Projet OS (2) : couche périphériques matériels

1. Introduction

Le but du projet OS est d'explorer les fondements d'un système d'exploitation simple pour ARM en le réalisant soi-même (do it yourself :-)

Le logiciel est scindé en deux parties :

- Le **code applicatif** qui s'exécute en mode **USER**.
  - Plusieurs **tâches** (*processus*) peuvent s'exécuter en pseudo-parallèle.
  - L'accès de la couche applicative aux services du système d'exploitation (création de tâche, de sémaphore, accès au matériel, …) se fait via les **appels systèmes**.
  - Les interruptions sont autorisées.
  - Lors de l'appel de la fonction `main`, le processeur se trouve en mode **USER** et les interruptions sont inhibées. Une initialisation minimale du système a été effectuée.
- Le **système d'exploitation** composé :
  - du **noyau** qui s'exécute en mode privilégié (**SUPERVISOR/IRQ**)
    - Il implémente le code et met en place les structures de données nécessaires pour offrir un certain nombre de services à la couche applicative. Ces services sont invoqués par l'application utilisateur via des appels systèmes. Les appels systèmes sont des opérations atomiques (non interruptibles).
    - Il gère la **commutation de tâches** et décide de la tâche applicative qui peut utiliser le processeur à un instant donné.
    - Il gère les **interruptions** et peut provoquer une commutation des tâches applicatives en réponse au traitement d'une interruption.
  - d'une **couche driver** qui s'exécute en mode **USER** qui permet de la communication entre la partie applicative et le matériel à travers une interface standard. On s'appuie sur les outils de synchronisation fournis par le noyau (sémaphores) pour offrir un fonctionnement adéquat.

Il s'agit de la deuxième phase de développement. Par commodité, le projet source disponible sur Moodle fournit, sous la forme de la bibliothèque `kernel/libkernel.a`, toutes les fonctionnalités développées pendant la première partie. Vous êtes cependant libres d'utiliser les fichiers sources (les vôtres) complétés pendant la première partie.

Cette partie va permettre de développer les éléments suivants :

– couche VFS (Virtual File System) qui implémentera la notion de fichier et de répertoire de manière générique avec une interface d'entrée-sortie (IO) standardisée (open, close, read, write, ioctl, lseek, readdir),

– intégration de l'accès aux périphériques de manière uniforme via la couche VFS,

– accès à une carte SD sur bus SPI et interfaçage avec un système de fichiers Microsoft® FAT32, et intégration du support pour FAT32 dans la couche VFS,

– intégration de l'accès aux périphériques dans la hiérarchie de répertoires du système de fichiers.

La ligne de conduite sous-jascente est toujours qu'il ne faut permettre à une tâche de s'exécuter (d'utiliser le CPU) que dans la mesure où elle a quelque chose "d'utile" à faire. Ainsi, une tâche qui attend que l'on ait appuyé sur un bouton ne doit pas continuer à s'exécuter (pas de scrutation).

Le répertoire du projet est organisé de la manière suivante :

```
/répertoire racine du projet/
 +- Makefile                        : fichier de construction du projet
 +- include/
 |  +- typedefs.h                   : redéfinition de types
 |  +- oslib.h                      : liste des appels systèmes supportés par le noyau
 |  +- termio.h                     : interface terminal série
 |
 +- user/
 |  +- main.c                       : code utilisateur (tests, application finale).
 |
 |                          C O D E   U T I L I S A T E U R
 | ------------------------------------------------------------------------------
 |                                C O D E   N O Y A U
 |
 +- kernel/ (code portable, exécution en mode SUPERVISEUR)
 |  +- syscall_dispatch.c : dispatcher d'appels systèmes
 |  +- list.[ch]    : fonction de manipulation de listes circulaires
 |  +- kernel.[ch] : implémentation du noyau
 |
 +- oslib/ (exécution en mode USER)
    +- kernellib.c ou libkernel.a  : fichier contenant la couche d'appels systèmes côté
    |                                utilisateur
    +- vfs.[ch]                    : couche VFS (Virtual File System)
    +- fat.[ch]                    : driver FAT32
    +- termio.c                    : gestion terminal sur liaison série
    +- drivers/ (code spécifique à l'architecture matérielle)
       +- target.[ch]              : interface générique avec le noyau (init + drivers
       |                             matériels)
       +- config/sram.lds          : config pour le placement mémoire
       +- startup/board_cstartup.s : code startup + sauvegarde/restitution de contexte
       |                             pour les modes SVC/IRQ
       +- include/                 : interface pour les accès au matériel à bas niveau
       |  +- cat1025.h             :   interface pour l'EEPROM sur le bus I2C
       |  +- diskio.h              :   interface de bas niveau carte SD via SPI
       |  +- eint.h                :   interface pour le bouton P0.14
       |  +- i2c.h                 :   interface bus I2C
       |  +- io.h                  :   interface GPIO + boutons + leds
       |  +- irq.h                 :   interface contrôleur d'interruptions
       |  +- lcd.h                 :   interface écran LCD
       |  +- lm75.h                :   interface capteur de température LM75
       |  +- timer.h               :   interface timer
       |  +- uart.h                :   interface UART
       |
       +- lib/                     : implémentation pour les accès au matériel de bas
                                     niveau
```

2. Couche VFS

Le rôle de la couche VFS est d'offrir à l'utilisateur du système d'exploitation un accès au matériel de manière abstraite via une interface standardisée (`open`, `close`, `read`, `write`, `ioctl`, `lseek` et `readdir`) permettant de manipuler les notions de fichier et de répertoire sans se soucier des détails d'implémentation.

Chaque fichier ou répertoire est caractérisé par un chemin d'accès dans une arborescence de répertoires. La fonction `int open(char *path, int flags)` permet d'associer à un nom de fichier ou de répertoire un descripteur de fichier (`fd` : *file descriptor*) qui est fourni en paramètre de retour (-1 si erreur). Ce descripteur de fichier[1] est utilisé par toutes les autres fonctions de manipulation des fichiers ou répertoires :

```
/* open : returns a file descriptor for path name */
int open(char *path, int flags);

/* close : close the file descriptor */
int close(int fd);

/* read : read len bytes from fd into buf, returns actually read bytes */
int read(int fd, void *buf, size_t len);

/* write : write len bytes from buf to fd, returns actually written bytes  */
int write(int fd, void *buf, size_t len);

/* ioctl : set/get parameter for fd */
int ioctl(int fd, int op, void** data);

/* lseek : set the offset in fd */
int lseek(int fd, unsigned int offset);

/* readdir : iterates over directory fd, returns 0 when ok, -1 else */
int readdir(int fd, DIR **dir);
```

Côté VFS, pour chaque fichier ou répertoire ouvert, on alloue dynamiquement une structure `FileObject` (`oslib/vfs.h`) :

```
typedef struct _FileObject FileObject;

struct _FileObject {
        char *          name;           /* name of the file */
        unsigned int    flags;          /* file characteristics */
        unsigned int    offset;         /* offset in file */
        FIL *           file;           /* specific data for file io */
        DIR *           dir;            /* specific data for dir io */
        Device *        dev;            /* device driver */
};
```

Cette structure permet de caractériser un fichier ou un répertoire en conservant les informations :

– `name` : le nom du fichier,

– `flags` : un champ de bits (voir `include/oslib.h`) indiquant le type d'accès souhaité (O_READ ou O_WRITE pour nous) et le type d'objet dont il s'agit :

  * `F_IS_DIR` : il s'agit d'un répertoire

[1] le terme *descripteur de fichier* pour l'entier renvoyé par `open` est un peu abusif. C'est plutôt un index qui permet d'accéder au descripteur de fichier de structure *FileObject* dans la table `opened_fds`.

* `F_IS_ROOTDIR` : c'est le répertoire racine "/"
* `F_IS_DEVDIR` : c'est le répertoire spécial "/dev"
* `F_IS_FILE` : il s'agit d'un fichier
* `F_IS_DEV` : c'est un fichier de périphérique matériel

– `offset` : utilisé pour les accès en lecture et écriture pour connaître le point de départ.

– `file` et `dir` : informations supplémentaires pour gérer des fichiers et répertoires qui existent physiquement sur un support de masse.

– `dev` : pointeur vers le descripteur du pilote de périphérique qui implémente les accès au matériel : une structure **Device**.

```
typedef struct _Device Device;

struct _Device {
        char        name[MAX_DEV_NAME_LEN];     /* name of device */
        int         refcnt;                     /* reference count */
        Semaphore * mutex;                      /* mutex semaphore */
        Semaphore * sem_read;                   /* blocking read semaphore */
        Semaphore * sem_write;                  /* blocking write semaphore */
        int         (*init)(Device *dev);       /* device initialization on OS init */
        int         (*open)(FileObject *f);             /* open device */
        int         (*close)(FileObject *f);    /* close device */
        int         (*read)(FileObject *f, void *buf, size_t len);  /* read from device method */
        int         (*write)(FileObject *f, void *buf, size_t len); /* write to device method */
        int         (*ioctl)(FileObject *f, int op, void **data);   /* set/get device parameters */
};
```

Chaque objet Device est associé à un périphérique physique et contient des méthodes qui seront utilisées pour accéder à ce périphérique. On peut ainsi utiliser des protocoles de communication différents pour lire des données, par exemple, provenant du port série et provenant d'une carte SD.

A l'initialisation de l'OS (avant l'appel de la fonction main) toutes les méthodes `init` des périphériques (si elles sont définies) sont appelées.

a. Compléter dans `oslib/vfs.c` la fonction `open`. Cette fonction renvoie le *descripteur de fichier* associé à la structure *FileObject* qui représente le fichier pour l'OS. Le descripteur de fichier est obtenu à partir de la table (`opened_fds`) des descripteurs de fichiers ouverts. On recherche le premier descripteur de fichier libre (entrée nulle) dans la table. Il faut alors allouer et initialiser la structure `FileObject`.

Puis, on traite le cas particulier du répertoire `/dev` qui est un répertoire virtuel, qui sera visible dans l'arborescence sans exister physiquement sur un support de masse : on ajoute, pour le champ `flags` l'attribut `F_IS_DEVDIR` et on renvoie le descripteur de fichier.

Si le nom passé en paramètre n'est pas `/dev`, il faut déterminer à quel **device** physique il faut s'adresser pour réaliser les opérations d'entrées-sorties. Tous les *devices* supportés (structures `Device`) sont enregistrés dans le tableau `device_table` (défini dans `oslib/drivers/target.c`). La fonction `dev_lookup` (dans `oslib/vfs.c`) permet de parcourir cette table à la recherche d'un nom de périphérique (ex, `/dev/dev_test`) et renvoie un pointeur sur la structure `Device` associée. Il faut initialiser le champ `dev` avec la valeur renvoyée par `dev_lookup`, puis, on exécute la méthode `open` (qui renvoie un booléen 0/1) *si elle existe*. Si tout s'est bien passé, on renvoie le descripteur de fichier, sinon -1 après avoir libéré les ressources.

b. Compléter la fonction `close` qui libère les ressources après exécution de la méthode du device associé.

c. Compléter les fonctions `read`, `write` et `ioctl` qui se contentent d'appeler la méthode du device associé.

d. Tester le fonctionnement de la couche VFS à l'aide du code `usr/tp2_ex1.c`. Analyser le code de `dev_test` défini dans `oslib/drivers/target.c`. Justifier l'utilisation du mutex.

3. Accès aux coupleurs périphériques

Pour chaque coupleur périphérique (les leds, un bouton poussoir, l'UART, …), une structure **Device** qui contient les méthodes pour accéder au périphérique est définie dans `oslib/drivers/target.c`.

3.1. Coupleurs simples

a. Implémentation du driver pour contrôler les leds : objet `dev_leds` (dans `oslib/drivers/target.c`). Compléter la fonction `dev_write_leds` qui permet de modifier l'état des leds en fonction de la valeur du paramètre fourni dans le buffer. Pour modifier l'état de la led, on utilisera la fonction `leds(uint8 val)` définies dans `oslib/drivers/include/io.h`.

b. Tester le fonctionnement de votre driver avec le programme `user/tp2_ex2.c`.

3.2. Coupleurs sur interruption

Principe de l'attente d'évènements asynchrones : une tâche qui attend un évènement asynchrone (par l'appel `read` sur le fichier de périphérique correspondant) doit être bloquée. Lorsque l'évènement intervient et la routine de traitement de l'interruption générée par le coupleur doit permettre de débloquer la tâche.

Remarque : à partir d'une routine de traitement d'IRQ, il n'est pas possible d'utiliser un appel système. Il faut appeler directement son implémentation `sys_*`.

a. Compléter, dans le fichier `oslib/drivers/target.c`, les fonctions relatives à l'utilisation du bouton P0.14. Tester avec le programme `usr/tp2_ex3.c`.

b. Les boutons P0.16 à P0.20 ne génèrent d'interruption. Pour générer les évènements, malgré tout, on teste l'état des boutons avec les fonctions `buttonX()` (définies dans `oslib/drivers/include/io.h`) dans la fonction de rappel `timeout_cb` associée à l'IRQ périodique du `_TIMER0`. Compléter le code nécessaire et tester avec le programme `usr/tp2_ex4.c`.

c. Le device `dev_console` permet d'encapsuler le fonctionnement de l'UART0. L'envoi des caractères sur la liaison série se fait par scrutation du registre d'état (pas de possibilité de bloquer la tâche), alors que la réception utilise les interruptions, et permet de stocker dans un buffer les caractères reçus.

   A chaque caractère reçu la fonction de rappel `on_receive_cb` est appelée. Cette fonction permettra de mettre des jetons dans le sémaphore de lecture. Lors d'une opération de lecture, il faut prendre un jeton pour chaque caractère lu. Les caractères peuvent être obtenus via la fonction `uart_getchar` définie dans `oslib/drivers/include/uart.h`.

   Implémenter le driver pour la console (liaison RS232). Le module `termio` utilise le device `dev_console` pour offrir des fonctions de gestion de terminal VT100, avec entre autres des fonctions `term_printf` et `readline`. Tester votre code avec le programme `user/tp2_ex5.c`.

4. Carte SD et système de fichiers FAT32

4.1. La carte SD et son organisation

La carte SD est un périphérique de masse comprenant de la mémoire Flash NAND, ainsi qu'un contrôleur qui prend en charge à la fois la gestion de la Flash NAND et le protocole de communication avec l'extérieur. La carte possède 2 protocoles de communication bas-niveau : un basé sur le bus de données SD 4 bits, l'autre compatible avec une liaison SPI. On utilise le mode SPI (pour l'autre, il faut que le microcontrôleur dispose d'un coupleur spécifique). Les échanges de données entre la carte et l'extérieur est réalisé via un protocole de plus haut niveau basé sur des échanges de messages. L'implémentation de la communication avec la carte SD est fournie dans `oslib/drivers/src/diskio.c`.

Des infos supplémentaires : **http://elm-chan.org/docs/mmc/mmc_e.html** (et la source du code de `diskio.c` ;-)

De notre point de vue, la carte SD est organisée comme un tableau de secteurs de 512 octets et l'accès (en lecture seule :-) consistera à utiliser l'interface de `diskio.c` pour lire des paquets de 512 octets (la plus petite quantité d'information accessible).

Ce tableau d'octets est utilisé pour y stocker la structure d'un système de fichiers FAT32. Le décodage des informations va nous permettre de retrouver et d'accéder au contenu des fichiers et répertoires présents dans le système de fichier.

Le programme `usr/tp3_ex1.c` implémente dans une tâche un mini-shell qui permet d'exécuter des commandes d'initialisation de la carte SD, de lire des informations d'état de la carte SD, et particulièrement utile, d'afficher sur le terminal série le contenu d'un secteur passé en paramètre. Afin de gagner de la place en RAM, on peut enlever les drivers des boutons et leds en enlevant la définition du symbole `__WITH_GPIO_DEV` du fichier Makefile.

a. Jouer un peu avec le programme et regarder le secteur 0 de boot (MBR).

4.2. Format du MBR et du système de fichiers FAT

Le premier secteur de la carte contient le MBR (Master Boot Record), avec en particulier une table pouvant contenir les informations pour 4 partitions à partir de l'offset 0x1BE :

offsets (hex)	length	description
1BE – 1CD	16	Table Entry for Primary Partition 1
1CE – 1DD	16	Table Entry for Primary Partition 2
1DE – 1ED	16	Table Entry for Primary Partition 3
1FE – 1FD	16	Table Entry for Primary Partition 4

Les deux derniers octets du secteur doivent contenir le marqueur 0x55 0xAA.

Chaque entrée de la table de partition a le format suivant :

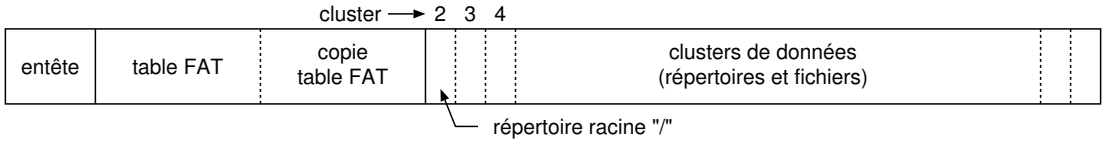
offsets	length	description
0	1	boot indicator (80h = active)
1 – 3	3	starting sector in CHS values
4	1	partition-type descriptor
5 – 7	3	ending sector in CHS values
8 – 11	4	starting sector
12 – 15	4	partition size (in sectors)

* CHS = Cylinder Head Sector (obsolète)
* Type de la partition : Linux (0x83), FAT16 (0x06), FAT32 (0x0B).
* Starting sector : n° du premier secteur du système de fichiers.
* Partition size : taille de la partition en nombre de secteurs.

Exemple : partition 0, système de fichiers FAT16

```
 offsets  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F
000001b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 02
000001c0  0a 00 06 3e fe 00 87 00  00 00 79 ff 3a 00 00 00
000001d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

Le numéro de 'starting sector' fournit le secteur de début de la zone utilisée pour implémenter le système de fichier FAT. La structure du système de fichier est la suivante :



L'entête contient des informations de gestion, la table FAT contient les liens de chaînage entre les différentes parties des fichiers. Par sécurité, il y a généralement une copie de la table FAT. Les données (fichiers et répertoires) sont stockées dans les clusters.

Le cluster est la plus petite unité d'allocation utilisable par le système de fichiers pour stocker les données d'un fichier ou d'un répertoire. La taille d'un cluster est équivalente à plusieurs secteurs.

4.3. Lecture de la table de partition et de l'entête FAT

L'entête du système de fichiers occupe au moins un secteur de 512 octets. Quelques éléments importants de l'entête :

offset (octets)	taille (octets)	description
0	1	la valeur 0xEB ou 0xE9, sinon ce n'est pas un système de fichiers FAT
11	2	la taille en octets d'un secteur (généralement 512)
13	1	le nombre de secteurs dans un cluster
14	2	le nombre de secteurs dans l'entête
16	1	le nombre de tables FAT (en général 2)
36	4	la taille d'une table FAT en secteurs
44	4	le numéro de cluster du répertoire racine /
510	1	la valeur 0x55, sinon ce ,est pas du FAT
511	1	la valeur 0xAA, sinon ce ,est pas du FAT

La fonction

```
int fat_mount(FatFS *fat, int partition);
```

du fichier `oslib/fat.c` permet de lire l'entrée la table de partitions passée en paramètre et d'initialiser les champs de la structure `FatFS`

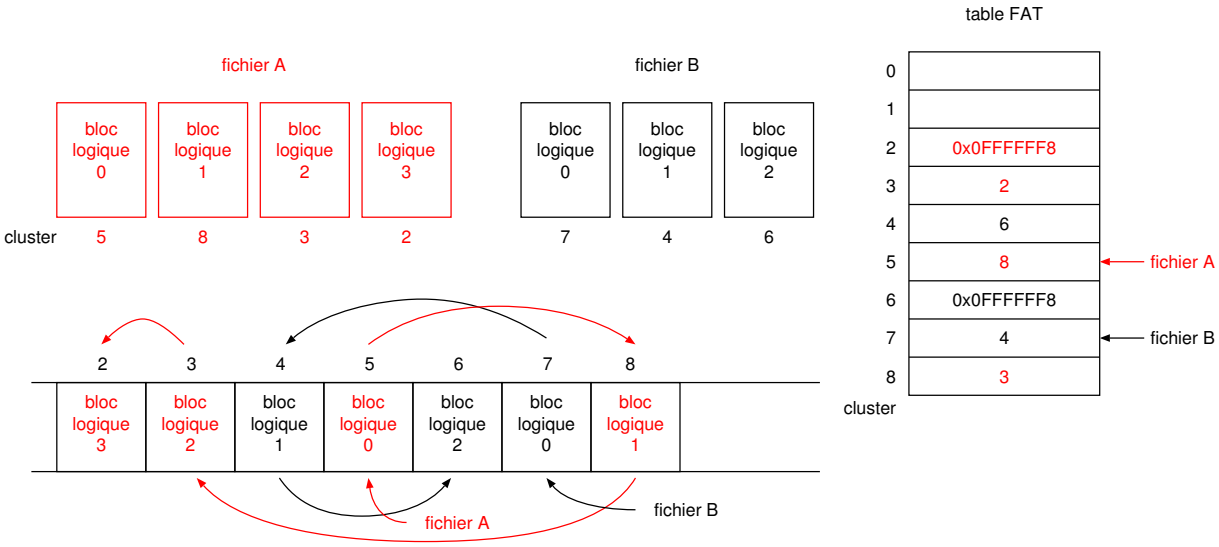
```
typedef struct {
        uint32 part_start;      /* starting sector for partition */
        uint32 part_size;       /* partition size */
        uint32  csize;          /* number of sectors per cluster */
        uint32 fat_start;       /* starting sector of FAT table */
        uint32 fat_size;        /* size of FAT table (in sectors) */
        uint32 data_start;      /* starting sector of data clusters */
        uint32 root;            /* starting cluster of root directory */
} FatFS;
```

Les secteurs sont lus à l'aide la fonction `disk_read` dans la variable globale `buf` définie dans `oslib/fat.c`. Etant donné qu'on ne dispose que de peu de mémoire RAM, on ne pourra avoir en mémoire à un instant donné qu'un seul secteur. Les macros `READ8(buf,offset)`, `READ16(buf,offset)`, `READ32(buf,offset)` ont été définies pour permettre de lire dans le buffer à un offset quelconque des valeurs sur 8, 16 ou 32 bits même si elles sont désalignées.

a. Modifier la fonction `fat_mount`. Cette fonction est utilisée la commande `mount` du mini-shell. Les paramètres importants sont affichés. Vérifier visuellement les valeurs en affichant les secteurs concernés (commande `dd` – *disk dump*).

4.4. Obtenir des informations sur la racine

Les données d'un fichier ou d'un répertoire peuvent s'étaler sur plusieurs clusters. Le chaînage des clusters est réalisé grâce à la table FAT.



Chaque entrée de la table FAT, correspondant à un numéro de cluster référençant un bloc de données faisant partie d'un fichier, contient le numéro de cluster dans lequel on trouvera le bloc logique suivant. Les numéros de clusters sont codés sur 32 bits (d'où FAT32 – en réalisté sur 28 bits). Les numéros de clusters doivent être inférieurs à 0x0FFFFFF8. Cette valeur ou toute valeur supérieure est un marqueur de fin de fichier.

a. Compléter la fonction `uint32 next_cluster(FatFS *fat, uint32 cluster)` qui permet d'obtenir le numéro du cluster suivant du paramètre *cluster* à partir la table FAT. Pour cela, il faut :

   – Calculer le numéro de secteur dans le quel se trouve la donnée cherchée (sachant que la table FAT commence à `fat->fat_start` et que dans un secteur de 512 octets, il y a 512/4 = 128 index sur 32 bits),

   – Lire le secteur concerné (`disk_read`),

   – Lire l'index à l'offset voulu

b. Quand on a un numéro de cluster, pour obtenir le numéro de secteur dans lequel on trouvera le début des données du cluster, on utilise la fonction

```
uint32 cluster2sector(FatFS *fat, uint32 cluster);
```

   qui renvoie le numéro de secteur correspondant au paramètre *cluster*. Le premier cluster est le cluster 2 (et c'est comme ça !!).

   Compléter la fonction `cluster2sector`

c. Finalement, pour être en mesure de lire les données d'un fichier ou d'un répertoire, il faut connaître le numéro du premier cluster. C'est le rôle de la fonction

```
uint32 fat_get_info(FatFS *fat, char *filename, uint32 *size, uint32 *isdir);
```

   qui permet en plus du numéro de cluster, qui est renvoyé, de récupérer des informations comme la taille du fichier et la nature de l'élément (fichier ou répertoire) à partir d'un chemin qui définit l'emplacement du fichier dans l'arborescence.

C'est une fonction relativement complexe. Il y a cependant un cas simple : le répertoire racine. Lorsque le nom du fichier est "/", c'est un répertoire, la taille est nulle et le numéro de cluster est `fat->root`.

Compléter la fonction `fat_get_info` pour gérer le cas du répertoire racine.

d. La commande `stat /` du mini-shell utilise les 3 fonctions développées ci-dessus. Le résultat doit être, avec la carte SD fournie,

```
>stat /
'/' is a directory
  size          : 0
  start cluster : 2
  start sector  : 9616
  block 0 --> cluster 0x2 (sector 9616 - 9623)
```

4.5. Obtenir des informations sur un fichier : parcourir les répertoires

Pour être en mesure d'obtenir des informations (n° cluster, size, isdir?) avec la fonction `fat_get_info` sur un fichier ou répertoire quelconque dans l'arborescence, il est nécessaire d'être capable de rechercher un nom de fichier dans un répertoire.

Organisation des répertoires

Un répertoire est organisé comme un *tableau* contenant les différents noms de fichiers et de sous-répertoires présents à ce niveau d'arborescence.

* Les données du répertoire sont stockées dans un ou plusieurs cluster (comme pour un fichier).
* Chaque entrée du répertoire consiste en un paquet de 32 octets organisé de la manière suivante :

offset (octets)	taille (octets)	description
0	11	le nom du fichier au format DOS (8 caractères + 3 caractères d'extension)
11	1	un champ de flags : * Si `(E[11] & 0x10) != 0` alors l'entrée est un nom de répertoire * Si `(E[11] & 0x08) != 0` alors l'entrée n'est pas une vraie entrée, mais soit un label, soit une extension VFAT (nom de fichier long), donc on ne s'en occupe pas (laissé pour vos longues soirées d'hiver :-)
28	4	la taille du fichier (0 pour un répertoire)
26	2	le mot de poids faible du numéro du premier cluster contenant les données du fichier ou du sous-répertoire
20	2	le mot de poids fort du numéro du premier cluster contenant les données du fichier ou du sous-répertoire (le numéro de cluster utilise en réalité sur 28 bits)

Le premier caractère peut en outre avoir une signification spéciale :

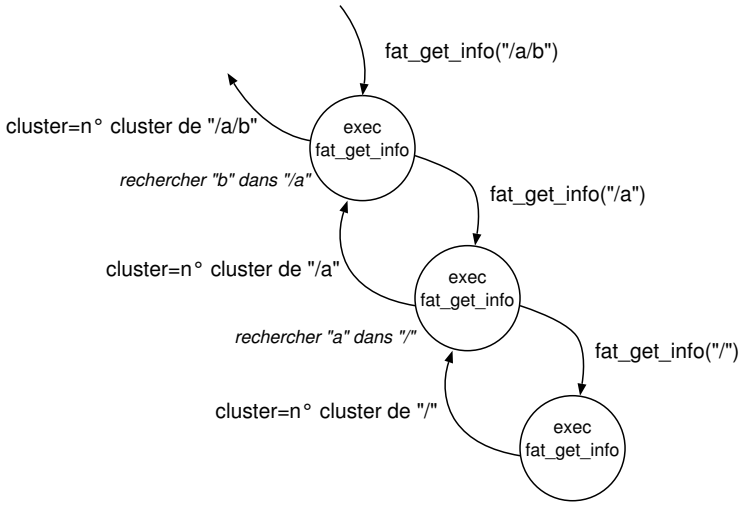
– S'il est égal à 0, cette entrée n'est pas une vraie entrée et il n'y a pas d'autre entrée valide après (c'est la fin de la liste des entrées du répertoire),

– S'il est égal à 0xE5, cette entrée n'est pas une vraie entrée, mais il en reste peut être d'autres après,

– S'il est égal à 0x05, cette entrée est une vraie entrée dont le premier caractère est en fait 0xE5.

Principe de la recherche des informations sur un nom dans une arborescence

`fat_get_info` fonctionne de manière récursive : pour connaître le numéro de cluster du fichier `/a/b`, il faut être capable de chercher le nom 'b', dans le répertoire `/a`, donc il faut avoir le numéro de cluster pour le nom `/a`. Pour obtenir le numéro de cluster, de `/a`, il faut être capable de rechercher le nom 'a' dans le répertoire `/`, donc il faut connaître le numéro de cluster de `/`.



a. Compléter la fonction `fat_get_info` qui aura la structure suivante :

```
si path=="/" alors c'est la racine
sinon obtenir le numéro de cluster du répertoire parent
  rechercher le nom de fichier dans le cluster du répertoire parent
  si trouvé dans la liste des entrées alors
    renvoyer les informations
  sinon erreur : le fichier n'existe pas (-1)
  finsi
finsi
```

Remarques :

1. Des fonctions utilitaires sont disponibles pour manipuler les noms de fichier :

   /* cpy_name : convert a src name from the 8.3 format (8 chars + 3 extension chars) to standard C string */
   void *cpy_name*(**char** **dest*, **char** **src*);

   /* returns parent's name (everything before the last '/') -- some extra space is allocated for new string (free it after use !!) */
   char * *dirname*(**char** **path*);

   /* returns the filename (everything after the last '/') */
   char * *basename*(**char** **path*);

2. On ne peut lire qu'un secteur à la fois (`disk_read`). Or, un cluster peut s'étendre sur plusieurs secteurs : il faut itérer la recherche sur tous les secteurs (qui sont contigus) ; voir le champ `fat->csize`.

3. Les données d'un répertoire peuvent s'étendre sur plusieurs clusters. Tant qu'on n'est pas arrivé au dernier cluster, (si l'entrée cherchée n'a toujours pas été trouvée), il faut continuer avec le cluster suivant (fonction `next_cluster`).

b. Tester le code. Maintenant, la commande `stat` du mini-shell s'exécute sur tous les fichiers (si on sait où ils sont dans l'arborescence).

```
>stat /dossier2/num4.lua
'/dossier2/num4.lua' is a file
  size          : 14357
  start cluster : 572
  start sector  : 14176
  block 0 --> cluster 0x23C (sector 14176 - 14183)
  block 1 --> cluster 0x23D (sector 14184 - 14191)
  block 2 --> cluster 0x23E (sector 14192 - 14199)
  block 3 --> cluster 0x23F (sector 14200 - 14207)
```

4.6. Lister les entrées d'un répertoire

Pour lister les entrées d'un répertoire, il faut parcourir les différentes entrées des clusters utilisés par le répertoire et renvoyer le nom (plus éventuellement d'autres info : size et isdir?). La commande `ls` du mini-shell permet de déterminer, grâce à `fat_get_info`, le numéro de cluster du répertoire. On initialise ensuite la structure DIR qui va permettre de sauvegarder l'état du parcours du répertoire entre deux appels à la fonction `fat_next_dir` dont est le rôle est de fournir les informations l'entrée suivante.

a. Compléter la fonction `fat_next_dir` pour récupérer les champs `entry`, `entry_size` et `entry_isdir` correspondant à l'entrée stockée dans le secteur `buf` à l'offset `dir->offset`.

b. Tester la commande `ls` : l'arborescence de répertoires et de fichiers est la suivante :

```
/
 +- dossier1
 |   +- dossier3
 |   |   +- fichier3.txt
 |   |
 |   +- fichier1.txt
 |   +- fichier2.txt
 |
 +- dossier2
 |   +- edu_lp~1.pdf
 |   +- um10139.pdf
 |   +- num4.lua
 |
 +- fichier0
```

4.7. Lire les données d'un fichier

La fonction

```
uint32 fat_read(FatFS *fat, uint32 cluster, uint32 offset, uint32 len, void* buffer);
```

permet de lire `len` octets de données d'un fichier dont on connaît le numéro de premier `cluster`, à partir de l'`offset` fourni et les stockent dans `buffer`.

La fonction devra pouvoir gérer des lectures de données réparties sur plusieurs secteurs, voire plusieurs clusters.

La commande `cat` du mini-shell permet d'afficher sur le terminal série le contenu du fichier. On pourra tester, par exemple,

```
cat /fichier0          // les données du fichier tiennent dans moins d'un secteur

cat /dossier2/num4.lua // les données s'étalent sur plusieurs clusters
```

5. Intégration dans le VFS

Les fonctions développées dans la partie précédente permettent d'accéder à un système de fichier FAT en lecture seule et ont été utilisées pour implémenter des commandes classiques de manipulation de fichier (`stat`, `ls` et `cat`). Pour aller plus loin, il est possible de rendre ces commandes indépendantes du système de fichier sous-jascent en intégrant le driver FAT dans le VFS développé précédemment.

Pour cela, on définit un device `dev_fs` qui permet d'interface les appels standard open, close, … avec les fonctions de manipulation du système de fichier.

a. La fonction `open` doit être modifiée pour indiquer que si le device n'est pas un device matériel fourni par `dev_lookup`, alors il s'agit d'un fichier ou d'un répertoire standard existant physiquement dans le système de fichier sur la carte SD. C'est donc ensuite la méthode `open` de `dev_fs` qui indique si le nom fourni en paramètre correspond à un fichier valide ou pas. Elle doit également :

   – dans le cas d'un répertoire, initialiser le champ `dir` de la structure `FileObject` en allouer une structure `DIR` et en positionnant les valeurs de départ,

   – dans le cas d'un fichier standard, initialiser le champ `file` en allouant une structure `FIL`.

b. La méthode `close` de `dev_fs` doit libérer les structures allouées. Compléter la méthode `close`.

c. Compléter méthode `read` de `dev_fs` qui appelle `fat_read` après avoir calculé la taille exacte des données à lire (l'utilisateur peut demander de lire 512 octets ; cependant, il faut fournir à `fat_read` la taille exacte des données à lire en tenant compte de l'offset de début et du fait que la taille du fichier peut être plus petite que ce qui est demandé). On renvoie le nombre d'octets lus, et on incrémente le champ `offset` de la structure `FileObject` d'autant (ce qui permet à chaque lecture d'avancer dans le fichier).

d. Tester les commandes du mini-shell avec le programme `usr/tp3_ex2.c`. En particulier, vérifier que les fichiers du répertoire `dev` sont visibles et qu'il est possible de faire du `ls /dev`, ou un `cat /dev/btn0`.

e. Ajouter une commande `echo val_num filename` qui écrit la valeur numérique *val_num* dans le fichier `filename`. Tester sur les leds (et pas sur un fichier standard vu qu'on n'a qu'un driver en lecture seule).

6. Pour aller plus loin …

a. Le répertoire `oslib/drivers/lib` contient des modules pour gérer le bus i2c, le capteur de température LM75 et l'EEPROM CAT1025. Ajouter des drivers pour ces éléments.

b. Le projet OS se poursuit à votre initiative …