istic

TP-Projet SGF

1. Introduction

L'objectif de ce mini-projet est de réaliser un Système de Gestion de Fichiers simplifié capable de gérer des accès à des fichiers séquentiels de type flot.

L'idée est ici de vous faire programmer en C une partie de la couche logicielle du SGF qui réalise l'interface entre, d'un coté, les E/S physiques permettant d'accéder au périphérique de stockage, et de l'autre, les E/S utilisateur qui permettent de travailler sur des fichiers logiques.

2. Organisation des fichiers sur le SGF

L'objectif de cette section est de présenter l'organisation des fichiers sur le support de stockage, et en particulier comment sont représentés :

- les descripteurs de fichiers
- le catalogue,
- les blocs de données,
- le bloc contrôle fichier.

Dans la suite, on supposera que la taille des secteurs, des blocs d'E/S et des unités d'allocation est de 512 octets.

2.1. Descripteurs de fichier

Les descripteurs de fichiers sont stockés sur le disque, ils servent à stocker les informations *permanentes* d'un fichier du SGF (nom, taille, table d'implantation). Dans notre cas, les descripteurs de fichiers sont représentés par un type structuré appelé **`file_descriptor`** (voir **`directory.h`**), dont la définition est donnée ci-dessous :

```
typedef struct {
   char name[16];     // Nom externe du fichier
   int  size;         // Taille (en octet) du fichier
   int  alloc[122];   // Table d'implantation des blocs de données
} file_descriptor;
```

Dans ce TP, un descripteur de fichier physique occupe 508 octets, il contient les informations suivantes (on rappelle que sur les machines Linux utilisées pour le projet, les entiers sont codés sur 32 bits, soit 4 octets).

- Nom du fichier (**`name`**) formé de 16 caractères au maximum (**`\0`** inclus).
- Taille du fichier en octets (**`size`**) codée sur un type **`int`** (4 octets)
- Table d'implémentation (**`alloc[]`**), un tableau contenant les adresses disques des blocs de données du fichier (par ex. **`alloc[0]`** contient l'adresse disque du 1er bloc de données du fichier, **`alloc[1]`** l'adresse disque du 2ème bloc de données, etc.)

2.2. Le catalogue

Le catalogue sert à stocker les informations correspondant aux fichiers stockés sur le système de gestion de fichier.

Un fichier catalogue contient des *entrées catalogue* dont la structure est donnée ci-dessous. Chaque entrée catalogue contient ainsi un descripteur de fichier (**`file_descriptor`**) et un marqueur (**`free`**) dont le rôle de d'indiquer si l'entrée du catalogue est occupée par un descripteur de fichier (valeur **`'U'`**) ou si elle est vide (valeur **`'F'`**).

```
#define FREE_ENTRY 'F'
#define USED_ENTRY 'U'
typedef struct {
    int free;              // Marqueur d'emplacement (libre/occupé)
    file_descriptor desc;  // Descripteur de fichier
} dir_entry;
```

Pour simplifier la gestion du catalogue, chaque entrée de catalogue occupe exactement 512 octets (508 pour le descripteur, 4 pour le marqueur libre/occupé), soit un bloc d'E/S (voir l'annexe A).

2.3. Organisation de la table des blocs de données

On considère ici que l'unité d'allocation (UA), l'unité d'échange (bloc d'E/S) ont également une taille de 512 octets. Les blocs de données des fichiers sont stockés sur le disque à partir du bloc n°256 , le bloc de données n°0 est donc stocké à l'adresse 256 du disque, le bloc de données n°1 à l'adresse 257, etc. (voir l'annexe 1 pour plus de détails).

Pour savoir si un bloc de donné n°i est libre ou occupé (c.-à-d. utilisé pour stocker les données d'un fichier), on utilise une table des blocs libres, qui est également stockée sur le disque. Cette table occupe les 8 premiers blocs du disque à partir de l'adresse 0 (voir l'annexe A).

Cette table associe, à chaque bloc de données du disque, un octet qui indique si le bloc de donnée est actuellement utilisé (valeur à **`USED_BLOCK`**) ou libre (valeur à **`FREE_BLOCK`**). Ainsi pour connaître le statut du bloc de données n°i (c.-à-d. le bloc d'adresse disque 256+i) du SGF, il faut :

1. Lire le bloc d'adresse disque (*i div 512*).
2. Lire l'octet n°(*i modulo 512*) dans ce bloc.

2.4. Le Bloc Contrôle Fichier

Le Bloc Contrôle Fichier est une structure de données qui est utilisée par un programme pour stocker les informations d'accès à un fichier. Contrairement aux informations stockées dans un descripteur de fichiers, les informations du BCF sont temporaires et liées à un contexte d'exécution particulier (voir le fichier **`syr1_file.h`**)

Le type structuré **`SYR1_FILE`** définit un BCF (Bloc Contrôle Fichier) :

```
typedef struct {
      file_descriptor descriptor; // Copie mémoire du descripteur physique
      unsigned char* buffer;  // Tampon d'E/S
      char mode [3];          // Mode d'ouverture du fichier ('r' ou 'w')
      int current_block ;     // Numéro du bloc en cours d'utilisation
      int file_offset;        // Position dans le fichier
      int block_offset ;      // Position dans le bloc courant
} SYR1_FILE ;
```

Dans un programme utilisateur, on ne manipule pas directement la structure **SYR1_FILE**, on préfère en général y accéder par un pointeur (en l'occurrence le type **SYR1_FILE ***).

3. Fonctions de la bibliothèque

3.1. Fonctions d'Entrée/Sortie niveau utilisateur

Les fonctions systèmes offertes au niveau utilisateur par notre SGF ont une interface très similaire à celles offertes par le SGF d'Unix (cf. cours de C). Elles sont succinctement décrites ci-dessous

- **SYR1_FILE * syr1_fopen(char *nom, char *mode)** : initialise les accès sur le fichier de nom externe **name**, et rend en résultat un pointeur sur un BCF.
  - Si la chaîne **mode** vaut la chaîne **"r"**, le fichier est ouvert en mode *lecture*, et on se positionne au début du fichier.
  - Si la chaîne **mode** vaut la chaîne **"w"**, le fichier est ouvert en mode *création* : s'il n'existe pas de fichier de nom externe **name**, on le crée, s'il en existe un, on l'écrase. Dans tous les cas on se positionne au début du fichier
- **int syr1_fwrite(SYR1_FILE * file, int size, int nbitem, char *buffer)** : écrit **nbitem** éléments de taille **size** à la position courante dans le fichier **file** passé en paramètre à partir du tampon mémoire **buffer**.
- **int syr1_fread(SYR1_FILE * file, int size, int nbitem, char * buffer)** : lit **nbitem** éléments de taille **size** à la position courante dans le fichier **file** passé en paramètre et les range dans le tampon mémoire **buffer**.
- **int syr1_putc(unsigned char c, SYR1_FILE * file)** : écrit le caractère **c** à la position courante dans le fichier logique **file** passé en paramètre.
- **int syr1_getc(SYR1_FILE * file)** : lit un caractère à partir de la position courante dans le fichier logique **file** passé en paramètre.
- **void syr1_fclose(SYR1_FILE * file)** : ferme les accès sur le fichier **file** (la fonction doit s'assurer que le contenu du disque est bien à jour par rapport aux données du BCF).

L'objectif de ce TP consiste à écrire l'ensemble des fonctions ci-dessous, en vous aidant d'autres fonctions décrites dans les paragraphes suivants.

- **syr1_fopen_read()** utilisée par **syr1_fopen(),**
- **syr1_fopen_write()** utilisée par **syr1_fopen(),**
- **syr1_fclose_read()** utilisée par **syr1_fclose(),**
- **syr1_fclose_write()** utilisée par **syr1_fclose()**
- **syr1_getc(),**
- **syr1_putc()**.

Le code de **syr1_fopen(),syr1_fclose(), syr1_fwrite()** et **syr1_fread()** vous est donné.

3.2. Fonctions de gestion du catalogue

Lorsque l'on souhaite créer, supprimer ou modifier des fichiers du SGF, il faut être capable de mettre à jour les informations du catalogue de fichier. Pour cela, on dispose de plusieurs fonctions décrites ci-dessous (une description plus complète est donnée dans le fichier entête **`directory.h`**):

- **`int search_entry(char* name, file_descriptor* desc)`** : recherche dans le catalogue d'une entrée correspondant au nom externe passé en paramètre.
- **`int update_entry(file_descriptor* entry)`** : mise à jour d'une entrée du catalogue disque.
- **`int create_entry(char* name, file_descriptor* entry)`** : ajout d'un fichier sur le catalogue disque.
- **`int remove_entry(char* name)`** : suppression d'un fichier du catalogue disque

3.3. Fonctions de gestion de la table des Blocs Contrôle Fichiers

Ces fonctions permettent d'allouer ou de libérer des entrées de la table des Blocs Contrôle Fichiers. Dans sa mise en œuvre, notre mini-SGF ne peut avoir plus de 10 fichiers logiques ouverts en même temps (voir le fichier **`syr1_file.h`**).

- **`SYR1_FILE * alloc_logical_file(char *nom, char *mode)`**: alloue une entrée vierge dans la table des BCF, en s'assurant que le fichier de nom externe **`nom`** n'est pas déjà ouvert dans un mode incompatible avec la chaîne **`mode`**.
- **`int free_logical_file(SYR1_FILE * bcf)`** : libère l'entrée la table des BCF passée en paramètres.

3.4. Fonctions de gestion des blocs de données

Un fichier est formé d'un ensemble de blocs de données qui contiennent les données du fichier.

- Lorsque l'on souhaite ajouter un bloc de données à un fichier, il faut trouver un bloc de données libre (c.-à-d. non utilisé par un autre fichier) sur le disque.
- De même, lorsque l'on supprime un fichier, il faut signaler au SGF que l'on libère les blocs du fichier (afin de les rendre disponibles pour d'autres fichiers). Cela est fait dans la fonction **`remove_entry.`**

L'allocation et la libération de blocs de données se font au travers de deux fonctions (une description plus complète est donnée en annexe, ainsi que dans le fichier entête **`directory.h`**) :

- **`int get_allocation_unit()`** : retourne l'*adresse disque* d'un bloc de données libre, et marque ce bloc comme occupé dans la table des blocs libres stockée sur le disque.
- **`free_allocation_unit(int data_block_id)`** : marque le bloc de données dont le *numéro* a été passé en paramètre comme libre dans la table des blocs de données libres stockée sur le disque (utilisée par la fonction **`remove_entry).`**

3.5. Fonctions d'Entrées/Sorties physiques

Ces fonctions permettent d'accéder directement au périphérique de stockage à partir d'une adresse disque. L'unité d'échange est le bloc d'E/S (un bloc d'E/S = 512 octets). On dispose de deux fonctions (une description plus complète est donnée dans le fichier d'entête **`physical_io.h`**):

- **`int read_block(int disk_addr, unsigned char *buffer)`** : lit le bloc d'adresse disque **`disk_addr`** et stocke son contenu dans le tampon **`buffer`**.
- **`int write_block(int disk_addr, unsigned char *buffer)`** : écrit le contenu du tampon **`buffer`** vers le bloc d'E/S d'adresse disque **`disk_addr`** .

4. Travail à effectuer

4.1 Mise en œuvre des lectures

Dans un premier temps, il vous est conseillé de vous concentrer sur la mise en œuvre des fonctions d'accès en mode lecture (dans le fichier **`file_read.c`**), à savoir :

1. **`syr1_fopen_read(…)`**
2. **`syr1_getc()`**
3. **`syr1_fclose_read()`**

Ces fonctions vous permettront de tester la lecture dans un fichier (par exemple en utilisant l'utilitaire **`syr1_cat`** ou en écrivant votre propre programme de test), et ainsi de valider partiellement le fonctionnement de votre TP.

4.2 Mise en œuvre des écritures

Une fois l'accès en lecture fonctionnel, vous pourrez passer à la mise en œuvre des fonctions d'écriture (dans le fichier **`file_write.c`**):

1. **`syr1_fopen_write(…)`**
2. **`syr1_putc(…)`**
3. **`syr1_fclose_write(…)`**

Attention, ces fonctions sont plus délicates à mettre en œuvre car elles modifient le contenu du disque, et peuvent donc rendre le SGF incohérent (par exemple en altérant par erreur un bloc catalogue ou un bloc de la table des blocs libres/occupés). Si c'est le cas, vous pourrez régénérez un système correct par la commande **make init-fs**.

Dans tous les cas assurez vous que vos fonctions respectent les spécifications données dans le fichier entête **`syr1_file.h`**.

4.3 Quelques conseils

1. Lisez bien les spécifications des fonctions avant de vous jeter la tête la première dans le codage, en particulier pour ce qui concerne les codes d'erreurs.
2. Assurez vous que vos sources compilent (avec l'option -*Wall*) sans provoquer le moindre message de *warning* (rappelez vous que les compilateurs C sont très permissifs). Ce qui est considéré comme un warning en C provoquerait vraisemblablement une erreur de compilation en JAVA
3. Vérifiez bien les codes de retour des appels de fonctions : en C, il n'y a pas d'exception, le seul moyen de signaler une erreur est justement de retourner un code d'erreur !

4. Attention aux pointeurs ! En particulier lorsque vous voulez copier/comparer des chaînes, il faut utiliser les fonctions **`strcpy(…)/strcmp(…)`** vues en cours ;
5. Lorsque l'on manipule un pointeur sur une structure, on accède à ses champs non pas par l'opérateur '**`.`**' mais par l'opérateur **`-> ;`**
6. Attention à l'opérateur **`&`** qui n'a pas toujours le même sens en C et JAVA. Attention également à la confusion entre **`=`** et **`==`** (en général le compilateur n'indique pas d'erreurs, mais seulement un avertissement pour ce type de confusions).

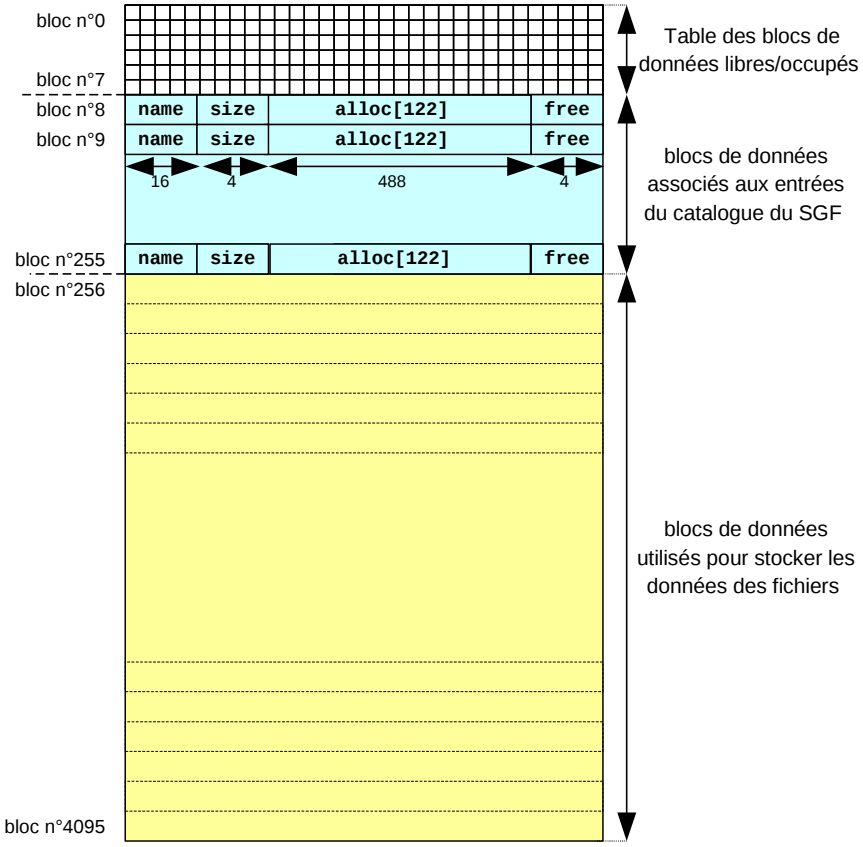
4.4 Rendu du TP :

Avant de rendre votre TP, assurez vous que :

1. Il compile (si certaines fonctions ne compilent pas, mettez tout leur contenu en commentaire).
2. Votre source C est propre, bien indenté, bien commenté, et les entêtes des fichiers contiennent bien vos nom/prénoms ainsi que votre groupe.
3. Il passe bien a l'impression (pas de lignes tronquées, etc.). Pour vous en assurer, vous pouvez utiliser la commande **`make listing`**, qui va produire un fichier **`tp_sgf.pdf`** à partir de vos sources C).

Annexe 1 : organisation physique du SGF

L'organisation physique des fichiers du SGF utilisé dans ce TP est illustrée ci-dessous.



On considère ici un support de stockage d'une capacité de 4096 blocs d'E/S. Cette capacité de stockage est découpée en trois zones distinctes :

1. La table des blocs libres/occupés, qui associe à chaque bloc de données du SGF un octet qui indique si le bloc de données considéré est libre (octet à la valeur ASCII 'F') ou occupé (octet à la valeur ASCII 'U').
2. Le catalogue, qui est utilisé pour stocker les descripteurs de fichiers physiques. Dans notre mise en œuvre, chaque descripteur occupe un bloc d'E/S complet, le catalogue contient au maximum 248 entrées.
3. La zone de stockage contenant les blocs de données des fichiers stockés sur le SGF.

Dans ce TP, nous allons simplement simuler le fonctionnement d'un périphérique de stockage en utilisant un fichier image du contenu du périphérique. Ce fichier image est appelé **`disk.img`** et est stocké à la racine du répertoire du TP.

Bien que ce fichier ne soit pas lisible par un éditeur de texte (il contient des informations binaires), il est cependant possible de le visualiser en utilisant un afficheur hexadécimal (essayez la commande **`hexdump -v -e '"%06_ax |" 16/1 " %02x"' -e '" |"' -e '4/4 " %10u"' -e '" |"' -e '16/1 "%_p" "\n"' disk.img | less`**). Vous pourrez ainsi consulter directement les données du catalogue ou de la table des blocs libres/occupés.

Annexe 2 : utilitaires de gestion de SGF

Afin de vous aider à utiliser le SGF, un certain nombre d'utilitaires vous sont fournis dans le répertoire **`./bin`**. Leurs programmes sources sont stockés dans le répertoire **`tests`**.

Ces utilitaires permettent de consulter, ou de modifier le contenu du SGF. Ils ont un fonctionnement très similaire aux outils de manipulation de fichier d'UNIX, et afin de les distinguer de ces derniers, ces programmes sont tous préfixés par la chaîne « **`syr1_`** ».

- **`syr1_ls [-l]`** permet de lister le contenu du catalogue, il dispose d'une option (**`-l`** permet de visualiser le contenu de la table d'implantation des fichiers)
- **`syr1_cat nom`** : permet d'afficher sur la S.S le contenu d'un fichier stocké sur le SGF du TP, et dont le nom est passé en argument à la commande.
- **`syr1_cp nom1 nom2`** : permet de copier un fichier (**`nom1`**) du SGF Linux vers le SGF du TP (**`nom2`**)
- **`syr1_rm nom`** : supprime un fichier (dont le nom est passé en argument à la commande) du SGF du TP.
- **`syr1_mkfs`** : formate le SGF (on obtient une image vierge avec un catalogue vide).