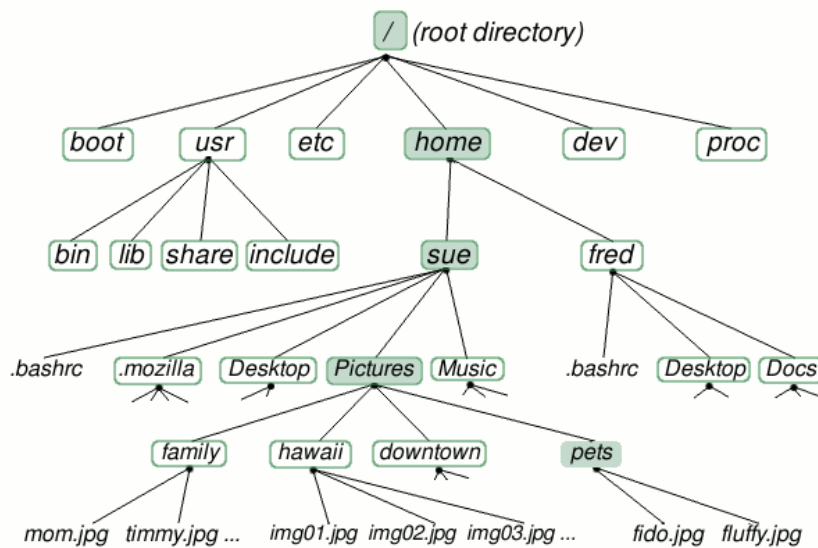


Rapport

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	description de l'API	3
2.2	Logs et exceptions	6
2.3	Problèmes rencontrés	7
3	Le serveur de blocs	7
3.1	la problématique de la connexion	7
3.2	Principe de fonctionnement	7
4	Exécution et tests unitaires	10

Table des figures

1	Principe de fonctionnement	3
2	Principe de fonctionnement du server	9

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 programme de test simple écrit en python utilise l'API mais les lectures/écritures sont faite par un server de blocs. Le server (écrit en C) modifiera le fichier selon les commandes reçues à travers d'une 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.

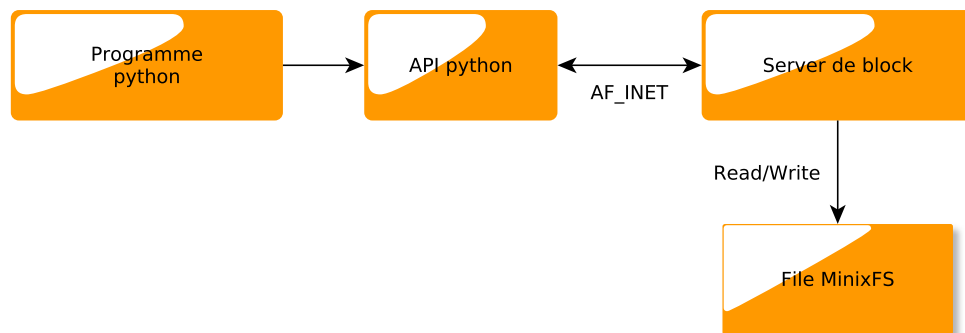


FIGURE 1 – Principe de fonctionnement

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

L'API implémentée est une version simplifiée d'un système de fichiers actuelle. On y retrouve les même méthodes et le fonctionnement global est le même.

2.1 description de l'API

ialloc()

alloue un inode - puisqu'il contient peut-être d'ancienne donnée, il est remis à zéro

ifree(inodenum)

change l'état de l'inode à libre dans la bitmap des inodes

balloc()

recherche et renvoie le premier block libre dans la bitmap des data block

bfree(blocknum)

change l'état du data block à libre dans la bitmap (libre pour une future utilisation)

bmap(inode, blk)

fait la transformation entre un numéro de block relatif et la position effective de ce block sur le disque

lookup_entry()

recherche une entrée dans un dossier et retourne l'inode de ce fichier

namei(path)

Recherche en partant de la racine et en suivant l'arborescence jusqu'à trouver le fichier et renvoie son numéro d'inode

ialloc_block(inode, block)

Recherche un block de libre sur le disque et place

add_entry(dinode, name, new_inode_number)

Ajoute une entrée dans un dossier

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()()

ifree(inode)

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

Listing 2 – pseudo code ifree()()

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()()

bfree(blocknum)

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

Listing 4 – pseudo code bfree()

bmap()

```
bmap( numero inode, position relative block ){
    si block < 7:
        return position du nieme block direct
5
    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()

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
                done
11
            sinon:
                si not next block:
                    creer nouveau block
                    si plus possible creer:
                        break

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

Listing 6 – pseudo code add_entry()

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 écrit 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()`

ialloc_bloc()

Cette fonction attribue un block à n'importe quel position - dans un nouvel inode, on peut vouloir commencer par écrire le block 2452 (dans la zone doublement indirect). Si le block indirect ou doublement indirect n'existe pas, il faut penser à pouvoir l'allouer à n'importe quel moment.

2.2 Logs et exceptions

Dans tous le code python, les erreurs lèvent une exception qui log une erreur *log.error()*.

Dans le reste du programme, deux niveaux de log sont utilisé *log.info()* et *log.debug()*. Il suffit de changer *log.basicConfig()* de *level=log.INFO* à *level=log.DEBUG*.

Il y a aussi moyen de sortir les logs dans un fichier.

```

import logging as log

3 log.basicConfig(format='%(levelname)s:%(message)s', level=log.INFO)
  #LOG_FILENAME = 'minixfs_tester.log'
  #log.basicConfig(filename=LOG_FILENAME,level=log.DEBUG)

class MinixfsException(Exception):
    """ Class minixfs exceptions """

    def __init__(self, message):
        super(MyBaseException, self).__init__(message)
        log.error(message)
```

Listing 8 – initialisation logs et exception MinixfsError()**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

2.3 Problèmes rencontrés

Une difficulté vient de la taille du fichier (dans l'inode). Savoir s'il faut la modifier la taille ou si c'est à l'appel system *write()* de le faire - p.ex en attribuant un block avec *ialloc_bloc()* faut-il corrigé la taille à ce moment là ? Avec des commandes tel que truncate est-ce vraiment au filesystem de se maintenir ?

3 Le serveur de blocs

3.1 la problématique de la connexion

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

3.2 Principe de fonctionnement

La communication est bidirectionnelle mais le client initie toujours la communication. Le server ne fait que répondre aux requêtes.

Dans ce travail, il n'y a pas de concurrence, un seul client est traité à la fois.

Un client se connecte (*accept()*) et le server rentre dans une boucle. On aurait également pu faire un fork ou lancer un thread pour traiter le client.

Tant que le client ne se déconnecte pas, le server bloc sur le premier read. Si read renvoi 0, le server ferme la connexion, sort de la boucle intérieur et bloque sur *accept()* en attente d'un nouveau client.

C'est cette solution qui a été mise en place (dans une boucle infinie) mais elle ne permet pas de se protéger d'un client malveillant. Si un client ne respecte pas le protocole le server peut être

```
while (1){
    accept new client;

    do
        read header request;
        si prob read:
            session = false;
            break;

        prepare header response;

        switch
            case read:
                read disk;

                si fatal:
                    session = false;
                    break;
                si erreur:
                    modifie response;
                    write header response to client;
                si no erreur:
                    write response to client

            case write:
                read client payload;
                si fatal:
                    session = false
                    break;
                sinon:
                    write payload to disk;
                    si erreur:
                        modifie response;

                write response to client;
        while session;
    }
```

Listing 9 – pseudo code server

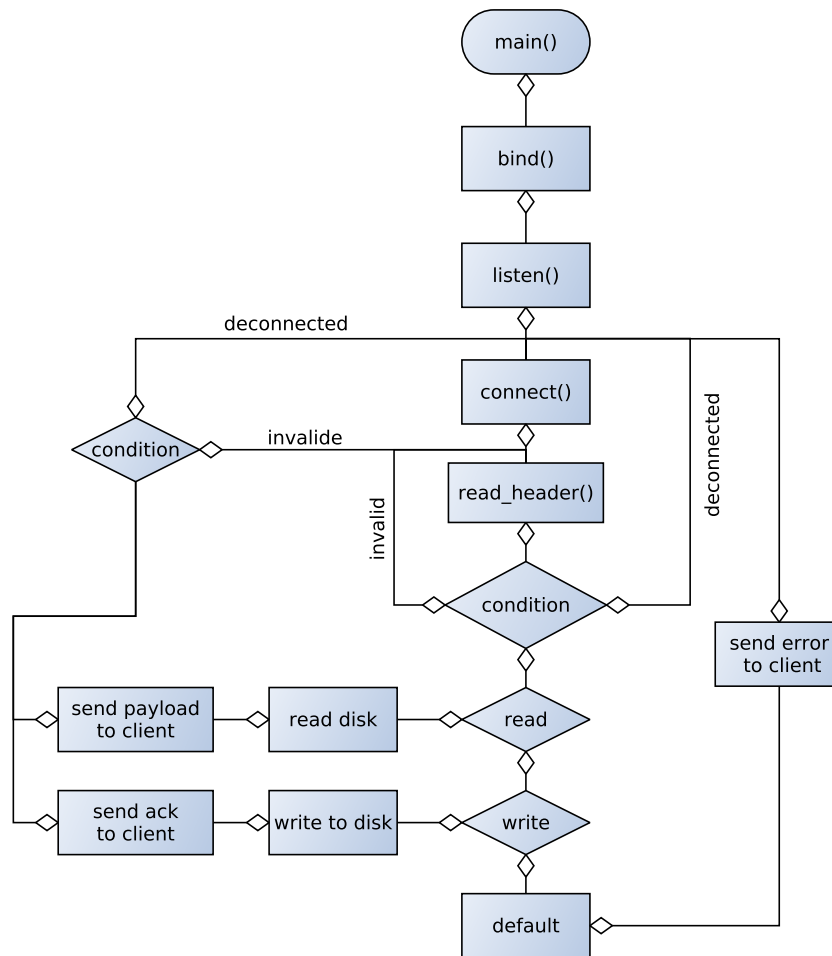


FIGURE 2 – Principe de fonctionnement du server

Solutions envisagées

Une des solutions les plus simple serait de traiter les demandes l'une après l'autre ; 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 puis traite la requête et 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 alternative possible est d'ajouter au protocole une requête (un message) de déconnexion qui synchronise le client et le server. Complexifier le protocole.

Une autre serait de faire une machine d'état afin de s'assurer de l'état dans lequel est le client et le server et le passage entre chaque état. Si un problème arrive, revenir dans un état connu.

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.

une solution serait de faire un *select()* avant le read avec un timeout

```

1 struct timeval tv;
  fd_set readfds;
  int state;

  FD_ZERO(&readfds);

  tv.tv_sec = 5;
  tv.tv_usec = 0;

11 client = accept(s, (struct sockaddr *) &addr_client, (socklen_t *) &s_len);
  FD_SET(client, &readfds);

  boucle {
    state = select(n, &readfds, NULL, NULL, &tv);

    if(FD_ISSET(client, &readfds))
      read_socket();
    else
21   continue;

    ...
  }

```

Listing 10 – pseudo code del_entry()

4 Exécution et tests unitaires

Pour tester le fonctionnement

Où avec l'utilisation de strace

```

5 $ cp ./filesystems/minixfs\_lab1.img.org ./filesystems/minixfs\_lab1.img
  $ ./server 1234 ./filesystems/minixfs\_lab1.img

  $ strace ./server 1234 ./filesystems/minixfs\_lab1.img

  $ make run
  $ make run_debug

```

Listing 11 – lancer le server

on peut enfin lancer le client

```

2  $ python2 tester_server.py

INFO:remote file system opened successfully
INFO:socket closed
INFO:remote file system opened successfully
(...)
.
-----
Ran 13 tests in 71.003s
12 OK

```

Listing 12 – lancer le client