Projet 2: Minix - Défragmentation de fichiers [1]

Ludovic GUSTIN , Guillaume SIMONS

I. INTRODUCTION

Minix est un OS minimaliste basé sur UNIX, et fondé par le Pr. A. Tanenbaum en 1987. Cet O.S., contrairement à UNIX et à Linux, repose sur une architecture en micro-kernel, réduit à ses fonctionnalités primaires (mécanismes d'interruption matériels, Scheduler, Appels systèmes et message passing, ...). Ce microkernel est entouré d'un nuage de processus servers et drivers s'exécutant en parallèle avec le kernel dans le kernel space. Dans le cadre de ce projet, nous nous sommes intéressés aux processus servers gérant l'ensemble du filesystem sous MINIX V3, le vfs et le mfs, et plus particulièrement sur l'implémentation d'appels système de défragmentation d'un fichier à travers ces servers.

Nous avons donc implémenté deux appels systèmes, nfrags() et defrag(), le 1er se chargeant de calculer le nombre de fragments non-contigus d'un fichier régulier sur le système de fichier MINIX, et le second se proposant de grouper et recopier ces fragments dans une zone de mémoire continue si elle existe.

II. ARCHITECTURE

Rappelons que le filesystem de MINIX, à l'instar de UNIX, est fondé sur des structures appelées i-nodes qui répertorient entre-autre les différentes zones de la mémoire disque hébergeant les fragments du fichier. Ces zones peuvent rapidement ne pas apparaitre contigües sur le support physique ; en effet le système d'exploitation se contente de chercher l'espace libre là où il peut rapidement le trouver, sans optimiser la question de la localité spatiale des blocs écrits.

A. Intégration des syscall et routage des messages

Avant toute implémentation pratique, il convient d'abord d'intégrer proprement nos appels système dans la structure de MINIX, c-à-d propager l'existence de nos appels à travers tous les fichiers de définitions , de mapping et de routage d'appels système. De plus, il faut gèrer tous les échanges de messages entre l'userspace, les librairies, le VFS et le MFS. Contrairement à la phase de pré-projet, nos appels ne sont plus introduits dans le pm ou processus manager, mais bien dans les mfs (*minix file system* et vfs (*virtual file system)*. Ci dessous, nous retrouvons en synthèse les étapes principales d'intégration d'un nouvel appel système , l'ordre n'a évidemment aucune influence.

1) Intégration du prototype du pointeur de fonction appel système dans l'userspace : /src/include/defrag.h (contient les 2 prototypes)
2) Attribution d'un numéro du vecteur d'appel sytème (vers le vfs) : /src/include/minix/callnr.h 114 et 115 ont été retenus
3) Prototype des fonctions do_nfrags et de do_defrag (vfs)) dans le kernel : /src/servers/vfs/proto.h :
4) Mapping du numéro callnr.h et de la méthode implémentée (pointeur de fonction) appelée dans le kernelspace, dans les tables d'appels système à l'intérieur des servers concernés (dans le *call_vec[]) : /src/servers/vfs : do_nfrags et do_defrag pour 114 et 115 respectivement
5) Implémentation du code du pointeur de fonction dans le kernel de nfrags et defrag dans /servers/vfs/defrag.c et nfrags.c, qui va retransmettre le message au mfs

6) Définition d'un numéro-type de requête, interface de communication entre le `vfs` et `mfs`[2] `/src/include/minix/vfsif.h` : `REQ_NFRAGS et REQ_DEFRAG`; 33 et 34 ont été retenus
7) Prototype des fonctions `fs_nfrags` et de `fs_defrag` (`mfs`)) dans le `kernel` : `/src/servers/mfs/proto.h` :
8) Mapping du numéro de requête avec de la méthode implémentée (pointeur de fonction) dans le `mfs`, dans les tables d'appels système (`*fs_call_vec[]`) `/src/servers/pm ou iso9660fs ou pfs ou hgfs ou ext2 /table.c :no_sys` pour 33 et 34 (`no_sys` ne renvoit qu'une erreur par convention) `/src/servers/mfs` : `fs_nfrags` et `fs_defrag` pour 33 et 34 respectivement
9) création et envoi du message relai entre le `vfs` et le `mfs` : `/src/servers/request.c` `req_nfrags` et `req_defrag`
10) Implémentation du code utile, du pointeur de fonction dans le `kernel` de `nfrags` et `defrag` dans le `mfs` : `/servers/mfs/defrag.c et nfrags.c` : `fs_nfrags()`et `fs_defrag()` (+ `alloc_bits_zone`)

L'on observe que la partie du code utile ne réside essentiellement que dans le dernier point ; le reste n'est que propagation de messages entre les divers processus.

B. L'appel système `nfrags()`

```
int nfrags( char* file);
```

La fonction prend en argument une chaîne de charactère représentant le chemin absolu ou relatif dans le `filesystem` MINIX, et doit pointer sur un fichier régulier. En effet, les répertoires, les fichiers spéciaux (accès aux device dans `/dev`, ...), les fichiers de génération de `char` (`/dev/zero` et `/dev/rand`) ne peuvent être défragmenté.

Afin de calculer le nombre effectif de fragment, nous avons recouru à la méthode **`block_t`** `read_map(rip, position)` issue de `/servers/mfs/read.c`, laquelle retourne le numéro de bloc où sont stockés les binaires à partir d'un l'offset absulu, en byte, dans le fichier (référencé par son i-node). Nous avons itéré la méthode par incrément d'un `block size` (information récupérée dans le superblock), sur toute la taille du fichier. Dans notre cas, cette valeur correspond à `4Kb`. Ainsi lorsque le numéro de bloc à l'itération suivante ne correpond pas exactement à un incrément d'une unité, nous avons un nouveau fragment.

Rappelons de prime abord la structure particulière des blocs directs et indirects en mémoire dans le cas idéal (contigu) :

Les 7 premiers blocs du fichiers sont des blocs à accès direct, après ces derniers vient le premier bloc de données indirectement référencé, puis le bloc des pointeurs indirects (le système écrit d'abord ce bloc puis crées le bloc d'indirection) , suivit enfin des autres blocs pointés par cette indirection simple. Un tel bloc peut référencer jusqu'à $\frac{4[kb]}{32[b]} = 1024$ blocs, soit $1024 * 4k = 4096k$, si l'on considère que la taille d'un bloc est de $4k$, et que les blocs sont addressés suivant $32bit$. Ensuite se trouve un autre bloc intermédiaire, destiné à l'indirection double. Ce bloc est à son tour suivit par, pour chaque indirection double, un bloc simplement indirect et ses blocs pointés et ainsi de suite. A nouveau, si le bloc à indirection double pointe vers 1024 blocs simplement indirect, qui à leur tour point vers 1024 blocs, soit $4Gb$ addressable de cette façon.

Dans notre cas, nous négligeons l'impact des blocs à indirection double, considérant la défragmentation de fichier de petite taille. Lorsqu'un tel bloc est rencontré, nous passons à un fragment supplémentaire. Un fichier de taille supérieure à 8 blocs sera toujours considérés comme fragmenté en au moins 2 part selon l'implémentation expliquée ci-dessus, si nous ne traitions pas l'existence du bloc simplement indirect. C'est pourquoi nous avons décidé de faire un test conditionnel pour

détecter ce bloc indirect lors du comptage des fragments, et d'ignorer cette pseudo-fragmentation. Cependant, par simplification, nous avons négligé l'existence du bloc doublement indirect.

C. *Méthode auxilliaire : `alloc_bits_zone()`*

```
PRIVATE bit_t alloc_bits_zone(sp, nb_zone)
```

Cette fonction est dédiée à la recherche d'un nombre de zone contigus spécifiés en argument, ou renvoit une erreure `NO_BIT` si un tel espace ne peut être réservés. Elle retourne le numéro de bit dans la `zonemap` correspondant à la 1ère zone libre.

D. *L'appel système `defrag()`*

```
int defrag( char* file);
```

La fonction centrale de ce projet MINIX peut s'articuler en quatres phases principales (pour plus de détails, consulter les `sources`.

a) Appel de `nfrags()`: Tout d'abord il convient de vérifier si le fichier est de type régulier d'une part, et d'autre part déterminer s'il faut ou non le défragmenter, soit l'existence de plus d'un fragment, ce que permet un appel à `nfrags()`. De plus, contrairement à `nfrags()`, cet appel se doit de vérifier si le fichier n'est pas ouvert par un autre processus, sous peine de mettre le `filesystem` dans un état incohérent.

b) Allocation d'une nouvelle zone de mémoire: La fonction privée `alloc_bits_zone()` est appelée pour trouver terre promise pour les fragments, soit une portion vierge du disque ; si une telle zone ne peut être trouver, c'est l'erreur assurée.

c) Copie bloc par bloc du fichier: Via la méthode `memcpy` (`/src/lib/ans`).

d) Update du `filesystem`: Dans cette ultime phase, nous remettons à jour les zones référencés par l'`i-node` du fichier en question, via la fonction `write_map()`. Parallèlement, nous libérons dans la `zone map` les bits correspondant aux fragments initiaux avec `free_zone()`

III. Stratégie de test

Pour recréer un fichier fragmenté sur un tout nouveau `filesystem`, nous avons commencer par générer deux fichiers via `dd` depuis `/dev/urandom` . Ensuite, grâce à la redirection `shell` (») et à la commande `cat`, nous pouvons redirigé le contenu du 2e fichier dans le premier. Ainsi si le `filesystem` a écrit les 2 fichiers séquentiellement, il a juste alloué la place nécessaire au 1er grâce à sa taille donnée. Mais si l'on fait croître ce fichier en répétant cette opération, l'OS va devoir trouver d'autres emplacements libres et par conséquents fragmente le fichier. 3 type de tests différents ont été ici appliquer pour valider nos implémentations, sur divers tailles de fichiers fragmentés :

A. *Tests de cohérence interne : `testTotal.c`*

Ce code permet d'automatiser la défragmentation d'un fichier en argument. Il procède à l'analyse du nombre de fragment :

- Si `nfrags()` a rencontrée une erreur (fichier non régulier entres autres) ou si le fichier ne présente pas de fragment, le programme s'arrête avec message d'état.
- Si la `defrag` a rencontré une erreur (fichier ouvert ailleurs ou autre), à nouveau arrêt + message d'état.
- Enfin `nfrag` est réappelé sur le fichier en question, pour assurer qu'on arrive bien à un seul fragment. Si l'on a plus d'un fragment, ou si `defrag` a renvoyé un nombre de fragment initial différent de la valeur calculée par le premier appel à nfrags, une erreur est à nouveau notifié. Il va sans dire que notre implémentation satisfait à ce test sans notification d'erreur.

B. Tests d'intégrité du FS et des fichiers défragmentés : `du, df, diff et fsck`

Avant toute chose nous effectuons une copie safe du fichier à défragmenter. Après exécution de `testTotal.c`, nous effectuons un `diff` entre la copie et le défragmenté. Ensuite `du` va contrôler si la taille de fichier est constante, et `df` procède de même pour la partition entière. Enfin nous exécutons `/sbin/fsck.mfs` pour révéler une quelquonque anomalie dans le `mfs`. Tous ces tests ont été parfaitement réussis sur nos fichiers de tests ; nous n'avons pas juger pertinent de rassembler tout ceci dans un script.

C. Tests visuels : `showZMAP()`

Nous avons décidé d'implémenter en toute fin un appel système additionnel qui permet d'afficher la `zone map` pour les fichiers de petites tailles (l'argument `REQ_COUNT` transmis dans le message appel système depuis `nb`, permet de limiter le nombre de bit à visualiser) afin de valider le bon fonctionnement de nos fonctions, surtout `nfrags()`. (cf. *II. A* pour l'intégration). L'on peut observer les deux premiers bits de la map toujours alloués à 1.

```
int showZmap( const char* path,int nb);
```

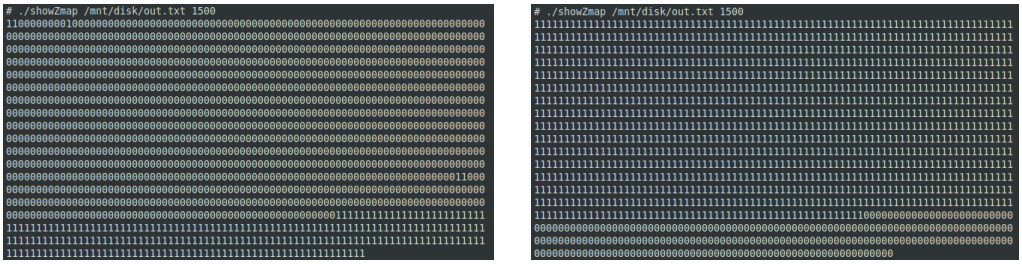

FIGURE 1. Ici l'on observe la présence du bloc indirect simple suivit du double (ligne 13) qui provoque une pseudo-fragmentation détectée par `nfrags()` ; la 2e figure est le fichier défragmenté

IV. PISTES D'AMÉLIORATION

Dans ce projet, nous n'avons presque pas tenu compte des blocs indirect et doublement indirect. Effectivement, nous ne prenons en compte que le bloc simplement indirect dans l'appel `nfrags`. En effet, dans un fichier nous ne considérons pas le premier bloc indirect comme un trou mais comme faisant partie de celui-ci. Mais afin d'effectuer une défragmentation idéale, il nous aurait fallu considérer en plus les bloc doublement indirect, et dupliquer ce raisonnement dans `defrag`. En effet,déplacer tous les blocs de donnée du fichier autre part tout en laissant les blocs indirect à leur précédente place forcera la tête de lecture du disque à naviguer entre les bloc de donnée et les bloc indirects, ce qui, au point de vue performance, n'est évidement pas fort optimal. Dans cet objectif, nous devrions placer les blocs indirect et doublement indirect judicieusement dans les blocs de donnée (7 blocs de donnée, un bloc indirect, 1024 blocs de donnée, 1 bloc doublement indirect et un bloc indirect, 1024 blocs de données, ...) comme le fait l'OS lorsqu'il a place libre dans le cas idéal (séquentiel). Par conséquent, cela requerrait, lors de la boucle de copie des bloc de donnée dans `defrag`, de vérifier chaque fois si l'on se trouvait à un emplacement du fichier où il est plus judicieux de placer un bloc indirect ou doublement indirect plutôt que le prochain bloc de donnée, et dans ce cas placé le bloc concerné suivi du prochain bloc de donnée.Cela n'étant pas demandé epxplicitement dans l'énoncé de notre projet, nous n'avons pas implémenté cette fonctionnalitée, mais néanmoins décrit la façon dont on procéderait.

V. CONCLUSION

Ce projet nous a permis d'en apprendre davantage sur le fonctionnement d'un système de fichier, et plus particulièrement ceux de la famille UNIX. De nombreux points ont été démystifiés par rapport à nos connaissance de bases, où l'on a appris à considérer tous ces mécanismes complexes de bas niveau comme une belle boîte noire. La propagation des appels système et l'envoi de messages de synchronisation entre les processus sytèmes, opérant aux côtés du microkernel, caractérise particulièrement MINIX, par rapport à UNIX et Linux notamment.