Documentation Projet PIGNOUFS

Projet Informatique de Gestation d'un Nouvel Outil Ultraverifié de File System

Un système de fichiers embarqué avec vérification d'intégrité

Sommaire:

---

1. Introduction

Le projet **PIGNOUFS** (*Projet Informatique de Gestation d'un Nouvel Outil Ultraverifié de File System*) a pour objectif de créer un système de fichiers embarqué dans un fichier régulier (conteneur), avec accès mémoire via `mmap` uniquement.

Ce système vise à reproduire certaines commandes UNIX tout en garantissant l'intégrité des données via un hachage SHA1 de chaque bloc.

---

2. Format et structures du système

Le système est divisé en **4 zones** principales:

Zone	Description
Superbloc	Un seul bloc contenant les métadonnées du système
Bitmaps	Suivi des blocs libres
Inodes	1 inode par bloc (métadonnées des fichiers)
Blocs allouables	Libres, données ou indirectionStructure des blocs

Chaque bloc a une **taille fixe de 4128 octets**, avec la structure suivante:

Section	Taille	Description
Données	4000 octets	Contenu principal du bloc
SHA1	20 octets	Hachage d'intégrité
Type	4 octets	Identifiant du type de bloc
Verrous (lecture)	72 octets	Mécanisme de verrouillage lecture
Verrous (écriture)	72 octets	Mécanisme de verrouillage écriture

Types de blocs

Code	Type de bloc
1	Superbloc
2	Bitmap
3	Inode
4	Bloc libre
5	Bloc de données
6	Bloc d'indirection simple
7	Bloc d'indirection double

3. Structures de données

Bloc générique

```
typedef struct {
    unsigned char data[DATA_SIZE];
```

```
    unsigned char sha1[SHA1_SIZE];
    uint32_t type;
    unsigned char lock_read[LOCK_SIZE];
    unsigned char lock_write[LOCK_SIZE];
} block_t;
```

Superbloc

```
typedef struct {
    char magic[8];
    uint32_t block_size;
    uint32_t num_blocks;
    uint32_t num_free_blocks;
    uint32_t bitmap_start;
    uint32_t inode_start;
    uint32_t data_start;
    uint32_t max_inodes;
} superblock_t;
```

Inode

```
typedef struct {
    uint32_t flags;             // bits : existence, lecture, écriture, verrou lecture,
verrou écriture, type répertoire
    uint32_t mode;              // droits
    uint32_t size;              // taille fichier
    uint32_t direct_blocks[10]; // pointeurs directs
    uint32_t indirect_block;    // indirection simple
    char filename[256];
} inode_t;
```

Entrée de répertoire

```
typedef struct {
    uint32_t inode_index;
    char name[256];
```

```
    uint8_t type;           // 0: fichier, 1: répertoire
} dir_entry_t;
```

4. Commandes implémentées

Le système de fichiers PIGNOUFS implémente les commandes suivantes, inspirées des commandes UNIX classiques:

Commande	Usage	Description
`mkfs`	`mkfs <fsname> <nombre inode> <nombre blocks>`	Créer un système de fichiers
`ls`	`ls <fsname>`	Lister les fichiers du système
`df`	`df <fsname>`	Afficher l'espace libre
`cp`	`cp <fsname> <source> <destination>`	Copier un fichier
`rm`	`rm <fsname> <fichier>`	Supprimer un fichier
`lock`	`lock <fsname> <fichier> <mode>`	Verrouiller un fichier (mode: read/write)
`chmod`	`chmod <fsname> <fichier> <mode>`	Modifier les droits d'accès
`cat`	`cat <fsname> <fichier>`	Afficher le contenu d'un fichier
`input`	`input <fsname> <fichier>`	Écrire l'entrée standard dans un fichier
`add`	`add <fsname> <source> <destination>`	Ajouter un fichier à un autre
`addinput`	`addinput <fsname> <fichier>`	Ajouter l'entrée standard à un fichier existant
`find`	`find <fsname> <fichier>`	Rechercher un fichier
`mkdir`	`mkdir <fsname> <dossier>`	Créer un dossier
`rmdir`	`rmdir <fsname> <dossier>`	Supprimer un dossier
`fsck`	`fsck <fsname>`	Vérifier l'intégrité du système de fichiers

> Note: La commande mount définie dans le code n'a pas été implémentée dans la version actuelle.

5. Contrôle d'intégrité

Notre implémentation du contrôle d'intégrité repose sur un hachage SHA-1 appliqué à chaque bloc de 4 000 octets de données. À chaque écriture, après avoir rempli ou modifié les données utiles d'un bloc, nous calculons la valeur SHA-1 et l'enregistrons dans les 20 octets suivants du bloc, puis mettons à jour son champ `type` et verrouillons éventuellement la zone.

À la lecture, nous recalculons le SHA-1 sur les 4 000 octets de données et le comparons au hash stocké : toute divergence est immédiatement signalée comme corruption.

De même, la commande `fsck` parcourt l'ensemble des blocs (superbloc, bitmaps, inodes et données) pour vérifier systématiquement la cohérence des SHA-1, du `magic` du superbloc et de la classification des blocs selon leur position, garantissant ainsi la détection fiable de toute altération accidentelle ou malveillante des données.

---

6. Gestion de la concurrence

Stratégie de verrouillage:

- **Stratégie de verrouillage par fichier (inode)**

Nous avons opté pour un verrouillage fin au niveau de chaque inode, afin de maximiser la concurrence des accès sur des fichiers différents tout en garantissant la cohérence des données sur chaque fichier individuel.

Dans notre implémentation, chaque inode dispose de deux mutex POSIX partagés entre processus :

- `lock_read` pour coordonner les accès en lecture,
- `lock_write` pour coordonner les accès en écriture.

Lecture (`cat`, `ls`, etc.)

1. On tente un `pthread_mutex_trylock(lock_read)` :
    - Si le mutex est libre, on le relâche immédiatement et on lit sans bloquer.
    - Sinon, on bloque sur `pthread_mutex_lock(lock_read)` jusqu'à ce que l'écriture en cours se termine.

Écriture (`cp`, `input`, `add`, etc.)

1. On acquiert `lock_read` par `pthread_mutex_lock(lock_read)` et `lock_write` par `pthread_mutex_lock(lock_write)`.
2. L'opération d'écriture s'effectue en excluant tout autre écrivain et lecteur , puis on relâche `lock_write`.

---

7. Extensions

Fonctionnalités supplémentaires:

Pour enrichir le système de fichiers de base, nous avons expérimenté deux extensions :

1. **Prise en charge préparatoire des sous-répertoires**
   - Définition d'une nouvelle structure `dir_entry_t` permettant de stocker, dans un bloc de données de type répertoire, un tableau d'entrées `{inode_index, name[256], type}`.
   - Implantation d'une logique minimale pour créer (`mkdir`) et parcourir (`ls //rep`) ces blocs de répertoires, sans toutefois gérer la navigation contextuelle (aucune notion de « répertoire courant »).
2. **Commande** `find`
   - Recherche récursive non-hiérarchique (parcours plat) sur tous les inodes alloués
   - Filtrage case-insensible du nom de fichier avec une fonction `contains_pattern()` qui normalise une copie des chaînes en minuscules et utilise `strstr()`
   - Affichage pour chaque correspondance du nom, de la taille et des permissions (r, w, d)
   - Bilan du nombre de résultats trouvés

Ces extensions démontrent la modularité du projet :

- la structure d'inode et de bloc de données se prête naturellement au stockage d'entrées de répertoire,
- la couche commune de parcours d'inodes est directement réutilisable pour implémenter des outils d'administration comme `find`.

---

8. Implémentation technique

Détails d'implémentation:

Le projet est compilé via un **Makefile** simple à la racine du dépôt, définissant les cibles `all`, `clean` et `extensions` pour produire l'exécutable unique `pignoufs` et gérer les modules optionnels.

La **bibliothèque pthread** est utilisée pour les verrous inter-processus (mutex partagés) afin de réaliser la synchronisation fine au niveau des inodes, conformément aux exigences de concurrence.

Le code respecte les **normes POSIX/Linux**, notamment pour l'utilisation de `mmap` (projection mémoire du conteneur), `fcntl` pour la gestion des fichiers et le format little-endian imposé ; seule la contrainte d'alignement sur la taille de page Linux (4 Kio) est spécifique au test d'intégrité des blocs, sans recours à des extensions non-standard.

---

9. Tests

Scénarios de test

1. **Initialisation et vérification de base**
   - `mkfs test 50 50` → création sans erreur, `fsck test` affiche « Système de fichiers valide ».
   - `ls test` → « Aucun fichier trouvé ».
2. **Création, lecture et suppression de fichier**
   - `echo "Hello" > text`
   - `pignoufs cp test text //text` → écrit 1 bloc, `fsck test` valide
   - `pignoufs cat test //text` → affiche « Hello ».
   - `pignoufs rm test //text` → libère le bloc, `fsck test` valide
3. **Verrouillage et concurrence**
   - Dans un terminal A : `pignoufs lock test //text r` (taper lentement).
   - Dans un terminal B : `pignoufs cat test //f1` → bloque tant que A n'a pas terminé.
   - Plusieurs lecteurs simultanés (`cat cat cat`) autorisés dès qu'aucun écrivain n'est actif.
4. **Commande `find`**

- Création de fichiers `a.txt`, `b.log`, `cTXT` → `pignoufs find test txt` liste `a.txt` et `cTXT` (recherche case-insensible).

5. **Gestion des permissions**
   - `pignoufs chmod test //f1 -r` → supprime la lecture, `pignoufs cat test //f1` renvoie une erreur « Permission de lecture refusée ».
   - `pignoufs chmod test //f1 +w` → ajoute l'écriture, vérifié via `ls -l`.

---

10. Équipe

Il n'y pas eu de répartitions de tâches particulières.

Membres et contributions:

- Ziad ACHACH 22406755 @achach
- Samuel TARDIEU 21953875 @tardie

Projet PIGNOUFS - Système de fichiers embarqué avec vérification d'intégrité

Documentation version 1.0