USTL
IUP MIAGE
Système d'exploitation

Un mini système de fichiers à la Unix

Philippe DURIF Philippe MARQUET

Mise à jour de février 2001

On présente ici une version simplifiée du système de fichiers d'Unix dont l'implantation peut être l'objet d'un TP... Un système de fichiers réside sur une partie d'un disque que l'on appelle **volume**. Dans votre cas vous ne disposez évidemment pas d'un disque mais vous simulerez un volume avec un fichier Unix traditionnel.

1 Les différentes couches du système de fichiers

L'ensemble du système se décompose en quatre couches, à chacune d'elles correspond une interface utilisable par les couches de niveau supérieur.

Les trois premières sont des interfaces de programmation (API), autrement dit des modules C qui proposent un certain nombre de services :

volume c'est la couche inférieure qui permet de créer, monter, démonter un volume et d'accéder à ses blocs ;

fs implante la structure du système de fichiers ;

fichier propose les opérations habituelles sur les fichiers (ouverture, fermeture, lecture, écriture...).

La quatrième couche est interactive et comporte, d'une part, la commande `ffs` qui fabrique un nouveau système de fichier, d'autre part, l'interpréteur de commandes `shell` permettant d'exploiter un système de fichiers et qui propose une version simplifiée des commandes habituelles d'Unix (`cp`, `cd`, `ls`, `mkdir`, `rmdir`...).

Enfin le module **erreurs** est utilisé par toutes les couches pour gérer les erreurs éventuelles.

La figure 1 représente l'agencement des différentes couches.

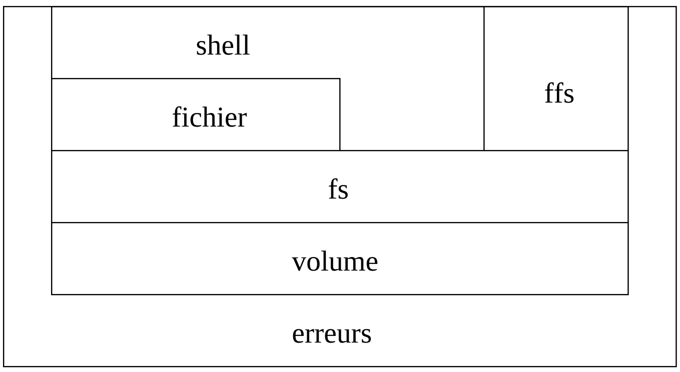

FIG. 1 – Agencement des différentes couches du système de fichiers

Id: fs.tex,v 1.10 2002/02/28 10:31:06 marquet Exp

2 Le volume

Dans le cas du système de fichiers considéré, les volumes et les **partitions** sont confondus : il n'y a qu'une unique partition par volume. Un volume est une suite contiguë de N **blocs** *logiquement* numérotés de 0 à $N-1$. La taille d'un bloc sera fixée à **64 octets** (habituellement c'est plutôt un multiple de 512 ou 1024 ; cette petite valeur doit permettre de tester le système jusque dans les moindres recoins).

L'unité d'accès à un volume est, bien entendu, le bloc. L'API des accès aux blocs du volume est donnée à la figure 2.

```
typedef u_short NUM_BLOC

#define BLOC_SIZE 64
typedef char VOL_BLOC [BLOC_SIZE] ;

int vol_lire      (NUM_BLOC n, VOL_BLOC b) ;
int vol_ecrire    (NUM_BLOC n, VOL_BLOC b) ;
```

FIG. 2 – API des accès aux blocs du volume

3 Structure logique du système de fichiers

Le système de fichiers contient deux sortes d'entités : les **répertoires** et les fichiers **ordinaires**. Les répertoires permettent de structurer le système de fichiers et les fichiers ordinaires contiennent les informations utiles. En fait, ces deux entités sont implantées comme des fichiers avec un fonctionnement un peu particulier pour les répertoires, le terme **fichier** recouvre donc à la fois la notion de répertoire et de fichier ordinaire.

La structuration des répertoires est purement hiérarchique. En plus de ses sous-répertoires et de ses fichiers ordinaires, un répertoire désigne toujours lui-même (pourquoi ?[1]) et son répertoire père[2]. La racine de cette hiérarchie est un répertoire particulier appelé **root** qui est son propre père.

Tout fichier ordinaire est désigné au moins une fois par un ou plusieurs répertoires[3] — sinon il est inaccessible et ne devrait pas exister.

On appelle **lien** toute entrée de répertoire qui désigne un fichier (répertoire ou ordinaire). Un répertoire contient une suite de liens de la forme `{i-nombre,nom}`. Le **i-nombre** est un entier qui permet de retrouver le fichier correspondant (plus exactement, il sert d'indice dans la table des i-nœuds) : il est unique dans le système de fichiers. Le nom est une chaîne d'au plus 14 caractères qui identifie de façon unique une entrée du répertoire. Un répertoire possède toujours au moins deux entrées qui sont créées et détruites en même temps que le répertoire, ce sont `{self,"."}` qui désigne le répertoire lui-même (self est le i-nombre du répertoire) et `{pere,".."}` qui désigne le répertoire père. Dans le cas du répertoire racine, ces deux entrées sont fixées à `{2,"."}` et `{2,".."}` : le i-nombre du répertoire racine est *toujours* 2. Ainsi root est son propre père !

À l'inverse d'un fichier ordinaire, il ne peut y avoir d'autres liens sur un répertoire que ceux créés par la structure même du système de fichiers. En effet cela aurait pour effet de transformer l'arborescence des fichiers en un graphe pouvant contenir des cycles et provoquer des boucles sans fin lors de la résolution de noms de fichiers.

[1] Y a-t-il une réponse satisfaisante à cette question si ce n'est que cela facilite l'expression de certaines opérations ?
[2] Ce qui permet, par exemple, d'exprimer `pwd`.
[3] Un fichier ordinaire peut donc *appartenir* à plusieurs répertoires.

Un fichier est décrit dans le système de fichiers par un unique **i-nœud** qui contient les informations suivantes[4] :

- la catégorie du fichier (ordinaire ou répertoire) ;
- le nombre de liens sur ce fichier (i.e. le nombre d'entrées de répertoire qui désignent ce fichier) ;
- la taille (logique[5]) en octets du fichier ;
- un tableau des 7 premiers blocs du fichier ;
- le numéro d'un bloc de simple indirection donnant les numéros des blocs pour les blocs suivants du fichier ;
- le numéro de bloc de double indirection qui contient lui-même des numéros de blocs de simple indirection pour les derniers blocs du fichier.

Ces informations sont rangées dans une structure de type `INOEUD_OCCUPE` (figure 3).

```
/* Taille d'un bloc */
#define BLOC_SIZE 64

enum CATEGORIE {ORDINAIRE, REPERTOIRE} ;

typedef u_short NUM_BLOC

typedef struct  INOEUD_OCCUPE {
   enum CATEGORIE type ;
   u_short    liens ; /* nombre d'entrees de repertoire referencant ce inoeud */
   u_long     taille ;  /* taille logique en octets du fichier */
#define NB_BLOCS_DIRECTS 7
   NUM_BLOC   direct [NB_BLOCS_DIRECTS] ;  /* les 7 premiers blocs du fichier */
   NUM_BLOC   indirect ;        /* bloc d'indirection pour les blocs suivants */
   NUM_BLOC   double_indirect ; /* bloc de double indirection ... */
} INOEUD_OCCUPE ;

typedef union { /* definition qui sera completee */
   INOEUD_OCCUPE occupe ;
   /* ... */
} INOEUD ;

/* Nombre de numeros de bloc par bloc (Pour les blocs d'indirection) */
#define NUMEROS_PAR_BLOC (BLOC_SIZE / sizeof (NUM_BLOC))
```

FIG. 3 – Définitions de types pour les i-nœuds

Le système de fichiers contient une table des i-nœuds et c'est le i-nombre qui sert d'indice dans cette table.

La figure 4 montre un exemple de i-nœud de répertoire avec les blocs attachés.

La figure 5 montre un exemple de fichier ordinaire.

Question 1

Sachant que les blocs font `BLOC_SIZE` $= 64$ octets et qu'un numéro de bloc tient sur `sizeof`

[4] Par rapport à un Unix classique, on a supprimé la notion de droits d'accès et les informations de date et on a simplifié les types de fichiers.

[5] Pour les fichiers à trous, *cf. infra*, la taille logique est égale au numéro du dernier octet du fichier effectivement écrit ; la taille physique est inférieure si certains blocs n'ont pas été alloués.

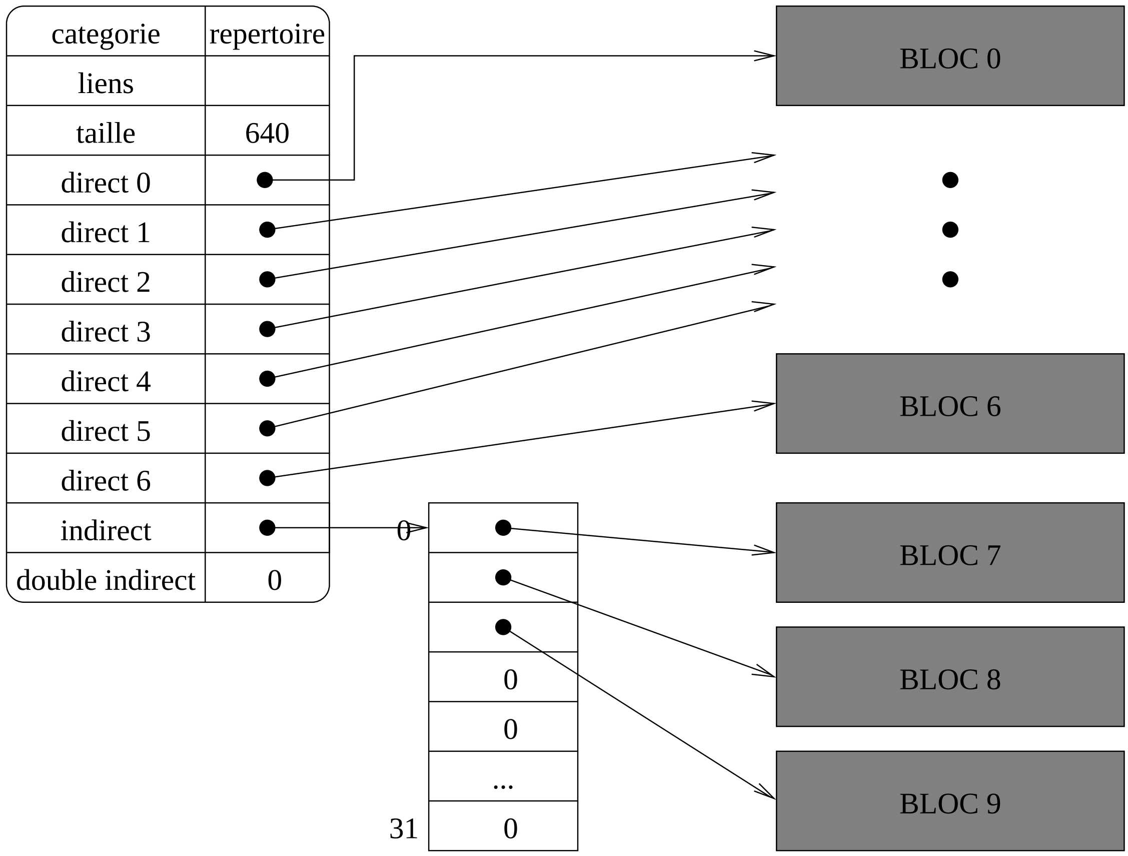

FIG. 4 – Un exemple de i-nœud répertoire. On remarque que le répertoire occupe tout l'espace des blocs utilisés (taille = 640) même si toutes les entrées ne sont pas utilisées. Dans le cas d'un fichier ordinaire, il se peut que le dernier bloc ne soit pas complètement utilisé.

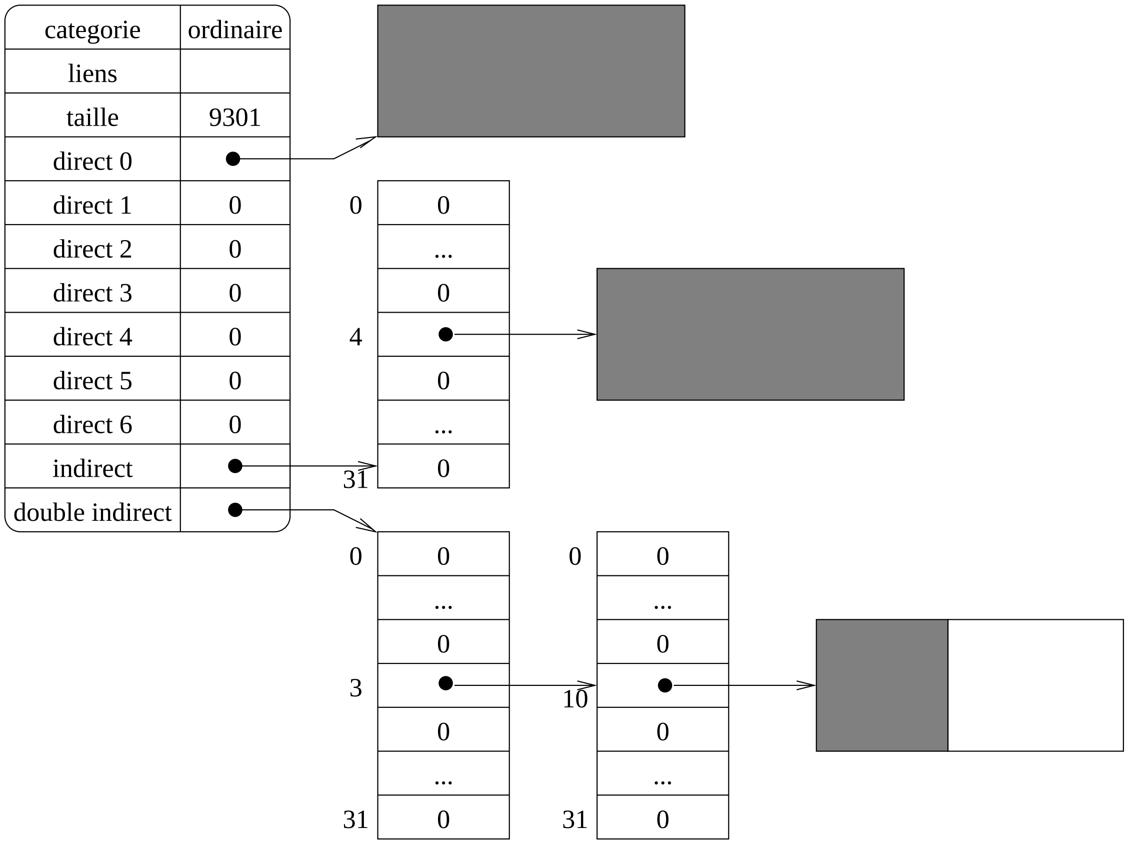

FIG. 5 – Un exemple de i-nœud. On remarque que : (1) la taille n'est pas nécessairement multiple de celle d'un bloc (64 octets), en effet le dernier bloc n'est pas forcément complètement utilisé ; (2) le fichier utilise moins de blocs que ce qui semblerait nécessaire si on regarde sa taille (i.e. il y a des *trous* dans le fichier), ceci est dû au fait qu'il est possible de faire des écritures bien au-dela de la fin du fichier. Lors de la lecture, les trous d'un tel fichier sont considérés comme contenant des 0. Ceci peut paraître étrange, cependant, cela facilite l'implémentation (pas de vérification à faire lors de l'écriture) et permet de gagner de la place lorsqu'on a d'énormes fichiers creux.

`(NUM_BLOC)` $= 2$ octets, donnez la taille logique maximale en octets d'un fichier. Quels sont alors les nombres totaux minimaux et maximaux de blocs utilisés.

Question 2
Vérifiez que la taille du fichier donné dans l'exemple de i-nœud (figure 5) n'est pas débile.

Question 3
Les blocs qui constituent un fichier sont numérotés relativement à partir de 0. Étant donné un i-nœud et un numéro relatif de bloc, donnez l'algorithme qui donne le numéro logique de bloc correspondant. (En particulier, on retourne une valeur nulle pour un bloc non effectivement alloué[6].)

[6] La valeur nulle (`NUM_NULL`) n'est pas une valeur valide pour un numéro logique de bloc, en effet la structure du volume est telle que l'utilisation de ce bloc est réservée.

Question 4
Donner l'algorithme de principe de la commande `du` qui, étant donné un i-nœud, retourne le nombre de blocs (ou d'octets, option `-b`) effectivement utilisés par le fichier.

Question 5
Dessinez les systèmes de fichiers en faisant figurer les répertoires, les i-nœuds avec leurs compteurs de liens et les fichiers dans les cas successifs suivant :

1. le système de fichiers vient d'être créé, il ne contient que le répertoire racine ;
2. on a ajouté un répertoire `usr` dans root ;
3. on a ajouté un fichier `fs.c` dans le répertoire `usr` ;
4. on a créé, dans root, un nouveau lien de nom `xxx` sur le fichier `fs.c`.

Question 6
Lorsqu'un fichier est ouvert (une ou plusieurs fois) il faut différer une demande de destruction jusqu'à la dernière fermeture de celui-ci, avez-vous une idée simple, mais pas forcément judicieuse pour assurer cela ? (indication : on peut utiliser le compteur de liens du fichier).

4 Les noms de fichiers

Pendant l'utilisation d'un système de fichiers il existe toujours un répertoire courant qui est initialement root. On peut désigner un fichier soit de façon absolue (en partant de root), soit de façon relative (en partant du répertoire courant).

Syntaxe des noms de fichier :

nom	::=	*nom-absolu* | *nom-relatif*
nom-absolu	::=	`'/'` *nom-relatif* | `"/"`
nom-relatif	::=	{ *nom-simple*`'/'` } *nom-simple*

Où *nom-simple* est une suite d'au plus 14 caractères différents de `'/'` et `'\0'` et éventuellement vide, auquel cas elle est comprise comme `"."`. Le nom particulier `"/"` désigne évidemment la racine root.

En interne, on peut mémoriser le répertoire courant sous la forme d'une pile de i-nombres. Le sommet de la pile est le i-nombre du répertoire courant, la pile étant constituée des i-nombres des répertoires pères jusqu'à la racine. Une telle pile permet de s'assurer par exemple que l'on ne détruit pas un répertoire du chemain courant. La figure 6 donne l'interface des fonctions de gestion de cette pile sous forme d'un tableau de i-nombres.

```
#define MAXPILE 100
INOMBRE tpile [MAXPILE] ;
int     spile = 0 ;
void    empiler (INOMBRE i_nombre) ;
void    depiler (void) ;
INOMBRE sommet (void) ;
void    vider (void) ;
int     est_en_pile (INOMBRE i_nombre) ;
```

FIG. 6 – Fonctions de gestion de la pile des i-nombres définissant le chemin courant

Question 7
Donnez quelques noms possibles pour le fichier `fs.c` dans le cas où root est le répertoire courant et dans le cas où c'est `usr`.

Question 8
Étant donné le i-nombre du répertoire courant, donnez l'algorithme de principe de `pwd`.

Question 9
Donnez l'algorithme de la commande `cd` qui change le répertoire courant.

Question 10
Étant donné un nom absolu de fichier, donnez l'algorithme qui identifie le inœud du fichier.

5 Structure physique du système de fichiers

Le système de fichiers utilise les blocs du volume sur lequel il réside de la manière suivante :

Bloc 0 **: bloc de boot** Dans notre cas il est inutilisé.

Bloc 1 **: super bloc** Il contient les informations générales du système. En particulier c'est le super bloc qui connaît l'état d'allocation des blocs et des i-nœuds : la zone des blocs libres est formée de deux sous-zones : une liste chaînée et une zone contiguë. Au départ, la zone contiguë contient tous les blocs libres et la liste chaînée est vide (la même technique est utilisée pour la zone des i-nœuds libres). Voici les informations que contient le super bloc ; ces informations sont organisées dans une structure de type `SUPER_BLOC` (figure 7) :
- taille en blocs du volume ;
- nombre total de i-nœuds ;
- premier i-nœud libre dans la liste chaînée des i-nœuds libres. Initialement, cette liste est vide ;
- premier i-nœud libre dans la zone contiguë des i-nœuds libres, initialement c'est le 2 (avant création de la root) ;
- premier bloc libre dans la liste chaînée des bloc libres. Initialement, cette liste est vide ;
- premier bloc libre dans la zone contiguë des bloc libres. Cette zone s'étend depuis depuis ce bloc jusqu'au bloc $N-1$ du volume. Initialement c'est le bloc $k+1$ (avant création du répertoire root) ;
- un numéro magique pour s'assurer qu'au montage d'un volume on a bien à faire à un système de fichiers.

Lors de l'allocation d'un bloc ou d'un inœud, on tente d'abord de l'allouer dans la zone contiguë. Si elle est vide alors on tente l'allocation depuis la liste chaînée. Une libération d'un bloc ou d'un inœud est toujours faite dans la liste chaînée.

Le super bloc doit bien entendu tenir sur un seul bloc.

Blocs 2 **à** k **: table des i-nœuds** Le nombre de i-nœuds est fixé une fois pour toutes à la création du système de fichiers. Chaque bloc de cette table contient un nombre entier de i-nœud (quitte à ce qu'il y ait de la place inutilisée dans ces blocs ; figure 8).

Un i-nombre est un indice dans cette table (à partir de 0). Le i-nombre 0 représente l'absence de i-nœud (ceci permet en particulier de marquer les entrées libres de répertoire), historiquement, le i-nœud 1 était utilisé pour mémoriser la liste des secteurs défectueux du disque, enfin le i-nombre 2 correspond toujours au répertoire racine. Après la création d'un système de fichiers, le premier i-nombre libre est donc toujours 3.

Blocs $k+1$ **à** $N-1$ **: blocs restants** Ces blocs contiennent les informations utiles des fichiers ainsi que les blocs d'indirections.

En particulier, les répertoires sont rangés dans ces blocs selon la définition de type de la figure 9. Remarquez que la taille d'une entrée de répertoire, 16 octets, est un sous-multiple de celle d'un bloc.

```
typedef u_short INOMBRE ;

typedef struct SUPER_BLOC {
   u_short    nb_blocs ;
   u_short    nb_inoeuds ;
   INOMBRE    i_contigu ; /* a partir de ce i_nombre, tous les i_noeuds sont libres *
   INOMBRE    i_chaine  ;
   NUM_BLOC   b_contigu ; /* a partir de ce bloc, tous les blocs sont libres */
   NUM_BLOC   b_chaine  ;
   char       magique [20] ; /* signature du systeme de fichier */
} SUPER_BLOC ;

typedef struct INOEUD_LIBRE {
   INOMBRE i_suiv ;  /* liste chainee des inoeuds libres */
} INOEUD_LIBRE ;

typedef union { /* definition complete */
   INOEUD_OCCUPE occupe ;
   INOEUD_LIBRE  libre ;
} INOEUD ;

typedef struct BLOC_LIBRE {
   NUM_BLOC b_suiv ; /* liste chainee des blocs libres */
} BLOC_LIBRE ;
```

FIG. 7 – Définitions de types pour le super bloc

```
/* Nombre de inoeuds par bloc */
#define INOEUDS_PAR_BLOC (BLOC_SIZE/sizeof (INOEUD [1]))
```

FIG. 8 – Définition complémentaire pour la table des i-nœuds (voir figure 3)

La définition du type `BLOC` unifie les différentes utilisations possibles des blocs (figure 10).

Question 11
Il est dit que « *le super bloc doit bien entendu tenir sur un seul bloc.* » Donner une expression C qui assure que cette affirmation est vraie.

Question 12
Expliquez la définition du type `union BLOC` (figure 10). Donnez la fonction du bloc lors de l'utilisation de chacun des champs de l'union.

Question 13
On considère la commande `ffs` (*faire fs*) de création d'un système de fichiers. Quelles sont les informations à passer à cette commande. Donnez le fonctionnement de principe de la commande. Donnez un schéma d'un système de fichier juste créé.

Question 14
La commande `df` affiche un état de l'utilisation d'un volume. Elle reporte en particulier le nombre de blocs libres sur le volume. L'option `df -i` reporte le nombre d'inœuds libres.

```
#define LONGMAXNOM 14

typedef struct ENTREE {
   INOMBRE i_nombre ;      /* NUM_NULL indique une entree libre */
   char nom [LONGMAXNOM] ;
} ENTREE ;

/* Nombre d'entrees de repertoire par bloc */
#define ENTREES_PAR_BLOC (BLOC_SIZE / sizeof(struct ENTREE [1]))
```

FIG. 9 – Définition de type pour les entrées de répertoire

```
#define BLOC_SIZE 64
typedef char VOL_BLOC [BLOC_SIZE] ;

/* Nombre d'entrees de repertoire par bloc */
#define ENTREES_PAR_BLOC (BLOC_SIZE / sizeof(struct ENTREE [1]))

typedef union BLOC {
   VOL_BLOC opaque ;   /* fixe la taille du BLOC */
   BLOC_LIBRE bloc_libre ;
   SUPER_BLOC super_bloc ;
   INOEUD inoeuds [INOEUDS_PAR_BLOC] ;
   ENTREE entrees [ENTREES_PAR_BLOC] ;
   NUM_BLOC blocs [NUMEROS_PAR_BLOC] ;
} BLOC ;
```

FIG. 10 – Définitions de types pour les blocs

Question 15
La macro-définition `IBLOC(i_nombre)` retourne le numéro logique de bloc contenant le inœud `i_nombre`. La macro-définition `INDICE(i_nombre)` retourne l'indice physique du `i_nombre` dans son bloc.

Question 16
Les entrées libres d'un répertoire sont identifiées par une valeur nulle (`NUM_NULL`) du champ `i_nombre`. Pourquoi est-il nécessaire d'identifier des entrées libres ?

Question 17
Étant donné le répertoire courant, la commande `du -s` retourne le nombre de blocs (ou d'octets, option `-b`) effectivement occupés par le répertoire et l'ensemble de ses entrées. (Voir la question 4.)

Question 18
Vérifiez que tout bloc est soit libre, soit alloué à un i-nœud. Vérifiez qu'un bloc n'est pas alloué à plusieurs inœuds.

Question 19
Vérifiez que toute entrée dans la table des i-nœuds est soit libre soit affectées à une entrée dans le système de fichiers.

Question 20
Étant donné un i-nombre, listez toutes les entrées du système de fichiers qui lui sont associées.

Question 21
Vérifiez la cohérence du compteur de liens d'un répertoire.

Question 22
Reconstruire la liste des blocs libres, idem pour les i-nœuds