Projet LIFE (Light Insertion File systEm)

Adelin-Flaviu FERCHE 3800655
Paris Felipe MOLLO 21305222
Adjévi KOVI 21314846

27 mai 2024

Table des matières

Introduction

L'objectif de ce projet est de mettre en oeuvre un système de fichier à insertion rapide, où l'ajout de nouvelles données au milieu d'un fichier sera plus rapide grâce à une gestion des blocs partiellement remplis contenu dans un fichier. En effet, dans un système de fichier classique, les blocs sont remplis de manière contiguë, et seul le dernier bloc peut être partiellement rempli. Par conséquent, lorsque l'on souhaite ajouter des données au milieu d'un fichier, il faut décaler toutes les données suivantes pour maintenir la séquentialité des blocs. Ainsi, dans ce projet, nous allons permettre à OuicheFS de remplir les blocs d'une manière plus flexible, ce qui permettra une insertion plus rapide des données.

Implémentaion d'un benchmark

Afin de pouvoir tester le bon fonctionnement de notre implémentation et surtout de comparer les performances avec un système de fichiers traditionnel, nous avons implementé une série de tests permettant des lectures et écritures de différents fichiers, de différentes tailles, et dans différentes positions. Basiquement, ce programme prend en entrée, un nombre de fichiers à générer, exécute des écritures et des lectures sur le système de fichier ext4, puis sur ouichefs, vérifie que les écritures et les lectures sont correctes et affiche les performances. (Fonctionnelle)

Implémentaion du read et du write

Pour le read, on récupère la liste des blocs du fichier sur le disque, puis nous sélectionnons l'indice du bloc qu'on va lire en prenant le quotient de la division entière de ppos par la taille maximale des blocs. Après, nous récupérons le bloc sur le disque, puis nous calculons l'offset dans le bloc, en prenant cette fois-ci, le reste de la division entière. Pour connaître le nombre d'octets à lire, nous prenons le minimum entre le size de l'inode et la taille maximale des blocs. Nous incrémentons ppos du nombre d'octets lus avant de retourner le nombre d'octets lues. On arrêtera de lire le fichier lorsque ppos sera égale au size de l'inode en retournant 0. (Fonctionnelle)

Pour le write, on récupère le bloc où écrire en faisant la même démarche que pour le read. Avant de récupérer le bloc sur le disque, on regarde s'il est contenu dans la liste des blocs du fichier, si ce n'est pas le cas, on l'alloue et on l'insère dans la liste. Pour le nombre d'octets à lire, on prend le minimum entre len et la taille maximale du bloc moins l'offset. Nous incrémentons ppos du nombre d'octets écrits, nous mettons à jour les informations nécessaires de l'inode, et nous retournons le nombre d'octets écrits. Cependant, nous nous sommes rendus compte tardivement que la fonction ne décale pas toutes les données suivantes pour maintenir la séquentialité des blocs, lors de l'insertion de données, au lieu de cela elle les écrase. Donc, on ne pourra pas la comparer, en terme de performance, avec le write avec fragmentation de bloc (Pas entièrement fonctionelle)

Implémentation d'un ioctl

Pour connaître les informations sur les blocs d'un fichier, nous avons dû d'abord diviser le bloc number sur 32bits : 12bits pour la taille effective du bloc et les 20 autres bits pour le bloc number. Pour cela, nous avons mis en place une fonction de conversion et des fonctions pour récupérer respectivement la taille effective et le bloc number. Nous avons également déclaré une nouvelle structure qui stocke pour chaque fichier : le nombre de bloc utilisé, le nombre de bloc partiellement rempli, le nombre d'octets perdus provoqué par la fragmentation interne et la liste des blocs du fichier. Cette structure nous permettra d'afficher ces informations lors de l'appel d'un ioctl sur le fichier. De plus, nous avons mis en place toutes les modifications nécessaires au bon fonctionnement de l'ioctl. (Fonctionnelle)

Implémentation du write avec fragmentation de blocs

L'objectif est de modifier le write déjà implementé pour permettre de rajouter des données sans avoir à décaler les données qui suivent. Pour cela, à chaque fois qu'on écrira sur des données existantes, on copiera le reste du contenu du bloc initial, à partir de la position où l'on veut insérer les données, dans un nouveau bloc alloué. Mais avant cela, on calculera le nombre de blocs nécessaires à cette insertion en divisant la somme de len et d'offset par la taille maximale des blocs. Puis, nous parcourerons, une première fois, la liste des blocs, à partir de la valeur de blocs dans l'inode, pour décaler chaque bloc du nombre de blocs nécessaires fois jusqu'à arriver au bloc où l'on souhaite insérer des données. Et une seconde fois, à partir du bloc où l'on veut insérer des données, pour effectuer les copies nécessaires pour permettre l'insertion. On veillera à toujours mettre à jour l'inode mais aussi la taille effective des blocs. (Fonctionnelle)

Implémentation du read avec fragmentations de blocs

On a adapté notre read pour permettre de lire un fichier après le passage d'un write avec fragmentation des blocs. Pour cela, on ne se base plus sur le size de l'inode pour arrter la lecture mais plutôt sur le nombre de blocs lu. Nous avons déclaré une variable globale qui comptera le nombre de blocs parcourus du fichier et, si le compteur est égale à blocs de l'inode moins un, pour le bloc d'indirection, alors on stoppera la lecture en retournant 0 et on réinitilisera le compteur à 0. De plus, pour le nombre d'octets à lire, on se base maintenant sur deux pointeurs qui parcoureront le bloc courant, dont l'un stockera la position du commencement des données et l'autre la position de fin. Ces deux modifications, nous permettront de lire un fichier ayant des blocs fragmentés. (Fonctionnelle)

Implémentation de la défragmentation

La fragmentation des blocs accélère l'écriture des données mais elle entraîne une dégradation des performances en lecture car les données ne seront plus contiguës et il

faudra donc lire plus de blocs pour récupérer la même quantité de données. Pour cela, nous avons mis en place un autre ioctl qui déclenchera la défragmentation, lorsqu'il sera appelé, pour rendre les données contiguës. D'abord nous avons créer une fonction qui permet de rendre les données contiguës au sein d'un bloc en supprimant les espaces vides entre les données s'il y en a. Puis, pour la défragmentation, nous parcourons, une première fois la liste des blocs du fichier et on appelle la fonction, citées ultérieurement, sur chaque bloc. Après, nous faisons une boucle imbriquée dont nous parcourons, une première fois, la liste des blocs, s' il reste de la place au bloc courant, alors nous parcourons, une deuxième fois, la liste des blocs mais, cette fois-ci, à partir du prochain bloc selon le bloc courant. On copie autant de données vers le bloc courant jusqu'à le remplir, en veillant à mettre à jour la taille effective des blocs modifiés, pour passer à l'itération suivante et ainsi de suite. A la fin, nous désallouerons les blocs qui seront vides. (Fonctionnelle)

Conclusion

Pour conclure, on remarque que la performance du read se dégrade lorsque les données ne sont pas contigues, environ deux fois plus lente que le read de la question 1.3. Cependant, lorsqu'on applique l'algorithme de défragmentation sur le fichier, la performance du read, de la question 1.5, redevient équivalente à celle du read avant fragmentation des données. Malheureusement, nous ne pouvons pas comparer les performances entre les deux writes, décalage et fragmentation. Nous supposons que le write avec fragmentation sera plus rapide que les writes dans un système de fichier classique.