PNL - Projet 2019/2020

`ouiche_fs` – le système de fichiers le plus classe du monde

Date limite le : **lundi 8 juin 2020 à 12h00**

Redha Gouicem, Rémi Oudin, Julien Sopena

Dans ce projet, vous implémenterez de nouvelles fonctionnalités dans un système de fichiers existant, `ouiche_fs`, sous la forme d'un module compatible avec la version 4.19.3 de Linux. Comme en TP, il sera nécessaire d'utiliser un site de référencement croisé du code du noyau Linux ; par exemple https://elixir.bootlin.com/linux/v4.19.3/source. Il est également fortement recommandé de lire la documentation du fichier `Documentation/filesystems/vfs.txt` présent dans les sources du noyau pour mieux comprendre l'implémentation d'un système de fichiers (une version en ligne est également disponible https://www.kernel.org/doc/html/latest/filesystems/vfs.html).

Rappel : Lire attentivement l'énoncé en entier !

Dans un premier temps, téléchargez les sources de `ouiche_fs` sur GitHub à l'adresse suivante : https://github.com/rgouicem/ouichefs.

En plus des sources, vous trouverez des explications sur la conception de ce système de fichiers, ainsi que les relations entre les structures de données du noyau entrant en jeu dans `ouiche_fs`. N'hésitez pas à lire les sources (amplement commentées) et à expérimenter `ouiche_fs` sur votre machine virtuelle afin d'en comprendre le fonctionnement.

1 Rotation du système de fichiers

Dans ce projet, on vous propose d'ajouter un ensemble de mécanismes permettant de supprimer automatiquement des fichiers lorsque l'espace disque devient critique. Cette fonctionnalité peut être particulièrement utile pour la gestion des logs.

Dans un premier temps, vous implémenterez une version offrant une rotation des fichiers, c'est-à-dire que le fichier supprimé par le mécanisme sera le plus vieux (date de dernière modification).

Votre mécanisme devra se déclencher sous deux conditions :

— lorsqu'il ne reste que x% des blocs libres de la partition, où x sera un paramètre de votre mécanisme ;

— lorsque tous les inodes d'un répertoire ont été utilisés.

Notez bien que votre mécanisme doit se déclencher à la volée dès que l'une des deux conditions est vérifiée, *i.e.* qu'il ne doit pas nécessiter d'intervention de la part de l'utilisateur (commande, démontage de la partition, ...)

D'autre part, vous veillerez à ne pas supprimer un fichier si celui-ci est en cours d'utilisation par un processus. Pour simplifier, on supposera qu'il existe toujours suffisamment de fichiers non utilisés pour faire les suppressions.

Pour tester ce système de fichiers, vous pourrez créer de nombreux fichiers et modifier leur date de dernière modification avec la commande `touch`. Vous pourrez aussi générer aléatoirement des données avec la commande *dd* et le device */dev/urandom* (toutes autres méthodes fonctionnelles seront acceptées).

2 Extensions du projet : fonctionnalités étendues

Une fois réalisée cette première étape, vous étendrez votre système de fichiers avec les fonctionnalités suivantes (il est conseillé de privilégier la qualité à la quantité).

2.1 Politique de suppression de fichiers

Suivant le contexte, il peut être intéressant d'utiliser un autre critère pour le choix des fichiers à supprimer, en utilisant par exemple le nombre d'ouvertures, la date de dernier accès ou encore sur la date de création.

On vous propose donc d'implémenter une nouvelle politique de suppression basée sur la taille des fichiers (suppression du plus gros fichier). En outre, cette modification du système de fichiers ne devra passer que par l'insertion d'un module.

2.2 Interaction user / fs

Afin de pouvoir anticiper le problème de congestion de la partition et donc d'éviter d'avoir une latence incontrôlée, il peut être intéressant pour un utilisateur de déclencher lui-même votre mécanisme de libération de l'espace disque.

Dans cette deuxième extension, on vous propose d'ajouter un moyen de communication avec le système de fichiers, qui permette à l'utilisateur d'activer le mécanisme. Vous êtes libre d'utiliser le moyen de communication que vous jugez le plus adéquat.

3 Bonus : Correction de bugs

Si vous décelez des bugs dans `ouiche_fs`, n'hésitez pas à les signaler et à les corriger (en échange de quelques points bonus bien sûr...). S'il est possible d'envoyer votre patch par mail, nous vous encourageons à soumettre une Pull Request qui sera intégrée dans le dépôt GitHub `ouiche_fs`, restant ainsi visible pour les futures promotions du master.

4 Soumission

4.1 Procédure

Vous devrez soumettre votre travail sur la plateforme Moodle au plus tard le **lundi 8 juin 2020 à 12h00** sous la forme d'une archive au format `tar.gz`.

Cette archive devra contenir :

— un répertoire *src_fs* contenant les sources modifiées de `ouiche_fs` sans résidus de compilation
— un répertoire *src_mod* contenant le(s) source(s) de module(s) ad hoc (chaque module doit être accompagné d'un README)
— un répertoire *tests* contenant les jeux de tests que vous avez utilisés pour valider vos développements (ces tests doivent être accompagnés d'un README)
— un court rapport au format `pdf`

Le rapport contiendra les explications de vos choix de conception (notamment les algorithmes et les structures de données utilisés), ainsi que l'état d'avancement de votre projet sous la forme des 3 sections suivantes (vide le cas échéant) :

— Liste des fonctionnalités implémentées et fonctionnelles
— Liste des fonctionnalités implémentées, mais qui ne fonctionne pas complètement (vous expliquerez dans ce cas le(s) problème(s), leurs origines ainsi que des pistes de résolution si vous en avez)
— Liste des fonctionnalités qui n'ont pas été implémentées

4.2 Évaluation

Vous serez bien entendu évalués sur le fonctionnement de votre projet, mais également sur la qualité et la lisibilité de votre code et de votre rapport. Pensez à utiliser le script `checkpatch.pl` pour le premier et un correcteur orthographique pour le deuxième.

5 Annexe

5.1 Buffer cache (include/linux/buffer_head.h)

`struct buffer_head *sb_bread(struct super_block *sb, sector_t block bno);` : renvoie un pointeur sur le buffer en cache du bloc `bno` de la partition `sb`. La taille du buffer lu correspond à `sb->s_blocksize`, et les données de ce buffer sont accessible via le champ `b_data` de la `struct buffer_head` renvoyée.

`void mark_buffer_dirty(struct buffer_head *bh);` : signale au buffer cache que le buffer `bh` a été modifié et devra être recopié sur le disque.

`int sync_dirty_buffer(struct buffer_head *bh);` : force l'écriture du buffer `bh` sur le disque. Renvoie 0 en cas de succès.

`void brelse(struct buffer_head *bh);` : relâche la référence sur le buffer `bh`.

5.2 Inode (include/linux/fs.h)

`void mark_inode_dirty(struct inode *inode);` : signale une modification de l'inode. Lorsque celle-ci sera libérée, la fonction `destroy_inode()` sera appelée sur cet inode.