TP3 : Manipulation et implantation de systèmes de fichiers[1]

Cet exercice est composé de trois parties. Dans les deux premières, on utilise les bibliothèques OCaml de :

```
Unix      http://caml.inria.fr/pub/docs/manual-ocaml/libref/Unix.html
Filename  http://caml.inria.fr/pub/docs/manual-ocaml/libref/Filename.html
```

pour parcourir la hiérarchie du système de fichiers en insistant sur un traitement robuste des erreurs. Dans la troisième partie, en partant d'un programme squelette, on travaille avec un système de fichiers idéalisé, mais pas complètement éloigné de la réalité.

Comme toujours, l'idée est de gagner, face à la machine, une compréhension plutôt viscérale des concepts enseignés dans les cours. Le chargé de TP est là pour vous aider à surmonter les frustrations inévitables (ce n'est pas un examen) et pour répondre à vos questions : profitez-en !

1 Répertoire de travail courant

Le but de l'exercice est d'écrire une commande `mon_pwd` qui affiche le répertoire courant équivalent à la commande `/bin/pwd` du système.

Question 1. Faire cela en une ligne avec la fonction `Unix.getcwd`. Utiliser la fonction

```
Unix.handle_unix_error: ('a -> 'b) -> 'a -> 'b
```

pour récupérer et afficher les éventuelles formes de l'exception `Unix_error` levées dans le module `Unix` lors des appels système.

La commande `Unix.getcwd` n'est pas toujours directement un appel système, mais simplement une fonction écrite en C (qui elle-même fait des appels système). On se propose donc de réécrire la fonction `getcwd` en OCaml. Cette fonction doit pouvoir être utilisée comme fonction de librairie : elle ne doit donc pas modifier les variables globales du programme ni créer une fuite de mémoire. Cette fonction remonte récursivement dans la hiérarchie jusqu'à la racine du système en recherchant à chaque étape le répertoire courant " `.` " dans le répertoire supérieur " `..` ".

Question 2. Écrire une fonction `equal_node: stats -> stats -> bool` qui teste si deux nœuds de la hiérarchie de fichiers sont identiques. Deux nœuds sont identiques si et seulement si leurs numéros de nœuds et leurs numéros de partition sont égaux.

Question 3. Écrire une fonction
`try_finally: ('a -> 'b) -> 'a -> ('c -> 'd) -> 'c -> 'b` qui prend quatre arguments `f`, `x`, `finally` et `y` et qui effectue le calcul `f x`, puis, avant de retourner le résultat, exécute `finally y`, y compris lorsque le résultat est une exception.

1. D'après des exercices qui ont été créés par Didier Remy. Le texte a été créé par Louis Mandel.

Question 4. Écrire une fonction `dir_find: (string -> bool) -> string -> string` qui prend en arguments un prédicat `f: string -> bool` et un nom de répertoire et recherche dans celui-ci le nom d'un fichier qui satisfait le prédicat `f`. Si la fonction trouve le fichier, elle retourne son nom, sinon elle lève l'exception `Not_found`. Pour écrire cette fonction, on utilisera les fonctions `Unix.opendir`, `Unix.readdir` et `Unix.closedir`. Pour ne pas créer une fuite de mémoire, on fera bien attention à refermer le répertoire ouvert, y compris lorsqu'une exception est levée, avant de rendre le résultat ou de relever une exception.

Question 5. Écrire une fonction `mon_getcwd` qui se comporte comme `Unix.getcwd`. L'algorithme manipule des chemins et des nœuds. Les informations sur les nœuds, indispensables à la comparaison de nœuds, sont obtenues par l'appel système `Unix.lstat` qui prend en argument un chemin (on n'utilise pas `Unix.stat` ici, car on recherche un chemin direct issu de la racine qui ne traverse pas de liens symboliques). Pour être portable, on utilisera les fonctions `concat`, `current_dir_name` (`.`) et `parent_dir_name` (`..`) du module `Filename` pour manipuler les chemins. On évitera l'utilisation de `chdir` qui affecterait alors le reste du programme.

Question 6. Terminer le programme. On n'oubliera pas d'appeler `handle_unix_error` pour reporter les messages d'erreurs éventuels qui peuvent se produire pendant le parcours de la hiérarchie.

2 Parcours de l'arborescence (optionnel)

La commande `find` d'Unix permet d'effectuer diverses recherches dans le système de fichiers. Par exemple, elle permet de :

- afficher tous les chemins (fichiers et répertoires) accessibles depuis le répertoire courant :
  `find .`
- afficher tous les chemins accessibles depuis le répertoire courant qui correspondent à des répertoires :
  `find . -type d`
- afficher tous les chemins accessibles depuis le répertoire courant qui correspondent à des répertoires et qui ont une profondeur inférieure à 2 dans l'arborescence :
  `find . -maxdepth 2 -type d`
- afficher tous les chemins accessibles depuis le répertoire home qui correspondent à des fichiers réguliers qui ont été accédés il y a 2 jours :
  `find ~ -type f -atime 2`

On désire écrire une commande `mon_find` qui se limite à un sous-ensemble des possibilités de la commande `find` du système.

Question 7. Écrire une première commande `mon_find0` équivalente à `find . -maxdepth 1`.

Question 8. Écrire une seconde version `mon_find1` équivalente à `find ..`

Question 9. Implanter les comportements équivalents aux options :

- `-L` ;
- `-type` ;
- `-atime` ;

Pour réaliser cette commande, on pourra utiliser le module `Arg` pour récupérer les arguments de la ligne de commande.

3 Implantation d'un système de fichier

Dans cette partie, nous allons étudier l'implantation d'un système de fichiers simple. Le disque dur sera modélisé par un fichier et le système par votre programme.

3.1 Description du système de fichiers

Le système de fichier est paramétré par trois valeurs :
- `nb_block`, le nombre total de blocs sur le disque
- `block_size`, la taille d'un bloc du disque en octets
- `inode_nb`, le nombre total d'inodes sur le disque

Le fichier contenant ce système de fichier est composé de :
- un superbloc (un bloc),
- `inode_nb` blocs inode,
- `nb_block - inode_nb - 1` blocs de données, et
- la valeur de `block_size` sur 4 octets.

La taille totale du fichier est donc `nb_block * block_size + 4`.

Le superbloc (premier bloc de la partition) contient dans cet ordre :
- `inode_nb`, le nombre total d'inodes sur ce disque
- `last_free_inode`, le numéro du bloc contenant le dernier inode libéré.
- `free_block_list`, le numéro d'un bloc index de la liste des blocs libres.
- `free_block_nb`, le nombre de blocs libres.
- `root_inode`, le numéro de l'inode correspondant au répertoire racine de la partition (toujours 1 dans notre cas)
- à la fin du block, `fstype`, une chaîne de 4 caractères permettant d'identifier le type de fichier. Dans notre cas, il doit s'agir de `sfs0`.

Les entiers sont stockés en quatre octets en ordre big-endian (une fonction `read_int` est fournie dans le squelette).

Le numéro d'un inode correspond au numéro du bloc le contenant. Chaque inode contient :
- `kind`, le type du fichier, un entier dont la valeur est 0 pour un fichier et 1 pour un répertoire,
- `nlink`, le nombre de liens durs vers ce fichier. Si cette valeur est nulle, l'inode est considéré comme libre, et peut être utilisé pour la création d'un nouveau fichier.
- `size`, la taille du fichier en octets,
- `blocks[]`, les numéros des blocs contenant les données du fichier. Le bloc 0 correspond à un bloc rempli de zéros, non encore alloué.

La liste des blocs libres est formée de blocs index : chaque bloc index contient les numéros des blocs libres, ainsi que le numéro du bloc index suivant en position 0. Lorsqu'on souhaite utiliser un bloc, et qu'il ne reste plus de blocs libres dans le bloc index courant, on utilise cette valeur comme nouveau bloc index, et on retourne l'ancien bloc index comme bloc à utiliser. Une valeur de bloc index de 0 indique qu'il n'y a plus de blocs libres dans le système de fichiers.

Finalement, les répertoires sont composés d'une suite d'entrées (`dirent`) contenant un nom de fichier sur `filename_max_size` octets, terminé par un caractère zéro s'il est plus court que `filename_max_size`, et d'un numéro d'inode. `filename_max_size` est fixée à 28 et une entrée prend donc toujours 32 octets. Une entrée d'un répertoire peut être inutilisée (si un fichier a été ajouté puis supprimé), auquel cas le nom du fichier est simplement vide (c.-à-d. le premier octet de l'entrée est 0). Pour rajouter un fichier à un répertoire, il faut donc soit trouver une entrée vide, soit ajouter une entrée au répertoire.

Un exemple de fichier partition `simplefs.bin` et le squelette du programme se trouvent à :

`http://www.di.ens.fr/~pouzet/cours/systeme/tp03/tp-03.tgz`

3.2 Vérification

Le but maintenant consiste à s'assurer que le fichier partition précédent n'est pas corrompu avant de l'utiliser. Pour cela, on désire effectuer les vérifications suivantes :

- Le compteur de liens durs de chaque inode correspond bien au nombre de fois où le fichier apparaît dans un répertoire.
- Un bloc utilisé par un fichier n'apparaît jamais dans la liste des blocs libres.
- Tout bloc non utilisé apparaît bien dans la liste des blocs libres.
- Un bloc utilisé par un fichier n'est jamais utilisé par un autre fichier.

Question 10. Écrire une fonction `open_partition: string -> partition` qui prend en argument un nom de fichier et retourne une valeur de type `partition` initialisée à partir du contenu du fichier (fonctions utiles : `Unix.openfile`, `Unix.lseek`, `String.sub`). Tester votre fonction en utilisant la fonction `print_partition`. On devrait trouver :

```
Partition simplefs.bin (alive)
  block_size: 1024
  block_nb: 20048
  block_word_size: 256
  i-nodes
    blocktbl_size: 253
    max_file_size: 259072
    blocktbl_offset: 12
  supernode
    inode_nb: 2048
    last_free_inode: 1650
    free_block_list: 9473
    free_block_nb: 7530
    fstype: sfs0
    root_inode: 1
```

Question 11. Écrire une fonction `lseek_block: partition -> int -> unit` qui positionne le descripteur de fichier de la partition au début du bloc dont le numéro est donné en argument.

Question 12. Écrire une fonction `read_block: partition -> int -> string` qui retourne une chaîne de caractère contenant le contenu du bloc dont le numéro a été passé en argument.

Question 13. Écrire une fonction `read_inode: partition -> int -> inode` qui retourne la valeur de type `inode` associée au numéro d'inode fourni en argument de la fonction.

Notre but est d'écrire une fonction `fsck` qui va vérifier l'état du système de fichiers. Pour cela, nous avons besoin de parcourir l'arborescence des fichiers sur la partition, à partir de l'inode d'un répertoire.

Nous définissons un type `file_descr` pour les descripteurs de fichiers, que nous allons utiliser pour manipuler les fichiers en général, et dans le cas immédiat, les répertoires :

```
type file_descr = { inode : inode; mutable pos : int; mutable closed : bool }

let open_inode inode =
  { inode = inode;
    pos = 0;
    closed = false; }
```

La fonction `open_inode` ci-dessus associe un descripteur de fichier à un inode.

Question 14. Écrire la fonction correspondant à l'appel système `read` en suivant l'interface de la fonction `Unix.read`. Cette fonction provoque les erreurs `EINVAL` si les arguments sont invalides (descripteur fermé, chaîne trop courte), ainsi que `EIO` si le fichier est plus grand que la taille maximale.

Question 15. Écrire la fonction `read_dirent: file_descr -> int * string`, qui lit l'entrée suivante du répertoire dont le descripteur a été passé en argument, et qui retourne la paire composée du numéro d'inode et du nom de fichier, ou lève l'exception `End_of_file` si la fin du répertoire est atteinte. Tester votre fonction en utilisant la fonction `print_directory`.

Notre fonction `fsck` doit calculer le type de chaque bloc et son utilisation, afin de vérifier la cohérence du système de fichier. Nous définissons donc ici les valeurs que peuvent prendre les différents blocs du système.

```
type block_type =
  | Supernode       (* Supernode du système *)
  | Inode of int    (* Inode, nombre de liens durs observés *)
  | Block of bool   (* Block, utilisé ou pas *)
  | IndexBlock      (* Block d'index de la liste des blocs libres *)
  | FreeBlock       (* Block libre confirmé *)
```

Question 16. Écrire la fonction `fsck` :

- Créer un tableau contenant le type de chaque bloc au départ.
- Parcourir l'arborescence des fichiers à partir de la racine.
  - Ne parcourir qu'une fois chaque inode, mais ne pas oublier de changer son type dans le tableau pour correspondre aux nombres de liens durs observés.
  - Marquer chaque bloc utilisé par un fichier.
  - Se plaindre si un bloc est utilisé par plusieurs fichiers.
  - Se plaindre si des blocs non nuls apparaissent après la fin du fichier.
- Parcourir la liste des blocs libres, et vérifier :
  - Que chaque bloc noté libre est bien inutilisé
  - Que tous les blocs inutilisés apparaissent dans cette liste.