Systèmes d'exploitation **GLO-2001**
Philippe Giguère H16

TRAVAIL PRATIQUE N^{o} 3 (Ver 1.0)

Implémentation d'un système de fichier UFS simplifié en langage C, avec montage FUSE

Travail en équipe de 1 à 2

De nombreux systèmes de fichiers modernes (ext2, ext3 et ext4, par exemple) se basent sur le système de fichier Unix File System (UFS). Le but de ce travail est de vous familiariser avec :

- l'utilisation plus poussée des pointeurs et des structures en langage C;
- les structures utilisées dans un système de fichier UFS;
- les opérations nécessaires pour gérer ces fichiers et les impacts sur les métadonnées;
- la traduction entre un nom de fichier et un numéro d'*i-node*;
- les patrons d'accès au disque résultants d'opérations simples comme `mv`;
- la manipulation des fichiers à l'aide de l'abstraction des *i-nodes*;
- la différence entre un lien symbolique et un lien hard;
- comment monter un tel système de fichier dans Linux via la libraire FUSE (Filesystem in Userspace).

Important : vous devez utiliser le rapport `TP3-rapport-glo2001-h16.rtf` avec ce TP lors de la remise de votre travail. N'oubliez-pas d'attacher TOUT le code source (en zip) avec votre remise.

Partie 1 Implémentation (30%) : Implémentez certaines fonctions d'un système de fichier simple, calqué sur l'Unix File System (UFS), en C. Ce code doit compiler et tourner sur la machine virtuelle Linux 32 bits du cours. Avant de commencer, assurez-vous de lire AU COMPLET cette partie 1, et aussi la partie 2 subséquente qui donne un script de test.

Le système de fichier est stocké dans un disque virtuel placé en stéganographie dans l'image `google-go.png`, où l'on utilise le dernier bit de chaque pixel pour stocker de l'information. Ce disque possède 40 blocs de 256 octets. Tous les fichiers ou répertoires mentionnés dans ce TP sont contenus dans ce fichier de disque simulé et camouflé (ne les cherchez donc pas dans le fichier .tar!). Pour accéder aux blocs du disque camouflé, vous devez utiliser obligatoirement les fonctions suivantes, qui sont implémentés dans `DisqueStegano.c` :

```
int ReadBlock(UINT16 BlockNum, char *pBuffer[1]);
int WriteBlock(UINT16 BlockNum, const char *pBuffer);
```

Ces deux fonctions simulent la lecture et l'écriture sur un disque, pour un numéro de bloc `BlockNum` spécifié.

Attention! Assurez-vous d'écrire les modifications faites sur le système de fichier sur le disque simulé avec `WriteBlock` avant qu'une opération ne soit terminée. Par exemple, si vous modifiez le bitmap des blocs libres et des *i-nodes*, ou les métadonnées d'un fichier, vous devez sauvegarder ce bloc avant la fin de la fonction. Ainsi, le disque simulé stocké dans l'image `google-go.png` sera consistant après l'exécution de n'importe quelle fonction. Aussi, les blocs des *i-nodes* contiennent plus d'un *i-node*. Faites attention lorsque vous manipuler deux *i-nodes* contenus dans un même bloc. Prenez pour acquis qu'il n'y aura pas d'autres processus qui ouvrent des *i-nodes*; vous n'aurez donc pas besoin de vérifier s'il est ouvert en ce moment.

---

[1] Pourquoi un tableau de **`char`**? Eh bien, le **`char`** est défini comme un octet dans les compilateurs, sur toutes les machines. Ainsi, un tableau de **`char`** est simplement un tableau d'octets, ou si vous préférez, un bloc de mémoire où chacun des octets est adressable directement. Il ne s'agit donc pas de lire et d'écrire des caractères ASCII lisibles, comme « A » ou « z ».

Structure du système de fichier

Le périphérique de stockage sera simulé dans ce TP par stéganographie de l'image `google-go.png`, donnant un disque camouflé de taille `DISKSIZE`=10,240 octets (40 x 256), avec une copie identique nommée `google-go.png.orig` utilisé pour le script de test. Chaque bloc dans ce disque simulé a une taille de `BLOCK_SIZE`=256 octets, avec `N_BLOCK_ON_DISK`=40 blocs dans ce disque. Le numéro d'*i-node* sera encodé avec un entier non-signé de 16 bits (`UINT16`). Le nombre d'*i-nodes* par bloc est défini par `NUM_INODE_PER_BLOCK`, qui est égal à 16.

La structure du système de fichiers est la suivante :
bloc 0 : *N'y touchez jamais (c'est le boot block)*
bloc 1 : *N'y touchez jamais (c'est le superblock)*
bloc 2 : Pseudo-bitmap des blocs libres
bloc 3 : Pseudo-bitmap des *i-nodes* libres
bloc 4 : *i-nodes* 0 à 15 (dont le répertoire racine, qui est l'*i-node* 1)
bloc 5 : *i-nodes* 16 à 31
blocs 6-39 : blocs de données.

Note : sur un vrai système UFS, le bloc 1 est le *superblock* qui contient des pointeurs vers les listes chaînées des blocs libres et des *i-nodes* libres. Pour le TP, ils sont plutôt implémentés comme des pseudo-bitmaps dans les blocs 2 et 3, à la manière d'`ext2` et d'`ext3`.

Pour simuler le pseudo-bitmap des blocs libres, simplement utiliser le bloc `FREE_BLOCK_BITMAP` comme un tableau de caractère où chaque élément indique si un bloc est libre (valeur non-nulle) ou pas (0). Par exemple, si vous avez la variable `char data[BLOCK_SIZE]` qui contient le bloc `FREE_BLOCK_BITMAP`, le test `(data[3]==0)` indiquera que le bloc 3 n'est pas libre. De la même manière `(data[6]!=0)` indiquera que le bloc 6 est libre. Notez ici que je n'utilise pas le caractère '`0`' mais bel et bien la valeur numérique 0. Pour relâcher le bloc `BlockNum`, vous pouvez utiliser le code suivant (faites de même pour le bitmap des *i-nodes* libres) :

```
int ReleaseFreeBlock(UINT16 BlockNum) {
      char BlockFreeBitmap[BLOCK_SIZE];
      ReadBlock(FREE_BLOCK_BITMAP, BlockFreeBitmap);
      BlockFreeBitmap[BlockNum] = 1;
      printf("GLOFS: Relache bloc %d\n",BlockNum);
      WriteBlock(FREE_BLOCK_BITMAP, BlockFreeBitmap);
      return 1;
}
```

Les noms de fichiers valides ont une longueur maximale de 13 caractères, excluant le caractère de terminaison de chaîne `NULL`. Les caractères permis sont l'alphabet, le point, et les chiffres. Pour alléger votre programme, pas besoin de vérifier la validité du string : simplement s'assurer que le fichier ou le répertoire indiqué est présent ou non. Par exemple `moi.txt` est valide, mais `ceciesttroplong.txt` ne l'est pas. Les répertoires sont séparés par le caractère *slash* (`/`).

Important! Pour chaque acquisition d'un bloc de données libre, vous devez imprimer un message à l'écran. Vous devez faire de même lorsque vous relâchez ce bloc et qu'il devient libre à nouveau. Ces messages auront la forme suivante :

```
GLOFS: Saisie bloc 25
GLOFS: Relache bloc 25
```

De la même manière, à chaque fois que vous saisissez un nouvel *i-node* ou que vous le relâchez, affichez le message ayant le format suivant :

```
GLOFS: Saisie i-node 5
GLOFS: Relache i-node 5
```

où 5 est le numéro de l'*i-node* en question. Ces messages vous aideront à mieux analyser votre programme, et nous aiderons à corriger votre solution. En particulier, ils vous permettront de voir si vous effacez réellement un fichier du disque.

Structure d'un *i-node*

Dans le TP, un *i-node* est représenté par la structure suivante :

```
struct iNodeEntry {
      gstat[2] iNodeStat;
      UINT16 Block[N_BLOCK_PER_INODE][3]; // numero des blocs de donnees du fichier
      char   junk[4];
};
```

Le tableau `Block` contient les numéros de blocs utilisé pour emmagasiner les données du fichier. La valeur de `N_BLOCK_PER_INODE` étant 1, vos fichier n'auront donc jamais plus d'un bloc de données, afin de grandement simplifier les opérations de lecture et d'écriture. Il n'y aura pas non plus de redirection simple, double ou triple pour votre système de fichier. La chaine `junk` ne sert à rien (sauf à donner une taille de puissance 2 à l'*i-node*). Au final, la taille d'un *i-node* sera de 16 octets, ce qui permettra d'en placer 16 par bloc de données (`NUM_INODE_PER_BLOCK`).

La structure `gstat` de l'*i-node* contient les métadonnées suivantes :

```
struct gstat {
      ino    st_ino;    // numero de l'i-node
      UINT16 st_mode;   // drapeaux G_IFREG, G_IFDIR et G_IFLNK et les permissions RWX
      UINT16 st_nlink;  // nombre de liens pointant vers l'i-node
      UINT16 st_size;   // taille des données du fichier, en octets. Peut être 0.
      UINT16 st_blocks; // nombre de blocs de donnees associé au fichier. Ici 0 ou 1.
};
```

Le champ `st_ino` contient le numéro de l'*i-node*. Le champ `st_mode` sera un entier 16 bit non-signé qui continent les drapeaux suivants :

- type de fichier : `G_IFREG` si c'est un fichier standard, `G_IFDIR` si c'est un fichier répertoire et `G_IFLNK` si c'est un lien symbolique;
- permissions de lecture/écriture/exécution :

```
#define G_IRWXU  0700   // Permissions rwx pour User
#define G_IRUSR  0400   // Permission r pour User
#define G_IWUSR  0200   // Permission w pour User
#define G_IXUSR  0100   // Permission x pour User
#define G_IRWXG  0007   // Permissions rwx pour Group
#define G_IRGRP  0004   // Permissions r pour Group
#define G_IWGRP  0002   // Permissions w pour Group
#define G_IXGRP  0001   // Permissions x pour Group
```

(En C, il vous faut utiliser les *bitwise operator* ou opérateur bit-à-bit (`&`, `|`, `^`, `~`)[4] pour tester/setter un bit. Par exemple, si vous voulez tester si un fichier représenté par le pointeur d'inode `pInode` est un répertoire, vous pouvez faire le test suivant :

```
if (pInode->iNodeStat.st_mode & G_IFDIR)
```

Pour mettre à 1 un drapeau, vous pouvez utiliser un « ou » logique `|` (montré ici avec une opération d'affectation composée) :

```
pInode->iNodeStat.st_mode |= G_IFREG;
```

Pour effacer un drapeau particulier, faites plutôt

```
pInode->iNodeStat.st_mode &= ~G_IRWXG;
```

qui ici mettra à zéro tous les bits reliés aux permissions rwx de Group. Vous trouverez d'autres exemples ici : http://stackoverflow.com/questions/3920307/how-can-i-remove-a-flag-in-c .)

---

[2] dans un vrai *i-node*, la structure est plutôt `stat`.

[3] déclarer un array de taille 1 est un peu overkill. Cette formulation permet, cependant, de facilement modifier le code pour supporter plus d'un bloc par fichier.

[4] en contraste avec les opérateurs logiques `&&` et `||`

Le champ `st_nlink` sert à compter le nombre de lien vers cet *i-node*. Le champ `st_size` indique la taille des données (en octets) dans le fichier (pas la taille des métadonnées). Finalement, le champ `st_blocks` indique le nombre de blocs de données utilisés par un fichier. Un fichier vide aura `st_size=0` et `st_blocks=0`. Quand vous modifiez la taille d'un fichier, n'oubliez donc pas de mettre à jour ces deux champs, si nécessaire.

Structure d'un répertoire

Comme dans UFS, le nom des fichiers dans votre système de fichier est emmagasiné dans un répertoire par un fichier spécial; son champ `Block[0]` contiendra le numéro de l'unique bloc où les données du fichier répertoire sont stockées. En effet, pour ce TP nous allons assumer qu'un répertoire n'utilise qu'un seul bloc de données (ce qui vous simplifie la vie!). La structure de ce fichier répertoire sera un tableau de `DirEntry` :

```
struct DirEntry {
      ino  iNode;
      char Filename[FILENAME_SIZE];
};
```

Le champ `iNode` constitue le numéro de l'*i-node* du fichier (qui peut être un sous-répertoire). Les 14 premiers octets (`Filename`) sont la chaîne de caractère du nom de fichier (*null-terminated string*). Ce tableau est emmagasiné dans un bloc de donnée. **Attention!** Un répertoire vide contient obligatoirement les deux répertoires suivants : « . » et « .. ». Le premier contient son propre *i-node* comme champ `iNode`, et le deuxième l'*i-node* du répertoire parent (sauf pour le répertoire racine, qui sera lui-même son propre parent). N'oubliez-pas de mettre à jour `st_nlink` pour les i-nodes de . et .. !

Une façon simple et rapide de lire l'entrée **n** d'un répertoire dont les données sont stockées dans le bloc 27 est la suivante :

```
char DataBlockDirEntry[BLOCK_SIZE];
ReadBlock(27, DataBlockDirEntry); // En supposant que le bloc 27 contienne les données du répertoire
DirEntry *pDE = (DirEntry *)DataBlockDirEntry;
printf("Le nieme fichier du repertoire : inode %d avec nom %s\n",pDE[n].iNode, pDE[n].Filename);
```

La taille de ce tableau sera variable, car il dépendra du nombre de fichiers/répertoires présents. Pour savoir le nombre d'entrées dans ce tableau, vous pouvez diviser la taille du fichier répertoire (`st_size`) par `sizeof(DirEntry)`. N'oubliez pas qu'il y a toujours au moins deux répertoires dans un répertoire quelconque, soit « . » et « .. ». Le nombre maximal d'entrées dans ce tableau sera donc `BLOCK_SIZE/ sizeof(DirEntry)`.

Manipulation de chaîne de caractères (string) en C

Pour savoir si deux strings sont parfaitement identiques, utilisez `strcmp`. Par exemple, pour `const char *pPath` :

```
      if (strcmp(pPath,"/")==0)
```

sera vrai si `pPath` contient un string identique à `"/"`.
Pour copier un string : `strcpy`
Pour la longueur d'un string : `strlen`
Pour chercher un caractère dans un string (comme un `'/'` lors de l'analyse d'un path) : `strrchr`
Pour comparer deux chaînes de caractères, utilisez `strcmp()` ou `strncmp()`.
À PROSCRIRE : `strtok`. Cette fonction modifie le string passé en argument, et induit de nombreux bugs bizarres. J'ai vu plus d'une équipe en arracher lorsqu'ils l'ont utilisé. Ne l'utilisez donc pas!
Consultez la section FAQ pour d'autres réponses.
Au besoin, consultez http://www.tutorialspoint.com/cprogramming/c_strings.htm

Commentaires généraux

Pour toutes les fonctions à implémenter, vous devez vous assurer que le fichier ou le répertoire est présent sur le disque. Le caractère « / » est utilisé pour séparer les répertoires. Les chemins d'accès n'ont pas plus de 255 caractères, incluant le `NULL` à la fin.

Liste des fonctions à implémenter dans le fichier `UFS.c`

```
int bd_countfreeblocks();
```
Cette fonction retourne le nombre de bloc de données libres sur le disque.

```
int bd_stat(const char *pFilename, gstat *pStat);
```
Cette fonction copie les métadonnées `gstat` du fichier `pFilename` vers le pointeur `pStat`. Les métadonnées du fichier `pFilename` doivent demeurer inchangées. La fonction retourne -1 si le fichier `pFilename` est inexistant. Autrement, la fonction retourne 0.

```
int bd_create(const char *pFilename);
```
Cette fonction vient créer un fichier normal vide (bit `G_IFREG` de `st_mode` à 1, taille=0, donc sans bloc de données) avec le nom et à l'endroit spécifié par le chemin d'accès `pFilename`. Par exemple, si `pFilename` est égal à `/doc/tmp/a.txt`, vous devez créer le fichier `a.txt` dans le répertoire `/doc/tmp`. Si ce répertoire n'existe pas, retournez -1. Assurez-vous aussi que ce fichier n'existe pas déjà, auquel cas retournez -2. Pour les permissions rwx, simplement les mettre toutes à 1, en faisant `st_mode|=G_IRWXU|G_IRWXG`. Retournez 0 pour indiquer le succès de l'opération.

```
int bd_read(const char *pFilename, char *buffer, int offset, int numbytes);
```
Cette fonction va lire `numbytes` octets dans le fichier `pFilename`, à partir de la position `offset`, et les copier dans `buffer`. La valeur retournée par la fonction est le nombre d'octets lus. Par exemple, si le fichier fait 100 octets et que vous faites une lecture pour `offset`=10 et `numbytes`=200, la valeur retournée par `bd_read` sera 90. Si le fichier `pFilename` est inexistant, la fonction devra retourner -1. Si le fichier `pFilename` est un répertoire, la fonction devra retourner -2. Si l'`offset` fait en sorte qu'il dépasse la taille du fichier, cette fonction devra simplement retourner 0, car vous ne pouvez pas lire aucun caractère. Notez que le nombre de blocs par fichier est limité à 1, ce qui devrait simplifier le code de lecture.

```
int bd_mkdir(const char *pDirName);
```
Cette fonction doit créer le répertoire `pDirName`. Si le chemin d'accès à `pDirName` est inexistant, ne faites rien et retournez -1, par exemple si on demande de créer le répertoire `/doc/tmp/toto` et que le répertoire `/doc/tmp` n'existe pas. Assurez-vous que l'*i-node* correspondant au répertoire est marqué comme répertoire (bit `G_IFDIR` de `st_mode` à 1). Si le répertoire `pDirName` existe déjà, retournez avec -2. Pour les permissions rxw, simplement les mettre toutes à 1, en faisant `st_mode|=G_IRWXU|G_IRWXG`. Assurez-vous aussi que le répertoire contiennent les deux répertoires suivants : « . » et « .. ». N'oubliez-pas d'incrémenter `st_nlink` pour le répertoire actuel « . » et parent « .. ». En cas de succès, retournez 0.

```
int bd_write(const char *pFilename, const char *buffer, int offset, int numbytes);
```
Cette fonction va écrire `numbytes` octets du `buffer` dans le fichier `pFilename`, à partir de la position `offset` (0 indique le début du fichier). La valeur retournée par la fonction est le nombre d'octets écrits. Vous devez vérifier que :

- Le fichier `pFilename` existe. Dans le cas contraire, cette fonction devra retourner -1.
- Le fichier `pFilename` n'est pas un répertoire. Dans le cas contraire, cette fonction devra retourner -2.
- L'`offset` doit être plus petit ou égal à la taille du fichier. Dans le cas contraire, cette fonction devra retourner -3. Par contre, si l'`offset` est plus grand ou égal à la taille maximale supportée par ce système de fichier, retournez la valeur -4.
- La taille finale ne doit pas dépasser la taille maximale d'un fichier sur ce mini-UFS, qui est dicté par le nombre de bloc d'adressage direct `N_BLOCK_PER_INODE`. et la taille d'un bloc. Vous devez quand même écrire le plus possible dans le fichier, jusqu'à atteindre cette taille limite. La fonction retournera ce nombre d'octet écrit.

N'oubliez-pas de modifier la taille du fichier `st_size`.

```
int bd_hardlink(const char *pPathExistant, const char *pPathNouveauLien);
```
Cette fonction créer un *hardlink* entre l'*i-node* du fichier `pPathExistant` et le nom de fichier `pPathNouveauLien`. Assurez-vous que le fichier original `pPathExistant` n'est pas un répertoire (bit `G_IFDIR` de `st_mode` à 0 et bit `G_IFREG` à 1), auquel cas retournez -3. Assurez-vous aussi que le répertoire qui va contenir le lien spécifié dans `pPathNouveauLien` existe, sinon retournez -1. N'oubliez-pas d'incrémenter la valeur du champ `st_nlink` dans l'*i-node*. Assurez-vous que la commande fonctionne aussi si le lien est créé dans le même répertoire, i.e.

```
bd_hardlink("/tmp/a.txt","/tmp/aln.txt")
```

Si le fichier `pPathNouveauLien`, existe déjà, retournez -2. Si le fichier `pPathExistant` est inexistant, retournez -1. Si tout se passe bien, retournez 0.

```
int bd_unlink(const char *pFilename);
```
Cette fonction sert à retirer un fichier normal (`G_IFREG`[5] à 1) du répertoire dans lequel il est contenu. Le retrait se fait en décrémentant de 1 le nombre de lien (`st_nlink`) dans l'*i-node* du fichier `pFilename` et en détruisant l'entrée dans le fichier répertoire dans lequel `pFilename` se situe. Si `st_nlink` tombe à zéro, vous devrez libérer cet *i-node* et ses blocs de données associés. Si après `bd_unlink` le nombre de lien n'est pas zéro, vous ne pouvez pas libérer l' *i-node*, puisqu'il est utilisé ailleurs (via un hardlink). N'oubliez-pas de compacter les entrées dans le tableau de `DirEntry` du répertoire, si le fichier détruit n'est pas à la fin de ce tableau. Si `pFilename` n'existe pas retournez -1. S'il n'est pas un fichier régulier `G_IFREG`, retournez -2. Autrement, retourner 0 pour indiquer le succès.

```
int bd_truncate(const char *pFilename, int NewSize);
```
Cette fonction change la taille d'un fichier présent sur le disque. Pour les erreurs, la fonction retourne -1 si le fichier `pFilename` est inexistant, -2 si `pFilename` est un répertoire. Autrement, la fonction retourne la nouvelle taille du fichier. Si `NewSize` est plus grand que la taille actuelle, ne faites rien et retournez la taille actuelle comme valeur. **N'oubliez-pas de marquer comme libre les blocs libérés par cette fonction, si le changement de taille est tel que certains blocs sont devenus inutiles. Dans notre cas, ce sera si on tronque à la taille 0.**

[5] Un lien symbolique est aussi un fichier régulier. Il faudra donc le retirer du répertoire.

`int bd_rmdir(const char *pDirname);`
Cette fonction sert à effacer un répertoire vide, i.e. s'il ne contient pas d'autre chose que les deux répertoires « . » et « .. ». Si le répertoire n'est pas vide, ne faites rien et retournez -3. N'oubliez-pas de décrémenter `st_nlink` pour le répertoire parent « .. ». Si le répertoire est inexistant, retourner -1. Si c'est un fichier régulier, retournez -2. Autrement, retournez 0 pour indiquer le succès.

`int bd_rename(const char *pFilename, const char *pFilenameDest);`
Cette fonction sert à déplacer ou renommer un fichier ou répertoire `pFilename`. Le nom et le répertoire destination est le nom complet `pFilenameDest`. Par exemple, `bd_rename("/tmp/a.txt","/doc/c.txt")` va déplacer le fichier de `/tmp` vers `/doc`, et aussi renommer le fichier de `a.txt` à `c.txt`. N'oubliez-pas de retirer le fichier (ou répertoire) du répertoire initial. Aussi, faites attention à ne pas faire d'erreur si vous manipuler le compteur de lien. Si le fichier `pFilename` est un répertoire, n'oubliez-pas de mettre à jour le numéro d'*i-node* du répertoire parent « .. ». En cas de succès, retournez 0. Attention! Il se peut que le répertoire soit le même pour `pFilename` et `pFilenameDest`. Votre programme doit pouvoir supporter cela, comme dans le cas `bd_hardlink`. Si le fichier `pFilename` est inexistant, ou si le répertoire de `pFilenameDest` est inexistant, retournez -1. Pour vous simplifier la vie, vous n'avez pas besoin de gérer le cas invalide où le fichier déplacé `pFilename` est un répertoire, et la destination `pFilenameDest` est un sous-répertoire de celui-ci.

`int bd_readdir(const char *pDirLocation, DirEntry ** ppListeFichiers);`
Cette fonction est utilisée pour permettre la lecture du contenu du répertoire `pDirLocation`. Si le répertoire `pDirLocation` est valide, il faut allouer un tableau de `DirEntry` (via `malloc`) de la bonne taille et recopier le contenu du fichier répertoire dans ce tableau. Ce tableau est retourné à l'appelant via le double pointeur `ppListeFichiers`. Par exemple, vous pouvez faire :

`(*ppListeFichiers) = (DirEntry*)malloc(`*taille_de_données*`);`

et traitez `(*ppListeFichiers)` comme un pointeur sur un tableau de `DirEntry`. La fonction `bd_readdir` retourne comme valeur le nombre de fichiers et sous-répertoires contenus dans ce répertoire `pDirLocation` (incluant . et ..). S'il y a une erreur, retourner -1. L'appelant sera en charge de désallouer la mémoire via `free`.

`int bd_symlink(const char *pPathExistant, const char *pPathNouveauLien);`
Cette fonction est utilisée pour créer un lien symbolique vers `pPathExistant`. Vous devez ainsi créer un nouveau fichier `pPathNouveauLien`, en prenant soin que les drapeaux `G_IFLNK` et `G_IFREG` soient tous les deux à 1. La chaîne de caractère `pPathExistant` est simplement copiée intégralement dans le nouveau fichier créé (pensez à réutiliser `bd_write` ici). Ne pas vérifier la validité de `pPathExistant`. Assurez-vous que le répertoire qui va contenir le lien spécifié dans `pPathNouveauLien` existe, sinon retournez -1. Si le fichier `pPathNouveauLien`, existe déjà, retournez -2. Si tout se passe bien, retournez 0. ATTENTION! Afin de vous simplifier la vie, si une commande est envoyée à votre système UFS sur un lien symbolique, ignorez ce fait. Ainsi, si `/slnb.txt` pointe vers `/b.txt` et que vous recevez la commande `./ufs read /slnb.txt 0 40`, cette lecture retournera `/b.txt` et non pas le contenu de `b.txt`. La commande suivante `bd_readlink` sera utilisée par le système d'exploitation pour faire le déréférencement du lien symbolique, plus tard quand nous allons le monter dans Linux avec FUSE.

`int bd_readlink(const char *pPathLien, char *pBuffer, int sizeBuffer);`
Cette fonction est utilisée pour copier le contenu d'un lien symbolique `pPathLien`, dans le buffer `pBuffer` de taille `sizeBuffer`. Ce contenu est une chaîne de caractère représentant le path du fichier sur lequel ce lien symbolique pointe. Cette fonction permettra ainsi au système de fichier, une fois montée dans Linux, de déréférencer les liens symboliques. Si le fichier `pPathLien` n'existe pas ou qu'il n'est pas un lien symbolique, retournez -1. Sinon, retournez le nombre de caractères lus.

Conseils

N'hésitez-pas à faire de nombreuses petites fonctions pour vous aider. Par exemple, faites des fonctions pour

- saisir et relâcher des *i-nodes*;
- lire et écrire un *i-node* sur le disque virtuel;
- saisir et relâcher des blocs de données;
- traduire un nom de fichier en *i-node*;

Aussi, dans ma solution, la plupart des fonctions bd_* s'occupaient de faire la traduction des strings en numéro d'*i-nodes*, puis le vrai travail est fait avec des helper functions acceptant principalement des numéros d'*i-nodes*. De cette manière, vous allez pouvoir réutiliser beaucoup de code d'une fonction à l'autre. Commencez aussi par les fonctions qui ne modifient pas le disque, i.e. `bd_countfreeblocks`, `bd_stat`, `bd_readdir`, `bd_read`. Puis progressez vers `bd_hardlink`, `bd_unlink`, `bd_create` (qui pourra réutiliser du code/helper function de `bd_hardlink` pour insérer le fichier dans le répertoire), `bd_rename` (qui au fond, n'est qu'un link suivi d'un unlink), `bd_rmdir` (encore là, possibilité de réutiliser du code relié à `bd_unlink`) et `bd_write`, et ainsi de suite.

Au final, mon fichier `ufs.c` solution fait environ 1000 lignes de code.

Structure de l'arbre de fichiers:

Le disque simulé fourni dans ce tp est pré-formaté. Il contient la structure suivante (notez le lien symbolique `slnb.txt` dans le répertoire racine) :

Répertoire /

```
drwxrwx               . size:     112 inode:  1 nlink: 5
drwxrwx              .. size:     112 inode:  1 nlink: 5
drwxrwx             doc size:      48 inode:  2 nlink: 3
drwxrwx             rep size:      32 inode:  3 nlink: 2
drwxrwx         Bonjour size:      48 inode:  4 nlink: 2
-rwxrwx           b.txt size:      29 inode:  7 nlink: 1
lrwxrwx        slnb.txt size:       7 inode: 10 nlink: 1
```

Répertoire /doc

```
drwxrwx               . size:      48 inode:  2 nlink: 3
drwxrwx              .. size:     112 inode:  1 nlink: 5
drwxrwx             tmp size:      48 inode:  5 nlink: 3
```

Répertoire /doc/tmp

```
drwxrwx               . size:      48 inode:  5 nlink: 3
drwxrwx              .. size:      48 inode:  2 nlink: 3
drwxrwx          subtmp size:      48 inode:  6 nlink: 2
```

Répertoire /doc/tmp/subtmp

```
drwxrwx               . size:      48 inode:  6 nlink: 2
drwxrwx              .. size:      48 inode:  5 nlink: 3
-rwxrwx           b.txt size:       0 inode:  8 nlink: 1
```

Répertoire /rep

```
drwxrwx               . size:      32 inode:  3 nlink: 2
drwxrwx              .. size:     112 inode:  1 nlink: 5
```

Répertoire /Bonjour

```
drwxrwx               . size:      48 inode:  4 nlink: 2
drwxrwx              .. size:     112 inode:  1 nlink: 5
-rwxrwx     LesAmis.txt size:       0 inode:  9 nlink: 1
```

La structure du disque vous est donnée pour vous aider à débugger votre code. En aucun cas les fonctions que vous créez ne peuvent utiliser l'information sur cette page directement. Par exemple, le numéro d'i-node de /doc/tmp/subtmp/b.txt est 8, mais votre programme doit retrouver cette valeur automatiquement : vous ne pouvez pas hardcoder cette valeur « 8 » nulle part dans votre programme.

Foire aux questions (FAQ)

1) *J'ai de la difficulté à bien saisir comment utiliser les blocs de données avec les DirEntry, pour manipuler les répertoires.*
Il suffit de *caster* un pointeur de **`DirEntry`** sur le bloc lu du disque, et d'accéder à chacune des entrées comme s'il s'agissait d'un tableau de ces derniers. Par exemple, voici du code arbitraire pour tester si l'entrée à l'index `iEntree` des données du fichier répertoire correspond au fichier **`toto.txt`** :

```
UINT16 DirBlockNum = 25; // Choix purement arbitraire ici
char DataBlockDirEntry[BLOCK_SIZE];

ReadBlock(DirBlockNum, DataBlockDirEntry);
DirEntry *pDirEntry = (DirEntry *)DataBlockDirEntry;

// et pour vérifier si l'entrée iEntree correspond au fichier toto.txt
if (strcmp("toto.txt",pDirEntry[iEntree].Filename)==0) {
     // On a trouvé l'entrée, on copie le numéro d'inode correspondant
     iNodeNumber = pDirEntry[iEntree].iNode;
}
```

De la même manière, vous pouvez remplir les données pour un répertoire vide comme suit :

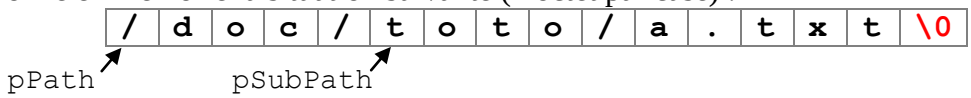
```
char BlockDirEntry[BLOCK_SIZE];
UINT16 BlockNumFile;
     . . .
DirEntry *pDirEntry = (DirEntry *)BlockDirEntry;
strcpy(pDirEntry[0].Filename,".");
pDirEntry[0].iNode = myInode;
strcpy(pDirEntry[1].Filename,"..");
pDirEntry[1].iNode = ParentInode;

WriteBlock(BlockNumFile, BlockDirEntry);
     ...
```

2) *Comment faire une copie du reste du chemin (path) sans les* n *premiers caractères?*
Il faut d'abord bien comprendre comment fonctionnent les chaînes de caractères en C. Elles ne sont qu'une suite des caractères ASCII (1 octet par caractère), qui se terminent par la valeur 0 (pas le caractère zéro, mais la valeur NULL, qu'on représente par '**`\0`**' en C.) Et souvent, il n'est pas nécessaire de faire des copies, comme dans les cas où l'on n'a pas besoin de modifier le string en question. Par exemple, supposons que vous parcourez le string `pPath` qui correspond au chemin d'accès complet `/doc/toto/a.txt`. Vous avez trouvé le numéro d'*i-node* du répertoire de base `doc`, et vous voulez donc continuer le traitement avec la sous-chaîne `toto/a.txt` . Il suffit simplement de faire avancer le pointeur d'autant de caractère que dans `/doc/` (ici 5 caractères) sans avoir besoin de faire de copie :

```
char *pPath = "/doc/toto/a.txt";
...
...
char *pSubPath = pPath+5;
```

Ce qui donne en mémoire la situation suivante (1 octet par case) :



Le string `pSubPath` correspondra à la chaîne `"toto/a.txt"`. Ceci vous permet « d'éplucher » un path progressivement sans faire de copies, comme dans le cas de la traduction d'un path complet vers l'*i-node* du fichier. Vous pouvez donc faire une récurrence pour faire la traduction.

3) *Comment lire un nom de fichier/répertoire au complet dans un string pPath jusqu'au prochain "/" ou jusqu'à "\0" et faire une copie de ce mot dans pFilename?*
(Attention! n'utilisez pas `strtok`, car il modifie le string passé en argument, entraînant une multitude de bugs mystérieux.) Vous pouvez `strrchr` pour trouver un / dans la chaine (ou la fin), et copier avec `strcpy`. Ou simplement y aller manuellement en faisant une boucle sur chaque caractère comme suit (dans cet exemple, je commence au début de la chaîne `pPath`. Si vous voulez commencer ailleurs dans `pPath`, voir question 2 plus haut) :

```
/* J'assume que pPath est un char *, passé en argument par exemple.
   J'assume aussi que pFilename sera utilisé uniquement dans cette
   fonction/scope, car il se trouve sur la pile. */
char pFilename[FILENAME_SIZE];
int iCar;
for (iCar=0; iCar<FILENAME_SIZE; iCar++) {
     if (pPath[iCar] == 0) {
          break;
     }
     if (pPath[iCar] == '/') {
          break;
     }
     pFilename[iCar] = pPath[iCar];
}
pFilename[iCar] = 0; // On ajoute le NULL à la fin
/* Continuez votre traitement sur pFilename. Attention! NE RETOURNEZ
PAS LE POINTEUR pFilename de cette fonction, car il est sur la pile! */
```

4) *Comment traiter les chemins débutant par "./" ou "../", c'est à dire, identifier l'i-node courante?*
Tous les paths fournis à vos fonctions débuteront par la racine `/` . Donc vous n'avez pas à traiter les cas mentionnés plus haut. Notez que votre système va naturellement être capable de traiter les cas comme `/doc/..` ou `/doc/.`, car `.` et `..` sont des répertoires valides.

5) *J'ai de la difficulté à déterminer comment implémenter un Bitmap en C.*
Vous n'avez pas à en implémentez un, à proprement parler. Relisez avec attention la section sur la gestion des blocs de données.

6) *J'ai un segfault pour bd_readdir(const char *pDirLocation, DirEntry **ppListeFichiers), qui semble venir du double pointeur (que ne ne comprends pas comment utiliser).*
Cette fonction doit allouer la mémoire, et la passer via ce double pointeur. Votre code qui traite ce double pointeur *ppListeFichiers* pourrait donc ressembler à ceci :

```
(*ppListeFichiers) = (DirEntry*)malloc(InodeDuRepertoire.iNodeStat.st_size);
memcpy( (*ppListeFichiers),  DonnéesDuRepertoire, InodeDuRepertoire.iNodeStat.st_size);
```

Partie 2 (60 %) Test des fonctionnalités implémentées

Lorsque vous faites la commande

```
make ufs
```

le programme `ufs` sera créé. Ce programme va utiliser vos fonctions `bd_*` pour faire la manipulation du système de fichier virtuel. À chaque appel de `ufs`, il va exécuter la commande de manipulation de fichier passée en argument, avec les arguments supplémentaires requis. Le parsing est fait pour vous dans `main_ufs.c`, et va appeler la fonction `bd_*` appropriée. Par exemple, pour voir les métadonnées `stat` du fichier `/b.txt`, la commande sera :

```
./ufs stat /b.txt
```

ce qui appellera votre fonction `bd_stat()`. Pour faire la liste des fichiers dans un répertoire, faites :

```
./ufs ls nom_de_repertoire
```

ce qui appellera vos fonctions `bd_stat()` et `bd_readdir()`. Pour voir le nombre de blocs libres restant sur le disque simulé, la commande sera

```
./ufs blockfree
```

Pour créer un fichier, la commande sera

```
./ufs create nom_de_fichier
```

Pour retirer (et potentiellement effacer) un fichier d'un répertoire :

```
./ufs unlink droits nom_de_fichier
```

Pour détruire un répertoire vide :

```
./ufs rmdir nom_de_repertoire
```

Pour créer un nom de fichier :

```
./ufs create nom_de_fichier
```

Pour déplacer ou renommer un fichier :

```
./ufs rename ancien_nom nouveau_nom
```

Pour créer un nouveau répertoire

```
./ufs mkdir nom_de_repertoire
```

Pour lire un fichier

```
./ufs read nom_de_fichier offset nombre_doctets_a_lire
```

Pour écrire dans un fichier

```
./ufs write nom_de_fichier "string à écrire" offset
```

Pour créer un hardlink

```
./ufs hardlink nom_de_fichier nom_du_lien
```

Pour créer un lien symbolique

```
./ufs symlink nom_de_fichier nom_du_lien
```

Comme ce programme `ufs` quitte après l'exécution de chaque commande, cela va vous permettre de vous assurer que le disque simulé sur `google-go.png` est consistant après chaque commande. Si vous avez oublié de sauvegarder des blocs (données, bitmap ou *i-nodes*), ces tests devraient faire ressortir ces faiblesses.

Afin de tester les fonctionnalités de votre programme, le script de test `TestEtudiant.sh` vous est donné. Pour vous aider à mieux comprendre ce que vos fonctions doivent accomplir, j'ai inclus la sortie d'écran de ma solution dans le fichier `MesResultats.txt`. J'ai aussi laissé mon exécutable dans la soumission, sous le nom `ufs_complet` . Attention, je ne garantis pas qu'il soit 100% sans bugs.

Pour la correction, nous allons faire tourner notre propre script automatisé de correction (non-fourni). Le score dépendra des résultats de ce test.

Partie 3 (10 %) montage du système de fichier dans Linux via FUSE

FUSE (Filesystem in USErspace, http://fuse.sourceforge.net/) est une librairie permettant de développer des systèmes de fichier dans Linux entièrement dans l'espace utilisateur. Grâce à cette libraire, nous allons donc pouvoir monter (*mount*) votre système de fichier dans l'arborescence, et ainsi le naviguer et sauvegarder des fichiers comme si c'était un vrai!

Avant de débuter, assurez-vous que le compte etu1 est sudoers avec le test suivant :

```
sudo ls
```

(le mot de passe est « etudiant »). S'il retourne un message d'erreur « etu1 is not in the sudoers file. » vous devez ajouter les droits sudo à etu1. Pour ce faire, changez d'identité dans la fenêtre *terminal* avec la commande

```
su etu2
```

avec comme mot de passe « etudiant ». Puis faites la commande

```
sudo adduser etu1 sudo
```

Ce qui ajoutera les droits sudo à etu1. Quitter le profil etu2 en faisant

```
exit
```

L'utilitaire d'installation de package Debian permet d'installer sans trop de peine les fichiers nécessaires pour compiler des nouvelles applications FUSE. Faites la commande

```
sudo apt-get install libfuse-dev
```

et acceptez les changements pour installer les fichiers nécessaires[6].

Pour compiler le progamme, faites :

```
make glofs
```

Pour pouvoir monter votre système de fichier, vous devez créer tout d'abord un répertoire. Faites la commande

```
mkdir /tmp/glo
```

pour créer ce répertoire. Puis, pour démarrer votre système de fichier à partir du répertoire qui contient votre exécutable et le fichier `google-go.png`, faites :

```
./glofs /tmp/glo -f -ouse_ino
```

Avec l'option `-f`, les `printf` dans votre programme vont s'afficher dans la fenêtre terminal (ce qui facilitera le débuggage), et l'option `-ouse_ino` fait en sorte que FUSE rapportera vos numéros d'*i-node*. Pour arrêter le programme, faites ctrl-c. À partir de l'arrêt de `glofs`, votre système de fichier devient inutilisable dans linux (vous devriez voir des erreurs comme `Noeud final de transport n'est pas connecté` dans la fenêtre terminal.) Notez que si vous oubliez de mettre `-f` , vous devrez manuellement démonter le système de fichier via la commande :

```
fusermount -u /tmp/glo
```

Après le démarrage de votre programme, ouvrez une deuxième fenêtre terminal. Déplacez-vous dans ce répertoire `/tmp/glo` et testez les fonctionnalités des commandes systèmes suivantes, en utilisant des fichiers/répertoires contenu dans `google-go.png` :

- `mv`
- `ls`
- `ls -la`
- `ls -i`

[6] les packages `xxx-dev` contiennent généralement les headers, de la documentation de développement et des fichiers d'exemples pour pouvoir compiler des programmes utilisant une libraire xxx. Ils ne sont pas inclus dans les distributions de base afin de sauver de l'espace disque.

- `mkdir`
- `touch`
- `ln`
- `rmdir`
- `cd`
- `cp`
- `gedit`
- `more`
- `ln -s`

et ainsi de suite, en utilisant comme argument vos fichiers.

Vous devriez constatez aussi que :

- l'utilisation de la touche [tab] pour l'auto-complétion fonctionne (et provoque de multiples appels à votre libraire);
- la traduction des chemins relatifs est faite automatiquement par le système d'exploitation; et
- la traduction des liens symboliques est aussi faite automatiquement par le système d'exploitation.

Dans votre rapport, incluez la sortie d'écran de vos tests des commandes linux. Vos tests devraient être *raisonnablement* complets. Au besoin, écrivez un fichier script bash.

Si le message d'erreur suivant apparait « Périphérique ou ressource occupé », créez un nouveau répertoire dans `/tmp` et utilisez-le en argument avec la commande `glofs`.