Programmation Système Avancée

TP3 – Système de fichiers

Objectif

Le but de ce TP est de continuer le développement de votre kernel en implémentant un système de fichiers très rudimentaire.

A la fin de ce TP, votre kernel sera ainsi capable de :

- lire un fichier et le charger en mémoire ;
- effacer un fichier ;
- déterminer si un fichier existe ;
- obtenir les informations d'un fichier ;
- itérer sur tous les fichiers du système de fichiers.

Pour parvenir à cet objectif, ce TP se décompose en deux parties distinctes.

La première partie est dédiée au développement des outils permettant de construire l'image du système de fichiers qui sera utilisée par votre kernel. L'image du système de fichiers ainsi créée sera passée à QEMU qui l'interprétera comme un disque physique. Pour manipuler cette image, il vous faudra implémenter les outils nécessaires : création d'une image vide, insertion de fichiers au sein de l'image, etc. Ces outils ne seront rien d'autre que des exécutables s'exécutant sur le système hôte, en l'occurence GNU Linux.

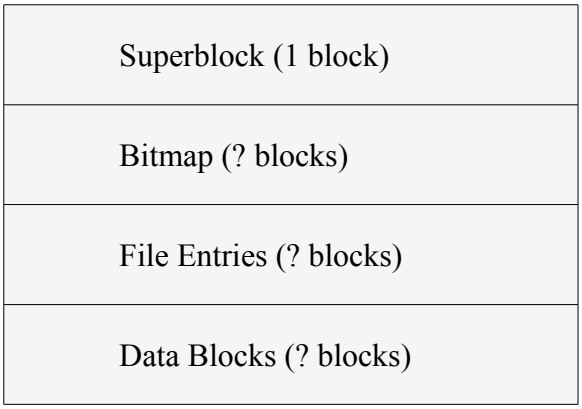
La deuxième partie du TP est dédiée à l'implémentation des fonctions systèmes permettant à votre kernel de manipuler le système de fichiers : itération sur les fichiers présents, lecture d'un fichier, etc.

Structure du système de fichiers PFS (Primitive File System)

L'étude des systèmes de fichiers est un sujet vaste pouvant aisément occuper un semestre entier.

Le but de ce projet est de créer un système de fichiers extrêmement simple illustrant les concepts et choix à faire lors de la conception et du développement d'un tel système.

Pour des raisons de temps et de simplicité, le système de fichiers que vous implémenterez ici est volontairement très limité et non hiérarchique. La structure d'un système de fichiers PFS (Primitive File System) est décrite ci-dessous :



L'unité de base du système de fichier est le block. La taille d'un block est définie par l'utilisateur et est toujours un multiple d'un nombre de secteurs. Il est considéré ici que la taille d'un secteur est toujours de 512 bytes. C'est presque toujours le cas aujourd'hui, même si physiquement ce n'est pas nécessairement vrai (les firmwares de périphériques exposent toutefois des secteurs logiques de 512 bytes).

Superblock

Le superblock fait une taille de 1 block et contient la signature du système de fichiers ainsi que ses caractéristiques principales, dans l'ordre :

- signature du système de fichiers (8 bytes) : PFSv0100
- nombre de secteurs par block (32 bits)
- taille du bitmap en blocks (32 bits)
- nombre de file entries (32 bits)
- taille d'une file entry en bytes (32 bits)
- nombre de data blocks (32 bits)

Bitmap

Le bitmap est un tableau de bits indiquant les blocks utilisés pour le stockage du contenu des fichiers. Le contenu des fichiers est stocké dans les data blocks. Chaque data block est représenté par exactement 1 bit dans le bitmap. De fait, 1 byte représente exactement l'état de 8 data blocks. Un bit à 0 signifie que le block est libre et un bit à 1 signifie que le block est utilisé. Afin de ne pas gaspiller des blocks inutilement, le bitmap a la taille minimum permettant de référencer tous les data blocks disponibles dans le système de fichiers.

File Entries

Chaque fichier est représenté par deux entités : les méta-données du fichier ainsi que le contenu de celui-ci. Une file entry stocke les méta-données du fichier alors que le contenu est stocké dans les data blocks. Les file entries de tous les fichiers sont stockées consécutivement juste après le bitmap. Le nombre de file entries est fixe et est choisi par l'utilisateur au moment de la création du système de fichiers. La structure d'une file entry est présentée dans la table ci-dessous :

File Name (32 bytes)	File Size (4 bytes)	Index of 1st data block (2 bytes)	Index of 2nd data block (2 bytes)	Index of 3rd data block (2 bytes)	...	Index of Nth data block (2 bytes)

Une file entry stock donc les informations suivantes :

- nom du fichier (32 bytes)
- taille (4 bytes)
- indices (2 bytes) des blocks constituant le contenu du fichier

La taille d'une file entry est spécifiée dans le superblock. Attention cependant : la taille d'une file entry doit pouvoir diviser (entièrement) la taille d'un secteur et doit être au minimum de 64 bytes !

32 bytes sont toujours réservés pour le nom du fichier, même si celui-ci est potentiellement plus court. Le nom suit la convention des chaînes de carctères en C (zéro terminal). Ainsi, la longueur maximum d'un nom de fichier est de 31 caractères. La taille d'un fichier est représentée sur 32 bits.

Les indices des data blocks indiquent où se trouve les différent blocks formant le contenu du fichier.

Il n'y a aucune contrainte quand à la localité des data blocks constituant le contenu du fichier. Ceux-ci peuvent être disséminés n'importe où sur le disque et n'ont donc nullement besoin d'être consécutifs ou ordonnés. Le premier indice valide pour un data block est la valeur 1, car un indice de 0 indique un block non utilisé. A noter que l'indice 1 référence le 2ème data block (du coup, le data block 0 n'est jamais utilisé). Chaque indice est encodé sur 2 bytes ce qui signifie qu'il nest pas possible d'adresser un block de donnée possédant un indice supérieur à 2^{16}-1.

Lors de l'effacement d'un fichier, le premier caractère du nom de la file entry est mis à 0 et les bits du bitmap référençant les blocks de données sont mis à 0. Il est donc inutile de toucher aux blocks de données eux-mêmes ce qui rend l'effacement beaucoup plus rapide, mais au détriment de la sécurité.

Data Blocks

Les data blocks sont les blocks de données stockant le contenu des fichiers. Ceux-ci se trouvent directement à la suite des file entries. Le contenu de chaque fichier est donc divisé en data blocks. Ce choix engendre un gaspillage important dans le cas de petits fichiers, car un fichier de seulement 1 byte nécessitera de toute façon un block de donnée entier sur le disque.

Partie 1 : Outils pour la gestion du système de fichiers

Pour le moment, votre kernel n'a aucun concept de système de fichiers. Il n'est d'ailleurs pas encore capable de lire un secteur sur le disque (chose qui sera faite dans la prochaine partie). Avant d'ajouter le support du système de fichiers au kernel, il est impératif de créer un tel système de fichiers.

Le but de cette première partie est de développer les outils nécessaires à la manipulation du système de fichiers PFS. En particulier, vous devrez développer une série d'outils permettant de :

1. Créer une image vide d'un système de fichiers PFS.
2. Ajouter à l'image du système de fichiers PFS un fichier présent dans le système d'exploitation hôte (GNU Linux).
3. Lister les fichiers présents dans l'image d'un système de fichiers PFS.
4. Effacer un fichier d'une image d'un système de fichiers PFS.

Ces outils doivent s'exécuter dans un système GNU Linux. La liste des commandes à implémenter est la suivante :

- `pfscreate img x y z`
  - créé un système de fichiers PFS vide dans le fichier image `img`
  - `x` spécifie la taille d'un block ; attention celle-ci doit être un multiple de la taille d'un secteur (512 bytes) ; exemple : 2048
  - `y` spécifie le nombre de file entries à créer ; à noter que pour des raisons de simplicité, chaque file entry fera 256 bytes
  - `z` spécifie le nombre de data blocks disponibles pour le stockage du contenu des fichiers.
- `pfsadd img file`
  - ajoute le fichier `file` au système de fichiers représenté par l'image `img`
- `pfslist img`
  - liste les fichiers présents dans le système de fichiers représenté par l'image `img`
- `pfsdel img file`
  - supprime le fichier `file` du système de fichiers représenté par l'image `img`

Le code source de ces utilitaires devra se trouver dans le répertoire `tools` de l'arborescence du projet. Tous comme pour le répertoire `kernel`, vous écrirez un makefile permettant de compiler ces utilitaires. La modularité du code étant primordiale, il est important de créer des fichiers définissant les structures et fonctions communes à chacun des utilitaires ci-dessus.

Rappelez-vous que les utilitaires *hexdump* ou *hexedit* sont vos amis, car ils permettent de facilement inspecter le contenu de fichiers binaires. Ainsi, vous pourrez accéder facilement à n'importe quel secteur de l'image de votre système de fichiers (par exemple, avec *hexedit* pour sauter à un offset spécifique du fichier courant : pressez entrée, puis indiquez un offset en bytes et pressez entrée à nouveau).

Partie 2 : Implémentation du support PFS dans le kernel

Accéder au système de fichiers PFS avec QEMU

Pour accéder au système de fichiers PFS nouvellement créé depuis votre kernel, il convient tout d'abord de l'indiquer à QEMU. Dans les TP1 et TP2, vous aviez créé une image ISO de votre système d'exploitation. Celle-ci est au format ISO-9660 qui est spécifiquement prévu pour être gravé sur CD-ROM ou DVD-ROM. Quand vous passez l'image à QEMU avec l'option `-cdrom image.iso`, celui-ci émule un CD-ROM ayant le contenu de l'image ISO passée en argument. Une fois la partie 1 du TP terminée, il est nécessaire d'indiquer à QEMU que la machine à émuler contient un disque dur connecté sur le premier contrôleur de disque. Ceci se fait avec l'option `-hda fs.img` où `fs.img` est une image de disque représentant les secteurs du premier disque dur de la machine.

Accéder au premier disque dur depuis le kernel

Comment peut-on accéder aux secteurs du premier disque dur depuis le kernel ? Le seul moyen de le faire est d'écrire un drivers permettant de gérer le protocole ATA. Afin de vous simplifier la vie et aussi parce que ce n'est pas le but de ce TP, je vous mets à disposition le code C permettant de lire et écrire des secteurs sur le disque dur connecté au premier contrôleur ATA. Ce code n'utilise pas le DMA et il est donc peu efficace, mais ce n'est toutefois pas un problème dans le cadre de ce TP. Vous pouvez trouver le code sur Cyberlearn dans l'archive `tp3.tar.gz`. Veuillez noter que ce code utilise les fonctions d'accès aux ports d'entrée/sortie implémentées lors du TP1.

Support de PFS dans le kernel

Le kernel ne connaissant rien au système de fichiers PFS, il est nécessaire d'en implémenter le support. Pour ce faire, vous créerez de nouveaux fichiers sources liés à l'implémentation du système de fichiers. L'implémentation des fonctions de base pour la manipulation des fichiers avec PFS implique au minimum les fonctions ci-dessous :

- `int file_stat(char *filename, stat_t *stat);`
  renvoie dans `stat` des informations liées au fichier passé en argument ; la structure `stat_t` doit au minimum contenir la taille du fichier (cf. plus bas). Retourne 0 en cas de succès et -1 en cas d'échec.
- `int file_read(char *filename, void *buf);`
  lit le contenu du fichier passé en argument dans `buf`. Le buffer dénoté par `buf` doit avoir été préalablement alloué par l'appelant. Retourne 0 en cas de succès et -1 en cas d'échec.
- `int file_remove(char *filename);`
  Supprime le fichier passé en argument. Retourne 0 en cas de succès et -1 en cas d'échec.

- `int file_exists(char *filename);`
  Retourne 1 si le fichier passé en argument existe et 0 dans le cas contraire.
- `file_iterator_t file_iterator`();
  Renvoie un itérateur permettant ensuite d'itérer sur tous les fichiers du système de fichiers grâce à la fonction `file_next`. Après appel à cette fonction, l'itérateur pointe sur le premier fichier (s'il y en a au moins un) du système de fichiers.
- `int file_next(char *filename, file_iterator_t *it);`
  Cette fonction permet d'itérer sur les fichiers du système de fichiers. L'appel à `file_next` renvoie 1 si l'itérateur `it` pointe sur le fichier courant. Dans ce cas, le nom du fichier courant est copié dans `filename`. Si l'itérateur a déjà itéré sur tous les fichiers, alors la fonction renvoie 0 et rien n'est copié dans `filename`. A noter qu'il est obligatoire d'initialiser l'itérateur `it` avec la fonction `file_iterator` avant d'appeler la fonction `file_next`.

Pour afficher les fichiers du système de fichiers, on écrira typiquement :

```
iterator_t it = file_iterator();
while (file_next(filename, &it)) {
     printf("%s\n", filename);
}
```

La fonction `file_stat` fait usage de la structure `stat_t` ci-dessous. Celle-ci contient uniquement le champs `size`, mais vous êtes libre d'y rajouter les méta-données de votre choix :

```
typedef struct {
        uint32_t size;
} stat_t;
```

Splash screen

Afin de rendre votre kernel un petit peu plus attractif, il est demandé que votre kernel affiche un « *splash screen* » à la manière de GNU Linux au moment de son démarrage. Vous pourrez par exemple afficher un logo en mode texte[1] ou alors un message en ASCII Art[2] affichant un message de bienvenue à l'utilisateur. Ce logo ou *splash screen* sera chargé depuis un fichier se trouvant dans votre système de fichiers et accessible par le kernel. Ainsi, il pourra être changé sans avoir à recompiler le kernel.

Make

Veillez à optimiser les dépendances de votre projet afin que lorsque l'utilisateur tape « `make run` », rien ne soit recréé ou recompilé inutilement. En particulier, l'image de votre système de fichiers ne devra pas être regénérée, à moins qu'un des fichiers la composant soit modifié (indice : relisez la théorie liée à `make`, en particulier la fonction `wildcard`).

Tests

Il est important que vous développiez des routines de tests permettant de vérifier que vos fonctions d'accès au système de fichiers fonctionnent correctement. Tout comme pour le TP1, développez un ensemble de procédures de tests permettant de vérifier la validité de votre implémentation.

1 http://www.glassgiant.com/ascii/
2 http://patorjk.com/software/taag

Remarques

Le plus facile est de commencer par développer les outils externes à la création de votre système de fichiers. Vérifiez que vos outils fonctionnent correctement avec *hexedit*. Une fois vérifié, vous pourrez alors commencer le support du système de fichiers PFS au sein de votre kernel.

Travail à rendre

Pour ce TP, vous me rendrez :

- Une archive au format ZIP comprenant l'arborescence complète de votre projet. Le projet étant un tout, votre archive doit contenir le code source complet que vous avez développé depuis le TP1.

Comme vous n'avez pas de rapport à me rendre pour ce TP, je tiens à ce que votre code soit extrêmement lisible et particulièrement bien commenté.

Le code doit respecter les consignes décrites dans le document **Consignes générales pour les travaux pratiques** sous la rubrique "Informations générales" sur la page Cyberlearn du cours.