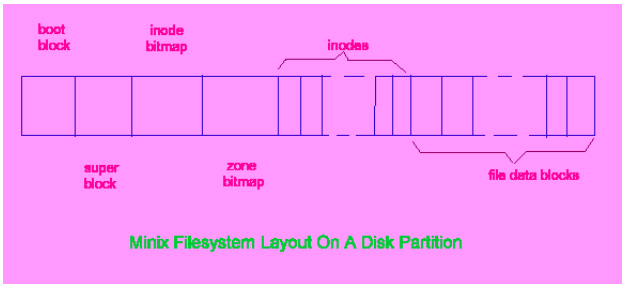
h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Rapport

Minix File system

ITI 2ème soir
2014 / 2015



Sebastien Chassot, Andre-Luc Robyr

Juin 2015

Table des matières

1 Introduction

2 Implémentation python

2.1 add_entry()

```
add_entry(inode dossier, nom fichier a inserer, inode du nouveau fichier){
   si nom de fichier non conforme;
      raise error
   si nom de fichier existe deja dans dossier:
      raise error

   tant que not done:
      prendre block dossier:
         si place libre:
            inserer inode et nom
            changer taille dossier
            done

         sinon:
            si not next block:
               creer nouveau block
               si plus possible creer:
                  break

   si done:
      valider modification sur disk
   sinon:
      raise error plus de place
}
```

Listing 1 – pseudo code add_entry()

2.2 del_entry()

On commence par rechercher l'inode de l'entrée dans le dir et on raise une error si le nom n'existe pas.

Pour chaque block du dir (le dir peut occuper plusieurs block) on recherche l'inode.

Si l'inode a plusieurs link, c'est qu'il est utilisé par un autre fichier il suffit donc de décrémenter le nombre de liens pointant sur lui.

Sinon, on libère tous les blocks de l'inode.

Dans les deux cas, on retire l'entrée.

Si l'entrée était la dernière du dir on peut effacer ce block du dinode. Attention on pourrait créer 300 fichiers et effacer par hasard tous les fichiers du 3^{e} block. Il faut donc supprimer le 3^{e} mais décaler les suivants (la commande *bmap()* renvoit les block qui se suivent)

On écrit finalement le nouveau contenu du block dir.

```
del_entry(inode dossier, nom fichier a supprimer){
   rechercher_entree;
   si (non trouve)
      raise error;

   tant que block dossier:
      rechercher entree dans block dossier
         trouver:
            si plusieurs liens: reduire
            sinon: effacer inode fichier

            enlever entree du block dossier;
```

```
            si dernier fichier du block dossier supprimer block inutile;

    valider modification sur disk;
}
```

Listing 2 – pseudo code del_entry()

2.3 symlink

Il n'était pas demandé de tenir compte des symlinks

read()

- la requête client est un read + un offset + length
- le server return ack + payload

write()

- la requête client est un write + un offset + payload
- le server return un ack

3 Le serveur de blocs

la problématique de la déconnexion du client

Détecter la déconnexion d'un client n'est pas trivial. En effet, lors d'un *shutdown()*,

Solutions envisagées

Une des solutions les plus simple serait de traiter un échange à la fois ; le client se connecte, envoit une requête, reçoit la réponse et se déconnecte. Le server fait la même chose ; attend sur *accept()* qu'un client se connecte, attend une requête y répond et se déconnecte.

Cette solution est simple, permet de mieux maîtriser l'état du client et du server mais est couteuse en connexions.

Une autre solution est d'accepter un client et rentrer dans une boucle, un *fork()* ou un thread et traiter les requêtes du client tant qu'il ne se déconnecte pas. Cette solution est plus élégante mais elle nécessite de détecter la déconnexion du client.

Une alternative possible est d'ajouter au protocole une requête (un message) de déconnexion qui synchronise le client et le server.

Problème de client malicieux

Si un client lance une requête et annonce une certaine longueur, le server va faire un read, tant que le client maintient la connexion et n'envoie pas les données, il bloque le server.

4 Implémentation du server