquer au tableau.

**Rmq :** Dans la librarie d'OCaml, le descripteur de fichier est abstrait. Pour une redirection de l'entrée/sortie standard, utiliser `Unix.dup2 fdesc stdin` et `Unix.dup2 fdesc stdout`.

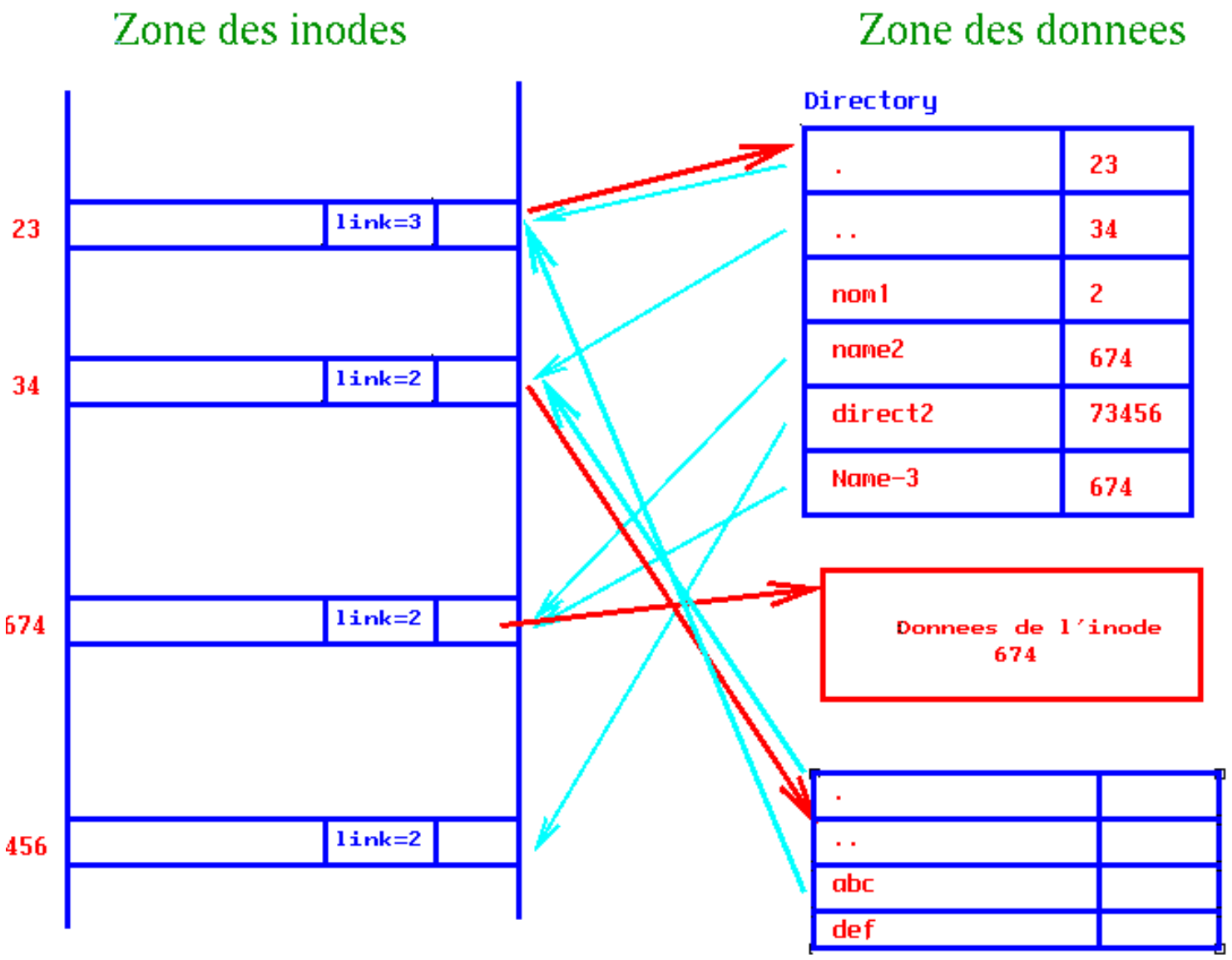
# Répertoires

Un nom pointe sur un numéro d'inode. Plusieurs noms peuvent être associés (pointer sur) le même inode : cas d'un lien "dur".

— L'inode 674 est référencé par les noms `name2` et `Name3`.

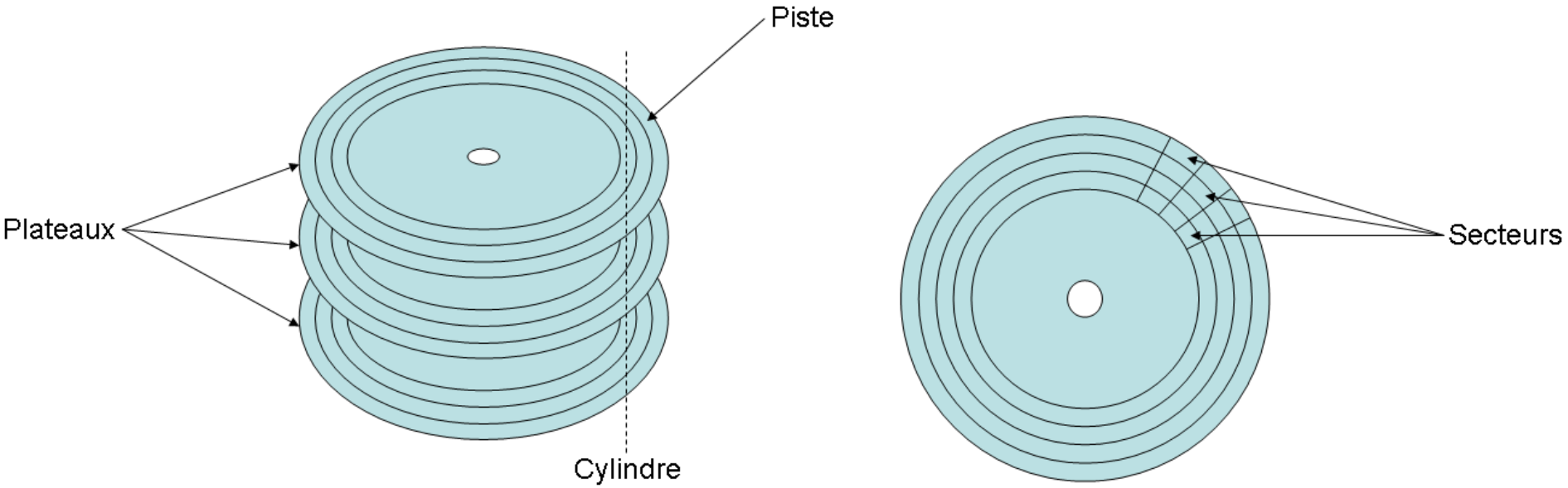
— Un répertoire est un fichier dont les données sont interprétées de façon ad-hoc.

— Un répertoire a toujours deux liens : `.` et son nom dans `...`


Zone des inodes
Zone des donnees
Directory
23
34
674
456
link=3
link=2
link=2
link=2
.
23
..
34
nom1
2
name2
674
direct2
73456
Name-3
674
Donnees de l'inode
674
.
..
abc
def

# Les systèmes de fichiers : implantation

# Structure physique d'un disque




— Secteur : plus petite unité qui peut être lue par un disque (typiquement 512 octets)

Remarque : figure provenant de Wikipedia

# Organisation et codage des secteurs

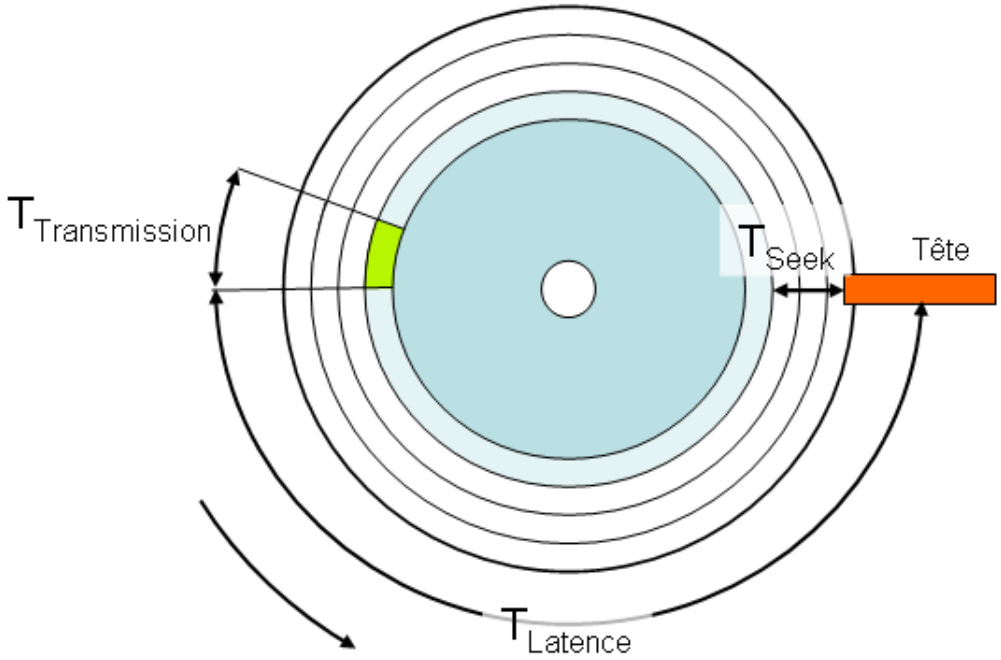
- Adressage
  - CHS (*Cylinder, Head, Sector*)
    {num. cylindre, num. face, num secteur dans la piste}
    - num. cylindre, num. face = num. piste
  - LBA (*Logical Block Address*)
    - Adresses logiques par numéros consécutifs
    - Le contrôleur de disque dispose d'une table de correspondance entre les adresses LBA et CHS

# Formatage de bas niveau

— Exemple de structure d'un secteur


— Informations auxiliaires
  — Synchronisation
  — Identification du secteur
  — Codes de détection et correction d'erreurs (CRCC = Cyclic Redundancy Check Code)
  — Séparations des données par des « *gaps* »
— Le formatage de bas niveau :
  — partage le disque en secteurs
  — marquage des informations auxiliaires

# Performances des disques




— Temps de recherche (*seek time*) : positionnement de la tête en face du bon cylindre
— Temps de latence : positionnement du secteur en face de la tête de lecture (dépend de la vitesse de rotation)
— Temps de transfert : transfert des données vers l'ordinateur

Remarque : figure provenant de Wikipedia

## Stratégies d'ordonnancement de requêtes

— Changer l'ordre des requêtes pour :
  — Réduire les délais de recherche de cylindres
  — Réduire les délais de latence pour un même cylindre
  — Optimisation globale des délais de positionnement et rotation

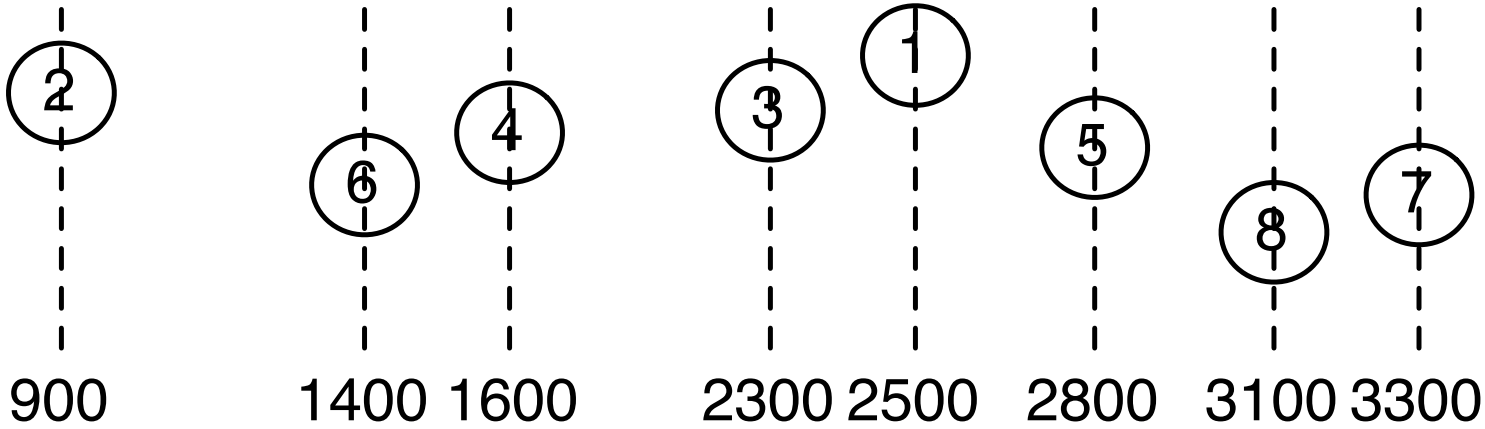
— *Attention* : Pour garantir que l'on accède toujours à la dernière version d'une donnée, les optimisations doivent conserver l'ordre des requêtes d'écriture et de lecture concernant un même secteur.

# Réduction des positionnements de cylindres

- Les stratégies doivent tenir compte des nouvelles requêtes qui sont dynamiquement déposées pendant les échanges
- Stratégie Premier Arrivé Premier Servi (FCFS = First Come First Served)
  - Avantage : équité de service
  - Inconvénient : Nombreux déplacement d'un bout à l'autre du disque
- Stratégie Plus Proche Piste (SSF = Shortest Seek Time)
  - Inconvénient : problème d'équité. L'optimisation doit éviter de ne servir qu'un sous ensemble de cylindres proches
- Stratégie de Balayage (Scan)
  - Parcourir tous les cylindres dans un sens en servant toutes les requêtes puis dans l'autre sens
  - Variantes :
    - Ne servir les requêtes que dans un sens (Circular Scan)
    - Limiter de nombre de requêtes servies pour chaque cylindre

# Réduction des positionnements de cylindres : Exemple

— Supposons que le disque doive traiter les demandes requêtes suivante :

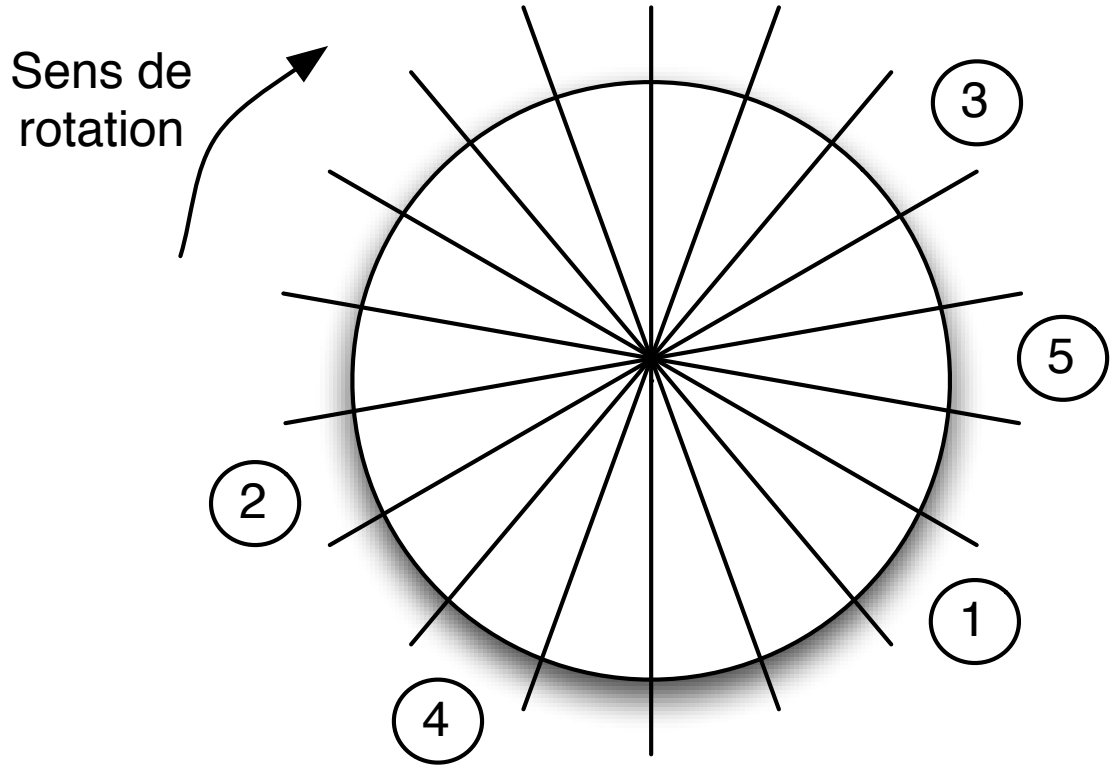



— Représenter les stratégies FCFS, SSF, Scan et Circular Scan

# Réduction des attentes de rotation

— Stratégie Premier Arrivé Premier Servi (FCFS)

— Stratégie Plus Proche Secteur (SATF = Shortest Acces Time First)



# Optimisations globales

- Pour des cylindres proches, les délais de positionnement et de rotation sont comparables
- Optimisation de la somme des temps
- STF = Shortest Time First
  - On sert d'abord la requêtes qui minimise la somme des deux délais
  - Problème d'équité
- GSTF = Grouped STF
  - On divise le disque en groupes de cylindres
  - On sert circulairement les groupes de cylindres
  - On applique STF dans chaque groupe de cylindres
- GSTF(x) où x est le nombre de cylindres par groupe

# Optimisation du temps de transfert

— Utilisation de caches
  — Mémoire cache du processeur d'entrées/sorties
  — Mémoire principale
    — Tampon système (*buffer cache*)
    — Projection de fichiers en mémoire (*memory mapped files*)

```
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot,
           int flags, int fd, off_t offset);
int munmap(void *start, size_t length);
```

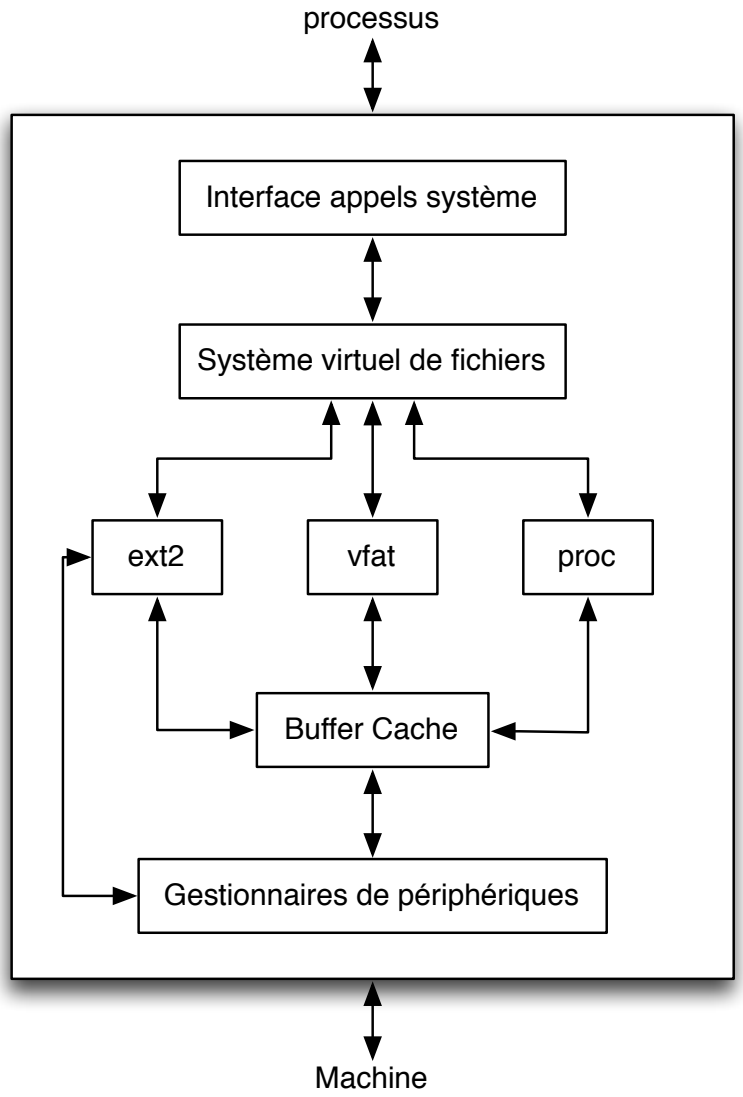
Une manière plus commode de faire des lectures/écritures dans un fichier (lire/écrire dans un fichier = lire/écrire dans un tableau).

# Allocation du disque

- Problèmes d'allocation du disque
  - Représentation de l'espace libre
    - limiter la taille des tables
  - Recherche d'un secteur libre ou d'un ensemble de secteurs libres
    - Fragmentation des fichiers : dispersion sur des cylindres éloignés
    - Fragmentation de l'espace libre : rend l'allocation difficile
- Les blocs = N secteurs successifs sur la même piste
  - Blocs petits :
    - décomposition peu efficace des accès
    - allocation trop fine
    - tables trop importantes
  - Blocs gros :
    - Fragmentation interne
- Un disque trop rempli a des performances dégradées

Implantation du système de fichiers de Linux

# Le système virtuel de fichiers



# Opération assurées par le VFS

— Les fonctionnalités assurées par le VFS sont indépendantes des systèmes de fichiers physiques.
— Le VFS implante
  — le cache des noms : association noms de fichiers / identifient (numéro de périphérique, numéro d'i-nœud)
  — le buffer cache : cache pour les entrées/sorties en mode bloque
  — les parties commune des appels systèmes
    — vérification des arguments
    — conversion des noms de fichiers
    — vérification des permissions
    — appel de la fonction du système de fichiers sous-jacent correspondant à l'appel système

# Système de fichiers virtuel

- Le VFS utilise une approche « orientée objet »
  - chaque système de fichiers monté doit implanter des opérations de bases
- Types d'opérations que doivent fournir un système de fichiers
  - Opérations sur des systèmes de fichiers
    - lecture/écriture d'un i-nœud, modification du super-bloc, etc.
  - Opérations sur les i-nœuds en cours d'utilisation
    - création/suppression de liens, répertoires, etc.
  - Opérations sur des fichiers ouverts
    - primitives d'entrées/sorties sur les fichiers, etc.
  - Opérations sur les quotas

# Buffer cache

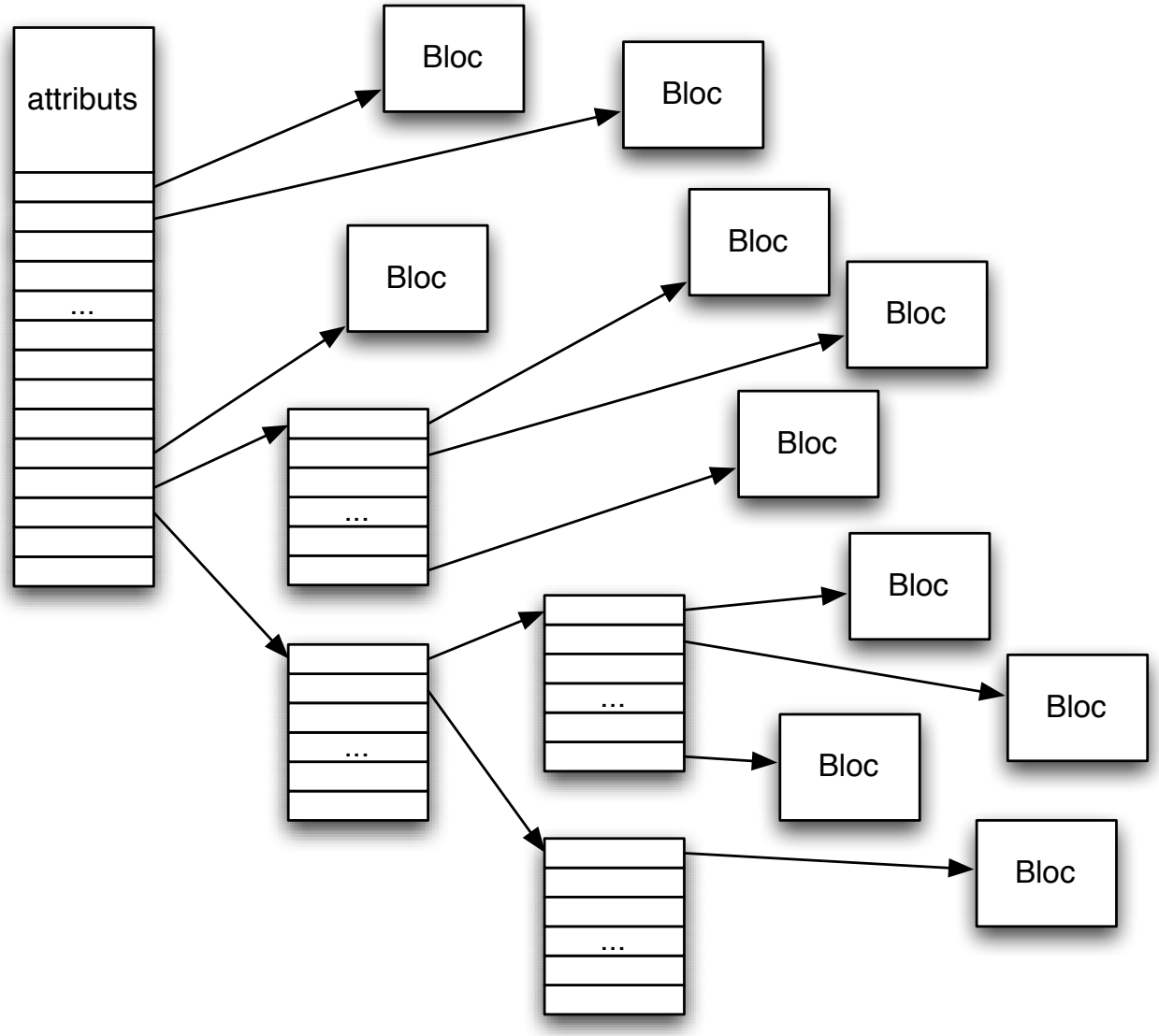
— Liste des tampons mémoire en cours d'utilisation
— Lecture d'un bloc :
  — le contenu du bloc est placé dans le tampon
— Écriture d'un bloc :
  — le changement est effectué dans le tampon
  — le tampon est marqué comme modifié
— À intervalles réguliers, le processus `update` appelle la primitive `sync` pour forcer la réécriture de tous les tampons modifiés sur les disques

# Implantation de systèmes de fichiers à i-nœuds : rappels

— Structure logique du disque


— Le bloc d'initalisation (bootstap) contient un exécutable utilisé au démarrage
— Le superbloc contient en particulier
  — nom et taille du système de fichier
  — date et mode de montage
  — la table d'allocation des blocs
    — Utilisation d'un bit par bloc
  — la table d'allocation des i-nœuds
— Le nombre d'i-nœud fixé au moment du formatage (de haut niveau)

# Rappels : Structure d'un i-nœud



# Avantages et inconvénients

- Avantages :
  - Structure arborescente des i-nœuds permet un accès quasi-immédiat à chaque bloc du fichier
  - Structure des répertoires est simple : toutes les attributs des fichiers sont directement dans les i-nœuds
  - Possibilité de créer des liens physiques
  - Table d'allocation petite
    - Permet de faire des petits blocs et donc de limiter la fragmentation interne
- Inconvénients :
  - Beaucoup de méta-données
  - Pointeurs inutiles pour les petits fichiers (compensé par la faible fragmentation interne)

# Exemple : EXT2

— Ext était le premier système de fichier à utilise le VFS
— Ext est un système de fichiers à base d'i-nœuds
— Ext2 est une extension de Ext
— Ext2 supporte tous les types de fichiers Unix
— Limitations :

| | Minix FS | Ext | Ext2 |
|---|---|---|---|
| Taille maximale | 64 Mo | 2 Go | 2 To |
| Taille maximale d'un fichier | 64 Mo | 2 Go | 2 Go |
| Taille maximale d'un nom de fichier | 14 c | 255 c | 255 c |

# Ext2 : Structure physique

— Structure à deux niveaux

— Les groupes de blocs

| | | | | |
|---|---|---|---|---|
| init | Groupe de blocs 1 | Groupe de blocs 2 | ... | Groupe de blocs N |

— Un groupe de blocs

| | | | | | |
|---|---|---|---|---|---|
| super bloc | descripteurs | bitmap blocs | bitmap i-nœuds | table des i-nœuds | blocs de données |

# Ext2 : Structure physique

- Chaque groupe de bloc contient
  - Une copie du superbloc
    - augmente la fiabilité du système de fichier
    - réduit les temps d'accès au superbloc
  - Table des descripteurs
    - Contient les adresses des blocs de bitmap et de la table des i-nœuds
  - Bitmap pour les blocs
    - Table de bits
    - Indique pour chaque bloc du groupe s'il est libre ou occupé
  - Bitmap pour les i-nœuds
    - Indique pour chaque i-nœuds du groupe s'il est alloué ou disponible
  - Table des i-nœuds
    - une partie de la table des i-nœuds du système de fichiers
  - Blocs de données

## Ext2 : Caractéristiques

- Caractéristiques de performances
  - Prélecture : lecture de plusieurs blocs contiguës à la requête de lecture
  - Allocation optimisées : allocation le plus possible dans le même groupe de blocs
  - Préallocation : allocation jusqu'à huit blocs adjacents lors de l'allocation d'un bloc
  - Lutte contre la fragmentation

# Ext2 : Caractéristiques

- Autres caractéristiques
  - Choix de la taille du bloc
  - Liens symboliques rapides
  - Mémorisation de l'état
    - Propre : il n'y a eu que des lecture
    - Sale : il y a eu au moins une écriture
  - Mémorisation du nombre de fois que le système a été monté et date de la dernière vérification d'intégrité
  - Blocs réservés au super-utilisateur
  - Effacement sécurisé : le contenu des i-nœuds sont physiquement effacés
  - Fichiers auxquels on peut seulement ajouter des données

# Détection et Réparation des incohérences du système de fichiers

— But : savoir remettre le système de fichier dans un état cohérent même en cas de problème (exemple : après une coupure de courant)

— Exemple de problème : suppression d'un fichier se fait en au moins deux étapes :

  — supprimer son entrée dans le répertoire courant

  — marquer l'i-nœud du fichier comme libre

  Que se passe-t-il si la machine s'arrête entre ces deux opérations ?

# Ext2 : File System Check (fsck)

- Comment retourner dans un système de fichiers « propre » après une erreur ?
- File System Check : Opération très couteuse !
- Vérification au niveau du fichier
  - Parcourir toute l'arborescence du système de fichier pour corriger
    - bitmap des i-nœuds
    - compte de liens dans les i-nœuds
- Vérification au niveau du bloc

# fsck : Vérification au niveau du bloc

— Pour chaque bloc `b` on construit :
  `TF[b]` = nombre de référence de `b` dans les fichiers
— On appelle `TA` la table d'allocation des blocs
— le système est cohérent si :
  (`TF[b] = 0` et `TA[b] = 0`) ou (`TF[b] = 1` et `TA[b] = 1`)
  — si `TF[b] = 0` et `TA[b] = 1` alors `b` est orphelin
  — si `TF[b]` $\geq$ `1` et `TA[b] = 0` alors il faut modifier `TA`
  — si `TF[b]` $\geq$ `2` et `TA[b] = 1` alors il faut allouer et recopier le bloc pour avoir dans chaque fichier des blocs distincts mais de contenu identiques
— Remarque : l'information contenu dans les fichiers est considérée comme plus pertinente

# Persistance des Fichiers : les systèmes à journal

— Idée : marquer les changements à faire dans un journal avant les effectuer.
  Ainsi, les changements sont
  — soit réalisés normalement
  — soit réalisés au redémarrage
  — soit pas réalisés si l'arrête arrive avant l'écriture dans le journal
— Formes physiques possibles d'un journal :
  — Fichier standard
  — Zone réservée du disque
— Types de journaux :
  — Journal complet : contient les modifications faites aux données et aux méta-données
  — Méta-journal : contient seulement les modifications faites aux méta-données

## Les systèmes à journal : Exemple

— Augmenter la taille d'un fichier peut se faire en trois étapes
  — changer l'information de taille dans l'i-nœud
  — allouer l'espace nécessaire dans la bitmap des blocs
  — ajouter les données à la fin du fichier
— Supposons un arrêt entre les étapes 2 et 3
  — Un journal complet sera détecter et corriger le problème
  — Un méta-journal laissera un fichier avec un contenu incorrect

# Ext3 ∼ Ext2 + journal

- Ext3 propose trois niveaux de journal :
  - Journal complet
  - Méta-journal
  - Journal ordonné
    - identique au méta-journal
    - mais les données sont écrites avant les méta-données
    - On peut perdre des données mais un fichier ne peut pas contenir des informations fausses
- Avantages
  - Compatibilité avec Ext2
  - Simplicité
- Inconvénients
  - Basé sur une architecture ancienne
  - Exemple : le nombre d'i-nœuds est fixé au moment du formatage

# NTFS

— Système inspiré des systèmes à base d'i-nœuds
— Système de fichiers journalisé
— Structure


  — Master Boot Record
  — Master File Table $\sim$ table des i-nœuds
— Structure d'une entrée de la MFT


— Traitement spécial des petits fichiers

Persistance des FichiersErreurs physiques

# Erreurs physiques

- Fautes possibles
  - mauvais état du disque ⇒ secteurs endommagés
  - faute de disque
  - destruction accidentelle de fichiers par des utilisateurs

- Gestion des blocs endommagés
  - Détection lors du formatage
    - Marquage des blocs défectueux
  - Détection en écriture
    - réécriture dans un nouveau bloc (perte de performance)
  - Détection en lecture
    - récupération éventuelle s'il existe une copie de sauvegarde assez récente

# Sauvegardes et Redondance

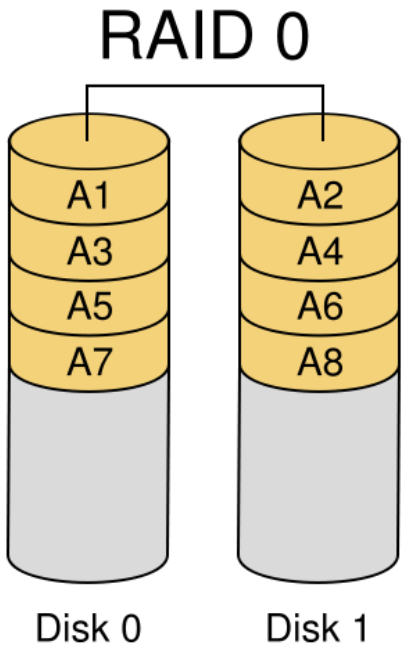
— Copies de sauvegarde (*Backup*)
  — Tous les fichiers sont copiés régulièrement sur un autre support

— Copies incrémentales
  — Le système gère des dates de sauvegarde
  — Sauvegarde des modifications

— Tableau de disques à contrôle de parité
  — RAID = Redundant Array of Inexpensive Disks
  — RAID = Redundant Array of Independent Disks

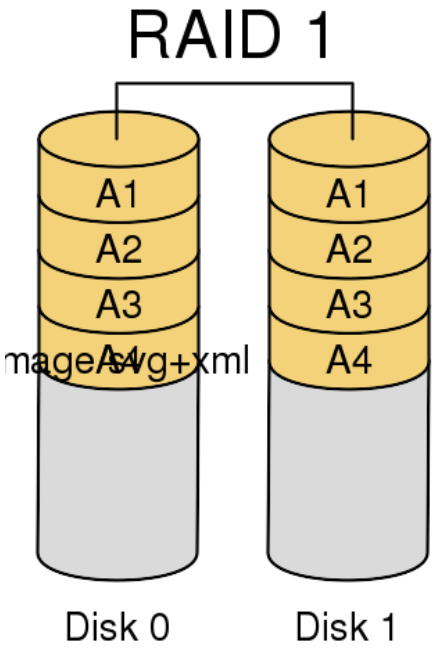
# RAID

— RAID 0 : Entrelacement (stripping)
  — Les blocs d'un fichier sont répartis sur plusieurs disques
  — Améliore les performances
  — N'augmente pas la fiabilité



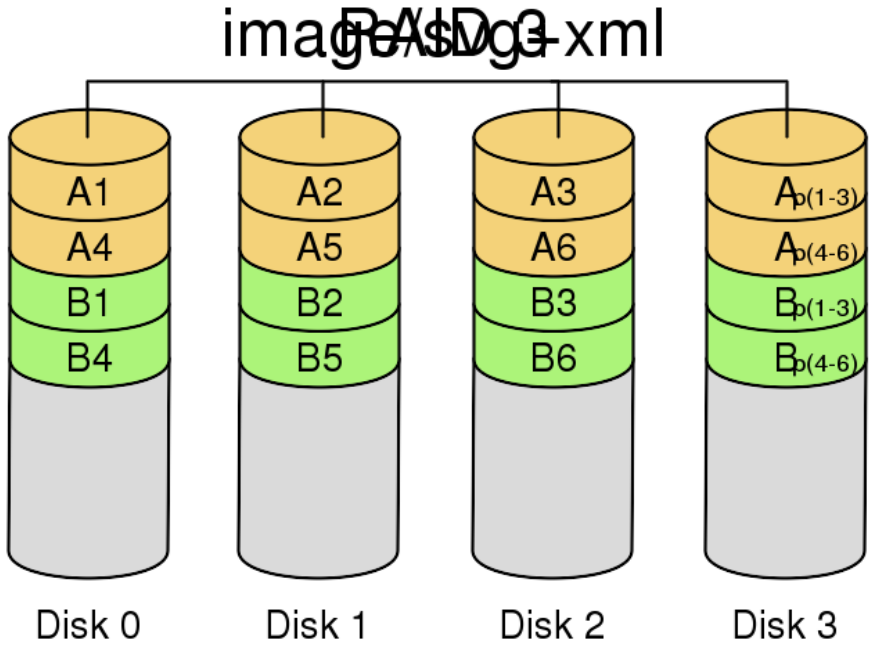
# RAID

— RAID 1 : miroir
  — Augmente la fiabilité
  — 2 fois plus de disques



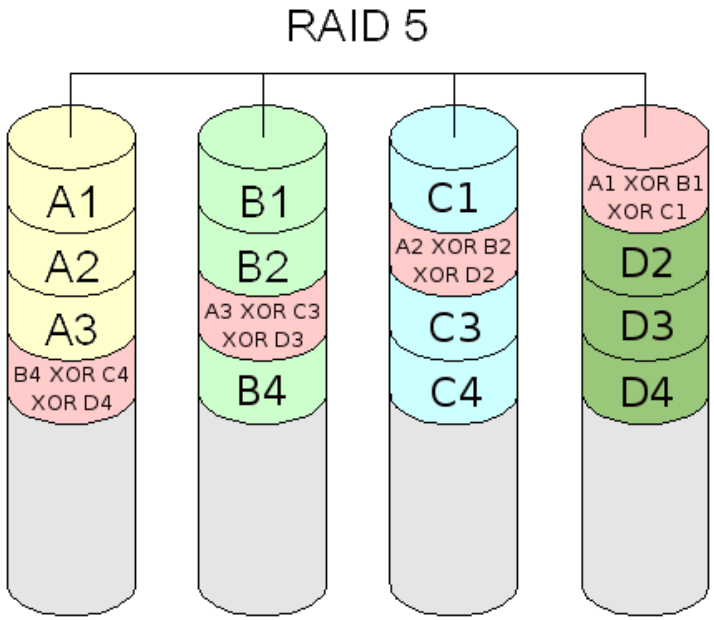
# RAID

— RAID 3 : RAID 0 + un disque de code correcteur d'erreur
  — Augmente la fiabilité à un coût inférieur (tolère une panne)
  — Surcharge du disque de parité
— RAID 4 : RAID 3 par blocs



# RAID

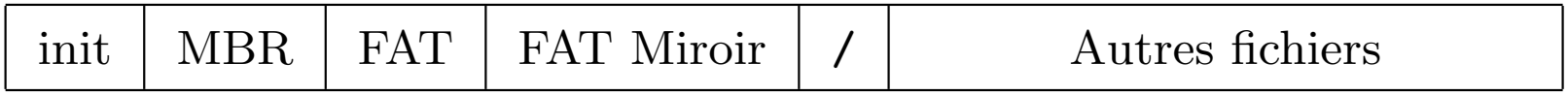
— RAID 5 : RAID 4 mais avec code correcteur réparti sur tous les disques
  — améliore les performances
  — meilleure disponibilité en cas de panne
— RAID 6 : RAID 5 avec plus de redondance (tolère 2 pannes)




Autres types de systèmes de fichiers

# Le système FAT (*File Allocation Table*) : MS-DOS

— Système de fichier qui *n'est pas* à base d'i-nœud
— Structure logique du disque

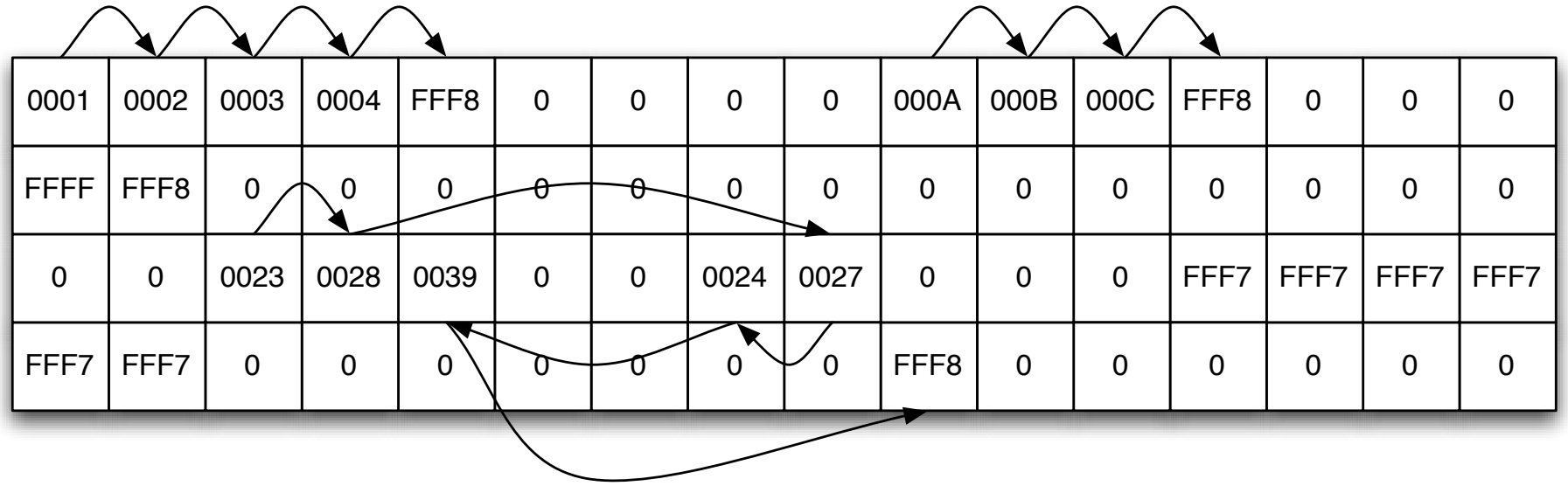



  — MBR (*Master Boot Record*)
    — nombre d'octets par secteurs, nombre de secteurs par cluster, nombre de secteurs sur le disque, etc.
  — Table d'allocation des clusters (FAT)
    — chaque case représente un cluster (clusters = bloc mais généralement plus gros : jusqu'à 128 Ko)
  — Le répertoire racine
  — Les clusters de données

# Table d'allocation des clusters

- Chaque case peut contenir une valeur codée sur 16 ou 32 bits
- La signification de ces valeurs est :
  - `0` : cluster vide
  - `1` : cluster réservé au système
  - `0xFFF7` : cluster défectueux
  - Une autre valeur `n` :
    - le cluster est occupé par un fichier
    - la suite du fichier est contenu dans le cluster `n`
    - si `n` est compris entre `0xFFF8` et `0xFFFF`, c'est le dernier cluster du fichier
- Un fichier est une liste chaînée de clusters

# Table d'allocation des clusters : exemple



## Structure d'un répertoire

— Un répertoire est une liste d'entrées contenant :
  — Le nom du fichier (8 + 3 caractères)
  — Les attributs (lecture seule, fichier caché, sous-répertoire, etc.)
  — Les dates (création, accès, modification)
  — Numéro du premier cluster
  — Taille du fichier

— Contrairement aux systèmes de fichiers avec i-nœuds
  — Les meta-données sur les fichiers sont stockées au niveau des répertoires
  — Il ne peut pas y avoir de liens physiques

# FAT : Avantages et inconvénients

- Avantages :
  - Simple et léger
  - Bien adapté à des petits disques ($\leq$ 1 Go)
- Inconvénients :
  - Limitations fonctionnelles : nom de fichier, propriétaires, etc.
  - Mauvaise adaptations aux opérations courantes sur les fichiers : accès aléatoire, écriture en fin de fichier (problème accentué sur les gros fichiers)
  - Limitation sur la taille des disques (table d'allocation)
    - Remarque : dans les systèmes à i-nœuds il suffit d'un bit par bloc
  - Fragmentation interne (dernier cluster sous-utilisé)

# procfs (Linux)

— Informations gérées par le noyau
— Fichiers virtuels (pas des données stockées sur disque mais en mémoire centrale)
— Exemples de fichiers de `/proc`
  — `cpuinfo` : description du ou des processeurs
  — `loadavg` : charge du système
  — `meminfo` : état d'occupation de la mémoire
  — `uptime` : temps écoulé depuis le démarrage du système
  — etc.
— En plus de ces fichiers, `/proc` contient des répertoires comme :
  — `net` : informations sur le réseau
  — `scsi` : informations sur les gestionnaires de périphériques SCSI
  — un répertoire par processus existant dans le système
    — `cmdline`, `cwd`, `environ`, `fd`, etc.