Implémentation

Maintenant on va voir comment la structure vue précédemment est implémentée dans le système d'exploitation.

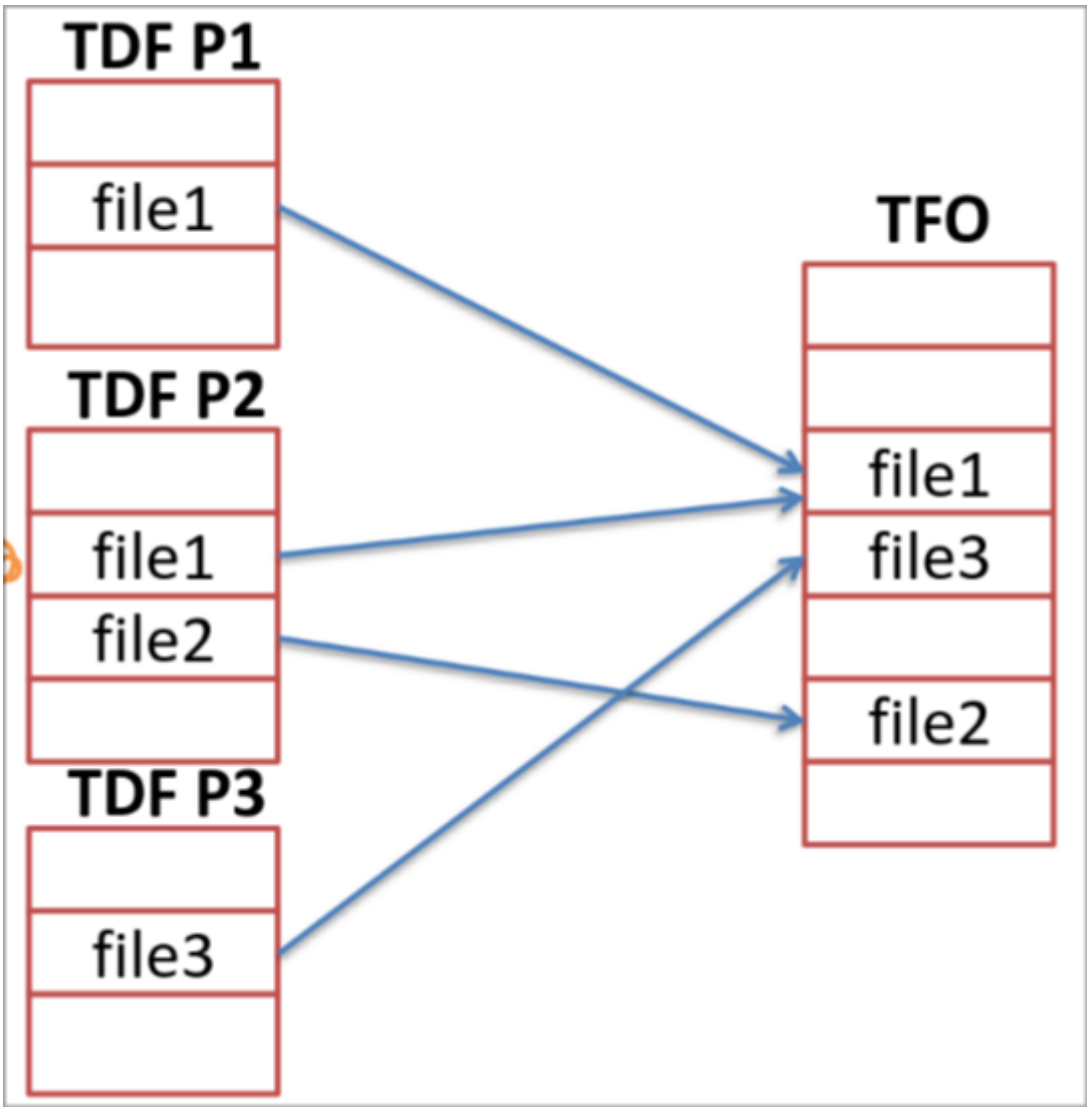
Informations stockées

Sur le disque on va stocker :

- Le **boot control block**, qui sont les informations nécessaires pour démarrer le système d'exploitation
- Le **partition control block**, qui contient les informations détallant les partitions en cours (nombre de blocs, taille, etc), le système le plus utilisé aujourd'hui est GPT, autrefois, c'était MBR.
- Le **root directory** qui est le répertoire principal pour le stockage des dossiers et fichiers
- Le **file control block** qui contient les informations (propriétaire, permissions, etc) sur les fichiers (sous Unix, on va parler d'**i-nodes**)

En mémoire, on va retenir :

- La table des partitions montée et des chemins d'accès (vous pouvez aussi la voir dans `/proc/mounts`)
- Les informations sur les répertoires récemment visités, car s'ils sont récemment visités, ils ont de grandes chances de l'être souvent.
- La **table générale des fichiers ouverts** (TFO) qui décrit tous les fichiers ouverts sur le système. À savoir que si un fichier est ouvert par deux processus différents, il n'apparaîtra qu'une fois dans cette table.
- La **table des fichiers par processus** (TDF, table des descripteurs de fichiers), décrit les fichiers ouverts pour le processus courant en incluant des informations supplémentaires par rapport à la TFO tel que la position courante et le mode d'ouverture.



Ainsi, lorsqu'un fichier est créé, on va stocker un FCB (file control block ou i-node) pour y inclure les permissions, la date, le propriétaire, le groupe, la taille, le premier bloc, etc.

Ensuite le fichier va être enregistré dans la TFO (qui comprend les informations du FCB ainsi que le nombre de processus lié à ce fichier).

Le fichier va aussi être enregistré dans la TDF du processus pour inclure le mode d'accès et le pointeur de position courante.

Lorsque le fichier est fermé, on supprime l'entrée de la TDF du processus et on met à jour la TFO. On ne supprimera ce fichier de la TFO que si plus aucun processus n'utilise le fichier.

Implémentation des répertoires

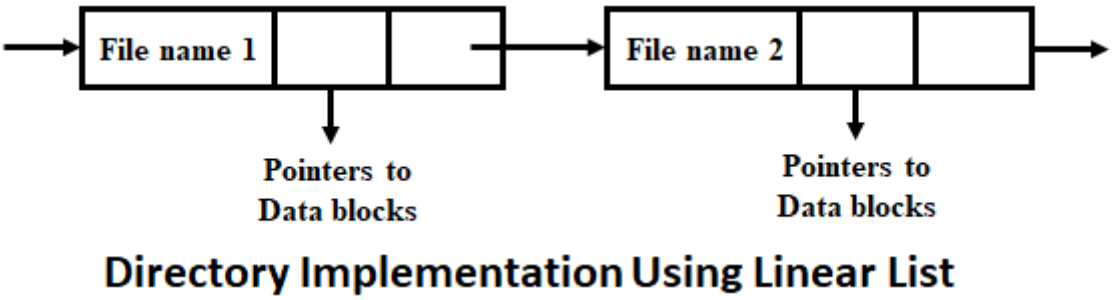
Maintenant, il faut trouver une manière d'implémenter les répertoires de manière que leur gestion puisse être rapide et qu'il soit facile de retrouver l'information.

Plusieurs méthodes sont possibles, mais le choix dépendra de si on souhaite favoriser la rapidité ou l'espace disque.

Il faut donc pouvoir gérer le fait qu'un dossier peut contenir un autre dossier aussi bien que des fichiers.

> Pour en savoir plus sur les deux solutions vu ci-dessous, vous pouvez consulter cet article.

Répertoires sous forme de liste



Directory Implementation Using Linear List

La première solution est simplement de considérer chaque répertoire comme une liste contenant les noms des fichiers avec un pointeur vers le début de ces fichiers.

Cela implique donc qu'il faudra faire une recherche séquentielle pour trouver un fichier dans un répertoire, de même cela implique qu'il faudra faire cette même recherche séquentielle pour créer le fichier (pour vérifier si le nom existe) ainsi que pour le marqué comme libéré.

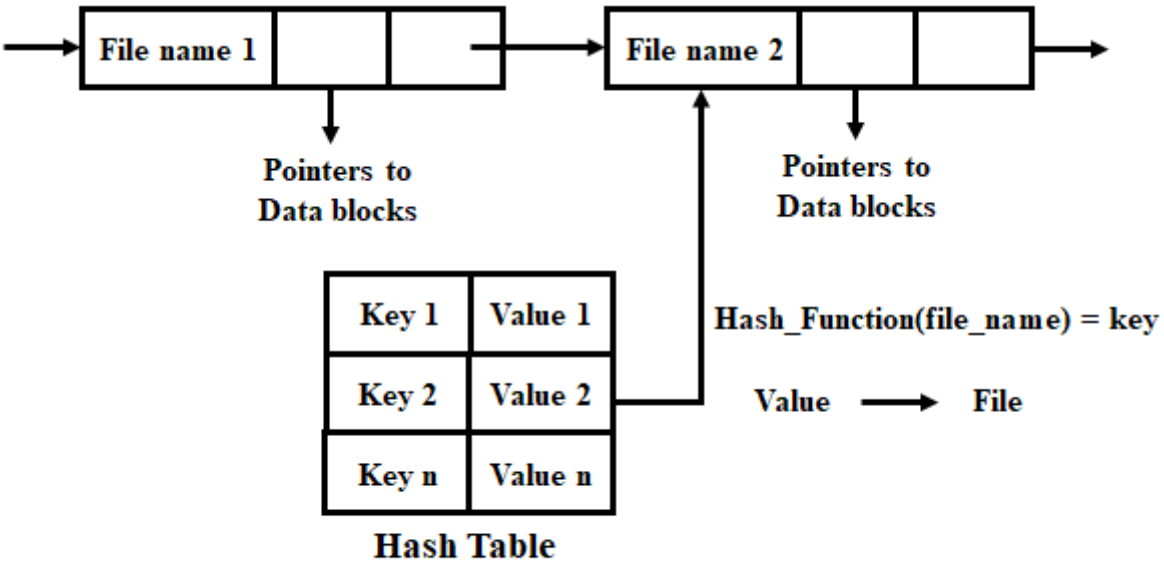
Les inconvénients de cette méthode sont que le parcours séquentiel ralenti considérablement l'utilisation. Si on décide de faire une liste triée, le problème sera toujours là, car il faudra toujours maintenir la liste triée.

Note sur la libération de l'espace

Marquer l'espace comme libéré consiste à marquer l'entrée du répertoire comme inutilisée.

> Il est important de se rappeler que lorsqu'on supprime un fichier ou un dossier, ce dernier n'est pas réellement supprimé, l'espace est simplement marqué comme étant libre. Il est donc tout à fait faisable de récupérer ces informations si on lit séquentiellement les espaces libres. Pour supprimer définitivement un fichier, on peut utiliser la commande `shred`

Répertoires sous forme de table hashée



Directory Implementation Using Hash Table

L'idée de cette deuxième solution est d'avoir une liste d'éléments, mais avec en plus une table de hash donnant directement le pointeur du fichier.

Ainsi, lorsque l'on cherche un fichier avec un certain nom, on va hasher le nom du fichier, ce qui va nous donner la clé dans la table. Ensuite, on va pouvoir trouver le pointeur vers le fichier directement en la récupérant depuis la table avec la clé.

Cette méthode a l'avantage d'être très rapide, car il n'y a pas besoin de parcourir quoi que ce soit pour trouver les fichiers. Il est toute fois important de noter qu'il faut trouver des solutions pour traiter les cas où plusieurs noms arriveraient au même hash (**collision**).

Un désavantage de cette méthode est que les tables hashées ont généralement une table fixe et que la performance de la fonction de hash est aussi dépendante de la taille de la table. C'est pourquoi il faut soit limiter la taille du dossier, ou bien étendre la table lorsqu'elle est pleine.

Allocation des fichiers

Maintenant, il va falloir trouver un moyen d'allouer efficacement les fichiers le plus rapidement possible.

On va parler ici de trois méthodes couramment utilisées, la méthode utilisée dépend du système d'exploitation.

Allocation contiguë

L'allocation contiguë signifie que chaque fichier occupe des blocs qui se suivent sur le disque. Ainsi, si quand un bloc est en cours de lecture, la lecture du bloc suivant ne requiert pas de mouvement de la tête de lecture, ce qui offre donc une performance intéressante.

Également, si on sait combien de blocs occupe le fichier et que l'on sait la position de son premier bloc, on peut calculer la position du dernier bloc du fichier, ainsi cela fait un accès séquentiel et direct, facile et rapide.

Problèmes

Le souci avec l'allocation contiguë est que l'on va se retrouver avec des problèmes de fragmentation externe à cause des allocations et libérations successibles. Pour résoudre ce problème, il faut donc utiliser le compactage (la défragmentation). C'est une opération qui peut être dangereuse si elle est interrompue ainsi que chronophage.

Un deuxième problème est de déterminer la taille initiale du fichier, car les fichiers ont tendance à croitre avec le temps. Si un fichier est suivi directement par un autre fichier, que faire si on veut faire croitre le fichier A ?

Si on veut copier le fichier à un autre endroit, cela va rendre le système beaucoup plus lent.

Si on prévoit un "buffer" pour permettre au fichier de croire, alors ça nous mène à de la fragmentation interne **et** externe.

Résolution du problème de la fragmentation sous Linux

Pour régler ces problèmes, cependant Linux (avec ext4) utilise une variante. À la place de placer chaque fichier à la suite des autres. Les fichiers sont éparpillés sur le disque de manière à maximiser les espace entre eux.

De cette manière, on s'assure que les fichiers ont toujours suffisamment de place pour grandir et quand les fichiers sont trop rapprochés, le système d'exploitation va les réarranger pour les espacer à nouveau.

Grâce à cette technique, la défragmentation est presque entièrement inutile sous Linux, car le taux de fragmentation reste toujours très bas.

> Si vous voulez en savoir plus, vous pouvez consulter cet article Wikipedia sur ext4 ou cet article comparant la fragmentation sous Linux et Windows.

Allocation chainée

L'allocation chainée consiste à ce que chaque fichier soit une liste de blocs chainés entre eux. Ainsi, le répertoire contient un pointeur vers le premier bloc de la liste et chaque bloc contient un lien vers le bloc suivant.

Cela a l'avantage de ne pas avoir de fragmentation externe et le fichier peut croitre sans problème puis ce que le bloc peut être écrit n'importe où.

Problème

Le problème de cette méthode est que le contenu des fichiers va être éparpillé partout, la tête de lecture va donc devoir beaucoup voyager et donc ralentir la lecture et l'écriture.

Pour résoudre ce problème, le système FAT alloue des groupes de blocs (cluster) plus tôt que des blocs. Le problème toute fois avec cette méthode, c'est que cela crée de fragmentation interne si les clusters ne sont pas utilisés complètement.

Allocation indexée

Le principe de l'allocation indexée est de garder un bloc "index" pour chaque fichier. Le bloc va contenir les informations des positions de tous les autres blocs du fichier.

L'avantage principal de l'allocation indexée est qu'elle permet un accès direct aux différents blocs utilisés dans le fichier. Cela permet donc un accès au fichier beaucoup plus rapide si on veut accéder à un point précis.

Problème

Un problème avec cette méthode est que l'on ne sait pas d'avance la taille nécessaire de l'index. Ainsi, on peut soit lier les index entre eux (via une liste chainée) ou alors utiliser des index d'index (index à plusieurs niveaux).

Un autre problème est que cela empire le problème de tête de lecture de l'allocation chainée, car il faudra, en plus de devoir passer sur tous les blocs éparpillés, passer sur tous les blocs d'index.

Également, pour les très petits fichiers (2 ou 3 blocs), l'allocation indexée garde un bloc complet pour l'index, ce qui est donc beaucoup moins efficace au niveau de l'utilisation du disque.

Quelle méthode d'allocation choisir ?

La méthode d'allocation à choisir dépends donc de la façon dont le système sera utilisé, car chaque méthode montre des différences (niveau temps, gaspillage, etc) comme nous l'avons vu juste avant.

Une idée est donc de permettre différents types d'allocations différentes ou de mêler plusieurs méthodes.

Par exemple, certains systèmes utilisent l'allocation contiguë pour les petits fichiers et l'allocation indexée pour les fichiers plus gros ou grandissent.

Gestion de l'espace libre

Nous avons vu précédemment le fonctionnement de la gestion de l'allocation, cependant pour faire cela, il faut prendre un bloc libre et l'utiliser. Il faut donc avant tout trouver un moyen de trouver un bloc libre.

On peut donc utiliser les mêmes méthodes que celles utilisées pour la gestion de mémoire. C'est-à-dire l'utilisation de table de bits ou de liste chaînées.

En utilisant une table de bits (ou bitmap en anglais), on va garder une table qui mentionne pour chaque bloc s'il est libre ou occupé. L'avantage est que cette méthode est assez simple et efficace. En revanche, le désavantage est que pour des gros volumes cette table va prendre beaucoup de place. Par conséquent, cette méthode est intéressante pour des volumes faibles, mais devient embêtante pour des volumes plus grands.

L'autre solution est d'utiliser une liste chainée, ainsi chaque bloc libre connait le bloc libre suivant. On peut également faire en sorte de grouper les blocs libre de manière à pouvoir les allouer plus rapidement sans devoir tous les parcourir un à un.

> `ext2` (précurseur de ext3 et ext4) utilisait une liste chainée pour stocker les blocs libres. Cependant, cela menait à plus de fragmentation, car les fichiers étaient collés les un aux autres (voir Résolution du problème de la fragmentation sous Linux pour comprendre pourquoi).
>
> Ensuite `ext3` a commencé à utiliser une table de bit. Cependant, ce système faisait les choses bloc par bloc, alors `ext4` a remplacé ce système par un nouveau qui fait plusieurs blocs d'un coup ce qui améliore drastiquement les performances. De plus, `ext4` est optimisé de manière à séparer les allocations pour éviter la fragmentation.

Restauration des données

Vérification et correction

Le système de fichier peut devenir incohérent et des erreurs peuvent apparaitre à cause d'arrêt inattendu du système ou encore de problèmes matériels. C'est pour ces raisons que le système d'exploitation doit mettre en place des mécanismes pour vérifier la cohérence du système et corriger les erreurs.

La vérification consiste à parcourir les différents blocs des différents fichiers et à résoudre les problèmes qui pourraient survenir, par exemple :

- Un bloc est défini 2 fois comme libre, dans ce cas, il suffit de supprimer l'une des deux occurrences (ou lien si c'est une liste chainée)
- Un bloc est défini comme à la fois libre et à la fois occupé. Alors, on considère qu'il est occupé.
- Un bloc n'est défini ni comme libre, ni comme occupé. Alors, on considère qu'il est libre.
- Un bloc (ou une séquence de blocs) est défini comme occupé par deux fichiers différents. Alors, on duplique ces blocs communs dans les deux fichiers.

Sauvegarde et restauration

La **sauvegarde** consiste souvent en la copie de donnée, ailleurs. Elle est prévue pour une restauration rapide.

L'**archivage** consiste à sauvegarder des données plus longtemps, dans quel cas il faut également faire attention au média utilisé pour qu'il soit fiable, éprouvé, robuste et durable.

Il y a deux types de sauvegardes, les sauvegardes **incrémentales** et **différentielle**.

La sauvegarde incrémentale consiste à seulement sauvegarder les changements. Par exemple, la première fois, on fait une sauvegarde complète, la deuxième fois, on fait une sauvegarde de ce qui a changé depuis la première fois, et la troisième fois, on fait une sauvegarde de ce qui a changé depuis la deuxième fois.

La sauvegarde différentielle consiste à ne copier que les modifications ayant changé depuis la dernière sauvegarde **complète**. Par exemple, la première fois, on fait une sauvegarde complète, la deuxième fois, on fait une sauvegarde de ce qui a changé depuis la première fois, la troisième fois, on fait une sauvegarde de ce qui a changé depuis la première fois (ce qui inclut donc une redondance de ce qui était dans la deuxième sauvegarde), etc.

La taille de la sauvegarde différentielle va donc beaucoup plus augmenter que celle de la sauvegarde incrémentale, jusqu'à la prochaine sauvegarde complète.

La sauvegarde incrémentale est donc plus légère et plus rapide, mais sera plus complexe à restaurer (car s'il y a eu 17 sauvegarde, il faudra restaurer les 17 éléments) tandis qu'avec la sauvegarde différentielle, il suffira de restaurer la dernière sauvegarde complète et la dernière sauvegarde différentielle.

Système de fichiers journalisé

Le système de fichier est quelque chose de complexe et sa manipulation nécessite beaucoup d'opérations. Un problème peut arriver n'importe quand et si l'opération en cours ne peut pas se terminer, alors cela peut mener à un état incohérent dont la correction pourrait engendrer une perte de donnée.

Le **système de fichier journalisé** permet de limiter les dégâts en gardant un historique des opérations en cours, de cette manière le système peut *défaire* les opérations non terminées en cas

de panne.

C'est un système qui est assez similaire à celui des transactions en base de données, il faut donc garantir l'atomicité (le fait qu'une action ne puisse pas être décomposée), la cohérence, l'isolation et la durabilité (ACID) pour chaque action du journal.

Revision #2
Created 4 January 2024 16:47:31 by SnowCode
Updated 6 January 2024 19:13:15 by SnowCode