Département d'informatique et de génie logiciel
Pavillon Pouliot

GLO-2001
Systèmes d'exploitation pour ingénieurs
Travail pratique 3 – Hiver 2019
Implémentation d'un système de fichier UFS simplifié en langage C, avec montage FUSE

Conditions

Machine virtuelle: Pour la machine virtuelle, vous devez utiliser celle sur le portail de cours

Norme : Le travail doit respecter la norme de programmation départementale présentée dans pixel.fsg.ulaval.ca >Applications >Utilitaires >Documents institutionnels>Normes de programmation

Remise : Vous devez suivre la procédure de remise de travaux disponible sur le portail de cours. Vous y trouverez aussi les dates de remise pour le travail préliminaire et le travail final.

Travail à faire **exclusivement en équipe de 2**. Toute communication entre équipes ou avec toute personne autre que l'enseignant est **strictement défendue**. Les politiques sur le plagiat, le français et le retard énoncées dans le plan de cours sont totalement applicables. Des logiciels de vérification de plagiat seront utilisés.

Vous avez des questions ? (ou même si vous n'en avez pas, allez regarder celles des autres) Vérifiez le document des questions fréquentes et des précisions sous l'énoncé dans la page du Travail sur le portail. Il se peut que votre question ait déjà été répondue. Si votre réponse ne s'y trouve pas, alors contactez l'enseignant. Il est là pour ça ☺

Biens livrables

1) Rapport de temps préliminaire TP3
   a) En utilisant le gabarit Excel fourni sur le portail de cours, faites un rapport de temps préliminaire (un **document Excel**) incluant pour chaque question (sous-questions à tous les niveaux) du travail une estimation des heures de travail que vous devrez effectuer pour terminer le travail.
   b) Remettre sur le portail de cours dans la page Rapport de temps préliminaire tp 3.
2) Travail pratique 3
   a) Dans un fichier .zip
      i) Votre rapport de temps final. C'est-à-dire le rapport de temps préliminaire (gabarit Excel .xlsx) dont la colonne temps réel a été complétée ainsi que le ratio temps réel divisé par temps estimé
      ii) Tous les fichiers de code source .c/.cpp et autres que vous avez créés, utilisés ou modifiés. Notez que le code doit compiler sur la machine virtuelle fournie, puisque cette machine est celle du correcteur. Des erreurs de compilations entraîneront des pénalités pouvant aller jusqu'à 50 %.
   b) Remettre sur le portail de cours dans la page Travail pratique 3 > Boîte de dépôt de la section Évaluations et résultats.

Objectifs

Ce travail vise à vérifier l'atteinte, en tout ou en partie, de ces objectifs du plan de cours :

1. Connaître les rôles, les fonctions et les composantes d'un système d'exploitation, et comprendre les principes théoriques de son fonctionnement. Ces principes comprennent :
    4. La gestion des systèmes de fichiers,
2. Comprendre les mécanismes, les méthodes et algorithmes utilisés dans la mise en application de ces principes théoriques.
3. Pouvoir utiliser certains systèmes d'exploitation courants (Linux, Windows).
4. Pouvoir développer des logiciels utilisant ou effectuant des fonctions système
5. Apprendre à évaluer la charge de travail
6. Développer de bonnes aptitudes de communication

Contexte

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*;

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 du portail de cours.

L'espace disque est représenté par un fichier de 65 536 octets qui s'appelle `DisqueVirtuel.dat`. Conséquemment, tous les fichiers ou répertoires mentionnés dans ce TP sont contenus dans ce fichier de disque simulé (ne les cherchez donc pas dans le fichier .tar!). Pour accéder aux blocs du disque simulé, vous devez utiliser obligatoirement les fonctions suivantes, qui sont fournis dans `disque.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 le disque simulé `DisqueVirtuel.dat`, pour un numéro de bloc `BlockNum` spécifié. Au besoin, regarder le code dans `disque.c` pour comprendre.

[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 ».

Important! Il vous est interdit de tous charger le disque en mémoire! Vous devez seulement accéder et modifier les blocs pertinents sur `DisqueVirtuel.dat`.

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 représenté par le fichier `DisqueVirtuel.dat` sera consistant après l'exécution de n'importe quelle fonction.

Structure du système de fichier

Le périphérique de stockage sera simulé dans ce TP par un fichier binaire de taille `DISKSIZE`=65,536 octets (256 x 256) portant le nom `DisqueVirtuel.dat`. Chaque bloc dans ce disque simulé a une taille de `BLOCK_SIZE`=256 octets, ce qui signifie que vous avez au maximum 256 blocs dans votre système de fichiers. Le numéro d'*i-node* sera encodé avec un entier non-signé de 16 bits (`UINT16`). La structure du système de fichiers est la suivante :
bloc 0 : *laisser libre (c'est le boot block)*
bloc 1 : *laisser libre (c'est le superblock)*
bloc 2 : Bitmap des blocs libres.
bloc 3 : Bitmap des *i-nodes* libres.
bloc 4 : i-nodes 0 à 7 (dont le répertoire racine, qui est l'i-node 1)
bloc 5 : i-nodes 8 à 15
bloc 6 : i-nodes 16 à 23
bloc 7 : i-nodes 24 à 31
blocs 8-255 : blocs de données.

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

Pour simuler le 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 qu'on n'utilise pas le caractère '`0`' mais bel et bien la valeur numérique 0.

Faites de même pour le bitmap des *i-nodes* libres. Par exemple, pour relâcher le bloc `BlockNum`, vous pouvez utiliser le code suivant :

```
int ReleaseFreeBlock(UINT16 BlockNum) {
     char BlockFreeBitmap[BLOCK_SIZE];
     ReadBlock(FREE_BLOCK_BITMAP, BlockFreeBitmap);
     BlockFreeBitmap[BlockNum] = 1;
     printf("Relache block %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]; // numero des blocs de donnees du fichier
};
```

Le tableau `Block` contient les numéros de blocs utilisés pour emmagasiner les données du fichier. La valeur de `N_BLOCK_PER_INODE` est 11. Il n'y aura pas de redirection simple, double ou triple pour votre système de fichier : toutes ces entrées sont donc des numéros de blocs. Au final, la taille d'un *i-node* sera de 32 octets, ce qui permettra d'en placer 8 par bloc de données (`NUM_INODE_PER_BLOCK`).

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

```
struct gstat {
    ino    st_ino;    // numero de l'i-node
    UINT16 st_mode;   // G_IFREG ou G_IFDIR. Contient aussi 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 bloc de donnees associé au fichier.
};
```

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, et `G_IFDIR` si c'est un fichier répertoire;

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

- 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 (`&`, `|`, `^`, `~`)[3] 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 .)

Le champ `st_nlink` sert à compter le nombre de lien vers cet *i-node*. Le champ `st_size` indique la taille des données 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.

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. L'*i-node* du répertoire aura donc son champ `Block[0]` qui contient le numéro du bloc où les données sont stockées. Pour vous simplifier la vie, assumez qu'un répertoire n'utilise qu'un seul bloc de données. 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 .. !

[3] en contraste avec les opérateurs logiques `&&` 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); // Je lis le bloc 27
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 combien de fichiers/sous-répertoires il y a dans un répertoire, 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 chercher une sous-chaîne de caractère dans un string : `strstr`

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.

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

```
int bd_countusedblocks();
```

Cette fonction retourne le nombre de bloc de données occupés 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 ne faites rien et retournez -2. Pour les permissions rxw, simplement

les mettre toutes à 0. Elles pourront être modifiées subséquemment par la fonction `bd_chmod`. 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.
Vous devez vérifier que :

- Le fichier `pFilename` existe. Dans le cas contraire, cette fonction devra afficher à l'écran `"Le fichier` ***`pFilename`*** `est inexistant!"`, où `pFilename` est le nom du fichier demandé. La fonction devra retourner -1.
- Le fichier `pFilename` est un répertoire. Dans le cas contraire, cette fonction devra afficher à l'écran `"Le fichier` ***`pFilename`*** `est un repertoire!"`, où `pFilename` est le nom demandé. La fonction devra retourner -2.
- L'`offset` est plus petit que la taille du fichier. Dans le cas contraire, cette fonction devra simplement retourner 0, car vous ne pouvez pas lire aucun caractère.

La principale difficulté ici est de trouver dans quel(s) bloc(s) sur le disque se trouvent les données demandées, et à quel(s) endroit(s) dans ce(s) bloc(s) faire la lecture. N'oubliez pas que vous devrez peut-être lire plus d'un bloc de données, par exemple si `numbytes` est plus grand que la taille `BLOCK_SIZE` d'un bloc sur le disque ou que la lecture chevauche deux blocs de données.

Petit truc de langage C : pour trouver dans quel `Block` n dans `iNodeEntry` commence la lecture :

```
n = offset/BLOCK_SIZE;  // Division entière, pas de retenue.
```

et l'offset dans ce bloc sera

```
offset_block = offset%BLOCK_SIZE; // modulo
```

```
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 afficher à l'écran `"Le fichier` ***`pFilename`*** `est inexistant!"`, où `pFilename` est le nom du fichier demandé. La fonction devra retourner -1.
- Le fichier `pFilename` n'est pas un répertoire. Dans le cas contraire, cette fonction devra afficher à l'écran `"Le fichier` ***`pFilename`*** `est un repertoire!"`, où `pFilename` est le nom demandé. La fonction devra retourner -2.
- L'`offset` doit être plus petit ou égal à la taille du fichier. Dans le cas contraire, cette fonction devra afficher à l'écran `"L'offset demande est trop grand!"`. La fonction devra retourner -3. Par contre, si l'`offset` est plus grand ou égal à la taille maximale supportée par ce mini-UFS, 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 blocs d'adressage direct `N_BLOCK_PER_INODE`. Dans le cas contraire, cette fonction devra afficher à l'écran `"Le fichier` ***`pFilename`*** `deviendra trop gros!"`, où `pFilename` est le nom demandé. Vous devez quand même écrire le plus possible dans le fichier, jusqu'à atteindre la taille limite. La fonction retournera ce nombre d'octets écrit.

Ici aussi vous avez la difficulté de bien trouver dans quel(s) bloc(s) les opérations sont faites. De plus, vous devez allouer les blocs sur le disque quand le fichier croît. Par exemple, si le fichier fait 200 octets au départ (donc ne nécessite qu'un seul bloc de données), **et que l'on demande d'écrire 100 octets à partir de l'offset 190**, il vous faudra prendre un bloc libre sur le disque, le marquer comme utilisé, l'ajouter dans le tableau `Block` de l'*inode*, incrémenter la valeur `st_blocks`, etc. N'oubliez pas de modifier la taille du fichier `st_size`.

Méfiez-vous aussi du fait qu'un tableau en C commence par l'index 0, vs. le nombre de blocs détenus par un fichier `st_blocks`. Par exemple, si un fichier n'a qu'un seul bloc de données, `st_blocks` =1 mais le numéro de ce bloc se situe à `Block[0]`.

```
int bd_chmod(const char *pFilename, UINT16 new_mode);
```

Cette fonction sert à changer les permissions rxw pour Group et User du fichier portant le nom `pFilename`. **Attention!** Ne faites pas `pInode->iNodeStat.st_mode = new_mode` car ceci effacerait aussi d'autres drapeaux (comme `G_IFREG`). Il vous faut plutôt uniquement recopier les bits de permissions contenus dans `new_mode` vers `st_mode`. Ignorez les autres bits dans new_mode. Retournez -1 si le fichier n'existe pas, sinon 0.

```
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 0. 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 « .. ».

```
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. En cas de succès, retournez 1.

`int bd_unlink(const char *pFilename);`
Cette fonction sert à retirer un fichier normal (`G_IFREG` 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* et en détruisant l'entrée dans le fichier de répertoire (contenu dans le `pFilename`). Si `st_nlink` tombe à zéro, vous devez effacer ce fichier en libérant les blocs de données associés à l'*i-node*, et libérer aussi l' *i-node*. Si après `bd_unlink` le nombre de liens 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 la structure du répertoire, si le fichier détruit n'est pas à la fin du tableau. Si `pFilename` n'existe pas retournez -1. S'il n'est pas un fichier régulier `G_IFREG`, retournez -2. Autrement, retourner 1 pour indiquer le succès.

`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 1 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 manipulez 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.

`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`) et recopier le contenu du 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.

`int bd_FormatDisk(void);`
La fonction de formatage vient effacer le disque virtuel et créer le système de fichier UFS. Cette fonction doit initialiser le bitmap des blocs libres dans le bloc prévu à cet effet. Initialiser

le bitmap des *i-nodes* libres dans le bloc 3. Marquez tous les blocs utilisés par le système de fichiers comme non libres. Créer les *i-nodes* dans les blocs prévus pour ça. Marquez tous les *i-nodes* comme libres (sauf 0 non libre), puis créer le répertoire racine / avec l'i-node 1. Il est à noter qu'il n'est pas nécessaire de sauvegarder le nom du répertoire racine, car il est unique.

```
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.

```
int bd_fct_perso(…);
```

Cette fonction est la vôtre. Elle doit avoir au moins un argument. Vous devez inventer une fonctionnalité qui est différente de celles demandées. Vous devez modifier le elseif de cette fonctionnalité dans main_ufs.c et changer ces 2 constantes dans UFS.h

```
#define FONCTION_PERSO "perso"
#define NB_ARG_FCT_PERSO 1
```

Vous devez ajouter votre ou vos arguments à la fonction suivante dans UFS.c et UFS.h

```
int bd_fct_perso();
```

Nous fournir 3 exemples de l'utilisation (valide et non valide) de votre fonction en les mettant à la fin du fichier `TestEtudiant.sh`

Conseils

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

- traduire un nom de fichier en i-node;
- saisir et relâcher des i-nodes;
- saisir et relâcher des blocs de données;

Structure de l'arbre de fichiers dans `DisqueVirtuel.dat`

Le disque simulé `DisqueVirtuel.dat` est préformaté et contient la structure suivante :

Répertoire /

```
 d              . size:      96 inode:  1 nlink: 5
 d             .. size:      96 inode:  1 nlink: 5
 d            doc size:      48 inode:  2 nlink: 3
 d            rep size:      32 inode:  3 nlink: 2
 d        Bonjour size:      48 inode:  4 nlink: 2
 -          b.txt size:      66 inode:  7 nlink: 1
```

Répertoire /doc

```
 d              . size:      48 inode:  2 nlink: 3
 d             .. size:      96 inode:  1 nlink: 5
 d            tmp size:      48 inode:  5 nlink: 3
```

Répertoire /doc/tmp

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

Répertoire /doc/tmp/subtmp

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

Répertoire /rep

```
 d              . size:      32 inode:  3 nlink: 2
 d             .. size:      96 inode:  1 nlink: 5
```

Répertoire /Bonjour

```
 d              . size:      48 inode:  4 nlink: 2
 d             .. size:      96 inode:  1 nlink: 5
 -    LesAmis.txt size:       0 inode:  9 nlink: 1
```

La structure du disque vous est donnée pour vous aider à déboguer 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.

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 utilisés sur le disque simulé, la commande sera

```
./ufs blockused
```

Pour créer un fichier, la commande sera

```
./ufs create nom_de_fichier
```

Pour modifier ses droits, faites

```
./ufs chmod droits 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
```

Voir le fichier .sh pour le reste des commandes

Comme ce programme `ufs` quitte après l'exécution de chaque commande, cela va vous permettre de vous assurer que le disque simulé `DisqueVirtuel.dat` est consistant après chaque commande. Si vous avez oublié de sauvegarder des blocs (données 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 fourni. Pour vous aider à mieux comprendre ce que vos fonctions doivent accomplir est incluse la sortie d'écran de la solution dans le fichier `MesResultats.txt`. J'ai aussi laissé l'exécutable dans la soumission, sous le nom `ufs_complet` .

Pour la correction, nous allons faire tourner notre propre script. Le score dépendra des résultats de ce test.

Fournir 3 commandes différentes d'utilisation de votre fonction personnalisée.

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[4].

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 `DisqueVirtuel.dat`, 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
```

[4] 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.

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 contenus dans `DisqueVirtuel.dat` :

- `mv`
- `ls`
- `ls –la`
- `ls -i`
- `mkdir`
- `touch`
- `ln`
- `rmdir`
- `cd`
- `cp`
- `gedit`
- `more`

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

Vous devriez constater aussi que :

- l'utilisation de la touche [tab] pour l'autocomplétion fonctionne (et provoque de multiples appels à votre libraire);
- la traduction des chemins relatifs est 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`.