Linux Gazette n55

Juan I. Santos Florido

Juan I. Santos Florido

Jérôme Fenal

jerome@fenal.org

1. Systèmes de fichiers journalisés

1.1. Introduction

Alors que Linux gagne en maturité, il doit satisfaire les besoins de différents types d'utilisateurs et s'adapter à toute situation. Ces dernières années, Linux a acquis un certain nombre de fonctionnalités et est utilisé dans des situations très différentes les unes des autres. Nous retrouvons maintenant Linux dans des systèmes embarqués, il existe des projets de routeurs basés sur Linux, des distributions tenant sur une seule disquette, les accélérations matérielles pour la 3D sont partiellement supportées, de même que l'affichage multi-écran avec XFree, il existe aussi des jeux sous Linux ainsi qu'un bon nombre de nouveaux gestionnaires de fenêtres. Ces fonctions sont importantes pour l'utilisateur final. Il y eu aussi un grand bond en avant pour les besoins de Linux en tant que serveur - essentiellement grâce au passage à la version 2.2 du noyau Linux.

En outre, grâce au support de différents acteurs de l'industrie, et surtout ceux de la communauté Open Source, Linux acquiert les capacités et fonctionnalités les plus importantes des Unix commerciaux et serveur de taille importante. Une de ces fonctionnalités est le support de nouveaux systèmes de fichiers capables d'utiliser des volumétries disque importantes, supportant une montée en charge de plusieurs milliers de fichiers, utilisables rapidement même après un arrêt brutal, ayant des performances supérieures, se comportant correctement tant avec les fichiers de petite taille que ceux de grande taille, tentant d'éviter fragmentation interne et externe et implantant de nouvelles fonctionnalités encore non supportés par leurs aînés.

Cet article est le premier d'une série de deux, où seront présentés au lecteur les différents systèmes de fichiers journalisé : JFS, XFS, Ext3, et ReiserFS. Les différentes fonctionnalités et concepts autour des systèmes de fichiers journalisés seront aussi discutées. Le second article passera lui en revue leur comportement ainsi que leurs performances respectives à travers différents tests de fonctionnalités et de performance.

2. Glossaire

2.1. Fragmentation interne

Le bloc logique est la plus petite unité d'allocation visible d'un système de fichiers à travers les appels systèmes. Cela signifie que le stockage d'un fichier dont la taille en octets est inférieure à la taille d'un bloc logique occupera sur disque la taille du bloc logique.

De même, si la taille d'un fichier n'est pas divisible de façon entière par la taille du bloc logique (c-à-d taille_fichier modulo taille_bloc != 0), le système de fichiers allouera un nouveau bloc qui ne sera pas rempli entièrement, gaspillant ainsi de l'espace disque. Ce gaspillage d'espace disque est ce que l'on appelle la fragmentation interne. Remarquons que plus la taille du bloc logique est importante, plus importante sera la fragmentation interne.

2.2. Fragmentation externe

La fragmentation externe d'un fichier est le fait d'avoir les blocs constituant le fichier répartis de façon non contiguë sur tout le disque, réduisant ainsi les performances d'accès à ce fichier, du fait de l'augmentation des mouvements des têtes de lecture du disque nécessaire à l'accès à ce fichier.

2.3. Extent[1]

Les extents sont des ensembles de blocs logiques contigus, utilisés dans divers systèmes de fichiers voire même dans certains SGBD. Le descripteur d'un extent ressemble à début, taille, offset, où début est l'adresse du premier bloc de l'extent, taille est la taille en nombre de blocs, offset est l'offset dans le fichier où se situe l'extent.

Extent

L'utilisation des extents améliore l'utilisation de l'espace, puisque tous les blocs d'un extent sont contigus. Cette amélioration amènera de meilleures performances en lecture séquentielle, car moins de mouvements de tête de lecture seront nécessaires. Remarquons ainsi que l'utilisation des extents réduit le problème de la fragmentation externe, puisque les blocs sont rassemblés. Mais l'utilisation d'extent n'est pas toujours la panacée. Par exemple, si nos applications demandent des extents d'une taille proche de celle des blocs logiques, tous les bénéfices seront perdus, puisque ces extents ressembleront à autant de blocs logiques.

Pour clore le chapitre amélioration des performances, l'utilisation d'extents améliore aussi les transferts simultanés de plusieurs blocs et améliore les taux de réussite des mémoires tampons des disques durs.

2.4. Arbres B+

Arbre B+[2] : les clés des noeuds sont ordonnées dans l'arbre, ce qui améliore les temps de recherche, recherche qui n'est plus séquentielle. Les feuilles sont chaînées entre elles.

1. Pour localiser le fichier resolv.conf, nous commençons à la racine de l'arbre, lisons séquentiellement, et trouvons qu'il n'y a pas de clé supérieure à celle de resolv.conf, donc nous suivons le dernier pointeur (en rouge).
2. Nous sommes redirigés sur un autre noeud de l'arbre. On fait de même qu'avant, on lit les clés jusqu'à "securetty", qui est supérieure à "resolv.conf". On suit donc le lien associé (en bleu).
3. Nous arrivons ainsi à une feuille de l'arbre. Il est temps de lire séquentiellement par ordre ascendant les clés présentes. Nous trouvons enfin la clé désirée, nous permettant d'utiliser le pointeur associé vers les données du fichier "resolv.conf"

La structure de données de type arbre B+ est depuis longtemps utilisée pour l'indexation de bases de données.

Cette structure apporte aux SGBD une manière rapide et stable dans les performances pour accéder à leurs enregistrements. Le terme arbre B+ vient de «arbre balancé», équilibré en français. Le signe «+» signifie que la structure arbre B+ est une version modifiée de l'arbre B. [3] original, modification qui consiste en le maintien de pointeurs entre les différentes feuilles de l'arbre, ce afin de ne pas pénaliser les accès séquentiels de feuille en feuille. Puisque les arbres B et B+ sont issus des bases de données, nous utiliserons des analogies avec celles-ci pour les expliquer.

Les arbres B+ sont constitués de deux types d'éléments : les noeuds et les feuilles. Les deux sont une paire (clé, pointeur), ordonnées de façon ascendante par les valeurs de clés, avec un pointeur final qui n'a pas de clé correspondante. Alors que les pointeurs des noeuds pointent sur d'autres noeuds ou feuilles, les pointeurs des feuilles pointent

directement sur l'information utilisable. Chaque paire (clé, pointeur) organise l'information au sein de l'arbre B. En base de données, chaque enregistrement possède un champ-clé, un champ qui sert à distinguer un enregistrement particulier des tous les autres enregistrements de même type. Les B-arbres utilisent ces champs-clés pour indexer les enregistrements d'une base de données pour améliorer les temps d'accès à l'information.

Ainsi que nous l'avons vu, un noeud (clé, pointeur) est utilisé pour pointer sur un autre noeud ou sur une feuille. Dans les deux cas, la clé associée au pointeur aura une valeur supérieure à toutes les valeurs de clés stockées dans les noeuds cibles. De plus, les enregistrements ayant des valeurs de clés identiques devront être adressés par la paire suivante dans le noeud. Ce point est la principale raison de l'existence du pointeur de fin qui n'a aucune valeur de clé correspondante. Notons qu'à partir du moment où une clé est utilisée dans une paire, il doit exister un autre pointeur pour atteindre les enregistrements ayant cette même valeur de clé. Ce pointeur de fin est utilisé dans les feuilles pour pointer sur la feuille suivante. On peut ainsi passer les contenus en revue aussi de façon séquentielle.

Les arbres B+ doivent être équilibrés. Cela signifie que la longueur du chemin nous menant de la racine de l'arbre à chacune des feuilles doit toujours être la même. En plus, les feuilles doivent contenir un nombre minimal de paires pour exister. Quand le contenu d'une feuille tombe sous le seuil, les paires seront intégrées à une autre feuille.

Pour rechercher un enregistrement particulier, nous procéderons ainsi :

Supposons que nous recherchons un enregistrement ayant une clé K. Nous partons du noeud racine, et passons en revue séquentiellement les clés qu'il contient, jusqu'à ce que nous trouvions une clé de valeur supérieure à celle de K. Nous passons ensuite au noeud (ou feuille, nous ne savons pas encore) pointé par le pointeur associé à la clé. A ce stade, nous répétons la même opération si nous ne sommes pas arrivés à une feuille. Arrivé sur une feuille, nous criblons séquentiellement les paires contenues jusqu'à la valeur K. Moins de blocs sont nécessaires d'être lus pour arriver au bloc désiré, ce qui qualifie cette technique comme étant de moindre complexité qu'une recherche purement séquentielle, où dans le plus mauvais cas, nous devons lire tous les enregistrements pour trouver celui que nous recherchons.

2.5. Unix File System (UFS)

C'est le nom du système de fichiers utilisé au début par SCO, System V[4], et d'autres Unix. Le noyau Linux inclut le support optionnel d'UFS. La plupart des Unix continuent à utiliser UFS, avec néanmoins quelques améliorations mineures.

2.6. Virtual File System (VFS)

VFS est la couche logicielle du noyau[5] qui fournit une interface de programmation applicative [API en anglais] unifiée pour accéder aux services des systèmes de fichiers, quel que soit le système de fichier particulier utilisé derrière. Tous les pilotes de systèmes de fichiers (VFAT, Ext2FS, JFS) doivent ainsi fournir un certain nombre de routines à la couche VFS pour être utilisables sous Linux. VFS est la couche qui permet aux applications de comprendre et d'utiliser tant de systèmes de fichiers différents, même les systèmes de fichiers commerciaux.

3. Le journal de transactions

3.1. Qu'est-ce qu'un système de fichier journalisé ?

Je pars du principe que chacun sait ce qu'est un tampon d'écriture : un tampon alloué en mémoire qui permet d'accélérer les entrées / sorties. Ce type de tampon est couramment utilisé par les systèmes de fichiers (le cache/tampon disque), et les SGBD pour augmenter les performances. Un problème apparaît si le système plante avant que ces tampons ne soient écrits sur disque, ce qui va provoquer un comportement incohérent du système après son redémarrage. Imaginons qu'un fichier soit effacé en mémoire, mais pas sur disque. C'est à cause de ce problème que les SGBD et les systèmes de fichiers ont les moyens de revenir à une situation cohérente. Bien que les SGBD aient cette capacité depuis des années, les systèmes de fichiers dérivés de UFS ont tendance à voir le temps de correction des erreurs augmenter avec l'augmentation

de la taille du système de fichiers. L'outil **fsck** pour Ext2FS doit parcourir toute la partition de disque pour pouvoir revenir à une situation cohérente. Cette opération consommatrice de temps inflige souvent une perte de temps d'exploitation des gros serveurs ayant en ligne plusieurs centaines de giga-octets, voire plusieurs tera-octets. Ceci est la principale raison de voir les systèmes de fichiers hériter des technologies de reprise sur incident des SGBD, et donc l'apparition des systèmes de fichiers journalisés.

3.2. Comment ça marche ?

Les SGBD « sérieux » utilisent la notion de transaction. Une transaction est un ensemble d'opérations qui satisfont à plusieurs propriétés. Ces propriétés sont surnommées ACID pour Atomicité, Cohérence, Isolation, et Durabilité. La propriété la plus importante pour notre explication est l'atomicité. Elle implique que toutes les opérations faisant partie d'une transaction sont soit intègres et terminées, soit en cas d'erreur ou d'annulation, ne produisent aucun changement. Cette fonctionnalité, associée à l'isolation, fait que les transactions peuvent être assimilées à des opérations atomiques qui ne peuvent être exécutées partiellement. Ces propriétés des transactions sont utilisées en bases de données du fait des problèmes rencontrés pour maintenir la cohérence des données tout en exploitant la simultanéité. Pour ce faire, les SGBD journalisent toutes les opérations simples (en écriture disque synchrone) d'une transaction dans un fichier dit journal. En fait, les opérations ne sont pas les seules journalisées : les arguments de ces opérations le sont aussi avant l'exécution de la transaction. A l'issue de chaque transaction, il doit y avoir une opération « commit », qui lance l'écriture des données sur disque. Ainsi, si le système plante, nous pouvons remonter dans l'historique du journal jusqu'à la première opération « commit », et ainsi écrire tous les contenus des précédentes transactions sur disque.

Les systèmes de fichiers journalisés utilisent cette même technique pour journaliser les opérations, autorisant ainsi une reprise d'activité dans un laps de temps très court.

Une différence majeure entre SGBD et système de fichier journalisé est que les SGBD journalisent et les données utilisateurs, et les données de contrôle, alors que les systèmes de fichiers ne journalisent que les meta-données. Les meta-données sont les structures de contrôle d'un système de fichier : inodes, tables d'allocation de l'espace

libre, tables d'inodes, etc...

4. Problèmes connus, comment répondre au besoin d'évolutivité

UFS et Ext2FS furent conçus alors que les capacités des disques durs et autres media de stockage étaient de taille moins importante que maintenant. La croissance rapide des media de stockage induirent des tailles de fichiers, de répertoires, et de partitions plus importantes, causant ainsi de plus en plus de problèmes aux systèmes de fichiers. Ces problèmes découlent des structures internes sous-tendant les systèmes de fichiers. Enfin, ces structures étaient adéquates pour les tailles moyennes observées des fichiers et répertoires, mais se sont avérées inefficaces pour les besoins actuels.

Il existe deux problèmes majeurs avec les anciennes structures :

- Elles sont incapables de gérer les nouvelles capacités de stockage : ainsi que nous l'avons dit, les anciens systèmes de fichiers étaient conçus pour certains paramètres de taille de fichiers, répertoires et partitions. Les structures des systèmes de fichiers ont un nombre fixé de bits pour stocker la taille d'un fichier, pour stocker l'adresse d'un bloc, etc... En conséquence de quoi, les tailles de fichier, de répertoire, et de partition et le nombre de fichiers dans un répertoire sont eux aussi limités. Il manque ainsi aux anciennes structures la capacité de gérer les tailles de certains objets.
- Elles sont peu adaptées à la gestion des nouvelles capacités de stockage : bien qu'elles puissent parfois gérer certaines tailles d'objets, elles sont parfois incapables de les gérer pour des raisons de performance. La principale raison étant que certaines structures se comportent correctement avec les anciennes volumétries, mais induisent des pertes parfois sévères de performances avec les nouvelles.

Les systèmes de fichiers de nouvelle génération ont été conçus pour pallier ces problèmes, gardant à l'esprit le problème de l'évolutivité. Un certain nombre de nouvelles structures de données et d'algorithmes ont ainsi été intégrés à ces nouveaux

systèmes de fichiers. Nous allons maintenant expliquer plus avant les problèmes décrits plus haut et les techniques employées pour les surmonter.

4.1. Résoudre l’impossible

La plupart des nouveaux systèmes de fichiers ont vu la taille de certains de leurs champs internes s’étendre, afin de passer outre les limitations des systèmes actuels. Les nouvelles limites pour ces systèmes sont :

Tableau 1. Limites des systèmes de fichiers

	Taille maximale du système de fichiers	Taille de blocs	Taille maximale de fichier
XFS	18 000 peta-octets (Po)	512 octets à 64 Ko	9 000 Po
JFS Blocs de 4 Ko : 32 Po	Blocs de 512 octets : 4 Po Blocs de 4 Ko : 4 Po	512, 1024, 2048 et 4096 octets	Blocs de 512 octets : 512 To
ReiserFS 2^10 Po en v3.6	4 milliards de blocs, soit 16 tera-octets	64 Ko maximum, fixé actuellement à 4 Ko	4 Go en v3.5
Ext3FS	4 To	De 1 à 4 Ko	2 Go

Actuellement, la taille maximale d’un périphérique en mode bloc limite la taille d’un système de fichiers à 2 To, la couche VFS induisant une autre limite : la taille d’un fichier inférieure à 2 Go. La bonne nouvelle est que nous avons maintenant des systèmes de fichiers capable d’évoluer en capacité, et qu’à la sortie du noyau 2.4, je suis sûr que ces limites seront étendues. Notons que XFS et JFS sont des portages de systèmes de fichiers commerciaux ; ils furent développés pour d’autres systèmes d’exploitation où ces limites n’existaient pas.

4.2. Eviter les usages inadéquats

4.2.1. La structure de gestion des blocs libres

La plupart des systèmes de fichiers maintiennent des structures ont sont suivis les blocs libres. Ces structures sont souvent des listes chaînées, contenant les adresses de tous les blocs libres. Ainsi, le système de fichiers est capable de satisfaire les besoins d'allocation de stockage des applications. UFS et Ext2FS utilisent ce que l'on appelle une bitmap[6] pour le suivi de ces blocs libres. Cette carte est un est un tableau de bits, où chaque bit est associé à un bloc dans le système de fichier de la partition. L'état de l'allocation de chaque bloc est reflété dans la carte, où un «1» représente un bloc logique alloué, et un «0» un bloc non-alloué, libre. Le principal problème avec ce genre de structure est que plus la taille d'un système de fichiers augmente, plus la taille de la carte augmente, puisque chaque bloc doit avoir son bit correspondant dans la carte. Et tant que nous utiliserons un algorithme de recherche séquentielle des blocs libres, nous constaterons une perte de performance puisque le temps de recherche augmentera de façon linéaire (complexité en O(n), où n est la taille de la carte). Notons que l'approche carte n'est pas si mauvaise quand la taille du système de fichiers est raisonnable, mais plus la taille augmentera, moins bien la structure se comportera.

La solution fournie par les systèmes de fichiers de nouvelle génération est l'utilisation d'extents conjuguée à une organisation en arbre B+. L'approche extent est intéressante car elle permet de localiser plusieurs blocs disponibles en même temps. De plus, ils permettent aussi de réduire la taille de la structure de gestion, puisque plusieurs blocs sont gérés avec moins d'information. Par conséquent, un bit pour chaque bloc n'est plus nécessaire. En outre, avec l'utilisation des extents, la taille de la structure de gestions des blocs libre ne dépend plus de la taille du système de fichiers (la taille de structure dépend du nombre d'extents gérés). Néanmoins, si le système de fichiers est si fragmenté qu'il existe un extent pour chaque bloc libre, la structure de gestion serait plus grande que pour l'approche à bitmap. Notez que les performances devraient être sensiblement augmentées si notre structure gérait uniquement les blocs libres, puisque moins d'éléments auraient à être visités. En outre, avec l'utilisation d'extents, même s'ils sont gérés en liste et que des algorithmes séquentiels de balayage soient utilisés, les performances en seraient quand même améliorées puisque la structure englobe

plusieurs blocs dans un extent, réduisant ainsi le temps de localisation d'un certain nombre de blocs libres.

La seconde approche pour surmonter le problème de gestion des blocs libres est l'utilisation de structures complexes qui amènent à l'utilisation d'algorithmes de balayage de moindre -complexité. Nous tous savons qu'il y a de meilleures voies d'organiser un ensemble d'éléments devant être localisés que l'utilisation des listes chaînées avec des algorithmes de balayage séquentiels. Les arbres B+ sont utilisés puisqu'ils peuvent localiser des objets rapidement. Ainsi, les blocs libres sont organisés en arbres B+ au lieu de listes, afin de tirer profit des algorithmes de recherche rapide. Quand plusieurs blocs libres sont demandés par les applications, le système de fichiers traverse «l'arbre de gestion des blocs libres», afin de localiser l'espace libre demandé. Il existe en outre l'approche «Arbre B+ d'extents», où des extents et non des blocs sont organisés dans l'arbre. Cette approche rend différentes techniques d'indexation possibles. L'indexation par taille d'extent, mais également par la position des extents, sont des techniques mises en application qui rendent le système de fichiers capable de localiser rapidement plusieurs blocs libres, que ce soit par leur taille ou par leur emplacement.

4.2.2. Gestion d'un grand nombre d'entrées de *répertoire*

Tous les systèmes de fichiers utilisent un objet spécial appelé répertoire. Les répertoires, vus du système de fichiers, est un ensemble d'entrées de répertoire. Ces entrées sont des paires (numéro d'inode, nom de fichier), où le «numéro d'inode» est le numéro d'inode (structure interne du système de fichiers) utilisé pour gérer les informations relatives aux fichiers. Quand une application veut rechercher un certain fichier dans un répertoire, par son nom de fichier, la «structure d'entrées de répertoire» doit être balayée. Les anciens systèmes de fichiers ont organisé les entrées de répertoire par des listes chaînées, utilisant alors des algorithmes de recherche séquentiels. Par conséquent, avec des répertoires de grande taille, référençant des milliers de fichiers et d'autres répertoires, les performances se réduisent d'autant. Ce problème, comme celui décrit pour les blocs d'espace libre, est étroitement lié à la structure utilisée. Les systèmes de fichiers de nouvelle génération ont ainsi besoin de structures de données et d'algorithmes plus performants pour localiser rapidement un fichier dans un répertoire.

Solution employée : les systèmes de fichiers passés en revue ici utilisent les arbres B+ pour organiser les entrées de répertoire, ce qui améliore les temps de recherche. Dans ces systèmes de fichiers, les entrées de chaque répertoire sont indexées par le nom de fichier.

Ainsi, quand un fichier d'un répertoire donné est demandé, l'arbre B+ de ce répertoire est traversé pour localiser rapidement l'inode correspondant. L'utilisation des arbres B+ est liée à chaque système de fichiers. Il existe en effet des systèmes de fichiers qui gèrent un arbre par répertoire, alors que d'autres gèrent un arbre par arborescence de fichiers.

4.2.3. Fichiers de grande tailles

Quelques anciens systèmes de fichiers ont été conçus avec une certaine idée des conditions d'utilisation. Ext2fs et UFS ont été conçus en partant du principe que les systèmes de fichiers contiendraient essentiellement de petits fichiers. C'est ce qui expliquent ce à quoi ressemblent les inodes de Ext2FS ou UFS. Pour ceux qui ne connaissent pas encore la notion d'inode, nous allons l'expliquer brièvement.

Un inode est la structure employée par UFS et Ext2FS pour gérer les informations relatives aux fichiers. Sont gérés dans l'inode les permissions sur le fichier, son type, le nombre de liens[7], et les pointeurs sur les blocs du système de fichiers.

Un inode peut contenir des pointeurs directs qui sont les adresses logiques des blocs logiques du système de fichiers utilisés pour stocker le contenu du fichier, mais il peut aussi contenir des pointeurs indirects simples, voire doubles ou même triples. Les pointeurs indirects contiennent l'adresse de blocs logiques où sont d'autres pointeurs qui eux donnent l'adresses des blocs du fichier. Les indirections doubles pointent sur des blocs contenant des indirections simples. De même, les indirections triples pointent sur des indirections doubles. Le problème avec cette technique d'adressage est que si la taille du fichier croit, indirections simples, doubles et même triples sont utilisées.

Notons que l'utilisation de pointeurs indirects mène à l'augmentation du nombre d'accès disques, car un plus grand nombre de blocs doit être lu pour trouver le bloc demandé. Cela mène à une augmentation du temps de recherche en corrélation avec l'augmentation de la taille des fichiers. Vous pourriez vous demander pourquoi les

créateurs d'Ext2FS n'ont pas uniquement utilisé les pointeurs indirects simples, démontrés plus rapides ?

La principale raison en est que les inodes ont une taille fixe, et que l'utilisation de pointeurs simples uniquement amènerait à utiliser des inodes de plus grande taille, pour stocker tous les pointeurs directs utilisables, ce qui gaspillerait inutilement de l'espace disque lorsque l'on stocke de petits fichiers.

Inode (Ext2FS) : les fichiers de plus grande taille nécessitent plus d'accès disque, car utilisant des indirections simples, voire doubles et même triples pour accéder aux données.

Solution employée : les nouveaux systèmes de fichiers doivent alors continuer à utiliser l'espace efficacement, et fournissent de meilleures techniques d'adressage pour des fichiers de plus grande taille. La principale différence avec les anciens systèmes de fichiers est, une fois de plus, l'utilisation d'arbres B+. Les systèmes de fichiers que nous étudions utilisent les arbres B+ pour gérer les blocs des fichiers. Les blocs sont classés par leur adresse (offset) dans le fichier; puis, quand une certaine adresse dans le fichier est demandée les sous-routines du système de fichiers parcourent l'arbre des blocs pour localiser le bloc requis. Ces techniques employées sont aussi propres à chaque système de fichier.

Afin de réduire au minimum l'utilisation des pointeurs indirects, nous pourrions penser à utiliser de plus grands blocs logiques. Ceci mènerait à un ratio d'information sur nombre de pointeurs plus élevé, ayant pour résultat une utilisation moindre des indirections. Mais, ces blocs logiques plus grands augmentant la fragmentation interne, d'autres techniques sont utilisées. L'utilisation des extents rassemblant un ensemble de plusieurs blocs logiques est une de ces techniques.

L'utilisation des extents en lieu et place des pointeurs sur blocs causerait le même effet que l'augmentation de la taille des blocs, puisque le taux information sur nombre de pointeurs augmente. Certains des systèmes de fichiers passés en revue utilisent des extents pour surmonter le problème de l'adressage dans les fichiers de grande taille.

D'ailleurs, des extents peuvent être indexés dans un arbre B+ par leur adresse à

l'intérieur du fichier, améliorant ainsi les temps de lecture. Les nouveaux inodes maintiennent ainsi, jusqu'à un certain point, quelques pointeurs directs sur les extents, et, dans le cas où le fichier aurait besoin de plus d'extents, ceux-ci seront organisés dans un arbre B+. Pour maintenir un bon niveau de performance lors de l'accès aux fichiers de petite taille, les systèmes de fichiers de nouvelle génération stockent les données directement dans l'inode. Ainsi, chaque accès à l'inode permettra un accès immédiat aux données. Cette technique est particulièrement utile pour les liens symboliques où l'information stockée est infime.

Tableau 2. Capacités et particularités des systèmes de fichiers étudiés

Techniques / Système de fichiers	Gestion des blocs libres	Utilisation d'extents pour l'espace libre	B-arbres pour les entrées de répertoire	B-arbres pour adresser les blocs des fichiers	Extents pour adresser les blocs des fichiers	Données de petits fichier dans l'inode)	Lien symbolique dans l'inode	Entrées de (petits) répertoires dans l'inode
XFS	Arbres B+, indexés par leur offset et leur taille	OUI	OUI	OUI	OUI	OUI	OUI	OUI
JFS	Arbre & Binary Buddy[a]	NON	OUI	OUI	OUI	NON	OUI	8 au maximum
ReiserFS[b]	Bitmap based	Non supporté pour l'instant	Sous-arbre de l'arbre du SF	Dans l'arbre du SF	A venir dans la version 4	[c]	[c]	[c]

Ext3FS	
Remarques : a. Binary Buddy : JFS emploie une approche différente pour organiser les blocs libres. I	

5. Autres améliorations

Il existe d'autres limitations sur les systèmes de fichiers comme UFS. Par exemple, l'incapacité de gérer les fichiers à trous (sparse files), ou encore le problème du nombre fixe d'inode sur un système de fichiers.

5.1. Support des fichiers à trous

Supposons que nous venons de créer un nouveau fichier, et avons écrit deux octets au début de ce même fichier. Rien de particulier à cela. Qu'arrivera-t'il si maintenant nous tentons d'écrire à l'offset 10000 de ce fichier ? Le système de fichier devra donc évaluer le nombre de blocs nécessaire pour combler le trou entre l'offset 2 et l'offset 10000. Cela pourrait prendre un moment. Maintenant, pourquoi le système de fichier doit-il allouer ces blocs, alors que nous n'avons pas exprimé d'intérêt à leur égard ? La réponse à cette question est le support des fichiers à trou, support intégré aux systèmes de fichiers de nouvelle génération.

Le support des fichiers à trous est étroitement lié à la technique d'adressage des extents regroupant les blocs d'un fichier. Le champ «offset dans le fichier» de l'extent sera utilisé à cette fin. Ainsi, quand le système doit chercher des blocs d'espace libre pour combler le trou ouvert dans une situation comme celle décrite ci-dessus, il alloue un nouvel extent qui aura comme offset l'offset dans le fichier désiré. Ensuite, quand une application lira les octets correspondant au trou, elle se verra retourner une valeur nulle,

puisque aucune information n'est présente. Enfin, le trou pourra être comblé par les applications qui voudront y écrire.

5.2. La solution à la fragmentation interne de ReiserFS

Quand nous avons parlé de fragmentation interne et de performance, nous avons signalé que les administrateurs système ont souvent le choix entre les performances et la perte d'espace utile. Si nous regardons le premier tableau, nous verrons que les systèmes de fichiers actuels peuvent gérer des blocs de tailles pouvant atteindre 64 Ko. Cette taille de bloc, voire même une taille inférieure, induit des pertes importantes d'espace utile, pertes dues à la fragmentation interne. Afin de rendre faisable l'utilisation de blocs de grande taille, ReiserFS met en application une technique qui résout le problème.

Comme nous l'avons dit plus haut, ReiserFS utilise un arbre B*[8] pour organiser les objets du système de fichiers. Ces objets sont les structures utilisées pour gérer les informations sur les fichiers, par exemple les dates d'accès, permissions, etc... En d'autres termes, les informations contenues habituellement dans un inode, les répertoires et les données des fichiers. ReiserFS appelle respectivement ces objets éléments de statut, éléments de répertoire, et éléments directs / indirects. Les éléments indirects sont une série de pointeurs vers des noeuds non formatés. Les noeuds non formatés sont des blocs logiques sans format particulier, utilisés pour stocker les données des fichiers. Les éléments directs contiennent directement les données des fichiers. En outre, ces éléments sont de taille variable et sont enregistrés dans les feuilles de l'arbre, parfois avec d'autres au cas où il y aurait assez d'espace dans la feuille. C'est pourquoi nous avons dit plus haut que l'information à propos d'un fichier est stockée au plus près des données des fichiers, puisque le système de fichiers essaye toujours de garder ensemble les éléments de statut et les éléments directs et indirects d'un même fichier. Remarquons enfin que, comparées aux éléments directs, les données du fichier pointées par les éléments indirects ne sont pas enregistrées dans l'arbre. Cette gestion spéciale des éléments directs est due au support des fichiers de petite taille.

Les éléments directs sont destinés enregistrer les données de fichiers de petite taille, et

même les queues[9] des fichiers. Par conséquent, plusieurs queues de fichiers peuvent être stockées dans la même feuille, réduisant ainsi le gaspillage d'espace dû à la fragmentation interne. Le problème corrolaire à cette technique est l'éventuelle augmentation de la fragmentation externe, puisque les données des fichiers s'éloignent des données de queues. D'ailleurs, la tâche de compactage des queues prend du temps et amène à une diminution des performances. C'est une conséquence des décalages de mémoire nécessaires quand quelqu'un ajoute des données à un fichier. Quoi qu'il en soit, l'utilisation de la technique de compactage des queues peut être interdite en fonction de que veut faire l'administrateur. Une fois encore, comme auparavant, c'est encore au choix de l'administrateur.

5.3. Allocation dynamique d'inode

Un des problèmes principaux des systèmes de fichiers du type de UFS est leur restriction à un nombre fixe d'inodes. Ces inodes contiennent les informations liées à chaque objet du système de fichiers. Ainsi, ce nombre fixe d'inode restreint le nombre d'objets gérables sur un système de fichiers. Au cas où tous les inodes du système de fichiers seraient utilisés, nous devons sauvegarder la partition, puis recréer le système de fichiers avec un plus grand nombre d'inodes. La raison de la fixité de ce nombre est que UFS emploie des structures de taille fixe pour gérer l'état des inodes, de la même façon que pour les blocs d'espace libre.

En outre, UFS alloue l'emplacement pour les inodes à des adresses fixées à l'avance. Cela évite de gérer d'autres pointeurs pour chercher et utiliser ces inodes.

Le problème pour les administrateurs est de savoir combien d'objets auront à gérer leur système de fichier. La politique « créer un système de fichier avec le maximum d'inodes possible » n'est pas toujours la meilleure, puisque l'espace pour les inodes est réservé, et ne peut être utilisé pour autre chose. Cela gaspillerait beaucoup d'espace.

Pour surmonter ce problème est apparue l'allocation dynamique d'inode. Cela évite aux administrateurs système de devoir deviner le nombre maximal d'objets au moment de la création du système de fichier. Mais cette utilisation de structures dynamiques apporte d'autres problèmes : allocation et adressage des inodes dans l'espace des blocs logiques, etc...

Les systèmes de fichiers vus ici utilisent des arbres B+ pour organiser les inodes. Mieux, JFS utilise des extents d'inodes qui forment les feuilles de l'arbre et qui peuvent contenir jusque 32 inodes. Il existe aussi d'autres structures permettant d'allouer les inodes auprès des autres objets du système de fichier. En conséquence, l'utilisation de techniques d'allocation dynamique d'inodes est complexe et prend du temps, mais aide à dépasser les limites des anciens systèmes de fichiers.

Tableau 3.

Techniques	Allocation dynamique d'inodes	Suivi dynamique des inodes	Support des fichiers à trous
XFS	OUI	Arbre B+	OUI
JFS	OUI	Arbre B+ et extents d'inodes	OUI
ReiserFS	OUI	Arbre B* principal[a]	OUI[b]
Ext3FS	ND	NON	ND
Remarques : a. car ainsi que nous l'avons expliqué dans la section sur la solution de ReiserFS à la fra			

6. Références

6.1. Pages Web des différents systèmes de fichiers

- Ext2FS (http://web.mit.edu/tytso/www/linux/ext2.html)
- ReiserFS (http://www.reiserfs.org/)
- XFS (http://oss.sgi.com/projects/xfs/)

- JFS (http://oss.software.ibm.com/developerworks/opensource/jfs/)

6.2. Bibliographie

- JFS overview and layout white papers by Steve Best and Dave Kleikamp
- *XFS: A Next Generation Journalled 64-Bit Filesystem With Guaranteed Rate I/O* by Mike Holton and Raj Das. SGI, Inc.
- *Scalability in the XFS File System* par Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. SGI, Inc.
- *Scalability and Performance in Modern File Systems* de Philip Trautman and Jim Mostek; ReiserFS web site papers.
- *Design and Implementation of the Second Extended Filesystem* by Rémy Card, Theodore Ts'o, Stephen Tweedie.
- Liste de distribution des développeurs ReiserFS. Pour s'inscrire, envoyer un message à reiserfs-list-subscribe@namesys.com (mailto:reiserfs-list-subscribe@namesys.com)
- Liste de distribution JFS. Pour s'inscrire, envoyer un message à majordomo@oss.software.ibm.com (mailto:majordomo@oss.software.ibm.com) avec le mot-clé «subscribe» dans le sujet et «subscribe jfs-discussion» dans le corps du message.
- *Database System concepts*, de Abraham Silberschatz et Henry F. Korth, McGraw-Hill, 1993

L'auteur aimerait remercier Stephen C. Tweedie, Dave Kleikamp, Steve Best, Hans Reiser, les assidus des listes de distribution de JFS et ReiserFS pour les discussions fructueuses et leurs réponses.

Paru dans le numéro 55 de la Linux Gazette de juillet 2000.

Traduction française par Jérôme Fenal (mailto:jerome@fenal.org).

Notes

1. NdT : le terme anglais extent ne trouvant sa traduction que dans une périphrase longue (à savoir : unité d'allocation d'espace de taille variable), le terme anglais sera conservé tel quel.
2. NDE : Dans le schéma, les clés sont les noms de fichiers. La ligne au dessus des cases rouges contient une clé pour chaque fichier du répertoire : ce sont les feuilles. Au dessus se situent les noeuds, les clés étant choisies par le système pour optimiser l'accès aux données.
3. NdT : on trouve aussi B-Arbre, dans la littérature en français de géométrie algorithmique
4. NdT : UFS est en fait d'origine BSD, ce qui nous le fait retrouver sur SunOS4 (et donc Solaris), Ultrix (et donc OSF/1 - Digital Unix - Tru64), et d'autres encore. Le système de fichiers des Unix System V est HFS, Hierarchical File System, que l'on retrouve par exemple sur HP-UX
5. NdT : noyau Linux, mais le lecteur trouvera aussi ce genre de chose sur d'autres Unix
6. NdT : carte de bits, mais nous conserverons le terme original
7. NdT : liens matériels, hard-links
8. NdT : un descendant des arbres B et B+
9. NdT : du terme anglais « tail », qui sont en fait les données en fin du fichier qui ne remplissent pas un bloc entier. Voir à ce sujet l'option de montage `notail` de ReiserFS, utilisée par exemple avec LILO pour démarrer Linux grâce à une simple liste de blocs.