

# Cours 8 :

## Système de gestion de fichiers

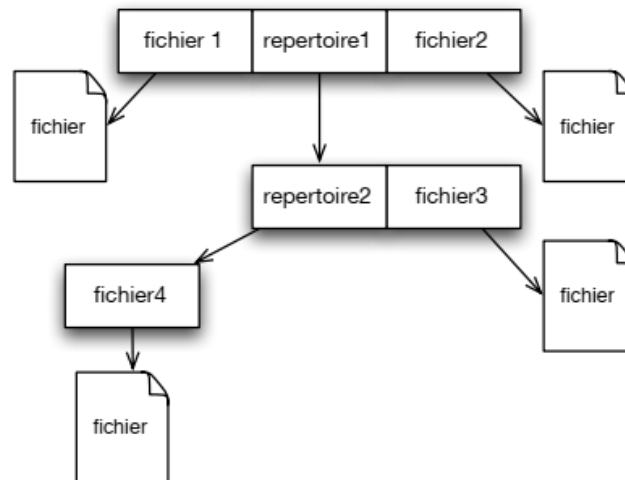
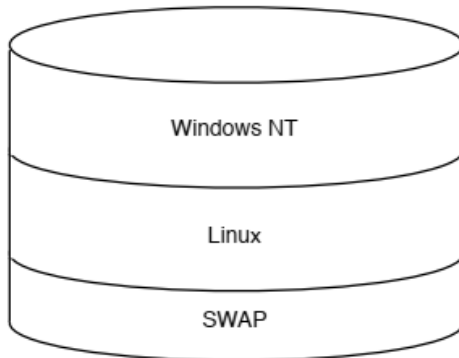
# Fichiers et répertoire

- Fichier = ensemble de blocs sur disque

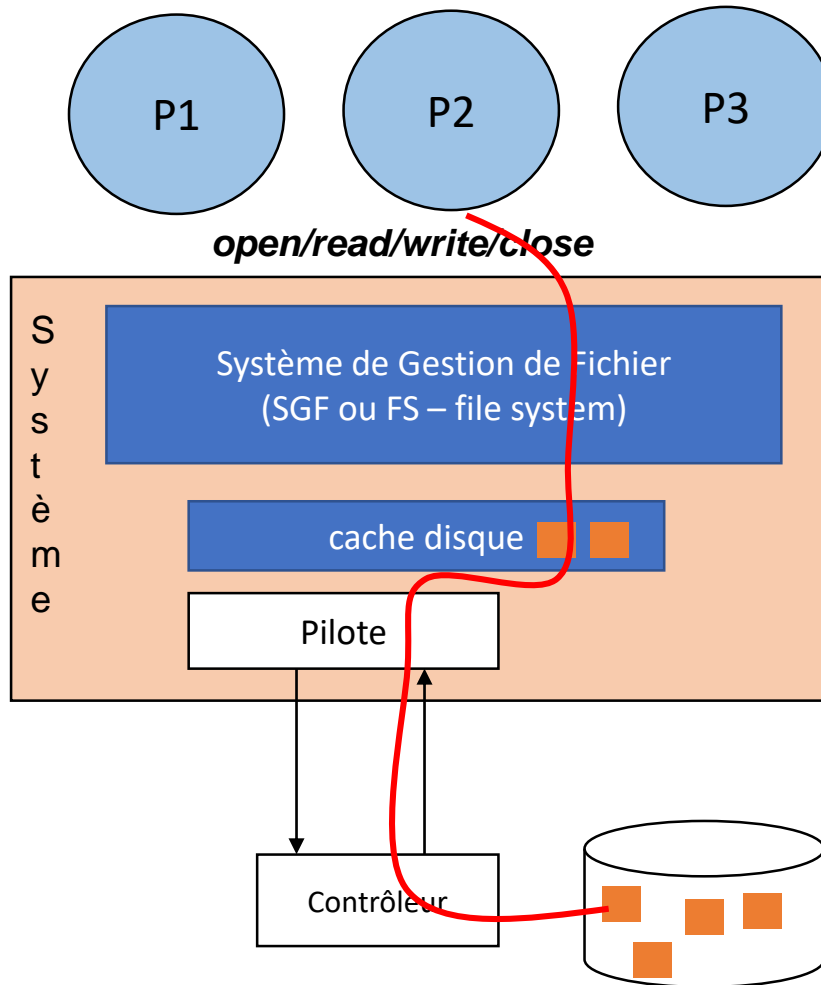


- Répertoire = ensemble de fichiers

- Les fichiers et répertoire sont regroupés dans des partitions



# Système de gestion de fichiers (SGF)



fichier vu comme un ensemble d'octets contigu  
*nom de fichier + déplacement*

fichier = 1 ensemble de blocs

blocs = cylindre/secteurs

# Fichier

---

- 2 types d'informations stockées sur disque

## 1. Méta-données :

- Attributs
  - Nom du fichiers
  - Propriétaire + groupe
  - Droits (Lecture/Ecriture/eXécution)
  - Taille
  - Type (Fichier normal, répertoire ...)

- Listes des numéros blocs

## 2. Données

- Blocs de données

# Appels systèmes : 4 appels de base

---

- `int open(char *name, int flags, [mode_t mode]);`

Ouvre le fichier `name` en mode `flags` (`O_RDONLY`, `O_WRONLY`, `O_RDWR`).

Si `flags` inclut `O_CREAT`, crée le fichier avec les droits `mode`

Retourne un descripteur de fichier (`fd` = file descriptor)

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

Lit `count` octets dans le fichier ouvert `fd`, copie dans `buf` (alloué)

Retourne le nombre d'octets lus

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

Ecrit `count` octets contenu dans `buf` dans le fichier ouvert `fd`

Retourne le nombre d'octets écrits

- `int close(int fd);`

Ferme le fichier `fd`

Fonctions retournent -1 en cas d'erreur

\$ man 2 open/read/write/close pour les "include" + options

# Exemple : affichage contenu fichier

---

...

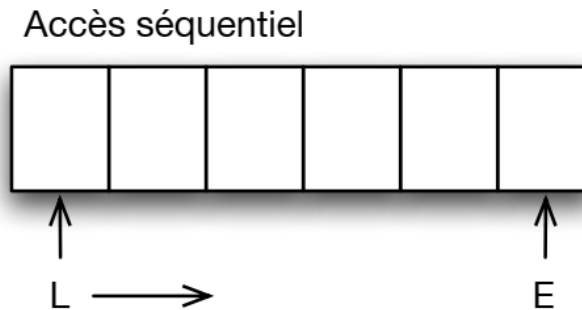
```
char buf[NBUF+1];
if ((fd = open("monfichier", O_RDONLY)) == -1) {
    perror("pb open");
    return 1;
}
while ( (nb = read(fd, buf, NBUF)) > 0) {
    buf[nb]=0;
    printf("%s", buf);
}
close(fd);
printf("\n");
return 0;
}
```

# Type d'accès aux fichiers

---

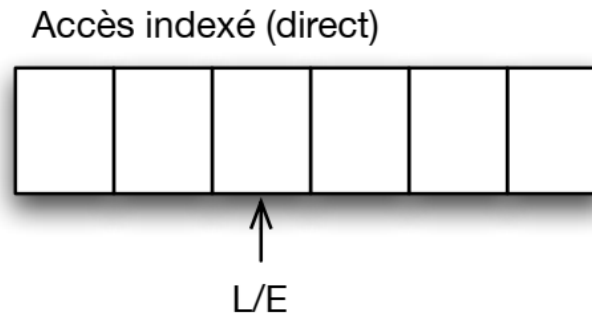
## Accès séquentiel

- Lecture d'un flux de données
- Ecriture en fin de fichier



## Accès indexé (direct)

- Accès direct aux blocs



# Rôles du SGF

---

## 1. Accès aux données des fichiers

- Contrôle d'accès
- Trouver les blocs qui composent le fichier à partir du nom
- Allocation des blocs

## 2. Gestion de l'espace libre

# Fichier : Allocation des blocs de données

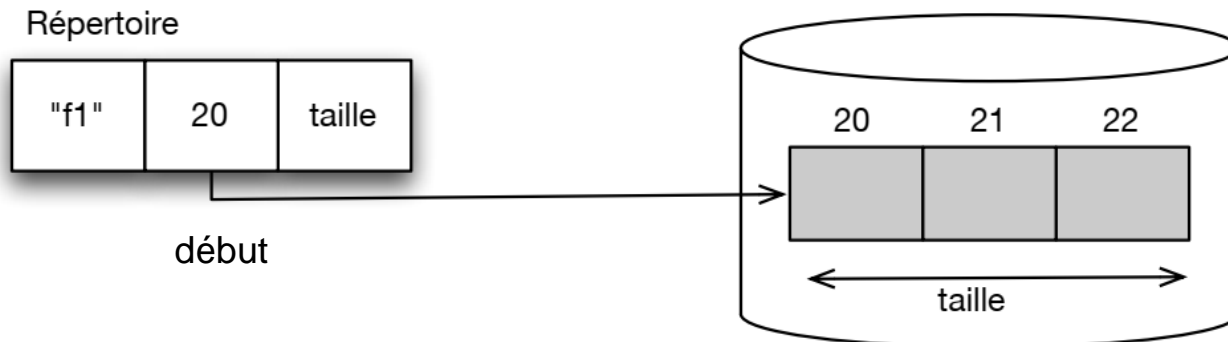
---

## 3 stratégies

1. Contiguë
2. Chaînée : Simple ou FAT
3. Indexée : simple / multi-niveau

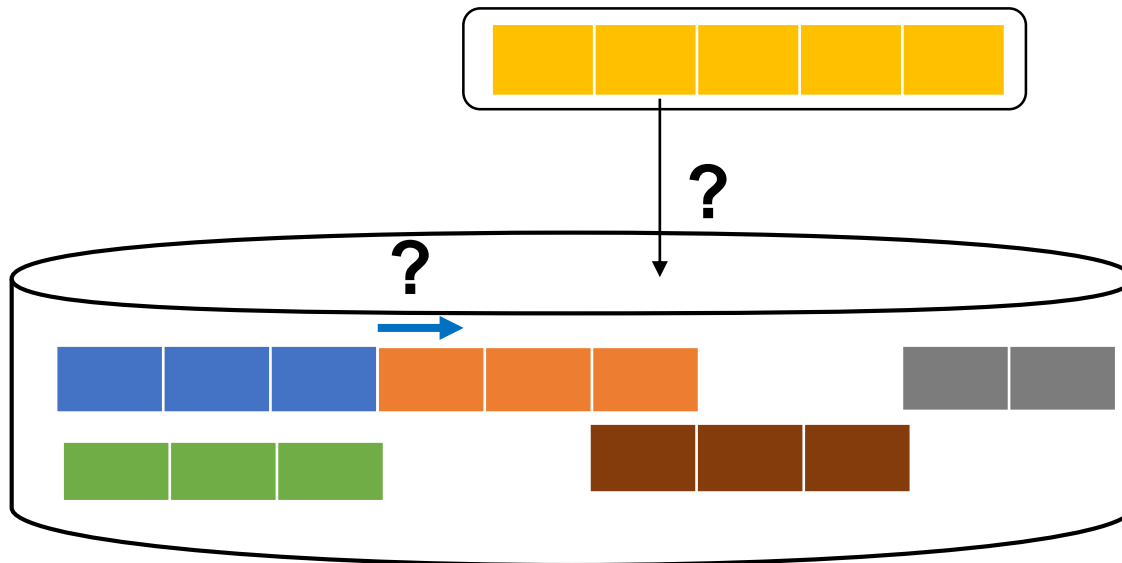
# Allocation contiguë

- Fichier = suite de blocs **consécutifs** sur le disque
- Accès aux données :
  - A partir du répertoire => accès aux méta-données
- Exemple : fichier "f1" composé de 3 blocs



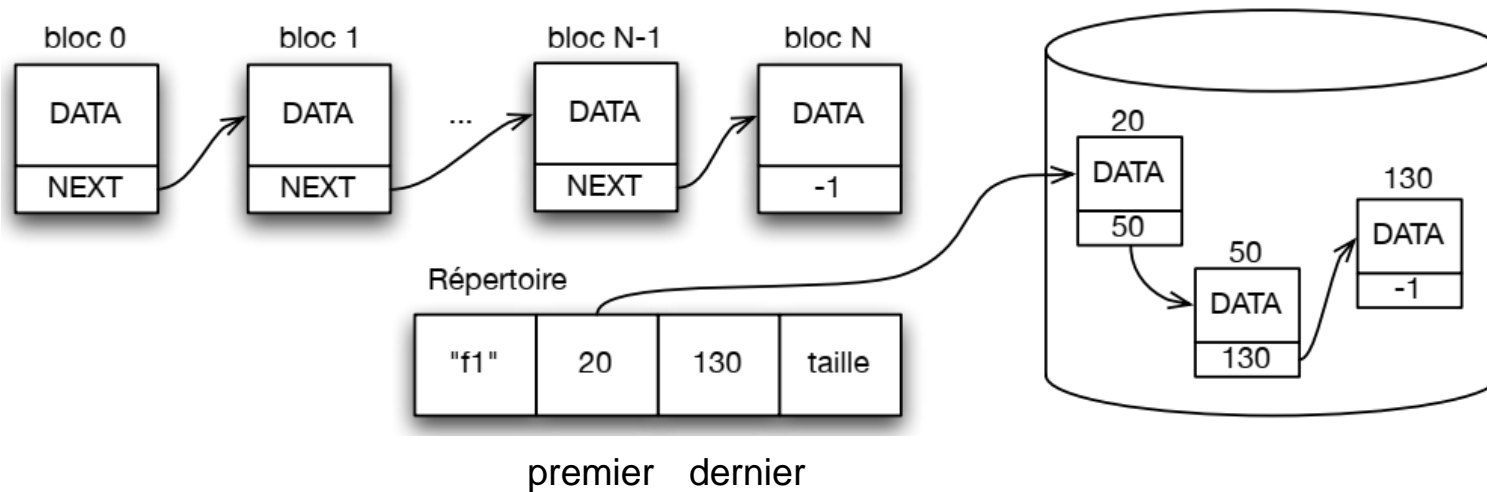
# Allocation contiguë (2)

- ⊕ Efficace en termes de déplacement de tête :
  - Le fichier est réparti sur peu de pistes
  - Accès séquentiel et direct simple à implémenter
- ⊖ Gestion complexe des tailles dynamiques de fichier
  - Fragmentation (nécessité de réorganisation régulière)



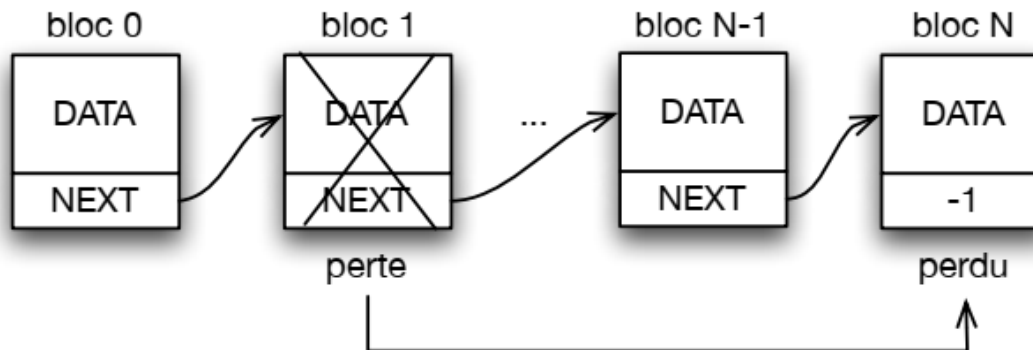
# Allocation chaînée

- Fichier = suite de blocs **dispersés** sur disque, chaînés entre eux



# Allocation chaînée

- + Allocation simple
  - Changement de taille
  - Accès séquentiel
- Pas d'accès direct
  - Fiabilité : perte d'un bloc  $\Rightarrow$  Perte de la fin du fichier



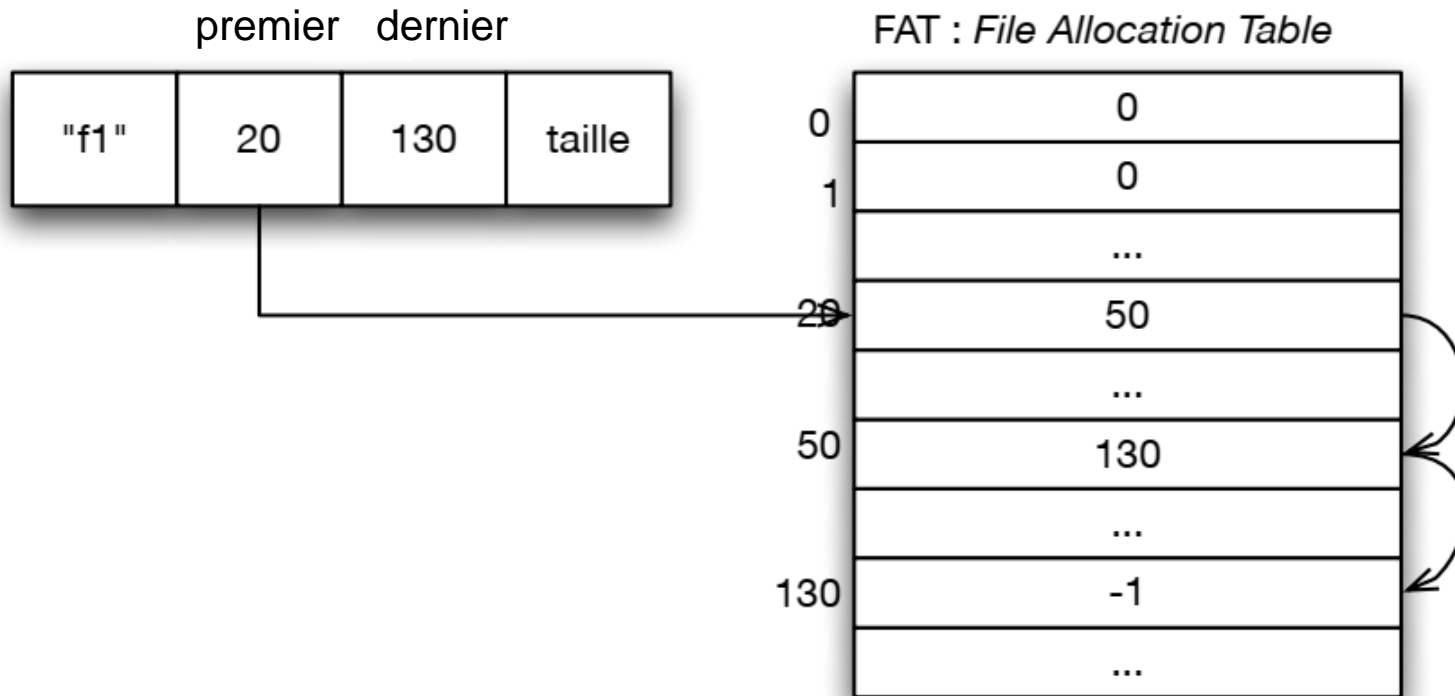
# FAT : File Allocation Table

---

- La table d'allocation de fichier (FAT) est une variante de l'allocation chaînée
- Fichier = liste de blocs sur disque
- FAT = 1 table au début de la partition qui stocke les liens (chaînage) avec une entrée par bloc
- Entrée de la FAT :
  - 0 : bloc libre
  - 1 (EOF) : dernier bloc
  - > 0 : numéro de bloc suivant (chaînage)

# FAT : exemple

---



# FAT

---

- ⊕ Accès direct possible (lecture d'entrées de la FAT)

Fiabilité : réplication de la FAT

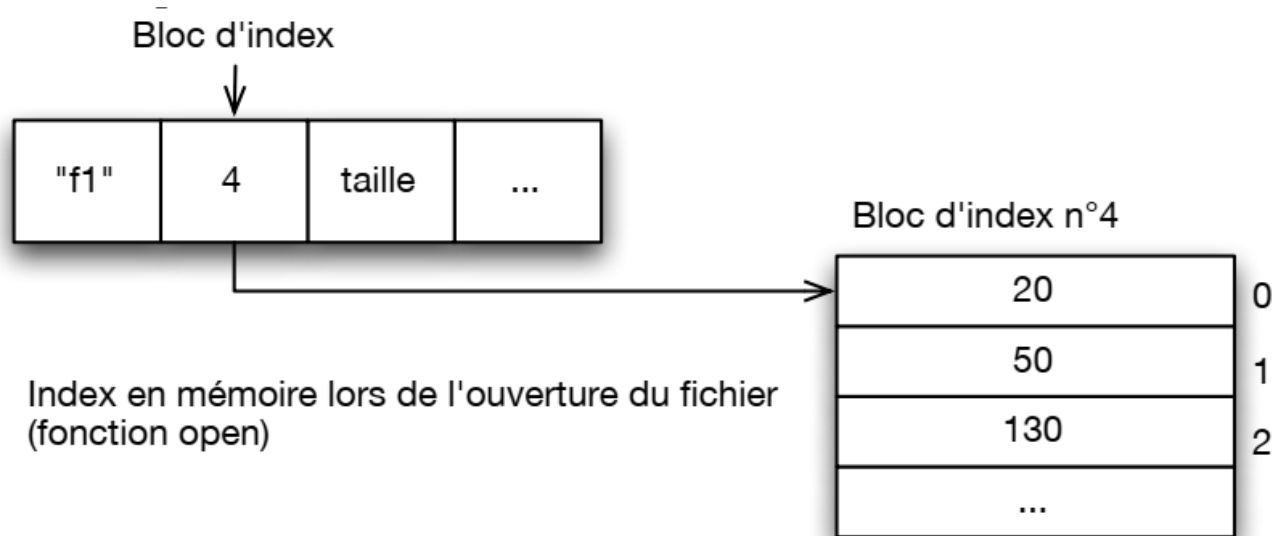
- ⊖ Accès direct peu performant

Déplacement de la tête de lecture : accès à la FAT en début de partition

Taille de la FAT (ex : partition 20 Go, bloc 512 octets  $\Rightarrow$  FAT à 20 Mo entrées  $\Rightarrow$  80 Mo pour la FAT)

# Allocation indexée

- **Blocs d'index** par fichier indique la liste des blocs composant le fichier
- Index simple : 1 seul bloc d'index par fichier



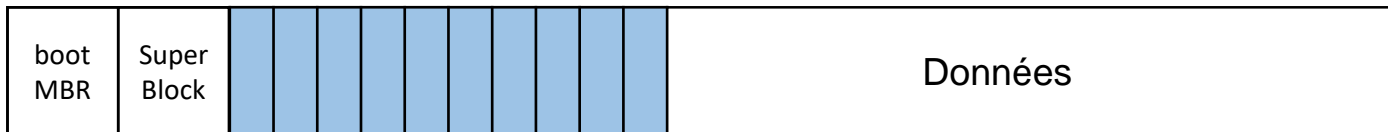
⊕ Accès séquentiel/direct

⊖ Perte d'un bloc par fichier  
⇒ Important car beaucoup de petits fichiers  
Taille limitée à la capacité du bloc d'index

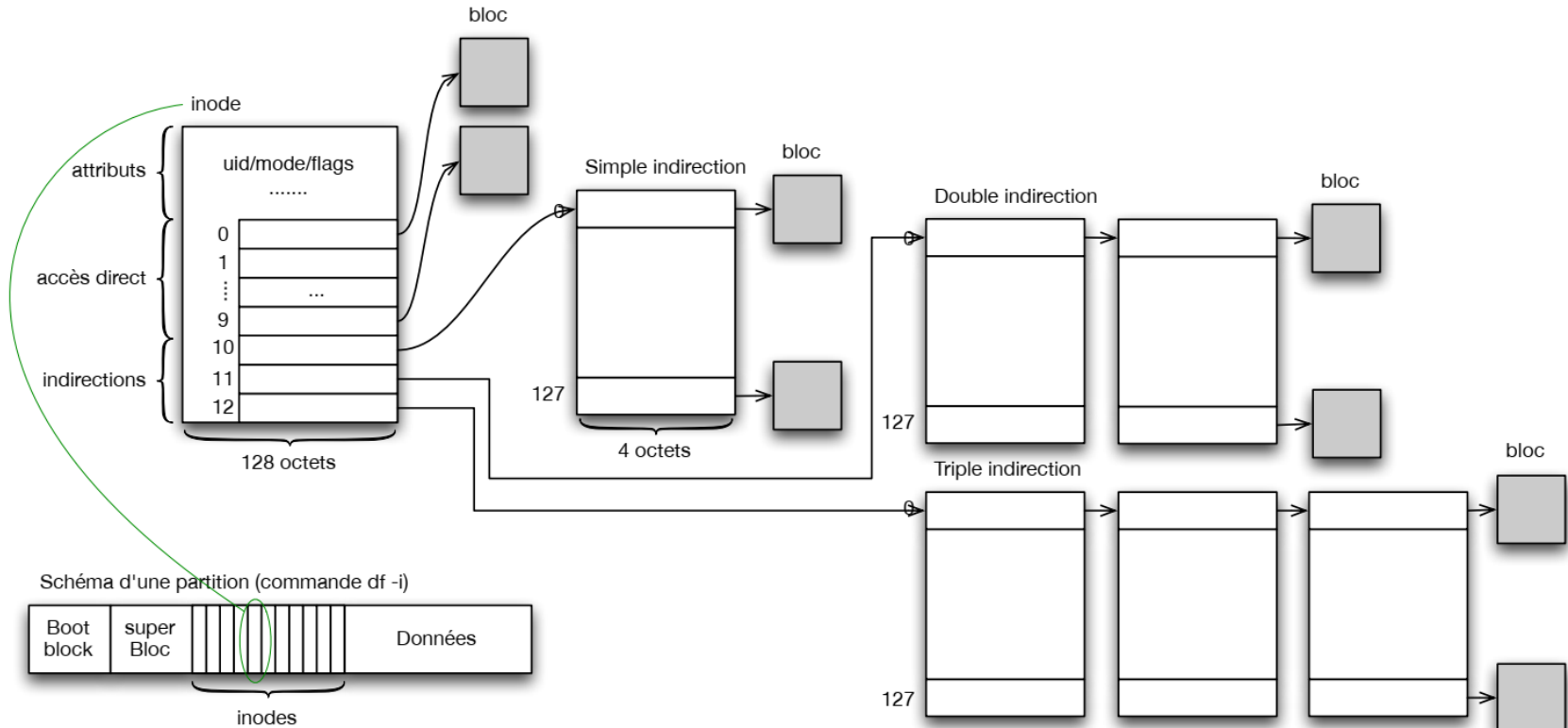
# Indexation multi-niveau (Unix)

- 1 index de taille petite est associé à chaque fichier  
inode - index node (ex: 128 octets dans ext3fs)
- 1 inode stocke les méta-données du fichiers :
  - uid / gid
  - droits
  - dates
  - **index** avec un nombre petit d'entrées (13 à 15)
  - Les 10 premières entrées pointent vers des **blocs directs**
    - numéro de blocs contenant des données du fichier
  - 3 entrées suivantes des blocs indirections :
    - Indirection simple / double / triple
- Table de inodes stockée au début de la partition

Table des inode (df -i)



# Indexation multi-niveau : Inode



# Index multi-niveau - Inode

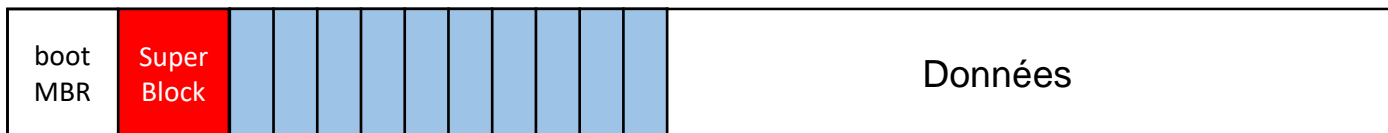
---

- ⊕ Accès rapide aux fichiers de petite taille  
Table des inodes de taille réduite  
Capacité à gérer des fichiers de grande taille
- ⊖ Implémentation complexe  
Accès non uniforme aux blocs

# Gestion de l'espace libre

---

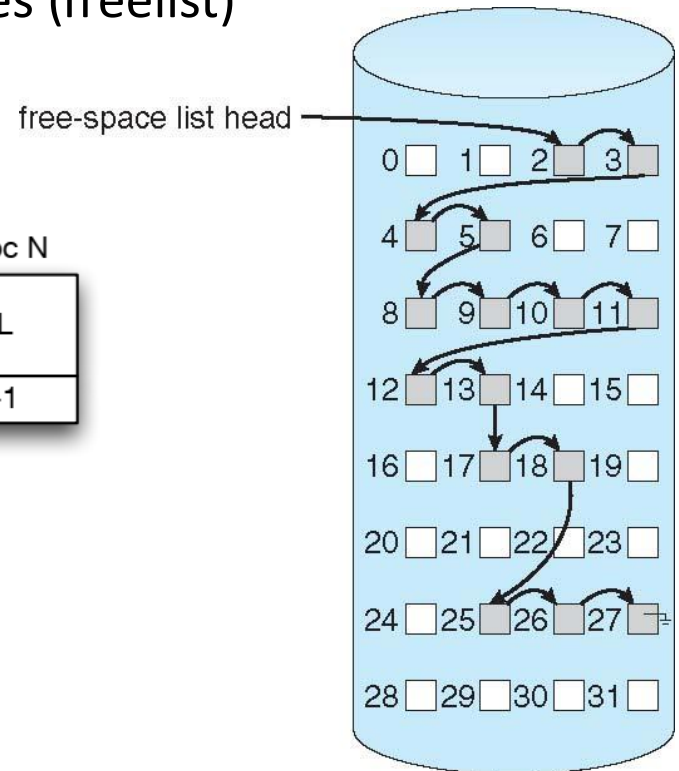
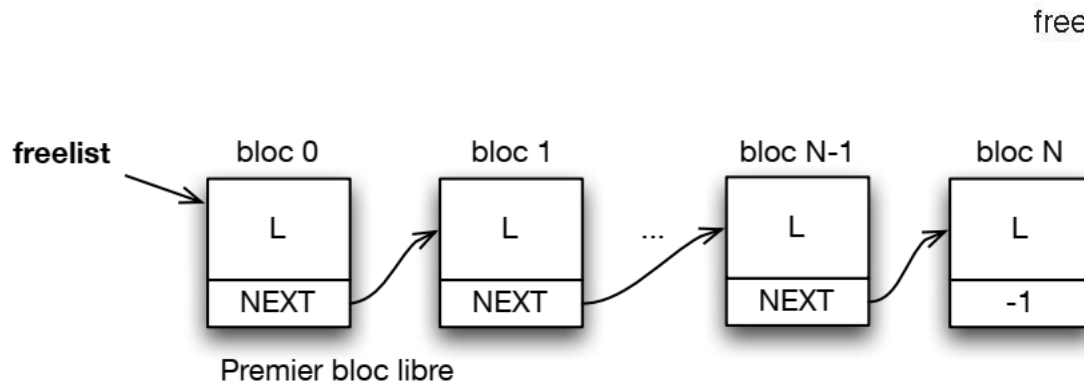
- Le système maintient l'ensemble des blocs libres sur disque
- 2 opérations
  - **Allocation de blocs** (eg. écriture de nouvelles données) : retirer des blocs dans l'ensemble des blocs libres
  - **Libérer des blocs** (eg. destruction d'un fichier) : insérer des nouveaux blocs libres dans l'ensemble des blocs libres
- Les informations permettant de connaître l'ensemble des blocs libres sont stockées sur disque




⇒ différentes organisations

# Liste chaînée

- ensemble des blocs libres = une liste chaînées (freelist)



 Simple à implémenter

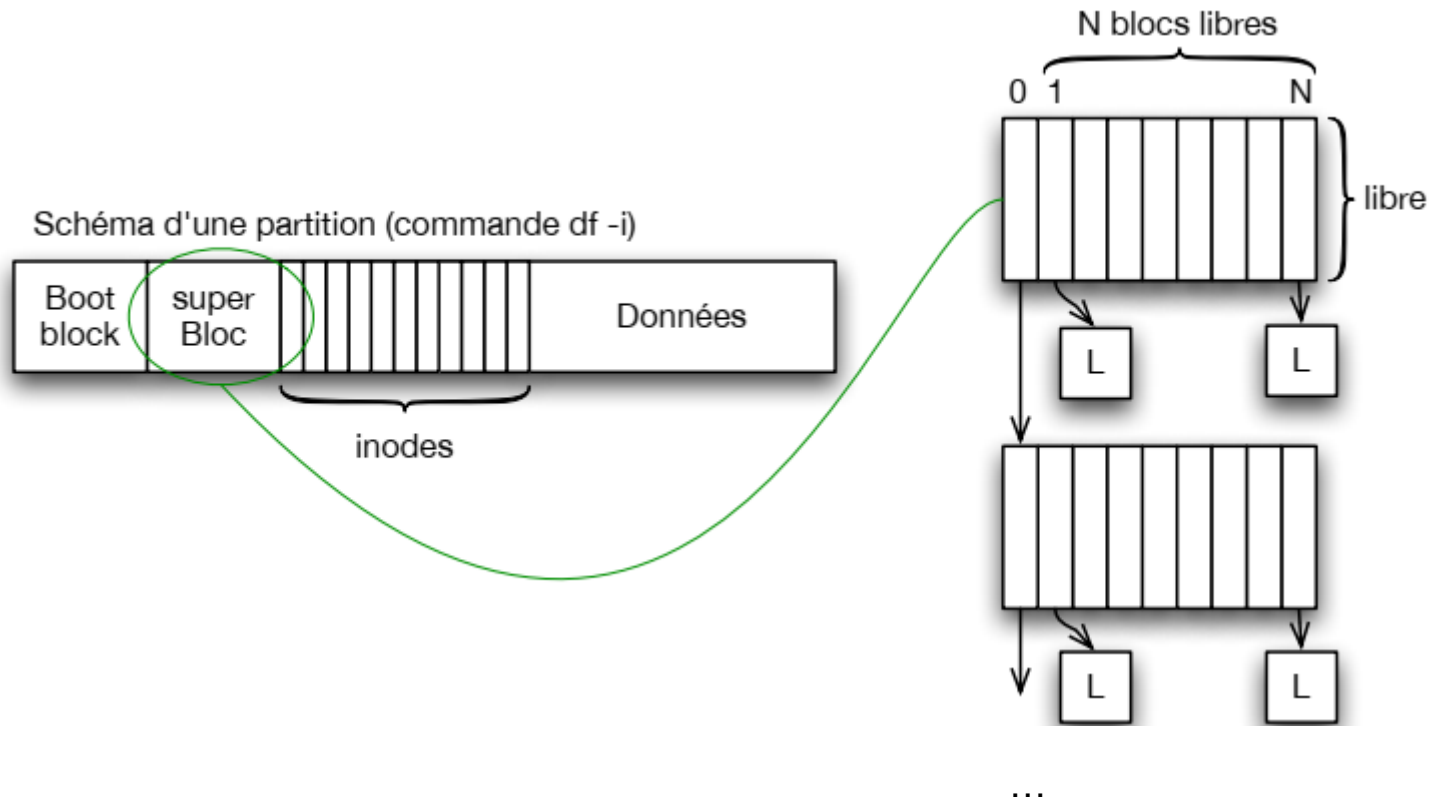
# Fragile

## Peu performant pour des demandes groupées

## Fragmentation des fichiers

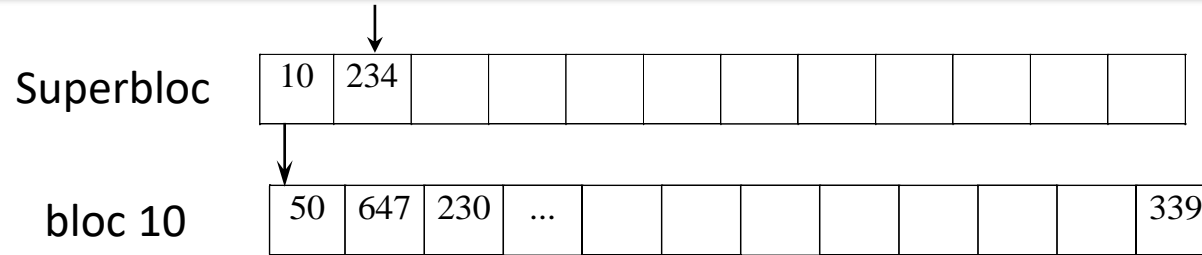
# Regroupement

- Le super bloc (bloc 1) contient les numéros de N-1 autres blocs libres + le numéro d'un bloc libre avec la même structure

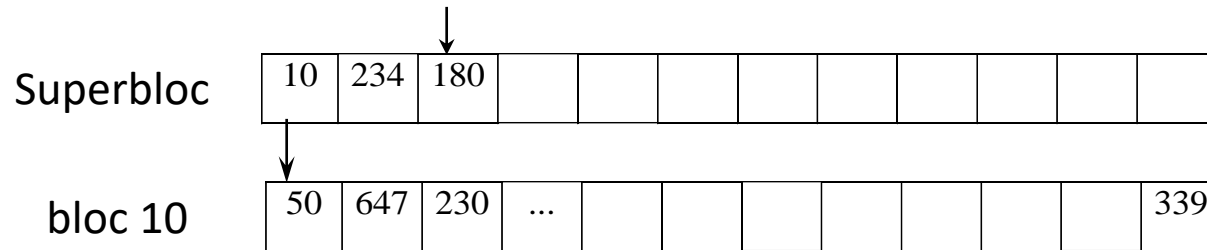


# Regroupement : exemple

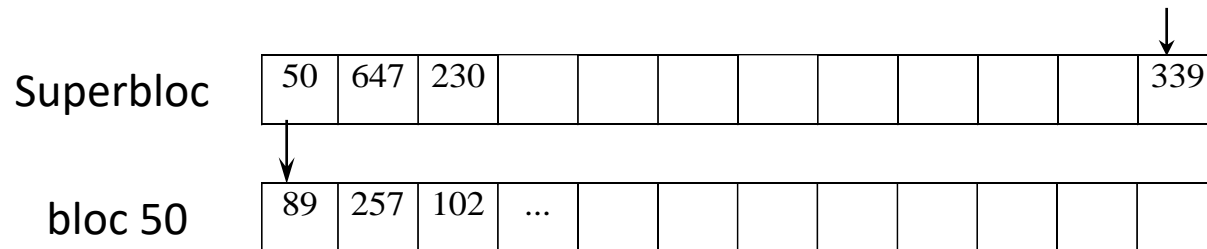
## Configuration initiale



## Libération du bloc 180



## Allocation de 3 blocs (180, 234, 10)



Contenu du bloc 10 copié dans superbloc

# Regroupement : avantages/inconvénients

---



Fournir rapidement beaucoup de blocs libres

Faible occupation sur disque



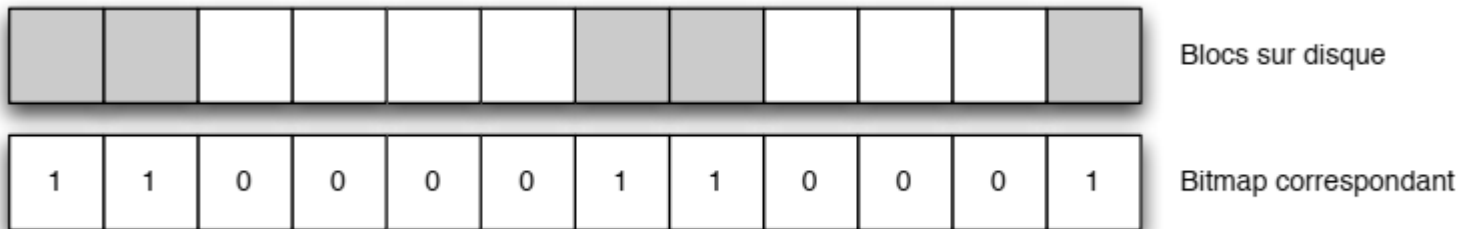
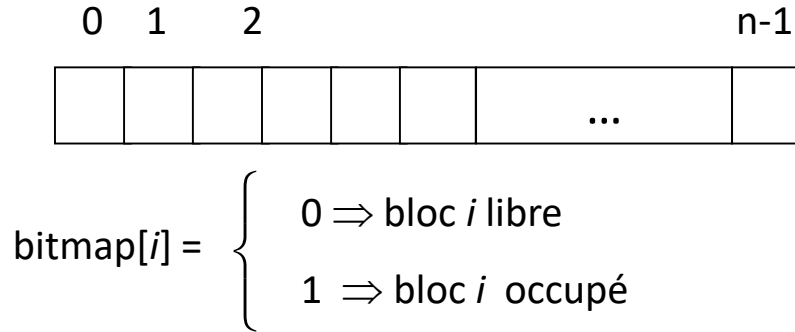
Difficile de fournir des blocs libres proches sur le disque

⇒ Fragmentation importante

# Vecteur binaire

- 1 vecteur avec 1 entrée par bloc (stockée en début de partition après le super bloc) :
 

0	1	2	...	n-1
---	---	---	-----	-----



Allocation : Trouver le 1<sup>er</sup> bit à 0 dans bitmap et le mettre à 1

Libération bloc x :  $\text{bitmap}[x] = 0$

# Vecteur binaire : avantages / inconvénients

---

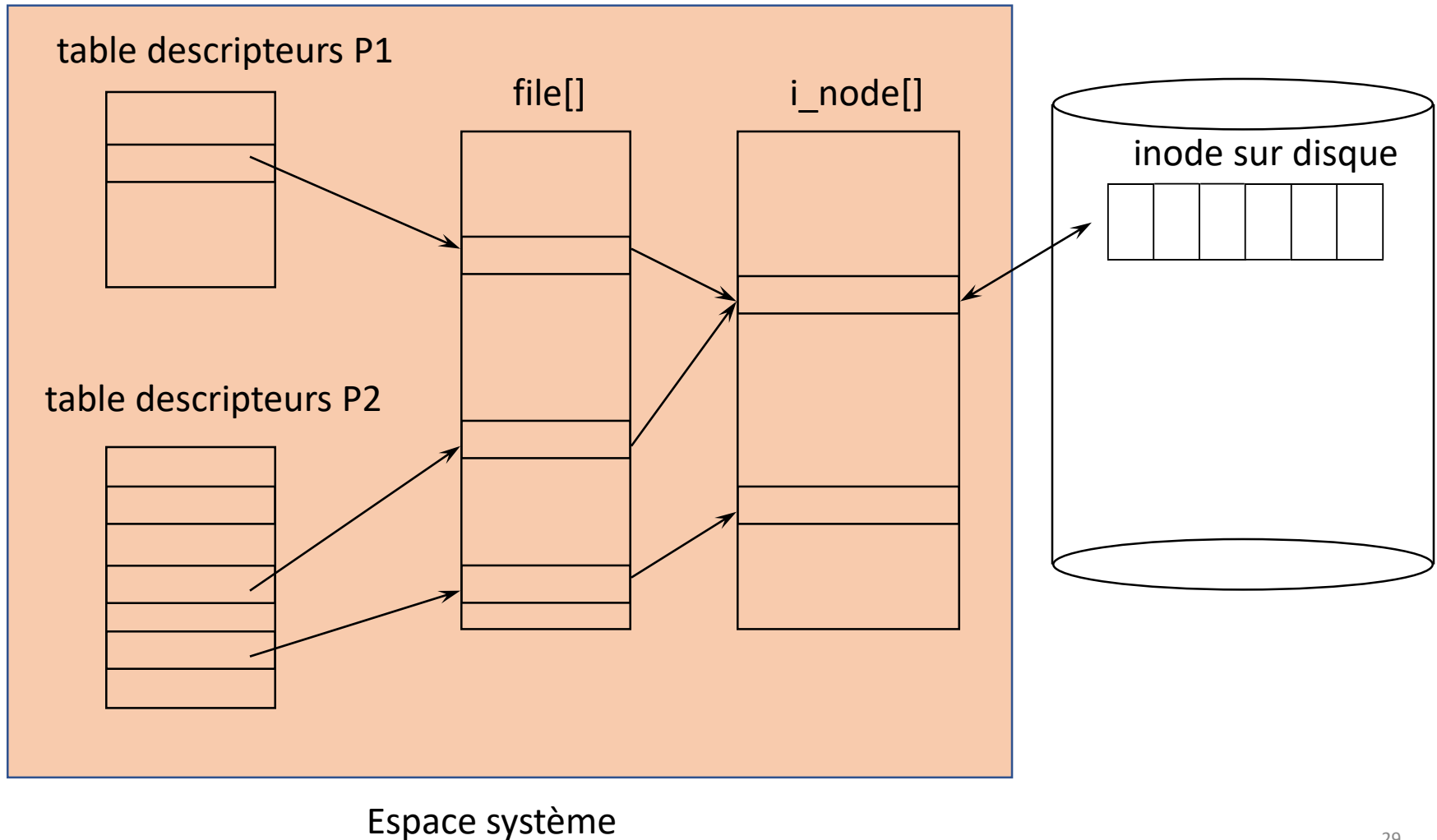
- ⊕ Limite la fragmentation : possibilité d'allouer des blocs contigus pour un même fichier
- ⊖ Taille : importante pour les grosses partitions.

# Structure en mémoire pour l'accès aux fichiers (Unix)

---

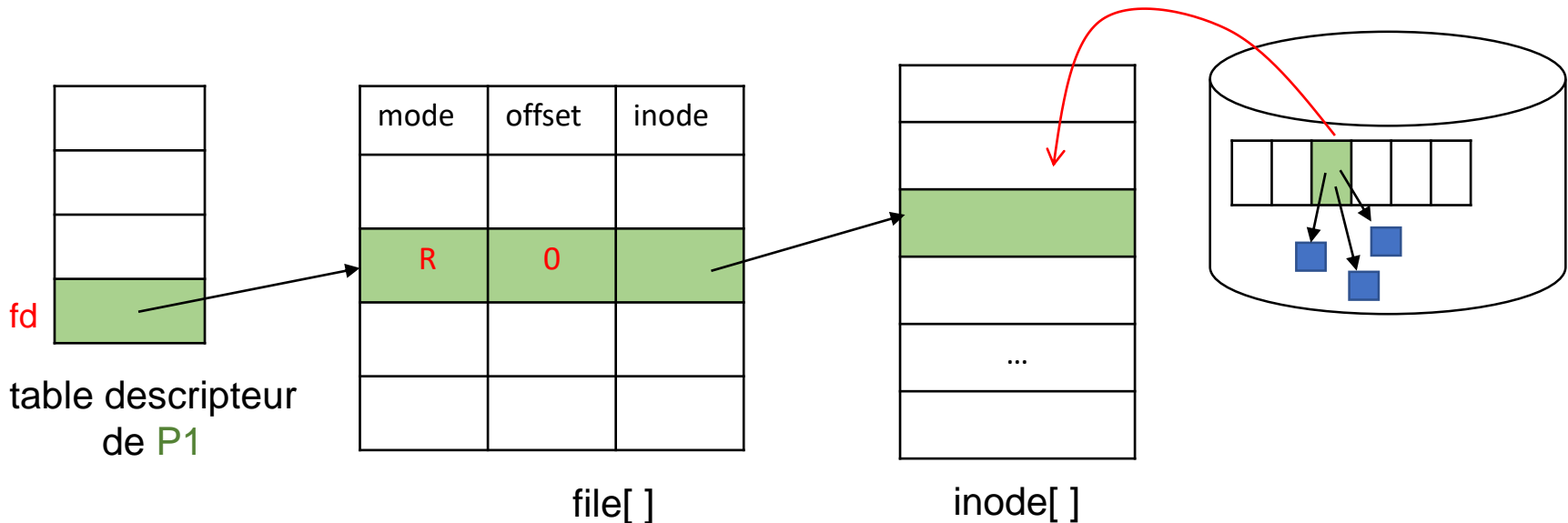
- Sur disque : table des inodes + freelist
- En mémoire dans l'espace du système :
  - 1 table globale des inodes en cours d'utilisation = **inode**[]
  - 1 table globale des modes d'utilisation (mode ouverture + déplacement - offset) = **file**[]
  - 1 table par processus : l'ensemble des fichiers ouverts par le processus

# Résumé des structures



# Opérations sur fichier : open

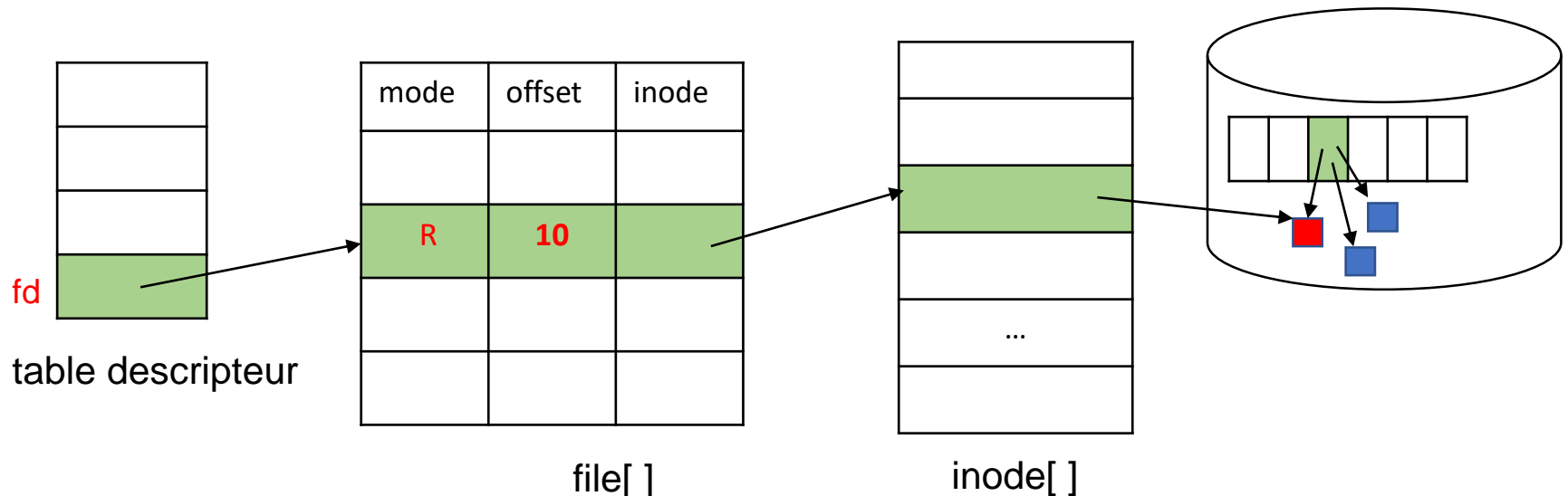
- Trouver l'inode sur disque (lecture des répertoires) + vérification droits
- Copier l'inode dans 1 entrée dans inode[] en mémoire
- Initialiser 1 entrée dans file[]
- Initialiser 1 entrée dans la table des descripteurs du processus



```
P1    fd = open("f1", O_RDONLY);
```

# Opérations sur fichier : read/write

- Accès à l'entrée file via table des descripteurs + vérification droits
- Accès à l'inode en mémoire
- Accès aux blocs de données
- Mettre à jour offset

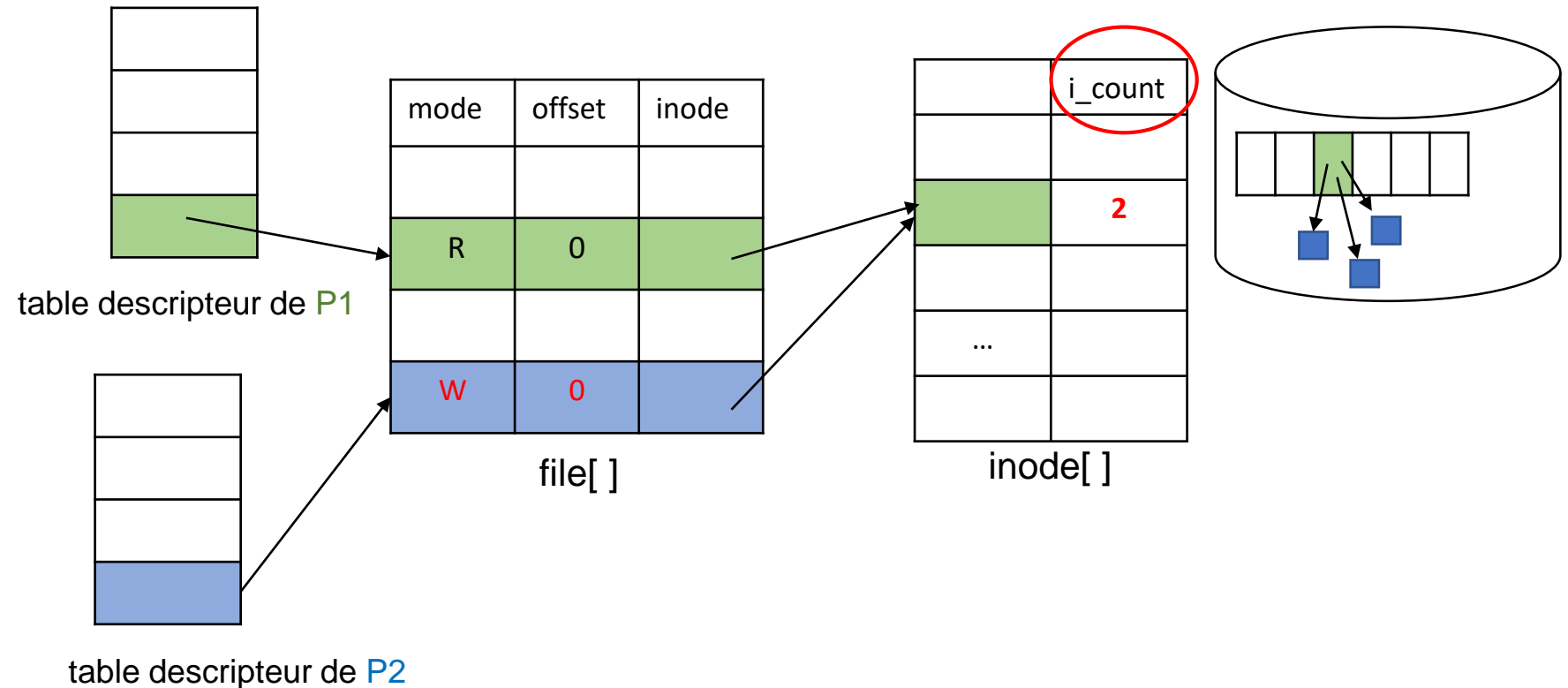


```
P1  read(fd, buf, 10) ;
```

# Fichiers partagés

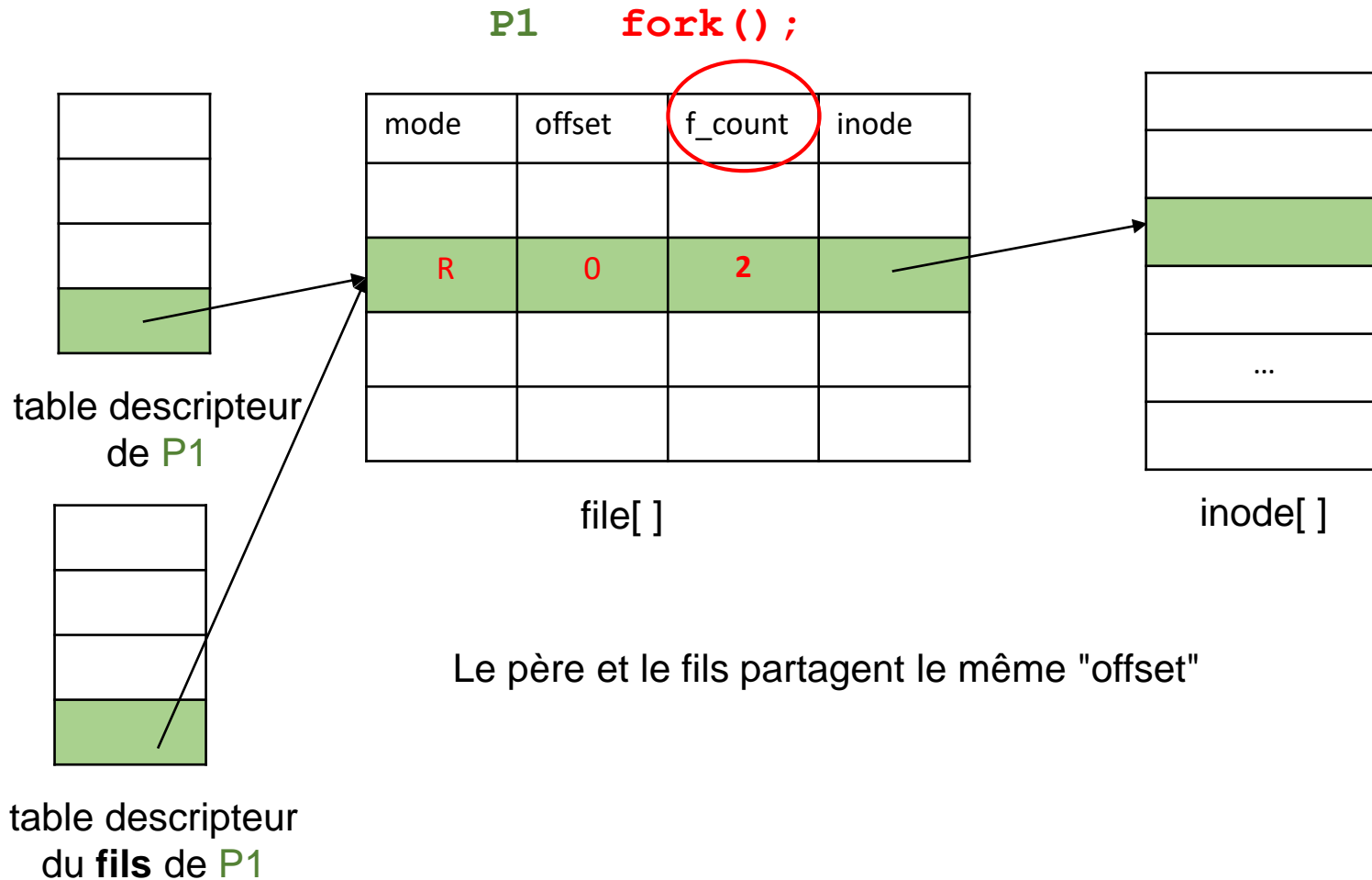
1. Plusieurs processus peuvent ouvrir un même fichier
2. Un fichier ouvert par un processus est partagé par ses futurs fils

Cas 1 :     **P2**                    **fd = open("f1", O\_WRONLY) ;**



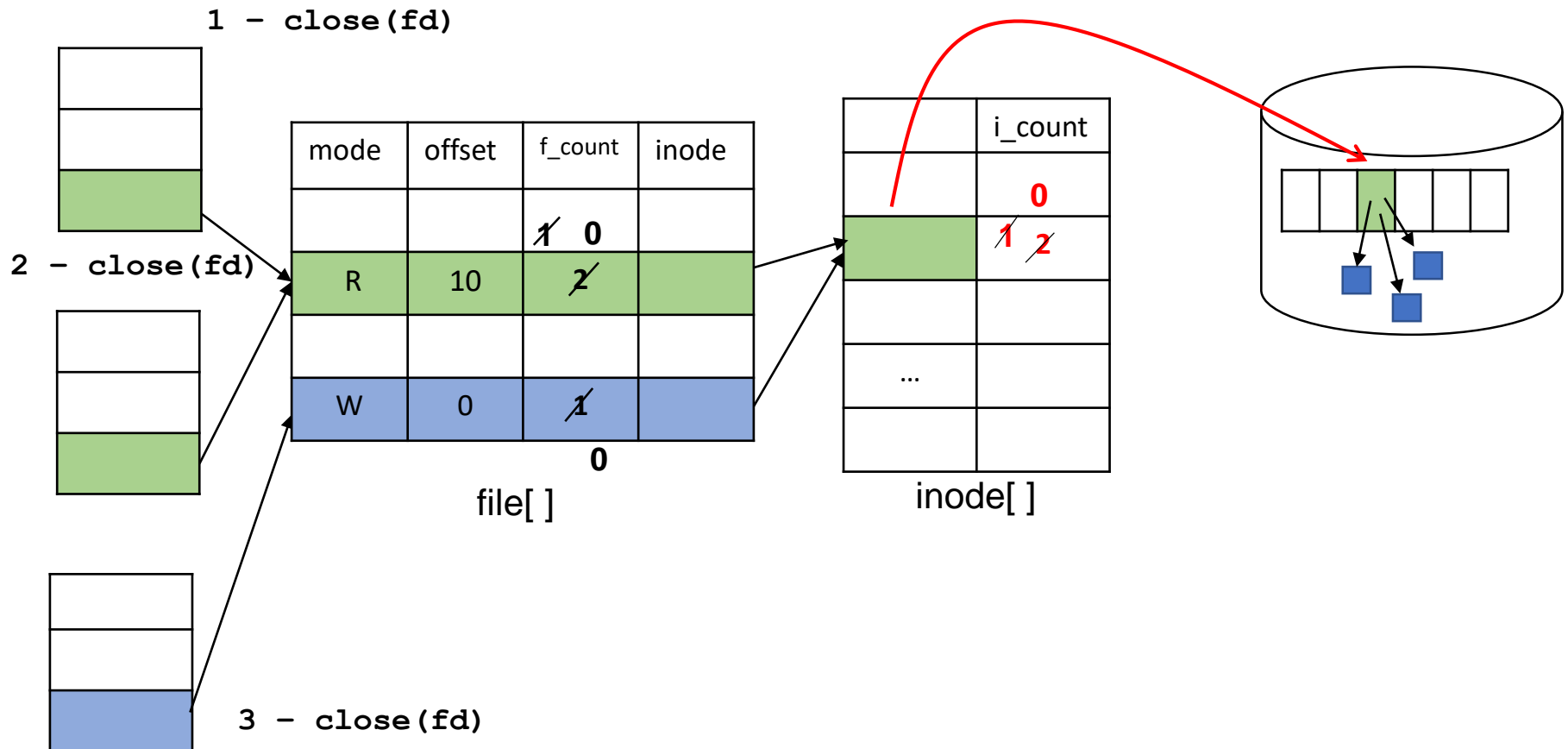
# Fichiers partagés

- Cas 2 : P1 crée un fils



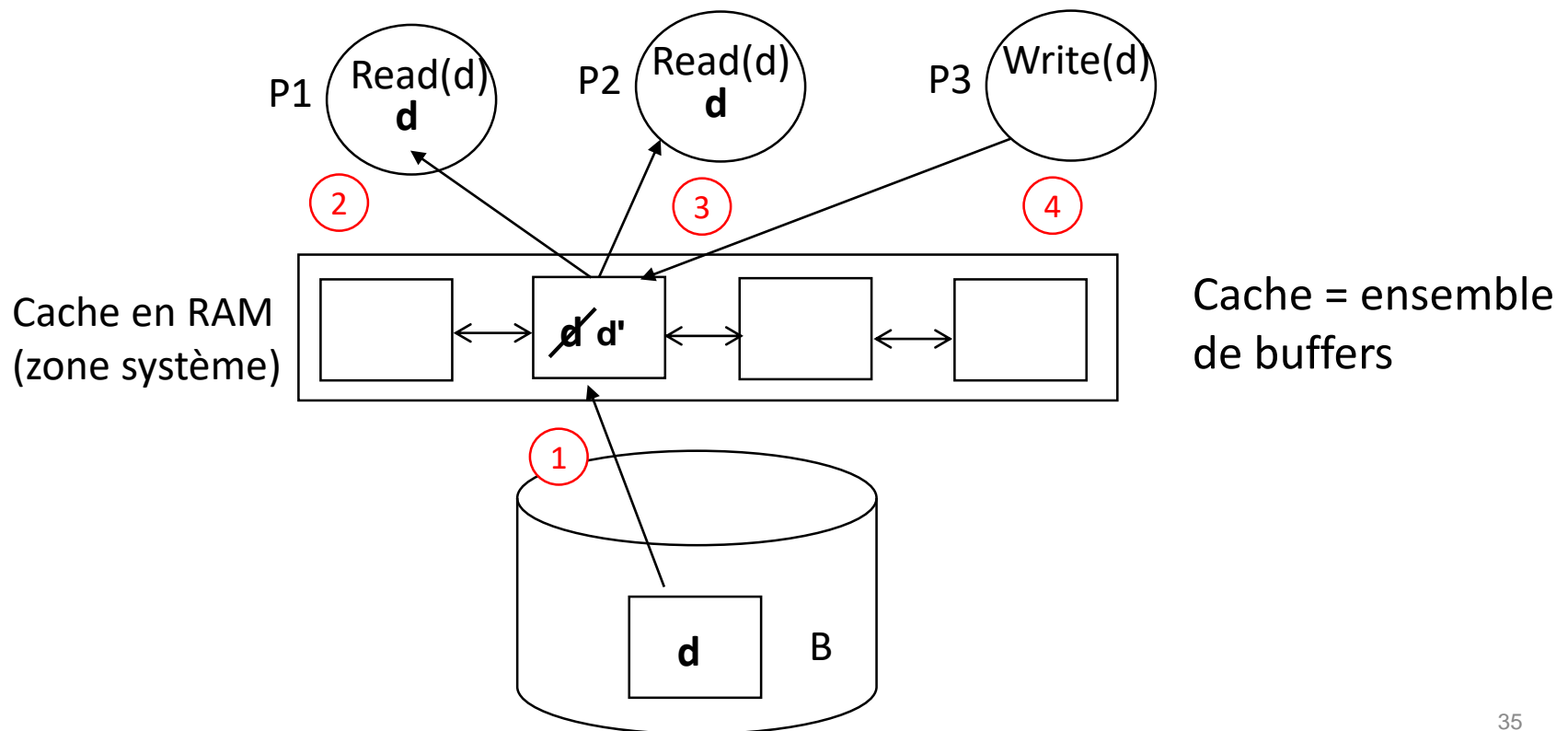
# Opérations sur fichier : close

- Compteurs décrémentés
- Lorsque le fichier n'est plus utilisé ( $i\_count = 0$ ), modifications inode copiées sur disque



# Cache disque

- Constat : forte localité temporelle des accès aux fichiers
- Principe du cache disque : maintenir en mémoire les blocs disque précédemment utilisés



# Expulsion des données du cache

---

- Lorsque cache plein : expulser le buffer B contenant le bloc le moins récemment utilisé (LRU)
- Si B contient de données modifiées ("sale") : écrire les modifications sur disque



Limiter le nombres d'E/S (localité)

Dissocier E/S logique et E/S physique (asynchronisme)



Risque d'incohérence (perte de données) en cas de défaillance  
⇒ écriture régulière des modifications (sync)