File System (Chapitre 4)

Francesco Bronzino
ArchiSys

File System

Sujets

1. Le disque dans les systèmes informatiques
2. Données et disque
3. Les fichiers
4. Les dossiers
5. Attribution des blocs
6. File Systems communs
7. Table de partition
8. Le File System en Linux

Le disque dans les systèmes informatiques

Le disque est un composant fondamental des systèmes informatiques

- Permet une mémoire persistante
  - Il survit lorsque l'ordinateur est redémarré.
- C'est une mémoire réinscriptible.
  - Contrairement aux ROM, PROM et EPROM

Le disque dans les systèmes informatiques

Il existe différentes technologies pour construire des disques

- **Bandes magnétiques:** (obsolète/historique)
- **Disques magnétiques:**
- **Solid State (mémoires flash):**

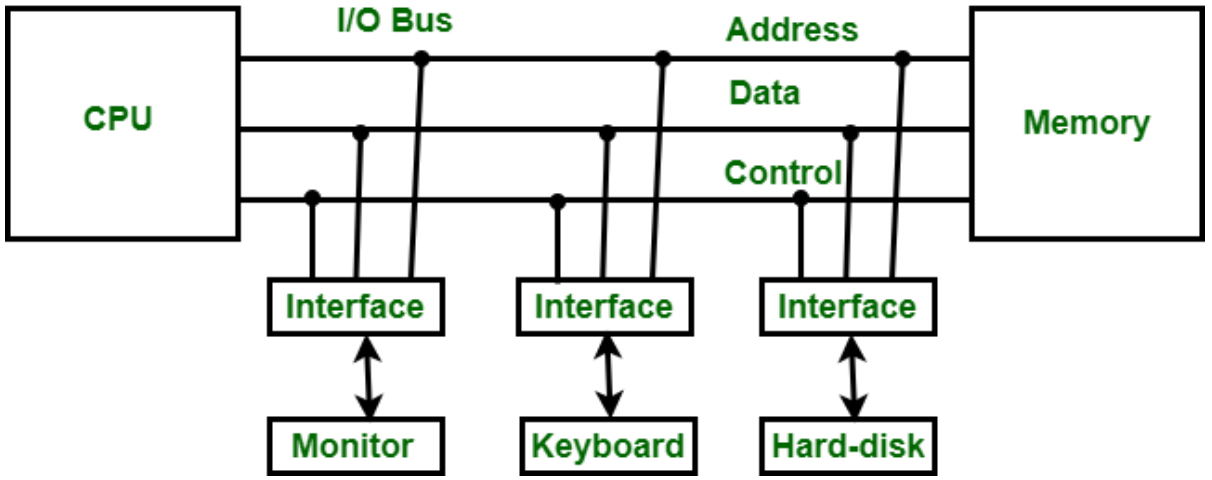
Elles diffèrent par leurs performances, leur coût, leur fiabilité et leur mécanisme d'accès.

- Dans les disques magnétiques, la tête se déplace
- L'accès au disque n'a pas un temps constant
  - Préférence pour les lectures séquentielles
- Mémoire flash : temps d'accès presque constant
  - Ecriture plus lente que la lecture

Le disque dans les systèmes informatiques

Le disque est un périphérique d'I/O

La CPU l'utilise via une *interface*



Accès au disque

En première approximation :

1. L'unité centrale écrit sur l'interface du disque l'emplacement de la mémoire qu'elle veut lire ou écrire.
    - *REMARQUE* : nous n'avons pas encore examiné la mémoire, nous pouvons supposer qu'il s'agit simplement d'un "endroit" où les données sont stockées en utilisant des adresses.
    - Avec que des informations de contrôle (par exemple, s'il faut lire ou écrire).
    - L'interface est accessible de la même manière que n'importe quel emplacement mémoire.
2. Le disque exécute l'opération
3. Le disque positionne des flags dans l'interface qui signalent que l'opération est terminée.
4. La CPU, en lisant les flags, se rend compte que l'opération est terminée.
    - Il lit éventuellement les données de l'interface (en cas de lecture)

Techniques d'optimisation

Il existe d'autres techniques pour rendre l'accès au disque plus efficace.

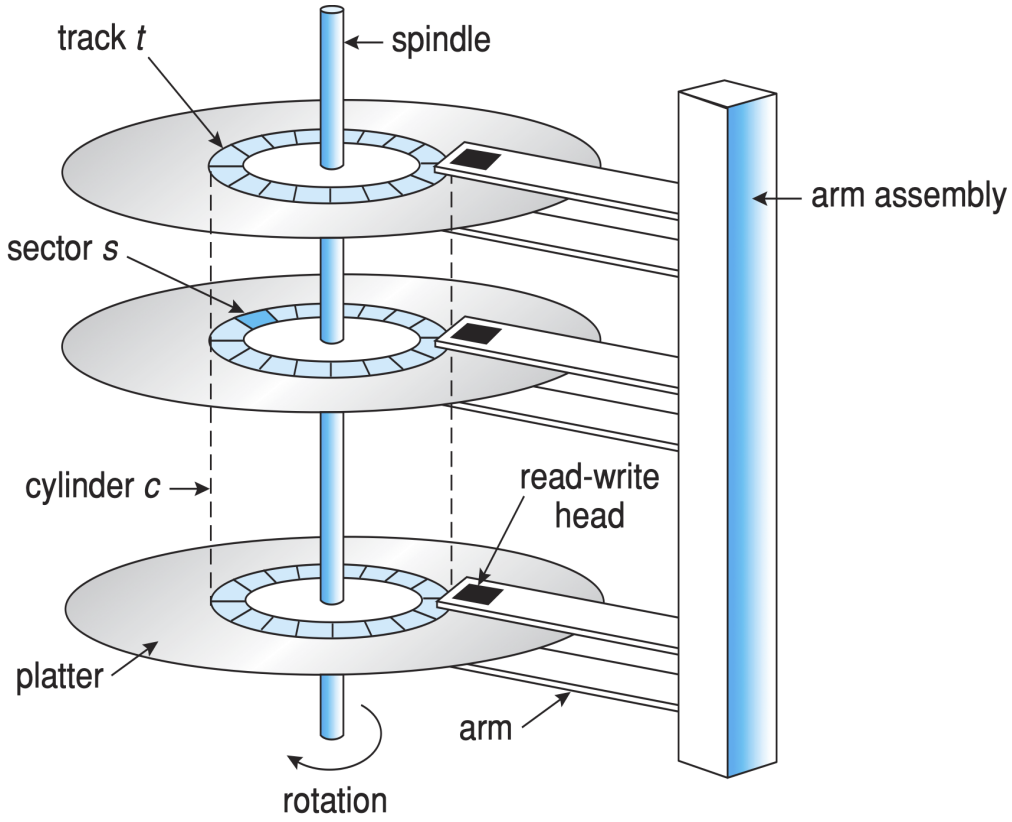
- **DMA** : Accès direct à la mémoire
  - L'unité centrale donne des instructions au disque sur la tâche à effectuer.
  - Le disque lit/écrit les données en mémoire de manière autonome.
- **Caching** : Le système d'exploitation conserve en mémoire vive les parties les plus lues du disque

Données et disque : Pistes et secteurs

Sur un disque magnétique, les données sont organisées en pistes et secteurs concentriques

Les disques à l'état solide sont similaires aux mémoires RAM

- Matrices des celles



Données et disque : Blocs

Dans chaque cas, les disques peuvent être utilisés comme un vecteur de blocs

- Blocs de 1 à 8 Ko

Blocco 0	Blocco 1	Blocco 2	Blocco 3	Blocco 4
Blocco 5	Blocco 6	Blocco 7	Blocco 8	Blocco 9
		....		

Le **File System** permet d'organiser ces blocs pour avoir des

- Des fichiers de taille variable
- Organisés dans une arborescence de dossiers

Les Fichiers

Les fichiers sont une séquence ordonnée de bits qui contiennent des informations.
Ils ont un nom et des attributs :

- Identifiant dans le système d'exploitation
- Permissions
- Heure de création, dernier accès

Les fichiers sont organisés en **dossiers** (ou *folders* ou *directories*)

- Ils peuvent être créés, modifiés ou supprimés comme les fichiers
- Contrairement aux fichiers, ils ne contiennent pas d'octets mais d'autres répertoires ou fichiers

Opérations

Sur les fichiers, un programme (via un appel système OS) peut effectuer les opérations suivantes :

- Création
- Lecture
- Écriture
- Effacer
- *Seek* (~recherche).

Les opérations de lecture et d'écriture sont toujours **séquentielles**. Le fichier est lu/écrit octet par octet via un curseur. Il est possible de repositionner le curseur via l'opération *seek*.

Les dossiers

Sont une collection de noeuds (fichiers ou autres dossiers) avec des propriétés communes.
Organisation typiquement arborescente.

- Un disque est divisé en une ou plusieurs **partitions**
- Chaque partition contient une arborescence de répertoires
  - Il existe un répertoire racine
  - Tous les fichiers et répertoires y sont contenus

Opérations

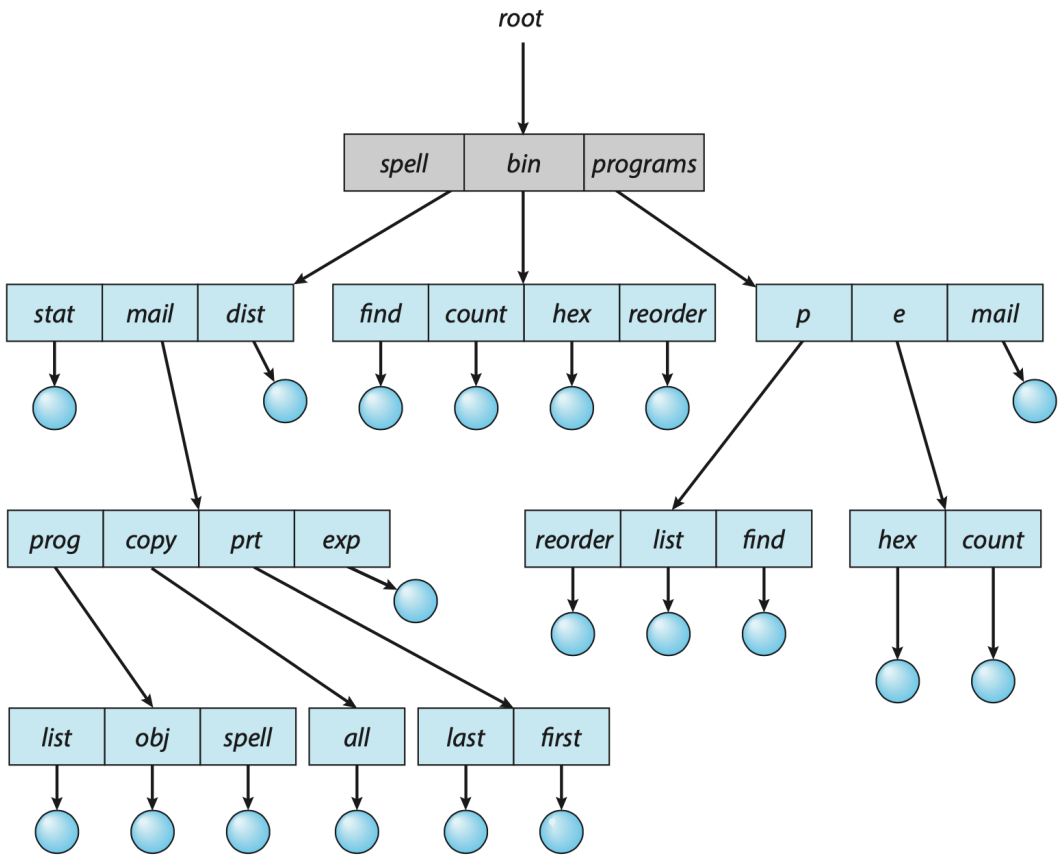
Opérations sur les répertoires. Similaires à celles sur les fichiers :

- Création
- Suppression
- Lister
- Renommer

Rappellez-vous : Le système d'exploitation fournit des **appels système** pour ces opérations.

- Ils sont de bas niveau. Ils peuvent être difficiles à utiliser
- La bibliothèque C standard fournit des fonctions de plus haut niveau (plus faciles à utiliser) qui utilisent les appels système nécessaires en leur sein.

Arbre des dossiers



Liens et boucles

L'arbre est l'organisation la plus naturelle. Cependant, les liens peuvent créer des boucles

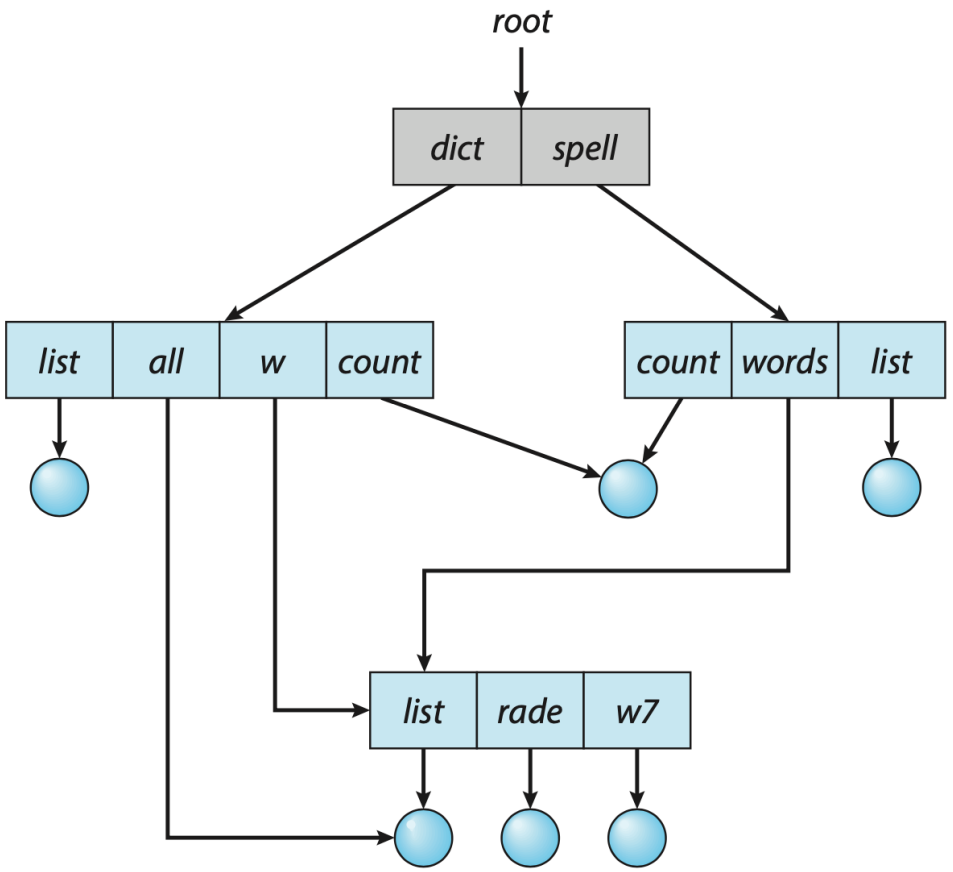
- Avec les liens, la suppression est plus complexe
  - Est-ce que je supprime le fichier original ou une copie ?

Les boucles compliquent beaucoup la gestion des systèmes de fichiers

- Imaginez un processus de recherche récursive dans un dossier avec des boucles.
  - Processus potentiellement sans fin s'il n'est pas géré correctement

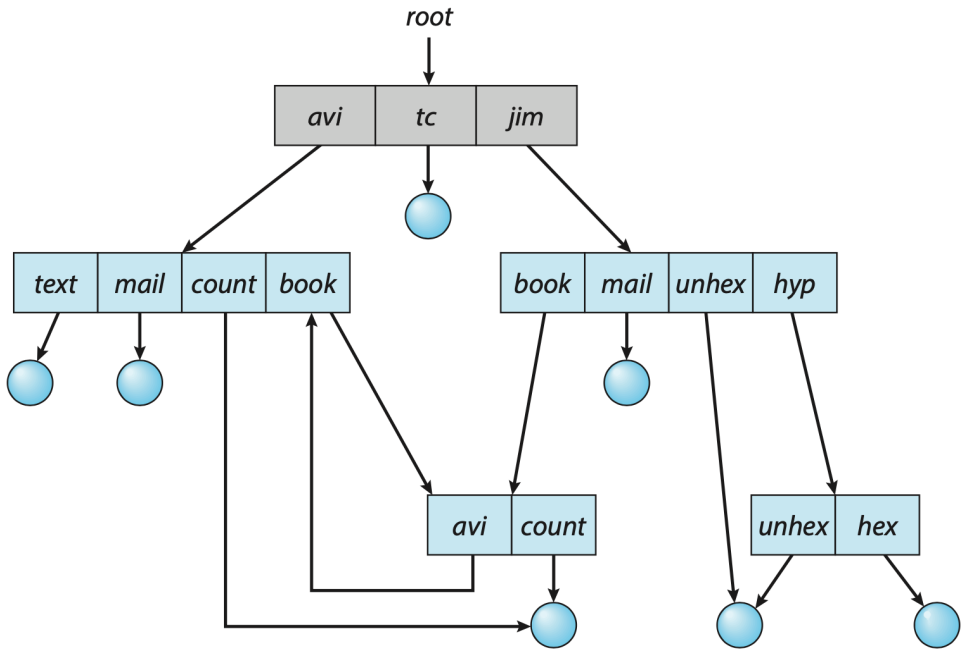
Liens et boucles

Les liens vers des fichiers ne peuvent pas générer de boucles :



Liens et boucles

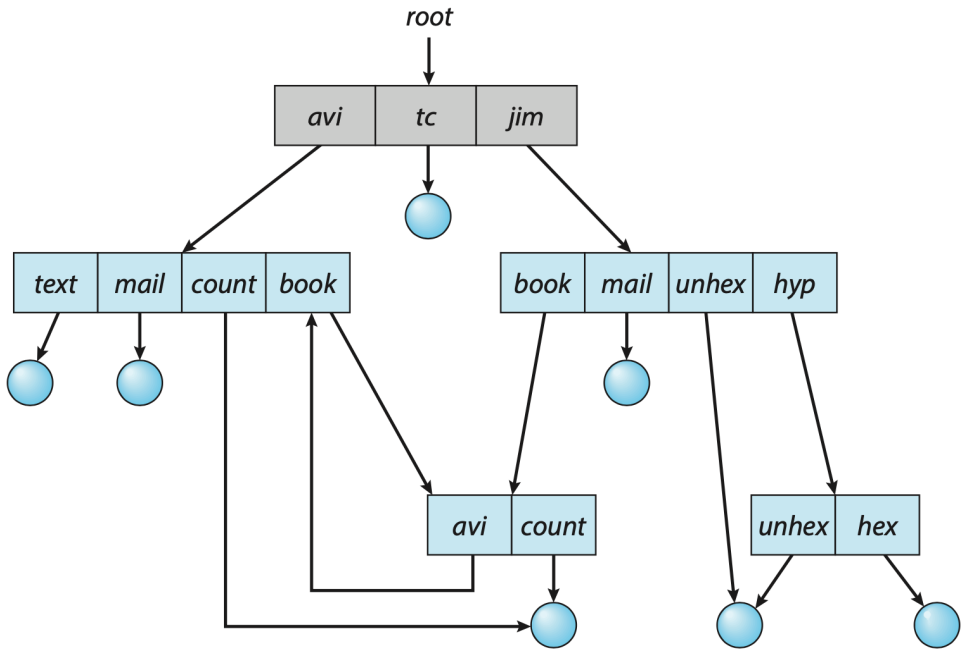
Les liens vers des répertoires peuvent générer des boucles :



Solutions possibles ?

Liens et boucles

Les liens vers des répertoires peuvent générer des boucles :



Solutions possibles: ne jamais visiter les liens pendant la recherche

Lecture et écriture de blocs

Les systèmes de fichiers résident sur les disques

Les disques permettent la lecture et l'écriture en **blocs**.

- Typiquement 512B à 8KB
- Il est possible de lire/écrire un bloc à la fois, et entièrement

Lecture et écriture de blocs

Les **drivers** de disque permettent l'accès à un bloc.

- Ils reçoivent des commandes telles que :

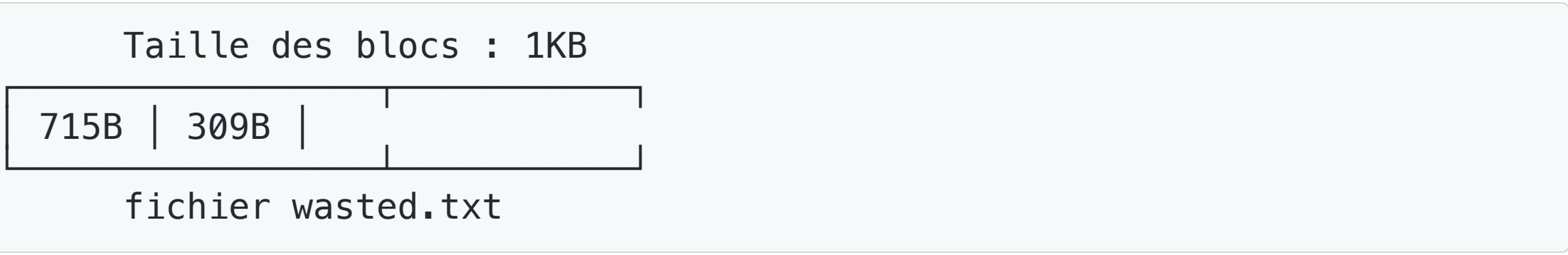
```
lire le bloc 431 à l'adresse mémoire 0x5984
écrire le bloc 126 à l'adresse mémoire 0x9163
```

Le **File System** mappe les accès aux fichiers et aux répertoires en commandes pour le driver.

Attribution de blocs

L'**allocation** est le mécanisme par lequel les blocs sont alloués aux fichiers.

- Chaque fichier occupe 1 ou plusieurs blocs.
- **Fragmentation interne** : gaspillage inhérent de capacité lorsqu'un fichier n'est pas multiple de la taille des blocs.



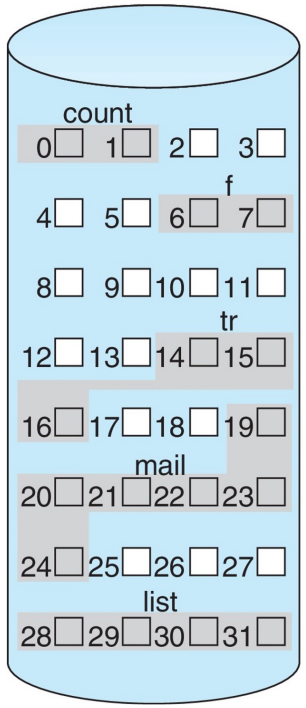
Allocation contiguë

Chaque fichier occupe un ensemble de blocs contigus.

Avantages :

- Simple et rapide
- Accès trivial à l'octet N, car le fichier est stocké de manière contiguë sur le disque.
- Peu de métadonnées par fichier sont nécessaires

Inconvénients ?



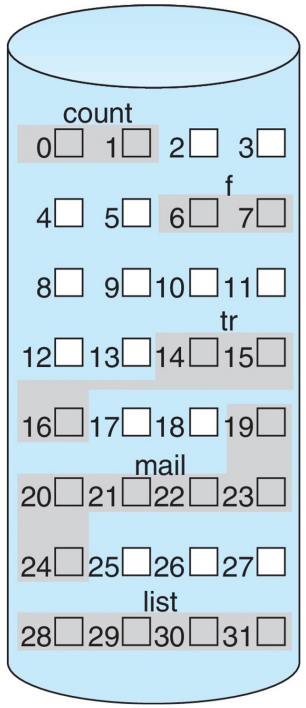
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Allocation contiguë

Inconvénients :

- Crée une **fragmentation externe** : des blocs vides restent éparpillés sur le disque, qui ne peut être utilisé que pour de très petits fichiers.
- Grave gaspillage



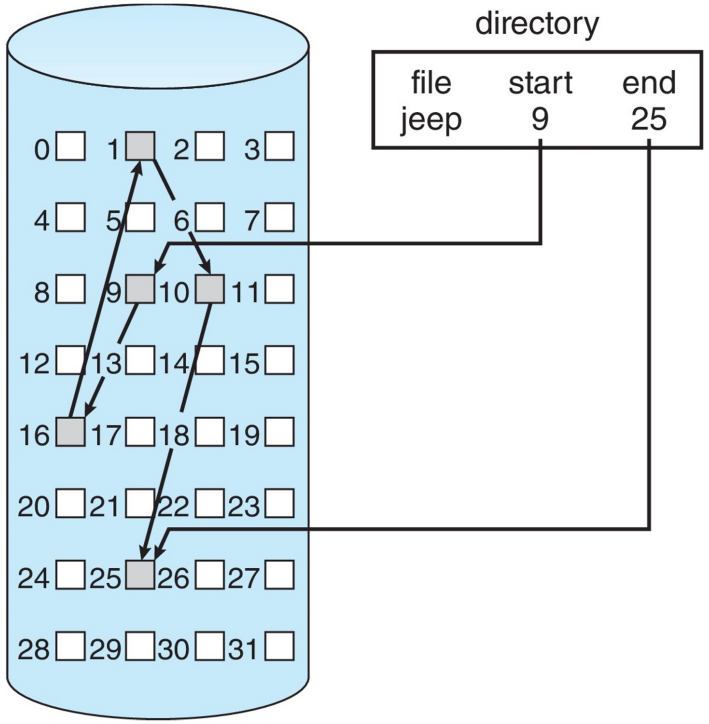
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Allocation concaténée

Chaque fichier est une *Liste liée* de blocs.

- Chaque bloc contient le numéro du bloc suivant
- Le dernier bloc contient un numéro spécial indiquant la fin.

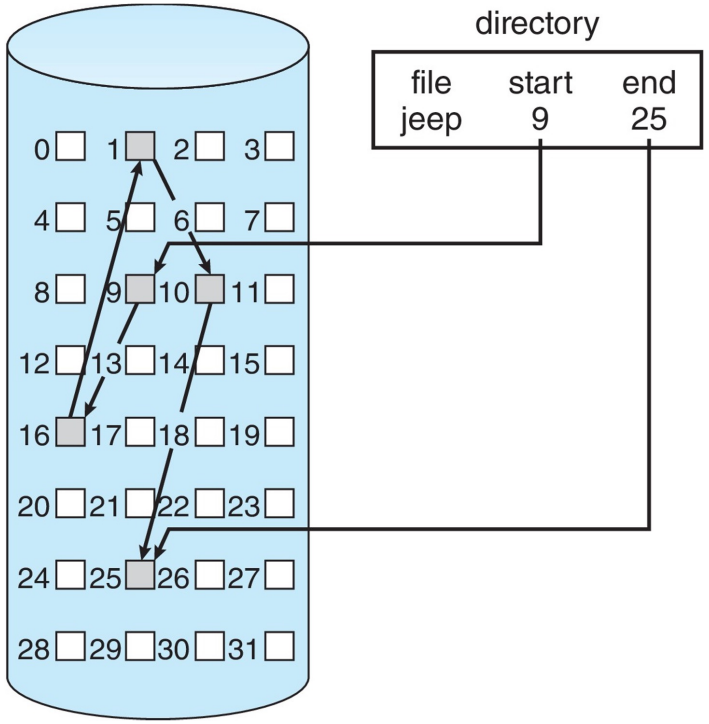


Allocation concaténée

Avantages :

- Pas de **Fragmentation externe**.
- Tous les blocs sont utilisables par fichier.

Inconvénients ?



Allocation concaténée

Inconvénients :

- Efficace pour un accès séquentiel uniquement
- Accéder aux derniers octets du fichier nécessite de faire défiler toute la liste.
  - La lecture d'un *pointeur* nécessite également la lecture de tout le bloc.
  - Rappelez-vous : les disques permettent de lire/écrire un bloc à la fois.
- Peu fiable : un bloc invalide invalide l'ensemble du fichier.
  - Problème pour les gros fichiers

.

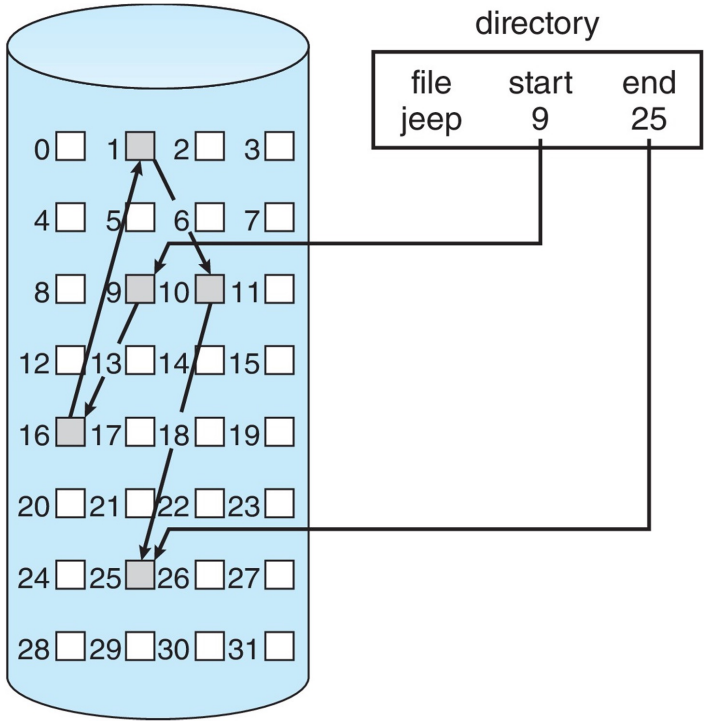


Table d'allocation des fichiers (FAT)

Les premiers blocs du disque contiennent une table FAT.

- Il s'agit d'une *Liste liée* de blocs
- Approche similaire à l'Allocation chaînée
  - Mais la liste est contenue dans les premiers blocs
  - Plus rapide, la FAT peut être mise en cache
- Utilisée dans Windows et MS-DOS avec les systèmes de fichiers *FAT* et *FAT32*.

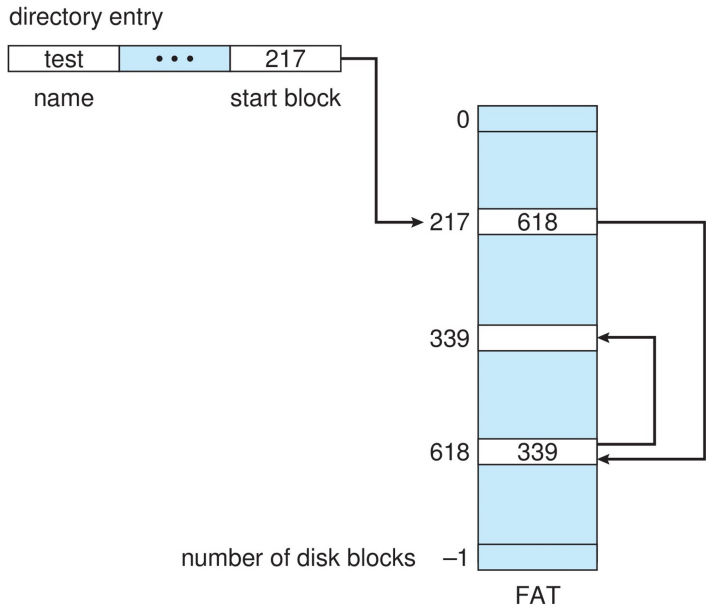


Table d'allocation de fichiers (FAT)

Avantages

- La FAT peut être mise en cache

Inconvénients ?

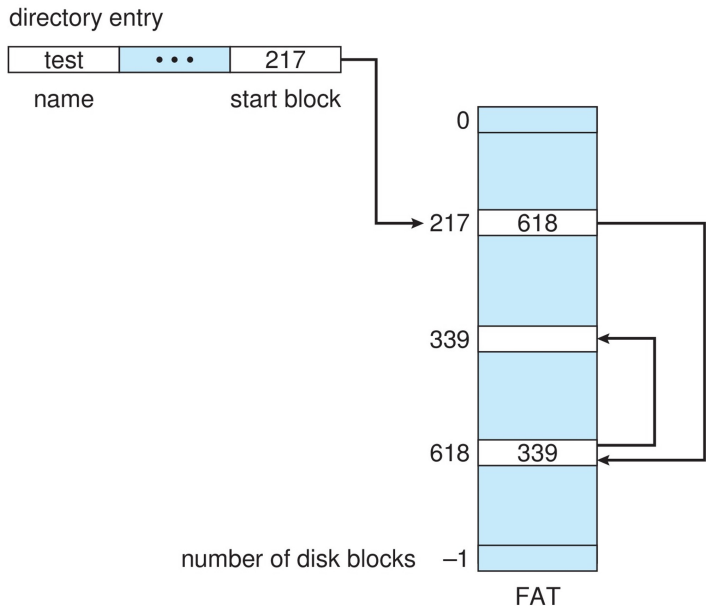


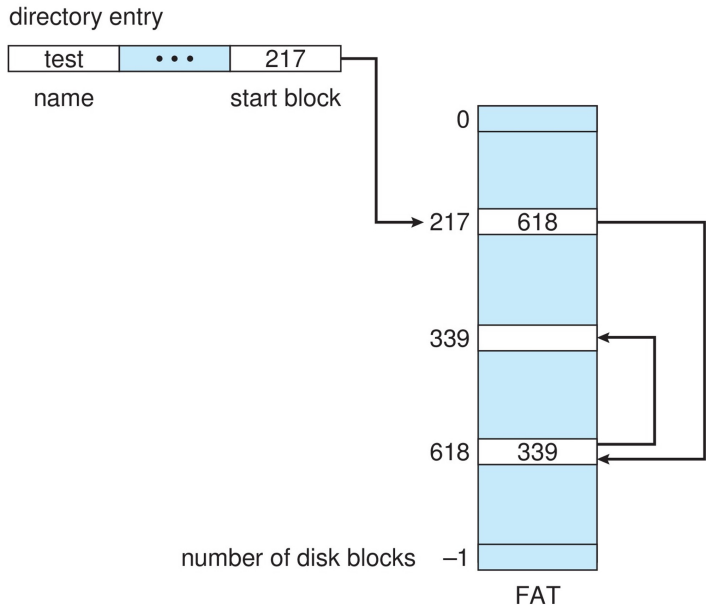
Table d'allocation de fichiers (FAT)

Avantages

- La FAT peut être mise en cache

Inconvénients

- Accès lent aux derniers octets du fichier (en raison de l'allocation concaténée)
- Si je perds la FAT, je perds tout
- La FAT devient grosse pour les gros disques



Allocation indexée

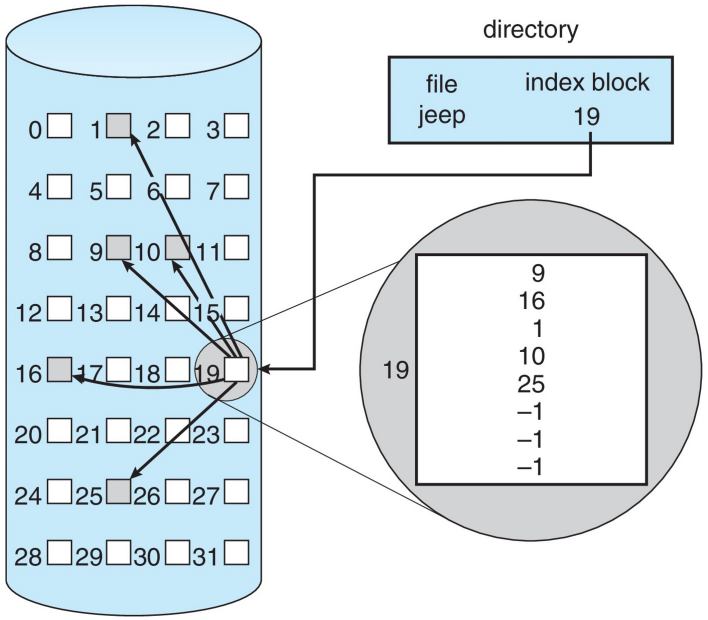
Chaque fichier a un *bloc d'index* qui contient les numéros de tous les blocs.

Avantages

- L'accès à un octet arbitraire est rapide
  - Il suffit de lire le bloc d'index et le bloc désiré

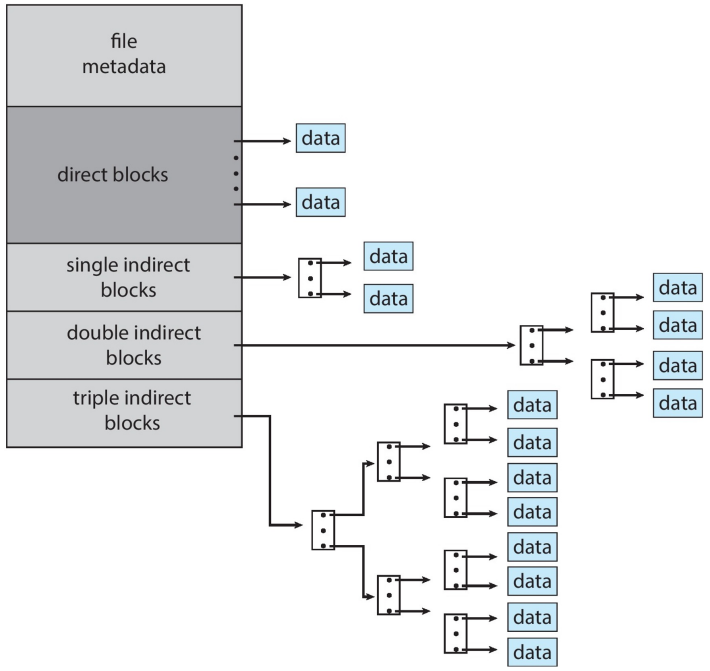
Inconvénients

- Vous gaspillez un bloc par fichier
  - En plus du fichier



Allocation combinée

- Utilisée sous Linux
- Considérée comme le meilleur compromis
- Chaque fichier ou répertoire dispose d'une structure appelée **inode**, qui contient
  - Les métadonnées et les permissions du fichier/dossier
  - Les numéros des 12 premiers blocs
    - Certains invalides, si le fichier est plus petit
  - Des pointeurs indirects
    - Les numéros de blocs qui contiennent eux-mêmes un tableau
    - Sur un, deux et trois niveaux



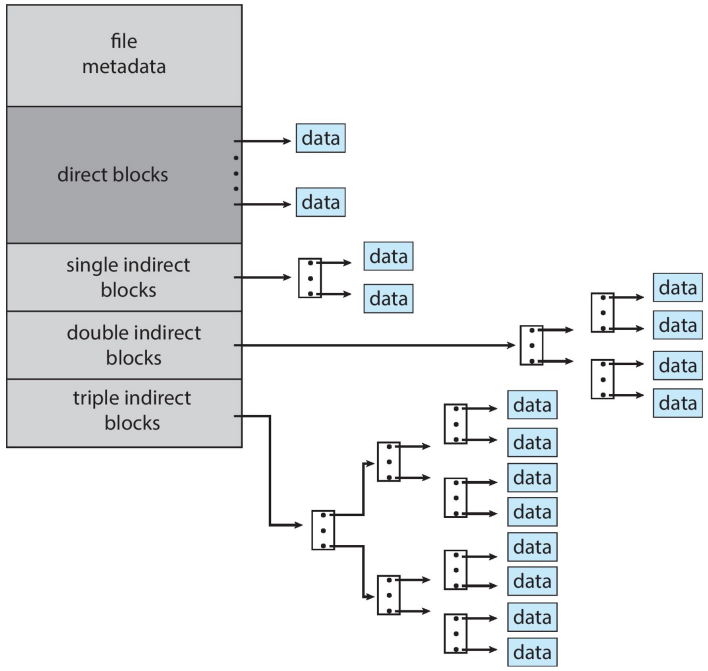
Allocation combinée

Avantages: bon compromis.

- Pas de fragmentation externe
- Aussi indexé pour les petits fichiers
- Peut indexer même les très gros fichiers

Inconvénients:

- Peut nécessiter la lecture de plus d'un bloc pour accéder à des positions avancées dans le fichier.



Systèmes de fichiers communs

Chaque système d'exploitation apporte ses propres systèmes de fichiers

- **Unix:** UFS, FFS
- **Linux:** beaucoup differents.
  - `ext3` et `ext4` sont les standards de facto. Ils utilisent l'allocation combinée
- **Windows:**
  - FAT, basé sur FAT32
  - NTFS : avec adressage par arbre
- **Apple:** HFS, HFS+, APFS
- Systèmes de fichiers distribués pour le Big Data : GoogleFS, HDFS, CEPH

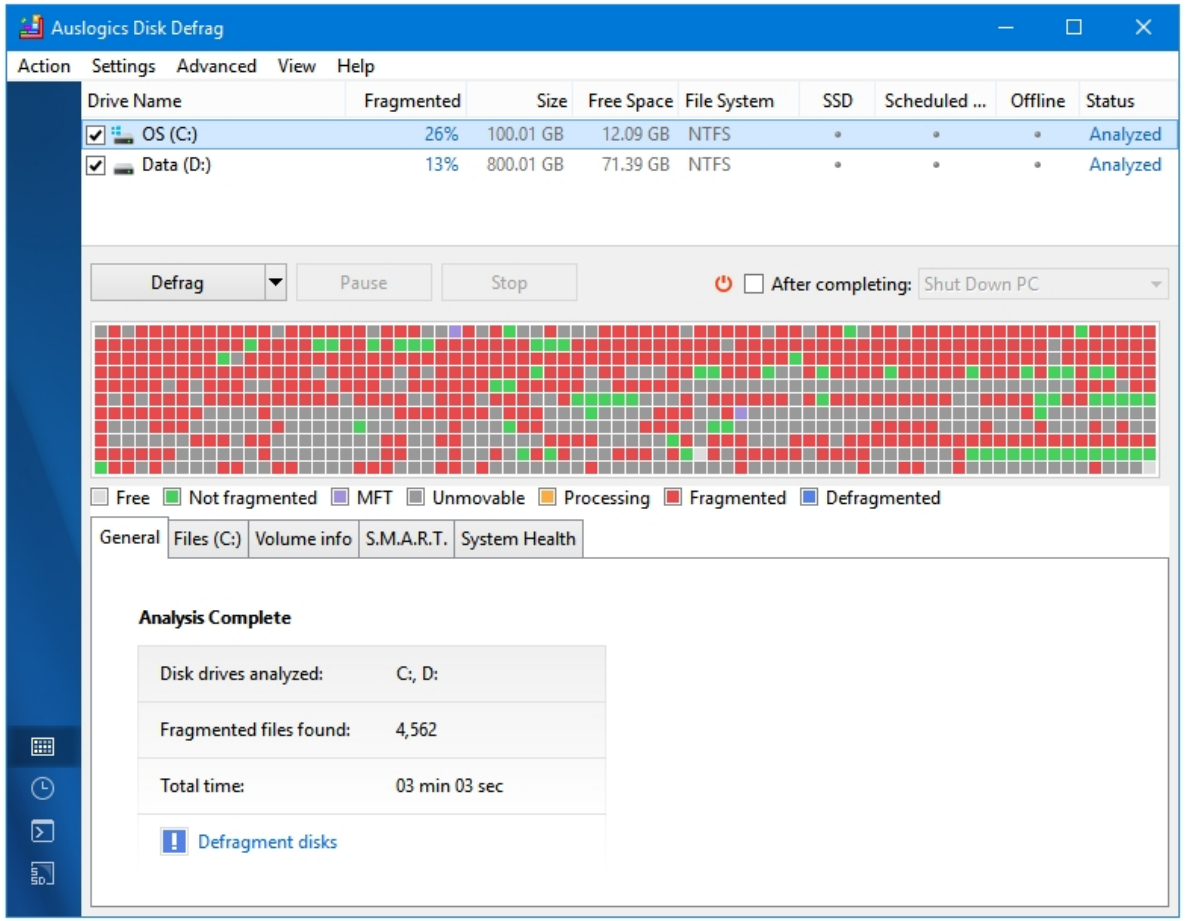
Tables de partition

En plus des File Systens, il existe des normes pour partitionner les disques en plusieurs partitions.

- **Master boot record (MBR):** Méthode classique. Les premiers secteurs du disque indiquent les partitions.
  - Il contient également le code initial pour démarrer l'ordinateur.
  - Disques de 2 To maximum et 4 partitions
- **Tableau de partition GUID (GPT)** moderne, faisant partie de la norme UEFI (Unified Extensible Firmware Interface)
  - Surmonte les limites du MBR

Bonus question from an old man

Pouvez-vous deviner le sujet de cette photo ?



File System sous Linux

Dans le monde Unix/Linux, de nombreux systèmes de fichiers existent.

- Le premier était le système de fichiers Unix (UFS).
- De celui-ci est né le système de fichiers étendu (`ext`) pour Linux.
- Nous sommes maintenant à la version `ext4` .

Comme vu, basé sur le concept d'un inode représentant un fichier ou un répertoire

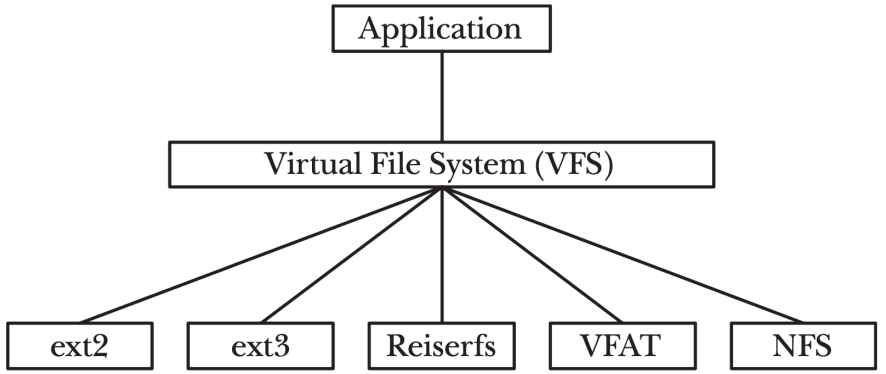
Système de fichier virtuel (VFS)

Différents systèmes de fichiers peuvent être utilisés.

Ils doivent implémenter l'interface **Virtual File System (VFS)**.

- C'est-à-dire qu'ils doivent permettre l'exécution de certaines fonctions de base :

```
open(), read(), write(), lseek(), close(), truncate(), stat(),
mount(), umount(), mmap(), mkdir(), link(), unlink(), symlink(), rename()
```



`mount` des systèmes de fichiers

Sous Linux, tous les fichiers de chaque système de fichiers se trouvent dans une seule arborescence de répertoires.

- Qui découle de `/`
- Les systèmes de fichiers supplémentaires sont *montés* en tant que sous-arbres de `/`.

Pour *monter* un FS :

```
mount device directory
```

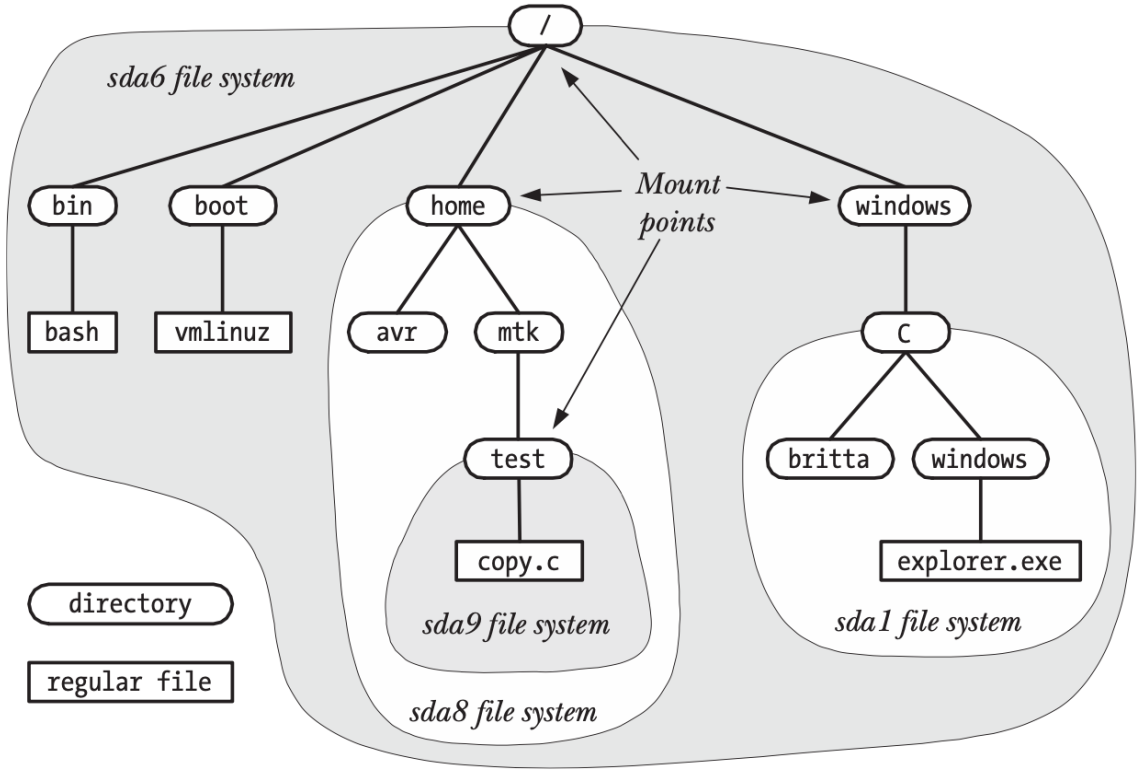
`mount` des systèmes de fichiers

Pour voir le FS *monté*

```
$ mount
/dev/sda6 sur / type ext4 (rw)
proc sur /proc type proc (rw)
sysfs sur /sys type sysfs (rw)
devpts sur /dev/pts type devpts (rw,mode=0620,gid=5)
/dev/sda8 sur /home type ext3 (rw,acl,user_xattr)
/dev/sda1 sur /windows/C type vfat (rw,noexec,nosuid,nodev)
/dev/sda9 sur /home/mtk/test type reiserfs (rw)
```

`mount` des systèmes de fichiers

Exemple d'une hiérarchie de FS *mount*.



`mount` des systèmes de fichiers

Dans un système Linux, les systèmes de fichiers qui sont montés automatiquement au démarrage sont spécifiés dans le fichier `/etc/fstab`.

- Il contient une ligne pour chaque système de fichiers
- Format `<file system> <mount point> <type> <options> <dump> <pass>`

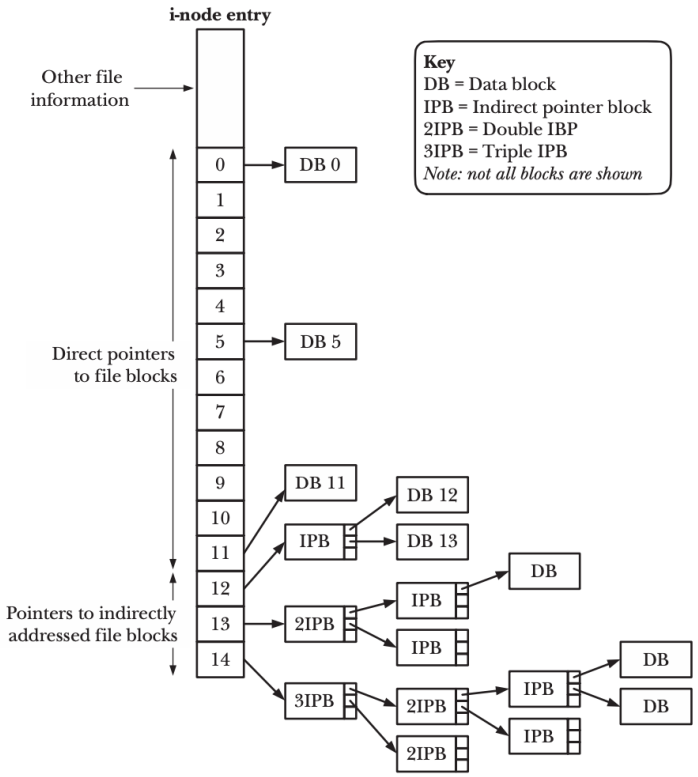
Exemple:

```
/dev/sda1 / ext4 errors=remount-ro 0 1
/dev/hda1 /media/hda1 vfat defaults,utf8,umask=007,gid=46 0 0
```

Inodes

Représentent un fichier. Stockés dans une table dans les premiers blocs.

- Chaque inode est une structure de quelques octets.
- Identifiée par le **numéro d'inode**.
- Sont en nombre fini et immuables
  - Ne peuvent pas stocker un nombre infini d'inodes

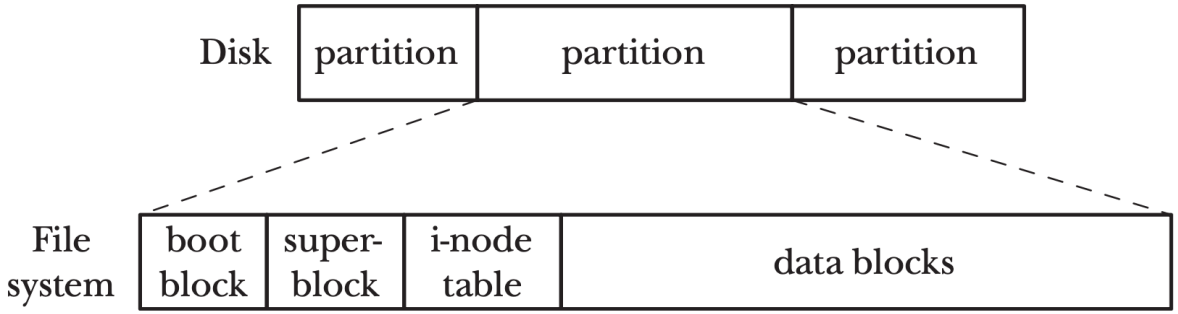


Disposition d'un disque

Un disque est divisé en partitions

Chaque partition contient

- Des informations de contrôle
- Table d'inodes
- Des blocs de données



Types de blocs

Les blocs de données sont de deux types :

- Data Block : ils ont le contenu d'un fichier. Données binaires
- Directory Block : ont le contenu d'un répertoire. Liste de paires $(name, inode)$

Dossiers

Chaque dossier est un inode

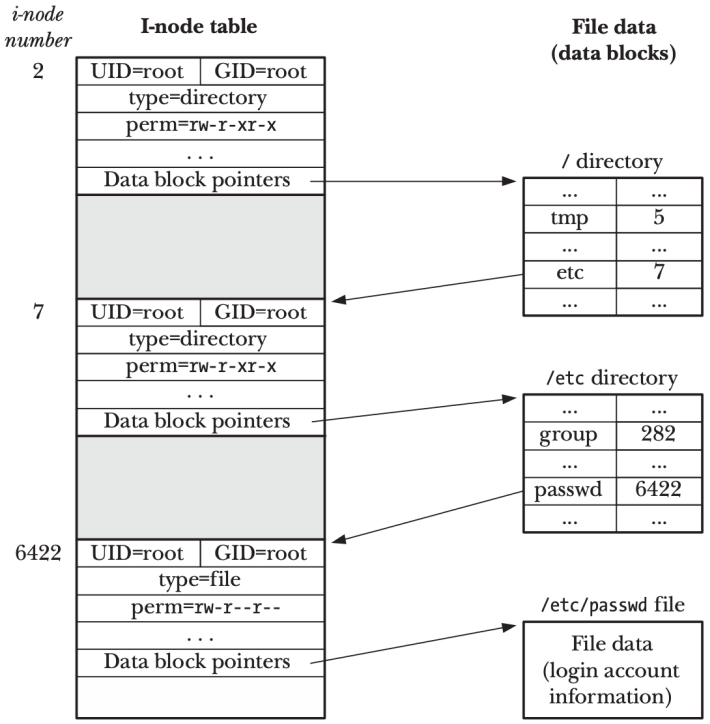
- Il possède au moins un bloc contenant la table des noeuds qu'il contient

On peut le considérer comme un fichier :

- Dont le contenu n'est pas un ensemble d'octets
- mais une liste de paires $(name, inode)$

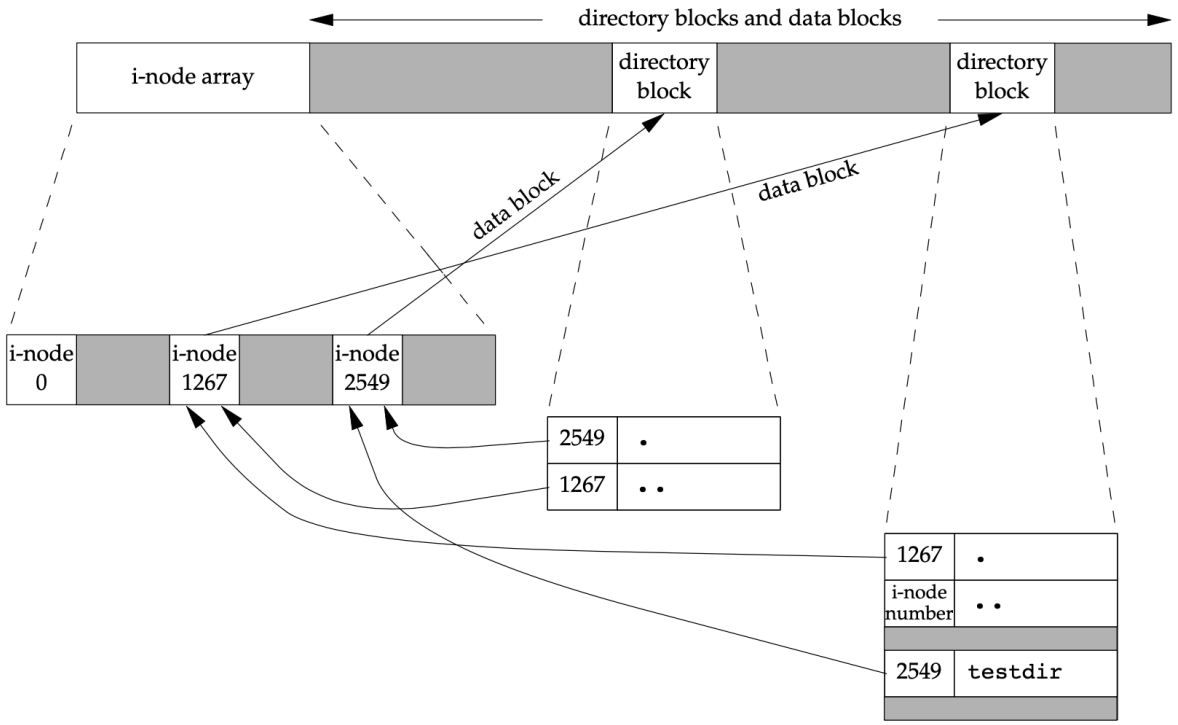
Exemple de répertoire

- inode 2 est le répertoire `/`
  - Contient `etc` : inode 7.
- inode 7 est le répertoire `/etc`.
  - Contient le répertoire `passwd` : inode 6442.
- L'inode 6442 est le fichier `/etc/passwd`.
  - Le contenu est dans un bloc de données



Exemple de dossier

Le dossier 1267 contient le dossier `testdir` 2549 qui est vide



Permissions dans Linux : Utilisateurs et groupes

Partiellement vu au début du cours

Sous Linux, il y a :

- **Utilisateurs** : comptes qui peuvent utiliser le système, créer des processus, accéder à des fichiers.
- **Groupes** : des ensembles d'utilisateurs. Chaque utilisateur a un :
  - **Groupe principal:** un seul
  - **Groupes secondaires:** dont le nombre n'est pas limité.

Chaque utilisateur et chaque groupe est identifié par un **nom** et un **id** numérique.

Utilisateur root

L'utilisateur **root** existe sur tous les systèmes

- A un *id* 0
- Contourne tous les contrôles de permissions

Note : **root** est un utilisateur avec des privilèges illimités. **Il ne fait PAS** partie du noyau, et son code ne s'exécute pas en mode noyau

Erreur courante : de dire que l'utilisateur **root** exécute des processus en mode noyau !

Fichiers pour les utilisateurs et les groupes

Informations sur les utilisateurs et les groupes actifs stockées dans des fichiers de configuration accessibles uniquement à root :

- `/etc/passwd` : liste des utilisateurs et détails (ID, *répertoire personnel*)
- `/etc/shadow` : mots de passe cryptés
- `/etc/group` : liste des groupes et de leurs composants

Classes

Chaque fichier/dossier a un utilisateur **propriétaire** et un groupe **propriétaire**.
Il est possible de séparer les permissions par classes d'utilisateurs.
C'est-à-dire que vous spécifiez les permissions séparément pour :

- L'utilisateur **propriétaire** : s'applique lorsque le propriétaire tente de se connecter.
- **Groupe du propriétaire** : s'applique lorsqu'un utilisateur du groupe du propriétaire se connecte.
    - Remarque : le groupe propriétaire n'est pas nécessairement le **groupe principal** du propriétaire, mais peut être un **groupe secondaire**.
- **Autres** : tous les autres utilisateurs

Permissions de base

- **Lecture:** pour les fichiers, lire le contenu. Pour les dossiers, lister les noeuds de contenu
- **Write:** pour les fichiers, écrire le contenu. Pour les dossiers, ajouter/supprimer des noeuds de contenu.
- **Exécution:** pour les fichiers, les exécuter. Pour les dossiers, les traverser, c'est-à-dire accéder aux fichiers contenus. Autre que la liste (permission de lecture).

Note: ce ne sont pas les permissions sur un fichier qui déterminent s'il peut être supprimé, mais les permissions sur le répertoire qui le contient.

Permissions spéciales

En plus des 3 permissions de base, il existe 3 autres permissions spéciales (ou drapeaux) qui peuvent être appliquées aux fichiers/dossiers

- **set user ID (suid):** pour les **fichiers**, si elle est exécutée, le processus est exécuté avec les privilèges de **l'utilisateur propriétaire**, et non les privilèges de l'exécuteur. Pour les **dossiers**, cela n'a aucun effet.
  Utilisation: Sur les PC, les commandes système (par exemple, `reboot`) ont le **suid**, pour permettre le redémarrage sans demander un mot de passe. Sur les serveurs, généralement pas !

Permissions spéciales

- **set group ID (guid):** pour les **fichier**, si executé, le processus est lancé avec les privilèges **owner group**, pas executor. Pour les **dossiers**, les fichiers créés ont le groupe dossier et non le groupe principal du créateur (qui est l'action par défaut).
  Utilisation: lors de la création de dossiers partagés entre utilisateurs appartenant à un groupe spécialement créé.
  Exemple : Pour un projet, vous créez le groupe `projectSysOp`, qui contient 3 utilisateurs. Vous créez le dossier partagé `/share/projet` et l'attribuez au groupe `projetSysOp`.

Permissions spéciales

- **sticky bit:** pour les **fichiers**, a (plus d'effet). Pour les **dossiers**, les fichiers qui s'y trouvent ne peuvent être supprimés et déplacés que par les utilisateurs qui en sont propriétaires, ou par l'utilisateur qui est propriétaire du dossier. **Utilisation:** Dans les dossiers `/tmp` et `/var/tmp`, tous les utilisateurs doivent pouvoir créer et modifier des fichiers. Personne, sauf le super-utilisateur, ne doit pouvoir supprimer ou déplacer les fichiers temporaires des autres utilisateurs.

Liens symboliques

?

Liens symboliques

Ils n'ont pas de permissions propres, mais héritent des permissions du fichier/dossier lié.

Note: leur création/destruction est rendue possible par les permissions du dossier dans lequel ils se trouvent.

Exemple :

```
$ ls /lib
...
lrwxrwx 1 root root 16 Feb 24 2020 sendmail -> ../sbin/sendmail
...
```

Représentation symbolique

Le premier caractère indique le type de fichier ou de répertoire listé, et ne représente pas correctement une permission :

- `-` : fichier ordinaire
- `d` : répertoire
- `b` : périphérique bloc
- `c` : périphérique de caractères
- `l` : lien symbolique
- `p` : pipe nommé
- `s` : socket de domaine Unix

Commandes Bash pour les disques

- `df` : affiche les disques et leur occupation
- `mount` : vous permet de
  - voir quels systèmes de fichiers sont utilisés
  - *monter* un disque, c'est-à-dire l'attacher à l'arborescence de fichiers de la machine
  - utiliser l'appel système `mount`.
- `fdisk` : voir les disques et les partitions, et créer des partitions
- `lsblk` : visualiser facilement les partitions et les disques
- `mkfs` : Formater et initialiser un système de fichiers sur un disque
- `lspci` et `lsusb` : lister les périphériques PCI et USB, y compris les disques.

Files

Sujets

1. Appel système pour les fichiers
2. Fonctions de bibliothèque et appel système
3. Fonctions C pour les fichiers
4. Commandes Bash pour les fichiers

Fonctions de bibliothèque pour les fichiers

Elles font partie de la `libc`.

- Elles peuvent être utilisées sur n'importe quel OS
- Il suffit de recompiler le programme
- Elles utilisent des appels système différents selon le système d'exploitation

Fonctions de la bibliothèque pour les fichiers

Fonctions principales :

- Pointeur de fichier : `FILE *`.
- Ouverture/Fermeture : `fopen`, `fclose`
- Lecture/écriture de caractères : `fgetc`, `fputc`
- Lire/écrire des lignes : `fgets`, `fputs`
- Lecture/écriture avec format : `fscanf`, `fgets`
- Lecture/écriture brute : `fread`, `fwrite`
- Repositionnement : `fseek`, `rewind`, `ftell`

Fonctions de la bibliothèque pour les fichiers

```c
#include <stdio.h>

int main ()
{
    char s[100], buffer [100];
    FILE * f;

    printf("Insert a path: ");
    scanf("%s", s);

    f = fopen(s, "r");
    if (f==NULL){
        printf("Impossible to open %s\n", s);
        return 1;
    }

    while ( fgets(buffer, 100, f) )
        fputs(buffer, stdout);

    fclose(f);
    return 0;
}
```

Fonctions de la bibliothèque pour les fichiers

La lecture/écriture dans les fichiers est **séquentielle**.
Un fichier ouvert possède un **curseur** similaire à celui d'un éditeur de texte

- Il détermine la position de départ de l'opération suivante
- Il est positionné à un **offset** précis dans le fichier
- Une lecture déplace le curseur des caractères lus vers l'avant
- Une ecriture insère des caractères à la position du curseur

Des fonctions existent pour déplacer le curseur de manière arbitraire

Funzioni di libreria per file

Considérons un fichier de 10 caractères contenant la phrase `Hello World`. Il est ouvert avec :

```
FILE * fp = fopen("hello.txt", "r");
```

Le curseur se trouve maintenant au début du fichier.

```
↓
Hello World
```

La lecture avec `fscanf` lit un mot (y compris l'espace final)

```
fscanf(fp, "%s", buffer);
```

Le curseur sera alors au début du mot suivant. Un `fscanf` ultérieur lira `World`

```
      ↓
Hello World
```

Fonctions de la bibliothèque pour les fichiers

Il est possible de déplacer le curseur manuellement avec la fonction

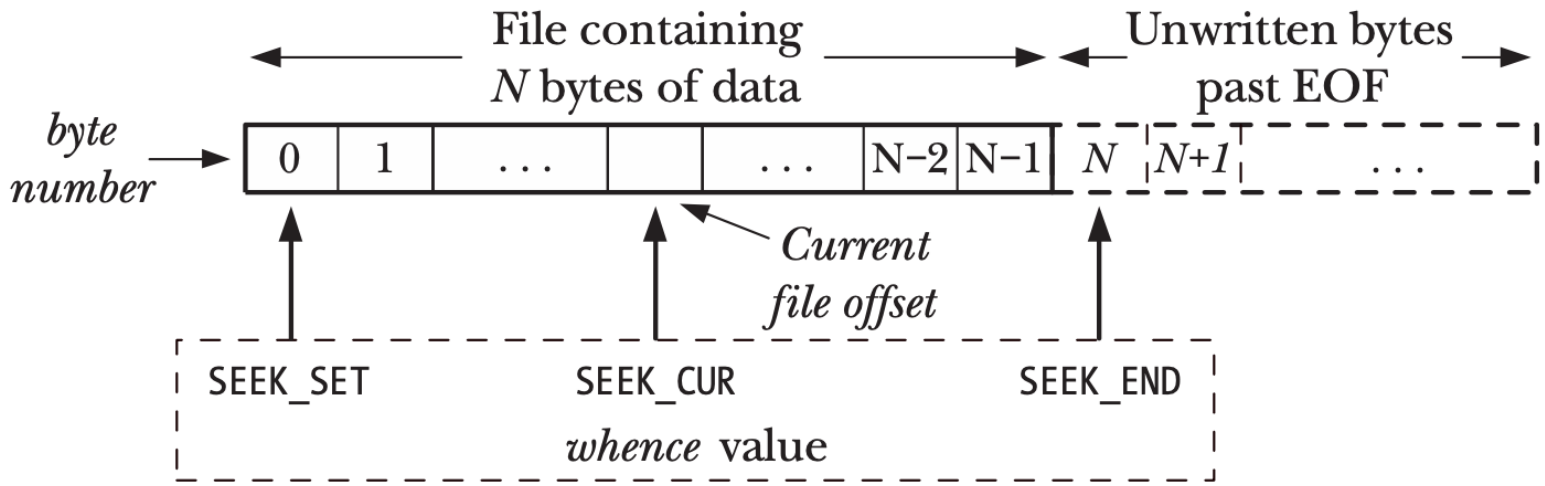
```
int  fseek(FILE *fp, long distance, int start)
```

Où :

- `FILE *fp` : est le pointeur du fichier sur lequel agir
- `long distance` : est le nouveau décalage.
- `int start` ou `whence` indique d'où est calculée la `distance`. Il peut prendre
  - `SEEK_SET` : début du fichier
  - `SEEK_END` : fin du fichier
  - `SEEK_CUR` : position actuelle du curseur

Fonctions de la bibliothèque pour les fichiers

Fonctionnement de `fseek`



Fonctions de bibliothèque pour les fichiers

Considérons un fichier de 11 caractères contenant la phrase suivante .

`Hello World` .

```
FILE * fp = fopen('hello.txt', 'r') ;
fscanf(fp, "%s", buffer) ; // Le buffer contient `Hello`. Le curseur est avant 'World'.

fseek(fp, 0, SEEK_SET) ; // Le curseur retourne au début du fichier.
fscanf(fp, "%s", buffer) ; // Relisez "Hello".
```

De même :

```
FILE * fp = fopen("hello.txt", "r") ;
fgets(buffer, 2, fp) ; // Le buffer contient 'He'. Le curseur est entre 'He' et 'llo World'.
fseek(fp, -3, SEEK_END) ; // Le curseur se trouve entre "Hello Wo" et "rld".
fgets(buffer, 2, fp);// Je lis "rl"
```

Fonctions de bibliothèque pour les fichiers

Fonctions `fread` et `fwrite`

Similaire à `fgets` et `fputs`, mais ignore le retour chariot `\n`.

- Lire un nombre fixe de caractères sans tenir compte des retours chariot.
- Bon pour les fichiers binaires : ils contiennent des caractères non imprimables, et aussi beaucoup de `'\0'` (terminateurs).
  - Impossible de lire correctement les terminaisons.

```
size_t  fread(void *restrict ptr, size_t size, size_t nmemb,
              FILE *restrict stream) ;
size_t  fwrite(const void *restrict ptr, size_t size, size_t nmemb,
              FILE *restrict stream) ;
```

Fonctions de bibliothèque pour les fichiers

Fonctions `fread` et `fwrite`

Elles vous permettent de lire/écrire des fichiers binaires (qui contiennent `'\0'`).

- Ou `struct` et vecteurs

Fonction : lire/écrire dans les objets `nmemb`, chaque gros `size` octet du pointeur de fichier `stream`, et les écrire/lire depuis `ptr`.
Valeur de retour : Le nombre d'éléments effectivement lus/écrits.

Fonctions de bibliothèque pour les fichiers

What about `fgets` et `fputs` ?

Fonctions de bibliothèque pour les fichiers

What about `fgets` et `fputs` ?

La fonction de la bibliothèque `char *fgets(char *str, int n, FILE *stream)` lit une ligne du flux spécifié et la stocke dans la chaîne pointée par `str`. Elle s'arrête soit lorsque $(n-1)$ caractères sont lus, soit lorsque le caractère de nouvelle ligne est lu, soit lorsque la fin du fichier est atteinte, selon ce qui se produit en premier

Fonctions de bibliothèque pour les fichiers

Lecture d'un vecteur d'entiers.

Supposons un fichier contenant (en binaire) les 2 entiers représentant les nombres 1990 et 2023.

En hexadécimal et en considérant `int` sur 4 octets, on a dans le fichier :

```
0x00 0x07 0xC6 0x00 0x07 0xE7
```

Pour les lire en C, on procède avec la fonction `fread`.

```
FILE * fp = fopen("hello.txt", "rb") ; // Notez le mode "rb".
int v [2] ;
fread(v, sizeof(int), 2, fp) ;
```

Note: il est erroné d'utiliser `fscanf(... , "%d", ...)` qui s'attend à des nombres écrits comme des chaînes de caractères !

Fonctions de la bibliothèque pour les fichiers

Observations de l'exemple précédent:

- `size_t` est un alias pour le type de données *integer unsigned* qui est utilisé pour représenter les tailles des structures de données.
- `sizeof` est un opérateur qui retourne le nombre d'octets qu'occupe un type de données.
    - Pendant la **compilation**, le compilateur remplace l'expression par son résultat

Fonctions de la bibliothèque pour les fichiers

En C, l'entrée et la sortie des fichiers sont **buffered**.

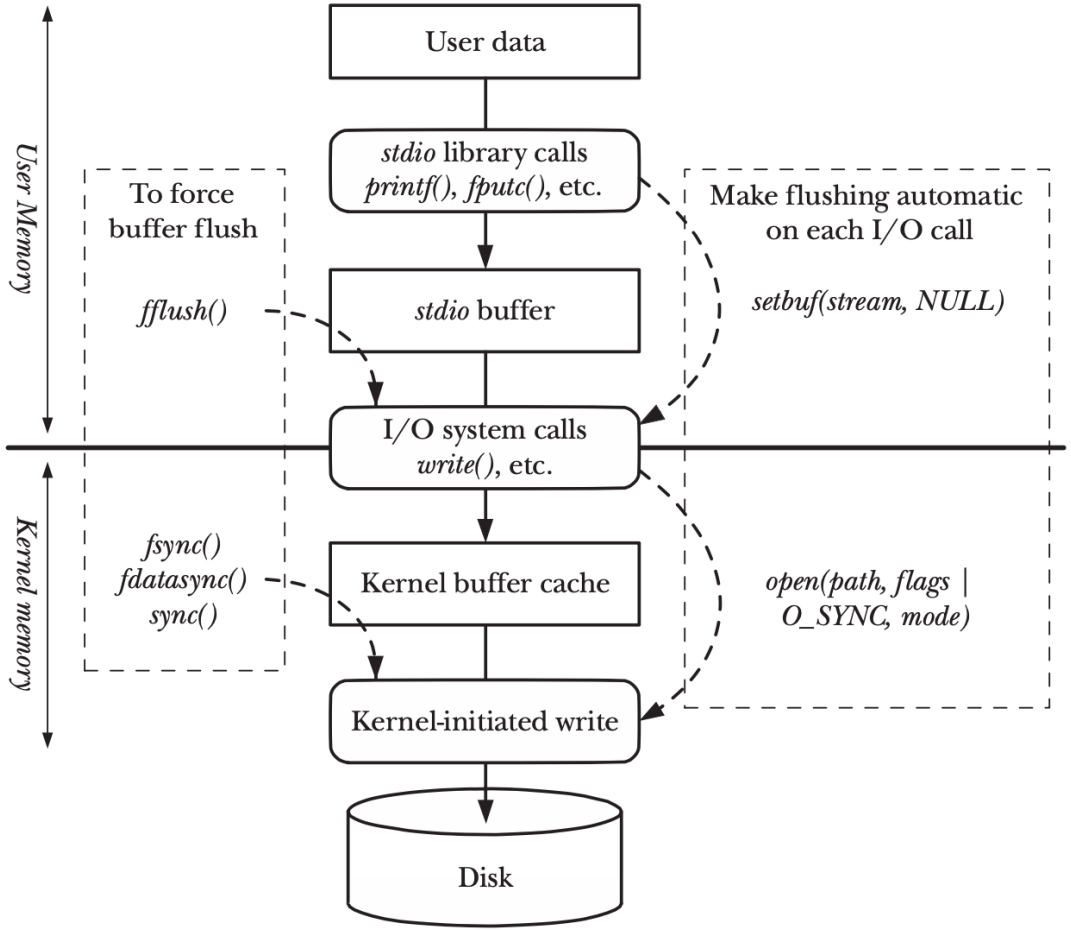
- Les fonctions d'écriture comme `fprintf` n'invoquent pas toujours l'appel système pour écrire dans un fichier.
- Elles écrivent dans un buffer en mémoire
- Quand il est plein, le contenu est réellement écrit dans le fichier.

Ce comportement améliore considérablement les performances pour de nombreuses petites écritures

- Le disque doit être accédé pour chaque opération

Fonctions de la bibliothèque pour les fichiers

Buffering:



Fonctions de bibliothèque pour les fichiers

Buffering:

La taille par défaut est la constante `BUFSIZ`.

Il est possible de définir la taille du tampon :

```
void  setbuf(FILE *stream, char *buf)  ;
```

Il est possible de forcer une écriture dans un fichier :

```
int  fflush(FILE *stream)  ;
```

Note : Quand la sortie standard est sur la console, elle est *nouvellement buffered en ligne*. Un retour chariot force le *flush*.

Pour rendre un flux *new line-buffered*, utilisez la fonction `setvbuf`.

Fonctions de la bibliothèque pour les fichiers

Suppression:

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

Le fichier n'a pas besoin d'être ouvert.

Sous Linux, `remove()` utilise l'appel système :

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

Traitement des erreurs en C

Nous avons vu que les fonctions de bibliothèque signalent une erreur via la valeur de retour.

Exemple:

```
f = fopen("file.txt", "r") ;
if (f==NULL){
    /*Erreur*/
}
```

Avec ce mécanisme, il n'est pas possible de savoir quoi que ce soit sur **la nature** de l'erreur.

- Le fichier n'existe pas ?
- Pas de permission de lecture ?

Gestion des erreurs en C

La bibliothèque standard du C utilise le mécanisme suivant pour préciser la cause des erreurs.

- Chaque programme en C possède la variable globale `int errno`.
- Une fonction de la bibliothèque qui échoue définit `errno` avec un code d'erreur explicatif.
- Si l'appelant, via la valeur de retour, détecte s'il y a eu une erreur
- S'il y a eu une erreur, l'appelant lit le code d'erreur dans `errno`.

Requis :

```
#include <errno.h>
```

Gestion des erreurs en C

La variable globale `errno` est entière et contient un code d'erreur.

- La page de manuel de chaque fonction précise quel code d'erreur elle peut retourner.
- `fopen` peut échouer avec `ENOENT` (fichier inexistant), `EACCES` (permissions insuffisantes) et bien d'autres.

Tous les codes d'erreur sont des constantes définies dans la bibliothèque standard.

- Le programmeur peut comparer `errno` avec les constantes pour identifier l'erreur.

Gestion des erreurs en C

Exemple:

```
FILE * f = fopen("file.txt", "r") ;
if (f==NULL){
    if (errno == ENOENT)
        printf("Le fichier n'existe pas") ;
    else if (errno == EACCES)
        printf("Permissions insuffisantes") ;
    else
        printf("Erreur générique") ;
    retourne 1 ;
}
```

Imprimer une erreur

Des fonctions de bibliothèque existent pour simplifier la gestion des erreurs.

```
#include <stdio.h>
void  perror(const char *s) ;
```

Imprime le message d'erreur pour la valeur actuelle de `errno`, préfixé par la chaîne `s`.

Exemple:

```
FILE * f = fopen('file.txt', 'r') ;
if (f==NULL){
    perror("Erreur") ;
    retournez 1 ;
}
```

Imprime : `Erreur : Aucun fichier ou répertoire de ce type`.

Imprimer une erreur

```
#include <string.h>
char  *strerror(int errnum) ;
```

Renvoie une chaîne de caractères expliquant l'erreur du code `errnum`.

Exemple:

```
FILE * f = fopen('file.txt', 'r') ;
if (f==NULL){
    printf("Impossible d'ouvrir le fichier. Erreur : %s\n", strerror(errno)) ;
    retourne 1 ;
}
```

Imprime `Impossible d'ouvrir le fichier. Erreur : Aucun fichier ou répertoire de ce type`

Limitations

La gestion des erreurs via la variable globale `errno` est une technique problématique, en cas de :

- En cas de signaux (**nous verrons**).
- Heureusement, `errno` est thread safe (**nous verrons**)

La gestion des erreurs via `errno` est considérée comme obsolète.

- Les langages plus modernes utilisent les constructions `try catch`.

Appel système pour un fichier

Les fonctions vues ci-dessus font partie de la bibliothèque C standard.

- Elles utilisent des appels système pour effectuer les opérations requises.
- Les appels système varient en fonction du système d'exploitation

Appels système utilisés

- Linux/POSIX : `open`, `read`, `write`, `lseek`, `close`.
- Windows : `CreateFile`, `WriteFile`, `ReadFile`, `CloseHandle`, `SetFilePointer`.

Appel système par fichier

Nous nous concentrons sur les appels système de Linux.

- Ils sont (très) similaires aux fonctions des bibliothèques, mais sont d'un niveau inférieur.
- Ils peuvent être utilisés sur les systèmes Linux et Posix.
- Ils n'existent pas sous Windows. Ils ne font pas partie de la bibliothèque standard du langage C

System calls	**Library functions**
file descriptor (*int*)	file stream (*FILE **)
open(), *close()*	*fopen()*, *fclose()*
lseek()	*fseek()*, *ftell()*
read()	*fgets()*, *fscanf()*, *fread()* ...
write()	*fputs()*, *fprintf()*, *fwrite()*, ...
–	*feof()*, *ferror()*

Appel système pour les fichiers

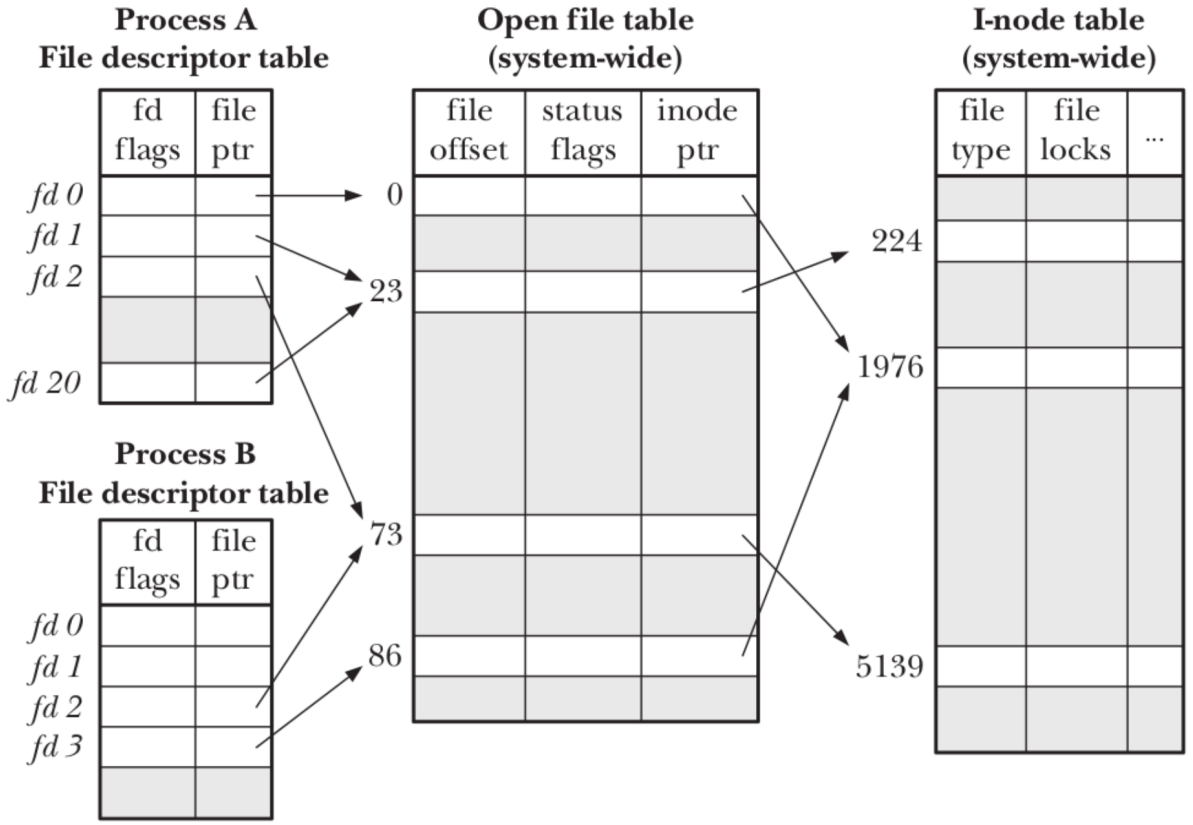
Sous Linux, un fichier ouvert est un *file descriptor*.
Il s'agit d'un nombre entier non négatif
Par convention :

- Entrée standard : descripteur 0
- Sortie standard : descripteur 1
- Erreur standard : descripteur 2

Dans les fonctions de bibliothèque, un fichier ouvert est un `FILE *`. Dans les appels système de Linux, c'est un `int`.

Appel système pour les fichiers

Le système d'exploitation maintient des tables qui font correspondre les *file descriptor* aux fichiers physiques sur le disque (inodes).



Appel système par fichier

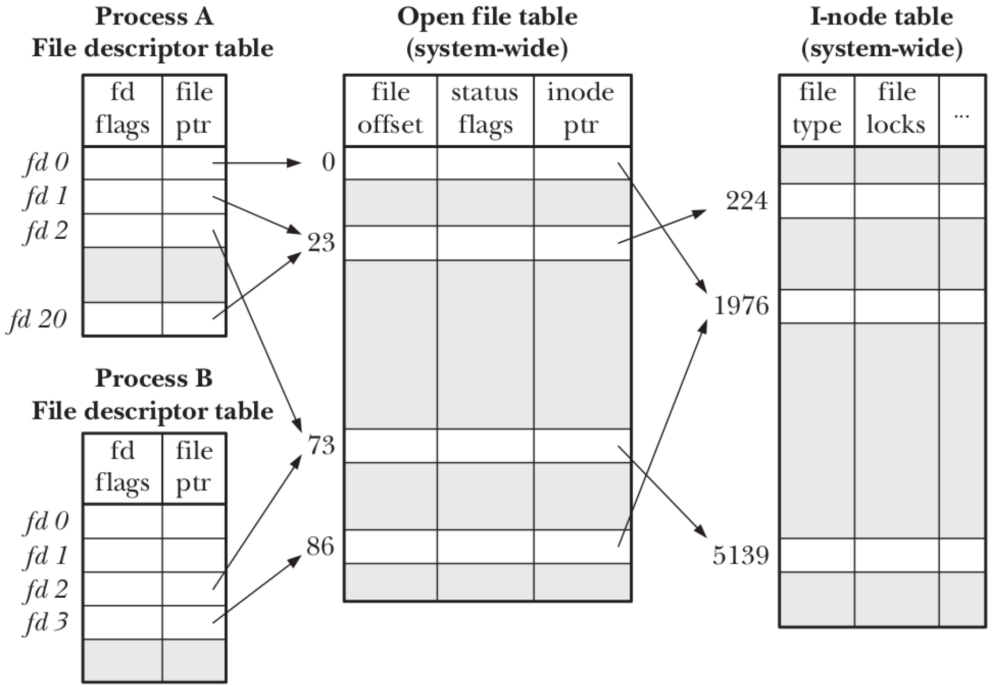
Pour chaque processus, il existe une table contenant le *file descriptor* :

- Il contient une référence à la table générale et des flags

La table générale (une pour l'ensemble du système d'exploitation), contient :

- Mode d'accès : R, W, RW
- File Offset : position du curseur

La table des inodes est simplement une copie en mémoire des inodes pertinents (situés sur le disque).



Appel système par fichier

Ouvrir un fichier

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

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

Ouvre le fichier identifié par le chemin `pathname`.

Appel système pour le fichier

Ouvrir un fichier

Les `flags` déterminent la façon dont le fichier est accédé.

- L'un de `O_RDONLY`, `O_WRONLY,` et `O_RDWR` doit être présent.
- Les autres drapeaux sont :
  - `O_CREAT` crée le fichier s'il n'existe pas.
  - `O_APPEND` ouvre le fichier en mode append.

    Les drapeaux sont ajoutés en utilisant l'opérateur OR bit à bit `|`.

Exemples :

```
int fd = open("file.txt", O_RDONLY)
int fd = open("file.txt", O_WRONLY | O_APPEND) ;
```

Appel système par fichier

Ouvrir un fichier

Si le flag `O_CREAT` est spécifié, le fichier est créé avec les permissions spécifiées dans `mode`.
Il y a 9 flag :

- `S_I[RWX]USR`, `S_I[RWX]GRP`, `S_I[RWX]OTH`.

Rappelez-vous que sous Linux, les fichiers ont 3 types de permissions (Read, Write, Execute), qui peuvent être gérées séparément pour le propriétaire, le groupe et les autres utilisateurs.

Appel système par fichier

Fermeture d'un fichier :

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

Ferme le *déscripteur de fichier*. Le numéro `fd` ne se réfère plus à un fichier ouvert et peut être réutilisé par le système d'exploitation lors des `ouvertures` suivantes.

Appel système par fichier

Lecture d'un fichier :

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

Lit `count` octets de `fd` et les met en mémoire à l'adresse `buf`.
Valeur de retour : le nombre d'octets lus. Il peut être inférieur à `count` si le fichier est épuisé.

- En cas d'erreur -1
- Si EOF 0

Appel système par fichier

Ecriture dans le fichier :

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

Écriture de `count` octets dans `fd` et les prend en mémoire à l'adresse `buf`.
Valeur de retour : le nombre d'octets écrits. Il peut être inférieur à `count` si le disque se remplit.

- En cas d'erreur -1

Appel système par fichier

Valeur de retour :

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

Très similaire à la fonction de la bibliothèque `fseek`.
Réinitialise le fichier descripteur `df` à l'offset `offset` selon la directive `whence` comme suit :

- `SEEK_SET` : `offset` est relatif au début du fichier
- `SEEK_CUR` : le décalage `offset` est relatif à la position actuelle.
- `SEEK_END` : `offset` est relatif au fichier. `offset` doit être négatif.

Appel système par fichier

Exemple : écriture avec appel système

```
const char *str = "Chaîne arbitraire à écrire dans un fichier" ;
const char* filename = "innn.txt" ;

int fd = open(filename, O_RDWR | O_CREAT) ;
si (fd == -1) {
    perror("open") ;
    exit(EXIT_FAILURE) ;
}

write(fd, str, strlen(str)) ;
printf("Fin de l'écriture!\n") ;

close(fd) ;
```

Appel système par fichier

Exemple : équivalent utilisant les fonctions de la bibliothèque

```
const char *str = "Chaîne arbitraire à écrire dans un fichier" ;
const char* filename = "innn.txt" ;

FILE* output_file = fopen(filename, "w+") ;
if (!output_file) {
    perror("fopen") ;
    exit(EXIT_FAILURE) ;
}

fwrite(str, 1, strlen(str), output_file) ; // Vous pouvez utiliser fputs ou fprintf
printf("Fini d'écrire!\n") ;

fclose(output_file) ;
```

System Call per file

```
printf("Hello World\n");
```

ou

```
fprintf(stdout, "Hello World\n");
```

ou

```
write(1, "Hello World\n", 13);
```

sont le meme :)

Fonctions de bibliothèque vs appel système

Les **Fonctions de bibliothèque** sont exécutées en mode utilisateur. Elles n'ont pas de privilèges particuliers.

- Ce sont simplement des fonctions qui facilitent l'utilisation des appels système.

Les **appels système** sont exécutés en mode noyau.

- Ils ont accès à la *mémoire physique*.
- Ils peuvent accéder aux interfaces des périphériques d'I/O

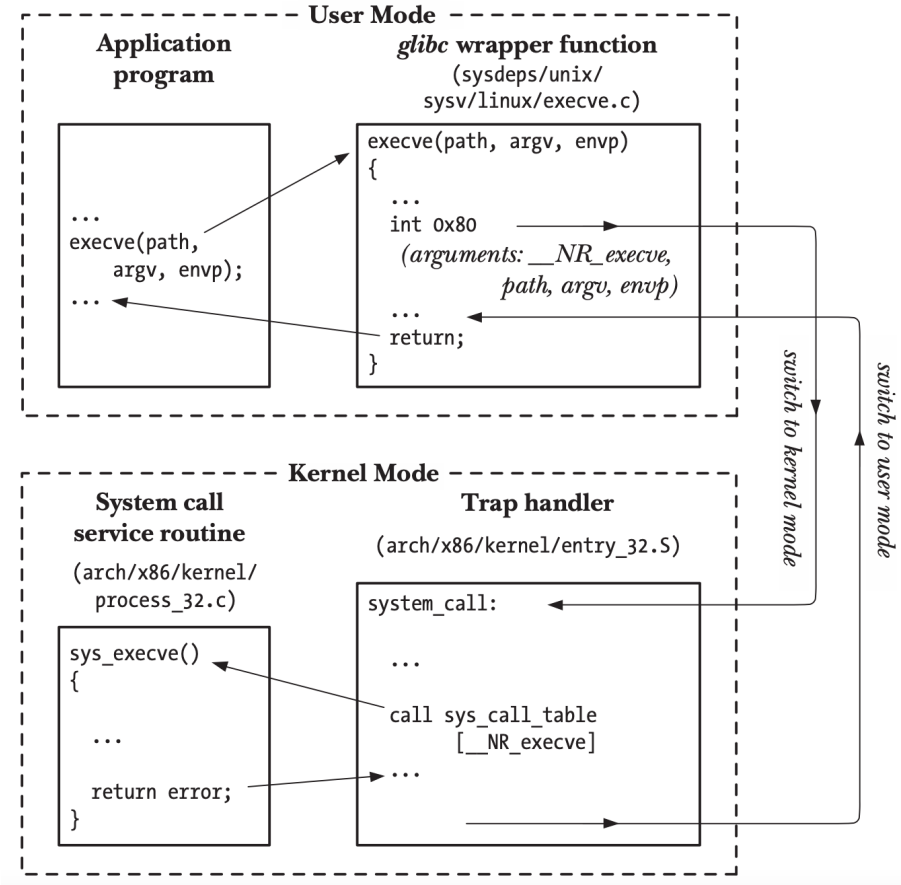
Fonctions de bibliothèque et appels système

Les applications peuvent invoquer à la fois des fonctions de bibliothèque et des appels système.
Si elles veulent utiliser les services du système d'exploitation, il est toujours nécessaire d'utiliser des appels système.

- Les applications le font directement
- Ou les fonctions de bibliothèque invoquées par les applications le font

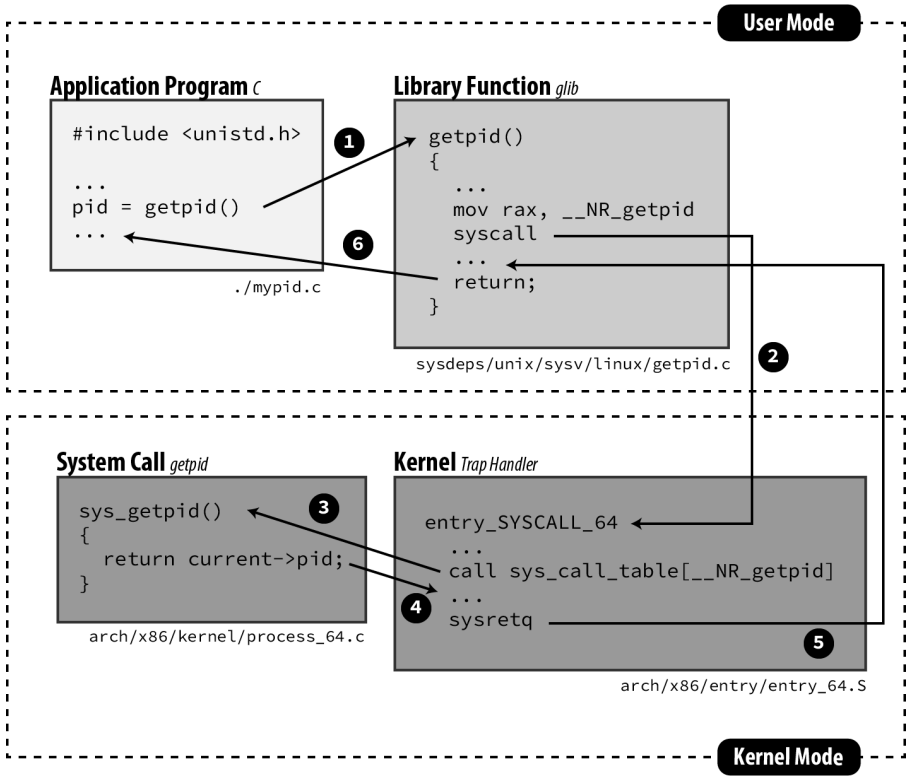
Fonctions de bibliothèque et appel système

Étapes pour invoquer un appel système Via `int 0x80` (méthode classique)



Fonctions de bibliothèque et appel système

Étapes pour invoquer un appel système via `syscall` (méthode moderne)



Fonctions de bibliothèque et appels système

Des outils de débogage existent pour voir quels appels système sont invoqués par un processus.
Cela peut être fait via un profil (par exemple, `valgrind`), un débogueur (par exemple, `gdb`) ou des outils natifs (par exemple, `strace`).

`strace`

Il ne nécessite pas de recompilation. Il fonctionne toujours, même si je n'ai pas la source du programme.
C'est un outil d'OS.
Functionement : `strace command`

Fonctions de bibliothèque et appels système

Exemple: Par défaut, `strace` liste les appels système

```
$ strace pwd
execve("/usr/bin/pwd", ["pwd"], 0x7ffd37cdfc80 /* 72 vars */) = 0
...
getcwd("/tmp", 4096) = 5
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x5), ...}) = 0
write(1, "/tmp\n", 5/tmp) = 5
```

Vous pouvez simplement compter les invocations de l'appel système

```
$ strace -c date
Tue Oct 25 2022, 10:16:48, CEST
% temps secondes usecs/appel appels erreurs syscall
------ ---------- ---------- --------- --------- ----------------
  0.00 0.000000 0 6 lecture
  0,00 0,000000 0 1 écriture
  0,00 0,000000 0 9 fermer
  ...     ...
```

Fonctions de bibliothèque et appel système

`ltrace`

Similaire à `strace`. Vous permet de visualiser les fonctions de bibliothèque utilisées par un processus.

Fonctionnalité : commande `ltrace`.

Note: ne fonctionne pas sur tous les exécutables.

Seulement ceux compilés avec *lazy binding* (option du linker), par défaut jusqu'à Ubuntu 16.

Pour être sûr, compilez avec l'option : `-z lazy`.

Exemple : `gcc sample.c -o myprog -z lazy`

Fonctions de bibliothèque contre appel système

Exemple: Par défaut, `strace` liste les appels système.

```
$ ltrace pwd
...
getcwd(0, 0) = ""
puts("/home/det_user/trevisan
) = 24
...
```

Pour compter les fonctions invoquées :

```
$ ltrace -c date
Tue Oct 25 10:35:00 CEST 2022
% temps secondes usecs/appels fonction
------ ----------- ----------- --------- --------------------
 15.59 0.000543 67 8 fwrite
 13.90 0.000484 60 8 fputc
  8,76 0,000305 305 1 setlocale
  ...
```

Links

Liens

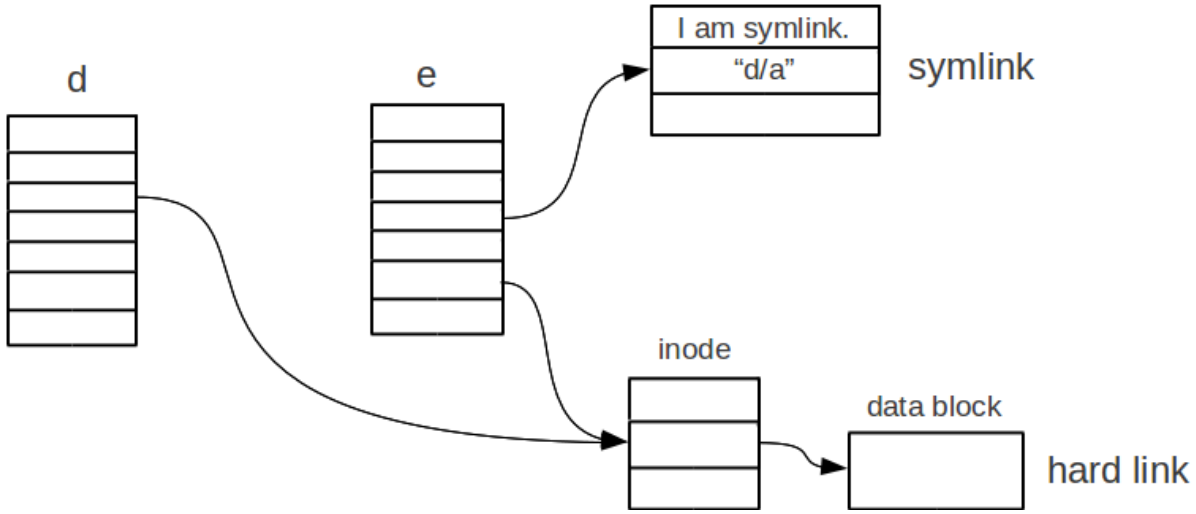
Un lien est un nom supplémentaire pour un autre fichier
Utile pour diverses tâches

- Gestion des fichiers de configuration
- Partage d'informations entre utilisateurs
- Maintenir une structure de fichiers ordonnée

Liens

Il existe deux types de liens :

- **Lien "hard":** l'inode apparaît dans un second répertoire qui pointe vers lui
- **Lien "soft":** est un alias vers un certain chemin



Soft Links

Ce sont des **"shortcuts"** pour un fichier ou un répertoire.

- Si j'ai un gros fichier avec un chemin long et complexe, je peux créer un lien logiciel vers ce fichier dans mon répertoire personnel.
- Si je supprime un lien logiciel, rien ne se passe avec le fichier original.
- Si le fichier d'origine est supprimé, le lien logiciel continue d'exister mais devient **invalide**.
- Si je crée un autre fichier portant ce nom, le lien logiciel redevient valide.

Soft Links

Il s'agit d'un concept Linux

Pour créer un Soft Link, vous utilisez l'appel système :

```
int  symlink(const char *target, const char *linkpath) ;
```

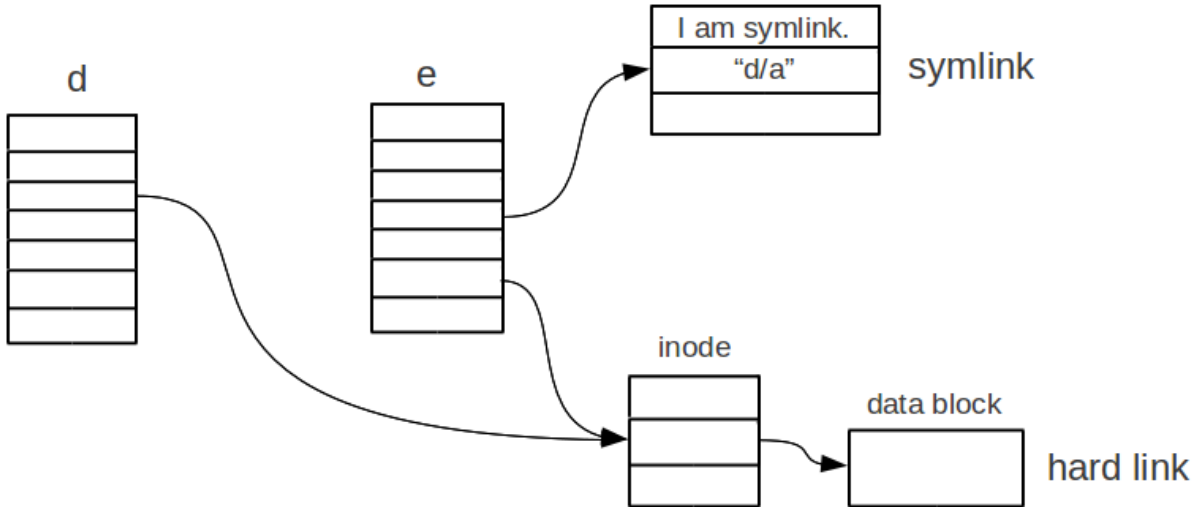
Vous supprimez un Soft Link comme s'il s'agissait d'un fichier normal.

Note: Vous pouvez créer des Soft Links vers des dossiers et vers d'autres disques. Les fonctions de recherche ne doivent pas traverser les Soft Links pour éviter les boucles.

Hard links

Un Hard Link est une référence supplémentaire à un inode.

Le répertoire où il est créé contient une nouvelle entrée qui a le même numéro d'inode



Hard links

Implication :

- Un Hard Link est un lien vers le contenu d'un fichier
- Il ne peut jamais être invalide
- Les hard links et les fichiers originaux ont la même importance et la même nature
- La suppression d'un hard link n'entraîne la suppression du fichier que s'il n'existe pas d'autres hard links (ou la référence originale)

Tâches du système d'exploitation

- Maintenir un *compte de références* pour chaque inode.
- Supprimer un inode et son contenu s'il passe à 0.

Hard links

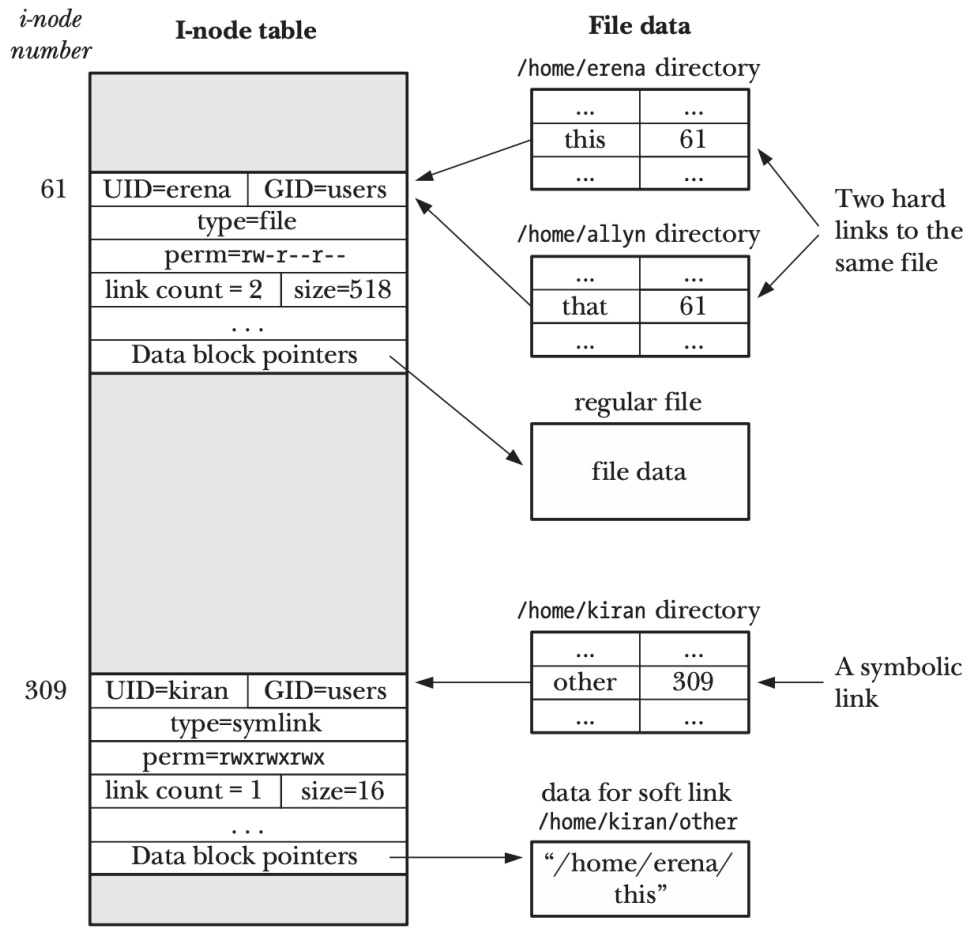
Ils sont créés avec l'appel système :

```
int  link(const char *oldpath, const char *newpath)  ;
```

Note : L'appel système pour supprimer un fichier sous Linux est `unlink`.
En fait, supprimer un fichier signifie décrémenter son *compte de références* (et éventuellement supprimer l'inode).
Vous ne pouvez pas créer des Hard Links vers les répertoires. Conceptuellement erroné. Pourquoi ?

Hard links

Différence entre soft links et hard links



Appel système pour le répertoire

Lecture d'informations sur un répertoire ou un fichier:

```
#include <sys/stat.h>
int  stat(const char *restrict pathname,
          struct stat * statbuf)  ;
```

Renvoie des informations sur `pathname`, les place dans la structure pointée par `statbuf`.

- Passe une variable par référence qui est en fait une valeur de retour.
- Retourne 0 en cas de succès, -1 en cas d'erreur

Appel système pour Directory

La `struct stat` retournée contient les champs suivants :

```
struct stat {
    dev_t st_dev ; /* ID du périphérique contenant le fichier */
    ino_t st_ino ; /* Numéro d'inode */
    mode_t st_mode ; /* Type et mode de fichier */
    nlink_t st_nlink ; /* Nombre de liens durs */
    uid_t st_uid ; /* ID utilisateur du propriétaire */
    gid_t st_gid ; /* ID du groupe du propriétaire */
    dev_t st_rdev ; /* Device ID (si fichier spécial) */
    off_t st_size ; /* Taille totale, en octets */
    blksize_t st_blksize ; /* Taille du bloc pour les E/S du système de fichiers */
    blkcnt_t st_blocks ; /* Nombre de blocs de 512B alloués */

    struct timespec st_atim ; /* Heure du dernier accès */
    struct timespec st_mtim ; /* Heure de la dernière modification */
    struct timespec st_ctim ; /* Heure du dernier changement d'état */
} ;
```

Appel système pour le répertoire

Le champ `mode_t` indique si le fichier est sous la forme d'un *masque binaire*.
Vous pouvez utiliser les macros suivantes pour tester facilement le `mode_t`

- `S_ISREG(m)` : Vrai si fichier régulier
- `S_ISDIR(m)` : Vrai si fichier répertoire
- `S_ISLNK(m)` : Vrai si le fichier est un lien symbolique

Appel système pour le répertoire

```
#include <stdio.h>
#include <stdlib.h> // Nécessaire pour la sortie
#include <sys/stat.h> // Nécessaire pour stat
int  main  (int argc, char * argv[])
{
     struct stat buf ;
     if  (argc!=2){
          printf("Spécifiez un chemin d'accès") ;
          return  1  ;
     }

     if  (stat(argv[1], &buf) < 0) {
          printf("Impossible de lire les informations du fichier") ;
          exit  (1) ;  /* Fin immédiate du programme avec le code 1 */
     }

     if  (S_ISREG(buf.st_mode))
          printf("%s : file\n", argv[1]) ;
     else  if  (S_ISDIR(buf.st_mode))
          printf("%s : répertoire\n", argv[1]) ;
     else  if  (S_ISLNK(buf.st_mode))
          printf("%s : lien symbolique\n", argv[1]) ;
     else  if
          printf("%s : autre file\n", argv[1]) ;

     return  0  ;
}
```

Appel système pour le répertoire

Création de répertoire

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

Suppression de répertoire

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

`mode` a le même rôle que dans `open`

Valeur de retour 0 en cas de succès -1 en cas d'erreur

Appel système pour Directory

Pour lister le contenu d'un répertoire, vous pouvez utiliser les appels système `open` et `getdents` et la `struct linux_dirent`.

1. Ouvrez un répertoire comme s'il s'agissait d'un fichier
   `intfd = open('path', O_RDONLY | O_DIRECTORY);`
2. Vous lisez des lots de `struct linux_dirent`
   `int nread = syscall(SYS_getdents, fd, buf, BUF_SIZE);`

Cette méthode est difficile, peu pratique et non portable.
Vous utilisez toujours les fonctions de la bibliothèque POSIX pour lire le contenu d'un répertoire

Fonctions de la bibliothèque des répertoires

Pour lister le contenu d'un répertoire, on utilise les fonctions de la bibliothèque

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

Ils fonctionnent sur les systèmes POSIX.

Sous Windows, vous utilisez `FindFirstFile()` et `FindNextFile().`

Fonctions de la bibliothèque de répertoires

Open: un répertoire, avant d'être lu, doit être ouvert avec `opendir`.

Elle retourne un pointeur sur `DIR *` si l'ouverture est réussie, sinon `NULL`.

Un `DIR *` est l'équivalent de `FILE *` pour les répertoires

```
DIR * d ;
d = opendir('/path/') ;
si (d!=NULL)
  ...
```

Fonctions de la bibliothèque de répertoires

Lister le contenu: on utilise la fonction `readdir`, qui retourne **une** `struct dirent *`. Elle doit être invoquée jusqu'à ce qu'elle renvoie `NULL`.

```
struct dirent * entry ;
while ((entry = readdir(d)) != NULL)
       printf(" %s\n", entry->d_name) ;
```

Fonctions de la bibliothèque des répertoires

Une `struct dirent` contient les champs

```
struct dirent {
   ino_t  d_ino ;  /* numéro d'inode */
   char  d_name[256] ;  /* nom de fichier */
   ...
} ;
```

Fonctions de la bibliothèque de répertoires

Un répertoire doit être fermé pour libérer les ressources qui lui sont associées.

```
int r = closedir(d);
```

Retourne 0 en cas de succès, sinon 1.

Fonctions de la bibliothèque du répertoire

```c
#include <stdio.h>
#include <stdlib.h> // Nécessaire pour la sortie.
#include <sys/stat.h> // Nécessaire pour stat
#include <dirent.h> // Nécessaire pour struct dirent *
int  main  (int argc, char * argv[])
{
     struct stat buf ;
     struct dirent *dirp ;
     DIR *dp ;

     if (argc!=2){
          printf("Spécifiez un nom de chemin") ; exit (1);}
     if (stat(argv[1], &buf) < 0) {
          printf("Cannot read file\n information") ; exit (1);}
     if (!S_ISDIR(buf.st_mode)){
          printf("%s must be a directory\n", argv[1]) ; exit(1);}
     if ( (dp = opendir(argv[1])) == NULL) {
          printf("%s impossible à ouvrir\n", argv[1]) ; exit (1);}

     while ( (dirp = readdir(dp)) != NULL)
       printf("%s\n", dirp->d_name) ;

     closedir(dp) ;

     return 0 ;
}
```

Commandes Bash pour les liens, les répertoires

Ce sont des programmes préinstallés qui facilitent l'utilisation des appels système pour les tâches courantes.

Commandes pour les liens:

```
ln dst src
```

Crée un lien vers le chemin `dst` vers un chemin `src` existant.

- Par défaut, crée un lien dur
- L'option `-s`
- Supprime les liens avec `rm`
- Vous pouvez aussi utiliser les commandes `link` et `unlink` plus simples

RAID

Problemes avec les disques

Dans les grands systèmes, une seule machine possède de nombreux disques.

- Jusqu'à 100 disques sur une seule machine
- Ils stockent d'énormes quantités de données

Il y a souvent des serveurs dédiés au *stockage*.

- Les ordinateurs accèdent aux données **via le réseau**.
- Via des protocoles de réseau

Exemple d'un serveur pour le *stockage*.

Exemple de serveur pour le *stockage*.

Des techniques sont nécessaires pour gérer les **défaillances**

- Un disque a une chance sur cent de se casser tous les mois.
  - Chiffre réel pour les disques magnétiques
- Si j'ai 100 disques, j'ai en moyenne une panne par mois.
- Il est impensable de perdre des données dans des systèmes professionnels.

Les techniques permettant d'augmenter les **performances** sont souhaitables

- Si 100 disques sont correctement utilisés en parallèle, ils peuvent multiplier la vitesse du système par 100.

Définition RAID

L'objectif des techniques **RAID** est de répondre aux problèmes de performance et de fiabilité.

- Proposé en 1988 par David A. Patterson (et d'autres) dans le document *A Case for Redundant Arrays of Inexpensive Disks (RAID)*.
- Famille de méthodes d'organisation des données sur des **batteries de disques**.

Concept RAID

Basé sur le ***striping***, c'est-à-dire la répartition des données sur N disques

- Au niveau **bit** : le i-ème disque contient les $n \mid n \ mod \ N = i$ bits.
- Au niveau **bloc** : le i-ème disque contient les $n \mid n \ mod \ N = i$ blocs.
- Cela améliore les performances en permettant des lectures parallèles

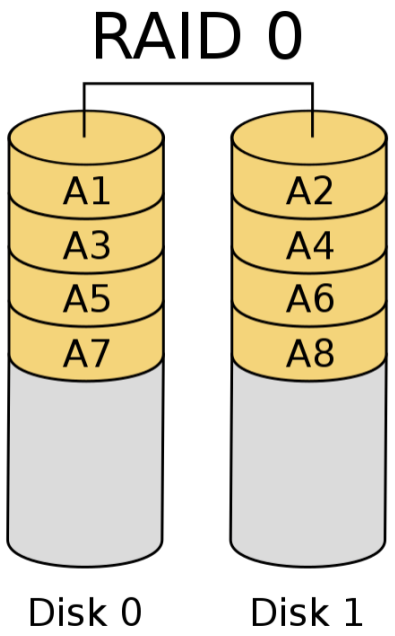
Eventuellement avec l'ajout de **codes de parité**.

- Pour être *tolérant aux défauts*.
- Aucune donnée n'est perdue en cas de défaillance d'un disque.

Niveaux RAID

Différentes configurations ou modèles de disques possibles. Ils diffèrent

- Selon qu'ils offrent des performances ou une fiabilité accrues
- Nombre minimal de disques requis
- Robustesse en cas de défaillances multiples



RAID 0 - Sectionnement

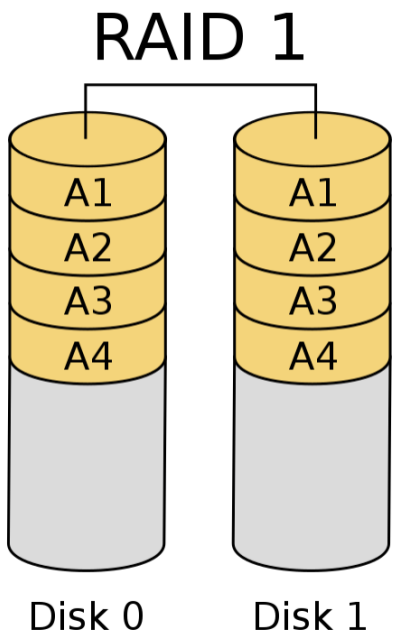
Les données sont **split** entre les disques par striping (au niveau des blocs généralement)

Nombre minimum de disques : 2.

Avantages : Vitesse élevée grâce aux accès parallèles

Inconvénients : Diminue la fiabilité du système !

- Avec une seule panne, j'ai perdu toutes mes données !



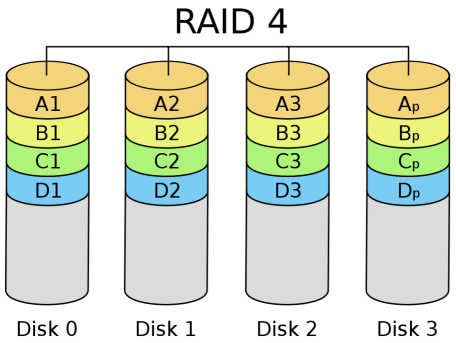
RAID 1 - Miroir

Les données sont **répliquées** sur plusieurs disques.

Nombre minimum de disques : 2.

Avantages : Avec N disques, résiste à $N - 1$ pannes

Inconvénients : Faible vitesse d'écriture limitée par le disque le plus lent



RAID 4 - Disque de parité

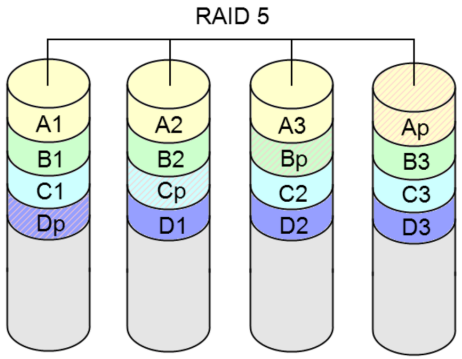
Le disque N stocke les données de parité sur le disque $N-1$.

Nombre minimum de disques : 3. Deux de données plus parité

Avantages : Résiste aux pannes et permet des lectures parallèles

Inconvénients : Écriture lente. Nécessité de calculer et d'écrire la parité

RAID 5 - Parité distribuée



Comme le RAID 4, mais les codes de parité sont distribués sur tous les disques de manière égale.

Nombre minimum de disques : 3.

Avantages :
Résiste aux pannes
Ecritures plus rapides que le RAID 4

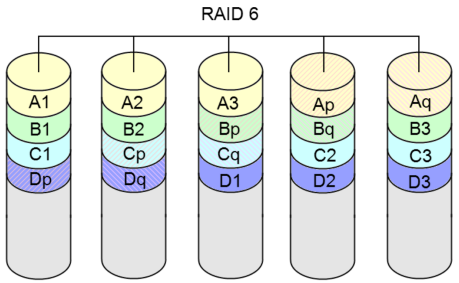
- Pas besoin d'accéder **toujours** au disque de parité.

Avantages:
Écritures toujours lentes (à cause de la parité)

Très utilisé dans les systèmes réels

RAID

RAID 6 - Double parité distribuée



Les codes de parité sont stockés deux fois. Parmi tous les disques

Nombre minimum de disques : 4.

Avantages : Résiste à **deux pannes**

Inconvénients : Ecriture très lente (due à la double parité)

Très largement utilisé dans les systèmes réels

Conclusions

Les schémas RAID améliorent les performances et la fiabilité lorsque vous avez de nombreux disques sur une machine.

Ils ne protègent pas contre une panne complète de la machine

- Temporaire : panne de courant
- Permanente : défaillance de la carte mère

Inacceptable pour les services *critiques*.

Les techniques RAID ne sont pas évolutives :

- Il y a un nombre maximum de disques qui peuvent être connectés à une machine.
- Le bus PCI a une limite

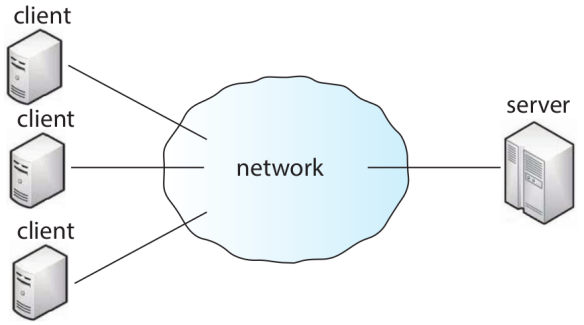
Pour les **très grands** systèmes, des **systèmes de fichiers distribués** sont utilisés.

Systèmes de fichiers distribués

Systèmes de fichiers en réseau

Il est possible d'utiliser un système de fichiers situé sur une autre machine.

- Généralement un serveur de stockage dédié
- Des protocoles dédiés sont utilisés
  - **Network File System (NFS)** : le plus utilisé et le plus flexible.
  - **Samba** : Microsoft
  - **File Transfer Protocol (FTP)** : Remember me ? :)



Définition Systèmes de fichiers distribués

Un **Système de fichiers distribués** est un système de fichiers qui réside sur plusieurs disques sur différentes machines.

- Il nécessite un logiciel **orchestrateur**.
- pour que l'utilisateur puisse l'utiliser comme un seul FS.

Un FS distribué :

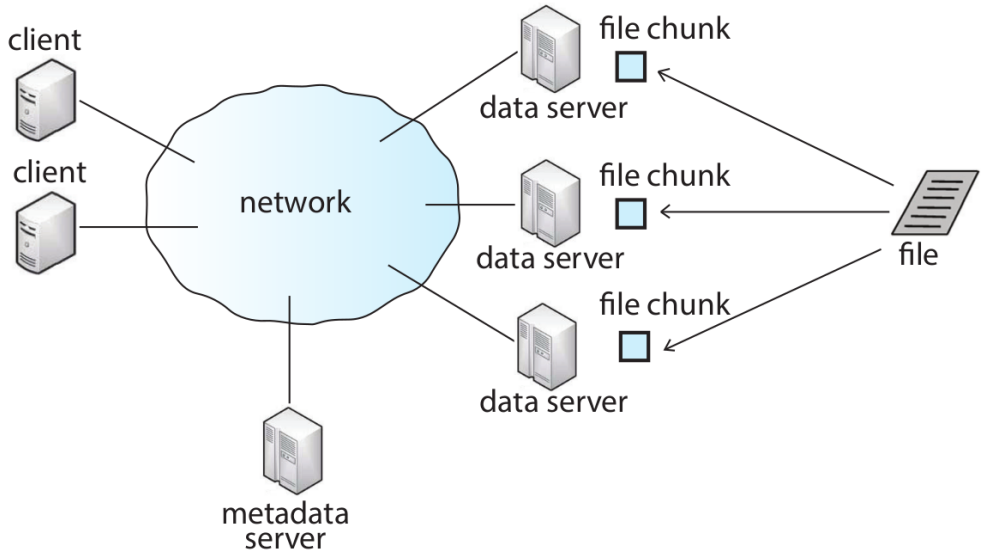
- Est perçu par les utilisateurs comme un seul **grand et fiable** FS.
- On y accède généralement comme à un disque réseau (c'est un système de fichiers réseau).

Système de fichiers distribué

Définition

Basé sur le modèle **client-serveur**.

- Le client consulte le serveur de métadonnées pour répertorier les répertoires et obtenir des informations sur les fichiers.
- Le client accède au contenu d'un ou plusieurs serveurs de données.



File System Distribuiti

Tecnologie per FS distribuiti

I FS distribuiti si installano con **software di orchestrazione** dedicati

- Organizzano i dati nei vari dischi e nodi
- Replicano i dati per aumentare le prestazioni
- Recuperano i dati quando un utilizzatore vi accede

Exemple : Hadoop Distributed File System

Fait partie de la suite Hadoop pour le Big Data. Il s'agit d'un FS distribué

- Il est installé sur un **cluster** (ensemble) de serveurs/nœuds.
- Les **Noeuds de nom** ont l'index des **fichiers**.
- Les **Nœuds de données** stockent le contenu des **fichiers**.
- Tout est répliqué N fois