Titre du projet: Mars Escape

Chapitre 1: Le vent solaire

Planète Mars, 12/12/2089 20:00

Radio: Commandant, merci de confirmer la réception de notre dernier bulletin d'alerte "vent solaire" ...
Vous: Alerte confirmée, je ne sais ce qu'il a le soleil en ce moment mais il doit être vraiment en colère ...
Radio: Pas d'inquiétude commandant, tous les équipements de la station sont conçus pour résister aux vents solaires; bon courage commandant et au mois prochain pour refaire le point et parler de votre avancement !

Vous: Ok ! ... Heu...attendez....

... Fin de la transmission ...

Vous: Mince je n'ai pas eu le temps de négocier mes heures sup...comme toujours je me fais avoir...je vais finir par rejoindre un syndicat...

Planète Mars, 13/12/2089 09:35

Vous: Quelle heure est-il ? Que se passe-t-il ? plus de lumière dans la station; l'ordinateur de bord disjoncte complètement... Tous les équipements sont éteints. Ca doit être le vent solaire! Je vais rebooter l'ordinateur central et lancer un diagnostic en toute urgence...

(Vous tapotez sur l'ordinateur de bord pour lancer le diagnostic)

Vous: Je n'arrive pas à le croire, l'électronique de la station a été sérieusement endommagé! C'est de l'électronique de contrebande, je le savais ! Je vais prendre contact avec la Terre pour signaler que j'ai un sérieux problème et que je ne pourrai pas tenir plus de trois jours ici!

(vous vous rendez à la station radio)

Vous: Station Mars Science 1 appelle le centre de Commandement de la Terre...répondez....
Radio: ....
Vous: Station Mars Science 1 appelle le centre de Commandement de la Terre...répondez....

Radio: ....

Vous: Pfff... la radio est endommagée également...je crois que je n'ai plus le choix je vais activer la capsule d'évacuation d'urgence pour rejoindre la Terre

(Vous cherchez votre xDroidPhone)

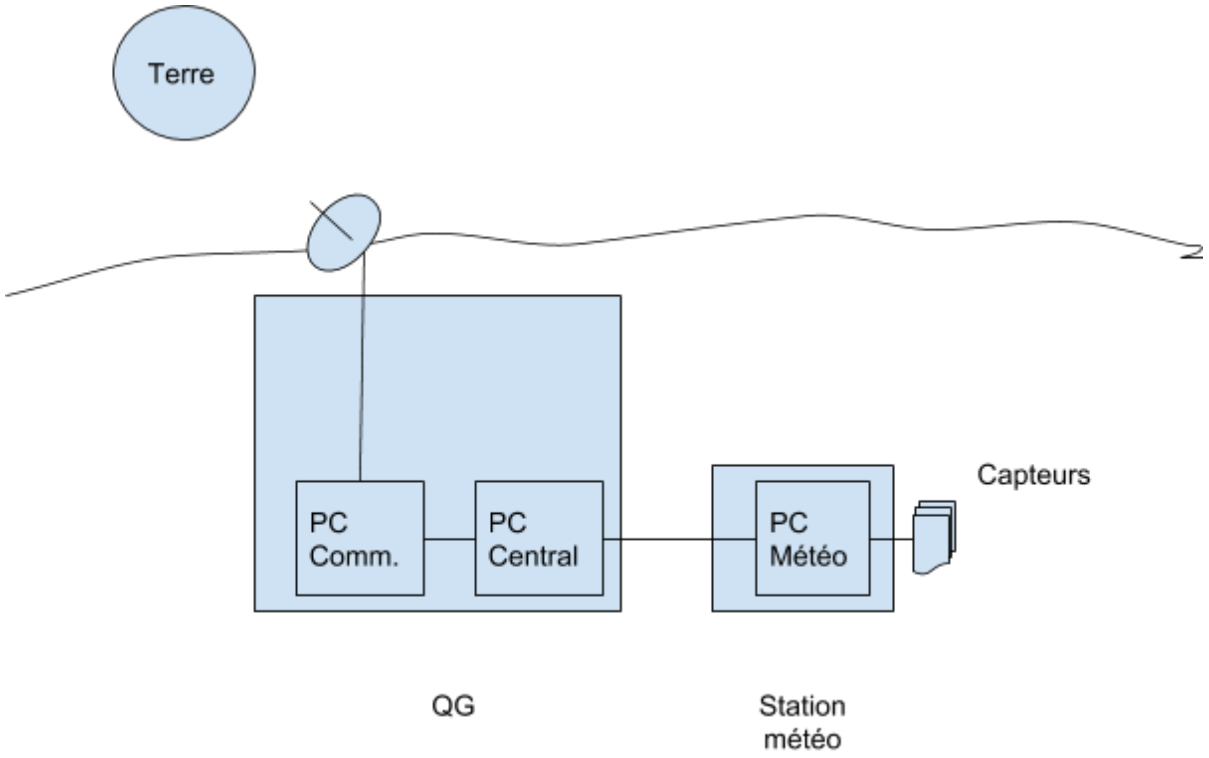
Vous: Heureusement que j'ai noté le code d'activation de la capsule dans ma mémoire externe...et dire que nos ancêtres utilisaient leurs cerveaux pour retenir des informations; quel gâchis...

(L'écran du xPhoneDroid reste inactif)

Vous: il ne s'allume pas il est également endommagé...je dois absolument avoir le code d'activation de la capsule sinon je vais crever ici !

Vous: Voyons voir le plan de la station:

Plan de la station



Vous: alors la station est composée d'une station météo autonome avec un PC linux et de différents capteurs. La station météo est reliée par une connexion filaire au PC de communication qui transmet périodiquement les données vers la terre. D'après le diagnostic

de l'ordinateur de bord, le PC de communication semblerait OK; par contre la station météo est down....Je vais lancer un diagnostic avancé sur la station météo.....

Ordinateur: "La librairie "gestionnaire de fichiers" est corrompu sur la station météo"

Vous: OK je vois le problème; si j'arrive à mettre en place un nouveau gestionnaire de fichiers sur la station météo elle sera capable d'envoyer des bulletins météo et je suis sûr que je pourrai hacker les messages météo pour transmettre un SOS à la terre et demander le code de la capsule...

Mission 1: Implanter un nouveau système de fichiers pour la station météo

---

Objectifs:

Vous devez construire une bibliothèque, myLibFS, qui va réaliser les fonctions de base d'un système de fichiers. Les programmes de la station météo vont utiliser cette librairie et pourront ainsi écrire et lire les données sur le disque.
La station météo utilise un disque particulier. L'API de ce disque ainsi que la librairie libDisk sont fournis.

Spécifications de myLibFS:

Nous allons décrire ici l'API du système de fichier afin qu'il puisse être utilisé par les programmes externes. Du point de vue d'un programme, le système de fichier sera considéré comme une librairie avec laquelle il sera lié.
L'API de myLibFS est divisée en trois parties:

- des fonctions génériques comme l'initialisation et la gestion des erreurs
- gestion des fichiers standards
- gestion des répertoires

Fonctions génériques du FS

- Gestion des erreurs: Pour signaler les erreurs, les fonctions devront retourner une valeur spécifique (-1 par exemple) et affecter un code d'erreur à la variable globale **osErrno.** Ceci va permettre aux programmes qui utilisent votre libraire de détecter qu'une erreur s'est produite et surtout avoir des informations sur la nature de l'erreur

- Initialisation

- **int FS_Boot(char *path) :** Cette fonction devra être appelée une seule fois avant de pouvoir utiliser les autres fonctions du FS. Le paramètre 'path' est le chemin (dans la machine *hôte*) vers l'image du disque utilisée par myLibFS. Si ce fichier n'existe pas, il devra être créé et initialisé correctement pour être utilisé par myLibFS (formattage). En cas de succès cette fonction retourne 0 et en cas d'erreur, elle retourne -1 et met la valeur E_GENERAL dans osErrno.
- **int FS_Sync() :** Cette fonction va assurer l'écriture de toutes les données sur le disque. En effet, pour améliorer les performances, vous pourrez retarder les opérations d'écriture (souvent coûteuses) jusqu'à l'appel de FS_Sync. Cette fonction retourne 0 en cas de succès et la valeur -1 en cas d'erreur, auquel cas la variable osErrno aura la valeur E_GENERAL.

API d'accès aux fichiers

Spécification des chemins: Nous allons prendre l'hypothèse que tous les chemins sont absolus. Par conséquent, tous les chemins vont commencer par le répertoire racine "/". Les noms de fichiers sont limités à 16 caractères (15 octets pour le nom et un octet pour le caractère de fin de string '\0') Enfin, la taille maximale d'un chemin est fixée à 256 caractères.

Spécification de l'API:

- **int File_Create(char *file)** : crée un nouveau fichier désigné par *file*. Si le fichier existe, vous devez retourner la valeur -1 et mettre osErrno à E_CREATE.
- **int File_Open(char *file) :** ouvre le fichier désigné par *file* et retourne un descripteur (un entier >= 0) qui sera utilisé pour lire/écrire sur ce fichier. Si le fichier n'existe pas ou que le nombre maximum de fichiers ouverts est atteint, la valeur -1 est retournée et la variable osErrno aura comme valeur E_NO_SUCH_FILE et E_TOO_MANY_OPEN_FILES respectivement.
- **int File_Read(int fd, void *buffer, int size) :** cette fonction va lire *size* octets du fichier désigné par le descripteur *fd* et les copier dans *buffer*. La lecture sera fera à partir de la position actuelle du pointeur du fichier (i.e la tête de lecture/écriture); la position du pointeur de fichier sera mise à jour après la lecture. Si le fichier n'est pas ouvert, alors la fonction retourne -1 et osErrno aura comme valeur E_BAD_FD. Si le fichier est ouvert, alors le nombres d'octets effectivement lus sera retourné. Si le pointeur est à la fin du fichier alors la fonction retournera la valeur 0.

- **int File_Write(int fd, void *buffer, int size) :** cette fonction va écrire *size* octets à partir de *buffer* dans le fichier désigné par *fd.* L'écriture se fera à partir du pointeur du fichier qui sera mis à jour après l'opération d'écriture. Voici la description des retours possibles pour cette fonction:

    - Si le fichier n'est pas ouvert alors cette fonction devra renvoyer la valeur -1 et mettre le code erreur E_BAD_FD dans la variable osErrno.
    - En cas de succès, le nombre d'octets écrits sera retourné.
    - En cas d'écriture partielle (espace insuffisant par exemple), alors la fonction retournera -1 et mettra la valeur E_NO_SPACE dans osErrno.
    - Si la taille du fichier dépasse la taille max, alors la valeur -1 est retournée et le code E_FILE_TOO_BIG sera mis dans osErrno.
- **int File_Seek(int fd, int offset) :** cette fonction va modifier la valeur du pointeur de fichier désigné par *fd.* La position du pointeur de fichier sera calculée comme un décalage de *offset* octets à partir du début du fichier. Les codes retours de la fonction sont:
    - Si le décalage *offset* est plus grand que la taille du fichier ou négatif alors la fonction retourne -1 et osErrno prend la valeur E_SEEK_OUT_OF_BOUNDS
    - Si le fichier n'est pas ouvert, alors la fonction retourne -1 et osErrno prend la valeur E_BAD_FD.
    - En cas de succès, la fonction retourne la valeur du pointeur de fichier.

- **int File_Close(int fd) :** cette fonction ferme le fichier désigné par le descripteur *fd.* Si le fichier n'est pas ouvert alors la valeur -1 est retournée et osErrno prend la valeur E_BAD_FD. En cas de succès, cette fonction retourne la valeur 0.
- **int File_Unlink(char *file) :** cette fonction va supprimer le fichier désigné par *file.* Il faudra supprimer le nom du fichier du répertoire contenant et libérer tous les blocs de données et inodes utilisés. Si le fichier n'existe pas, la fonction retourne -1 et met E_NO_SUCH_FILE dans osErrno. Si le fichier est actuellement ouvert alors la valeur -1 et renvoyée et osErrno aura comme valeur E_FILE_IN_USE. En cas de succès, cette fonction retourne la valeur 0.

L'API de gestion des répertoires:

- **int Dir_Create(char *path) :** création d'un nouveau répertoire désigné par *path*. La création d'un répertoire se fera en deux étapes: i) création d'un fichier de type 'directory'; ii) ajout d'une nouvelle entrée dans le répertoire contenant avec le nom et le numéro d'inode du nouveau répertoire. En cas d'erreur, la valeur -1 est renvoyée et osErrno est mise à E_CREATE.
- **int Dir_Size(char *path)** : retourne la taille en octets du répertoire désigné par *path.* Attention, ce n'est pas la taille des fichiers contenus dans le répertoire mais la taille de la structure du répertoire déterminée par le nombre d'entrées. Cette fonction est utile pour avoir la taille des entrées avant d'utiliser la fonction de lecture Dir_Read
- **int Dir_Read(char *path, void *buffer, int size) :** Cette fonction va lire le contenu d'un répertoire qui sera copié dans le segment mémoire à partir de *buffer.* Il s'agit de lister toutes les entrées du répertoire. Chaque entrée est un tableau de 20 octets: 16 octets pour le nom du fichier ou répertoire contenu et 4 octets pour coder le numéro

d'inode de cette entrée. *size* est la taille max du *buffer*; si cette taille est insuffisante pour contenir toutes les entrées, la fonction retourne la valeur -1 et osErrno prend la valeur E_BUFFER_TOO_SMALL. En cas de succès, le nombre d'entrées sera retourné.

- **int Dir_Unlink(char *path) :** Cette fonction va supprimer le répertoire désigné par *path.* Les blocs mémoires ainsi que l'inode seront libérés ainsi que l'entrée dans le répertoire père. Si le répertoire n'est pas vide alors la fonction retourne -1 et met le code erreur E_DIR_NOT_EMPTY dans osErrno. Si *path* désigne le répertoire racine ("/") racine, alors la fonction retourne -1 et met le code erreur E_ROOT_DIR dans osErrno.

Guide de réalisation du système de fichier

Le disque:

Vous allez avoir accès à un émulateur de disque. Un disque est un tableau de mémoire qui va conserver les données de façon persistante. Ce tableau est décomposé en 10000 secteurs de même taille (512 octets par secteur). Le secteur est l'unité d'échange avec le disque pour les opérations de lecture et d'écriture. L'image entière du disque sera représentée avec un fichier standard dans le système hôte Linux. L'implémentation de l'émulateur du disque est disponible sous la forme d'une librairie et d'un fichier d'entête à inclure dans vos programmes.

Pour ce projet, nous proposons de fixer le nombre max d'inodes à 8192 inodes.

Voici le résumé des paramètres du disque et du FS:

Taille Secteur	512 octets
Nb Secteurs	10000
Nb inodes	8192

Attention 1: le code source de l'émulateur de disque est donné à titre indicatif pour les curieux dans le fichier "Disque.c" et il ne doit pas être modifié ! Le seul fichier à modifier sera le fichier myLibFS.c.
Attention 2: Dans le fichier myLibFS.c vous devez utiliser les fonctions de l'émulateur de disque. Il est interdit par exemple d'utiliser les fonctions open, close, read, write du système hôte.

Les structures de données:

Sur le disque vous devez sauvegarder des (méta) informations sur votre système de fichiers. Ceci nous permettra de savoir que le disque utilise bien votre FS. Vous pouvez utiliser un

superblock qui occupera une adresse connue par convention (par exemple le premier bloc du disque) Dans le 'superblock' vous pouvez mettre un 'magic number' (un entier) qui va identifier votre FS.
Cette astuce vous permet de vérifier que c'est bien un disque qui utilise votre FS: après chargement de l'image disque, vous devriez retrouver le magic number à l'endroit attendu.

Concernant les blocs, vous avez besoin de deux types: des blocs inode et des blocs de données.
L'inode doit contenir les informations suivantes:

- le type de fichiers (répertoire ou fichier standard) (codé avec un entier)
- sa taille en octets (codé sur un entier)
- une table (de taille fixe) de pointeurs (i.e adresse disque codée par un entier) vers les blocs de données. Nous pouvons prendre l'hypothèse qu'un fichier ne peut pas contenir plus de 30 blocs de données (tableau de 30 entiers sera donc nécessaire)

Pour simplifier la gestion du FS, nous considérons que les blocs de données sont de même taille que les secteurs du disque soit 512 octets.

Attention: Un secteur du disque (512 octets) peut contenir plusieurs inodes (128 octets par inode)

Afin de déterminer si un inode ou un bloc de données est libre, vous pouvez utiliser deux bitmaps à sauvegarder dans le superblock. Il s'agit d'un tableau de 'bits' où l'index indique le numéro du bloc et le contenu sa disponibilité (0: libre et 1: occupé)

Sachant que le disque a une capacité limitée et connue (10000 secteurs de 512 octets) la taille des deux bitmaps sera déterminée par le nombre max d'inodes gérés par l'OS et le nombre de blocs de données.

Pour résumer, voici le schéma d'organisation du disque:

1. le superblock qui contient un entier magicnumber et un espace libre pour aligner les adresses
2. bitmap pour la disponibilité des inodes
3. bitmap de disponibilité des blocs de données
4. un tableau d'inodes
5. le reste du disque sera composé de blocs données.

En utilisant les paramètres de notre FS nous obtenons le schéma de mémoire suivant:

@	Taille Octets	Description
0	508	PADDING
508	4	MAGIC NUMBER
512	1024	INODE BITMAP
1536	1024	DB BITMAP
2560	1048576	INODE TABLE
1051136	4068864	DB TABLE
5120000		

Amorçage du système:

Au démarrage du système, vous devez fournir le nom de l'image disque.
Si le fichier existe, il faudrait alors vérifier qu'agit bien d'une image disque de votre FS: vous pouvez essayer de retrouver votre magicnumber (i.e charger le bon secteur et lire un entier au bon offset)
Si le fichier image n'existe pas, cela veut dire que nous souhaitons créer un nouveau disque et une initialisation (formatage du disque) est indispensable. Pour initialiser le disque, vous devez créer correctement le superblock (i.e magicnumber, inode bitmap et db bitmap) et ensuite créer le répertoire racine qui aura l'inode 0. N'oubliez pas d'écrire effectivement sur le disque avec l'opération FS_Sync().

Conversion chemin vers numéro d'inode:

L'utilisateur du FS identifie un fichier par chemin (*path*) Nous avons pris l'hypothèse que tous les chemins sont absolus: Ils commenceront par la racine ("/")
Afin de retrouver l'inode du fichier à partir d'un chemin, vous devez faire un parcours depuis la racine en passant par tous les répertoires intermédiaires:

1. Pour chaque répertoire, nous allons lire son contenu afin de retrouver le bon sous-répertoire en utilisant son nom;
2. L'entrée du sous-répertoire nous donne également son numéro d'inode. L'opération (1) est donc répétée récursivement jusqu'à récupérer l'inode du fichier final.

Ouverture des fichiers:
Pour ouvrir un fichier vous devez chercher son inode avec la conversation chemin-inode. Afin d'éviter de répéter cette recherche, nous proposons de maintenir certaines informations dans une table de fichiers ouverts. L'index de cette table représente le descripteur de fichier (fd) qui sera renvoyé au processus qui a fait la demande d'ouverture. A chaque demande d'ouverture, une nouvelle entrée sera utilisée dans la table des fichiers ouverts (cela veut dire un nouveau descripteur sera donné pour **chaque** open)

Nous allons limiter le nombre de fichiers simultanément ouverts à 256. La table des fichiers ouverts aura donc une taille de 256 entrées.

Séquence de réalisation du projet:

Vous être libre de choisir l'ordre de réalisation des fonctions définies dans LibFS.h; vous pouvez également créer des fonctions intermédiaires privées.

Voici une proposition de séquence pour réaliser le projet:

1. Implantation de la fonction: int FS_Boot(char *path);
2. Implantation des fonctions des répertoires:
    a. int Dir_Create(char *path);
    b. int Dir_Size(char *path);
    c. int Dir_Read(char *path, void *buffer, int size);
    d. int Dir_Unlink(char *path);
3. Implantation de la fonction de recherche d'un inode à partir d'un path
4. Implantation des fonctions de fichiers:
    a. int File_Create(char *file);
    b. int File_Unlink(char *file);
    c. int File_Open(char *file);
    d. int File_Read(int fd, void *buffer, int size);
    e. int File_Write(int fd, void *buffer, int size);
    f. int File_Close(int fd);
    g. int File_Seek(int fd, int offset);