Université Lille1
Sciences et Technologies

Master d'informatique
1re année
Module d'architecture
des systèmes d'exploitation

Un système de fichiers « à la Unix »

Philippe MARQUET Gilles GRIMAUD

Janvier 2003
Révision majeure, octobre 2009

Nous remercions à Philippe DURIF, Damien DEVILLE, et François INGELREST pour leur contribution à ce document.

Ce sujet est disponible en ligne à `www.lifl.fr/~marquet/ens/fs/`. Cet accès en ligne autorise des copier/coller... ne vous en privez pas.

Ce sujet de TD est aussi la base de TP. Les séances de TP consistent à mettre en pratique les algorithmes et les structures de données élaborées durant les TD.

On s'intéresse ici à la réalisation d'un système de fichiers au dessus d'un disque magnétique organisé en pistes et secteurs. Le système de fichiers que nous allons étudier est composé de différentes couches logicielles successives. Nous allons progressivement détailler les fonctionnalités et une implantation possible de ces couches.

1 Première couche logicielle : accès au matériel

Nous disposons d'un disque dur organisé en pistes (aussi nommées cylindres), chacune des pistes étant organisée en secteurs. La couche logicielle la plus base définit une interface C avec ce matériel. Cette interface est une forme simplifiée de la norme ATA-2 supportée par les fabricants de disque dur de type IDE (ceux installés dans des PC « standard »). La bibliothèque `hardware` qui constitue cette première couche vous est fournie comme point de départ sous la forme des fichiers `hardware.h` et `libhardware.a` ; voir l'encart page suivante. Cette bibliothèque permet d'émuler certains composants matériels d'un ordinateur, comme une carte ethernet ou un disque dur. Nous ne nous occuperons pour ce projet que du disque dur maître.

Le fichier `hardware.h` définit un jeu réduit de fonctions C qui permettent de contrôler, entre autres, l'activité du disque magnétique. Un fichier de configuration paramètre le fonctionnement du matériel émulé. Un tel fichier, nommé `hardware.ini`, vous est fourni avec la bibliothèque `hardware` ; il contient des valeurs par défaut. Ce paramétrage vous permet d'activer ou de désactiver les différents composants matériels émulés, il est impératif, pour ce qui nous concerne ici, que le disque dur maître soit bien activé (la valeur `ENABLE_HDA` doit être positionnée à 1 dans le fichier de configuration).

La bibliothèque `hardware` définit la fonction :

```
int init_hardware(const char *config_file);
```

Cette première fonction permet d'initialiser le matériel émulé (et donc le disque dur) à partir du fichier de configuration dont le chemin d'accès est fourni en paramètre. L'appel à cette fonction met sous tension le disque et calibre position et mouvement de la tête de lecture. Après initialisation, la tête de lecture est placée sur le secteur 0, piste 0.

Si cette phase d'initialisation du matériel n'est pas effectuée, le comportement du disque n'est pas prédictible. Il faut donc appeler cette fonction avant toute autre opération en guise d'initialisation de vos programmes.

Un fichier est créé pour contenir les informations du disque maître. Le nom par défaut de ce fichier est `vdiskA.bin`. Il peut être modifié grâce au fichier de configuration.

Mise en place des travaux pratiques

Pour commencer le TP il faut récupérer le fichier :

```
/home/enseign/ASE/src/tpfs.tgz
```

Il faut décompresser le fichier à l'aide de la commande

```
% tar xzvf tpfs.tgz
```

Un répertoire `tpfs` est créé. Vous trouverez dans ce répertoire un fichier `mkhd.c` (*make hard disk*) qui montre comment initialiser les disques. Pour la suite du TP vous n'aurez besoin que du disque « MASTER ».
Un fichier `Makefile` est disponible. Il compile le fichier `mkhd.c` avec la bibliothèque `hardware`. Tapez simplement `make`.
Notez encore que dans la bibliothèque `hardware` que nous vous fournissons simule le fonctionnement des primitives ATA en utilisant en guise de disque un fichier Unix nommé `vdiskA.bin` (option par défaut pour le disque maître) créé dans le répertoire courant. Si ce fichier n'existe pas notre simulateur de disque le recrée, **dans un état non initialisé**. En supprimant ce fichier vous créez donc un nouveau disque...
En cas de problème de place sur votre compte Unix, vous pouvez modifier le fichier de configuration pour désigner un autre emplacement pour le disque maître. Utilisez par exemple `/tmp/vdiskA.bin` comme valeur. Le disque virtuel sera alors placé dans un répertoire temporaire qui n'imputera pas votre quota disque. Ce répertoire est périodiquement effacé. Pensez dans ce cas à sauvegarder le fichier si vous voulez garder le disque en l'état...

La communication avec le matériel (envoi de commandes et de données, envoi/réception de données) se fait via des ports, soit en écriture soit en lecture. Les numéros des ports utilisés pour la communication avec le disque dur sont définis eux aussi dans le fichier de configuration. On trouve par exemple dans le fichier `harware.ini` fourni :

```
# Paramètres du controlleur IDE
ENABLE_HDA      = 1             # 0 => simulation du disque désactivée
HDA_CMDREG      = 0x3F6         # registre de commande du disque maitre
HDA_DATAREGS    = 0x110         # base des registres de données (r,r+1...r+15)
HDA_IRQ         = 14            # Interruption du disque

# Paramètres de la simulation
HDA_FILENAME    = "vdiskA.bin" # nom du fichier de stockage du disque simulé
```

On ne peut lire ou écrire qu'un seul octet sur un port. Pour écrire une valeurs codées sur plusieurs octets, on utilise un premier port pour l'octet de poids fort et les ports suivants pour les octets de poids plus faible. De la même manière, si une fonction retourne une valeur de plusieurs octets, l'octet de poids fort sera lu sur le premier port, les octets de poids plus faibles sur les ports suivants.

La fonction

```
int _in(int port);
```

réalise la lecture sur le port désigné. La valeur retournée correspond à l'octet qui a été lu sur ce port. Les numéros de port sont identifiés dans le fichier de configuration du matériel `hardware.ini`. La fonction

```
void _out(int port, int value);
```

réalise l'écriture d'une valeur d'un octet sur le port désigné.

L'envoi de commandes se fait également en écrivant sur le port désigné comme port de commande dans le fichier de configuration. Cela permet au microprocesseur de solliciter une opération du disque magnétique. Une fois que le microprocesseur envoie une commande, l'exécution de celle-ci débute immédiatement. Si la commande nécessite des données, il faut obligatoirement les avoir fournies **avant** de déclencher la commande. De la liste des commandes

TABLE 1 – Commandes ATA-2

Nom	code	port de données (P0 ; P1 ; ... ; P15)	objet
`SEEK`	0x02	numCyl (int16) ; numSec (int16)	déplace la tête de lecture
`READ`	0x04	nbSec (int16)	lit `nbSec` secteurs
`WRITE`	0x06	nbSec (int16)	écrit `nbSec` secteurs
`FORMAT`	0x08	nbSec (int16) ; val (int32)	initialise `nbSec` secteurs avec `val`
`STATUS`	0x12	R.F.U.	R.F.U.
`DMASET`	0x14	R.F.U.	R.F.U.
`DSKNFO`	0x16	nbCyl (int16) ; nbSec (int16) ; tailleSec (int16)	retourne la géométrie d'un disque
`MANUF`	0xA2	Id du fabricant du disque (16 octets)	Identifie le disque
`DIAG`	0xA4	status	diagnostic du disque : 0 = KO / 1 = OK

ATA-2 nous avons retenu le sous-ensemble décrit dans la table 1. Le fichier `harware.h` définit aussi des macros identifiant ces commandes :

```
#define CMD_SEEK        0x02
#define CMD_READ        0x04
...
```

Une fois une commande fournie au circuit ATA, celui-ci met un certain temps à la réaliser. Si une deuxième commande est passée entre temps, la première commande est « interrompue » laissant le matériel dans un état indéterminé... Pour informer le microprocesseur (et donc le système) de l'état d'avancement d'une commande le circuit ATA génère un signal d'interruption particulier.

Ce signal est émis après que la commande `SEEK` ait atteint la position demandée, ou pour chaque secteur lu (`READ`), écrit (`WRITE`), ou formaté (`FORMAT`). Enfin ce signal est émis après qu'un diagnostic complet ait été accompli, à ce moment seulement la valeur `OK` ou `KO` peut être lue dans le premier registre de données. Pour les autres commandes le résultat est immédiat.

Pour attendre que le disque ait terminé l'exécution de la commande en cours, il faut utiliser la fonction :

```
void _sleep(int irq_level);
```

Le niveau d'IRQ passé en paramètre dépend du matériel visé (ici le disque dur), ce niveau étant défini dans le fichier de configuration.

De plus, un traiteur d'interruption doit être associé à chacune des 16 IRQ (0 à 15). Un traiteur d'interruption est une fonction de type

```
typedef void (*func_irq_t)();
```

Le vecteur `IRQVECTOR[16]` identifie ces fonctions et se doit d'être initialisé. La fonction `IRQVECTOR[n]()` est appelée lorsque l'interruption de niveau `n` est déclenchée par le matériel.

Enfin la valeur `MASTERBUFFER` est un pointeur sur un `unsigned char` qui identifie le tampon exploité par le contrôleur de disque maître. Ce tampon est exploité par les commandes `READ` et `WRITE` pour stocker les données lues/à écrire.

Exercice 1 (Afficher un secteur : *dump sector*, `dmps`)
Comme premier outil, nous allons concevoir un petit programme qui prend deux arguments en paramètres, un numéro de piste et un numéro de secteur et qui affiche le contenu, octet par octet, du secteur (sous forme hexadécimale par exemple) du disque maître.

Question 1.1 Quelle est la suite de commandes matérielles qu'il faut solliciter pour lire un secteur ? □

Première étape des travaux pratiques — Validation de la bibliothèque d'accès au matériel

Une validation *minimale* de la bibliothèque `drive` peut être obtenue en écrivant les commandes `dmps` et `frmt` au dessus de la bibliothèque. Observer attentivement le résultat de commandes telles les suivantes :
- création d'un disque (`mkhd`) ;
- visualisation d'un secteur quelconque (`dmps`, en particulier, pensez à tester les valeurs extrèmes des paramètres `cylinder` et `sector` des fonctions `read_sector()` et `write_sector()`) ;
- comparaison avec le contenu du fichier Unix `vdiskA.bin` (ou `vdiskB.bin` pour le disque esclave) dans lequel est simulé le disque ATA (commande Unix `od -x`) ;
- formatage du disque (`frmt`) ;
- visualisation d'un secteur quelconque (`dmps`).

Vous pouvez penser à comparer les résultats de votre commande avec ceux produits *sur le même disque* par la commande de vos camarades.
Pour tester l'écriture sur le disque, il est *nécessaire* de développer un programme ad hoc.
Deux remarques :

1. **Il est illusoire de poursuivre les développements qui s'appuieront sur cette bibliothèque `drive` sans que celle-ci ait été validée.**
2. **Il vous faudra rendre ou démontrer vos programmes de tests lors de l'évaluation de votre travail.**

Question 1.2 En supposant que les variables `int cylinder` et `int sector` contiennent respectivement le numéro de piste et de secteur à lire, expliquez les valeurs à écrire dans les ports pour désigner la position que la piste doit atteindre sur le disque. □

Question 1.3 Réalisez le programme `dmps`. □

Exercice 2 (Formater un disque : `frmt`)
On se propose maintenant d'écrire un programme qui détruit entièrement le contenu d'un disque physique en formatant chaque secteur du disque. Proposez un tel programme. □

Exercice 3 (Une bibliothèque pour l'accès physique : `drive`)
Pour pouvoir simplifier notre tâche dans les développements à suivre, nous nous proposons d'écrire une première série de fonctions utilitaires qui formerons notre bibliothèque `drive` d'accès au périphérique.

```
void read_sector(unsigned int cylinder, unsigned int sector,
                 unsigned char *buffer);
void write_sector(unsigned int cylinder, unsigned int sector,
                  const unsigned char *buffer);
void format_sector(unsigned int cylinder, unsigned int sector,
                   unsigned int nsector,
                   unsigned int value);
```

□

2 Seconde couche logicielle : gestion de volumes

D'un point de vue logique, les secteurs d'un disque sont regroupés pour former un « volume ». Un volume peut correspondre à l'ensemble des secteurs d'un disque donné, mais il arrive qu'un disque soit décomposé en plusieurs volumes distincts, chaque volume représentant une partie de la surface du disque.

Pour définir la place de chaque volume (ou partition) sur le disque on structure le disque. Le premier secteur du disque correspond au « master boot record », MBR. Nous convenons

ici que ce premier secteur définit le nombre de volumes présents sur le disque, la position (en coordonnée piste/secteur) du premier secteur de chaque volume, ainsi que le nombre de secteurs consécutifs associés au volume. On conviendra d'un maximum de 8 volumes présents sur un disque. De plus une information associée à chaque volume permettra de déterminer si le volume est :

- le volume de base pour le système de fichiers ;
- un volume annexe du système de fichiers ;
- un autre type de volume qui ne peut être associé au système de fichiers.

Exercice 4 (Secteur d'amorce primaire)
Proposez une structure de donnée pour stocker les informations inscrites dans le MBR. □

Exercice 5 (Initialisation des volumes)
Proposez une fonction C qui lit le MBR est initialise, avec le résultat de cette lecture, une structure de donnée globale accessible par toutes autres procédures. Cette structure de donnée sera gardée en mémoire durant toute l'utilisation du disque.

Proposez de même une fonction de sauvegarde de la structure de données vers le MBR qui sera appelée en fin d'utilisation du disque. □

Exercice 6 (Conversion d'adressage)
Un bloc est un secteur du disque qui est associé à un volume. Les blocs d'un volume sont contigus. Aussi un bloc est identifié par un simple numéro de bloc relatif au volume. Proposez une formule de conversion qui permette de transformer un couple (numéro de volume, numéro de bloc) en un couple (numéro de cylindre, numéro de secteur). □

Exercice 7 (Bibliothèque d'accès aux volumes : `vol`)
Pour pouvoir utiliser l'organisation en volumes du disque, nous nous proposons de réaliser un ensemble de fonctions qui permettront de lire, écrire ou formater des blocs :

```
void read_bloc(unsigned int vol, unsigned int nbloc,
               unsigned char *buffer);
void write_bloc(unsigned int vol, unsigned int nbloc,
                const unsigned char *buffer);
void format_vol(unsigned int vol);
```

Notez que l'utilisation de ces fonctions suppose que le disque ait été initialisé et que le MBR ait été lu en mémoire. □

Exercice 8 (Gestionnaire de partitions)
Il s'agit de réaliser un petit programme qui permette de lister les partitions présentes sur un disque, de créer une nouvelle partition, de supprimer une partition... Voir l'encart page suivante à ce propos. □

3 Troisième couche logicielle, 1re partie : structure d'un volume

Chaque volume manipulé par notre système de fichiers dispose d'un descripteur de volume, son superbloc, inspiré par celui décrit dans le cours. Il dispose à la base

- d'un mot magique en guise de détrompeur ;
- d'un numéro de série ;
- d'un nom composé au maximum de 32 caractères ;
- d'un identifiant qui donne le lien vers le premier inœud (associé au fichier de base du disque : son répertoire racine).

De plus le système de fichiers utilise un mécanisme de gestion des blocs libres gérés sous forme d'une liste chaînée de blocs.

Squelette d'un gestionnaire de partitions

L'archive

```
/home/enseign/ASE/src/vm-skel.tgz
```

contient le squelette d'un gestionnaire de volumes. Vous pouvez l'utiliser comme point de départ de votre développement d'un gestionnaire de partitions. En particulier cela vous décharge complètement de la gestion de l'interaction avec l'utilisateur.
Il vous faut copier le fichier `vm-skel.c` pour créer un fichier `vm.c` que vous compléterez. Un `Makefile` vous est fourni pour construire les différents exécutables.
Pour faciliter les choses, on peut introduire dans la bibliothèque `drive` d'accès au matériel une fonction `init_master()` qui initialisera le disque maître utilisé. Dans la suite des développements, nous ne nous préoccuperons plus du disque esclave.
La fonction principale de notre gestionnaire de partitions se doit de faire appel aux fonctions d'initialisation de ce disque et se doit de charger le MBR en mémoire :

```
/* init master drive and load MBR */
init_master();
load_mbr();
```

Validation de la bibliothèque de gestion de volumes

Comme précédemment, il s'agit de valider votre travail avant de poursuivre les développements.
Le protocole suivant vérifie un minimum de cohérence dans votre implantation :
– créez une partition `P1` encadrée d'une partition `P0` et d'une partition `P2` ;
– listez l'état de votre disque ;
– détruisez la partition `P1` ;
– listez l'état de votre disque ;
– recréez la partition `P1` ;
– listez à nouveau l'état de votre disque.
En particulier, vous pouvez ou non quitter et relancer le gestionnaire de volume entre chaque étape.

Exercice 9 (Descripteurs de volume)
Définissez les structures de données du superbloc de chaque volume, du chaînage des blocs libres. □

Exercice 10 (Initialiser un volume)
Proposez une fonction

```
void init_super(unsigned int vol);
```

qui permet d'initialiser le superbloc d'un volume, en particulier d'y associer le chaînage des blocs libres. □

Exercice 11 (Sélection et mise à jour d'un volume)
Il s'agit de réaliser les deux fonctions suivantes :

```
int load_super(unsigned int vol);
void save_super();
```

qui permettent respectivement de sélectionner le volume courant en chargeant le superbloc dans une variable globale et de mettre à jour le superbloc chargé en mémoire, après qu'il ait été modifié. □

Exercice 12 (Allouer et libérer des blocs)
Proposer des fonctions de gestion des blocs pour allouer et libérer des blocs dans un volume :

```
unsigned int new_bloc();
void free_bloc(unsigned int bloc);
```

□

Partition courante — Commandes `mknfs` et `dfs`

Dans l'ensemble des programmes qui seront maintenant développés, nous travaillerons sur un unique volume désigné par la valeur de la variable d'environnement `$CURRENT_VOLUME`.
Dans le même esprit, le fichier de configuration de la bibliothèque `hardware` utilisé sera désigné par la variable d'environnement `$HW_CONFIG`. Une valeur par défaut pourra être choisie.
On développera en particulier un programme `mknfs` (*make new filesystem*), pendant de la commande Unix `mkfs`, qui initialisera ce volume courant et un programme `dfs` (*display filesystem*), pendant de la commande Unix `df`, qui affichera l'état des partitions, et pour la partition courante sont taux d'occupation.

Validation de la bibliothèque d'allocation/libération de blocs

Pour valider votre travail, concevez un programme qui
– fait appel à la fonction `new_bloc()` jusqu'à ce qu'elle retourne une erreur ;
– vérifie que le disque est plein ;
– itère un nombre aléatoire de fois sur la libération d'un bloc `free_bloc()` ;
– affiche le statut du disque (taille libre) ;
– alloue des blocs tant que le disque est non plein et retourne le nombre de blocs ayant pu être alloués.

4 Troisième couche logicielle, 2e partie : structure d'un fichier

Chaque fichier ou répertoire est défini par un inœud, tel que le cours le présente, avec un type, une taille de fichier en octets, et les tables de numéro de blocs, « direct », « indirect » et « double indirect ».

Exercice 13 (Structure d'un inœud)
Définissez la structure `struct inode_s` qui sera utilisée pour représenter un inœud. □

Les inœuds sont bien entendu enregistrés sur le disque. Nous choisissons d'enregistrer un unique inœud par bloc. (Cela est inhabituel, la taille d'un inœud étant petite devant celle d'un bloc.) Un inœud est identifié par son inombre qui ne sera dans notre cas rien d'autre qu'un numéro de bloc.

Nous définissons deux fonctions utilitaires pour réaliser l'écriture et la lecture sur le disque d'un inœud.

```
void read_inode(unsigned int inumber, struct inode_s *inode);
void write_inode(unsigned int inumber, struct inode_s *inode);
```

ainsi que deux fonctions de création et destruction d'un inœud :

```
unsigned int create_inode(enum file_type_e type);
int delete_inode(unsigned int inumber);
```

La fonction `create_inode()` est chargé de l'allocation d'un bloc pour y ranger l'inœud, et de l'initialisation de celui-ci. Elle retourne le inombre correspondant.

La fonction `delete_inode()` libère l'ensemble des blocs de données associés à l'inœud puis le bloc de l'inœud lui-même.

Exercice 14 (Lecture et écriture d'inœuds)
Donnez une implémentation des fonctions `read_inode()` et `write_inode()`. □

Exercice 15 (Création et suppression d'un inœud)
Donnez une implémentation des fonctions de création et destruction d'un inœud : `create_inode()` et `delete_inode()`. □

Validation de la troisième couche logicielle : structure d'un volume et d'un fichier

Vous pourrez valider votre bibliothèque de gestion d'un volume et de gestion d'inœuds à l'aide de l'archive que nous vous fournissons et qui permet de produire les commandes listées dans l'encart « Validation de la bibliothèque de manipulation de fichier par leur inombre » page 10. Voir

```
/home/enseign/ASE/src/ifile.tgz
```

Afin de développer la couche logicielle supérieure, nous allons présenter un fichier sous la forme d'une suite continue de blocs de données. Il est donc nécessaire de convertir un indice de bloc d'un inœud en un numéro de bloc dans le volume. Une telle conversion repose sur le parcours de la table des numéros de blocs directs, indirects et double indirects de l'inœud.

Exercice 16 (Bloc volume d'un bloc inœud)
Développez une fonction

```
unsigned int vbloc_of_fbloc(unsigned int inumber,
                            unsigned int fbloc);
```

qui retourne le numéro de bloc sur le volume qui correspond au `fbloc`-ième bloc de l'inœud. Dans un premier temps, cette fonction retourne une valeur nulle si le bloc n'a pas été alloué. □

Cette fonction `vbloc_of_fbloc()` est utilisée pour accéder en lecture à un bloc de données correspondant à un inœud. Si l'on désire accéder en écriture à un tel bloc, il est nécessaire d'allouer ce bloc et de le rattacher à la structure des blocs directs, indirects, double indirects de l'iœud.

Nous ajoutons un paramètre booléen à la fonction `vbloc_of_fbloc()` qui indique si le comportement de la fonction doit être de retourner une valeur nulle ou d'allouer le bloc en cas de bloc non encore existant.

Exercice 17 (Allocatation d'un bloc d'inœud)
Développez la nouvelle version de la fonction

```
unsigned int vbloc_of_fbloc(unsigned int inumber,
                            unsigned int bloc,
                            bool_t do_allocate);
```

qui retourne le numéro du bloc dans le volume qui correspond au `fbloc`-ième bloc de l'inœud. Si ce bloc n'est pas alloué et `do_allocate`, la fonction se charge de l'allocation du bloc et de le connecter à la structure de l'inoœud. □

5 Quatrième couche logicielle : manipulation de fichiers

Pour manipuler les fichiers, les utilisateurs du système disposent de différentes fonctions : création, destruction, ouverture, fermeture, lecture, écriture, positionnement, vidage du tampon d'écriture.

Dans un premier temps, notre système de fichiers identifie un fichier directement par son numéro d'inœud. Nous modifierons cette interface par la suite pour identifier les fichier par des noms.

Ces fonctions sont donc les suivantes (`ifile` pour *inumber file*) :

```
unsigned int create_ifile(enum file_type_e type);
int delete_ifile(unsigned int inumber);

int open_ifile(file_desc_t *fd, unsigned int inumber);
void close_ifile(file_desc_t *fd);
void flush_ifile(file_desc_t *fd);
void seek_ifile(file_desc_t *fd, int r_offset);  /* relatif */
```

```
void seek2_ifile(file_desc_t *fd, int a_offset); /* absolu */

int readc_ifile(file_desc_t *fd);
int writec_ifile(file_desc_t *fd, char c);
int read_ifile(file_desc_t *fd, void *buf, unsigned int nbyte);
int write_ifile(file_desc_t *fd, const void *buf, unsigned int nbyte);
```

Exercice 18 (Création et suppression d'un fichier)
Réalisez la fonction `create_ifile()` en allouant et renseignant un inœud dont le numéro sera retourné. Puis réalisez `delete_ifile()`, qui supprime un fichier en libérant tous les blocs qui peuvent être associés au fichier, y compris le bloc d'inœud. □

Lorsque un fichier est manipulé par un programme, celui-ci utilise la fonction `int open_ifile(file_desc_t *fd, unsigned inumber)` qui « ouvre » un fichier en initialisant une structure de donnée `file_desc_t`. Cette structure de donnée, qui sert la manipulation d'un fichier, contient en fait toutes les informations nécessaires à la gestion d'un fichier. On y trouve notamment :

- le numéro de l'inœud qui décrit le fichier visé ;
- la position en octet du curseur dans le fichier. Cette position donne l'octet à lire/écrire dans le fichier ouvert, et elle est incrémentée après chaque opération d'accès ;
- la taille totale du fichier (utilisée lors des accès en lecture) ;
- un tableau d'octets qui sert de tampon entre le bloc du disque et le programme ;
- un drapeau qui indique si le tampon courant à été modifié ou pas.

Exercice 19 (Structure d'accès au fichier)
Décrivez la structure de données `file_desc_t`. □

Exercice 20 (Accès au fichier)
Proposez d'abord le code pour la fonction `open_ifile()` qui initialise un descripteur de fichier en « ouvrant » le fichier désigné. La fonction `open_sfile()` retourne un statut.

Puis implantez la fonction `flush_ifile()` qui vide le tampon courant sur le disque (si le drapeau indique que le tampon et modifié). Et enfin proposez le programme qui termine l'utilisation d'un fichier : `close_ifile(file_desc_t *fd)`. □

Exercice 21 (Se déplacer dans un fichier)
Les deux fonctions `seek_ifile()` et `seek2_ifile()` déplacent le curseur d'accès dans le fichier ouvert. Ces fonctions doivent mettre à jour le tampon afin que les données du tampon mémoire soient cohérentes avec le curseur dans le fichier. Bien entendu, un déplacement relatif peut être implanté à partir d'un déplacement absolu, ou (exclusif !) l'inverse. □

Exercice 22 (Lire et écrire un octet)
Finalement réalisez les fonctions d'accès : `int readc_ifile()` et `writec_ifile()` qui permettent de lire ou d'écrire un octet dans un fichier. Une fois l'accès réalisé, le descripteur de fichier se place « automatiquement » sur l'octet suivant.

Si la fin de fichier est atteinte l'acte de lecture retourne une valeur négative (`READ_EOF`) et la position dans le fichier n'est pas modifiée.

En cas d'écriture au delà de la fin, le fichier est automatiquement « étendu » en conséquence. Notez encore qu'un accès en lecture sur un bloc qui n'est défini retourne zéro. Un accès en écriture sur le même octet implique l'allocation d'un nouveau bloc, initialisé à zéro avec seulement l'octet écrit de modifié. □

Exercice 23 (Lecture et écriture dans un fichier)
À l'aide des fonctions précédentes, écrivez les fonctions :

```
int read_ifile(file_desc_t *fd, void *buf, unsigned int nbyte);
int write_ifile(file_desc_t *fd, const void *buf, unsigned int nbyte);
```

Validation de la bibliothèque de manipulation d'un fichier par son inombre

Pour valider votre travail, vous pouvez écrire des versions des commandes de l'exercice 27 qui identifie les fichiers paramètres par leur inombre ; la création d'un fichier retournant le inombre associée au fichier :

`if_status` liste les informations associées à la partition courante.

`if_pfile` (*print file*) affiche sur la sortie standard le contenu d'un fichier dont le inombre est passée en paramètre.

`if_nfile` (*new file*) crée un fichier. Le contenu du fichier est lu sur l'entrée standard. Le inombre du fichier est retournée sur la sortie standard ; il servira à identifier le fichier lors de futures commandes.

`if_dfile` (*delete file*) supprime le fichier dont le inombre est passé en paramètre.

`if_cfile` (*copy file*) copie le contenu du fichier dont le inombre est donné en paramètre dans le second fichier. Ce second fichier est donc créé ; son inombre est affiché sur la sortie standard.

qui, respectivement, lit `nbyte` octets depuis le fichier `fd` en les stockant en mémoire centrale dans le tableau de caractère `buf`, et écrit `nbyte` octets lus depuis le tableau de caractère `buf` en mémoire centrale vers le fichier `f`. □

6 Cinquième couche logicielle : système de noms de fichier

Exercice 24 (Structure d'un répertoire)
Les répertoires sont « simplement » des fichiers particuliers. Ils contiennent au minimum une liste de noms (les fichiers et répertoires contenus dans le répertoire) et pour chaque nom un numéro d'inœud associé. Ainsi un répertoire est un tableau dont chaque entrée est le descripteur d'un fichier. Soit `struct entry_s` le type d'un tel descripteur. La taille du tableau est directement calculable en fonction de la taille d'une entrée et de la taille du fichier-répertoire.

Pour simplifier le retrait d'un fichier dans la liste des fichiers d'un répertoire, on identifie les entrées détruites (par exemple avec une valeur remarquable pour un des champs de `entry_s`).

Question 24.1 Donnez la déclaration de la structure de donnée associée à une entrée dans le répertoire : `struct entry_s`. □

Question 24.2 Proposez un programme qui affiche la liste des fichiers contenu dans un répertoire (le répertoire étant identifié par son numéro d'inœud). □

Question 24.3 Proposez les fonctions utilitaires

```
unsigned int new_entry(file_desc_t *dir_fd);
int find_entry(file_desc_t *dir_fd, const char *basename);
```

`new_entry()` retourne l'index dans un répertoire (vu comme un tableau d'entrées) de la première entrée libre. Le répertoire est identifié par un descripteur de fichier, il a donc préalablement été ouvert.

`find_entry()` retourne l'index dans un répertoire d'une entrée dont le nom est donnée, une valeur négative si aucune entrée est trouvée. □

Question 24.4 Proposez maintenant les deux fonctions de création et destruction d'une entrée dans un répertoire connu `add_entry()` et `del_entry()`. □

Exercice 25 (Nom de fichier et inœud)
Il s'agit d'identifier le inœud correspondant à un fichier dont on connaît le nom complet absolu. Cette identification se fait de proche en proche, par exemple pour le fichier `/usr/bin/emacs` :

1. on identifie le inœud de la racine `/` : il est contenu dans le superbloc ;

2. on recherche un fichier nommée `usr` dans le répertoire correspondant à ce inœud, on en identifie le inœud ;
3. on recherche alors un fichier nommé `bin` dans le répertoire correspondant à ce inœud ;
4. etc.

Pour réaliser cette opération, on peut écrire successivement les fonctions suivantes :

```
unsigned int inumber_of_basename(unsigned int idir, const char *basename);
unsigned int inumber_of_path(const char *pathname);
unsigned int dinumber_of_path(const char *pathname, const char **basename);
```

La fonction `inumber_of_basename()` retourne le inombre de l'entrée de nom `basename` (qui ne doit pas comporter de `/`) dans le répertoire `idir`, 0 en cas d'échec.

La fonction `inumber_of_path()` retourne le inombre d'un nom de fichier *absolu*, 0 en cas d'échec.

La fonction `dinumber_of_path()` retourne à la fois le inombre d'un nom de fichier absolu, mais aussi le nom relatif de ce fichier dans son répertoire dans le paramètre `basename`. La valeur retournée pour `basename` est un pointeur dans `pathname`. □

Exercice 26 (Une bibliothèque pour travailler avec les fichiers)
La couche finale de votre travail consiste en une bibliothèque d'accès aux fichiers, avec les fonctions de création, suppression, ouverture, lecture, écriture, déplacement, vidage des tampons et fermeture de fichiers en respectant l'interface donnée dans l'encart page suivante. □

7 Des programmes de base : commandes de manipulation de fichiers

Exercice 27 (Des programmes de base pour le système de fichiers)
En utilisant votre bibliothèque, réalisez les programmes de base suivants. Il travaille tous avec le fichier de configuration désigné par la variable d'environnement `$HW_CONFIG` et la partition désignée par `$CURRENT_VOLUME`, voir l'encart page 7 :

`mknfs` (*make new file system*) crée un système de fichiers sur le volume courant.

`dfs` (*display file system*) liste les informations associées à la partition courante ; voir l'encart page 7

`pdir` (*print directory*) affiche la liste de entrées du répertoire donné en paramètre sous la forme d'un nom absolu.

`ndir` (*new directory*) crée un répertoire dont le nom (absolu) est passé en paramètre.

`pfile` (*print file*) affiche sur la sortie standard le contenu d'un fichier dont le nom (absolu) est passée en paramètre.

`nfile` (*new file*) crée le fichier dont le nom est passé en paramètre. Le contenu du fichier est lu sur l'entrée standard. Si le fichier existait ; il est préalablement détruit.

`dfile` (*delete file*) supprime le fichier ou répertoire (qui doit être vide) dont le nom est passé en paramètre.

`cfile` (*copy file*) copie le contenu du premier fichier dont le nom est donné en paramètre dans le second fichier dont le nom est donné en paramètre.

□

Fin du sujet

Interface de manipulation de fichiers

```
/* Most of the following functions returns RETURN_FAILURE (a <0 value)
   in case of failure. */

/*-------------------------------
  Initialization and finalization
  ------------------------------------------------------------------*/
   One must mount a volume/file system before any other operation.
   The environement variable $CURRENT_VOLUME contains this partition
   number.
   A sole file system mount is allowed. The file system must be umount.
*/

int mount();
int umount();

/*-------------------------------
  File creation and deletion
  ------------------------------------------------------------------*/

/* return RETURN_FAILURE in of failure (pre-existing file, full
   volume...) */
int create_file(const char *pathname, enum file_type_e type);

/* return RETURN_FAILURE in of failure (non pre-existing file, non
   empty directory...) */
int delete_file(const char *pathname);

/*-------------------------------
  File management
  ------------------------------------------------------------------*/

int open_file(file_desc_t *fd, const char *pathname);
void close_file(file_desc_t *fd);
void flush_file(file_desc_t *fd);
void seek_file(file_desc_t *fd, int offset);

/*-------------------------------
  File accesses
  ------------------------------------------------------------------*/

/* return the conversion to an int of the current char in the file.
   Return READ_EOF if the file is at end-of-file. */
int readc_file(file_desc_t *fd);

/* write a char in the file.
   Fail if there is no place on device. */
int writec_file(file_desc_t *fd, char c);

/* read nbytes char from the file and copy them in the buffer.
   Return the number of actually read char; READ_EOF if the file is at
   end-of-file. */
int read_file(file_desc_t *fd, void *buf, unsigned int nbyte);

/* write nbyte char from the buffer on the file.
   Return the number of char writen, RETURN_FAILURE in case of error. */
int write_file(file_desc_t *fd, const void *buf, unsigned int nbyte);
```