350 Implémentation des systèmes de fichiers

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Types de systèmes de fichiers

- Nombreux types existent
- Wikipédia en liste et compare une centaine

Nombreux, car spécifiques

- À des systèmes et/ou organisations
  Contrôle de l'évolution, mais aussi syndrome NIH
- À des contraintes physiques des périphériques et des ordinateurs
- À des besoins spécifiques des utilisateurs

Types de systèmes de fichiers

Format de stockage

Spécifie la représentation des données sur disque

- Champs de bits
- Structures de données

Implémentation

- Dans les systèmes d'exploitation
- Dans les outils annexes (`mkfs(8)`, `fsck(8)`, etc.)

$\rightarrow$ C'est compliqué, surtout si le format n'est pas documenté

$\rightarrow$ Besoin maximal de fiabilité : ne pas manger les données !

Découpage en blocs

- Blocs de taille fixe, configurable, ou variable (ça dépend)
- Découpe tout l'espace disque
- Simplifie la gestion : blocs au lieu d'octets
- Tout est des blocs ensuite : données ou gestion

Limites principales

- Nombre maximal d'inodes
- Taille maximale d'un fichier
- Nombre d'entrées maximal par répertoire
- Taille maximale du volume
- Etc.

Exemples de limites

Type	Taille fichier	Taille volume
FAT32	4 Go	16Tb
NTFS	16 Eo*	16 Eo
ext4	16 To	1 Eo
btrfs	16 Eo	16 Eo
ZFS	16 Eo	256 Zo†

Ce sont des limites du format : la plupart des implémentations (systèmes et outils) ne les atteignent pas forcément.

*16 Eo $= 2^{64}$ octets $=$ 18 446 744 073 709 552 000 octets (20 chiffres)

†256 Zo $= 2^{78}$ octets $\approx 3 * 10^{23}$ (24 chiffres)

Détermination des limites

- Contraintes internes au type de système
  Tailles en octets de valeurs numériques
- Paramètres configurés par l'utilisateur
  Lors du formatage: `mke2fs(8)`, `mkfs(8)`, etc.
- Limites d'implémentations
  Le format peut stoker plus, mais les logiciels ne peuvent pas lire
- Combinaisons directes et indirectes de tout ça

Besoins et fonctionnalités

De base

- Stocker les données des fichiers (gros et petit)
- Stocker les métadonnées (y compris étendues `xattr(7)`)
- Stocker les entrées des répertoires
- Gérer l'espace libre (inodes et blocs)

Plus avancés

- Chiffrement et compression
- Journalisation (on y reviendra...)
- Instantanés (*snapshots*) et branches
- Déduplication
- Multi-volumes, RAID, etc.
- Somme de contrôle (*checksum*)
- Correction d'erreurs

Allocation et adressage des fichiers

Allocation contiguë

- Les blocs de données d'un fichier sont contiguës
- Exemple : ISO 9660 (CDs)
- Naïf : en général, la taille des fichiers est inconnue et évolue

Allocation chainée

- Un bloc de données connait l'adresse du suivant
- Exemple : FAT
- Problème : accès direct lent (`lseek(2)`)

Allocation indexée

- Un fichier connait la liste de ses blocs de données
- Problème : comment stocker des gros fichiers ?

Allocation indexée Unix

Pointeurs vers les blocs de données

- Pointeur direct : contient l'adresse d'un bloc de données
- Pointeur indirect : contient l'adresse d'un bloc contenant des pointeurs directs
- Pointeur indirect double : contient l'adresse d'un bloc contenant des pointeurs indirects
- Etc.

Question

- Comment déterminer où s'arrêtent les données ?

Adressage des fichiers - Exercice

Exemple : adressage indexé de ext2/3 (détails)

Dans la table des inodes

- Il y a 15 pointeurs de blocs :
  - 12 sont des pointeurs directs
  - 1 est indirect
  - 1 est indirect double
  - 1 est indirect triple
- Un bloc fait 4ko (défaut typique, mais configurable)
- Un pointeur de bloc est représenté sur 32 bits (4o)

Questions : quelles sont les tailles maximales

- D'un fichier si tous ses blocs sont pleins ?
- D'un fichier si la taille est codée sur 32 bits ?
- D'un volume si tous les blocs sont utilisés ?

Allocations modernes

Extents

- *extent* = suite de blocs contigus
- On stocke 2 nombres plutôt que tous les blocs de la suite
- ext4, btrfs, ntfs, etc.
- Problème: cf. allocation contiguë

Arbres B (*B-tree*)

- Structure de données arborescente équilibrée (voir INF3105)
- Adaptée aux systèmes de fichiers et base de données
- btrfs, zfs, ntfs, etc.
- Problèmes: nombreux détails algorithmiques

Journalisation

Problème : corruption

- Panne lors d'une écriture

$\rightarrow$ Données partiellement/mal écrites

$\rightarrow$ Incohérences données et métadonnées

Solution : écrire en deux temps

- On écrit les données dans un **journal**
- Quand le journal est écrit, on **recopie** dans le disque

$\rightarrow$ Problème de coût : écriture **plus chère** (copie intermédiaire)

Principe de la journalisation

Si panne pendant l'écriture dans journal

- On jette les données du journal (tant pis!)
- Les données du disque sont vieilles, mais **cohérentes**

Si panne pendant l'écriture du disque

- Le journal est complet et **cohérent**
- On termine l'écriture depuis le journal

Détails de la journalisation

- Nombreux détails spécifiques
- Configurations possibles

Question

- Journaliser seulement les métadonnées est-il un bon compromis ?

Copie sur écriture (*copy-on-write*)

Principe

- Au lieu de **modifier** quelque chose
- On en fait une **copie** modifiée
- On utilise la copie au lieu de l'original
- On libère l'original

$\rightarrow$ Plus efficace que la journalisation classique

Généralisation de l'approche

- Utilisé par ZFS et btrfs
- Instantanés : on peut garder d'anciennes versions
- Branches : des versions peuvent évoluer indépendamment