Défragmentation pour le système de fichier Minix

Groupe 22: Benoît Daccache, Raphaël Bauduin

I. Architecture

Nous avons décidé d'utiliser le plus possible les fonctions déjà présentes dans minix, ce qui nous a permis d'implémenter la solution en un nombre de lignes de code limité tout en s'assurant de la solidité de la solution, notament pour le traitement des nombreux cas spéciaux qui se fait par les fonction minix.

Comme nous ajoutons deux appels systèmes, nous modifions la valeur de NCALLS dans src/include/minix/callnr.h (113 au lieu de 115). Nos appels systèmes sont définis dans la libc grâce aux fichiers /src/include/defrag.h et /src/include/nfrags.h et nous modifions les fichiers Makefile en conséquence (inclusion de ces deux fichiers). Nous rajoutons nos deux appels systèmes à /src/include/minix/vfsif.h pour leur assigner à chacun un request number, et adaptons en conséquence la variable NREQS contenant le nombre de requêtes définies pour le VFS.

A. Libc

Au niveau de la libc, nous définissons une fonction pour chaque appel système, et ces fonctions construisent un message envoyé au process gérant la couche du virtual file system. Dans ce message est passé une rérérence vers l'adresse mémoire où doit être placé la valeur de retour (dans notre cas le nombre de fragments constituant le fichier).La valeur renvoyée par ce process est la valeur de retour de la fonction au niveau de la libc. Ces fonctions se trouvent dans les fichiers /src/lib/libc/other/_nfrags.c et /src/lib/libc/other/_defrag.c. Dans les fichiers /src/lib/libc/syscall/nfrags.S et /src/lib/libc/syscall/defrag.S nous configurons la routine à exécuter lors de l'appel système. Nous mentionnons ces fichiers dans le Makefile.inc gérant cette partie du code.

B. File System

Au niveau des file systems, nous modifions le fichier table.c pour définir quelle fonction doit être exécutée pour chaque appel système. Comme nous n'implémentons la défragmentation que pour mfs, nous mentionnons no_sys pour les autres systèmes de fichiers. no_sys est définie dans /src/servers/mfs/utility.c et imprime un message indiquant un appel système invalide, sans effectuer d'autre opération.

Pour le système de fichier mfs, les appels systèmes sont gérés par les fonctions do_nfrags et do_defrag dont les noms suivent la convention de nommage de fonction au niveau du virtual file system. Ces fonctions sont donc mentionnées dans le fichier table.c pour les numéros d'appel système correspondants et leurs prototypes sont placés dans proto.h. Ces fonctions au niveau du vfs se contentent de faire quelques vérifications (par exemple fichier introuvable), avant de transmettre la requête au système de fichier proprement dit par l'appel de req_nfrags ou req_defrag. Pour que la valeur de retour de l'appel à req_nfrags (ou req_defrag) soit transmis à la fonction appelante au niveau libc, les fonctions do_(nfrags|defrag) passent comme argument à req_nfrags/req_defrag l'adresse mémoire initialisée au niveau libc (et reçue dans le message) ainsi qu'un grant id y donnant accès (généré par la fonction cpf_grant_direct). Le résultat de l'appel sera placé dans cette adresse mémoire par la fonction au niveau fs grâce à la fonction sys_safecopyto.

L'implémentation des appels systèmes pour le système de fichiers mfs se trouve respectivement dans les fichiers /src/servers/mfs/nfrags.c et /src/servers/mfs/defrag.c. Le code partagé par ces deux fonctions est placé dans le fichier fragments.c situé dans le même répertoire.

1) Nfrags : Commençons par définir un fichier défragmenté. Un fichier est défragmenté si tous ses blocs de données se suivent dans l'ordre sur le disque, sans espaces libres. Les numéros des blocs du fichier défragmenté doivent donc être consécutifs. En particulier, les zones contenant les blocs référençant les zones indirectes et doublement indirectes ne peuvent s'intercaler entre les zone stockant les données du fichier.

Nous parlons en terme de bloc car la fonction que nous utiliserons retourne le numéro de bloc. Cette approche est la même que la taille de zone soit d'un bloc ou plus.

Pour déterminer si un fichier est fragmenté, nous parcourons donc le fichier et vérifions que les blocs traversés sont bien consécutifs. Si le numéro de bloc actuel est inférieur au numéro de bloc précédent, ou s'il est supérieur au numéro de bloc précédant+1, on a un nouveau fragment. Le numéro de bloc d'une position dans un fichier est obtenu par l'appel à la fonction read_map prenant comme argument l'inode en plus de la position dans le fichier. La position testée dans le fichier est à chaque fois incrémentée de la taille d'un bloc pour limiter le nombre de tests. Une fois le nombre de fragments déterminé, ce résultat est retourné à l'appelant grâce à la fonction sys_safecopyto comme expliquée ci-dessus.

Your name is with xyz Department...

2) Defrag: Tout comme au niveau vfs, nous avons suivi les conventions de nommage des fonctions et la fonction implémentant l'appel defrag s'appelle fs_defrag. Si le fichier n'est pas fragmenté, aucune opération n'est à effectuer. Si par contre le fichier est fragmenté, il faut commencer par trouver une région libre assez grande pour contenir toutes les zones data du fichier (les blocs utilisés pour stocker les zones indirectes et doublement indirectes peuvent être à un autre endroit, mais ne doivent en aucun cas se trouver au milieu des zones de données). La taille du fichier en bloc est premièrement déterminée. La recherche de cette région libre est effectuée par la fonction search_free_region, dont le code se base sur la fonction alloc_bit définie dans /src/servers/mfs/super.c. Alors que alloc_bit parcourt la bitmap jusqu'au premier bit à zéro pour trouver une zone libre, search_free_region continue à itérer sur la bitmap jusqu'à avoir trouvé une région libre assez grande, ou jusqu'à avoir déterminé qu'une telle zone n'est pas disponible. Comme search_free_region travaille avec la zone map, il faut convertir la taille du fichier comptée en bloc en taille comptée en zone. Cela se fait facilement par un shift sur le nombre de blocs d'un nombre s_log_zone_size se trouvant dans la structure du super bloc. search_free_region retourne comme valeur le premier bit de la région libre dans la zone map, NO_BIT sinon.

Si une région libre suffisamment grande a été trouvée, la défragmentation proprement dite a lieu. Tout d'abord, les zones où sera placé le fichier sont marquées comme utilisées dans la zone map. Nous utilisons pour cela une fonction alloc_this_bit qui se base sur la fonction alloc_bit de minix. Nous n'avons pas pu utiliser alloc_bit pour la raison que voici. Un des arguments de alloc_bit s'appelle origin, qui est un bit près duquel on désire utiliser un bit libre. Même si le bit correspondant à la valeur de origin passée en argument, il n'est pas sûr que l'on utilise ce bit. Voici un scénario qui le prouve. Imaginons que nous passions 29 comme valeur origin. alloc_bit va alors charger le mot contenant le bit 29 en mémoire, et va itérer sur les bits du mot jusqu'à tomber sur un bit à zéro. Mais cela veut dire qu'il utilisera le premier bit à zéro du mot, qui peut être avant le bit 29! En voici l'illustration: soit le mot contenant le bit 29 : 11011000. Le bit 29 est le deuxième 0 du mot et alloc bit réservera le premier bit du mot qui a la valeur 0, c'est-à-dire dans ce cas le bit 26. La fonction alloc_this_bit par contre réservera le bit 29. Ce problème se pose lors de l'allocation des premières zones de la région libre qui sera utilisée pour le fichier défragmenté.

Ensuite les blocs de données sont copiés vers leur nouvelle position. L'ancien bloc et le nouveau bloc sont chargés dans le buffer par un appel à get_block, les données de l'ancien bloc sont copiées (en mémoire) vers le nouveau bloc par un memcpy, le nouveau bloc est marqué comme dirty, et les blocs sont ensuite libérés par appels à put_block. Cette manière de faire nous permet d'exploiter au mieux les fonctions existantes de minix. Ensuite, nous libérons toutes les zones référencées par l'inode. Cela se fait en itérant sur le fichier (par pas de la taille d'un bloc) et pour chaque position en faisant un appel à write_map, avec comme dernier argument WMAP_FREE pour indiquer qu'il faut libérer la zone correspondant à la position dans le fichier également qui est passée en argument. write_map gère les zones directes et doublement indirectes. Une fois l'inode "nettoyé", nous le faisons référencer les nouvelles zones par une nouvelle boucle faisant à nouveau des appels à write_map, mais en passant à chaque fois la nouvelle zone correspondante (nous convertissons la position dans le fichier en numéro de bloc et en suite en numéro de zone). Nous marquons alors l'inode comme dirty avant de le libérer. Le nombre de fragment du fichier initial est alors retourné de la même manière que dans nfrags.

Il est à noter qu'il suffit de trouver une région libre ayant comme taille (en nombre de zones) au moins le nombre de zones data utilisée par le fichier. En effet, les blocs utilisés pour les zones indirectes et doublement indirectes n'ont pas d'impact du fait que nous libérons d'abord toutes les zones référencées par l'inode, avant de le faire référencer les nouvelles zones du fichier défragmenter. Cela nous assure qu'un bloc pour les zones indirectes sera toujours trouvable.

II. Stratégie de tests

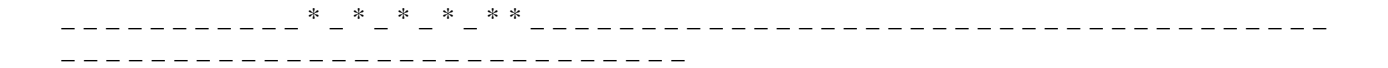
Nous avons créé une petite partition de 1Mb nous permettant facilement de la remplir de petits fichiers. En effaçant un fichier sur deux, de l'espace se libère, mais fragmenté sur toute la partition. En créant un fichier utilisant une partie de l'espace libre (pas toute!), on est sûr que ce fichier est fragmenté. En effaçant tous les petits fichiers restant, on libère de l'espace, dont une région assez grande pour contenir le fichier lorsqu'il sera défragmenté.

Utiliser une petite partition nous a également permis d'avoir des résultats utilisables d'une fonction print_map que nous avons écrite. Cette fonction, qui n'est pas incluse dans le code remis, parcourait la zone map et imprimait '*' pour un bit à 1 et '_' pour un bit à zéro. Pour une grande partition, le résultat est inutilisable, car la map est beaucoup trop grande, mais pour une partition de 1Mb, cette représentation nous permettait d'identifier les régions libres, et de vérifier que la défragmentation utilisait bien les zones attendues.

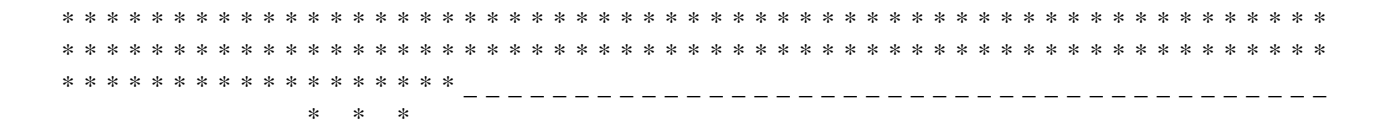
Pour pouvoir facilement répéter certains scénarii de tests, nous créions les partitions avec la configuration voulue, et prenions une copie des fichiers .cow pour pourvoir les réutiliser par après sans devoir recréer tous les fichiers. Nous avons effectué des tests avec des gros fichiers, des petits, de grands espaces libres, des espaces trop petits pour couvrir toutes les situations auxquelles nous avons pensé.

Voici un exemple de test avec les print_map correspondant. Nous commençons avec une partition contenant de grands espaces libres fragmentés:

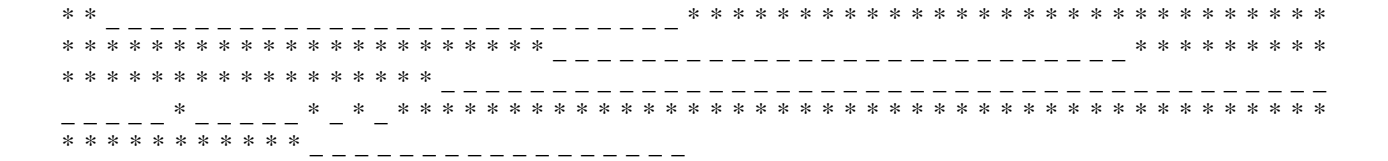
```
* * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * * * * * *
* * * * * * * * * * * * * * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```



Nous effaçons alors trois petits fichiers d'un bloc (les trois dernières étoiles ci-dessus) et créons un fichier de 52 blocs pour obtenir cette map:



Le fichier créé utilise maintenant les régions qui étaient libres au début du disque. Le fichier défragmenté n'a pas assez d'espace au milieu, et pour être défragmenté, devra être placé après les dernières zones occupées, ce qui est bien confirmé par la map ci-dessous:



Les erreurs qui peuvent se produire sont détectée le plus tôt possible dans la chaîne d'appel libc -> vfs -> mfs et un code d'erreur est retourné. Nous avons réutilisé le plus possible les codes d'erreur de minix (par exemple lors de la demande de défragmentation d'un fichier inexistant). D'autres erreurs sont propres à notre code, mais nous avons là aussi réutilisé les codes d'erreur de Minix. Voici une liste donnant les erreurs et leurs code d'erreur respectifs:

Table I
CODES D'ERREURS

Description de l'erreur	Code retourné	Niveau de détection
fichier en cours d'utilisation	EBUSY	VFS
type de fichier incorrect	EPERM	VFS
inode introuvable	EINVAL	MFS
espace insuffisant pour la défragmentation	ENOSPC	MFS

Voici les test que nous avons implémentés:

- défragmentation d'un fichier inexistant
- défragmentation d'un fichier non fragmenté
- espace insuffisant pour effectuer la défragmentation
- effacement d'un fichier après sa défragmentation pour vérifier avec df que l'espace est bien libéré (nécessaire pour valider que l'inode est bien libéré au niveau mfs)
- comparaison du contenu du fichier défragmenté avec une copie du fichier original avec l'outil diff

Notre solution passe tous les tests avec succès. Notre solution fait également la détection de fichiers ouverts, mais ce test n'est pas inclus dans notre script.

III. Améliorations possibles

Limitation: si un gros fichier est fragmenté sur la partition, mais est le seul fichier sur la partition et que l'espace libre est plus grand que la taille du fichier, il est théoriquement possible de défragmenter le fichier, mais pas selon notre implémentation. Il faudrait deux étapes:

1) regrouper les fragments du fichier
2) défragmenter en utilisant l'espace libéré à l'étape 1

Nous n'avons implémenté que la défragmentation.

Au lieu de définir alloc_this_bit, une option serait d'écrire new_alloc_bit pour qu'elle accepte les arguments de alloc_bit avec un argument supplémentaire forward_search_only de sorte que l'on puisse choisir de ne sélectionner que les bits à partir de l'origin passée en argument. La fonction alloc_bit ferait alors appel à cette fonction en passant false comme valeur du dernier argument, et notre code appellerait alors cette fonction avec pour argument true.

Le fait que nous découvrions Minix fait que notre code est très certainement améliorable. C'est ainsi que nous avons découvert à la fin du projet la fonction next_block, que nous aurions pu utiliser dès le début.