SCIENCES SORBONNE UNIVERSITÉ

M1 - SAR

Projet PNL
- MU4IN402 -

Life : Light Insertion File systEm

Réalisé par :
Yassine Alallah - 28707696
XUE Lorie - 28705252
Cyril LIN - 28633410

Encadré par:
Julien Sopena

Année universitaire 2023 - 2024

I. Algorithmes et structures de données

a. write

Dans une première version write, dans l'implémentation de l'étape 3 :
Les blocs de données sont contigus, nous pouvons donc directement récupérer le bloc associé à la position en divisant la position par la taille d'un bloc.
Nous devons allouer tous les blocs s'ils ne le sont pas, qui sont au début des blocs à la position souhaitée.
Ensuite, nous bouclons jusqu'à qu'on ait parcouru le dernier bloc ou qu'on ait écrit toutes les données. Dans cette boucle, nous allouons les blocs inexistants. Nous calculons la taille à écrire et copions les données utilisateur vers le bloc de données, sans oublier de mettre à jour la position, les données restantes à écrire et la taille du fichier.

Dans la deuxième version write, dans l'implémentation de l'étape 5 :
Nous devons retrouver le bloc correspondant à la position. Il n'est désormais plus possible de simplement diviser la position par la taille d'un bloc, car nous créons des blocs non contigus.
Pour y remédier, nous devons parcourir les blocs du bloc d'index, et avons besoin d'un offset au sein d'un bloc correspondant à l'endroit de la position demandée. Dès qu'un bloc peut contenir notre position, nous enregistrons ces valeurs et quittons la boucle, en prenant soin qu'à chaque tour de boucle, nous diminuons notre offset de la taille du bloc.

À chaque appel à write, nous divisons le bloc par 2 au niveau de l'offset, c'est-à-dire que toute la partie du bloc 1 sera de 0 à offset et la partie du bloc 2 (nouveau) sera d'offset à la taille max du bloc de base.
Le bloc d'index est décalé et le nouveau bloc est ajouté. Ensuite, tout comme dans la 1ère version du write, dans la boucle, nous calculons la taille à écrire et copions les données utilisateurs vers le bloc de données. En prenant en compte l'allocation de nouveaux blocs et les décalages blocs dans le bloc d'index.

b. read

Pour l'étape 3, l'algorithme est le suivant :

1. Nous récupérons l'index bloc récupéré à l'aide des informations de l'inode données par le système de fichier ouichefs en lisant sur le disque.

2. À l'aide de cet index bloc, nous récupérons le bloc associé à la position que l'on souhaite lire.

3. Ce bloc correspond à une partie de notre fichier.

- Se mettre à la bonne position
- vérifier qu'on ne peut pas copier plus que la taille maximale du buffer
- copier à l'espace utilisateur
- mettre à jour les positions

4. Retourner le nombre d'octets copiés

Etant donné que la nouvelle version du write permet de créer de nouveaux blocs, les données ne sont pas contiguës. Nous utilisons donc la même méthode que dans la deuxième version du write pour récupérer l'offset et le bloc contenant la position que nous souhaitons lire.
À l'aide de l'offset et du bloc récupéré, nous copions le plus petit entre la taille du bloc et la taille des données demandées, sans oublier de mettre à jour les positions dans la structure file.

c. Défragmentation

Pour défragmenter les blocs de données, il est nécessaire de préparer des éléments tels qu'un compteur, qui nous permettra de calculer le nombre de données contigu et de libérer les blocs qui ne sont pas dans la partie du fichier défragmentée.

Pour commencer, il faudra une boucle imbriquée :

1. une boucle pour déduire le bloc dans lequel nous allons stocker les données
2. une boucle pour parcourir les blocs suivants et remplir le bloc de données(1.)

Pour chaque début de tour de boucle, nous vérifions que nous avons lu la taille du fichier. Si nous l'avons atteint, nous pouvons libérer le bloc de notre bloc d'index et quitter la boucle.
Dans la deuxième boucle, nous parcourons les blocs jusqu'à ce que le bloc de la première boucle soit rempli. De ce fait, nous préparons les informations du bloc (taille et numéro) du bloc courant et du bloc à remplir. Avec ces informations, nous calculons les données à transférer depuis le bloc courant au bloc à remplir. Après le transfert, les numéros de bloc seront mis à jour.

La complexité de l'implémentation était de pouvoir transférer les données de plusieurs blocs à la suite dans un seul bloc, d'où l'idée de devoir faire une seconde boucle

II. Statut du projet

a. Liste de fonctionnalités implémentées

- read
- commande ioctl pour afficher les informations d'un fichier
- commande ioctl pour défragmentation
- benchmark : il existe 2 versions du benchmark avec descripteur de fichier (benchmark_fd.c) et avec la structure FILE en C. Les deux versions sont fonctionnelles, mais la version avec descripteur de fichier est très lente, car les appels ne sont pas mis en tampon, il devient difficile de faire exécuter le programme.

b. Liste de fonctionnalités implémentées et non entièrement fonctionnelles

- write: notre write n'est pas optimisé, il va diviser le bloc en deux à chaque appel. De plus, la taille est maintenant codée sur 12 bits, la taille maximum d'un bloc devient 4 095, car on veut représenter le 0. Donc, il se peut qu'il y ait des blocs de taille 1 lorsqu'on écrit un bloc de taille de multiple à 4 096.

c. Liste de fonctionnalités non implémentées

- Implémentation avec page cache

III. Conclusion

Nous avons essayé d'implémenter toutes les fonctionnalités, la majorité d'entre elles fonctionnent bien que le benchmark avec un descripteur de fichier soit lent. De plus, sachant que les blocs ne sont plus contigus, notre read doit chercher le bon bloc sur lequel lire. Pour cela, il faut parcourir tous les blocs, alors qu'avec des blocs contigus, on peut le retrouver directement. Donc, par conséquent, notre nouveau read est plus lent.