Systèmes de fichiers

La notion de fichier et de répertoire

La notion de système de fichiers

mount

Partage et protection

Représentation des fichiers

Représentation des répertoires

Espace libre

La notion de fichier

Selon les systèmes, un fichier peut avoir différente structure

- Aucune: un séquence de bytes
- Une séquence d'*enregistrements* de taille fixe
- Une séquence d'*enregistrements* de taille variable

De nos jours, toujours le premier cas

D'autres structures ajoutées par dessus (l'insu du SE)

Attributs d'un fichier

Identifiant: un nombre unique; aussi appelé *inode*

Type: fichier simple, répertoire, lien symbolique, ...

Taille

Lieu: où trouver ses données sur le disque

Dates: création, dernière modification, dernier accès, ...

Protection: possesseur, droits d'accès

Attributs d'un fichier

Identifiant: un nombre unique; aussi appelé *inode*

Type: fichier simple, répertoire, lien symbolique, ...

Taille

Lieu: où trouver ses données sur le disque

Dates: création, dernière modification, dernier accès, ...

Protection: possesseur, droits d'accès

~~*Nom*~~: généralement, n'est pas un attribut du fichier!

Opérations sur fichier

Create: généralement commence vide

Open: Trouver un fichier pour y opérer

Read: Lire un certains nombre de bytes d'un fichier

Write: Écrire par dessus, ou étendre un fichier

Truncate: Effacer une partie de la fin du fichier

Close: Indiquer qu'on a finit d'opérer

- `read(pos,size)` et `write(pos,size,bptr)`
- `read(size)` et `write(size,bptr)` et `seek(pos)`

Pourquoi Open+Close

Permet de séparer opérations coûteuses:

- Recherche dans les répertoires
- Trouver les méta-données du fichier
- Vérification des droits d'accès

Permet aussi de maintenir un *état* entre plusieurs opérations

- Éviter d'effacer un fichier en cours d'usage
- Garder un pointeur sur la *position courante*
- Éviter l'accès concurrent (*locking*)

Donne une certaine information d'*intention* au SE

Synchronisation: file locking

Étonnamment problématique en général

Deux approches:

- *Mandatory*: accès interdit si on a pas le verrou
- *Advisory*: les verrours sont là, si vous voulez les utiliser

Vérouiller le contenu d'un fichier

Vérouiller le nom d'un fichier (i.e. une entrée de répertoire)

Vérouiller une partie de son contenu (une tranche de bytes)

Répertoire

Le répertoire est une table de traduction *nom* $\Rightarrow$ *fichier*

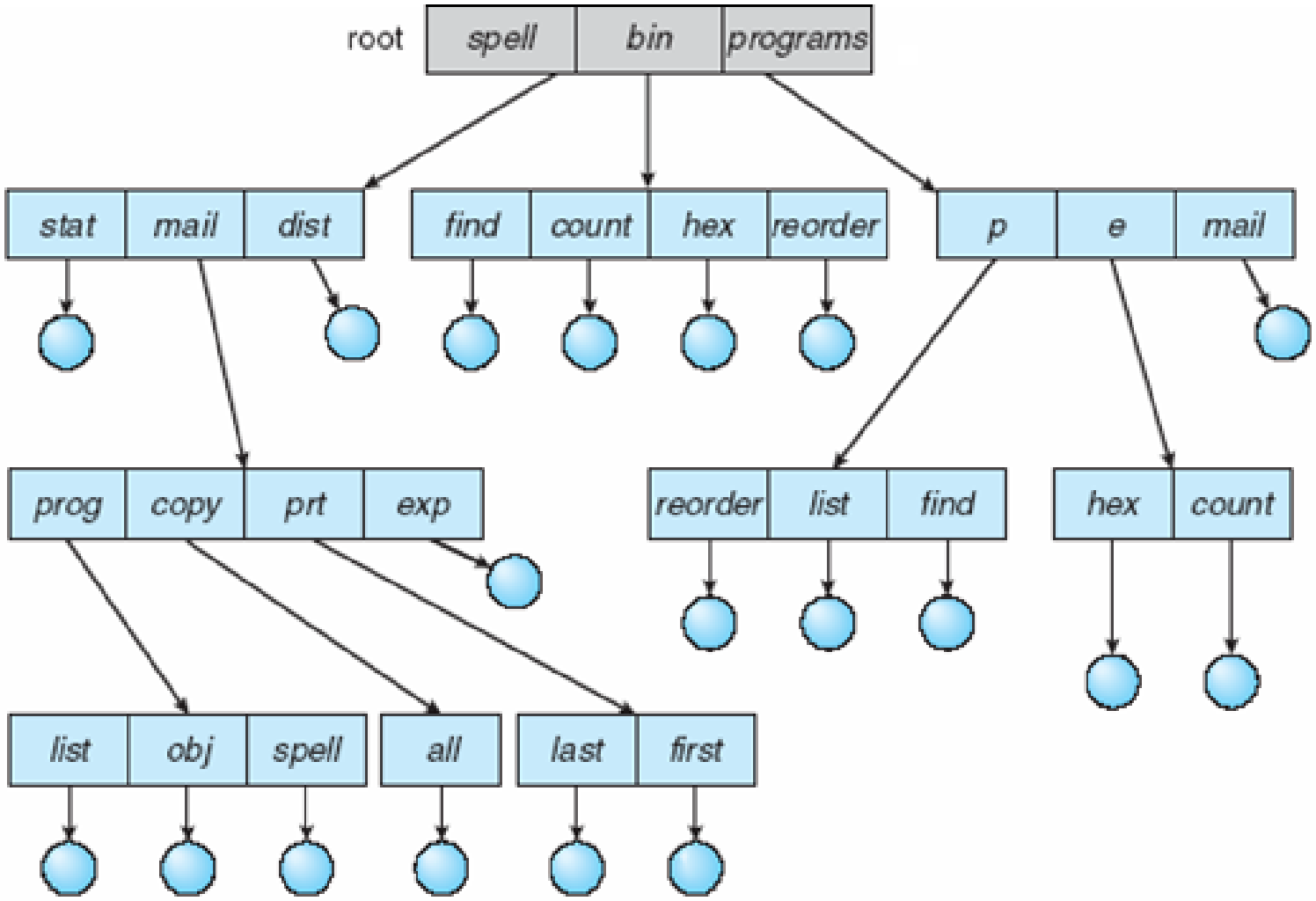
Stockée aussi sur le disque

- *Create*: création d'une nouvelle entrée
- *Delete*: Enlever une entrée
- *Lookup/Search*: Trouver une entrée
- *List*: énumérer les entrées de la table

Généralement structure hiérarchique: un répertoire est un fichier spécial

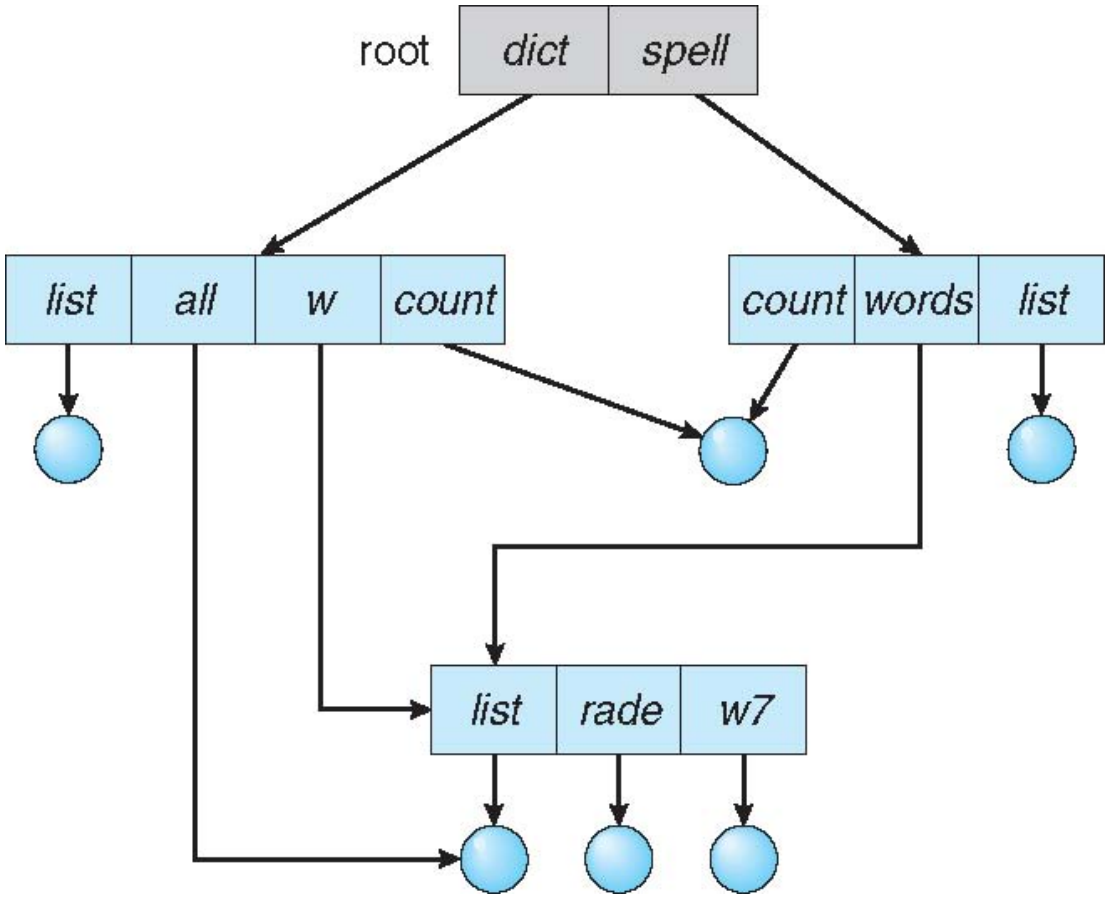
- *MakeDirectory*: créer un nouveau répertoire

Arbre de fichiers



Graphe de répertoires

Un fichier peut être référencé par plusieurs noms



Problèmes de graphes

Lors d'un *Delete*, vérifier si l'objet devient inaccessible

En général, propriété globale: recherche sur *tout* le disque!

Solution: compteurs de références

- Ne marche pas en cas de cycle
- Interdire les cycles

Détecter un cycle, en général, très coûteux

- Une seule référence par répertoire ($\Rightarrow$ pas de cycles)
- Liens symboliques pour compenser

Partitions et systèmes de fichiers

Système de fichiers: arborescence de fichiers sur un disque

Appliqué à un disque logique (e.g. partition)

mount: placer un système de fichiers dans un autre arbre

- Cache une entrée existante par une nouvelle arborescence

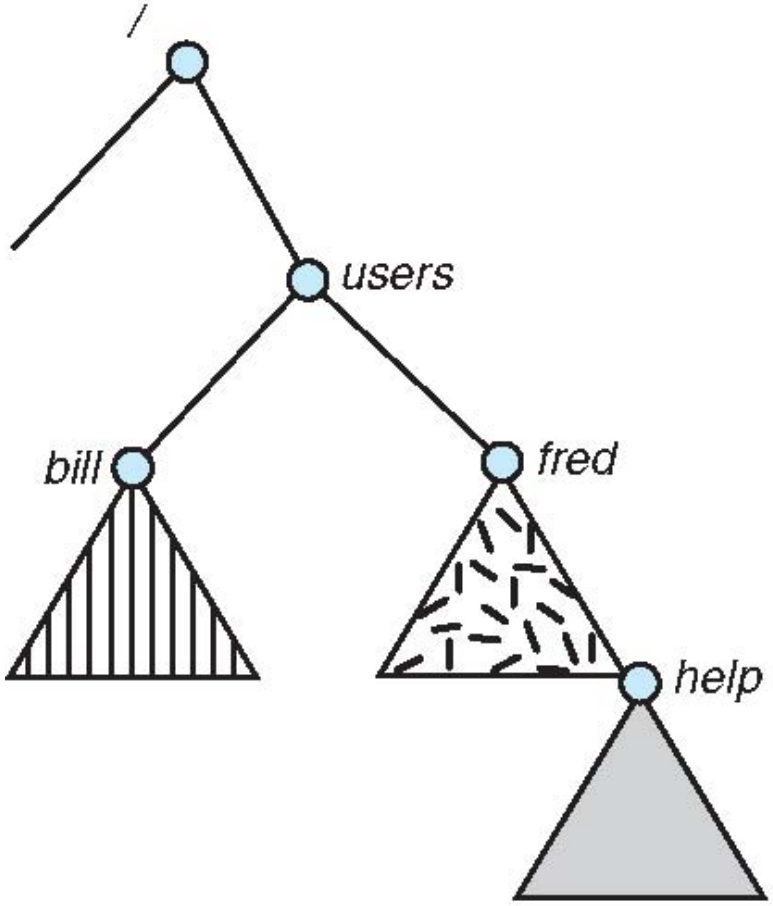
Action purement en mémoire, pas d'effet sur le disque

La *mount table* associe *mount points* $\Rightarrow$ *système de fichiers*

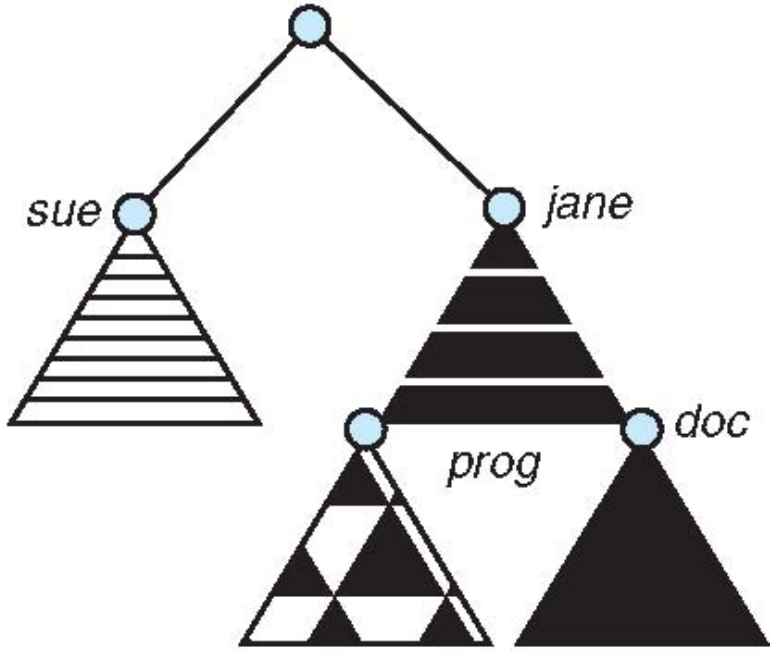
Lors de l'ouverture d'un fichier:

1. Consulter la *mount table* pour trouver le *système de fichiers*
2. Consulter le système de fichiers pour trouver le fichier

Avant mount

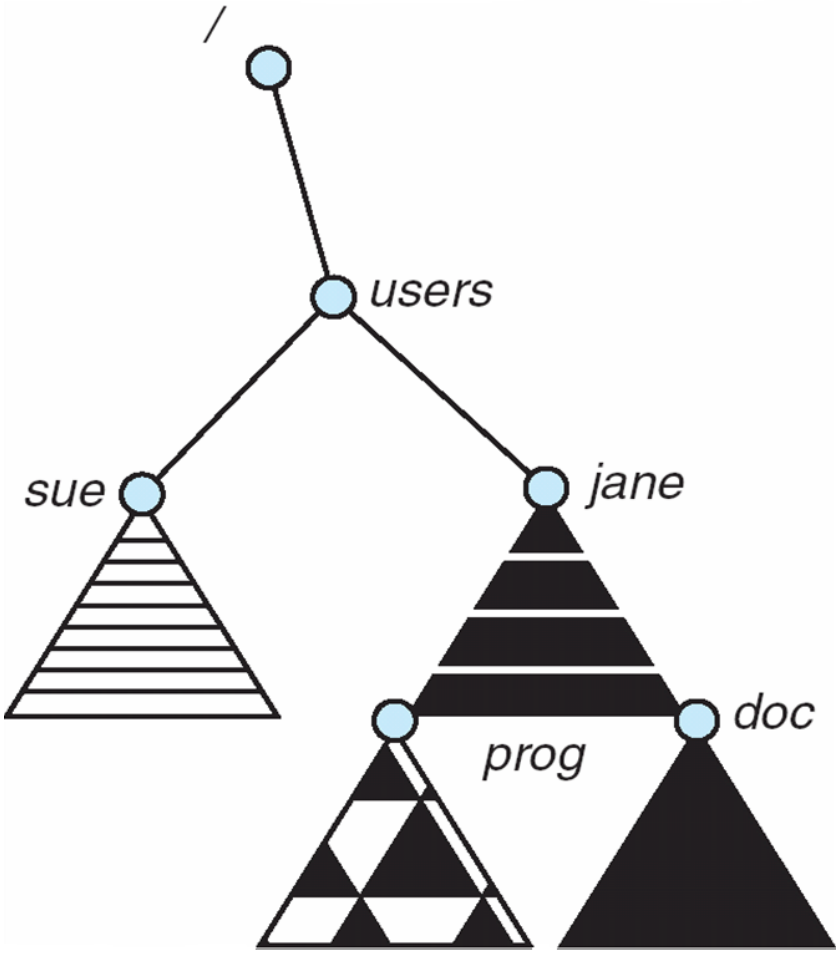


(a)



(b)

Après mount



Partage et protection

Contrôle d'accès

Contrôler qui:

- Certains utilisateur (*user ID*)
- Certains groupes

Contrôler quels accès:

- Lecture, Écriture, Exécution du fichier/répertoire
- Ajout d'une référence dans un répertoire
- Enlever une référence
- Changer les droits d'accès

Virtual File System (VFS)

Plusieurs types de systèmes de fichiers: ext4, tmpfs, ntfs, nfs, hfs, ...

VFS est l'API qui interface le SE avec un système de fichier particulier

Séparer opérations génériques

Cacher détails d'implantation

Modulariser

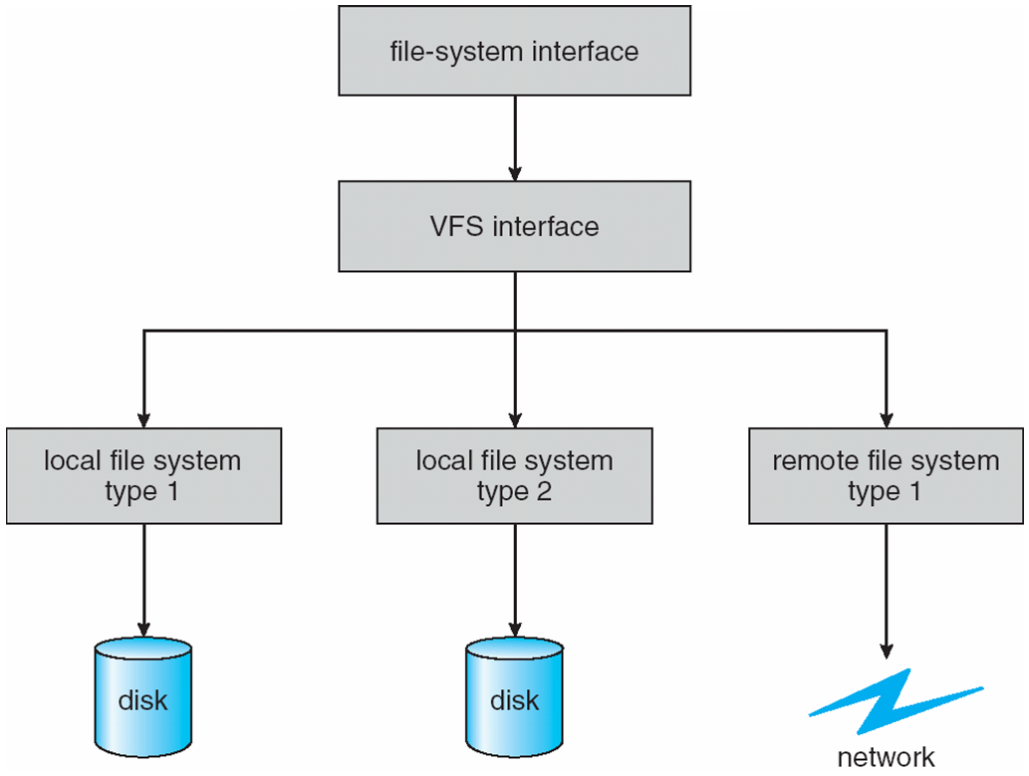
Spécifique à un SE

Un ensemble de *types*:

- *file*, *superbloc*, *dentry*, ...

Un ensemble d'*opérations*:

- *open*, *read*, *link*, ...



Structure d'un système de fichiers

Généralement constitué des éléments suivants:

- *blocs*: les blocs de donnée
- *inode*: structure décrivant un fichier; numéroté
- *répertoire*: type spécial de fichier
- *superbloc*: structure décrivant le système de fichiers
- *free map*: table des blocs libres
- *indexes*: table des blocs constituant un fichier

Structure d'un répertoire

Répertoire est une table *String* $\Rightarrow$ *InodeNb* (plus, si entente)

Peut être implanté comme:

- Liste séquentielle d'entrées
- Table de hachage
- Arbre équilibré, e.g. B-Tree

Mais "linéarisé" en un ensemble de blocs

Important: minimiser le nombre d'accès de blocs

- Pour la recherche (*lookup*), surtout
- Aussi pour l'insertion et l'élimination d'entrées

Méthodes d'allocation

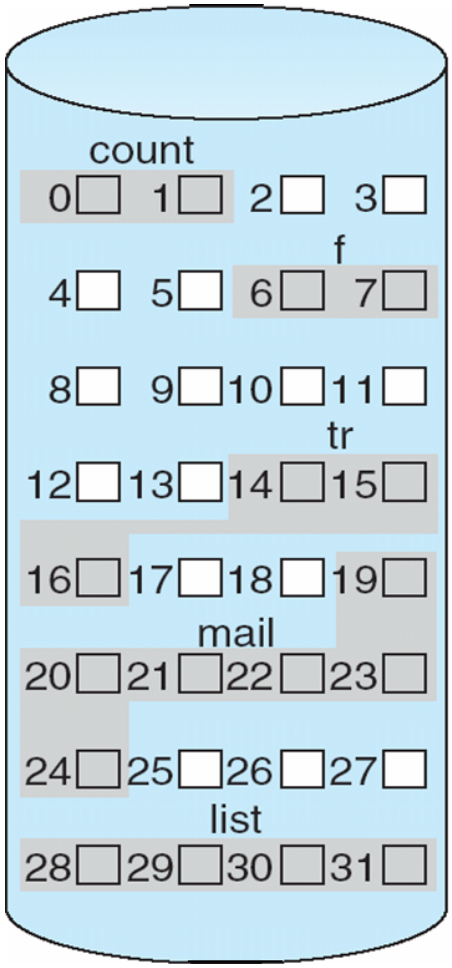
Comment placer les blocs d'un fichier

- Allocation contiguë
- Chaîné
- Index
- *Extents*

Généralement on veux allouer les blocs de manière contiguë

Mais taille du fichier pas connue à l'avance

Allocation contiguë



directory

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

Idéal, sauf pour la fragmentation (*externe*)!

Allocation en chaîne

Inode contient index du premier bloc

Chaque bloc contient index du suivant

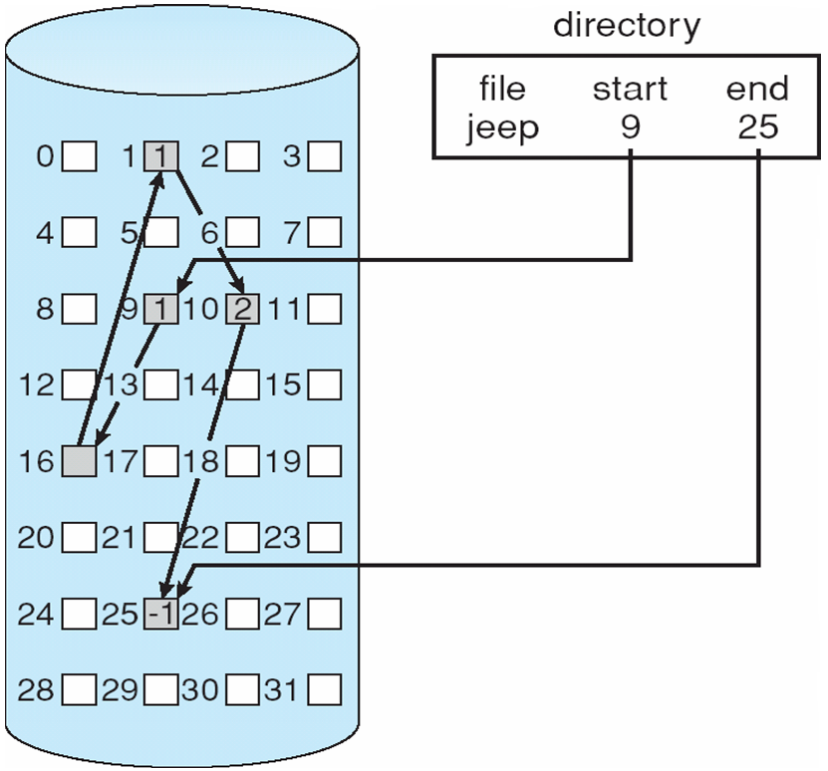
Pas de problème de fragmentation

Pas besoin de compacter

Accès direct inefficace

Risque: accès séquentiel inefficace

Compacter nécessaire quand même



Utilisé dans le système FAT, mais avec liens séparés

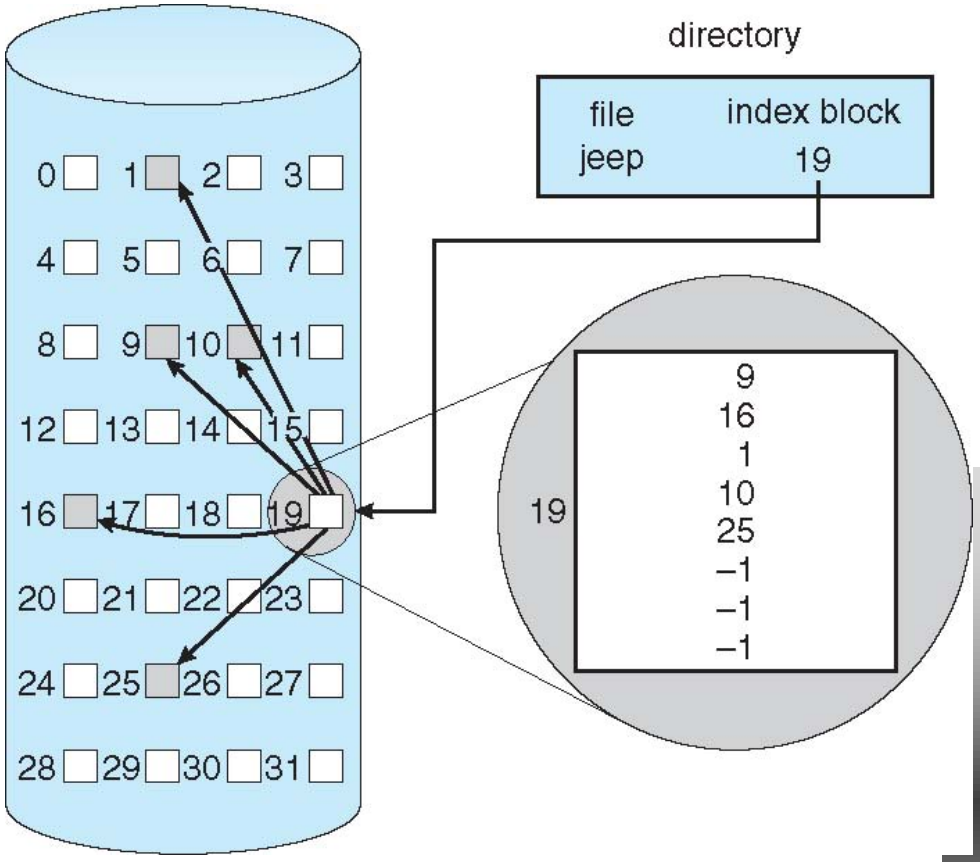
Allocation indexée

Inode contient un emphindex de tous les blocs

- Permet accès aléatoire efficace
- Pas de fragmentation
- Ni besoin de compacter

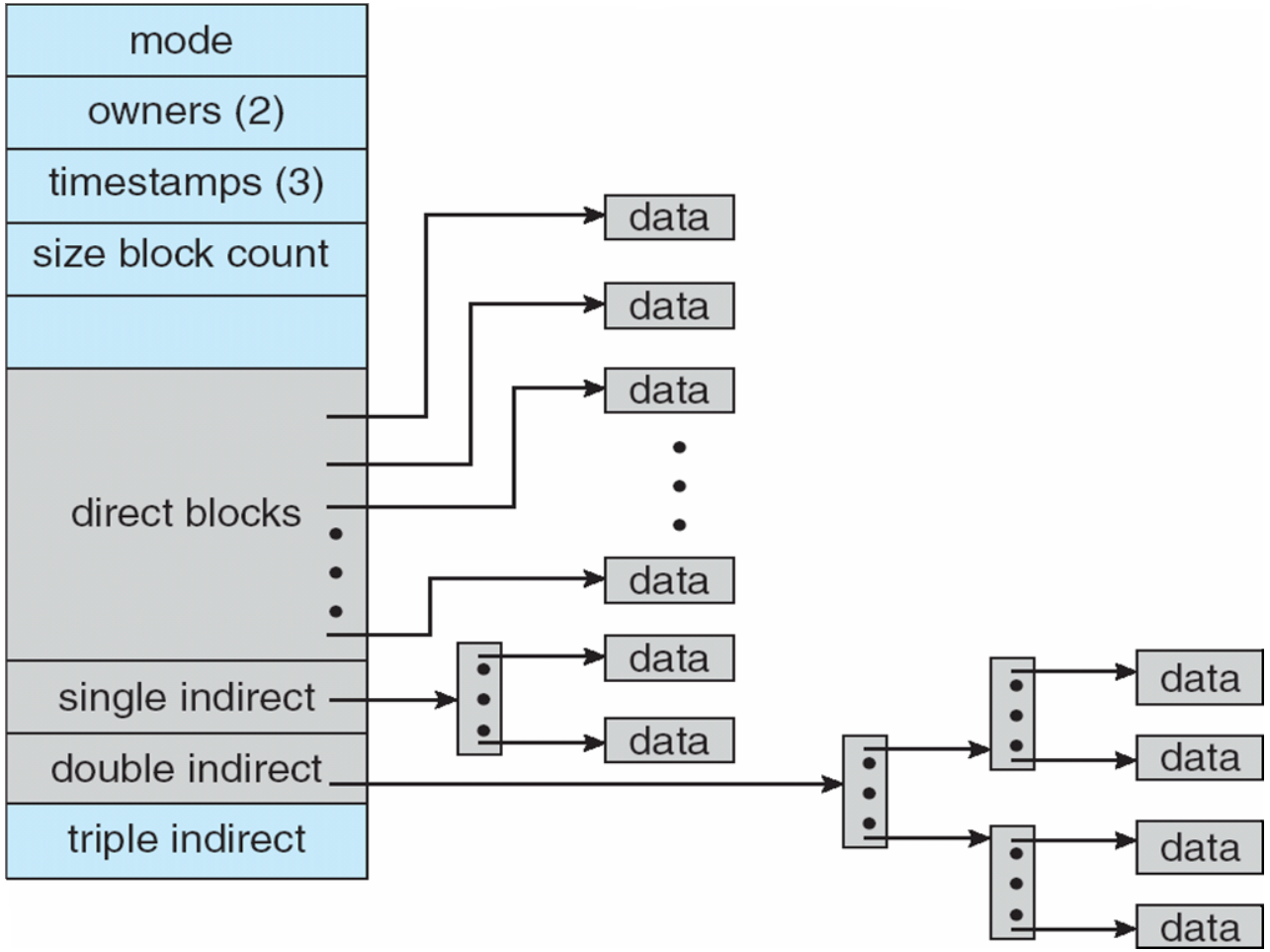
L'index peut occuper plusieurs blocs

- Index contigu?
- Chaîné?
- Ou indexé?



Indexation indirecte progressive

Système “Unix File System”



Allocation par extents

Extent: ensemble de blocs contigus

Inode contient liste ou index des *extents* d'un fichier

- Fichier contigu: comme allocation contiguë
- Fichier fragmenté: pire qu'allocation indexées

Fichiers fragmentés = problème de toute façon

- Profite des efforts de défragmentation des fichiers
- En général, beaucoup plus compact qu'un index classique
- Plus compact $\Rightarrow$ plus rapide

(Dé)Fragmentation

Allocation par bloc $\Rightarrow$ pas de problème de *fragmentation externe*

Autre *Fragmentation*: le fait qu'un fichier n'est pas contigu

Défragmentation périodique: pas nécessaire

Éviter la fragmentation:

- Comprendre la source de la fragmentation
- Placer un nouveau fichier là où il reste de la place
- Éviter de bloquer un fichier encore ouvert avec un nouveau fichier
- Allocation paresseuse

Gestion des blocs libres

Système de fichiers doit garder trace des *blocs libres*

Peuvent être maintenu dans

- Une liste chaînée
- Un *bitmap*
- Extents

Stocker la liste chaînée dans les blocs libres!

Liste chaînée peu pratique pour trouver des blocs contigus/proches

Coût du bitmap négligeable: 1bit/4kB $\Rightarrow$ 0.003%