LINGI1113 - SYSTÈMES INFORMATIQUE 2

Projet MINIX

Professeur :

Marc LOBELLE

Programme :

SINF13BA

Étudiants : (Groupe 7)

Julien COLMONTS 41630800

Vincent VAN OUYTSEL 19890900

Année académique 2012-2013

1 Introduction

Le projet MINIX est le troisième projet du cours de sytèmes informatiques de $3^{ième}$ Bac. Cette année, le but de ce projet était de modifier le système d'exploitation afin qu'il soit capable de défragmenter des fichiers. Afin de remplir ces objectifs, deux appels systèmes ont dû être implémentés. Nous détaillerons comment nous avons construit ces appels ainsi que les tests effectués dans ce rapport.

2 NFRAG

Avant de penser à l'implémentation de la fonction principale du calcul de fragments, nous devions connaître le moyen d'ajouter des appels systèmes dans le système d'exploitation. Pour ajouter un appel système dans MINIX, un nombre conséquent de fichiers doit être modifiés afin de reconnaître cet appel. Premièrement, il fallait savoir à quel processus système nous devions nous adresser. La fonction de calcul s'appliquant sur un système de fichier, le processus à atteindre était VFS. Il fallait donc ajouter le nom de la fonction dans un slot libre de la table d'appel de ce processus. Les numéros d'appel système de ce niveau étant répertoriés dans le fichier "/SRC/INCLUDE/MINIX/CALLNR.H", nous avons également ajouté une constante représentant la position, dans la table, de l'appel nfrag. Puis, dans l'optique de respecter les conventions de programmation dans l'environnement de MINIX, nous avons ajouté les prototypes de fonction dans "/SRC/SERVERS/VFS/PROTO.H". Nous avons implémenté ces fonctions dans un nouveau fichier (dans le même répertoire) : "FRAG.C". Les opérations réalisées dans ce premier niveau de l'appel système sont :

- Vérification du *path* donné en argument et récupération du chemin absolu vers ce fichier. (fonction *fetch_name*)
- Récupération du *vnode* associé au fichier.
- Vérification du type de fichier (renvoi d'erreur s'il s'agit fichier non régulier) et du nombre d'utilisateurs du fichier (si > 1, renvoi d'une erreur).

Ensuite, nous avons découvert que les opérations précises sur des informations d'un système de fichier se faisaient en passant un message du processus gérant l'ensemble des systèmes de fichiers (VFS) au processus du système de fichier lui-même (MFS).

- Transmission de la requête (numéro de l'inode à traiter) à "/SRC/SERVERS/VFS/REQUEST.C".

L'objectif suivant était donc d'ajouter un appel système s'exécutant dans le processus du système de fichier MINIX. A nouveau, il fallait ajouter des constantes pour les numéros d'appel système (cette fois, dans le fichier "/SRC/INCLUDE/MINIX/VFSIF.H"), les noms des fonctions correspondantes dans la table (MFS/TABLE.C) et définir les fonctions dans le fichier de prototypes. Ces nouvelles fonctions sont implémentées dans le fichier "/SRC/SERVERS/MFS/FRAGS.C". L'opération à effectuer pour NFRAG consiste à compter le nombre de fragments du fichier donné en paramètre.

Choix d'implémentation

Plusieurs possibilités s'offraient à nous pour atteindre notre objectif. Nous avons choisi la méthode suivante :

- On récupère l'INODE correspondant à notre fichier grâce au numéro d'INODE passé par message dans l'appel système.

– On calcule le nombre de blocs dans une zone. Celui-ci est enregistré sous forme logarithmique (base 2) dans le super bloc du système de fichier. Pour obtenir le nombre réel d'une zone, il suffit d'utiliser l'opérateur "<<" (shift left). On peut obtenir la taille d'une zone en multipliant ce nombre par la taille d'un bloc.
– Pour compter les fragments, on va parcourir le fichier zone par zone et vérifier que celle-ci sont contigues. Comme l'accès à une zone ne se fait qu'à travers les blocs, on va utiliser le premier bloc de chaque zone. Si, pour deux zones consécutives dans les données du fichier, leurs premiers blocs respectifs sont situés à une longueur de zone près, les zones sont contigues.On va donc itérer l'offset d'une longueur de zone et utiliser la fonction READ_MAP pour obtenir le bloc voulu.
– On remet l'INODE sur disque et on renvoie le nombre de fragments à travers le message.

Cette méthode permet de compter efficacement le nombre de fragments. Elle est valide car une zone ne peut appartenir qu'à un seul fichier. La fonction de librairie associée à l'appel système renvoie soit le nombre de fragments (résultat ≥ 0) soit l'erreur signalant un problème dans l'exécution de l'appel système. L'erreur respecte les conventions de constantes pour ce type d'événement.

3 DEFRAG

Au niveau de la fonction de défragmentation, la première phase est similaire à ce que nous avons produit pour NFRAG. Le seul changement se situe dans la fonction implémentée dans le processus MFS. Ici, il ne s'agit plus de seulement compter le nombre de fragments, mais bien de le déplacer si nécessaire, afin que le fichier soit contigu. Les fichiers modifiés sont :

– FRAGS.C : ajout de l'implémentation de la méthode de défragmentation
– CACHE.C : ajout de la méthode FIND_N_CONTIG_ZONE renvoyant le premier bit d'identification d'une suite de zone suffisament grande permettant de placer le fichier de manière contigue. Elle renvoie la constante NO_BIT si aucun espace n'a été trouvé.

Choix d'implémentation

Comme MINIX est un système d'exploitation mono-utilisateur, nous avons émis l'hypothèse qu'aucun autre processus ne va essayer d'accéder au fichier en cours de défragmentation. Voici notre manière de gérer le problème posé :

– Récupération de l'INODE et comptage du nombre de fragments similaire à NFRAG. S'il n'y a qu'un seul fragment, l'appel se termine, la défragmentation est inutile.
– On calcule le nombre de zones qu'occupe le fichier. Attention, lors de la division de la taille du fichier par la taille d'une zone, s'il y a un reste, il faut incrémenter le nombre de zone pour avoir de la place pour tout le fichier.
– On exécute ensuite une méthode permettant de trouver l'index du début d'une suite de zone permettant de placer le fichier de manière contigue. Cette méthode est implémentée dans le fichier CACHE.C aux côtés des fonctions d'allocation de zone. Nommée FIND_N_CONTIG_ZONE, elle prend en argument la partition à utiliser, ainsi que

le nombre de zones libres consécutives à trouver. On démarre notre recherche au premier bit libre de la ZMAP, connu grâce au super bloc. Après quelques calculs de taille des différents index et éléments de la ZMAP, on charge le bloc contenant le premier bit libre. En convertissant les morceaux du bloc en binaire, on peut faire des tests bit à bit pour savoir si les zones sont libres. Une fois chaque morceau du bloc testé, on replace le bloc et on charge le suivant jusqu'à trouver une zone correspondant à celle demandée ou atteindre la fin de la ZMAP. Le premier bit de la zone trouvée est renvoyé, la constante NO_BIT est utilisée si l'espace libre n'a pas été trouvé.

- L'appel système se termine si aucune zone n'a été trouvée. Dans le cas inverse, la méthode alloue tous les bits nécessaires au placement du fichier (dans la ZMAP).
- L'objectif suivant est de recopier le fichier dans la zone trouvée. On va donc copier bloc par bloc le fichier en incrémentant un offset d'une longueur de bloc pour les blocs sources. Pour le bloc de destination, on incrémente son index de '1'.
- Il ne nous reste qu'à mettre à jour les pointeurs de zones dans l'INODE du fichier. On va d'abord libérer les zones mémoires utilisées précédemment et ensuite relier l'INODE aux nouvelles zones grâce à la méthode WRITE_MAP

L'appel système affiche le statut de l'opération avant et après la défragmentation. Il renvoie le nombre de fragments (normalement '1' si l'opération s'est déroulée complètement) ou le code d'erreur, comme la fonction NFRAG.

4 Stratégie de tests

Nous avons constitué un MAKEFILE complet pour les tests à effectuer. D'abord, on crée une nouvelle partition MFS et on monte cette partition. Ensuite, On crée cinq fichiers de $4kB$. On retire le premier et le quatrième afin de créer deux trous dans la mémoire. Après, on écrit un fichier de $16kB$, celui-ci sera placé d'abord dans les trous créés et la suite sera mise après le cinquième fichier. On obtient donc trois fragments. En utilisant le programme de test ("TEST.C"), les résultats sont corrects. Le nombre de fragments est de 3 avant défragmentation et de 1 après. La commande diff n'affiche aucune différence et le programme de scan du système de fichier affiche un nombre d'inodes et de zones libres égaux aux valeurs précédant l'exécution du test. Nous testons également l'exécution sur un fichier non-existant et sur un répertoire, les erreurs renvoyées sont correctes.

Les commandes make à utiliser pour lancer les tests sont :

- initmfs : initialise le système de fichier.
- mountdisk : monte la partition créée avec le path /MOUNT/DISK.
- createfiles : crée les fichiers nécessaires aux tests en fragmentant le fichier 6.
- test : compile le programme de test.
- runnfrags : effectue le test du calcul du nombre de fragments.
- rundefrag : effectue le test de défragmentation.
- checkfs : lance le programme de vérification du système de fichier.
- clean : efface les fichiers créés et démonte la partition.

L'utilisation de la commande MAKE effectue tous les tests à la suite.

5 Améliorations

Nous avons pensé à plusieurs améliorations à réaliser si nous disposions de plus temps :

- D'abord, les deux appels systèmes exécutent des fonctions forts similaires. Il serait dès lors possible de ne créer qu'un appel système, pour ne pas surcharger l'OS. En ajoutant un argument dans le message, la fonction se terminerait avant la défragmentation si l'utilisateur ne demande que le nombre de fragments.
- Ensuite, nos hypothèses sont valides pour un système mono-utilisateur simple. Si des mises à jour de MINIX ont lieu, un système multi-utilisateurs pourrait rendre nos fonctions obsolètes. Nous avons cependant veillé à utiliser des méthodes d'allocation qui ne forcent pas l'allocation d'une zone. Il s'agit d'une réutilisation de fonctions utilisées par le système d'exploitation. Si un autre processus veut changer l'attribution des zones, notre fonction ne va pas crasher. Cependant, il est possible que la défragmentation soit moins performante et laisse deux ou plusieurs fragments.
- Enfin, la défragmentation fonctionne ici pour un fichier. Son pouvoir est assez limité car elle ne cherche même pas à réutiliser les zones allouées au fichier à défragmenter. De plus, il serait également possible d'augmenter le nombre de fragment même après défragmentation. Ici, dès qu'il n'y a pas moyen de mettre le fichier en un fragment, la fonction s'arrête. Il serait sans doute plus performant d'essayer d'au moins diminuer le nombre de fragments.

On peut cependant affirmer que malgré quelques comportements à risque, le programme est fiable.