
Enseirb-Matmeca :
Filière Informatique

# IT202 : Projet Système
# Equipe DominOS

*Equipe 2037 :*
Omar Hayak
Gaëtan Combes
Hugo Dodelin
Florian Le Vern

*Encadrante :*
Nathalie Furmento

10 mai 2016

# Table des matières

# Introduction

L'objectif de ce projet était de réaliser un système de fichiers permettant d'organiser des fichiers comme des images ou de la musique à l'aide de tags, grâce à un répertoire contenant les fichiers à organiser et un fichier sauvegardant les tags appliqués à chaque fichier.

Il fallait également que ce système soit le plus performant possible, en maîtrisant la complexité de chaque opération, et le plus complet possible, grâce à l'ajout d'extensions permettant au système de gérer de nouvelles situations ou d'être permissif dans son utilisation.

# 1 Structures de données

## 1.1 Les Contraintes

La nature du système de fichier nous oblige à créer une base de données pour stocker les données à traiter. Il existe deux types de données : les noms des fichiers et leurs tags, un fichier peut avoir plusieurs tags et un tag peut être présent sur plusieurs fichiers.
La quantité des données varie d'une exécution à l'autre, donc il faut choisir une solution flexible en fonction de la taille des données. Ainsi, le langage de programmation imposé étant le `C`, il faut penser à la facilité d'intégration du code.

## 1.2 Les Choix

Afin de respecter les contraintes citées ci-dessus, nous avons décidé de stocker les informations dans deux tables de hachage. La première pour les fichiers et leurs tags et la deuxième pour les tags et leurs fichiers correspondants. Les informations concernant les tags (respectivement les fichiers pour la deuxième table) sont stockées dans des listes chaînées. Certes, il y a une certaine duplication d'informations, mais cela est tolérable vu la simplicité du modèle par rapport à une solution avec une seule table. Nous nous sommes basés sur les librairies `uthash` et `utlist` pour implémenter une interface qui facilite l'utilisation des tables de hashage et des listes chaînées. Les deux librairies sont faciles à utiliser et installer car elles proposent de manipuler les tables de hachages et les listes à travers des macros uniquement.

L'unité de base à stocker dans les tables est une structure à deux champs :
— le nom du fichier/tag.
— une liste chaînée de tags/fichiers.

# 2 Système de fichier `tagfs`

L'objectif du système de fichier `tagfs` est de pouvoir circuler dans un dossier contenant des images en fonction des tags qui leurs sont attribués. Il faut pouvoir facilement ajouter, modifier ou supprimer un tag associé à une image.

`fuse` permet de monter notre système de fichier personnel `tagfs` à partir d'un répertoire source (que nous appellerons 'images') dans un répertoire de destination (que nous appellerons 'mnt'). On lancera donc ce daemon en lançant à la racine du projet : `make && ./tagfs images mnt`

## 2.1 Prérequis

Avant de lancer le **main()** de `fuse`, notre **main()** cherche le fichier `.tags` dans le dossier à partir duquel l'utilisateur souhaite monter le système `tagfs` pour le parser. C'est un prérequis pour que le système fonctionne. Néanmoins, il serait aussi envisageable de monter le système en créant nous-même le fichier caché `.tags` vide s'il n'existe pas, il pourrait alors être rempli en fonction des actions de l'utilisateur.

## 2.2 Informations sur un noeud du système

Pour gérer la cohérence des opérations faites par l'utilisateur dans le dossier mnt, `fuse` fait appel à notre implémentation de la fonction **getattr()**. Celle-ci permet d'identifier chaque élément de mnt.
Nous souhaitons donc dans cette fonction, indiquer précisément si le chemin demandé existe ou non. Plusieurs configurations sont possibles :

**Ou bien le chemin demandé concerne un fichier :** dans ce cas, il faut vérifier que le fichier existe dans la structure de données (cela signifie qu'il est bien présent dans le dossier images) et qu'il possède tous les tags du reste du chemin. Par exemple si le chemin est *foo/bar/toto.jpeg*, toto.jpeg est-elle dans notre table de hashage ? possède-t-elle les tags foo et bar ?

**Ou bien le chemin demandé concerne un répertoire :** il faut alors vérifier qu'il existe bien un fichier dans le système qui possède tous les tags du chemin. Cette approche a été revue lors de l'ajout de la fonctionnalité d'ajout automatique des tags avec `ln` et cette vérification a disparue (voir la section 3).

Une fois la vérification de l'existence faite, il suffit de transmettre les bonnes informations sur le noeud. Si c'est un fichier, alors il se trouve à la racine du dossier images donc on utilise la fonction **stat()** de la libc sur le véritable chemin (dans images et non dans mnt). Sinon, comme les sous-répertoires de images ne sont pas pris en compte, le noeud concerne un tag. C'est donc un dossier qui a les mêmes attributs que images ; on utilise une nouvelle fois **stat()** sur images.

Nous sommes donc parti d'une version de **getattr()** très précise mais elle a été rendue plus laxiste en fonction des extensions que nous avons rajoutées.

## 2.3 Lecture d'un dossier

Pour que notre système se comporte comme un véritable système de fichier, l'utilisateur doit pouvoir circuler dans son arborescence. Nous avons donc défini la méthode **readdir()** utilisée par `fuse`.

Considérons d'abord le cas particulier de la racine. Lorsque l'utilisateur souhaite accéder au contenu du dossier mnt, il souhaite voir tous les fichiers disponibles ainsi que tous leurs tags associés (sous forme de dossier). Il suffit donc de déclarer tous les éléments de notre structure de données.
Pour un chemin non trivial, il ne faut sélectionner que les fichiers qui possèdent tous les tags présents dans le chemin puis afficher les répertoires correspondants aux tags de ces fichiers quand ils ne sont pas

déjà présents dans le chemin.

Ainsi notre fonction **readdir()** s'exécute avec une complexité en temps dans le pire des cas de l'ordre de $O(nombredefichiers \times (nombredetags)^2)$.

## 2.4 Ajout, renommage et suppression d'un tag

Ces opérations ont été définies dans les fonctions **mkdir()**, **link()**, **rename()**, **unlink()** et **rmdir()** utilisées par `fuse`.

Dans un premier temps, nous avons réalisé la version minimale du projet dans laquelle ces opérations sont effectuées une par une par l'utilisateur (ajout d'un seul tag à la fois etc.). Ainsi, l'ajout, le renommage et la suppression d'un tag consistent en une recherche dans nos deux tables de hashage, un parcours de liste chaînée et une modification en temps constant ; elles ont donc une complexité en temps de $O(nombredefichiers + nombredetags)$.

Nous avons ensuite ajouté des fonctionnalités supplémentaires, et notamment l'ajout, le renommage et la suppression de plusieurs tags à la fois ce qui modifie les complexités de ces opérations avec le facteur du nombre de tags présents dans le chemin. Et nous avons aussi développé l'extension d'ajout et de suppression à la volée qui a rendu la fonction **mkdir()** inutile (voir la section 3).

# 3 Extensions disponibles

Nous avons ajouté un certain nombre de fonctionnalités supplémentaires au système de fichiers afin de le rendre plus complet et plus facile à utiliser.

## 3.1 Création et suppression des tags à la volée avec `ln` et `rm`

Le système de fichiers a été modifié afin que mkdir et rmdir n'aient jamais à être appelés explicitement.

Lorsqu'un tag est supprimé d'une image, on effectue une vérification de l'entrée de la table de hachage correspondant à ce tag. Si la liste chainée qui lui correspond est vide, c'est que ce tag n'est plus appliqué à aucune image, et le dossier est donc supprimé.
A l'inverse, lorsqu'on ajoute à une image un tag qui n'existe pas encore, le dossier correspondant à ce tag sera automatiquement créé.

Cette modification a eu quelques effets sur le système de fichiers : il est par exemple possible d'effectuer une commande cd vers un dossier qui n'existe pas, cependant ce n'est pas un problème car le dossier sera vide et il n'apparaitra pas après être revenu au dossier précédent.
Par ailleurs, il n'est plus possible de créer manuellement un dossier car getattr retourne le stat du dossier racine pour tout tag, même inexistant, ce qui fait que le dossier existe donc déjà du point de vue du système de fichier.

## 3.2 Ajout, renommage et suppression de plusieurs tags d'un coup

Cette extension ne complique que peu la version minimale du projet. En effet, elle implique uniquement de parser le chemin passé en argument des fonctions concernées et d'effectuer les opérations sur la base de données pour chacun des tags trouvés. Cela ajoute un facteur à la complexités des opérations qui est le nombre de tags dans le chemin et qui vaut dans le pire des cas le nombre total de tags.

Le renommage est vu comme une suppression puis un ajout des tags indiqués dans les deux chemins fournis.
Au début, le renommage multiple n'était pas possible vu la rigueur de la fonction **getattr()**. En effet, si le dossier de destination comporte plusieurs tags qui n'existent pas, **getattr()** indiquait que le chemin n'existait pas. Cependant, avec la modification de cette fonction pour autoriser plus de flexibilité pour l'extension d'ajout à la volée, cette contrainte fut aussi dégagée.

## 3.3 Consulter les tags d'un fichier

Pour consulter les tags d'un fichier, il faut pouvoir fournir à l'utilisateur une commande spécifique or les commandes disponibles par défaut ont déjà un traitement attitré : on ne pourrait pas en effet utiliser **read()** qui sert déjà à afficher le contenu du fichier. Il fallait donc ajouter une nouvelle commande. C'est ainsi que nous avons réalisé le programme *printtags.c* qui se trouve à la racine du projet. Ce programme partage avec *tagfs.c* un même header *tagioctl.h* qui définit un code de requête et une structure de données partagée. Ainsi, *printtags.c* peut envoyer une requête au système de fichier avec ce code et celui-là connaît la manière d'y répondre avec la structure de données partagée. Pour ce faire, on utilise l'appel **ioctl()** dans *printtags.c* et on traite cet appel système dans *tagfs.c*.

Quelques problèmes se sont soulevés au moment de passer les données depuis *tagfs.c* dans *printtags.c*. En effet, nous voulions utiliser comme structure partagée une liste chaînée (tagfs la remplit et printtags la lit). Cependant, une telle implémentation implique que *tagfs.c* alloue les structures qu'il veut transmettre sur son tas, mais celles-ci ne sont pas allouées chez *printtags.c* ! Si on veut que *tagfs.c* écrive dans la mémoire de *printtags.c*, il faut que la mémoire indiquée dans **ioctl()** soit située sur la pile. Cela soulève

donc une limite : la structure partagée est un tableau de chaînes de caractères dont la taille est fixée (nous avons choisi 100 tags maximum pour un fichier et une longueur maximum de 100 caractères pour un tag).

## 3.4 Détecter l'ajout la suppression ou le renommage des fichiers dans le dossier source

L'utilisateur voudrait pouvoir modifier le contenu du dossier images pendant que le système `tagfs` fonctionne et que celui-ci prenne en compte ses modifications dans le dossier mnt. Par exemple, il ne peut pas renommer un fichier dans le dossier mnt mais il voudrait pouvoir le faire dans images et que mnt prenne cela en compte. Il faut donc détecter ces modifications pour mettre à jour notre structure de données (tables de hashage).

Pour détecter les modifications effectuées dans le dossier source, nous utilisons la librairie `inotify`. Il faut scanner en permanence le dossier source et donc créer une deuxième boucle infinie (la première étant le main de `fuse`). Nous faisons donc un **fork()** dans le programme *tagfs.c* :

**le fils** détecte avec les méthodes de `inotify` les évènements dans le dossier images

**le père** exécute le main de `fuse`

Ensuite, il faut pouvoir faire communiquer le processus fils avec son père afin d'indiquer les changements à effectuer à notre structure de données utilisée par nos implémentations des fonctions utilisées par `fuse`. Nous définissons donc un protocole de communication entre les deux processus pour des messages qu'ils s'enverront à travers un `pipe`.
Le processus fils écrit dans le pipe *"A <nom du fichier>\n"* lorsqu'un fichier est ajouté et *"D <nom du fichier>\n"* lorsqu'il y a une suppression. Le père vérifie à chacun de ses appels à la fonction **getattr()** ou **readdir()** si quelque chose est écrit dans le pipe (attention : il a fallu rendre la lecture non bloquante) pour mettre à jour les tables de hashage. Nous avons choisi d'effectuer cette mise à jour au moment de l'appel à **getattr()** ou **readdir()** car nous avons remarqué que `fuse` fait régulièrement appel à ces fonctions (par exemple il fait appel à **getattr()** avant de faire un **link()**) donc les mises à jour devraient être très régulières et ne jamais passer inaperçues.

On notera que nous avons traité le renommage comme une suppression puis un ajout ce qui est discutable puisque tous les tags associés sont donc supprimés. Nous pourrions être plus précis dans une future version en ajoutant une lettre au protocole.

## 3.5 Sauvegarde de la base de donnée au cours de l'exécution

Afin de pouvoir observer les modifications effectuées dans `tagfs` en temps réel, il faut régulièrement mettre à jour le fichier *images/.tags*. Pour des raisons de comparaisons et de reprise à zéro, on écrira les modifications plutôt dans un nouveau fichier *images/.tags.new*.
A chaque modification de la structure de données, plutôt que de faire une recherche dans le fichier pour effectuer la modification, nous avons préféré réécrire tout le fichier. Cela n'est globalement pas beaucoup plus coûteux et cela facilite grandement l'implémentation.

# 4 Extensions inachevées

## 4.1 Gestion des regex

Pour donner plus de flexibilité en effectuant des recherches il se trouve qu'il est utile de supporter les expressions régulières. Ceci facilite la tâche de l'utilisateur, en effet, il serait capable de rechercher entre les tags sans les connaître tous et élargir ainsi les mots-clés qu'il peut utiliser pour chercher. Les expressions régulières seront les arguments de la commande `ls`, nous avons choisi une solution simple qui ne nécessite pas de changer radicalement le code de base. Au niveau des fonctions `getattr` et `readdir` la façon dont la vérification des tags dans le chemin a été changée. Au lieu de comparer les tags avec la base de données, on vérifie si l'expression régulière correspond à l'une des entrées de la base ; dans le cas où il y en a plusieurs, on choisit la première. Cependant, cette solution n'est pas disponible dans le code final. En effet, `getattr` et `readdir` vérifient la cohérence du système et cette cohérence devait être maintenue avec l'utilisation des expressions régulières or nous n'avons pas eu le temps de régler ce problème (les fichiers trouvés dans une arborescence utilisant des regex n'étaient pas reconnus comme valides).

## 4.2 Utilisation des données EXIF

Les fichiers images contiennent des informations EXIF, des métadonnées qui donnent des renseignements sur la nature de l'image considérée.

Nous avions prévu d'utiliser le module exiftool, qui permet de récupérer ces données, de la manière suivante :
Au moment de la création du système de fichiers, exiftool est appelé sur toutes les images du répertoire images et on écrit sa sortie dans un fichier. les informations sont ensuite parcourues afin de créer des tags (exemple : taille de l'image, date de création) indépendamment du contenu du fichier .tags.

Malheureusement, par manque de temps, cette extension n'a pas été terminée et n'est donc pas présente dans le code final.

# 5 Tests

Afin de tester notre code, nous disposons de différents outils. Nous utilisons en premier lieu, le fichier de log créé au montage du système `tagfs`. Ensuite pour tester nos modules séparément nous nous servons de `gdb` et `valgrind`. Nous testons aussi grâce à des **main()** appropriés ou des scripts imitant une utilisation type du système de fichier avec tags.

## 5.1 Le parser

Nous avons réalisé un test pour verifier l'API et le parser. le principe est simple, scanner le fichier `.tags` puis créer les deux tables de hashages contenant les informations à propos les fichiers et les tags, et enfin reproduire un fichier `.tags` à partir des deux tables créées. On compare les deux version du `.tags` pour identifier les incohérences.

## 5.2 Les fonctionnalités

Pour tester l'ensemble des fonctionnalités du système de fichier, nous avons d'abord utilisé les scripts fournis avec le sujet pour la version minimale. Puis nous avons modifié et ajouté d'autres scripts permettant de tester les extensions. De manière générale, les tests effectuent des opérations dans mnt et vérifient les modifications en testant l'existence de fichiers et en comparant avec `diff` le fichier *images/.tags.new* avec le résultat attendu. On notera que `diff` fait la différence dans l'ordre d'apparition des tags et des fichiers alors que celui-ci n'importe pas donc il faut être vigilant car un test qui ne passe pas peut tout simplement être dû cet ordre.

# Conclusion

En conclusion, nous avons réussi à mettre en place un système de fichiers fonctionnel, intégrant même un bon nombre d'extensions proposées par le sujet, même si nous n'avons pas réalisé toutes les extensions proposées, ni même tout ce que nous avions prévu à la base.