INGI1113 : Rapport
Projet 2: Minix - Défragmentation de fichiers

Cristian Voicu, *SINF13BA,*
Nicolas Varotto, *SINF13BA*

CONTENTS

I. ARCHITECTURE DU PROGRAMME

Cette section contient la description et les explications de notre choix d'architecture.

A. Analyse du problème

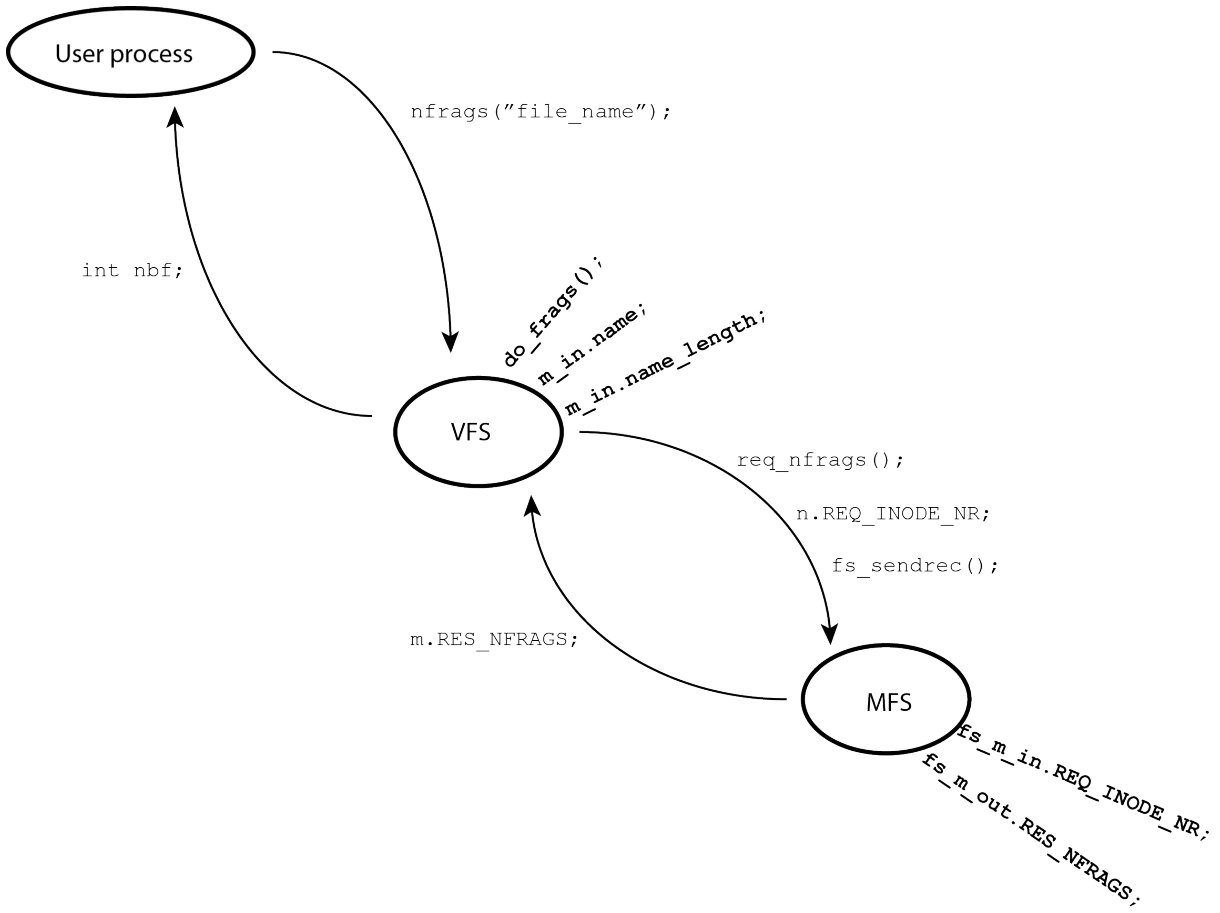
Après avoir lu attentivement l'énnoncé du projet, nous avons chercher à identifier les principales difficultés auxquelles nous auront à faire face. Nous sommes parvenus à identifier 4 difficultés majeures :

1) Comprendre le fonctionnement de Minix dans son ensemble, dans sa globalité.
2) Comprendre le système VFS et sa communication avec le MFS.
3) Connaître et comprendre les différents fichiers à modifier lors de l'ajout d'un appel système, comprendre comment se passe un appel système.
4) Comprendre comment fonctionne le traitement des blocs mémoires et comment faire des opérations sur ces blocs.

Avant d'écrire notre code, nous nous sommes fixés comme objectif de maîtriser un maximum ces 4 points.

B. Protocole de communication VFS-MFS

Le premier point pratique de notre projet était d'implémenter un protocole de communication entre d'une part le processus appelé par l'utilisateur (`nfrag` ou `defrag`) et le *virtual file system* (VFS) et, d'autre part, entre ce même VFS et le *minix file system* (MFS). Nous allons illustrer cela par un schéma que nous commenterons par la suite.



Partons de l'*user process* :

1) L'user process va appeler la fonction `nfrags` ou `defrag` qui va arriver dans le VFS.
2) Le VFS va saisir le path passer en paramêtre par la fonction provenant de l'user process et va récupérer le *vnode* associé au fichier référencé par ce path.
3) Le VFS va envoyé une requête au MFS. Cette requête contient le message et le numéro de l'*inode* associé au vnode.
4) Le MFS va s'occuper de récupèrer la structure de l'inode à partir du numéro transmis par le VFS. Une fois la structure récupérée, il fait les opérations nécessaires (`nfrags` ou `defrag`).
5) MFS envoit le résultat au VFS. Ce résultat est le nombre de fragments du fichier.
6) Le VFS renvoit à son tour ce numéro à l'user process.

C. Fichiers modifiés

Comme mentionné dans l'analyse du problème, un point important était de connaître les différents fichiers à modifier lors de l'ajout d'un appel système. Dans notre cas, voici la liste de ces fichiers :

```
include/defrag.h          include/minix/callnr.h    include/minix/vfsif.h
lib/libc/other/_defrag.c  lib/libc/other/_nfrags.c  lib/libc/syscall/defrag.S
lib/libc/other/nfrags.S   servers/mfs/Makefile      servers/mfs/defrag.c
servers/mfs/proto.h       servers/mfs/table.c       servers/vfs/Makefile
servers/vfs/defrag.c      servers/vfs/proto.c       servers/vfs/request.c
servers/vfs/table.c       lib/libvtreefs/table.c    servers/hgfs/table.c
servers/pm/table.c
```

D. Choix d'implémentation

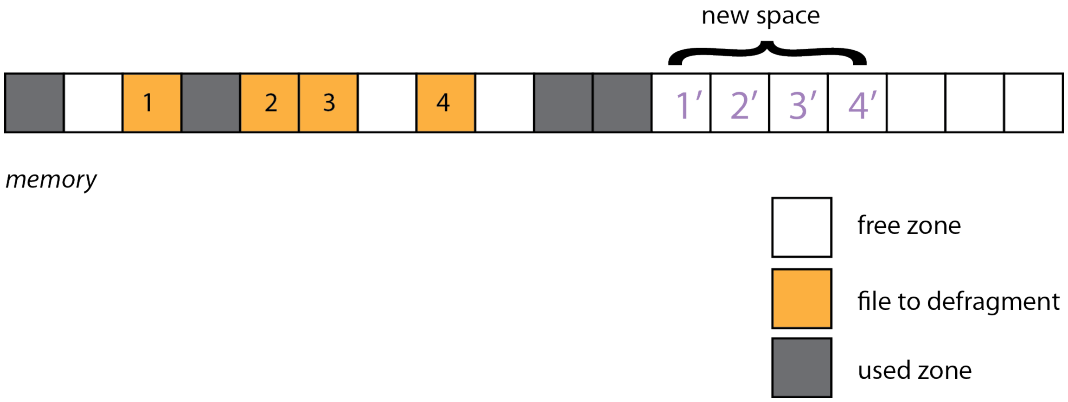
Notre choix d'implémentation se base sur l'exemple de l'appel système que nous avons reçu dans la présentation du projet (`stat()`). Au niveau de la communication entre le VFS et le MFS, nous avons opté pour deux requêtes différentes plutôt qu'une seule requête avec un flag. En effet, nous avons tenu a bien distinguer les deux fonctions *nfrags* et *defrag*.

Au niveau de la communication du résultat entre le MFS et le VFS, nous avons décidé d'utiliser le modèle de la fonction `fs_create()`, qui se trouve dans le fichier `open.c`. Nous avons ainsi suivi la procédure de passage d'un entier comme dans la fonction `req_create()`.

E. Traitement de la mémoire

L'appel de la fonction `defrag` nécessite une modification des blocs mémoire. Comment cela se passe-t-il? Il y a 3 étapes:

- La première étape consiste à trouver une zone libre dans la mémoire suffisamment grande pour pouvoir accueillir le fichier défragmenté.
- La seconde consiste à déplacer les blocs mémoires fragmenté dans le nouvel espace alloué.



- La troisième étape consiste à changer les référence des zones du fichier dans l'inode.

II. TESTING

Nous n'avons pas pu consacrer autant de temps pour la phase de testing que lors du projet précédent. Nous avons cependant eu le temps d'élaborer une stratégie de test ainsi que d'en vérifier leur fiabilité.

A. Stratégie de tests

Notre stratégie de tests porte sur deux points :

1) la transmission des messages et des requêtes
2) la bonne réussite de la défragmentation

Bien entendu, la fiabilité des appels systèmes est impliquée par ces deux points. Tant que nous n'étions pas assurés que la transmission des requêtes et des messages déroulait correctement, nous ne passions pas à la phase suivante qui consistait à implémenter le processus de défragmentation au niveau du MFS.

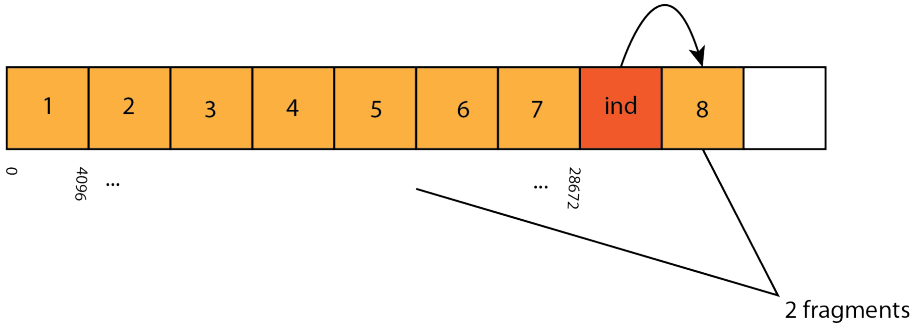
B. Fiabilité des tests

Au niveau du testing de la transmission de message et des requêtes, nous nous sommes basé sur le schéma de la communication MFS-VFS présenté plus haut. Nous avons lancé un appel de fonction defrag à partir de l'user process et, à chaque fois qu'un message atteignait le VFS ou le MFS, l'entité faisait un `printf`, ce qui garantissait le fait que le message ou la requête avait bien atteint sa cible.

Pour le testing de la défragmentation, nous avons appelé *nfrags* avant et après l'opération, et nous avons vérifier que le deuxième appel renvoyait bien chaque fois 1 pour les fichiers inférieurs à $28KB$, 2 pour les fichiers compris entre 28KB et 4MB, etc.

Néanmoins, il se pouvait que ce soit nfrags qui soit défectueux. Nous avons donc testé nfrags au moyen de cette procédure :

1) On créé un fichier d'environ 27 KB. Cette taille garanti que les 7 premiers zones de l'inode (les zones directes) sont occupées. En effet, chaque zone fait 4KB.
2) On défragmente, afin d'être sûr que le nombre de fragment soit égal à 1.
3) On fait un premier test de nfrags et on vérifie qu'il est égal à 1.
4) On augment la taille du fichier d'environ 5 KB. Dans ce cas, il y aura au moins une zone indirecte occupée. Du coup il y a 2 fragments dans le fichier (voir schéma).
5) On test si désormais nfrags renvoit bien 2.



III. BILAN

Ce projet fut une chouette expérience qui nous a permis de bien comprendre le fonctionnement d'un système d'exploitation et comment cela se passait en background lors d'un appel système.

A. Principaux problèmes rencontrés

Notre premier problème était que nous ne comprenions pas pourquoi il fallait passer par le VFS alors qu'il aurait été beaucoup plus simple de passer directement par le VFS.

Nous avons aussi éprouvé des difficultés au niveau de la notion de *bloc* et de *zone*. Il n'est pas évident de savoir distinguer ces notions. Surtout lorsqu'on se réfère au livre car ces notions sont confondues.

Nous avons eu des difficultés au niveau de la définition d'un fichier fragmenté. Dans notre programme, ce genre de fichier n'est pas considéré comme fragmenté :

Un autre problème fut rencontré lors de la recherche d'une zone libre suffisament grande pour le fichier défragmenté. En effet, lors de cette recherche nous décrémentons une variable à chaque zone libre contigüe rencontrée. Le problème, c'est lorsque nous rencontrons une zone non-libre et que la variable n'est pas à zéro, il faut alors désalouer les zones précédemment allouées. Nous avons donc du créer une méthode, `free_all_zone`, qui désalloue cette mémoire.

Enfin la dernière difficulté majeure rencontrée fut le déplacement des blocs mémoires. Au départ, nous ne passions pas par la cache, nous avons mis un certain temps avant de nous rendre compte de cette erreur.

B. Améliorations possibles

Notre projet gère la défragmentation au niveau des zones directes et indirectes? Nous n'avons cependant pas pris en compte les zones doublement indirectes, c'est-à-dire les fichiers de taille supérieure à 4MB. Une amélioration possible serait donc la gestion de ces fichiers.

Nous aurions également pu faire un affichage de l'état de la map[1]. Mais nous nous sommes rendu compte que c'était plus compliqué qu'il n'y paraissait: la map était tellement grande que uniquement la fin de la map apparaissait en console. Il fallait donc trouvé une commande capable de faire défiler cette map[2]

[1] comme dans les vieux défragmenteurs sous Windows
[2] comme pour un `man ls` sous Linux, par exemple.