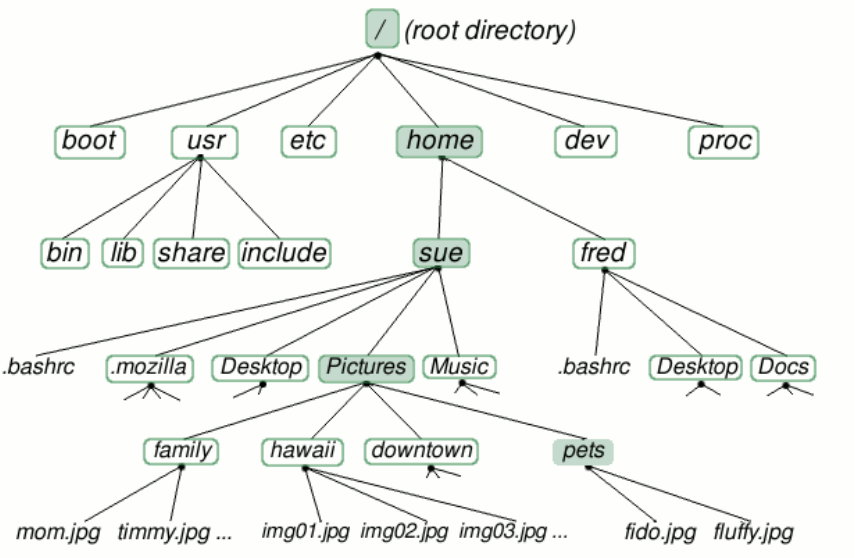
h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Rapport

Minix File system

ITI 2ème soir
2014 / 2015



Sebastien Chassot

Juin 2015

Table des matières

Table des figures

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 au travers d'une sockets AF_INET. On réutilisera le travail fait en python et son API mais les blocs seront transmis au server selon un protocole relativement simple.

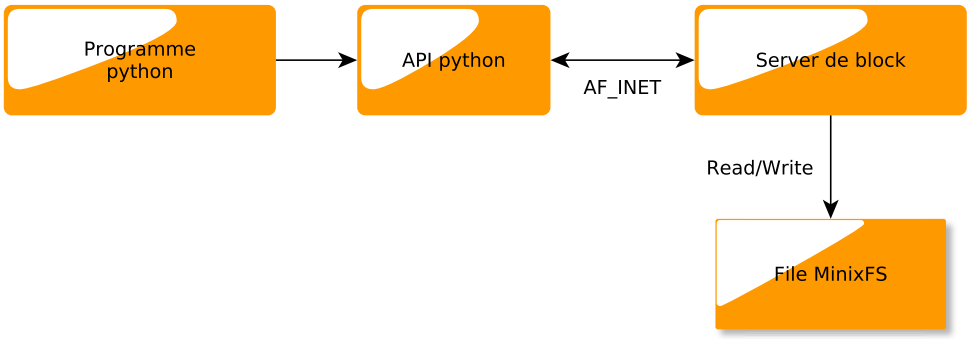


FIGURE 1 – Principe de fonctionnement

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 Exécution du programme

Toutes les commandes peuvent être lancées avec make.

Le projet est hébergé sur https://github.com/selinux/tp-minixfs

Listing 1 – lancer le server

```
pour les tests en local

    $ make test1
ou
    $ make test2

pour lancer la doc
    $ make doc

    $ ./server
    Usage : ./server <port> <file>


pour lancer le server avec make
    $ make server && make run

ou avec strace
    $ make server && make run_debug

dans un autre terminal
    $ make test_server
```

3 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.

3.1 brève description de l'API

ialloc()
alloue le premier inode de libre - puisqu'il contient peut-être d'ancienne donnée, il est remplacé par un nouvel inode vierge.

ifree(inodenum)
change l'état de l'inode à *libre* dans la bitmap des inodes. Cet inode sera donc libre pour une nouvelle allocation.

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_bloc(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

3.2 Complément sur certaines méthodes

bmap()

Il y a un problème avec le test unitaire numéro 8. Si *bmap()* vérifie la taille du fichier (valeur de l'inode). Le test echoue en effet, les 3 zones de l'inode sont testées. Une boucle va de 0 à 6 (direct), une autre de 7 à 518 (indirect) et la troisième va de 519 à 1024 (dbl_indirect) or la taille du fichier est de 659'384 octets et occupe donc 644 blocks. Pour passer le test, il faut soit tester de 0 à 643, soit commenter les lignes suivantes.

Listing 2 – test inode_size in bmap

```
        size = inode.i_size

        # the data block (>firstdatazone) must fit the inode size
        if self.is_file(inode) or self.is_link(inode):
            if blk > int(size / self.disk.blksize):
                raise MinixfsException('Error block is out of file
                    boundary')
```

Dans le test b, *ialloc_ bloc()* ajoute deux blocks sans modifier la taille ce qui fait échouer le test également.

lookup_entry()

Cette fonction va parcourir tous le dinode et créer un dictionnaire inode ;nom_fichier. À la fin, on retourne simplement l'entrée du dictionnaire associée au nom (le numéro d'inode). Cette solution est lourde puisqu'un dictionnaire est créé à chaque appel. C'était juste pour tester les fonctionnalités de python mais il serait bien plus efficace de faire un break dès que l'entrée est trouvée.

del_entry()

Avant de supprimer une entrée, il est préférable de tester s'il s'agit bien d'un dossier (et pas un fichier). Vérifier avec lookup_entry() que le fichier existe bien dans le dossier.

Il faut également vérifier que le lien soit à 1, sinon il faut simplement décrémenter les liens mais garder l'inode (un autre fichier pointe dessus).

Une fois une entrée supprimée, le bloc est comparée à un bloc vide et supprimé s'il n'est plus nécessaire.

Listing 3 – test after del_entry()

```
        # after deleting entry, if dir_content is empty free
            the empty data bloc
        if dir_content == "".ljust(BLOCK_SIZE, '\x00'):
            self.bfree(dir_block)
```

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 doublement indirect indirect n'existe pas, il faut le créer, mettre l'adresse du nouveau bloc du fichier dedans et placer l'adresse du bloc dbl_indirect dans le blocs dbl_indirect.

Plus généralement, en allouant un bloc il faut pouvoir le faire quel que soit sa position.

3.3 Logs et exceptions

Dans tous le code du client, les erreurs lèvent une exception qui log une erreur *log.error()* et l'affiche.

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.

Listing 4 – initialisation logs et exception MinixfsError()

```
import logging as log

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)
```

4 Le serveur de blocs

4.1 la problématique de la connexion

Détecter la déconnexion d'un client est trivial (read() renvoit *0*) mais comment réagir ? Le server en cas de déconnexion libère la mémoire et attend de nouveaux clients.

4.2 Principe de fonctionnement du server

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

Un client se connecte et le server accept() la connexion puis 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 et attend une requête.

Le server tourne dans une boucle infinie (le code suivant la boucle est inaccessible dans l'état actuel). Puis rentre dans une boucle (traitement du client) qui fait successivement :

lecture header requête (20 octets)	
read()	**write()**
lecture disque selon demande client	Lecture payload client
écriture header réponse (12 octets)	écriture payload sur le disque
écrit le payload réponse	écrire header réponse au client (12 octets)

TABLE 1 – Ordre lecture/écriture

À chaque lecture ou écriture, si le client se déconnecte, le server revient à accept().

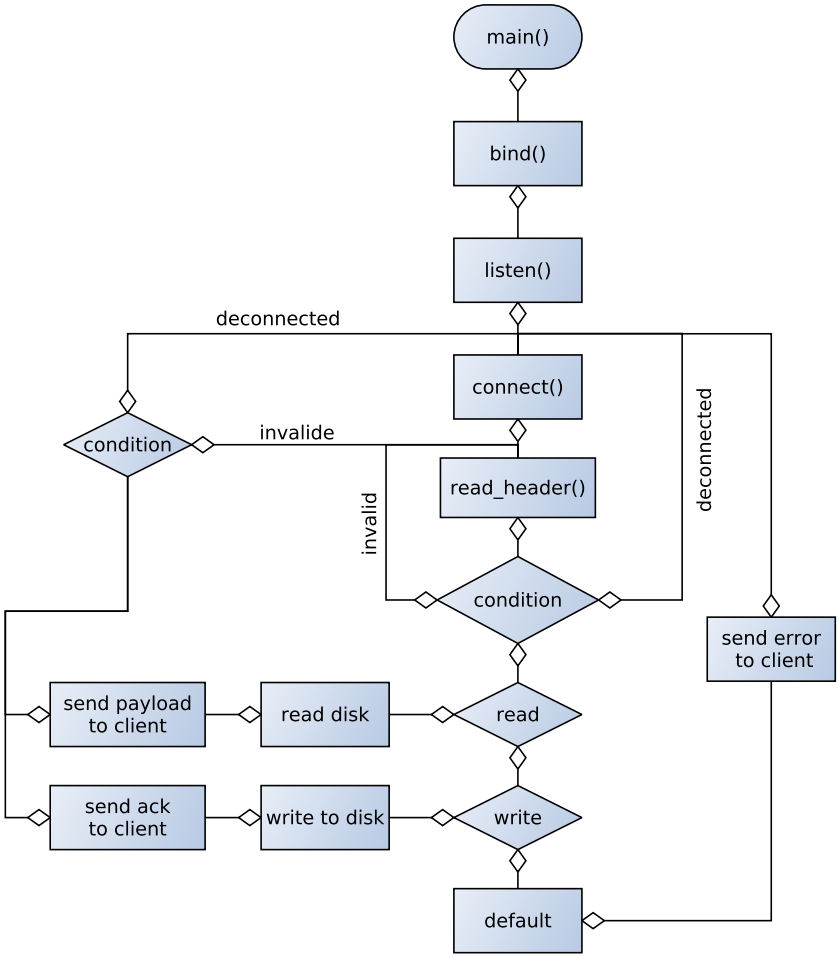


FIGURE 2 – Principe de fonctionnement du server

4.3 Problème de sécurité

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 se retrouver bloqué. Si un client annonce un payload d'une certaine longueur mais envoit d'autres données, il décale le flux et la communication serait désynchronisée laissant le server bloqué sur un read().

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 la synchronisation client/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. On ajoute de la complexité au 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.

5 Conclusion

L'API python fonctionne bien et passe tous les tests unitaires. Il ne manquerait plus maintenant qu'à ajouter les appels systèmes en user space pour qu'il soit fonctionnel.

Le server de bloc fonctionne bien également et passe tous les tests mais est assez sensible à un client qui ne respecterait pas le protocole.