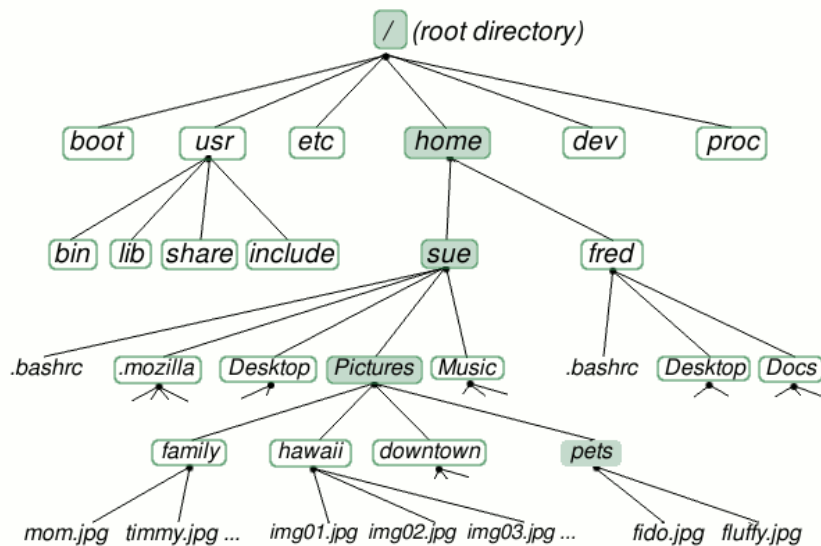


Rapport

Minix File system

ITI 2^{ème} soir
2014 / 2015



Sebastien Chassot

Juin 2015

Table des matières

1	Introduction	3
2	Implémentation python	3
2.1	API	3
2.2	ialloc()	3
2.3	ifree()	4
2.4	balloc()	4
2.5	bfree()	4
2.6	bmap()	4
2.7	add_entry()	5
2.8	del_entry()	5
2.9	symlink	6
3	Le serveur de blocs	6
4	Implémentation du server	7

Table des figures

1	Principe de fonctionnement	3
---	--------------------------------------	---

Liste des tableaux

1 Introduction

Le but de ce travail est d'implémenter un système MinixFS V1 en python en proposant une API standard et de modifier un fichier *formaté* en minixfs via cet interface.

Dans un deuxième temps, les modifications seront faites à travers le réseau. Un server (écrit en C) modifiera le fichier par block reçu via des sockets AF_INET. On réutilisera le travail fait en python et son API mais les block seront transmis au server selon un protocole relativement simple.

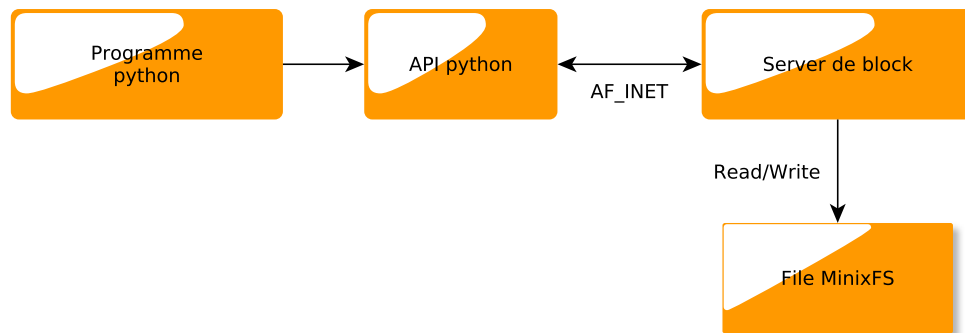


FIGURE 1 – Principe de fonctionnement

Un seul client est traité à la fois ce qui évite les problèmes d'accès concurrent - traité dans d'autres cours. Le principe est simple. le client fait une requête, le server y répond.

2 Implémentation python

2.1 description de l'API

ialloc()
alloue un inode

— ialloc()

2.2 ialloc()

```

1 ialloc(){
    recherche premier libre dans inode_map
    si not trouve:
        raise error plus aucun inode de libre
    inode trouve = occupe
    del ancien inode
    new inode
  
```

```
11 |  
    return numero inode  
}
```

Listing 1 – pseudo code ialloc()()

2.3 ifree()

```
ifree( inode ){  
    inode = libre  
    return est-ce que inode est libre?  
}
```

Listing 2 – pseudo code ifree()()

2.4 balloc()

```
balloc( ){  
    recherche premier block dans block map  
    si non trouver:  
5     raise error plus de place sur disk  
  
    block = occuper  
    return numero de block  
}
```

Listing 3 – pseudo code balloc()()

2.5 bfree()

```
bfree( data block ){  
    data block = libre  
    return est-ce que data block est libre?  
}
```

Listing 4 – pseudo code bfree()()

2.6 bmap()

```
bmap( numero inode, position relative block ){  
    si block < 7:  
5     return position du nieme block direct  
  
    block -= 7  
    si block < nombre inode par block:  
        return position du nieme block de indirect  
  
    block -= nombre inode par block  
    si block < (nombre inode par block)^2:  
        adresse data block = block / nombre inode par block  
        position block = block mod nombre inode par block  
15     return numero block a adresse.posistion  
  
    sinon:  
        raise error depassement de capacite  
}
```

Listing 5 – pseudo code bmap()()

2.7 add_entry()

```

1 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
11         done

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

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

```

Listing 6 – pseudo code add_entry()

2.8 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()* renvoie les block qui se suivent)

On Ã©crit finalement le nouveau contenu du block dir.

```

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

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

        enlever entree du block dossier;
        reduire taille dossier
        si dernier fichier du block dossier supprimer block inutile;

16    valider modification sur disk;
}

```

Listing 7 – pseudo code del_entry()

2.9 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, envoie 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