Systèmes de fichiers

Introduction

- Un processus à l'exécution peut stocker une quantité limitée d'information
  - Limitation due à l'espace d'adressage
- Quand le processus meurt, cette information disparaît
- Plusieurs processus peuvent avoir besoin d'accéder à la même information
- Les données doivent donc être indépendantes d'un processus donné
- Besoins
  - On doit pouvoir sauvegarder une grande quantité d'information
  - L'information doit survivre à la terminaison d'un processus
  - Plusieurs processus doivent avoir accès à l'information de manière concurrente

Introduction

- La solution à ces problèmes est de stocker l'information dans des *fichiers*
- Les processus peuvent lire/écrire/créer des fichiers
- Ils sont stockés dans une zone persistante
- La gestion des fichiers est de la responsabilité du système d'exploitation
- 2 points de vue
  - Utilisateur :
    - De quoi est fait un fichier
    - Comment un fichier est nommé
    - Quelles sont les opérations possibles…
  - OS designer
    - Comment gérer l'espace disque
    - Comment assurer de la fiabilité et de bonnes performances

Utilisateur : fichiers

- Abstraction importante : nommage des fichiers
- La plupart des FS supportent un nommage en 2 parties
  - Nom (nombre de caractères variables)
  - Extension
    - En général 1-3 caractères (DOS)
    - Plusieurs extensions possibles suivant les FS (.tar.gz…)
- En général l'extension n'est qu'une indication
  - Certains programmes insistent sur l'extension
- Windows associe des opérations aux extensions
  - Démarrage de certains programmes

Structure des fichiers

- Un fichier peut être structuré de multiples façons
- Non structuré
  - Le fichier est une séquence d'octets
  - La sémantique vient des applications
  - Utilisé par Linux/Windows
- Séquence d'enregistrements
  - Un fichier est une série de blocs de taille fixe
  - Une lecture retourne un enregistrement, une écriture écrit un enregistrement
  - Plus utilisé depuis les cartes perforées
- Arborescence
  - Un fichier est une structure arborescente
  - Enregistrements placés dans un arbre et triés suivant une clef
  - On peut un enregistrement suivant une clef donnée
  - Utilisée dans les gros systèmes de gestion de données

Types de fichiers

- Les Os supportent différents types de fichiers
- *Regular Files* : contient les données utilisateurs
  - ASCII : facilement manipulables
  - binaires : format spécifique pour être exécuté
    - Header (contient un *magic number* pour éviter une exécution accidentelle)
    - Text
    - Data
    - Relocation bits
    - Symbol table (utilisé pour debug)
- *Directories* : maintiennent la structuration du FS
- *Character Special Files* : Liés aux I/O et utilisés pour les périphériques à accès séquentiel
- *Block Special Files* : Utilisés pour représenter les disques

Accès aux fichiers et attributs

- Accès séquentiel
  - Les octets sont lus dans l'ordre
  - Impossible de sauter/revenir en arrière
  - Reprise depuis de début du fichier possible
- Accès aléatoire
  - Les octets sont lus dans n'importe quel ordre
  - Utilisation de *read* avec paramètre ou *seek*
- Les OS ajoutent des informations aux fichiers, ce sont les attributs
  - Dates création/modification
  - Protection (droits d'accès)
  - Password
  - Flags (archive, hidden…)…

Mapping de fichiers en mémoire

- Utiliser un fichier consiste à utiliser des *read/write*
- Moins pratique que l'accès mémoire
- MULTICS a introduit la possibilité de mapper un fichier sur la mémoire
  - Appels systèmes *map* et *umap*
- Exemple:
  - Un fichier est mappé à l'adresse 2048K
  - Lire 1 octet à cette adresse effectue la lecture du premier octet du fichier
  - Écrire dans cette zone mémoire provoque l'écriture dans le fichier
- Limitations/difficultés
  - La mémoire est gérée par pages, donc un fichier crée avec ce mécanisme a une taille égale à un nombre de pages
  - Problématique si 1 processus accède en mapping et un autre en normal
  - Si le fichier est plus grand que l'espace adressable, l'OS doit pouvoir n'en ouvrir qu'une partie

Disposition d'un FS

- Les FS sont stockés sur des disques
- Les disques sont souvent divisés en partitions
- Le secteur 0 du disque est le *Master Boot Record*
  - Utilisé pour démarrer l'ordinateur
  - La fin du MBR contient la table des partitions
  - Une partition est marquée active
- Quand l'ordinateur démarre
  - Le bios lit et exécute le MBR
  - Le MBR trouve la partition active et lit les premiers blocs : *boot block*
  - Ce programme charge le système d'exploitation
- Le contenu d'une partition varie suivant l'OS
- L'espace est divisé en blocs, unité d'allocation la plus petite
- Il faut maintenir une association entre les fichiers et les blocs sur le disque

Allocation contigu

- Un fichier est une suite de blocs contigus sur le disque
- Localisation d'un fichier = adresse du premier bloc et nombre de blocs
- Lecture très performante, il suffit de se placer au premier bloc. Pas besoin de chercher les blocs
- Problèmes
  - Au cours du temps, des trous apparaissent (fichiers supprimés) : fragmentation
  - On peut défragmenter le disque mais très coûteux
  - On peut réutiliser les espaces libres mais il faut maintenir une liste, et connaître la taille finale d'un fichier
  - Utilisé dans les CDRoms

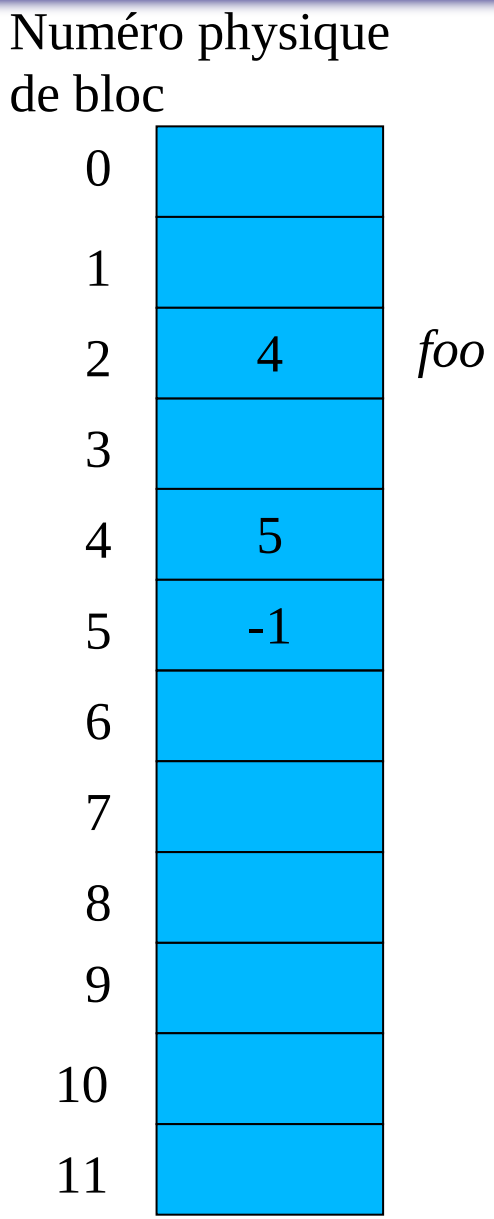
Allocation par liste chaînée

- Un fichier est une série de blocs maintenus dans une liste chaînée
- Le début d'un bloc est utilisé comme pointeur vers le suivant
- Pas de fragmentation externe
- Fragmentation interne sur le dernier bloc
- Un fichier est trouvé par son adresse de premier bloc
- La lecture séquentielle est très rapide
- Accès aléatoire très lent

Allocation par liste chaînée et table

- On peut supprimer les problèmes des listes chaînées en sortant le pointeur du bloc
- Les pointeurs sont placés dans une table en mémoire
  - C'est une File Allocation Table (FAT)
- Le bloc redevient donc entièrement disponible pour les données
- L'accès aléatoire revient à suivre des références en mémoire
- Mais nécessite beaucoup de mémoire
  - Un disque de 20GB avec des blocs de 1KB a une table de 20 millions d'entrées
  - Chaque entrée nécessite 3 ou 4 octets => 60-80MB

Allocation par liste chaînée et table



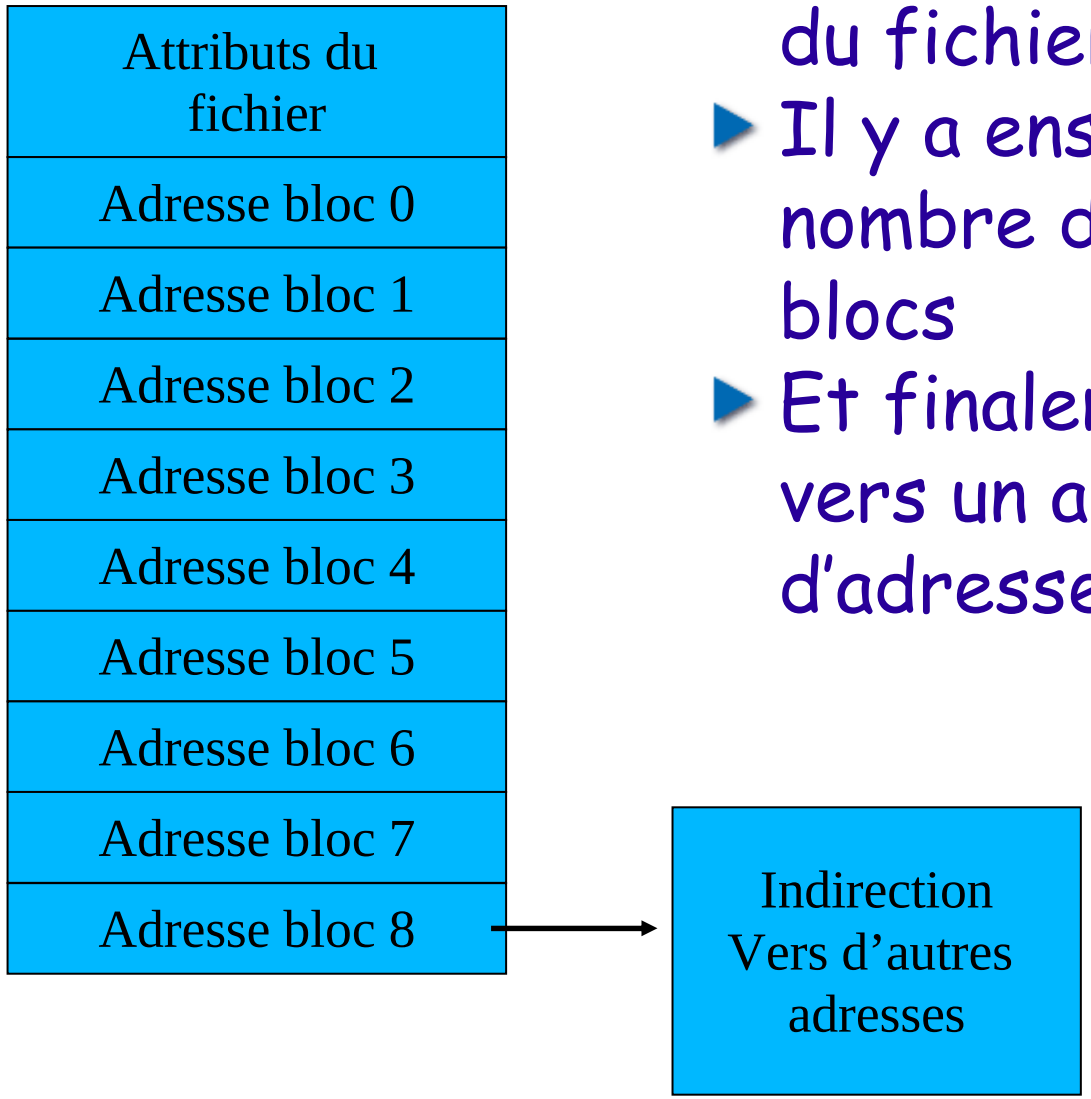
- Le fichier *Foo* commence au bloc 2
  - Le bloc suivant est le 4
  - Le dernier est le 5
- Pour lire le fichier
  - On commence par lire le bloc 2 sur le disque
  - Ensuite on regarde dans la FAT le pointeur 2
  - On lit le bloc 4 sur le disque, et on continue
- Les blocs libres n'ont pas de pointeurs
- Les blocs terminaux ont une adresse de bloc non valide (-1)

Inodes

- On associe à chaque fichier un *Index-Node* (i-node)
- Il contient les attributs et les adresses des blocs sur le disque
- L'avantage par rapport à une FAT est que seul l'inode d'un fichier ouvert a besoin d'être en mémoire
- Une FAT augmente linéairement avec la taille du disque
- La mémoire occupée par les inodes augmente avec le nombre de fichiers ouverts
- Problème
  - Si un inode a une taille max, il ne peut contenir qu'un nombre limité d'adresses
  - Un fichier aura donc une taille maximale
  - Utilisations de plusieurs indirections (UNIX)

Inodes

- L'inode contient les attributs du fichier
- Il y a ensuite un certain nombre de références vers des blocs
- Et finalement une indirection vers un autre groupe d'adresses



Répertoires

- La fonction principale du système de répertoire est de maintenir l'association entre le nom du fichier et les informations pour le trouver sur le disque
- On peut aussi stocker les attributs du fichier dans les entrées du répertoire
- Implémentation simple
  - Un répertoire est une liste d'entrées
    - Nom du fichiers
    - Attributs
    - Blocs sur le disque
- Systèmes avec inode
  - L'entrée du répertoire est juste le nom du fichier et l'inode

Noms de fichiers longs dans les répertoires

- On peut réserver suffisamment d'espace pour les noms (255 caractères)
  - Gâchis la plupart du temps
- Entrée de répertoire de taille variable
  - Chaque entrée dans un répertoire contient une partie fixe (attributs)
  - Suivie du nom de taille variable
  - Création de fragmentation après suppression
  - Mais défragmentation possible/facile
- Entrées de taille fixe et tas
  - Les entrées ont toujours une taille fixe
  - Les noms sont mis à la fin de la structure de répertoire, dans un tas
- Pour trouver un fichier, on parcourt séquentiellement les entrées du répertoire
  - Mais peut être coûteux
  - Possibilité d'utiliser une table de hachage avec chaînage

Fichiers partagés

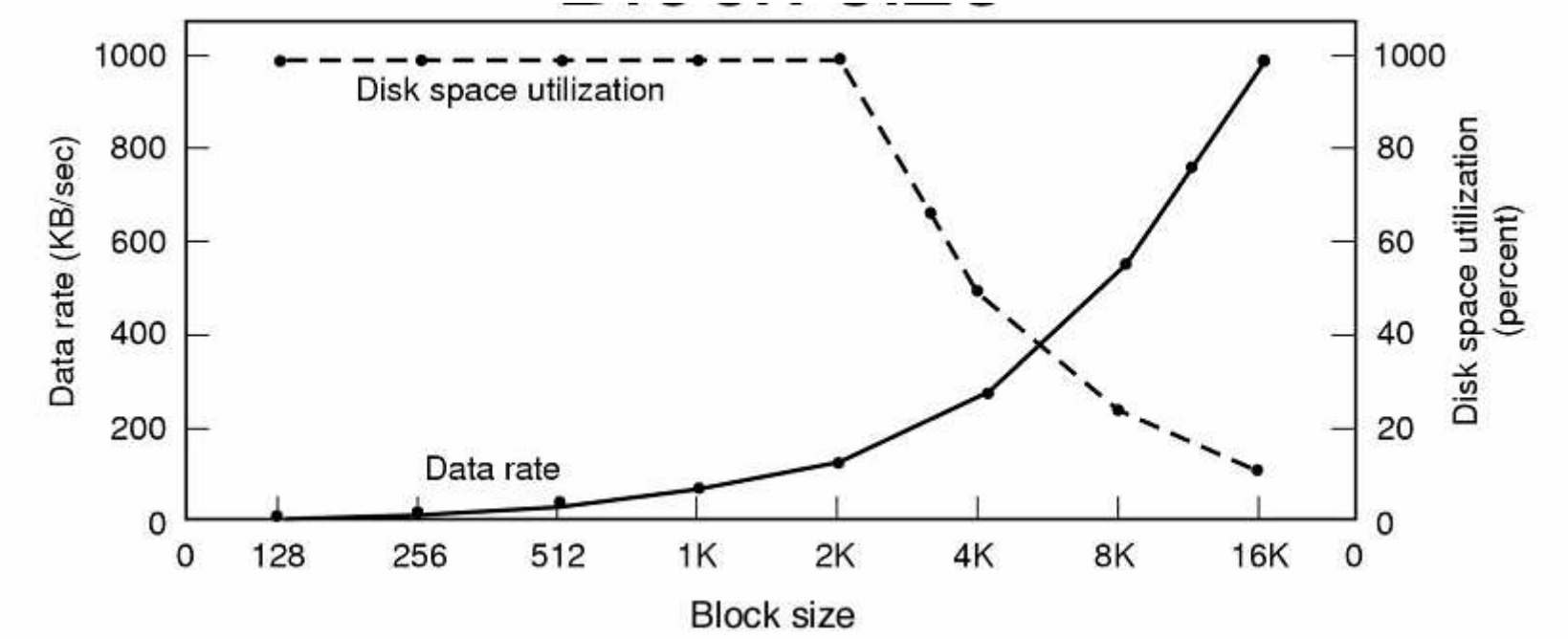
- Comment permettre à un même fichier d'apparaître dans des répertoires différents
  - /A/B/C/foo et /A/B/D/foo sont le même fichier
  - Le système de fichiers n'est plus un arbre mais un DAG
- Cas des entrées de répertoire contenant les adresses des blocs
  - Copie de ces adresses dans chacun des répertoires
  - Problème quand ajout de blocs
- Utilisation d'inodes
  - Mais qui est le propriétaire?
- Utilisation de liens (LINK)
  - L'entrée dans le répertoire D contient le chemin vers l'entrée du répertoire C
  - Ajout un compteur de liens à l'inode (Hard links)
  - Mais si on retire le fichier de C, il n'est pas vraiment effacé car toujours utilisé par D : problèmes de quotas
  - Si liens symboliques, pas de compteur dans l'inode, « disparition » de fichiers possible…
  - Création de fichiers lors de sauvegardes

Gestion de l'espace disque

- On ne considère que le cas d'une gestion de l'espace par blocs non contigus
- Les blocs sont de taille fixe, mais quelle taille?
  - Un bloc trop gros amènera du gâchis d'espace
  - Des blocs trop petits sont coûteux, il faut les lire sur le disque (déplacement du bras…)
- La bonne taille dépend de la taille des fichiers
  - Mullender et Tanenbaum 1984: Taille médiane de fichiers Unix 1KB
  - Avoir des blocs de 32KB est une perte de 97% d'espace
- Temps pour lire les donnés sur le disque
  - Déplacement du bras (*seek time*)
  - Déplacement du plateau (*rotational delay*)
  - Transfert

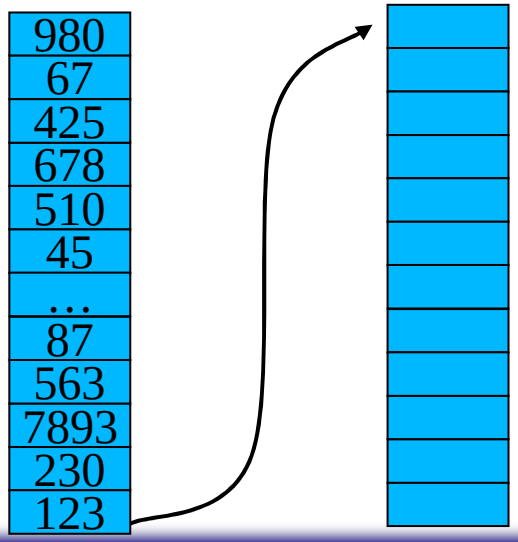
Débit et espace en fonction de la taille des blocs

- D'après *Modern Operating Systems, Tanenbaum*
- Fichiers de 2KB



Espace libre

- Comment maintenir la liste de blocs libres
- Liste chaînée de blocs
  - Chaque bloc contient le numéro de blocs libres
  - Exemple: les blocs du disque font 1KB et un numéro de bloc est codé sur 32 bits
    - On peut y mettre 256 valeurs sur 32 bits
    - Donc ce bloc contient l'adresse de 255 blocs libres, et l'adresse d'un autre bloc de la chaîne



Espace libre

- On peut aussi utiliser une bitmap
- Un disque de *n* blocs utilise un tableau de *n* bits
- Les 0 représentent les blocs occupés, les 1 les blocs libres
- Beaucoup moins coûteux en espace que la liste chaînée
- Permet aussi d'avoir une allocation proche en espace

10101110001110001110001110
10101110000000001110001110
10101110111111111110001110
10001000000000000000000000

Performance - Caching

- Pour améliorer les performances, on met les blocs dans un cache
- Utilisation d'une table de hachage avec chaînage
  - Clé = hache du numéro du périphérique et adresse du bloc
- Quand le cache est plein, un bloc est choisi et écrit sur disque pour faire de la place
  - LRU souvent utilisé
- Crash
  - Quand le système plante, les blocs en cache sont perdus
  - Mais tous les blocs ne sont pas équivalents
    - Un bloc d'inode est plus important qu'un bloc de data
  - LRU modifié pour tenir compte de l'importance
- Mise à jour forcée
  - Unix: Appel de *sync()*
  - MS-Dos : *Write Through Cache*
    - Très utile pour les média amovibles

Performance – *Read ahead*

- Essaie de mettre en cache les blocs avant qu'ils ne soient demandés
- En supposant une lecture séquentielle des fichiers
  - On demande le bloc *k* au FS
  - Il va regarder si *k+1* est dans le cache, et l'y mettre
- Contre productif si lecture aléatoire
  - Lecture depuis le disque de blocs inutiles
  - Encombrement du cache
- Hypothèse et vérification par l'OS
  - Tout fichier est « séquentiel »
  - Si un *seek* est fait, il devient « aléatoire »
  - Permet d'avoir de bons résultats

Performance – Réduire les mouvements de bras

- Chercher un bloc sur le disque est très coûteux
- Réduire les mouvements du bras
  - Mettre les blocs susceptibles d'être accédés séquentiellement proches
- Facile avec une gestion par *bitmap*
  - Regarder les endroits ou il y a des 1 successifs
- Plus compliqué avec une liste de blocs libres
  - Solution simple, gérer des groupes de blocs consécutifs dans la liste
- Placement des inodes
  - Toujours accédés en premier pour trouver un fichier
  - Habituellement placés à l'extérieur du disque
  - Optimisation simple : les placer au centre du disque
    - Divise le déplacement des bras par 2

Journalisation

- Les FS sont d'énormes structures de données en constantes modifications
- Exemple : Effacement d'un fichier Unix
  - Supprimer l'entrée dans le répertoire
  - Marquer l'inode comme libre
- Une interruption brutale peut poser des problèmes
  - Inodes marqués comme utilisés…
- Solution simple : scanner le FS à la recherche d'incohérences après un plantage
  - Coûteux (*fsck*)
- Utilisation d'un journal où les changements à faire sont écrits
  - Si plantage, on regarde le journal pour faire les modifications
  - Si succès, on efface l'entrée du journal
- Souvent le journal ne sert pas à préserver l'intégrité des données mais du FS!

Journalisation

- Journalisation totale
  - Les données sont d'abord écrites dans un journal
  - Puis elles sont commitées dans le FS
  - Coûteux (double écriture)
  - Robuste
- Journalisation des méta data
  - Seules les opérations sur la structure du FS sont inscrites dans un journal
  - Compromis entre performance et fiabilité
  - Peut induire de la corruption de données
- Exemple : Ajout en fin de fichier
  1. Augmentation de la taille du fichier dans l'inode
  2. Allocation de l'espace disque
  3. Ajout des données
- Si plantage le fichier contiendra n'importe quoi
- Solution
  - Commiter après l'écriture des données sur le disque
  - Mais nécessite de jouer avec le cache disque qui réordonne les demandes d'écriture

FAT

FAT

- Système de fichiers de MS-DOS
- Noms de fichiers 8+3
- Entrées de répertoire
  - Structure de 32 octets
  - Contient le nom de fichier, les attributs, heure et date de création, adresse du premier bloc et taille du fichier
- Attributs
  - *Read Only*
  - *Archive* : mis à 1 lors de la modification d'un fichier par certains programmes, mis a 0 lors de leur sauvegarde
  - *Hiden* : masque le fichier lors d'un *dir*
  - *System* : masqué et non effacable
- Date
  - Date de création/modification
  - Heure en secondes codée sur 2 octets => précis a 2secondes
  - Année (depuis 1980) codée sur 7 bits => Max 2107

FAT

- MS-Dos utilise une FAT en mémoire pour les blocs des fichiers
- L'entrée du répertoire donne l'adresse du premier bloc d'un fichier
- La taille du fichier est sur 32 bits => 4GB max par fichier en **théorie**
- 3 versions pour MS-Dos suivant le nombre de bits dans l'adressage disque
  - FAT-12
  - FAT-16
  - FAT-32 (mais seulement 28 utilisables)
- La taille des blocs est toujours un multiple de 512 octets
  - FAT-12 : 512B, 1KB, 2KB, 4KB
  - FAT-16 : 2-32KB
  - FAT-32 : 4-32KB

FAT

- Taille maximale des partitions
  - Chaque entrée dans la FAT a une taille définie (12,16,28 bits)
  - Dépend de la taille des blocs
  - FAT 12 : 2^{12} x 512B = 2MB (moins en pratique)
  - FAT 32 : 2^{28} x 2^{15} = …
    - Beaucoup moins en fait car le système garde des informations sur les secteurs de 512B sur 32 bits
    - En pratique 2^{9} x 2^{32} =2TB
- Taille occupée en mémoire par la FAT
  - Il faut calculer le nombre d'entrées et la taille de chacune
  - Nombre d'entrées = taille max / taille des blocs de la partition
  - FAT12 : 2^{12} x 2B = 8KB
  - FAT32
    - Partition de 2 GB et blocs de 4KB : 2GB / 2^{14} = 2^{19} = 2MB

ext2FS

ext2fs

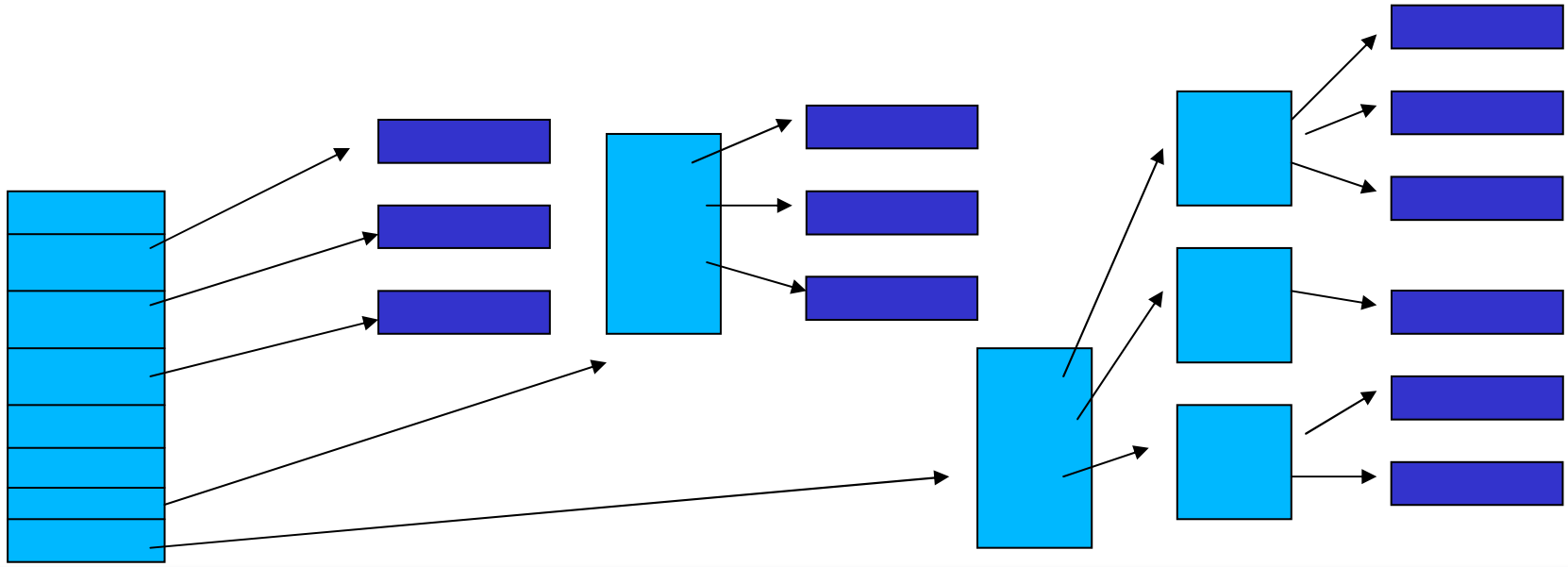
- *Second Extended File System*
- Désigné par Rémy Card pour améliorer extfs, remplaçant du FS de Minix utilisé par Linux
  - Adresses de blocs codés sur 16 bits
  - Nom de fichier de 14 caractères max dans un répertoire
- Adresse les blocs sur 32 bits et permet des noms de fichiers de 255 caractères
- Utilise les concepts des premiers systèmes de fichiers Unix
  - Inode : représentation d'un fichier
  - Répertoires : liste de fichiers
- Le disque est divisé logiquement en blocs
  - Taille sélectionnable à la création : 1K,2K,4K
- Un nombre fixe de blocs est regroupé en *group block*
- Chaque *group block* peut contenir un nombre maximum d'*inodes* décidé à la création du FS

ext2fs – blocks group

- Chaque *block group* contient un *group descriptor* qui contient les informations suivantes
- Une bitmap d'allocation des blocs du groupe
- Une bitmap d'allocation des inodes du groupe
- Le numéro de bloc où se trouve les inodes de ce groupe
- Des informations statistiques
  - Nombre de blocs libres
  - Nombre d'inodes libres
  - Nombre de répertoires utilisés
- Dans un *block group* le deuxième bloc contient l'ensemble des *group descriptor*
  - Sauvegarde
  - Le kernel utilise la copie du groupe 0

ext2fs - inode

- Un fichier est représenté par un inode qui contient
  - Le type de fichier
  - Les droits d'accès
  - Le propriétaire
  - La date de création/modification
  - La taille
  - Des Pointeurs vers les blocs de données
- Un inode utilise une double indirection pour les données



ext2fs - inode

- Chaque inode occupe 128B sur le disque (XXX A VOIR XXXX)
  - Par défaut 1 inode / 4KB (*mke2fs)*
- Des études montrent que les fichiers sont souvent petits
  - Un inode peut contenir l'adresse de 12 blocs de données
  - La 13ème entrée contient l'adresse de la première indirection
  - La 14ème la deuxième indirection
  - La 15ème la troisième indirection
- Un inode contient des adresses sur 4 octets
- Taille maximale d'un fichier en adressage directe
  - 12 * taille Bloc
- Si taille supérieure, utilisation d'une indirection
- Première indirection
  - L'adresse d'un bloc qui contient des adresses de blocs de données
  - Combien d'adresses? tailleBloc/4B
- Deuxième indirection
  - Adresse d'un bloc qui contient des adresses de blocs qui contiennent des adresses de blocs de données

ext2fs - répertoires

- Les répertoires sont structurés dans un arbre hiérarchique
- Un répertoire est un fichier contenant une liste d'entrées
  - Utilisation des indirections
- Chaque entrée est un numéro d'inode et un nom
- Quand un programme ouvre un fichier
  - Le noyau regarde le chemin
  - Trouve le numéro d'inode correspondant
  - Charge l'inode en mémoire

ext2fs - superblock

- Le superblock contient les informations internes sur le système de fichier
- Il est toujours localisé à l'offset 1024 du périphérique
- Sa taille est de 1024B
- Il est copié dans tous les *block groups* pour sauvegarde
- Il contient
  - Les paramètres fixes décidés à la création
    - Nombre total d'inodes dans le fs
    - Nombre total de blocs
    - Nombre de blocs par groupe et d'inodes par groupe
  - Les paramètres modifiables
    - Compteur de montage
    - Nombre de blocs réservés au super utilisateur
  - Des informations
    - Date de la dernière vérification
    - Nombre de blocs et inodes libres
- Identification
  - Le superblock est identifié par un *magic number* (0xEF53)

NT File System

NTFS

- Système de fichiers développé spécifiquement pour Windows NT
- Utilise des adresses 64 bits pour les blocs
- Noms de fichiers : 255 caractères
- Chemins : 32767 caractères
- Utilisation de l'Unicode en natif
- Gestion de la casse
- Un fichier NTFS n'est pas une séquence d'octets
  - C'est un ensemble de d'attributs
  - Chaque attribut est un flux d'octets (*stream*)

streams

- Les *streams* proviennent originellement du FS des Macs
  - Pour une image, données dans le stream principal, thumbnail dans un autre stream
- La plupart des fichiers ont
  - Un *stream* pour le nom
  - Un *stream* pour sont ID (64bits)
  - Un *stream* pour les données
    - Mais un fichier peut avoir plusieurs de *streams* de données
- Les *streams* sont nommés
  - nomFichier:nomStream
- Chaque *stream* a une taille et peut être verrouillé séparément des autres
- La taille maximale d'un *stream* est 2^{64} soit 18.4 exa-octets (1 milliard de Go)
- Exemple d'utilisation
  - echo Toto > hello.txt:hidden
  - Utilisation d'outils pour les manipuler (lads)

Structure

- Chaque partition NTFS contient
  - Des fichiers
  - Des répertoires
  - Des bitmaps
  - Des structures de données
- Une partition est découpée en succession de blocs (*clusters*)
  - Taille de 512B a 64KB, en général 4KB
- Master File Table
  - Séquence linéaire d'entrées de 1KB
  - Chaque entrée décrit 1 répertoire ou 1 fichier
  - Contient les attributs (nom, date…) et les adresses des blocs sur disque
  - Un gros fichier peut nécessiter plusieurs entrées pour les adresses
- Le MFT est un fichier
  - peut être placé n'importe ou sur le disque
  - Peut contenir 2^{48} entrées

Structure

- Les entrées d'un MFT sont des couples *header-value*
- Le *header* indique l'attribut et la taille de sa valeur
  - Si valeur petite, placée directement dans l'entrée
  - Sinon pointeur vers valeur
- Les 16 premières entrées de la MFT (0-15) sont réservées pour les fichiers qui décrivent le FS
  - Commencent tous par $
  - Entrée 0 : Infos sur le fichier MFT!
  - Entrée 1 : Copie de la première
  - Entrée 2 : Journal
  - Entrée 6 : Adresse du fichier bitmap pour l'espace libre
  - Entrée 8: Liste des clusters défectueux

structure

- Comment le système trouve la MFT?
  - C'est un fichier sur disque, donc n'importe où
  - L'adresse de la première entrée de la MFT est inscrite dans le secteur de boot à l'installation
- Enregistrement données
  - Contient le nom du stream si présent
  - Ensuite, le fichier lui-même (si petit) ou les adresses des blocs sur le disque
- Les blocs sont normalement attribués consécutivement
  - On met dans l'enregistrement l'adresse du premier bloc et le nombre de blocs (2 x 64bits)
  - Si « trous » alors on met les autres enregistrements à la suite
  - Si trop de groupes de blocs alors utilisation d'un autre enregistrement
- Le nombre d'entrées pour 1 fichier dépend des allocations de blocs
- Des méthodes de compression sont utilisées pour réduire la taille des adresses à 32 bits

Systèmes de fichiers journalisés

ext3fs

- Moins performant que d'autres systèmes journalisés comme ReiserFS ou XFS
- Ajoute à ext2 un système de journal
- 3 niveaux de journalisation
  - *Journal* : Méta données et données sont écrites dans un journal
  - *Writeback* : seules les méta données sont journalisées
  - *Ordered* : méta données journalisées avec écriture des données en premier

Network File System

Systèmes de fichiers distribués

- Un système de fichier centralisé permet à plusieurs utilisateurs d'accéder aux fichier d'un unique Système
- Les *DFS* permettent l'accès depuis plusieurs machines reliées par un réseau
- Ils sont implémentés en client-serveur
  - Client : utilise le fichier
  - Serveur : stock le fichier
- Propriétés importantes
  - *Network Transparency*
  - *Location Transparency*
  - *Location Independence*
  - *User Mobility*
  - *Fault Tolerance*
  - *Scalability*
  - *File Mobility*

Propriétés

- *Network Transparency*
  - Un client doit pouvoir accéder à un fichier distant en utilisant les même opérations que localement
- *Location Transparency*
  - Le nom d'un fichier ne doit pas trahir sa localisation
- *Location Independence*
  - Le nom d'un fichier ne doit pas changer quand sa localisation physique change
- *User Mobility*
  - Un utilisateur doit pouvoir accéder à un fichier depuis n'importe quelle machine du réseau

Propriétés

- *Fault Tolerance*
  - Le système doit continuer à fonctionner lors d'une panne d'une machine ou d'une partie du réseau
  - Il peut néanmoins voir ses performances dégradées ou avoir une partie des fichiers non accessibles
- *Scalability*
  - Le système devrait passer à l'échelle si la charge ou le nombre de machines augmente
- *File Mobility*
  - Possibilité de déplacer un fichier d'une machine à une autre à l'exécution

Choix d'implémentations

- Plusieurs points sont difficiles à implémenter dans un DFS
- Espace de Nommage
  - Espace uniforme
  - Espace hétérogène : un client peut monter un fichier distant dans une hiérarchie différente d'un autre client
- *Stateful/Stateless operations*
  - Un serveur *stateful* conserve des données entre les appels d'un client
  - Ces informations sont utilisées pour traiter les requêtes suivantes
  - Ex : open/read/seek…
  - Dans un système *stateless* chaque requête contient toutes les informations nécessaires
    - Le client fournit l'offset à chaque fois
  - En général
    - *Statefull* : plus performant mais plus difficile à implémenter
    - *Stateless* : moins performant mais plus facile à implémenter

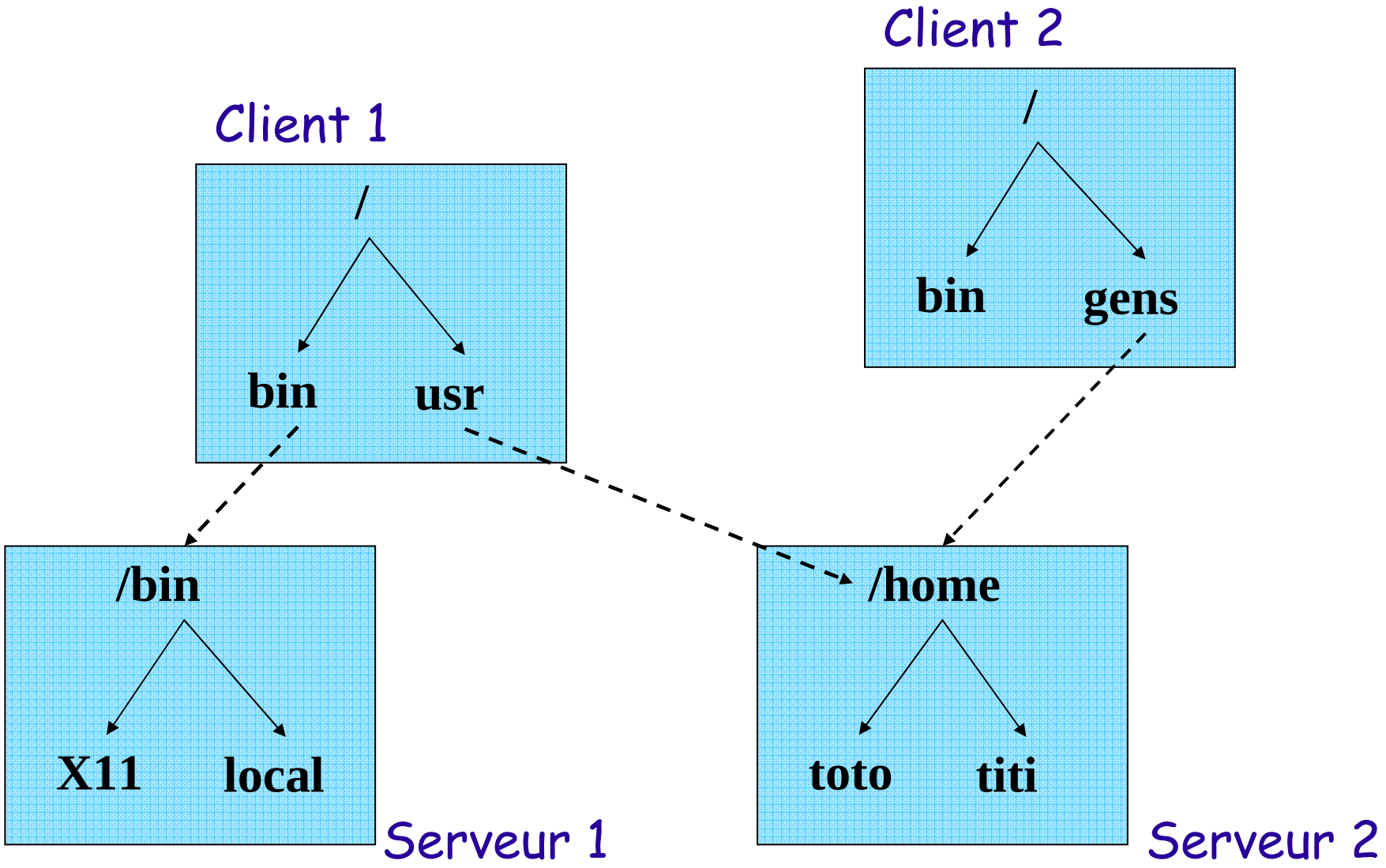
Choix d'implémentation

- Sémantique de partage
  - Quelle sémantique appliquer quand plusieurs clients accèdent au même fichier ?
  - Sémantique Unix
    - une modification est visible par tous
  - Sémantique session
    - Les modifications ne sont visibles que lors d'*open/close*
- Méthodes d'accès distant
  - Si pure client-serveur, les actions sont initiées par le client, le serveur ne fait qu'exécuter ce qu'on lui demande
  - Mais dans les systèmes *stateful* le serveur a souvent un rôle plus actif
  - Il participe à la cohérence des caches, notifie les clients quand le cache n'est plus valide…

Network File System

- Présenté par Sun en 1985
- C'est avant tout un protocole
  - Sun a fourni également une implémentation de référence
- Architecture client-serveur
  - Un serveur de fichier exporte un ensemble de fichiers
  - Les clients sont des machines qui accèdent a ces fichiers
  - Une machine peut être client et serveur pour des fichiers différents
  - Mais le code est séparé pour le client et le serveur
- Communications par *Remote Procedure Call* (RPC)
  - Requêtes synchrones

Exemple


Client 2
/
bin
gens
Client 1
/
bin
usr
/bin
X11
local
Serveur 1
/home
toto
titi
Serveur 2

Objectifs

- NFS ne doit pas être limité aux Unix
  - N'importe quel OS doit pouvoir l'implémenter
- Le protocole ne doit pas dépendre du hardware
- Les mécanismes de reprise sur panne doivent être simples
- Les applications doivent pouvoir accéder aux fichiers de manière transparente, sans recompilation ou librairie ou nom spécifique
- La sémantique Unix doit être maintenue pour les clients Unix
- Les performances doivent être comparables à celles d'un accès local
- L'implémentation doit être indépendante de la couche transport

Composants NFS

- Une implémentation NFS est constituée de plusieurs composants
- Certains sont du coté client, d'autres serveurs, ou communs
- Certains composants ne sont pas indispensables pour les fonctionnalités de base
- *NFS Protocol*
  - Défini l'ensemble des requêtes pouvant être effectuées par le client ou le serveur, ainsi que les arguments et valeurs de retour
  - Version 1 jamais publique, NFSv2 supporté par toutes les implémentations
- RPC protocol
  - Format des interactions entre client et serveur
  - Les requêtes NFS sont envoyées dans des paquets RPC

Composants NFS

- *Extended Data Representation*
  - Mécanisme indépendant du hardware pour encoder les données à envoyer
  - Toutes les requêtes RPC utilisent XDR
- *NFS server code*
  - Traite les requête client et gère l'accès aux fichiers exportés
- *NFS Client Code*
  - Implémente les appels du client avec des requêtes RPC
- *Mount Protocol*
  - Fournit la sémantique des opérations *mount/unmount*
- *Deamon processes*
  - Sur le serveur *nfsd* écoute et répond aux clients, *mountd* traite les requêtes de montage
  - Sur le client, des *biod* traitent les I/O asynchrones pour des blocs de fichiers NFS
- *Network Lock Manager* et *Network Status Monitor*
  - Permettent de verrouiller des fichiers
  - Ces fonctionnalités ne sont pas dans le protocole de base

XDR

- Des machines d'architecture différente traitent les données de façons différentes
  - Ordre des octets, taille des types (entier…)
  - Chaînes de caractère, tableaux
- Quand les données doivent être interpretées par des machines différentes, problème
  - Ex: 259_{10} sur un big-endian vaudra 769_d sur un little endian
- XDR définit une représentation indépendante de la machine
  - Entiers
    - 32 bits avec octet le plus significatif à gauche
    - Complément à 2 pour le signe
  - Chaînes
    - Un champs longueur suivi par le code ASCII de la chaîne
    - Ajout de NULL pour arriver à 4 octets, pas de NULL obligatoire à la fin
  - Tableaux
    - Champs longueur suivi des valeur dans l'ordre naturel
    - Toutes du même type mais taille multiple de 4 octets
  - Structures
    - Encodées dans l'ordre naturel, avec chaque élément multiple de 4 octets

NFS est sans état (*Statelessness*)

- Le serveur NFS ne maintient aucune information concernant les clients
- Chaque requête est indépendante des autres et contient toutes les informations nécessaires pour son traitement
- Pas de requête pour ouvrir/fermer un fichier
  - Sinon on aurait un état
- Permet de reprendre facilement après des pannes
  - Si le client crash
    - le serveur ne le sait pas
    - Apres reboot, le client remonte les fichiers et relance les applications
  - Si le serveur crash
    - le client n'obtient pas de réponses
    - Il continue à envoyer la même requête
    - Quand le serveur revient, le client a une réponse
    - En fait, le client ne sais pas si le serveur a planté ou été juste lent…
- Ne pas avoir d'état implique de commiter immédiatement les modifications du côté serveur

Implémentation

- Utilise la partie de VFS consacrée aux systèmes de fichiers réseau
  - Vnode au lieu de Inode
- *File Handle*
  - Le procotol NFS associe un *file handle* à chaque fichier ou répertoire
  - Le serveur le génère quand le client accède à un fichier pour la première fois (ou création)
  - Il est retourné au client qui l'utilise pour les appels suivants
  - Le client traite cet objet comme un entier 32 bits
  - Contenu typique
    - Id du système de fichiers
    - Inode
    - Numéro de génération de l'inode
      - Permet de distinguer une réutilisation d'un inode
      - Évite de servir le mauvais fichier

Permissions

- Dans un FS Unix local, les permissions d'un fichier ne sont vérifiées que lors d'un open
- Si les permissions sont changées par la suite, il est toujours possible d'écrire/lire dans le fichier
- NFS n'a pas de notion de fichier ouvert
  - Risque de refus d'écriture alors qu'un open a été fait avant
  - le serveur NFS autorise toujours la lecture/écriture d'un fichier par son propriétaire
  - Quand un fichier est ouvert, le client récupère les permissions du fichier en plus du handle
  - La sécurité est donc assurée par le client

Suppression de fichier

- Un supprimé est retiré de l'entrée du répertoire correspondant
- Mais toujours accessibles par les processus qui l'avaient déjà ouvert
- Mais NFS ne connaît pas la notion de fichier ouvert
- Modification du client
  - Quand le client détecte une demande de suppression, il change l'opération en renommage
  - Le choix du nouveau nom est suffisamment bon pour éviter un conflit
  - Permet d'éviter une nouvelle ouverture du fichier par un autre processus
  - Ceux qui accèdent à ce fichier par le même client n'ont pas de problème
  - Quand le dernier processus a fini, le client demande la suppression
  - Par contre, ceux qui utilisent un autre client ne voient plus le fichier… (*stale file handle*)
  - Si le client crash, fichier inutile sur le serveur

Lecture - écriture

- Une opération de lecture/écriture est atomique
  - Si 2 processus demandent à écrire sur le même fichier, leurs demandes seront sérialisées
- Du coté client NFS, même principe, une sérialisation est effectuée
- Mais si requêtes de machines différentes?
  - Une demande d'écriture peut nécessiter plusieurs appels RPC
  - Mais le serveur est sans état, donc il ne sait pas quand finit une lecture/écriture
- Verrouillage du fichier avec le NLM
  - Mais seulement conseillé…

Performance

- NFS devait être performant comme un disque local
- Il faut bien choisir la métrique
  - Débit : non
  - Temps pour des opérations « courantes » : oui
- Bottlenecks
  - Les opérations qui modifient un fichier sont lentes car elles doivent être commitées immédiatement sur disque
  - Demander les attributs d'un fichier nécessite un appel RPC
    - ls -l génère beaucoup de requêtes
  - Si le serveur ne répond pas, le client renvoie la même requête
    - Un serveur lent deviendra chargé inutilement

Performance

- Cache coté client
  - On met en cache des blocs et les attributs d'un fichier coté client
  - Mais dangereux car on ne sait pas quand les données seront modifiées
  - Les données ont une durée de vie (60s pour les attributs)
  - Pour les données, le client vérifie l'heure de modification du fichier
  - Mais toujours risque de *race conditions*
- Écriture délayée
  - Le serveur doit commiter immédiatement les écritures
  - Mais rien n'oblige les clients à lui envoyer immédiatement
  - Certains clients peuvent donc envoyer les écritures à retardement
  - Par exemple envoie de 10 écritures du même fichier

Performance

- Cache de retransmission
  - Un client envoie des requêtes RPC jusqu'à obtenir une réponse
  - Parfois, la réponse arrive avec du retard, le client a déjà fait une retransmission
  - Le serveur doit détecter ces retransmissions
  - Certains requêtes sont idempotentes (read…), d'autres sont non idempotentes (modifications du FS…)
  - Exemple : suppression d'un fichier
    - Client demande suppression
    - Serveur efface fichier et répond ok
    - Réponse perdu, le client redemande une suppression
    - Serveur reçoit deuxième demande, qui échoue
    - Client reçoit une notification d'échec
  - Le serveur maintient un cache des requêtes récentes

Sécurité NFS

- Le control d'accès est vérifié à 2 moments
  - Montage
  - Chaque requête NFS
- Le serveur peut refuser le montage en fonction de la machine cliente, mais pas de l'utilisateur
- Un client envoie l'UID et le GID de l'utilisateur
- Mais ils ne sont pas uniques parmi plusieurs systèmes
- NFS ne peut pas faire la distinction entre des utilisateurs différents de même UID/GID
- Attaques possibles
  - Création d'un compte local avec les uid/gid qui vont bien
  - Modification des paquets au vol
- Solution : *UID remapping*
  - Le serveur maintient une table pour changer les uids et les rendre unique pour l'ensemble de ses clients

NFS v3

- Adresse certaines limitations de la v2
- Passage en 64 bits pour les tailles et offsets des fichiers
- En V2, le serveur doit commiter toutes les modifications sur disque
  - La V3 permet les écritures asynchrones
  - Quand des données sont envoyée au serveur, le client les garde en mémoire
  - Fermeture du fichier => Envoie d'un *COMMIT* au serveur
  - Si succès le client peut effacer les données
- Trop de requêtes par ls
  - Ajout d'une commande pour obtenir tous les files handles, noms et attributs des fichiers d'un répertoire
  - Nécessite un transfert important d'informations
- Un client/serveur NFSv3 doit aussi supporter la v2
  - Négociation du protocole à l'exécution