Système d'exploitation MIAGE 2 : système de fichiers : structures de données.

Licence MIAGE — Université Lille 1
Pour toutes remarques : Alexandre.Sedoglavic@univ-lille1.fr

Semestre 6 — 2008-09

1 Stratégies de stockage des fichiers

Un fichier est divisé en blocs. Nous allons expliciter quelques points concernant le choix de la taille de ces derniers.

1. On montre que la taille moyenne des fichiers UNIX est de 1Ko. En allouant une taille de 32Ko à chaque bloc, quel pourcentage de l'espace disque est utilisé ?
2. Si on représente les blocs libres par une liste chaînée et que l'on suppose que les blocs font 1Ko alors que leurs numéros physique sont codés sur 16bits, de combien de blocs à besoin un disque de 20Mo pour contenir tous les numéros physique des blocs libres ?
3. Même question si on représente les blocs libres par une table de bits (une bitmap).
4. Dans ce contexte, si on représente un fichier par une liste chaînée de blocs et que l'on désire lire l'octet 2^{15}, combien de blocs doit on parcourir ?
5. Si on représente les blocs libres par une table de bits, quelle serait la taille de cette table pour un disque de 1.3Go avec des blocs de 512Ko.
6. Dans ce codage, donner la formule qui permet d'avoir le premier bloc libre.
7. Si on représente un fichier grâce à une FAT, que la taille du disque soit de 64Mo et que les numéros physique des blocs sont représentés par 2 octets, combien d'espace occupe la FAT ?
8. Un système de fichiers UNIX utilise des blocs de 1Ko et des numéros absolus codés sur 2 octets. Le nœud d'informations renferme 8 adresses de blocs de données, une adresse d'un bloc à simple indirection et une adresse d'un bloc à double indirection. Quelle est la taille maximale d'un fichier ?

2 Une implantation d'un système de fichiers

Ces notes se basent sur les cours, travaux dirigés et travaux pratiques rédigés par Ph. DURIF, Ph. MARQUET et ali. On se propose de reconstruire un système de fichiers. Un disque (ou volume) sera implanté par un fichier UNIX. L'ensemble des procédures permettant cette implantation seront disponibles en travaux pratique.

2.1 Implantation d'un système de fichier

Dans cette séance de travaux dirigés, nous allons approfondir notre compréhension d'un système de fichiers en préparant la séance de travaux pratique.

Tout d'abord, on va créer un fichier qui simule notre disque. Pour ce faire, on doit préciser :
- un nom ;
- un nombre de blocs ;
- un nombre d'inœuds.

Exercice 1 — Espace disque et inœuds.
En fonction de la taille d'un disque à construire, quel nombre d'inœuds vous paraît raisonnable et pourquoi ?

2.1.1 Gestion des blocs

Une fois construit, notre disque virtuel se présente comme une suite contiguë de blocs.

Définition 1 — *Chaque bloc est identifié par un numéro. Ce numéro est le numéro absolu du bloc. La taille des blocs est fixée arbitrairement à* 64 *octets.*

On suppose que l'on dispose d'un certain nombre d'appels système i.e. de procédures permettant d'interagir avec le disque ; pour les utiliser il faut connaître l'Application Programmer Interface (API). Dans notre cas, on a notamment les déclarations suivantes :

```
/* pour la gestion des num\'eros de bloc */
typedef u_short NUM_BLOC
#define NUM_NULL 0 /* num\'ero de bloc ou de inoeud inexistant */
/* on fixe la taille d'un bloc*/
#define BLOC_SIZE 64
typedef char VOL_BLOC [BLOC_SIZE] ;
/* Lit/\'ecrit le "n"i\‘eme (num\'ero absolu) bloc volume dans/depuis la zone m\'emoire b */
int vol_lire   (NUM_BLOC n, VOL_BLOC b) ;
int vol_ecrire (NUM_BLOC n, VOL_BLOC b) ;
enum ERREUR { OK, PARAMETRES_INCONSISTANTS, TABLE_INOEUDS_SATUREE, SYSTEME_DE_FICHIER_SATURE,
LIEN_SUR_REPERTOIRE, REPERTOIRE_INEXISTANT, VOLUME_INEXISTANT, VOLUME_DEJA_MONTE, VOLUME_NON_MONTE,
EXISTE_DEJA, FICHIER_INEXISTANT, PAS_UN_REPERTOIRE, PAS_UN_ORDINAIRE, PAS_UN_SYSTEME_DE_FICHIERS,
 REPERTOIRE_ACTIF, REPERTOIRE_NON_VIDE, LECTURE_IMPOSSIBLE, ECRITURE_IMPOSSIBLE,
ACCES_FICHIER_HORS_LIMITE, OPERATION_IMPOSSIBLE } ;
extern enum ERREUR erreur ;
```

Remarquez que le type `u_short` correspond à `unsigned short` comme `u_long` correspond à `unsigned long`.

En plus des types ci-dessus, on peut ainsi utiliser les fonctions `vol_lire` et `vol_ecrire` sans se soucier de leur implantation.

De plus, pour uniformiser les entrées-sorties, on définit l'union :

```
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 ;
```

Hormis le type `VOL_BLOC`, nous ne donnons pas pour l'instant les définitions exactes des types utilisés ci-dessus. Remarquons qu'un bloc peut contenir soit des données de fichiers, soit un inœud, soit des blocs d'indirections. Ainsi, il n'est pas aisé d'unifier les types de données manipulés par les procédures `vol_lire` et `vol_ecrire`. On peut toutefois utiliser le type `BLOC` de la manière suivante :

```
BLOC b ; NUM_BLOC n = 666 ; vol_lire(n,b.opaque) ;
```

Suivant le contexte, la variable `b` contiendra en sortie un des types définis dans l'union.

API des inœuds. Rappelons qu'un fichier est *décrit* par un inœud. Dans notre implantation d'un système de fichiers, la structure C suivante représente un inœud :

```
#define NB_BLOCS_DIRECTS 7

enum CATEGORIE  {ORDINAIRE, REPERTOIRE} ;

typedef struct  INOEUD_OCCUPE {
  enum CATEGORIE type ;
  u_short       liens ; /* nombre d'entr\'ees de r\'epertoire r\'ef\'eren\c{c}ant
                           cet inoeud */
  u_long        taille ; /* taille logique en octets du fichier */
  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 ;

#define NUMEROS_PAR_BLOC (BLOC_SIZE / sizeof (NUM_BLOC))
```

On retrouve dans cette implantation une partie des informations contenues par un inœud.

Exercice 2 — Compréhension d'un inœud.

À quoi correspond la constante `NB_BLOCS_DIRECTS` ? Que représente le tableau `direct` dans cette structure ? À quoi correspond la constante `NUMEROS_PAR_BLOC` ? Quel serait un choix logique d'initialisation des champs `indirect` et `double_indirect` si la taille du fichier ne nécessite pas leur utilisation ? Quelles autres informations pourrait contenir cet inœud ?

Exercice 3 — Conversion relatif–absolu.

Les blocs qui constituent un fichier sont numérotés arbitrairement à partir de 0. Il s'agit là d'un numéro *relatif*. On se propose d'écrire une procédure qui convertit un numéro de bloc relatif d'un fichier en un numéro *absolu*.

En tenant compte des éléments d'API fournis jusqu'à présent, complétez le programme suivant :

```
static NUM_BLOC get_numero_absolu (NUM_BLOC numRel, INOEUD_OCCUPE *inoeud){
/* traduit le num\'ero relatif en num\'ero absolu */
  return NUM_NULL ;
}
```

Exercice 4 — Taille d'un fichier.

Construire la fonction C qui, étant donné un inœud, retourne le nombre de blocs effectivement utilisés par le fichier (les blocs d'indirection font partis du compte).

En tenant compte des éléments d'API fournis jusqu'à présent, complétez le programme suivant :

```
static int du_inoeud (INOEUD_OCCUPE *inoeud) {
   int nblocs = 0 ;
    return nbblocs ;
}
```

Déplacement dans l'arborescence. Un inombre est associé à chaque inœud. Ce nombre correspond à l'entrée de la *table des inœuds* qui permet de localiser les inœuds sur le disque.

Dans notre système de fichiers, le répertoire courant est représenté par une pile d'inombres. Le sommet de la pile est l'inombre du répertoire courant, la pile étant constituée des inombres des répertoires pères jusqu'à la racine. L'API est notamment constituée par :

```
#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) ;
```

Exercice 5 — Commande `cd`.

Donnez le squelette en pseudo-code de la fonction `cd` qui prend en entrée un nom (`.`, `..` ou `foo` par exemple) et qui modifie la pile. On suppose que l'on dispose de l'API de fonctions raisonnables (localiser l'inombre associé à un nom, etc.).