mickael.hoerdt@hesge.ch 2014/2015

Mini-projet de programmation Système/Systèmes d'exploitation

Une image Minix-FS version 1.0 accessible localement et à distance

MINIX-FS est un système de fichier créé par Andrew Tanenbaum en 1987, il se base sur le système de fichiers UNIX dont les aspects complexes ont été retirés pour garder une structure simple et didactique[1]. L'objectif de ce projet est d'implémenter un système de fichier MINIX accessible en lecture et en écriture depuis un fichier image formaté en au format MINIX version 1.

En raison de la définition du superbloc, MINIX version 1 est limité à une image divisée en un maximum de 65535 blocs et ne pourra contenir qu'un maximum de 65535 inodes. C'est peu mais amplement suffisant pour mettre en pratique les principes généraux de conception d'un système de fichier.

L'accès local au fichier image contenant le système de fichier MINIX devra être implémenté en python en utilisant les primitives d'accès aux fichiers classiques. L'accès distant devra être implémenté via des sockets python pour la partie cliente. Le client se connectant à un serveur de bloc implémenté en langage C.

La manipulation locale des structures de données stockées sur l'image récupérée via une lecture de blocs locale ou à distante se fera exclusivement en langage python **version 2**, par l'intermédiaire du canevas des classes python fourni avec le projet.

Pour simplifier les traitements, l'entièreté de la table des inodes du système de fichier sera chargée en mémoire dans une liste, donc l'index 0 ne contiendra qu'un inode vide. Dans un système de fichier réel, seul un sous-ensemble des inodes sont en mémoire à un instant donné.

Étape 1 : Manipulation locale des structures de données de base stockées sur l'image

Récupérer le canevas du projet sur :
https://infolibre.ch/hepia/syst-exploitation-2014-2015/minixfs.tgz

- Compléter les méthodes `__init__`, `read_bloc` et `write_bloc` de la classe `bloc_device` du fichier `bloc_device.py`

- Compléter l'initialisation de la classe `minix_super_bloc` à partir d'un objet `bloc_device` déjà initialisé, remplir en particulier les champs `s_ninodes`, `s_nzones`, `s_imap_blocks`, `s_zmap_blocs`, `s_firstdatazone`, `s_log_zone_size`, `s_max_size`, `s_magic` et `s_state`

- Compléter la méthode `__init__` de la classe `minix_file_system`. Celle-ci devra initialiser un champs `inode_map` de type `bitarray` à partir du bitmap d'inodes libres sur le fichier image. Elle devra aussi initialiser un champs `zone_map` de même type à partir du bitmap des blocs de données libres sur le fichier image.

- Une fois complété la méthode `__init__` de la classe `minix_inode` pour une initialisation

1 http://fr.wikipedia.org/wiki/MINIX_fs

d'un objet inode à partir d'une liste de bytes, initialiser le champs `inodes_list` dans la méthode `__init__` de la classe `minix_file_system` pour qu'il contienne la liste de l'ensemble de inodes du fichier image à son ouverture.

- Compléter les méthodes `balloc()`, `bfree()`, `ialloc()` et `ifree()` de la classe `minix_file_system` d'après le cours donné sur les systèmes de fichier.

- Compléter les méthodes `bmap()`, `lookup_entry()`, `namei()`, `ialloc_bloc()`, `add_entry()` et `del_entry()` de la classe `minix_file_system` d'après le cours donné sur les système de fichiers.

Important : Tester avec un interpréteur python version 2 la conformité de votre implémentation à travers le programme `tester.py` fournit dans le canevas. L'évaluation de votre implémentation sera basée sur la conformité de votre code en utilisant le programme `tester.py` mais sur un jeu de données différent que celui fournit dans le canevas.

Les modules ou classes python suivants/tes peuvent vous être utiles :

- `file` pour les opérations sur le fichier image, `open()` pour retourner un objet `file`
- `bitarray` pour gérer les bitmaps des inodes et des blocs de données libres/occupés.
- `struct` pour transformer des types binaires en type python et inversement.
- `hexdump` pour afficher une chaîne de bytes en hexadécimal.

Ces modules/classes sont tous documentés dans l'aide en ligne de python (voir la commande help(<nom_du_module/nom_de_la_classe>), après avoir été importée dans l'interpréteur avec la commande `import <nom_du_module>` sauf si il s'agit d'une classe du module `__builtin__`

Étape 2 : Manipulation des structures de données via un serveur de blocs écrit en C

Le but de l'étape 2 est de réaliser un serveur de bloc qui servira un fichier image accessible à distance par une version modifiée de la classe `bloc_device` de l'étape 1.

Le mini-serveur de blocs sera écrit en C et servira des blocs issus d'un seul fichier donné en paramètre. Les blocs d'un fichier, adressé par leur offset et leur longueur dans le fichier, pourront être écrits ou lus à distance par un seul client (i.e pas d'accès concurrent possible), les ordres d'écritures ou de lecture apparaissant dans les données transportées par le réseau.

Pour cela le serveur devra implémenter un simple protocole requête-réponse qui respecte le format de messages suivant, transporté sur TCP/IP via des sockets C de famille `AF_INET` de type `SOCK_STREAM`.

Le serveur sera exécuté en ligne de commande avec la syntaxe suivante.

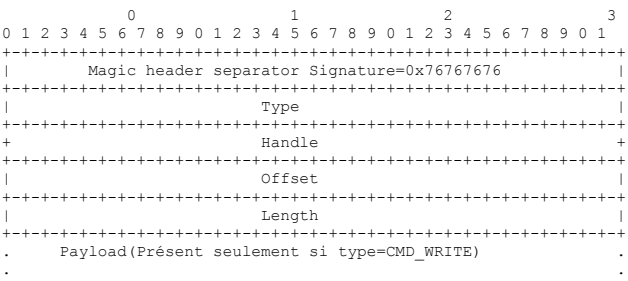
```
hoerdtm@A406-01:~$ ./bloc_server <nom_du_fichier_a_servir>
```

Il tournera continuellement en attente de messages de la part du client.

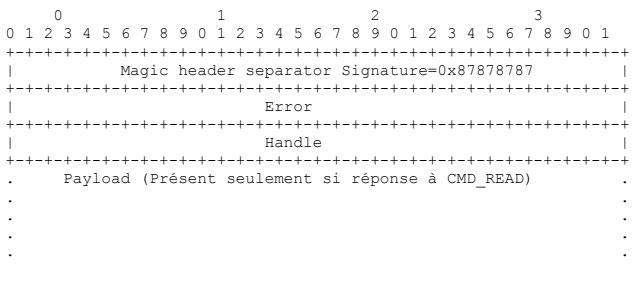
On implémentera le client en modifiant les méthodes `__init__`, `read_bloc` et `write_bloc` de la classe `bloc_device` de l'étape 1, pour qu'elles envoient des requêtes de lecture/écriture au serveur de bloc selon le format spécifié qui suit. On utilisera les sockets (utiliser le module python socket avec `import socket`). Ces méthodes devront bloquer sur l'attente de la réponse du serveur puis renvoyer le contenu reçu de la part du serveur aux objets pythons qui y font appel.

Format des messages :

requêtes :



réponses :



`Magic header separator Signature` : Simplement un nombre bien connu sur 32 bits qui permet d'éviter le traitement de message qui ne commenceraient pas par cette signature protocolaire. C'est un indicateur de début de message puisque ceux-ci arrivent dans un flux d'octets dont on ne peut prévoir la vitesse d'arrivée. En clair : un `read()`, ne renverra pas forcément le nombre d'octets attendu et le traîtement des messages devra se faire en deux temps : le prermier qui s'assure qu'un message est arrivé en entier, le deuxième qui traitement le message proprement dit. Une requête commence par le numéro magique `0x76767676`, une réponse commence par le numéro magique `0x87878787`

`Type` : un entier sur 32 bits qui définit le type de la requête. Deux valeurs possibles :
`0x0 = CMD_READ` ou `0x1 = CMD_WRITE`

`Error` : Uniquement dans les messages de type réponse. Il indique le résultat d'une commande qu'un client à demander à effectuer, une valeur à zéro indique que la commande a été effectuée avec succès, une valeur négative indique une erreur, qui corresponds à la valeur qu'on trouverait dans errno si la requête était effectuée localement.

`Handle` : 32 bits qui identifient de manière unique une requête et qui permet aux clients d'identifier une reponse à une requête et donc de demander plusieurs requêtes en même temps avant de traiter leur réponse : c'est du pipelining. Dans une réponse, le champs Error indique le retour d'erreur/non erreur pour une requête dont le numéro de handle était identique à celui indiqué dans la réponse. L'initialisation du Handle pour une requête devra utiliser la fonction rand(3) de la librairie C associée à une seed initialisée à sur la valeur de l'heure locale du système du client.

`Offset` : un entier sur 32 bits qui indique à partir de quel offset en octet doit commencer la requête d'écriture ou de lecture.

`Length` : longueur des octets à lire ou à écrire qui suivent l'entête du protocole.

`Payload` : Les données lues dans une réponse à une requête de lecture, les données à écrire dans une requête d'écriture. Dans une réponse à une requête d'écriture, le payload est vide, le message

contenant uniquement un code d'erreur et le handle correspondant.

Le format des requêtes devra être codée avec une structure C. Pour s'assurer que leur sérialisation prenne bien le nombre d'octets spécifié, les déclarer de la manière suivante :

```
struct __attribute__ ((__packed__))" <nom_de_la_structure> {
}
```

L'ensemble des données devront être représentées en big endian, c'est à dire l'ordre conventionnel pour l'ensemble des protocole réseaux TCP/IP : les bits de poids fort sont en premier : utiliser les primitives `htonl()` et `ntohl()`.

Pour simplifier, le serveur sera conçu pour servir une seule requête à la fois et un seul client à la fois, identifiée par le champs Handle des messages. Il n'est donc pas nécessaire d'écrire un serveur multi-processus ni d'utiliser les mécanismes tels que `select()` ou les descripteurs de fichiers non bloquants.

Important : Comme les sockets de type `SOCK_STREAM` sont utilisées pour transmettre un flux de données qui est agnostique au type de données transporté, elles ne préservent pas la séparation entre les différents structures qui composent la communication. Ceci veut dire que sur un flux composé d'une requête d'écriture longue de N octets telle que illustrée sur la figure suivante :

```
------------------->
|N|...|6|5|4|3|2|1|0|
------------------->
```

Un premier appel à `read(s,buffer,10)` sur la socket, peut renvoyer un nombre d'octet dans le buffer compris entre 0 et 10. Si on veut récupérer les N octets, il faudra s'assurer que la somme des octets reçus successivement dans le flux composent bien une requête en entier.

Exemple de communication :

1. Client envoie au serveur : `0x76767676, 0x0, 0x12345678, 0x1000, 0x400` : c'est une requête (`0x76767676`) de lecture (`0x0`), de handle `0x12345678`, à l'offset `0x1000` du fichier, de 1024 (`0x400`) octets.

2. Le serveur se positionne à l'offset 1024 du fichier passé en paramètre (cf `lseek(2)`), lit 1024 octets (cf `read(2)`) dans un buffer, puis renvoie la réponse suivante : `0x87878787,0x0, 0x12345678`, données lues (1024 octets). c'est une réponse (`0x87878787`), sans erreur (`0x0`), pour la requête `0x12345678`, qui contient 1024 octets.

Travail à rendre

La totalité du projet est à réaliser par groupe de 2 personnes. Elle devra impérativement être rendu avant le **12/6/2015** sous la forme d'un rapport de maximum **10 pages** et d'une vidéo de présentation du projet de maximum **10 min** comprenant une démonstration du projet, ainsi qu'une explication détaillée d'une fonction que vous trouvez techniquement intéressante dans l'étape numéro 2. Le projet comptera pour 50 % de la note finale de la matière.

Le rapport devra comprendre une partie expliquant la répartition du travail réalisé entre les deux personnes du binôme.