Fichiers

Thèmes traités

- robustesse des E/S
- performances des E/S
- caches d'E/S
- mise en œuvre des fichiers Unix : structures de données et fonctionnement.
- Opérations essentiellles de l'API fichiers Unix, protocole d'usage.
- Autres opérations : `unlink`, `fstat`, `mount`, `ioctl`, `fcntl`

1 Questions

1. Comment rendre le code de gestion de fichiers indépendant des caractéristiques matérielles des différents périphériques ?
2. Comment peut-on représenter l'espace disque alloué à chaque fichier ?
3. Quel sont les avantages et les inconvénients d'une allocation contiguë pour les données d'un fichier ?

2 Rendre robustes et efficaces les accès aux fichiers

On étudie les possibilités pour améliorer la tolérance aux pannes des accès à un système de fichiers simple. L'étude présentée ici s'appuie largement sur le chapitre 42 de l'ouvrage « Operating Systems : three easy pieces » (A et R Arpaci-Dusseau), référencé sur la page Moodle de l'enseignement.

2.1 Position du problème

Question. *Proposer (dessiner) un système de fichiers minimal, avec une carte d'implantation pour les blocs libres, pour les i-nœuds (4 max), des i-nœuds limités à une carte d'implantation avec 3 pointeurs directs, et un espace de 8 blocs pour le stockage du contenu des fichiers.*

Question. *Décrire le détail des accès mis en œuvre lors d'une écriture*

2.2 Recherche de solutions

Question. *Quels problèmes peuvent survenir en cas de panne ? Peut-on les détecter (si oui, comment) ? Peut-on les corriger (si oui, comment) ?*

Sur la robustesse On peut distinguer quelques grandes familles de solutions :

2.2.1 Détection+guérison (`fsck`)

Le principe est de détecter des incohérences, à partir de contrôles de vraisemblance, puis de (tenter de) reconstituer un état sain. `fsck` est un outil typique de cette catégorie. Il s'appuie largement sur l'exploitation des métadonnées de la table des i-nœuds. Les principaux tests effectués sont :

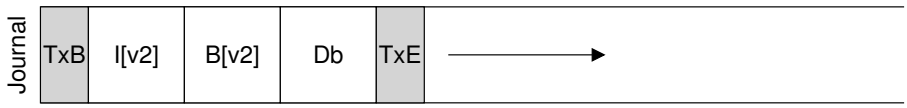
- *vérification du superbloc* : la taille du système de fichiers est-elle inférieure à la taille allouée ? égale à la taille allouée plus la taille de l'espace libre ?
- *vérification des blocs libres* : on parcourt les blocs d'index pour vérifier la cohérence de la carte d'implantation des blocs libres (on ne doit pas trouver de bloc alloué marqué libre ; les blocs non alloués doivent être libres).
- tests de *cohérence du contenu des inœuds*. Les inœuds incohérents sont effacés.
- réévaluation (et correction éventuelle) des compteurs de références. Les fichiers perdus (compteur nul) se retrouvent dans le répertoire `lost+found`.
- recherche de *doublons dans les adresses de blocs*. Le cas échéant, le bloc est dupliqué et l'un des index est corrigé.
- *cohérence des adresses de blocs*. Le cas échéant, l'index est annulé, mais il est difficile de faire plus.
- vérification de la *cohérence des répertoires* (., .., existence des i-nœuds du répertoire...)

Le principal inconvénient de ce type d'outil est, outre l'impossibilité de réparer dans de nombreux cas, sa lenteur, d'autant plus sensible que la taille des espaces de stockage augmente.

2.2.2 Journalisation

Le principe est de réaliser les écritures (données + métadonnées) sur une **copie en mémoire stable** et non directement dans le fichier. L'ensemble des écritures (données + métadonnées) liées à une même opération d'écriture de données est appelé une *transaction*. L'espace dans lequel les transactions sont enregistrées avant recopie dans le fichier est appelé un *journal*.

Chaque transaction du journal est délimitée par un bloc marquant le début de la transaction et par un bloc marquant la fin de la transaction. Les blocs d'une même transaction sont successifs dans le journal.



Une fois les blocs de la transaction enregistrés, les données de la transaction sont recopiées dans le fichier.

Question. *Est-il nécessaire que le journal soit conservé en mémoire stable ? Pourquoi ?*

Les blocs de la transaction pourraient être écrits simplement, l'un après l'autre. Cette stratégie serait cependant très inefficace, car elle demanderait de réaliser 5 écritures distinctes. En outre, la transaction comportant plusieurs blocs, il est très plausible que le périphérique va réordonnancer les requêtes d'écriture, sans qu'il soit possible de contrôler ce réordonnancement. Il n'est donc pas possible de supposer que les écritures de la série de blocs de la transaction vont se faire séquentiellement, ou dans un ordre donné.

Question.
- *Dans ces conditions, est-il nécessaire d'attendre que le bloc de fin de la transaction soit écrit avant de procéder à la recopie ?*
- *Est-ce suffisant ?*
- *Quelle solution simple pourrait-on envisager pour surmonter cette difficulté ?*

Question. *Comment maintenir malgré tout des performances acceptables ?*

2.2.3 Pages d'ombre

Cette technique, également très employée, s'appuie sur l'usage de *blocs d'index*, et sur le *principe de copie sur écriture*. Les données et les métadonnées (bloc d'index référencé par le i-nœud) sont écrites dans des copies, puis la référence du i-nœud vers le bloc d'index est actualisée de manière atomique. On retrouve le principe consistant à **écrire les données référencées avant la référence**, afin de garantir la cohérence.

Sur l'amélioration des performances
- cylindre (`FFS`) : placer les blocs disques correspondant à un même fichier dans un même cylindre, métadonnées comprises, de manière à limiter les déplacements de la tête de lecture.
- lectures : chargement anticipé des blocs suivant le dernier bloc lu.
- écritures
  - *caches d'E/S* : les écritures sont différées ; elles sont dans un premier temps effectuées dans un cache de blocs/pages, puis réalisées périodiquement de manière groupées (`sync`). Un autre avantage est que ce cache de blocs sales est directement disponible pour d'éventuelles lectures, ou peut éviter des écritures successives sur un même bloc.
  - *LFS (log-structured file system)* : afin d'accélérer les écritures (données et métadonnées), celles ci sont mises en cache, groupées, et écrites séquentiellement. Le résultat est que les inœuds se trouvent dispersés sur le support, ce qui alourdit la gestion, mais peut permettre de réduire les temps d'accès.

Question. *On ne peut lire moins d'un secteur de quelques Koctets sur un volume disque. Cela est-il un inconvénient lorsqu'un programme lit séquentiellement dans un fichier par paquets de quelques octets ?*

3 Déroulement

- rappel de cours : *structure d'un SGF et inœuds*
  planches 4-6 du support d'accompagnement des TD (vu en cours)
- panorama des opérations au niveau du SGF. Le but est de connaître leur rôle et leur existence : minimum de syntaxe (quand même : identifiants des fonctions :))
  - planches 5-7 : présentation rapide des opérations sur les i-nœuds, `fstat` (non vu en cours)
  - planche 10 : opérations sur les répertoires
- fonctionnement du sous-système d'E/S d'Unix
  planche 11
  - synthèse des structures de données en jeu : de la table des descripteurs aux données sur disque
  - amélioration des performances : cache (blocs) d'E/S, image des i-nœuds
  - déroulement de l'accès à un fichier
  - héritage des descripteurs de fichiers
- Exercice : mise en œuvre de la robustesse
- API fichiers/répertoires : présentation rapide, protocole d'usage (`open;(read/write/lseek)*;close`), tout est fichier
- attention : tampons langage et tampons système (cache de blocs) (planche 16)

4 Testez vous

Vous devriez maintenant être en mesure de répondre clairement aux questions suivantes :
- Comment arrive-t-on à réduire les temps d'accès au disque ?
- Que se passe t-il lorsqu'un processus ouvre un fichier ?
- Où sont et quelles sont les (méta)données utilisées pour la gestion des fichiers ?
- Quel est le circuit suivi par une donnée au cours de sa lecture, ou de son écriture ?