INGI1113 : Système d'exploitation

Projet MINIX

Défragmentation de fichiers

Groupe 14

NUTTIN Vincent 5772-05-00
LAMOULINE Laurent 3597-05-00

13 mai 2009

Table des matières

1 Introduction

Lors de ce projet, nous avons implémenté deux appels système destinés dans un premier temps à vérifier si un fichier est fragmenté, ce qui siginfie que les blocs composant le fichier sont non contigüs, dans ce cas le second appel système aura pour fonction de défragmenter ce dernier. Ces deux appels système ont respectivement comme signature

int nfrags(**char** *file) et **int** defrag(**char** *file)

Tout d'abord, il est important de signaler que certains types de fichiers ne pourront être défragmentés, comme par exemple les répertoires, les fichiers spéciaux (ex. les fichiers d'accès à des périphériques dans */dev*), ... Les fichiers ouverts ne pourront pas non plus être défragmentés sous peine de rendre le système de fichiers dans un état incohérent.
Parallèlement à ces appels système, il nous a parru intéressant d'implémenter une fonction qui imprime la "zone map" afin de d'avoir une représentation "graphique" de l'état de cette dernière, nous avons nommé cette fonction *printmap*. Cette fonction imprime un zéro là où la zone mémoire correspondante est libre et un 1 si la zone mémoire correspondante est réservée.

Nous allons tout d'abord expliquer la façon dont nous avons intégré les appels système à MINIX 3, c'est-à-dire décrire les fichiers que nous avons modifé et ajouté afin de faire reconnaitre nos appels par le système de fichier. Ensuite, nous expliquerons la façon dont la fonction *nfrags* procède afin de déterminer si un fichier est fragmenté ou non. Nous poursuivrons alors par la description de la manière dont *defrag* procède pour défragmenter un fichier pour terminer par une brève présentation de la fonction que nous avons créé pour afficher la carte des zonnes mémoire.

2 Intégration des appels système au système de fichiers

Comme c'était le cas dans le pré-projet, il a tout d'abord été nécessaire de faire reconnaitre nos appels système par le système MINIX. Contrairement au pré-projet, il s'agit ici d'intégrer les appels système dans le système de fichier et non plus dans le gestionnaire des processus. La métodologie générale est similaire, nous avons tout d'abord commencé par intégrer le prototype de nos appels système dans les fichiers *include/sys/nfrags.h* et *include/sys/defrag.h* afin que ceux-ci soient reconnu. Pa la suite, nous avons du attribuer un numéro à nos appels système, pour cela nous avons du modifier le fichier *callnr.h* dans lequel *nfrags* porte le numéro 69 et *defrag* le 70. Ensuite, nous avons ajouté nos appels système au tableau réprésenté dans les fichiers *table.c* dans le gestionnaire des processus et dans le système de fichier aux indexes correspondant aux numéros attribués dans le fichier *callnr.h* afin de faire correspondre les numéros des deux appels système aux routines qui les réalisent.
Nous avons ensuite implémenté les appels système *nfrags* et *defrag* proprement dit et intégré les fichiers dans le répertoire *servers/fs*, ces fonctions vont être décrites ci-dessous.
Les autres modifications à apporter ont été décrites lors du pré-projet, nous n'allons dès lors pas nous attarder sur la description de celles-ci.

3 Description de nfrags

Comme dit précédemment, la fonction de *nfrags* est de vérifier si un fichier est fragmenté ou non. Pour celà, nous avons utilisé la fonction *read_ map(rip, offset)* implémentée dans MINIX afin de trouver le numéro des blocs correspondant à la postion donnée dans le fichier associé à l'inode. Nous savons que la condition pour qu'un fichier soit fragmenté est que 2 de ses blocs ne soit pas consécutifs en mémoire. Afin de vérifier cette condition, nous avons parcouru le fichier en incrémentant *offset* au fur et à mesure (avec *offset* ayant une valeur nulle au départ) et nous avons associé le résultat de cet

appel à la variable `position` ; et le résultat de l'appel à *read_map* pour le même inode et un offset de OFFSET + BLOCK_SIZE à `next_position`, où BLOCK_SIZE correspond à la taille d'un bloc (dans le cas standard, la valeur de *block_size* est de 4096 bytes). Ce test est effectué jusqu'à ce que la fin du fichier soit atteinte.
Il est cependant important de faire une remarque concernant le nombre de fragments qui vont être trouvés. En fait, lorsque le fichier sera constitué de plus de 7 blocs, le système va placer un bloc indirect en mémoire pour référencer les blocs supplémentaires, s'il y en a, car un inode ne peut adresser directement que 7 blocs. Ce bloc va permettre de référencer 256 blocs mémoire si la taille de ces derniers est de 1 KB ou 1024 si la taille de ceux-ci est de 4 KB et que les adresses sont codées sur 32 bits.
On obtient les schémas de mémoire suivants lorsque les fichiers font respectivement 8 et 9 blocs :

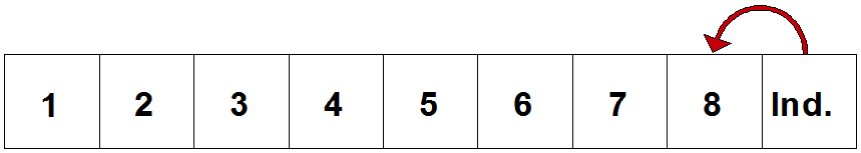

FIGURE 1 – Fichier de 8 blocs

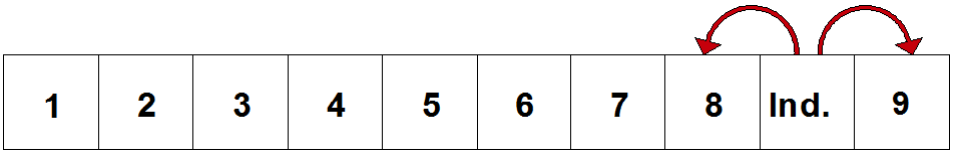

FIGURE 2 – Fichier de 9 blocs

Le bloc indirect n'étant pas compté comme faisant partie du fichier, le fichier est donc considéré comme fragmenté car la différence entre `position` (correspondant au bloc numéro 8) et `next_postion` (correspondant au bloc numéro 9) est supérieure à 1 alors que le fichier n'est pas réellement fragmenté. Nous avons décidé de tenir compte de ce phénomène dans notre implémentation, le fichier ne sera donc pas défragmenté (voir *servers/fs/nfrags.c*).
Les inodes peuvent également référencer des blocs en mémoire via un bloc doublement indirect lorsqu'un fichier est composé de plus de 263 blocs (256 + 7), si la taille des blocs et de 1 KB, ou 1031 blocs (1024 + 7) pour des blocs de 4 KB. Ce dernier fait référence à 256 blocs simplement indirects dans le cas de blocs de 1 KB ou 1024 blocs simplement indirects, qui font eux même références à 256 ou 1024 blocs mémoires respectivement. Nous n'avons pas tenu compte de ces blocs doublement indirects dans notre implémentation car cela compliquait notre code alors que dans le cas de blocs de 4 KB et d'adresses codées sur 32 bits (configuration de base de MINIX), cela correspondrait à un fichier de plus de 257 MB à défragmenter, ce qui n'est pas très courant mais il n'empêche que le fichier serait tout de même correctement mais inutilement défragmenté suite à un appel à *defrag*. Les blocs simplement et doublement indirects sont donc considérés comme des blocs faisant partie d'un autre fichier.

4 Description de printmap

La fonction *printmap* a la signature suivante

int printmap(**struct super_block** *sp) ;

Elle nous permet d'afficher une représentation "graphique" de la "zone map", c'est-à-dire une représentation de chacun des bits spécifiant si une zone mémoire est utilisée par un fichier ou non. Les zones vides sont représentées par un "0" tandis que les zones occupées sont représentées par un "1". Grâce à cela, il est possible de visualiser la disposition d'un fichier avant et après défragmentation. On peut voir un exemple de map imprimée ci-dessous :

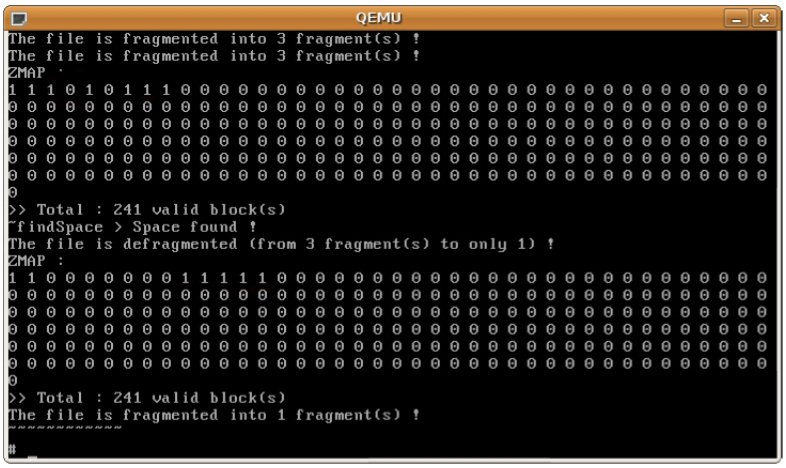

FIGURE 3 – Exemple de map

Sur la figure représentée ci-dessus, on constate qu'il y a 2 bit en début de map qui sont à "1", suite à de nombreux tests, nous avons constaté que ces deux bits étaient toujours réservé. Nous supposons donc que les blocs correspondant en mémoire sont des blocs réservé par le système car ils ne correspondent à aucun fichier créé précédement.

5 Fonctions auxiliaires de *defrag*

Afin d'alléger quelque peu *defrag*, nous avons implémenté deux fonctions auxiliaires que nous allons détailler ci-dessous. Ces dernières sont respectivement destinées à trouver une série de blocs contigüs afin d'y déplacer le fichier à défragmenter, la seconde permet de réaliser la copie des blocs du fichiers vers les nouveaux blocs déterminé par la fonction *findSpace*.

5.1 Fonction auxiliaire *findSpace*

La fontion à la signature suivante

bit_t findSpace(**size_t** size, **struct super_block** *sp, **int** blocks_per_zone, **int** nb_blocks) ;

Cette fonction permet de trouver le nombre de blocs mémoire contigüs suffisant pour y placer le fichier dont on donne la taille en argument. Une fois l'espace trouvé, la fonction renvoit le numéro du premier bloc vide trouvé ou un code d'erreur s'il n'y a pas assez d'espace. Cela permettra ensuite de copier les blocs du fichier à défragmenter à partir du numéro de bloc renvoyé grâce à la fonction auxiliaire que nous allons détailler ci-dessous.

5.2 Fonction auxiliaire *make_ cpy*

Cette fonction peut être vue comme une succession de trois étapes. Tout d'abord, la copie des blocs de données vers le nouvel espace grâce à l'utilisation judicieuse de *memcpy*. Ensuite, la mise à jour de l'inode pour référencer les nouvelles zones sur lesquelles se trouvent les données via la fonction *write_ map*. Enfin, la suppression dans la "zone map" des blocs précédemment utilisés par le fichier (avant défragmentation).

6 Description de defrag

Si la fonction *findSpace* trouve suffisament de blocs contigüs en mémoire pour y placer le fichier à défragmenter, les blocs de celui-ci sont alors déplacé de façon à ce qu'ils soient contigüs. Dans le cas où la fonction *findSpace* ne trouve pas d'espace suffisant que pour déplacer le fichier, il n'est alors pas défragmenté et le programme se ferme après avoir annoncé le problème à l'utilisateur.
Les figures ci-dessous affichent une exécution du programme pour un fichier de 5 blocs et de 20 blocs respectivement :

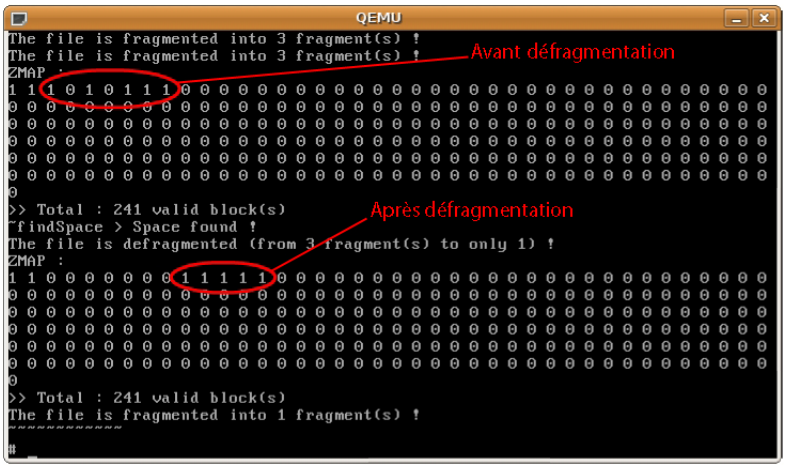

FIGURE 4 – Défragmentation d'un fichier de 5 blocs

Comme cela a été expliqué dans la section 3, le fichier de 5 blocs ne nécessite pas de bloc indirect car les 7 zones d'adressage directes de l'inode suffisent pour référencer entièrement les blocs du fichier. Dans la figure ci-dessus, on remarque que le fichier est fragmenté avant l'exécution de *defrag*, cette configuration est obtenue par ajout et suppression de fichiers. La fonction *printmap* confirme bien nos attentes car elle imprime des "0" ,correspondant aux fichiers qui ont été supprimés, entre les "1" représentant les blocs du fichier (cfr. Zone rouge sur la figure 4). Suite à la défragmentation, on remarque que le fichier est effectivement déplacé et que les blocs sont effectivement contigüs. Le premier bloc du fichier, correspondant au premier "1" de la zone entourée, est le premier bloc qui était libre avant défragmentation. Le résultat obtenu est bien celui qui était attendu.

Nous allons maitenant analyser le cas où une zone indirecte doit être créée pour adresser les 13 derniers blocs d'un fichier composé de 20 blocs. La figure 5 illustre le résultat obtenu grâce à la fonction *printmap*.

Dans un premier temps, on voit que le fichier est bien fragmenté car les blocs sont séparés par des blocs vides. Après défragmentation, le premier bloc du fichier est correctement placé dans le premier bloc qui était libre avant défragmentation et tous les blocs du fichier sont placé de manière consécutive en mémoire. Cepedant, on remarque qu'il reste un bit dont la valeur est "1" dans la série de "0" qui

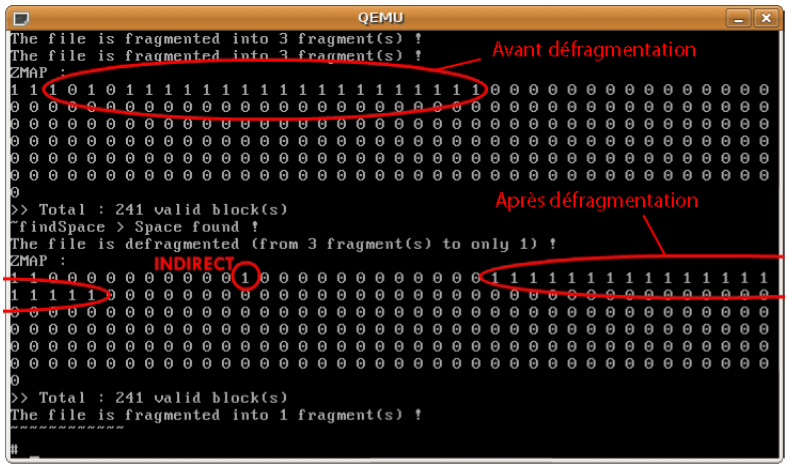

FIGURE 5 – Défragmentation d'un fichier de 20 blocs

étaient à "1" avant défragmentation. Ce bloc correspond en fait au bloc indirect de l'inode destiné à adresser les 13 derniers blocs du fichier.

7 Conclusion

Lors de ce projet, nous avons été ammené à manipuler des structures de données auxquelles nous n'avions jamais été confronté de manière pratique, qui plus est à un niveau relativement bas. Nous avons également dû nous immerger et interpréter un code beaucoup plus important que ceux que nous avons rencontrés jusqu'ici. Nous pensons avoir répondu de la manière la plus complète possible aux attentes de ce projet.