Programmation Avancée

Projet : Système de fichiers

Simon Forest

I) Introduction

Un système de fichiers est une arborescence composée de nœuds. Un nœud est soit un fichier, soit un dossier. Chaque nœud a un nom. Un nœud qui est un dossier peut contenir plusieurs nœuds « enfants », c'est-à-dire qu'il peut contenir plusieurs autres dossiers et fichiers. Un nœud qui est un fichier ne contient pas de nœud enfant, mais a un contenu. Une arborescence a un nœud spécial, qui est le nœud racine. Ce dernier est un dossier, et l'arborescence du système de fichiers doit former un arbre dont la racine est le nœud racine.
Sous Linux, on peut afficher l'arborescence associée à un dossier avec la commande `tree` (à installer avec `apt` auparavant). Par exemple, on peut obtenir la sortie suivante en appelant `tree` sur un dossier nommé `racine` :

```
racine
|-- agenda.org
|-- divers
|   |-- foo.txt
|   |-- goo.txt
|-- listes
|   |-- à-faire.txt
|   |-- fruits.txt
|   |-- légumes.txt
|-- prog
    |-- projet
    |   |-- main.c
    |   |-- README.txt
    |-- tp
        |-- exo1.c
        |-- exo1.h
```

Dans cet exemple, on voit qu'il y a 16 nœuds, dont 6 dossiers et 10 fichiers (ici, ces derniers sont les nœuds dont les noms ont des extensions en `.txt`, `.org`, `.c` et `.h`). On voit ici que chaque nœud qui est un dossier a un certain nombre d'enfants. Par exemple, le nœud `racine`, qui est un dossier, a ici quatre nœuds enfants qui sont `agenda.org`, `divers`, `listes` et `prog`. Parmi ces quatre nœuds, seul `agenda.org` est un fichier, et les autres sont des dossiers.
Chaque fichier dans cet exemple a de plus un contenu qui n'est pas montré sur l'arborescence ci-dessus. Par exemple, le fichier `fruits.txt` a le contenu suivant :

```
- pomme
- banane
- fraise
```

Dans une arborescence, on peut localiser un nœud en utilisant des *chemins*. Il existe deux types de chemins : des chemins *absolus* et *relatifs*. Les chemins absolus localisent des nœuds par rapport à la racine. Suivant la convention Linux, ils commencent par `/`. Par exemple, dans l'arborescence ci-dessus, le chemin absolu du nœud `README.txt` est `/prog/projet/README.txt`. Les chemins relatifs localisent des nœuds par rapport à un dossier de l'arborescence. Il ne commencent pas par `/`. Par exemple, dans l'arborescence ci-dessus, le chemin chemin du nœud `README.txt` par rapport au dossier `prog` est `projet/README.txt`.
Les systèmes de fichiers sont normalement stockés sur des mémoires de long terme, comme les disques durs, afin que les données puissent être conservés même lorsque l'alimentation est débranchée. Dans ce projet, on simplifie : il s'agira juste d'écrire un système de fichier qui tient en mémoire vive (sur la RAM).

II) Bibliothèque

Dans un premier temps, il vous faudra écrire une bibliothèque permettant de manipuler votre système de fichiers. Par bibliothèque, on entend un ensemble de fichiers `.c` codant les fonctionnalités du système de fichiers, et un ou plusieurs `.h` décrivant les structures et les prototypes des fonctions écrites, et permettant ainsi leur utilisation par d'autres programmes.
Il vous faudra réfléchir à plusieurs questions pour écrire cette bibliothèque, notamment :

- comment représenter les nœuds du système de fichiers ?
- comment représenter les dossiers et les fichiers comme spécialisations des nœuds ?
- quels pointeurs mettre dans ces structures pour représenter correctement l'arborescence et / ou permettre des opérations efficaces ?

Votre bibliothèque devra au moins déclarer les structures suivantes :

- une `struct filesystem`, représentant un système de fichiers ;
- une `struct node`, représentant un nœud d'un système de fichiers ;
- une `struct directory`, contenant les données supplémentaires d'un nœud d'un système de fichiers qui est de plus un dossier ;
- une `struct file`, contenant les données supplémentaires d'un nœud d'un système de fichiers qui est de plus un fichier ;
- une `struct file_content`, décrivant le contenu d'un fichier.

Il vous faudra au moins implémenter les fonctions suivantes :

- `void filesystem_init(filesystem *fs)`, qui initialise un système de fichier avec un nœud racine ;
- `void filesystem_free(filesystem *fs)`, qui supprime un système de fichiers et libère la mémoire qu'il utilise ;
- `node* filesystem_get_root(filesystem *fsys)`, qui renvoie le nœud racine du système de fichiers ;
- `node* directory_find(node* dir, const char* name)` qui renvoie le nœud enfant du dossier `dir` qui s'appelle `name` ou `NULL` s'il n'existe pas (suivant vos choix d'implémentation, vous pouvez alternativement prendre `directory*` pour le type de `dir`) ;
- `node* directory_add_file(node* dir, const char* name)`, qui ajoute un fichier nommé `name` au nœud-dossier `dir` et renvoie un pointeur vers ce nœud en cas de succès et `NULL` sinon. Il ne devra pas y avoir d'autre nœud enfant du même nom avant l'ajout ;
- `node* directory_add_directory(node* dir, const char* name)`, qui ajoute un dossier nommé `name` au nœud-dossier `dir` et renvoie un pointeur vers ce nœud en cas de succès et `NULL` sinon. Il ne devra pas y avoir d'autre nœud enfant du même nom avant l'ajout ;

- `int directory_remove_node(node* dir, const char* name)`, qui supprime le nœud nommé `name` du dossier `dir`. Cette fonction renverra `0` si un nœud est en effet supprimé, et une valeur différente sinon ;
- `file_content* file_get_content(node* file)`, qui renvoie le contenu du nœud-fichier `file` ;
- `int file_set_content(node* file,file_content* content)`, qui change le contenu du nœud-fichier `file`. La fonction renverra `0` l'opération s'est bien passée et une autre valeur sinon ;
- `void file_print(node* file,int with_content)`, qui affiche joliment les données du nœud-fichier `file`. Le contenu ne sera affiché que si `with_content != 0` ;
- `void directory_print(node* dir,int depth,int with_content)`, qui affiche joliment les données du nœud-dossier `dir`. Si `depth == 1`, on n'affichera que les informations du dossier `dir`. Si `depth > 1`, on affichera les données des nœud enfants récursivement jusqu'à la profondeur `depth`. Si `depth <= 0`, on affichera les données des nœud enfants récursivement sans limite de profondeur. Le contenu des fichiers ne sera affiché que si `with_content != 0` ;
- `void filesystem_print(filesystem* fs,int depth,int with_content)`, qui affiche joliment le contenu d'un système de fichiers, en étant paramétré de façon similaire à précédemment.

Les noms des nœud devront être de taille strictement positives, sauf celui de la racine qui sera la chaîne vide. Ces noms ne contiendront pas le caractère `'\0'` ni le caractère `'/'`. Les contenus des fichiers pourront contenir n'importe quelle séquence d'octets. Préférentiellement, votre conception des structures doit faire intervenir des listes chaînées quelque part. Vous illusterez votre bibliothèque sur un exemple d'arborescence : vous coderez un `main` utiliseant la bibliothèque et créant un système de fichiers `fs` tel que `filesystem_print(fs,0,1)` affiche quelque chose d'analogue à

```
filesystem
+ directory: ""
 + file: "rootfile", size: 23, content: "This is a file at root."
 + directory: "dirA"
  + file: "fileAA", size: 16, content: "This is file AA."
  + file: "fileAB", size: 16, content: "This is file AB."
 + directory: "dirB"
  + file: "fileBA", size: 16, content: "This is file BA."
  + file: "fileBB", size: 16, content: "This is file BB."
```

Votre affichage des dossiers et des systèmes de fichiers peut bien évidemment être différent et afficher plus de choses, mais il devra au moins faire ressortir la structures arborescente par des indentations comme ci-dessus.

III) Terminal

Dans un deuxième temps, il s'agira d'écrire un terminal simplifié permettant d'interagir avec un système de fichiers de votre bibliothèque. Comme un terminal normal, ce terminal sera en permanence dans un dossier « courant » de l'arborescence. Le chemin absolu de ce dossier pourra être récupéré avec la commande `pwd`. D'autres commandes commumes doivent être présentes, comme :

- `cd`, qui permet de changer de dossier courant ;

- `cat`, qui permet d'afficher le contenu d'un fichier ;
- `touch`, qui permet de créer un fichier ;
- `mkdir`, qui permet de créer un dossier ;
- `ls`, qui permet de lister le contenu du dossier courant ;
- `tree`, qui permet d'afficher l'arborescence à partir du dossier courant.

On ne demande juste des versions de base de ces commandes, sans option, de façon à pouvoir modifier, afficher et se déplacer dans l'arborescence de façon convenable. Noter cependant que certaines de ces commandes prennent logiquement des arguments. On créera en plus des commandes ci-dessus une commande `edit`, qui permet d'éditer de façon sommaire un fichier (par exemple, on affiche l'ancien contenu et on demande le nouveau).

IV) Bonus

On pourra chercher à implémenter des fonctionnalités supplémentaires parmi les suivantes :

- autoriser les chemins relatifs et absolus comme arguments des commandes du terminal ;
- permettre d'importer des dossiers et des fichiers du « vrai » système de fichiers vers le système de fichiers en mémoire, et symétriquement d'exporter des dossiers et des fichiers du système de fichiers en mémoire vers le « vrai » système de fichiers.
- ajouter des options aux commandes ou rajouter d'autres commandes utiles (par exemple, des commandes comme `find` ou `grep` qui permettent de chercher suivant un nom de fichier ou un contenu) ;
- ajouter un système de permissions « à la Linux », où un nœud a un propriétaire et un groupe, et des autorisations en lecture et en écriture pour le propriétaire, les membres du groupe, et le reste des utilisateurs.

Vous pouvez aussi coder vos propres idées de fonctionnalités supplémentaires. Pour être sûr qu'elles vous rapporteraient des points supplémentaires, n'hésitez pas à demander.

V) À rendre

Il s'agira de rendre une archive `.tar` ou `.zip` contenant

- le code (fichiers `.c` et `.h`) ;
- un `Makefile` permettant de compiler votre code ;
- un fichier `.txt` ou `.pdf` ou Word ou Writer décrivant les fonctionnalités que vous avez implémentées et incluant de plus tout commentaire jugé pertinent pour la correction.

Ce projet est à faire seul ou en binôme. Le partage de code entre différents groupes n'est pas autorisé (le constat de partage de code pourra faire descendre la note des groupes impliqués à 0).