TP11 : Système de fichiers distribué ¶

1. Objectifs

Le **système de fichier** est une partie importante du système d'exploitation, puisqu'il contient toutes les informations *non volatiles*, c'est à dire les informations qui doivent être conservées lorsqu'on éteint la machine entre deus sessions de travail, et sont donc enregistrées sur un périphériques de stockage externe (disque magnétique, clé USB, etc).

En règle générale, le **système de fichiers** est (beaucoup) plus gros que la capacité totale de la mémoire vive. Au cours d'une session de travail particulière, seule une (toute petite) partie des informations stockées dans le système de fichiers ont effectivement besoin d'être amenées en mémoire. Tous les systèmes d'exploitation définissent donc un **cache des fichiers**, qui est une représentation en mémoire, d'une partie du **sytème de fichiers** stocké sur disque.

Ce **cache des fichier** est un bon exemple de structure de donnée du système d'exploitation susceptible de créer un goulot d'étranglement. Cette structure est partagée, puisqu'elle peut être accédée par n'importe quel thread de n'importe quel process souhaitant accéder à un fichier, et elle est potentiellement trop volumineuse pour être stockée dans un seul cluster. Elle doit donc être distribuée sur plusieurs clusters.

Dans ce TP, on présente les techniques de distribution utilisées dans Almos-mkh pour minimiser la contention lors des accès concurrents au système de fichier.

2. Principes Généraux

Le périphérique de stockage externe contenant le système de fichier est toujours organisé comme un tableau de *secteurs* physiques. Chaque *secteur* peut contenir un bloc de 512 octets, et est identifié par un *numéro* de secteur, appelé LBA (Logic Block Address). Bien que le plus petit élément de stockage adressable soit le *secteur*, presque tous les systèmes de fichiers définissent une unité d'allocation plus grosse, appelée *cluster*. Généralement, le cluster a une capacité égale à une page, c'est à dire 4 Koctets (8 secteurs consécutifs). Le système de fichier utilise donc le périphériques comme un tableau de clusters. Un fichier occupe toujours un nombre entier de clusters, et un même cluster ne peut pas être partagé par deux fichiers.

Dans les systèmes UNIX ou Microsoft, le système de fichier possède une structure hiérarchique arborescente, puisque certains fichiers sont des *répertoires*, qui contiennent eux-mêmes des *liens* vers d'autres fichiers ou répertoires. N'importe quel fichier (ou répertoire) dans le système de fichiers peut être désigné sans ambiguité par un *cheminom*, décrivant le chemin allant du répertoire *racine* du système de fichier à un fichier particulier.

Tout système de fichier doit donc stocker trois types d'information

1. certains clusters du disque sont utilisés pour stocker les **fichiers terminaux**, autres que les répertoires.
2. d'autres clusters sont utilisés pour stocker les **fichiers répertoires**, qui définissent la structure arborescente du système de fichier.
3. d'autres encore sont utilisés pour stocker la **FAT** (File Allocation Table) qui définit quels clusters ont été alloués à chaque fichier.

Il existe évidemment de nombreux types de système de fichier, qui sont différentes implémentations de la structure arborescente générique définie ci-dessus. C'est le format utilisé pour représenter les répertoires et la FAT qui différencient les différents types de systèmes de fichier. Pour pouvoir s'adapter facilement à plusieurs types de systèmes de fichiers, les systèmes UNIX définissent généralement une API indépendante du type de système de fichier appelée VFS (Virtual File Système).

A ce jour, l'API VFS de Almos-mkh a été portée sur trois systèmes de fichiers : FATFS, DEVFS, RAMFS. Dans ce TP, on utilisera un disque contenant un système de fichier FATFS, respectant le format Microsoft FAT32.

3. Implémentation du Cache des Fichiers

Le *cache des fichiers* est le nom générique de la représentation interne - en mémoire - du *système de fichiers*. Almos-mkh utilise deux techniques pour réduire l'encombrement du cache de fichiers en mémoire par rapport à encombrement du systèmes de fichiers sur le disque:

- pour les **fichiers répertoires**, seulement un sous-ensemble des liens vers les fichiers fils contenus dans le répertoire est présent dans le cache: seuls les liens permettant d'atteindre un fichier présent en mémoire sont représentés. Un répertoire peut donc contenir 10 liens vers 10 fichiers fils sur le disque, alors que sa représentation en mémoire contient un seul fils, si les 9 autres liens pointent vers des fichiers qui n'ont pas eu besoin d'être chargés dans le cache.
- pour les **fichiers terminaux**, seuls les pages contenant des données accédées en lecture ou en écriture sont effectivement chargées dans le *cache des fichier* en mémoire. Un fichier occupant 1 Moctets sur le disque (256 pages) peut n'occuper que 4 Koctets en memoire (1 page), si une seule page est accédée.

3.1 Arbre des Inodes

La structure générale du système de fichier n'est pas un arbre, mais un DAG (Graphe orienté sans cycles), puisqu'il peut exister plusieurs cheminoms permettant d'atteindre un même fichier par plusieurs chemins partant de la racine. Malgré cela, on utilise le nom *arbre des inodes* pour désigner la structure représentant le sous ensemble des fichiers et répertoires présents à un instant donné dans le cache des fichiers. Cette structure est implémentée par un graphe bi-parti contenant deux types de noeuds:

- un noeud de type **inode** représente un fichier, terminal ou répertoire. Il est implémentée par la structure *vfs_inode_t*.
- un noeud de type **dentry** représente un lien entre un fichier répertoire et un autre fichier fils contenu dans le répertoire. Il est implémentée par la structure *vfs_dentry_t*.

L'arbre des inodes est une structure distribuée: Les structures *vfs_inode_t* représentant les fichiers sont distribuées le plus uniformément possible sur les différents clusters de l'architecture. Dans le cas cas d'un inode *père* représentant un répertoire, toutes les structures *vfs_dentry_t* décrivant les liens vers les inodes *fils* sont toujours allouées dans le cluster contenant l'inode *père*. Mais l'inode *père* et l'inode *fils* sont généralement allouées dans deux clusters différents. Deux inodes *fils* d'un même inode *père* sont également rangés dans des clusters différents. Par conséquent, l'arbre des inodes utilise principalement des pointeurs étendus.

3.2 Mapper d'un fichier

A chaque noeud de type **inode** de l'arbre des inodes est attaché une structure annexe, appelée **mapper** contenant une copie partielle, en mémoire, du contenu du fichier représenté par cet inode. Cette structure mapper_t est implémentée comme un *radix tree* de profondeur 3. Ce *radix tree* est indexé par le numéro de la page dans le fichier, vu comme un tableau de pages de 4 octets, et chaque entrée du *radix tree* contient un pointeur sur une page de 4Koctets en mémoire.

- pour les **fichiers terminaux**, seules les pages qui ont été accédées (en lectures ou en écriture) sont effectivement copiées dans le **mapper**.
- pour les **fichiers répertoires**, tout le contenu du fichier est copié dans le **mapper**, mais la majorité des répertoires ne nécessitent pas plus d'une page.

La structure mapper_t est entièrement localisée dans le cluster contenant la structure vfs_inode_t à laquelle il est attaché.

3.3 File descriptor

Chaque fois qu'un processus ouvre un fichier avec un appel système *open()*, le noyau crée un descripteur de fichier ouvert, et renvoie au processus client un index identifiant ce descripteur de fichier. Ce descripteur de fichier doit être utilisé par le processus pour accéder au fichier, car de descripteur contient en particulier un offset (pointeur) définissant quelle zone du fichier doit être accédée.

Ce descripteur de fichier ouvert est implémenté par le structure *vifs_file_t*.

Il peut exister à un instant donné plusieurs descripteurs de fichiers ouverts pour un même fichier X, mais toutes les structures vfs_file_t concernant le fichier X sont allouées dans le cluster contenant l'inode représentant le fichier X.

3.4 Conclusion

Pour minimiser la contention, les structures *vfs_inode_t* sont distribuées le plus uniformément possible sur tous les clusters. Et les structures annexes attachées à un fichier X particulier: la structure *mapper_t* contenant les données du fichier, les structures *vfs_dentry_t* attachées à un fichier répertoire, et les structure *vfs_file_t* permettant l'accès au fichier, sont toutes allouées dans le cluster contenant la structure *vfs_inode_t* représentant le fichier X. Il s'agit donc d'une politique de distribution possédant la granularité *fichier*, qui minimise la contention dans le cas ou un grand nombre de threads accèdent parallèle à un grand nombre de fichiers différents, mais qui ne sera pas efficace dans le cas où un grand nombre de threads accèdent en parallèle à un même fichier.

4. Questions

Pour répondre aux questions ci-dessous, il faut plonger dans le code du VFS dans les fichiers almos-mkh/kernel/fs/vfs.c et almos-mkh/kernel/fs/vfs.h.

Vous pouvez également consulter le code du *mapper* dans les fichiers almos-mkh/kernel/mm/mapper.c et almos-mkh/kernel/mm/mapper.h.

Le code du *radix_tree* générique est défini dans les fichiers almos-mkh/kernel/libk/grdxt.c et almos-mkh/kernel/libk.grdxt.h.

1. L'arbre de inodes contient des pointeurs *descendant* permettant d'atteindre les inodes des fichiers *fils* à partir de l'inode du fichier répertoire *père*. Pourquoi l'appel système open() nécessite-t-il un parcours descendant, de la racine vers un fichier terminal? Quelle fonction du VFS implémente-t-elle ce parcours ? Quels sont ses arguments d'entrée? Que retourne-t-elle?

   > L'appel système open() crée un descripteur de fichier dans le cluster contenant l'inode représentant un fichier identifié par son cheminom. L'OS doit donc analyser le cheminom et effectuer un parcours descendant dans l'arbre des inodes, en partant de la racine, et en recherchant dans chaque répertoire intermédiaire le *bon* fils pour atteindre le répertoire suivant, jusqu'à atteindre le fichier cible. C'est la fonction vfs_lookup() qui effectue ce parcours. Ses entrées sont principalement un pointeur étendu sur l'inode racine, et un cheminom identifiant le fichier. Elle retourne principalement un pointeur étendu sur l'inode représentant le fichier recherché.

2. L'arbre de inodes contient également des pointeurs *ascendant* permettant d'atteindre l'inode d'un fichier *père* à partir de l'inode d'un fichier fils. Pourquoi l'appel système getcwd() nécessite-t-il un parcours ascendant, d'un fichier terminal vers la racine, sachant que le descripteur de processus contient un pointeur étendu sur l'inode représentant le répertoire courant? Quelle fonction du VFS implémente-t-elle ce parcours? Quels sont ses arguments d'entrée? Que retourne-t-elle?

   > L'appel système getcwd() retourne un cheminom définissant le répertoire de travail courant du processus appelant. L'OS doit donc partir de l'iode représentant le répertoire courant, et remonter dans l'arbre vers la racine, pour construire progressivement le cheminom. C'est la fonction vifs_get_path() qui implémentée ce parcours.

3. Comment est représenté, dans la structure *vfs_inode_t* représentant un fichier répertoire *père*, l'ensemble des liens vers les structures de données *vfs_dentry_t* représentant les liens vers les fichiers *fils* contenus dans ce répertoire ?

   > Les différentes entrées d'un même répertoire sont identifiées par leur nom. Même si le nombre d'entrées dans un répertoire est le plus souvent très petit (une dizaine), il peut parfois être très grand (quelques centaines ou quelques milliers). Dans ce cas, la recherche associative par nom peut devenir pénalisante. Pour accélérer la recherche, Almos-mkh utilise donc une table de hash plutôt qu'une liste chaînée pour représenter l'ensemble des structures *vfs_dentry_t* représentant les entrées d'un même répertoire.

4. Un fichier terminal pouvant être accédé par plusieurs chemins, un fichier terminal *fils* peut avoir plusieurs répertoires *pères*. Comment est représenté, dans la structure *vfs_inode_t* l'ensemble des liens *pères* ?

   > Seuls les fichiers terminaux peuvent être accédés par plusieurs cheminoms, et peuvent donc avoir plusieurs pères. Il n'y a donc en pratique qu'un tout petit nombre de fichiers qui possèdent plus d'un père. Par conséquent, l'ensemble des structures *vfs_dentry_t* pères d'un même fichier X est représenté par une simple liste chaînée, enracinée dans la structure *vfs_inode_t* représentant le fichier X, qui ne contient généralement qu'un seul élément.

5. En cas de miss sur le cache des fichiers, Il faut parfois créer, dans l'arbre des inodes, un nouveau couple (dentry/inode) pour introduire un nouveau fichier *fils* dans un répertoire *père* existant. Le VFS utilise utilise pour cela la fonction *vfs_add_child_in_parent()*. Quelles actions sont réalisées par cette fonction ? Dans quel cluster est créée la nouvelle structure *vfs_dentry_t* représentant la nouvelle entrée dans le répertoire *père* ? En analysant le code de la fonction appelante, par exemple *vfs_lookup()*, dites comment est choisi le cluster dans lequel est allouée la mémoire pour la nouvelle structure *vifs_inode_t* représentant le fichier *fils* ?

   > Cette fonction crée et initialise une structure *vfs_dentry_t* représentant une nouvelle entrée dans le répertoire père, dans le cluster contenant le répertoire père. Puis elle crée et initialise une structure *vifs_inode_t* représentant le nouveau fichier dans un autre cluster, défini par la fonction appelante. Enfin cette fonction établit les liens (pointeurs) entre ces différentes structures : inonde père, gentry, et inode fils. En analysant le code de la fonction appelante *vfs_lookup()*, on peut voir que le choix du cluster contenant le nouvel inode fils, est réalisé par la fonction *cluster_random_select()*, qui implémentée un choix pseudo-aléatoire favorisant une distribution uniforme.

6. L'arbre des inodes est une structure partagée, qui peut être accédée de façon concurrente par un grand nombre de threads. Toute modification de cette structure nécessité évidemment un accès exclusif interdisant tout parcours de l'arbre par un autre thread, pendant la modification de la structure. Quel mécanisme de protection est utilisé pour garantir un accès exclusif en cas de modification, tout en permettant des accès parallèles en lecture ? Quelle est la faiblesse principale de ce mécanisme ?

   > La structure vfs_inode_t contient un verrou de type *read/write lock*. Ce type de verrou garantit un accès exclusif pour une écriture, mais permet plusieurs lectures simultanées. Il est bien adapté, car les modifications de la structure de l'arbre des inondes (introduction d'un nouveau fichier dans le cache des fichiers) sont des événements beaucoup plus rares que les parcours de cet arbre. Avant tout accès à l'arbre des inondes, un thread doit donc prendre le lock (unique) situé dans l'iode racine du VFS.
   >
   > L'ensemble de la structure distribuée *arbre des inondes* est donc protégée par un verrou global, qui peut constituer un point de contention dans le cas d'applications parallèles multi-threads effectuant un grand nombre de creations / destructions de fichiers. Il reste donc du travail pour remplacer ce *giant lock* par des verrous distribués...

7. La structure *mapper_t* attachée à chaque inode (et donc à chaque fichier) peut être considérée comme un cache extensible dont la capacité augmente dynamiquement en fonction des besoins. Quelle fonction permet elle à un thread s'exécutant dans n'importe quel cluster C d'accéder n'importe quelle page de n'importe quel fichier située dans n'importe quel autre cluster C' ? Quelle fonction du mapper permet-elle de traiter un miss (c'est à dire un accès à une page du fichier qui n'est pas encore chargée dans le mapper) ? Décrivez précisément ce que font ces deux fonctions.

   La fonction *mapper_get_page()* prend pour arguments un pointeur étendu sur un mapper distant, et le numéro de la page dans le fichier. Elle retourne un pointeur étendu sur le descripteur de la page physique. En cas de miss, cette fonction appelle la fonction *mapper_handle_miss()* pour charger la page manquante dans le mapper. Cette seconde fonction alloue une page physique dans le cluster distant C', initialise le descripteur de page, insère la page dans le radix_tree, et initialise la page lorsque c'est nécessaire.

8. En cas de miss sur un mapper, dans quel cas la page manquante doit-t-elle être initialisée dans le mapper à partir des informations stockées dans le système de fichiers (sur le disque) ? Dans quel cas l'accès au disque est-il inutile ? Que peut-on en conclure que la politique d'écriture des caches mapper (*write-through / write-back*) ?

   Il existe un mapper par fichier (répertoire ou terminal), plus un mapper pour la FAT elle-même. En cas de miss, la page manquante doit être initialisée à partir du disque dans 3 cas :

   - il s'agit du mapper de la FAT,
   - il s'agit du mapper d'un fichier répertoire,
   - il s'agit du mapper d'un fichier terminal, et la page manquante contient des informations présentes sur le disque.

   Le seul cas ou un miss ne nécessite pas un accès au disque, est le scénario d'une extension de la taille du fichier, suite à une écriture: dans ce cas, la page manquante dans le mapper n'existe pas sur le disque. Les données seront donc écrites dans le mapper, et seront écrites plus tard sur le disque. Par conséquent, la politique est *write_back* pour les fichiers terminaux. Elle est *write-through* pour les répertoires et pour la FAT.

9. Un même fichier, et donc un même mapper, peut être accédé de façon concurrente par plusieurs threads. Quel mécanisme est utilisé pour permettre ces accès concurrents tout en maximisant le parallélisme ?

   La structure mapper_t contient un verrou de type *read/write lock* car l'introduction d'une nouvelle page dans le radix tree nécessite un accès exclusif. Il y a donc un verrou par mapper, qui permet des accès parallèles au mapper tant que ce sont des lectures. Seules les extensions (rares) d'un mapper créent des points de contention.

Dernière modification le 13 janv. 2022 à 09:24:23)