

# Stockage et systèmes de fichiers

Guillaume Salagnac

Insa de Lyon – Informatique

# Systèmes de fichiers : pour quoi faire ?

## Persistence

- conserver les données même quand la machine est éteinte
- dissocier les données du processus qui les a créées
- ▶ stockage en dehors de la mémoire principale (RAM)
- ▶ notion de **fichier non structuré** = une séquence d'octets

## Organisation

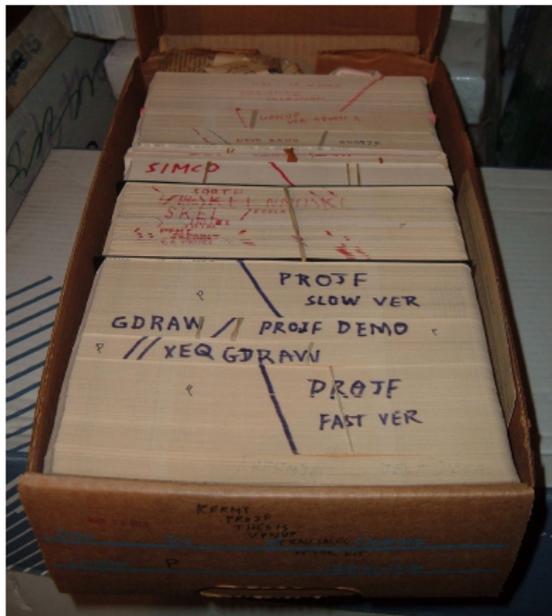
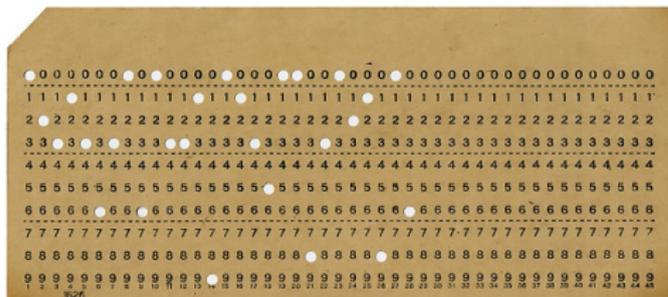
- identifier chaque fichier par un **nom** intelligible
- grouper des noms ensemble pour **ranger** les fichiers
- ▶ notion de **répertoire** (= **dossier**)
- ▶ hiérarchie **arborescente** de répertoires et sous-répertoires

## Abstraction

- masquer les détails technologiques : **disque**, flash, réseau
- ▶ architecture logicielle en **couches**

**Bonus** : protection, performance, robustesse

# Vocabulaire : vous avez dit «fichier» ?

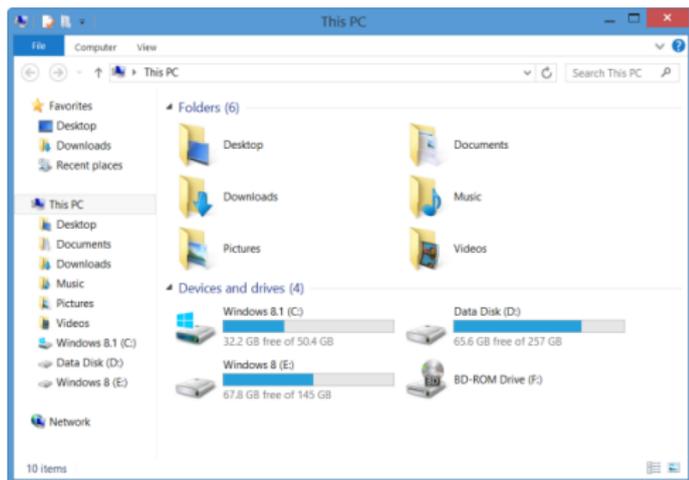


- VF fichier = VO file

# Vocabulaire : vous avez dit «dossier» ?



≈ 1960

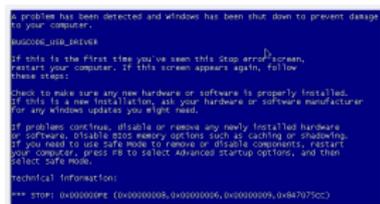


≈ 2015

- VF répertoire (dossier) = VO *directory* (*folder*)

# Systèmes de fichiers : pourquoi c'est difficile

**Non-volatilité** : chaque écriture sur le disque est définitive



VS



► comment résister aux pannes et aux plantages ?

**Latence** : un accès disque  $\approx$  cent mille accès mémoire !

DRAM  $\approx$  50ns      Flash  $\approx$  50 $\mu$ s      disque  $\approx$  5ms

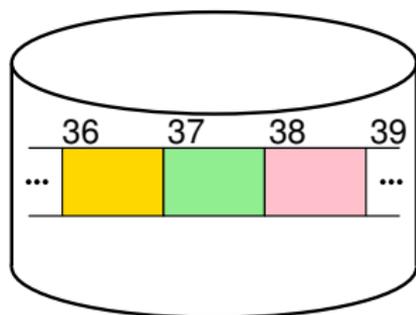
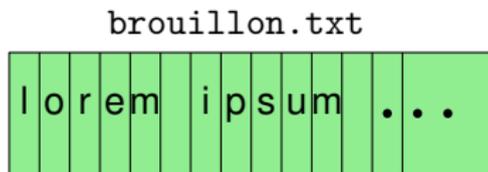
**Taille** : 100 $\times$ , 1000 $\times$  plus vaste que mémoire

► comment offrir une performance acceptable ?

# Plan

1. Introduction : définitions, vocabulaire
2. Interface utilisateur : fichiers, répertoires, volumes
3. Implémentation et interface vers le matériel

# Positionnement



- fichier : conteneur nommé pour une séquence d'octets
  - application interprète «librement» le contenu binaire
  - convention : suffixe du nom (.mp3, .jpg...) = **format** des données
- disque : tableau de **secteurs** numérotés
  - taille typique : un secteur = 512B ou 4kB
  - unité de transfert atomique entre disque et RAM = 1 secteur

## Rôle du système de fichiers (en VO File system = FS)

Lors de chaque accès à un fichier, le FS doit traduire une paire **nom+position** vers une «adresse disque» **n° secteur+offset**

## Interface utilisateur : appels système

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

- ▶ Lire *count* octets dans le fichier *fd* et les écrire en mémoire à l'adresse *buf* (donc de *buf* jusqu'à *buf+count-1*)

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

- ▶ Lire en mémoire les *count* octets commençant à l'adresse *buf*, et les écrire dans le fichier *fd*

Remarque : pas besoin de préciser systématiquement le nom du fichier accédé (ni l'offset dans le fichier)

- le numéro *fd* est un **file descriptor** attribué par le noyau
  - `int open(char *pathname, int flags)` rend un *fd*
- notion de **position courante** dans chaque fichier ouvert
  - `off_t lseek(int fd, off_t offset, int whence);`
- ▶ liste des fichiers ouverts (et positions) dans le PCB

## Opérations disponibles sur un fichier

- créer un nouveau fichier : `creat()`
- supprimer un fichier : `unlink()`
- ouvrir : `open()`
- fermer : `close()`
- lire  $n$  octets depuis la position courante : `read()`
- écrire  $n$  octets à la position courante : `write()`
- se repositionner dans le fichier : `lseek()`
- ajouter  $n$  octets en fin de fichier : `append()`
- lire les méta-données : `stat()`
- changer les méta-données : `chmod()`, `utime()`
- renommer un fichier : `rename()`
- ...

Attention, piège : renommer  $\neq$  déplacer ! (cf plus tard)

# Organisation des fichiers en dossiers

**Objectif** : pouvoir retrouver un fichier à partir de son nom

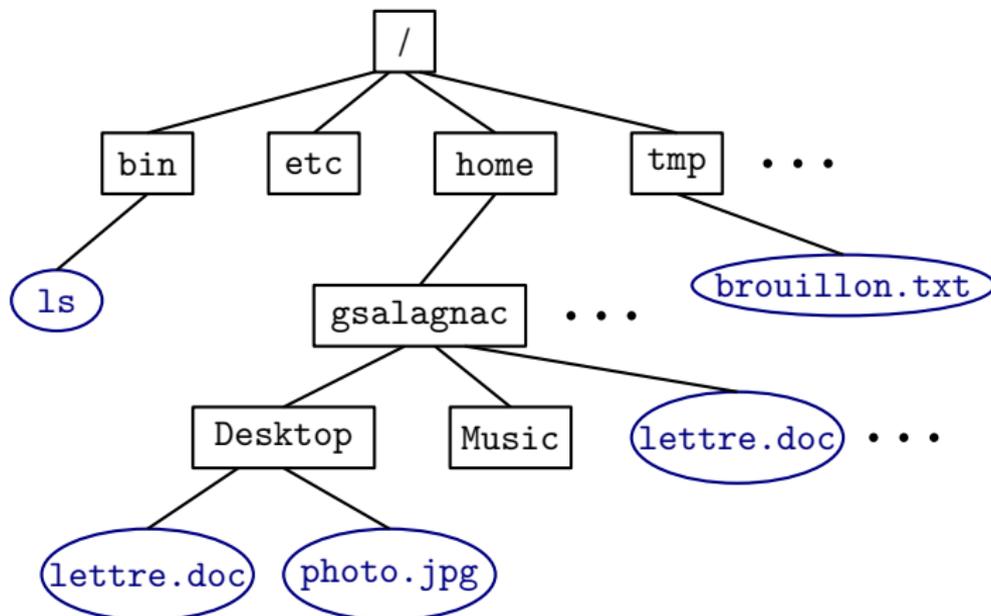
**Solutions historiques** :

- un seul répertoire contenant tous les fichiers ?
  - structure fixe à 2, 3, N niveaux ?
- ▶ inconfortable à utiliser, et inefficace à l'exécution

**Approche moderne** : structure de nommage récursive

- chaque dossier peut contenir **fichiers et sous-dossiers**
- le système de fichiers forme un **arbre**
- implem : un répertoire = un fichier spécial
  - contient la liste de ses sous-dossiers et fichiers
  - appels système idoines : `opendir()`, `readdir()`...

# Arborescence de fichiers Unix



Exemple de **chemin absolu** :

`/home/gsalagnac/Desktop/photo.jpg`

# Arborescence Unix : noms spéciaux

## Noms spéciaux du système de fichiers

- racine du système de fichiers `/`
- répertoire courant `.`
- répertoire parent `..`

## Mécanisme similaire dans le shell

- mon répertoire personnel `~`
- joker `*.txt` pour dire «tous les noms qui finissent par `.txt`»

## Commandes shell

- `cd truc` pour «aller» dans le répertoire *truc*
  - exemples `cd /` `cd /home/gsalagnac/Desktop` `cd ..`
- `ls` pour lister le contenu du répertoire courant

# Répertoire courant et chemins relatifs

Chaque processus a un **current working directory** CWD

- appel système `chdir()` pour changer de CWD
- `cd truc` demande au shell de faire un `chdir("truc")`
- implem : PCB contient un champ pour le CWD

Résolution des noms **par le noyau**

- chemin commençant par un / ► **chemin absolu**
  - interprété en partant de la racine
  - par exemple `/usr/bin/emacs`
- chemin commençant par autre chose ► **chemin relatif**
  - interprété en partant du CWD
  - par exemple `Photos` qu'on peut écrire aussi `./Photos`
  - par exemple `../../Documents/lettre.doc`
- attention : à ne pas confondre avec les raccourcis de syntaxe implémentés dans votre shell (`~`, `jokers...`)

## Notion de «montage»

- plusieurs supports : disques, DVD, clé USB, réseau...
  - une arborescence sur chaque support : notion de **volume logique**

vs

- une seule arborescence de fichiers dans le système
  - une seule racine / dans le système

Problème : comment accéder à tous ces fichiers à la fois ?

Solution : **inclure** les différents volumes dans une unique vue

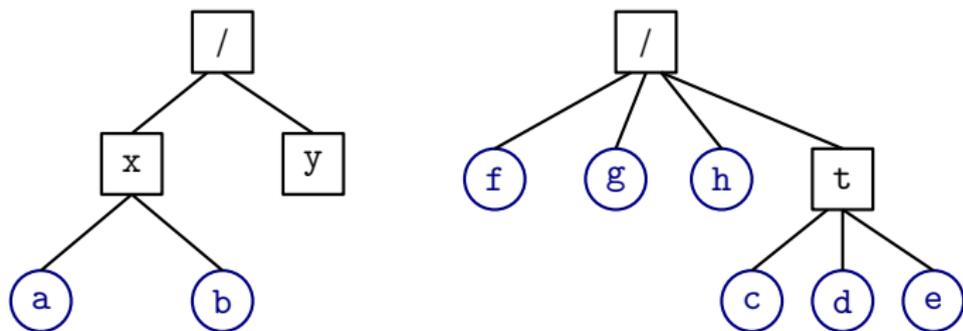
**Définition : montage d'un volume logique**

**Monter** un volume dans un dossier = masquer le contenu de ce dossier et le remplacer par l'arborescence du volume

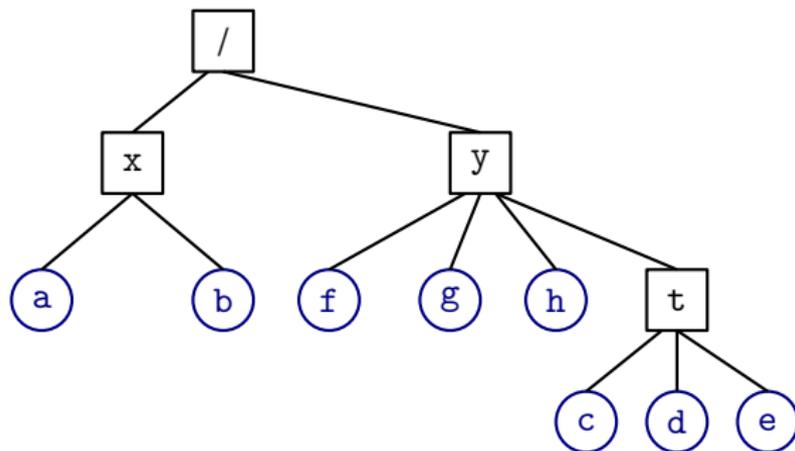
- Unix : tout répertoire peut servir de **point de montage**
- Windows : 26 points de montage possibles A: B: ... Z:

## Montage : illustration

Avant :



Après :



# Montage : remarques

## Unix

- commandes `mount` et `umount`
  - exemple : `mount -l` pour lister les montages

## Démontage

- aussi appelé «retirer le périphérique en toute sécurité»
- demande au noyau de
  - finir toutes les écritures en cours
  - refuser tous les nouveaux accès
  - ensuite, dissocier le sous-arbre de son point de montage
- on peut ensuite débrancher la clé, éjecter le CD...

## Remarque : déplacer ≠ renommer ?

- sur un même volume ► simple modification de répertoires
- entre deux volumes ► il faut copier les données !

# Plan

1. Introduction : définitions, vocabulaire
2. Interface utilisateur : fichiers, répertoires, volumes
3. Implémentation et interface vers le matériel

# Systèmes de fichiers : pourquoi c'est difficile

**Non-volatilité** : chaque écriture sur le disque est définitive



VS



► comment résister aux pannes et aux plantages ?

**Latence** : un accès disque  $\approx$  cent mille accès mémoire !

DRAM  $\approx$  50ns      Flash  $\approx$  50 $\mu$ s      disque  $\approx$  5ms

**Taille** : 100 $\times$ , 1000 $\times$  plus vaste que mémoire

► comment offrir une performance acceptable ?

# Profils d'accès

## Accès séquentiel

- contenu du fichier traité au fur et à mesure
- scénario le plus courant
- exemples : lecture de film, sauvegarde d'un document

## Accès arbitraire (VO *Random Access*) :

- contenu du fichier traité dans le désordre
- exemples : bases de données, swap de mémoire virtuelle

## Remarques :

- lire un certain secteur  $\approx 5\text{ms}$
- lire un certain secteur et les  $n$  suivants  $\approx 5\text{ms} + n \times \varepsilon$

# Formulation du problème

## Objectifs du système de fichiers

- pouvoir retrouver un fichier à partir de son nom
- accéder efficacement aux données (séquentiel **et** arbitraire)
- gérer l'espace libre : allocation et désallocation des secteurs

## Hypothèses

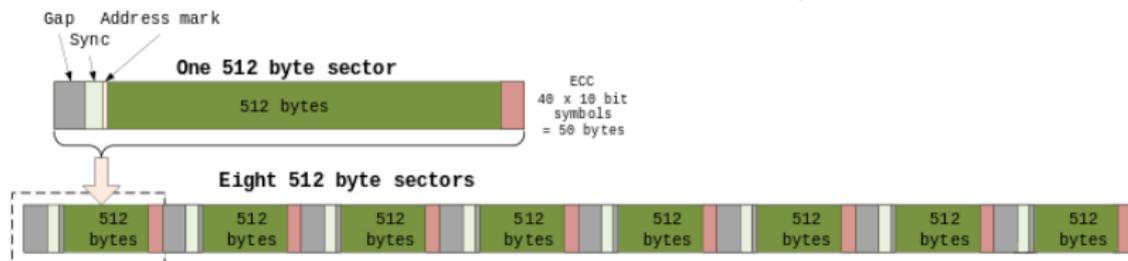
- granularité d'accès par les applications : 1 octet
- granularité de stockage et de transfert : 1 secteur
- méta-données uniquement stockées sur le disque

## Observations empiriques

- grande majorité de petits fichiers ( $\leq 10\text{kB}$ )
- mais : espace surtout occupé par les gros fichiers (qq MB)
- et parfois quelques très très gros fichiers ( $\geq 10\text{GB}$ )

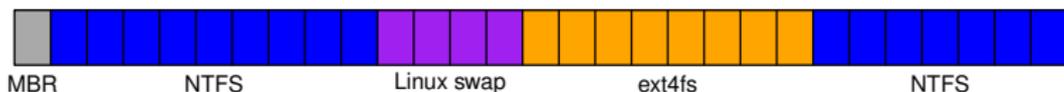
# Notion de «formatage»

**Secteur** = unité de lecture ou écriture atomique

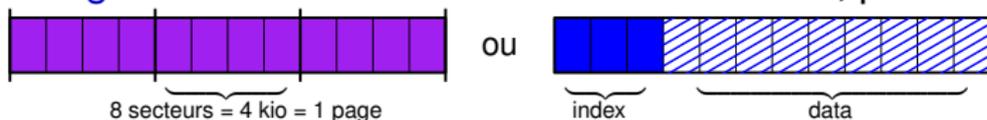


**Partitionnement** = découpage du disque en **volumes**

- un volume = un même *File System*
- fichiers (NTFS, FAT, ext4) vs swap vs DBFS vs ...



**Formattage** d'un volume = écriture d'un FS «vide», par ex :



**Index** = table des matières. typiquement exprimé en termes de numéros de **blocs** AKA clusters. 1 bloc =  $2^n$  secteurs

# Approche 1 : allocation contiguë

Idée : un fichier = une suite de blocs consécutifs

- info nécessaire = n° du 1<sup>er</sup> bloc, taille du fichier



index = pointeur vers le dossier racine

## Avantages

- implem simple, accès séquentiel et arbitraire efficaces

## Inconvénients

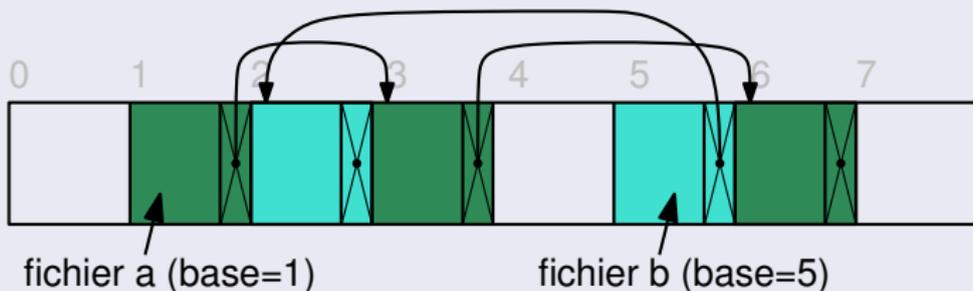
- obligation de connaître la taille du fichier à l'avance
- suppression ► fragmentation (externe)

► parfait pour supports en lecture seule : CD, DVD

# Approche 2 : liste chaînée

Idée : un fichier = une liste chaînée de blocs

- premier mot du bloc = indique le n° du bloc suivant
- info nécessaire = n° du 1<sup>er</sup> bloc



## Avantages

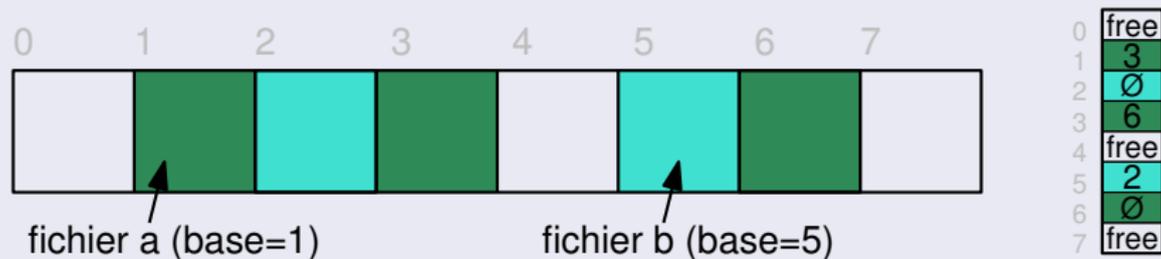
- taille dynamique, pas de fragmentation, accès séquentiel

## Inconvénients

- accès arbitraire inefficace
- surcoût de stockage des pointeurs
- fragile en cas de bug / crash

Idée : séparer le chaînage et le contenu des fichiers

- FAT = tableau avec 1 case par bloc
- chaque case = n° du bloc suivant (dans le même fichier)



**Index** = pointeur sur le dossier racine + File Allocation Table

## Avantages

- FAT mise en cache en RAM ► accès arbitraire OK
- meilleure robustesse
  - redondance : plusieurs exemplaires sur le disque
  - données vs méta-données

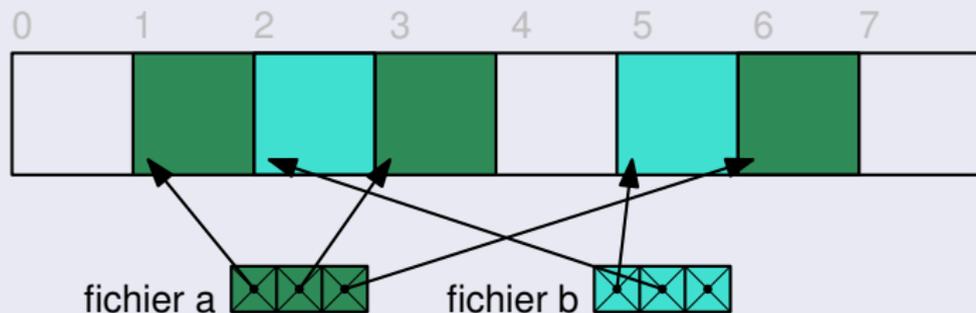
## Inconvénients

- consommation mémoire

## Approche 3 : fichiers indexés

Idée : associer à chaque fichier un tableau de  $n^{\text{os}}$  de blocs

- **inode** = «index-node»
- alloué (dans l'index) lors de la création du fichier
- vs secteurs alloués au fur et à mesure de la croissance



### Avantages

- accès séquentiel et arbitraire efficaces

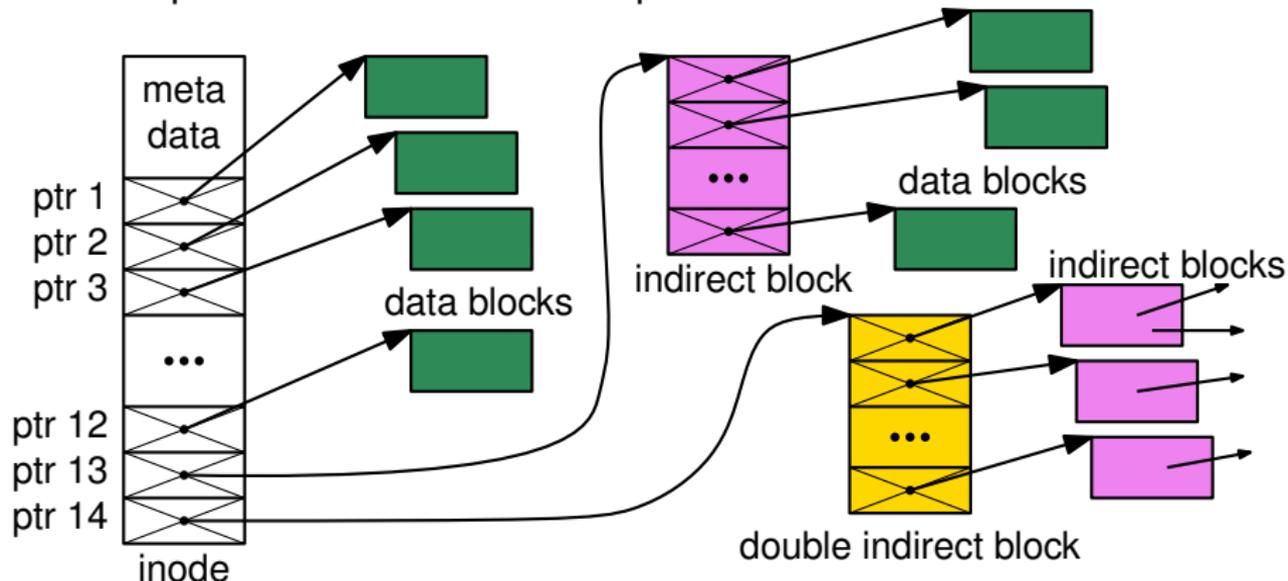
### Inconvénients

- chaque inode doit être contigu : problème de **fragmentation** !



Idée : organiser l'index lui-même comme un arbre

- une seule taille d'inode : métadonnées +  $N$  pointeurs
- 12 premiers pointeurs ► 12 premiers blocs du fichier
- 13<sup>e</sup> pointeur ► un tableau de pointeurs (*indirect block*)
- 14<sup>e</sup> pointeur ► un tableau de pointeurs vers des *indirect blocks*



# Indexation multi-niveaux Unix : remarques

## Avantages

- implem simple, accès efficace aux petits fichiers
- taille max des fichiers bornée, mais grande
  - ext4fs : 15<sup>e</sup> pointeur vers un *triple indirect block*

## Inconvénients

- nombre d'accès au pire cas pour une lecture ?
  - pire surcoût en espace ?
- ▶ systèmes de fichiers modernes = hybrides
- allocation contiguë pour petits fichiers
  - indexation multi-niveaux pour gros fichiers

# Plan

1. Introduction : définitions, vocabulaire
2. Interface utilisateur : fichiers, répertoires, volumes
3. Implémentation et interface vers le matériel

# Systèmes de fichiers : à retenir

## Rôles du système de fichiers

- persistance, organisation, gestion du stockage, protection...
- interface : appels système `open()`, `read()`, `write()`...

## Notions clés

- fichier = séquence d'octets, identifié par un nom
- dossier = contient des fichiers et d'autres dossiers
- volume = arborescence occupant tout un support

## Implémentation

- dossier = fichier spécial contenant des noms
- fichier = éparpillé sur plusieurs secteurs du disque
- inode = index des secteurs composant le fichier