Systèmes de Fichiers (File Systems)

Gestion du Stockage Persistant et Organisation des Données

Ilias TOUGUI

L'École Supérieure d'Informatique et du Numérique

INF2132 - Systèmes d'Exploitation

PLAN DU COURS

1. Introduction aux Systèmes de Fichiers
2. Fichiers : Concepts Fondamentaux
3. Attributs et Opérations Fichier
4. Organisation des Répertoires
5. Vue d'Ensemble : Implémentation vs Interface
6. Layout des Systèmes de Fichiers
7. Méthodes d'Allocation de Fichiers

Problèmes du Stockage en Mémoire de Processus

Les applications nécessitent un stockage persistant. La mémoire de processus pose trois problèmes majeurs :

- **Capacité limitée** : L'espace d'adressage virtuel est restreint
- **Persistance perdue** : Les données disparaissent à la terminaison du processus
- **Accès exclusif** : Un seul processus peut accéder aux données simultanément

Exemple problématique :

- Réservations aériennes (très gros volume)
- Systèmes bancaires (données critiques)
- Bases de données (accès multi-utilisateur)

Trois Exigences Essentielles

Pour un stockage à long terme efficace :

Exigences Fondamentales

1. Stocker une **très grande quantité** d'information
2. L'information doit **survivre** à la terminaison du processus
3. **Plusieurs processus** doivent accéder simultanément

Solution : Disques magnétiques ou disques SSD offrent une abstraction simple :

- Lire bloc k
- Écrire bloc k

L'Abstraction « Fichier »

Résolution des Questions de Gestion

Les opérations disque brutes (lire/écrire bloc) soulèvent des problèmes critiques :

- Comment trouver l'information ?
- Comment éviter que les utilisateurs lisent les données d'autrui ?
- Quels blocs sont disponibles ?

Solution : L'abstraction ***Fichier***

Fichier

Unité logique d'information créée par un processus, indépendante des autres, offrant une *persistance* et un **accès protégé**.

Les trois abstractions essentielles du système d'exploitation :

1. **Processus** (threads)
2. **Espaces d'adressage** (mémoire virtuelle)
3. **Fichiers** (stockage persistant)

Système de Fichiers : Perspectives

Point de Vue Utilisateur

- Comment les fichiers apparaissent
- Qu'est-ce qu'un fichier ?
- Nommage et protection des fichiers
- Opérations permises

Point de Vue Concepteur

- Implémentation interne
- Gestion des blocs libres

Ce chapitre couvre ces **deux perspectives** : du modèle utilisateur à l'implémentation interne.

Nommage des Fichiers

Le nommage est l'aspect le plus important de tout mécanisme d'abstraction.

Règles de nommage (varient selon le système) :

- Plupart des systèmes : 1 à 8 caractères minimum
- Beaucoup supportent jusqu'à 255 caractères
- Digits et caractères spéciaux souvent autorisés

Sensibilité à la casse :

Système	Sensible à la casse
UNIX / Linux	✓ Oui
MS-DOS / FAT	X Non
Windows (NTFS)	Partiellement (Unicode)

Exemple UNIX : `ilias`, `Ilias`, `ILIAS` = **trois fichiers distincts**

Extensions de Fichiers

Format : `nom.extension` (séparation par point)

Extension	Signification
`.c`	Source C
`.o`	Fichier objet (compilation)
`.tex`	Source LaTeX
`.pdf`	Portable Document Format
`.txt`	Fichier texte ASCII
`.zip`	Archive compressée

Interprétation des extensions :

- **UNIX** : Convention (pas forcée par l'OS)
- **Windows** : Associées à des programmes (double-clic lanc)

Exemple UNIX : `script` peut être exécutable *sans* extension spéciale.

Trois Types de Structures de Fichiers

Les fichiers peuvent être organisés différemment selon le système :

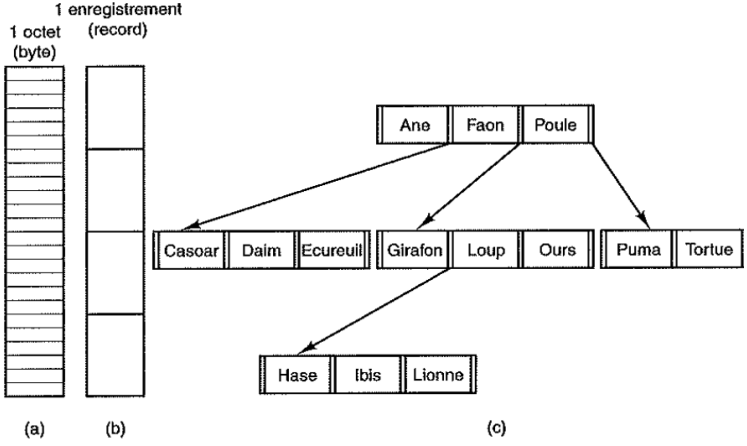

Figure: Trois Modèles de Structure Fichier

Type (a) : Séquence d'Octets

Modèle utilisé par UNIX et Windows

Caractéristique

Le fichier est une simple séquence d'octets non structurée. L'OS ne connaît pas la structure interne.

Avantages :

- **Flexibilité maximale** : les programmes contrôlent la structure
- **Pas d'intervention OS** : pas de contraintes de format
- **Liberté d'innovation** : chaque application définit son format

Implication :
C'est le modèle qui domine aujourd'hui. La structure du fichier (entête, données, etc.) est gérée *au niveau de l'application*, pas par l'OS.

Type (b) : Séquence d'Enregistrements

Modèle historique, peu utilisé aujourd'hui

Concept

Un fichier est une séquence d'enregistrements de longueur fixe. Chaque opération `read` retourne un enregistrement complet.

Contexte historique :

- Ordinateurs mainframe des années 1960-1970
- Cartes perforées 80 colonnes
- Fichiers d'enregistrements de 80 ou 132 caractères
- Lecture/écriture par *unité d'enregistrement*

Raison de l'abandon :
Avec les disques, le modèle byte-sequence s'est avéré plus flexible. Aucun système moderne n'utilise ce modèle comme système de fichiers principal.

Type (c) : Fichiers Structurés en Arbre

Modèle pour les bases de données commerciales

Concept

Le fichier est un arbre d'enregistrements de longueur variable. Chaque enregistrement contient une **clé** en position fixe. L'arbre est trié sur cette clé pour recherche rapide.

Opérations :

- **Accès par clé** : « Obtenir l'enregistrement avec clé = `pony` »
- Non pas « obtenir le *n*-ième enregistrement »
- L'OS place les nouveaux enregistrements automatiquement

Usage :
Systèmes mainframe pour traitement commercial de données. Moins courant sur workstations et PC modernes.

Types de Fichiers dans l'OS

Les systèmes d'exploitation reconnaissent plusieurs **types** de fichiers :

Types principaux (UNIX / Windows) :

- **Fichiers réguliers** : contiennent des données utilisateur
- **Répertoires** : fichiers système maintenant la hiérarchie
- **Fichiers spéciaux caractère** : périphériques série (terminaux, imprimantes)
- **Fichiers spéciaux bloc** : disques et stockage bloc

Focus du Cours

Cette section se concentre sur les **fichiers réguliers**. Les répertoires seront abordés à la section suivante.

Fichiers Réguliers : ASCII vs Binaires

Deux catégories principales :

1. Fichiers ASCII

- Lignes de texte imprimable
- Peut être affiché et édité avec n'importe quel éditeur texte
- Facilitent les pipelines shell (`cat | grep | sort`)

2. Fichiers Binaires

- Toute autre format (pas ASCII)
- Impression = « junk aléatoire »
- Structure interne connue seulement des programmes qui les utilisent
- Exemples : exécutables, archives, images

Note : Cette distinction est une **convention**. L'OS traite tous les fichiers comme des séquences d'octets.

Modèles d'Accès aux Fichiers

Deux modèles historiques :

1. Accès Séquentiel (anciens systèmes)

- Lire tous les octets/enregistrements dans l'ordre
- Pas de « sauts » possibles
- Rebobinage permis (relire depuis le début)
- Approprié pour bandes magnétiques

2. Accès Aléatoire (systèmes modernes avec disques)

- Lire/écrire n'importe quel octet/enregistrement
- **Essentiel** pour les bases de données
- Deux méthodes : position dans chaque opération OU appel `seek`

Exemple critique :
Réservation aérienne : accès direct au dossier d'un client *sans* lire tous les autres clients d'abord.

L'Appel Système `seek()`

Méthode UNIX et Windows pour accès aléatoire :

seek()

Place le **pointeur de fichier** à une position spécifique. Les opérations `read`/`write` suivantes commencent à cette position.

Workflow typique :

1. `seek(file, position)`
2. `read(file, buffer, count)` ← lit à partir de `position`
3. `seek(file, other_position)`
4. `write(file, buffer, count)` ← écrit à partir de `other_position`

Alternative (moins courante) : Spécifier la position dans chaque opération `read(file, position, buffer, count)`.

Attributs de Fichier (Métadonnées)

Au-delà du nom et des données, chaque fichier possède des **attributs** :

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure: Attributs Typiques d'un Fichier

Attributs de Protection et Contrôle d'Accès

Contrôle de qui peut accéder au fichier et comment :

Attribut	Signification
Protection	Qui peut lire/écrire/exécuter
Password	Mot de passe pour accès
Creator	ID du créateur
Owner	Propriétaire actuel
Read-only flag	0 = r/w, 1 = lecture seule

Flags spéciaux :

- **Hidden** : Ne pas afficher dans les listes
- **System** : Fichier système (ne pas modifier)
- **Archive** : Flag de sauvegarde
- **Temporary** : Supprimer à la terminaison du processus
- **Lock flags** : Verrouillage

Attributs de Dates et Tailles

Gestion temporelle et dimensionnement :

Attribut	Usage
Creation time	Date/heure création
Time of last access	Dernier accès
Time of last change	Dernière modification
Current size	Taille actuelle (octets)
Maximum size	Taille max autorisée

Exemple d'usage (Unix `make`) :

- Compare `time_of_last_change` du source avec le binaire
- Si source > binaire : recompilation nécessaire.

Note : Les anciens mainframes *exigeaient* une taille max au moment de la création. Les systèmes modernes l'allouent dynamiquement.

Opérations sur Fichiers : Vue d'Ensemble

Les systèmes d'exploitation fournissent plusieurs appels système POSIX pour gérer les fichiers. Les plus courants :

1. **Create** : Créer avec zéro donnée, définir attributs
2. **Delete** : Supprimer et libérer espace disque
3. **Open** : Charger attributs et adresses en mémoire pour accès rapide
4. **Close** : Libérer ressources internes, forcer écriture du dernier bloc
5. **Read** : Lire des données à partir de la position courante
6. **Write** : Écrire des données, possiblement créer/overwrite
7. **Append** : Ajouter à la fin seulement
8. **Seek** : Repositionner le pointeur (accès aléatoire)
9. **Get attributes** : Lire les métadonnées
10. **Set attributes** : Modifier les métadonnées
11. **Rename** : Changer le nom d'un fichier

Commandes Unix/Linux pour les Opérations sur Fichiers

Voici la correspondance entre les opérations sur fichiers et les commandes Unix/Linux :

1. **Create** `creat()` ou `open()` avec flags `O_CREAT | O_WRONLY`
2. **Delete** `unlink()` pour supprimer un fichier
3. **Open** `open()` pour ouvrir un fichier existant ou le créer
4. **Close** `close()` pour fermer un fichier ouvert
5. **Read** `read()` pour lire des données à partir de la position courante
6. **Write** `write()` pour écrire des données dans un fichier
7. **Append** `open()` avec le flag `O_APPEND` pour ajouter à la fin
8. **Seek** `lseek()` pour repositionner le pointeur de fichier
9. **Get attributes** `stat()` ou `fstat()` pour récupérer les métadonnées
10. **Set attributes** `chmod()` (permissions), `chown()` (propriétaire), `utimes()` (dates)
11. **Rename** `rename()` ou `link()` suivi de `unlink()`

Commandes Shell Équivalentes

Les commandes shell courantes pour manipuler les fichiers :

1. **Create** `touch fichier` pour créer un fichier vide
2. **Delete** `rm fichier` pour supprimer un fichier
3. **Read** `cat`, `less`, `more` pour afficher le contenu
4. **Write/Append** Redirection > (écraser) ou » (ajouter)
5. **Get attributes** `stat fichier` pour tous les attributs, `ls -l` pour un résumé
6. **Set attributes** `chmod` (permissions), `chown` (propriétaire), `touch -t` (dates)
7. **Rename** `mv ancien nouveau` pour déplacer ou renommer

Système à Niveau Unique : Principe

Simplicité maximale, organisation limitée

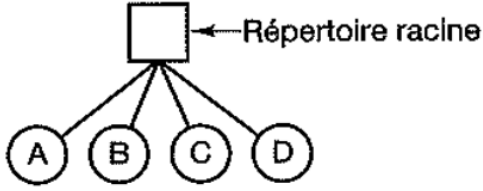

Figure: Système à répertoire unique

- Simplicité d'implémentation et de recherche de fichier.
- Utilisé dans les tous premiers OS ou systèmes embarqués.
- **Limites**: impossible de structurer logiquement plusieurs projets ou utilisateurs.

Hiérarchie de Répertoires

L'arbre des fichiers modernes

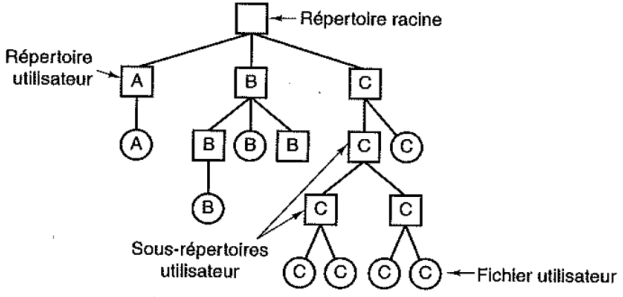

Figure: Organisation en arborescence hiérarchique

- Regroupe logiquement fichiers par utilisateur, projet ou type.
- Tous les OS modernes (UNIX, Windows, MacOS) utilisent cette structure.
- **Chaque utilisateur** possède son propre « root » et peut organiser en profondeur.

Chemins et Navigation

- Deux types principaux :
  1. Chemin absolu (`/home/$USER/Desktop`): depuis la racine
  2. Chemin relatif (`fichier` ou `../Desktop`): depuis le répertoire courant
- Séparateurs :
  - UNIX : /
  - Windows : \
  - MULTICS : >
- Toutes les arborescences UNIX fournissent :
  - `.` (dot) : le répertoire courant
  - `..` (dotdot) : le répertoire parent
- Permet navigation efficace sans retaper le chemin complet.
- **Exemple** : Copier un dictionnaire système dans son propre dossier :

```
$ cp ../lib/dictionary .
```

- Dot et dotdot irréductibles : on ne peut jamais supprimer `.` ni `...`

Opérations sur Répertoires : Appels Système POSIX

Voici la correspondance entre les opérations sur répertoires et les commandes Unix/Linux :

1. **Créer** : `mkdir()` crée un nouveau répertoire
2. **Supprimer** : `rmdir()` supprime un répertoire vide
3. **Ouvrir** : `opendir()` ouvre un répertoire pour lecture
4. **Fermer** : `closedir()` ferme le répertoire et libère la ressource
5. **Lire** : `readdir()` récupère l'entrée suivante formatée
6. **Renommer** : `rename()` modifie le nom du répertoire
7. **Lier** : `link()` crée un lien, `unlink()` le supprime

Note

Le système d'appels sur dossiers varie plus selon les OS que celui sur les fichiers !

Opérations sur Répertoires : Commandes Shell

Les commandes shell courantes pour manipuler les répertoires :

1. **Créer** : `mkdir nom_dossier` crée un nouveau répertoire
2. **Supprimer** : `rmdir nom_dossier` supprime un répertoire vide
3. **Ouvrir/Lire** : `ls` affiche le contenu d'un répertoire
4. **Lire détaillé** : `ls -la` affiche tous les fichiers avec détails
5. **Naviguer** : `cd nom_dossier` change de répertoire courant
6. **Renommer** : `mv ancien_nom nouveau_nom` renomme le répertoire
7. **Copier** : `cp -r source destination` copie un répertoire récursivement
8. **Supprimer** : `rm -r nom_dossier` supprime un répertoire et son contenu

Du Point de Vue de l'Utilisateur à Celui de l'Implémenteur

Interface (Point de Vue Utilisateur) :

- Nommage et organisation hiérarchique des fichiers
- Opérations : créer, lire, écrire, supprimer
- Permissions et protection d'accès

Implémentation (Point de Vue Concepteur) :

- Comment stocker fichiers et répertoires sur disque ?
- Comment gérer l'espace libre efficacement ?
- Comment assurer fiabilité et performance ?
- Structures de données internes, allocations, gestion disque

Objectif de Cette Section

Comprendre les **compromis** (trade-offs) entre simplicité, performance et gestion d'espace.

Structure Interne d'une Partition

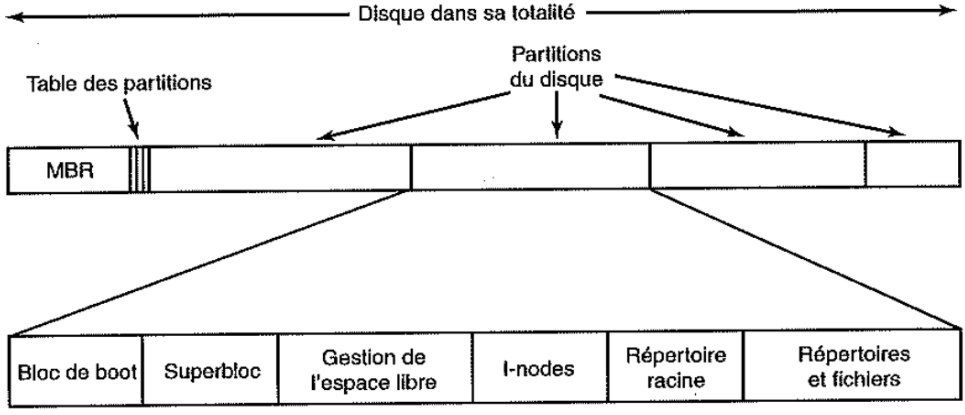

Figure: Layout typique d'une partition avec système de fichiers

Partitionnement du Disque

Partition : Division logique du disque physique, chacune avec un système de fichiers indépendant.

Secteur 0 : Master Boot Record (MBR)

- Contient la **partition table** (adresses de début/fin)
- Une partition marquée comme **active**
- Utilisé lors du démarrage du système

Processus de démarrage :

1. BIOS lit et exécute le MBR
2. MBR localise la partition active
3. Charge le **boot block** (premier bloc de la partition)
4. Boot block charge l'OS

Note : Même sans OS, chaque partition a un boot block (peut être utile à l'avenir).

Boot Block et Superblock

Boot Block (premier bloc de la partition)

- Programme d'amorçage spécifique à ce système de fichiers
- **Vide ou non-utilisé** si le système n'est pas bootable

Superblock

Superblock

Bloc critique contenant tous les **paramètres clés** du système de fichiers. Chargé en mémoire au boot ou premier accès.

Contenu typique du superblock :

- Magic number (identifie le type de filesystem)
- Nombre de blocs totaux
- Taille des blocs
- Nombre d'i-nodes
- État du filesystem (clean/dirty)
- Autres paramètres

Composants du Layout : Gestion d'Espace

1. **Superblock**
   - Paramètres globaux du filesystem
2. **Free Space Management**
   - **Bitmap** : bit par bloc (0 = libre, 1 = alloué)
   - **Linked list** : liste chaînée des blocs libres
   - Ou autre structure (selon OS)
3. **I-nodes Array**
   - Tableau de structures (une par fichier)
   - Attributs et adresses disque de chaque fichier
4. **Root Directory**
   - Sommet de la hiérarchie
5. **Files and Directories**
   - Reste du disque : données et sous-dossiers

Le Problème Central : Allocation de Blocs

Problème Fondamental

Comment garder trace de quels blocs disque appartiennent à quel fichier ?

Différentes approches selon l'OS :

1. **Allocation contiguë** : blocs consécutifs
2. **Linked-list** : liste chaînée de blocs
3. **FAT (File Allocation Table)** : table en mémoire
4. **I-nodes** : structure de métadonnées par fichier

Compromis : Simplicité <-> Performance <-> Flexibilité

Méthode 1 : Allocation Contiguë

Principe : Chaque fichier = blocs consécutifs sur disque.

Exemple :

- Fichier A (4 blocs) → blocs 0-3
- Fichier B (3 blocs) → blocs 4-6
- Fichier C (6 blocs) → blocs 7-12
- ...

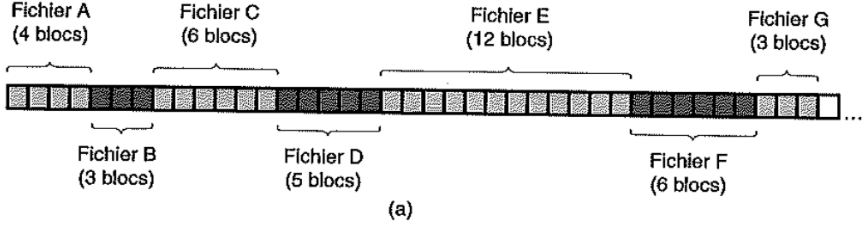

Figure: Allocation contiguë : fichiers dans blocs consécutifs

Allocation Contiguë : Avantages

Avantages

1. **Simple à implémenter** : Deux nombres suffisent (adresse début, nombre blocs)
2. **Performance excellente** : Lecture complète en une seule opération disque
3. **Un seul seek** nécessaire (au premier bloc), puis lecture continue à la bande passante maximale

Exemple (calcul d'adresse) :
Si fichier commence bloc 10 et compte 5 blocs :

- Bloc logique 0 $\rightarrow$ adresse physique 10
- Bloc logique 3 $\rightarrow$ adresse physique 13 (simple addition)

Allocation Contiguë : Principal Inconvénient

Fragmentation externe : Le vrai problème !

Au fil du temps, après suppression de fichiers :

- Des « trous » (blocs libres) parsèment le disque
- Impossible de *compacter* simplement (millions de blocs à recopier $\rightarrow$ heures/jours)
- Disque finit par ressembler à : [fichier - trou - fichier - trou - ...]

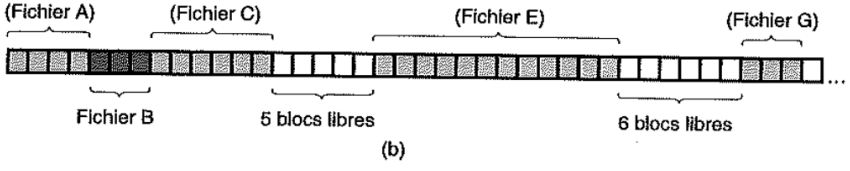

Figure: Fragmentation externe après suppression de fichiers

Allocation Contiguë : Autre Inconvénient

Prédiction obligatoire de la taille finale

Scenario cauchemardesque :

```
Utilisateur : Ouvre un éditeur de texte
Éditeur : "Quelle sera la taille finale du document ?"
Utilisateur : "500 KB"
[Utilisateur tape...tape...tape...]
Éditeur : "ERREUR : Disque plein, le trou de 500 KB est épuisé"
```

Solutions peu pratiques :

- Demander taille maximale → gaspillage d'espace
- Taille insuffisante → arrêt prématuré
- Relancer le programme avec autre valeur → frustration !

Verdict : Impraticable pour usage général → abandonné sur disques magnétiques.

Allocation Contiguë : Quand C'est Utile ?

Cas où allocation contiguë est *encore utilisée* :

1. CD-ROM et DVD

- Taille fichier connue d'avance (gravure unique)
- Ne change jamais après gravure
- Performance optimale : lecture contiguë en streaming

2. DVD-Video ou Blu-ray

- Un film 90 min = 4.5 GB
- UDF limite taille fichier à 1 GB (30-bit file length)
- Solution : découper en 3-4 fichiers de 1 GB, chacun contigu (appelés *extents*)

Leçon historique : Bonne idée « ancienne » revient sur **nouveaux médias write-once**.

Méthode 2 : Allocation par Linked-List

Principe : Chaque bloc contient un pointeur au bloc suivant du même fichier.

Structure :

- Premiers bytes du bloc : pointeur au bloc suivant
- Reste du bloc : données

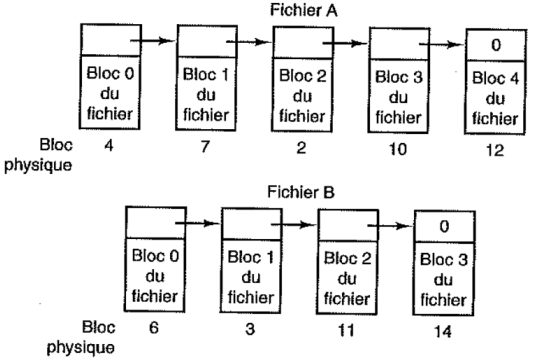

Figure: Allocation Linked-List : pointeurs chaînant les blocs

Linked-List : Avantages et Inconvénients

Avantages

- **Zéro fragmentation** : chaque bloc libre peut être utilisé
- **Pas besoin prédire taille** : fichier grandit dynamiquement
- Toute l'espace disque peut servir

Inconvénients

- **Accès aléatoire très lent** : doit traverser tous les blocs précédents
- **Taille bloc diminuée** : espace perdu pour pointeurs
- Performance séquentielle acceptable, mais random access = cauchemar

Exemple : Accéder au bloc 1000 $\rightarrow$ suivre 999 pointeurs = très coûteux !

Méthode 3 : FAT (File Allocation Table)

Idée clé :
Sortir tous les **pointeurs de chaînage** des blocs de données et les mettre dans une **table centralisée en mémoire RAM**.

Structure :

- Table `FAT[]` avec une entrée **par bloc disque**
- `FAT[i]` = numéro du **bloc suivant** dans la chaîne
- Chaîne terminée par un **marqueur spécial** (ex: -1)
- Table **entièrement chargée en RAM**

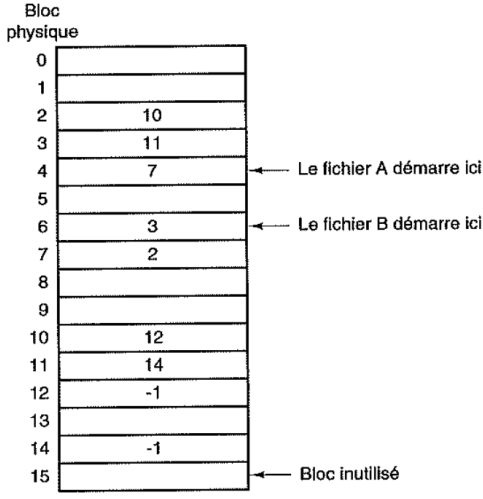

Figure: FAT : table des pointeurs en mémoire

FAT : Avantages

Avantages de la FAT

- ✓ **Blocs entièrement disponibles pour les données**
  - Aucun pointeur stocké *dans* les blocs
  - Zéro octet gaspillé pour les métadonnées de chaînage
- ✓ **Accès aléatoire plus facile et rapide**
  - Table FAT **entièrement en RAM**
  - Suivre une chaîne = lookups rapides en mémoire
  - Une seule I/O disque à la fin pour lire le bloc demandé
- ✓ **Flexible : fichiers croissent sans prédiction de taille**
  - Aucune obligation de pré-allouer une taille
  - Ajouter des blocs = ajouter des entrées dans la table FAT

FAT : Avantages - Suite

Avantages de la FAT

✓ **Zéro fragmentation externe**
- Blocs peuvent être disjoints sans gaspillage
- Utilisation optimale de l'espace disque

Répertoire : Stocke simplement le **numéro du premier bloc** du fichier (ex: 4 pour fichier A).

Exemple : FAT = système MS-DOS et Windows hérité, toujours supporté sur clés USB.

FAT : Inconvénient Majeur

Problème Critique : Taille de la Table en RAM

Hypothèses réalistes pour un disque moderne :

Données

- Capacité disque : **1 TB** = 1×10^{12} octets
- Taille d'un bloc : **1 KiB** = 1024 octets
- Taille d'une entrée FAT : **4 octets** (FAT-32)

Calcul détaillé :

- Nombre de blocs = $\frac{1 \times 10^{12}}{1024} = \frac{10^{12}}{2^{10}} \approx$ **1 milliard de blocs**
- Taille table FAT = 1 milliard $\times$ 4 octets $=$ **4 GB de RAM**

FAT : Inconvénient Majeur - Suite

Problème d'Échelle

FAT **ne scale pas** avec les disques modernes !
Réserver **4 GB de RAM** juste pour une table d'allocation est **impratique et dispendieux**.
Pire : un disque de 10 TB nécessiterait **40 GB de RAM** !

Formule générale :

$$\text{Taille FAT} = \frac{\text{Capacité disque}}{\text{Taille bloc}} \times \text{Taille entrée} = \frac{D \cdot E}{B}$$

La taille FAT est **proportionnelle à la taille du disque** $\Rightarrow$ **mauvaise scalabilité**.

Verdict :

- **FAT fonctionne :** Petits disques (DOS, clés USB anciennes)
- **FAT échoue :** Disques modernes (serveurs, SSD, ...)

Méthode 4 : I-nodes (Index-Nodes)

Meilleure solution pour systèmes modernes

Principe clé :
Associer à **chaque fichier** une structure appelée **i-node** (index-node) qui contient les attributs et les adresses disque des blocs du fichier.

Contenu d'un i-node :

- **Attributs fichier**
  - Propriétaire, permissions (rwx)
  - Timestamps (création, modif, accès)
  - Taille du fichier, type

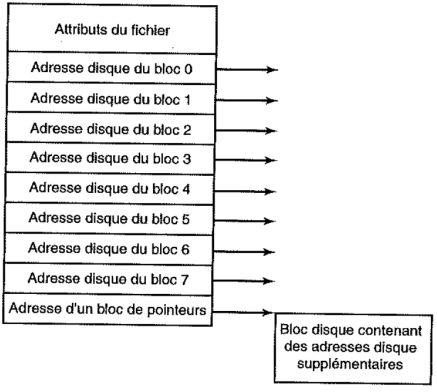

Figure: Structure d'un I-node UNIX avec pointeurs directs et indirects

Méthode 4 : I-nodes (Index-Nodes)

Meilleure solution pour systèmes modernes

Contenu d'un i-node :

- **Adresses disque**
  - 12 pointeurs **directs**
  - 1 pointeur **simple indirection**
  - 1 pointeur **double indirection**
  - 1 pointeur **triple indirection**

Analogie

L'i-node = **fiche technique** d'un fichier (description + adresses des blocs)
Les blocs = **données brutes** du fichier

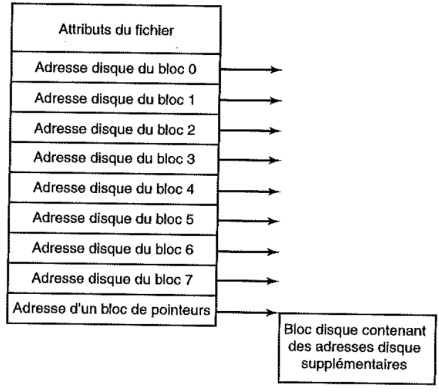

Figure: Structure d'un I-node UNIX avec pointeurs directs et indirects

I-nodes : Avantages Décisifs

Grand Avantage : Charge en RAM que si fichier ouvert

✓ **Scalabilité mémoire exceptionnelle**
- I-node en RAM **uniquement si fichier ouvert**
- Fichiers fermés = i-nodes sur disque (zéro RAM)
- Mémoire dépend du **nombre de fichiers ouverts**, pas de la taille du disque

✓ **Calcul mémoire prévisible**
- Par i-node : n octets (ex: 256 octets)
- Max fichiers ouverts : k (ex: 1000)
- **Total RAM** = $k \times n$ octets
- Exemple : $1000 \times 256 = 256$ Kio (négligeable)

✓ **Flexibilité pour gros fichiers**
- Pointeurs directs pour petits fichiers (rapide)
- Indirection pour gros fichiers (flexible)

I-nodes : Avantages Décisifs

Comparaison directe FAT vs I-nodes :

	FAT	**I-nodes**
RAM dépend de	Taille disque	Fichiers ouverts
Disque 1 TB	4 GB RAM	256 Kio RAM
Scalabilité	Mauvaise	Excellente

Conclusion : I-nodes = solution **moderne** (UNIX, Linux, ext2/3/4).

I-nodes : Gestion des Gros Fichiers

Problème : Si un i-node a seulement 12 pointeurs directs, comment gérer des fichiers plus gros ?

Solution : Indirection (plusieurs niveaux)

Mécanisme d'Indirection

- **12 pointeurs directs** : pointent directement vers 12 blocs de données
- **1 pointeur simple indirection** : pointe vers un bloc contenant 256+ adresses de blocs
- **1 pointeur double indirection** : pointe vers un bloc pointant vers d'autres blocs d'indirection
- **1 pointeur triple indirection** : pour très gros fichiers (rare)

I-nodes : Gestion des Gros Fichiers

Calcul de capacité (blocs de 4 KiB) :

Type	**Calcul**	**Capacité**
12 pointeurs directs	12×4 KiB	48 KiB
Simple indirection	256×4 KiB	1 MiB
Double indirection	$256 \times 256 \times 4$ KiB	256 MiB
Triple indirection	$256 \times 256 \times 256 \times 4$ KiB	64 GiB
TOTAL		$\approx$ **64 GiB**

Efficacité du mécanisme :

- **Petits fichiers** (< 48 KiB) : accédés **directement** (1 lookup I/O)
- **Fichiers moyens** (< 1 MiB) : **simple indirection** (2 lookups I/O)
- **Très gros fichiers** (> 1 MiB) : **indirection multiple** (acceptable)
- Pas d'overhead pour les petits fichiers

Q/R : Comprendre le Concept d'I-node

Question 1 : Quelle est la différence entre un i-node et un bloc de données ?

Réponse :

- **I-node** = fiche technique du fichier
  - Contient : attributs (propriétaire, permissions) + adresses de blocs
  - Stocké sur disque, charge en RAM que si fichier ouvert
  - Petit (256 octets typiquement)
- **Bloc de données** = contenu brut du fichier
  - Contient : octets du fichier (texte, image, vidéo, etc.)
  - Stocké sur disque
  - Gros (4 KiB typiquement)

Q/R : Comprendre le Concept d'I-node

Question 2 : Pourquoi les i-nodes sont mieux pour la mémoire RAM que FAT ?

Réponse :

- **FAT** : table centralisée en RAM avec une entrée par bloc disque
  - Pour disque 1 TB : 4 GB de RAM requis (même si aucun fichier ouvert)
- **I-nodes** : charger en RAM seulement les fichiers ouverts
  - Pour 1000 fichiers ouverts : 256 Kio de RAM
  - Indépendant de la taille du disque

Q/R : Pointeurs Directs vs Indirects

Question 3 : À quoi servent exactement les 12 pointeurs directs ?

Réponse :

- Les 12 pointeurs directs pointent **directement** vers 12 blocs de données du fichier
- Ils permettent d'accéder aux **premiers 48 KiB** du fichier **sans indirection**
- Avantage : pour petits fichiers (la majorité), accès **très rapide**
- Les adresses sont écrites directement dans l'i-node

Q/R : Pointeurs Directs vs Indirects

Question 4 : Que se passe-t-il quand un fichier dépasse 48 KiB ?

Réponse :

- Le SE n'a plus de place dans l'i-node pour les 13e adresse
- Il utilise alors le pointeur d'indirection simple
- Ce pointeur siffle vers un **bloc spécial** qui contient 256 adresses supplémentaires
- Le fichier peut alors utiliser jusqu'à $12 + 256 = 268$ blocs (≈ 1 MiB)

Q/R : Pointeurs Directs vs Indirects

Question 5 : Et pour les fichiers de 100 MiB ?

Réponse :

- Simple indirection suffit pour 1 MiB, mais pas pour 100 MiB
- Le SE utilise alors la **double indirection**
- Double indirection = bloc pointant vers des blocs d'indirection
- Permet $256 \times 256 = 65536$ blocs de plus = 256 MiB supplémentaires

Q/R : Comment le SE Accède aux Données

Question 6 : Pas à pas, comment lit-on le bloc logique 10 d'un fichier ?

Réponse (bloc logique 10, < 12 = pointeur direct) :

1. Charger l'i-node en RAM (s'il n'est pas déjà chargé)
2. Lire la 10e entrée de l'i-node (pointeur direct)
3. Cette entrée contient l'adresse physique du bloc : ex, bloc 4567
4. Aller lire le bloc 4567 sur le disque
5. **Résultat** : 2 I/O (i-node + bloc données)

Q/R : Comment le SE Accède aux Données

Question 7 : Et pour le bloc logique 100 (> 12 = indirection) ?

Réponse (bloc logique 100 = simple indirection) :

1. Charger l'i-node en RAM
2. Lire le pointeur d'indirection simple (13e entrée)
3. Ce pointeur dit : "va lire le bloc 789 qui contient des adresses"
4. Lire le bloc 789 (bloc de pointeurs) $\Rightarrow$ trouver adresse du bloc logique 100
5. Lire le bloc de données demandé
6. **Résultat** : 3 I/O (i-node + bloc indirection + bloc données)

Conclusion : Accès plus coûteux pour gros fichiers, mais acceptable et flexible.

Fonction Principale du Système de Répertoires

Le lien entre l'humain et la machine

Avant qu'un fichier ne puisse être lu, il doit être ouvert. Le système d'exploitation utilise le chemin fourni par l'utilisateur pour localiser l'entrée correspondante dans le répertoire.

Définition Clé

La fonction principale du système de répertoires est de faire correspondre le **nom ASCII** du fichier aux **informations nécessaires pour localiser les données**.

Cette information de localisation dépend du système :

- Adresse disque du fichier entier (allocation contiguë)
- Numéro du premier bloc (listes chaînées)
- **Numéro d'i-node** (systèmes Unix modernes)

La Problématique des Attributs

Où stocker les métadonnées ?

Chaque système de fichiers doit maintenir des attributs pour chaque fichier (propriétaire, date de création, permissions, etc.).

Deux approches architecturales principales existent :

1. Stockage **directement dans l'entrée** du répertoire.
2. Stockage **dans une structure séparée** (i-node), référencée par le répertoire.

Approche 1 : Attributs dans le Répertoire

Structure simple (ex: systèmes MS-DOS anciens)

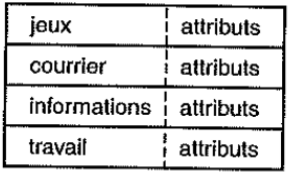

Figure: Approche A : Le répertoire contient les attributs et les adresses disque

Caractéristiques :

- Le répertoire est une liste d'entrées de taille fixe.
- Chaque entrée contient : le nom, la structure des attributs, et les adresses disque.
- **Avantage** : Simplicité de conception.

Approche 2 : Utilisation des I-nodes

Séparation des responsabilités (ex: Unix)

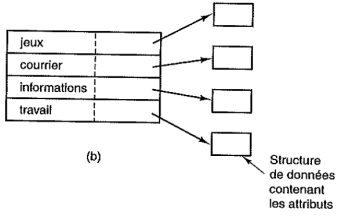

Figure: Approche B : Le répertoire pointe vers un i-node

Caractéristiques :

- L'entrée de répertoire est courte : **Nom de fichier + Numéro d'i-node**.
- Les attributs sont stockés dans l'i-node lui-même.
- Cette méthode présente des avantages significatifs pour la gestion des liens et l'efficacité (comme nous le verrons plus tard).

Structure d'une Entrée de Répertoire (UNIX)

Simplicité et Efficacité

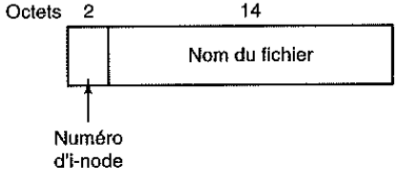

Figure: Structure d'une entrée de répertoire UNIX V7 (16 octets)

Une entrée de répertoire contient uniquement deux informations :

```
Struct DirectoryEntry {
    u_short inode_number;  // 2 octets : Index vers la table d'i-nodes
    char filename[14];     // 14 octets : Nom du fichier (pad avec 0)
};
```

Conséquence : Le nombre de fichiers par système est limité à 64K (2^{16}).

Les Attributs de l'I-node

Contrairement aux systèmes simples (FAT), les attributs ne sont **pas** dans le répertoire, mais dans l'**i-node**.

Contenu d'un i-node UNIX

- **Taille du fichier** (bytes)
- **Horodatage** (x3) : Création, Dernier accès, Dernière modification
- **Propriétaire et Groupe** (Owner/Group ID)
- **Permissions** (rwx)
- **Compteur de liens** (Link count)

Gestion des liens :

- Création d'un lien $\rightarrow$ incrémente le compteur.
- Suppression d'un lien $\rightarrow$ décrémente le compteur.
- Compteur à 0 $\rightarrow$ l'i-node est récupéré et les blocs libérés.

Stratégie d'Adressage des Blocs

L'approche multi-niveaux

Pour gérer efficacement petits et très gros fichiers, UNIX utilise une généralisation de l'i-node.

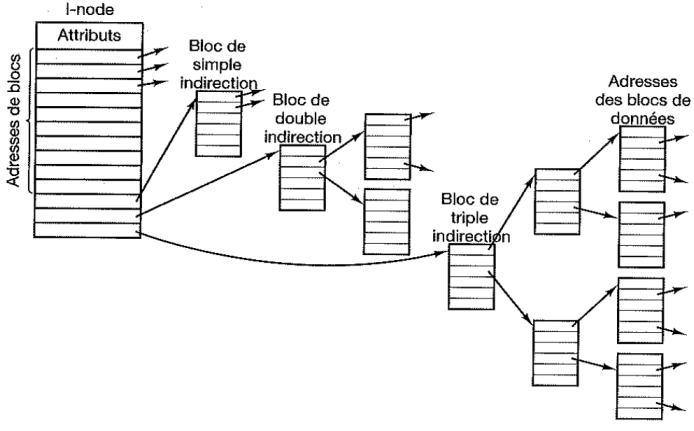

Figure: Structure d'adressage d'un i-node UNIX

Stratégie d'Adressage des Blocs

L'approche multi-niveaux

Pour gérer efficacement petits et très gros fichiers, UNIX utilise une généralisation de l'i-node.

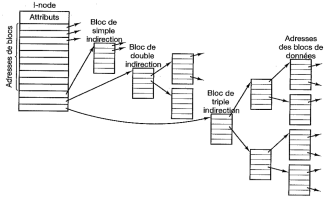

Figure: Structure d'adressage d'un i-node UNIX

- **Fichiers courts** : 10 premières adresses stockées directement dans l'i-node.
- **Fichiers moyens** : 1 pointeur vers un bloc *simple indirect*.
- **Fichiers larges** : 1 pointeur vers un bloc *double indirect*.
- **Fichiers géants** : 1 pointeur vers un bloc *triple indirect*.

Algorithme de Recherche de Fichier

Exemple : /usr/ast/mbox

Comment le système trouve-t-il les blocs de `/usr/ast/mbox` ?

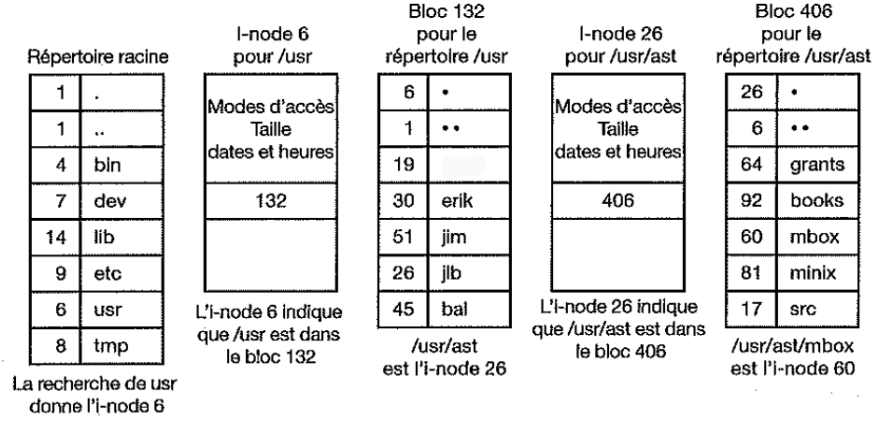

Figure: Étapes de résolution du chemin /usr/ast/mbox

Algorithme de Recherche de Fichier

Exemple : /usr/ast/mbox

Comment le système trouve-t-il les blocs de `/usr/ast/mbox` ? Le processus est itératif :

1. Répertoire Racine (/) : Le système commence toujours par la racine (dont l'i-node est connu, souvent le numéro 1 ou 2). Il lit le bloc de données de la racine (colonne de gauche) et cherche l'entrée "usr". Il trouve qu'elle correspond à l'i-node 6.
2. Répertoire /usr : Le système lit l'i-node 6 pour trouver son bloc de données (le bloc 132). Il lit ce bloc (contenu du répertoire /usr) et cherche l'entrée "ast". Il trouve l'i-node 26.
3. Répertoire /usr/ast : Il répète l'opération avec l'i-node 26, accède au bloc 406, cherche "mbox" et trouve finalement l'i-node 60.