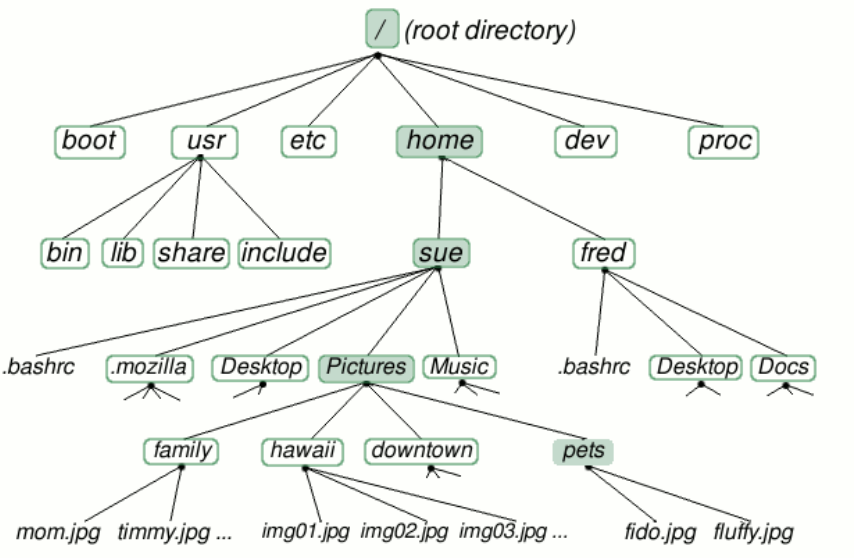
h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

# 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 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.

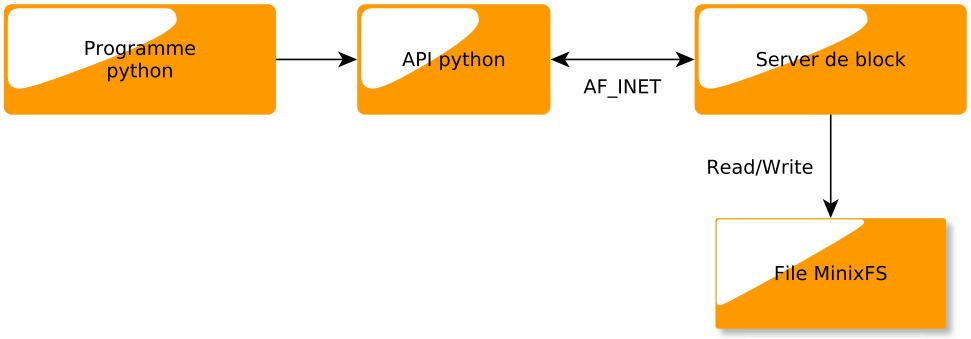



FIGURE 1 – Principe de fonctionnement

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 - 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_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

## 2.2 ialloc()

```
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

   return numero inode
}
```

Listing 1 – pseudo code ialloc()()

## 2.3 ifree(inode)

```
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:
      raise error plus de place sur disk

   block = occuper

   return numero de block
}
```

Listing 3 – pseudo code balloc()()

## 2.5 bfree(blocknum)

```
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:
      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

      return numero block a adresse.posistion

   sinon:
      raise error depassement de capacite
}
```

Listing 5 – pseudo code bmap()()

## 2.7 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 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 3e block. Il faut donc supprimer le 3e 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;

   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;

   valider modification sur disk;
}
```

Listing 7 – pseudo code del_entry()

## 2.9 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.

## 2.10 Logs et exceptions

Dans tous les 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*.

```
import logging as log

log.basicConfig(format='%(levelname)s:%(message)s', level=log.INFO)

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

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

Listing 8 – initialisation logs et exception MinixfsError()

## 2.11 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