

Travail Pratique 3

Systèmes de fichiers

À remettre, par le portail du cours le 22 décembre 2023 avant 23h59

Ce travail doit être fait en équipe de 2 ou 3.

1 Objectif

Ce travail pratique a pour objectif d'implémenter un mini-système de fichiers en s'inspirant du système UFS (*Unix File System*) que nous avons vu dans le cours surtout qu'il est la base de nombreux systèmes de fichiers modernes (ext2, ext3 et ext4, par exemple). Dans ce travail vous devez vous familiariser entre autres avec les opérations nécessaires pour gérer les fichiers et les impacts sur les métadonnées, la traduction entre un nom de fichier et un numéro d'i-node et les patrons d'accès au disque résultants d'opérations simples comme ls, mkdir, rm, etc.

2 Travail à faire

Vous devez utiliser obligatoirement le fichier "**GabaritRapportTP3.docx**" pour le rapport, afin d'uniformiser la présentation pour le correcteur. Écrivez les noms et matricules au début du document. Lors de votre remise sur le portail du cours, incluez ce rapport ainsi que les codes sources, zippés dans un seul fichier (voir section 3). 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 %. Notez également que 10 points sont donnés pour le respect et la qualité des biens livrables. Nous vous demandons d'implémenter le mini-système de fichiers en utilisant le langage C++. Nous vous conseillons d'utiliser les normes C++11. Assurez-vous que votre code est suffisamment commenté pour nous aider dans la compréhension.

2.1 Structure du système de fichiers

L'espace disque est représenté par un vecteur de la STL (i.e. `m_blockDisque`) se trouvant dans la partie privée de la classe **DisqueVirtuel**. Ce vecteur contient au maximum 128 blocs qui sont des objets de la classe **Block**. Chaque bloc va contenir différents types de données, car il peut représenter soit un bitmap de blocs libres soit un bitmap d'inodes libres soit un i-node soit une liste d'entrées d'un répertoire. 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.

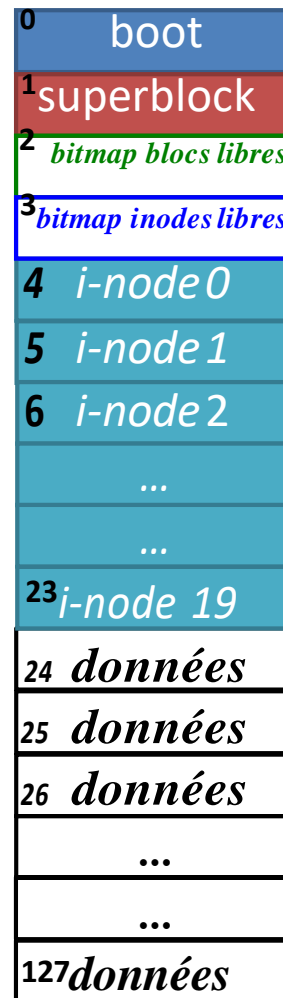
bloc 3 : Bitmap des i-nodes libres.

bloc 4 : i-node 0 → laisser libre (Sur Unix, contient la liste des blocs défectueux).

bloc 5 : i-node 1 → répertoire racine /

blocs 6-23 : espace pour les autres i-nodes (pour fichiers ou répertoires).

blocs 24-127 : 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. Aussi, sur un système UFS, on peut placer plus d'un i-node dans un bloc du disque. Pour simplifier le TP, nous allons utiliser un bloc par i-node.

Nous utilisons les flags suivants pour identifier le type de données stockées dans le bloc :

```
#define S_IFBL    0010 // indique que le bloc contient le bitmap des blocs libres
#define S_IFIL    0020 // indique que le bloc contient le bitmap des inodes libres
#define S_IFIN    0030 // indique que le bloc contient des métadonnées (donc un i-node)
#define S_IFDE    0040 // indique que le bloc contient la liste des dirEntry (informations d'un répertoire)
```

D'ailleurs, un bloc contient les membres privés suivants :

```
size_t m_type_donnees;           // peut être S_IFIL, S_IFBL, S_IFIN ou S_IFDE
std::vector<bool> m_bitmap;       // pour stocker la liste des blocs libres ou les inodes libres
iNode * m_inode;                 // pour stocker les métadonnées d'un fichier ou un répertoire
std::vector<dirEntry*> m_dirEntry; // pour stocker la liste des dirEntry (les informations d'un répertoire)
friend class DisqueVirtuel;       // La classe DisqueVirtuel est amie pour avoir accès à la partie privée !
```

De ce fait, nous pouvons identifier la nature des données stockées dans le bloc en utilisant la variable *m_type_donnees*. Le vecteur de booléens *m_bitmap* permet de stocker la liste des blocs libres ou les inodes libres. *m_inode* est utilisé pour stocker d'une façon dynamique (sur le tas) un objet *iNode* représentant les métadonnées d'un fichier ou d'un répertoire (voir section Structure d'un i-node). Par contre, *m_dirEntry* est utilisé pour stocker la liste des pointeurs vers des *dirEntry* représentant les informations d'un répertoire (voir section d'un répertoire).

Pour simuler le bitmap des blocs libres, simplement utilisez le bloc *FREE_BLOCK_BITMAP* comme un vecteur de booléen où chaque élément indique si un bloc est libre (*true*) ou pas (*false*). Faites de même pour le bitmap des *i-nodes* libres. Par exemple, pour saisir un bloc, vous pouvez utiliser le code suivant:

```
int positionFreeBlock = findFreeBlock(); m_  
    [FREE_BLOCK_BITMAP].m_bitmap[positionFreeBlock] = false;
```

Pour alléger votre programme, vous n'avez pas besoin de vérifier la validité de la chaîne de caractères représentant le nom d'un fichier ou le nom d'un répertoire. Simplement s'assurer que le fichier ou le répertoire indiqué est présent ou non. Les répertoires sont séparés par le caractère slash (/). En cas de doute, consultez les fichiers *Test1* et *Test2*, qui contiennent les commandes que nous allons tester avec votre programme.

2.2 Structure d'un *i-node*

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

```
struct iNode  
{  
    size_t st_ino;    // numero de l'i-node  
    size_t st_mode;    // S_IFREG ou S_IFDIR. Normalement contient aussi RWX  
    size_t st_nlink; // nombre de lien pointant vers l'i-node  
    size_t st_size;    // taille du fichier, en octets  
    size_t st_block; // numero du block (un seul bloc dans le TP pour simplifier)  
  
    iNode(size_t i, size_t m, size_t n, size_t s, size_t b) : st_ino(i),  
        st_mode(m), st_nlink(n), st_size(s), st_block(b) {}  
};
```

Le champ *st_block* contient le numéro du bloc utilisé pour emmagasiner les données d'un fichier. Dans le TP et comme les fichiers sont petits, ne vous préoccupez donc pas de gérer plus d'un bloc de données, afin de simplifier votre TP. Il n'y aura pas de redirection simple, double ou triple pour votre système de fichier.

Le champ `st_ino` contient le numéro de l'*i-node*. Dans le TP, le champ `st_mode` va servir uniquement à indiquer si c'est un fichier ou un répertoire. Le champ `st_nlink` sert à compter le nombre de lien vers cet *i-node*. Lorsque vous linker un fichier dans un répertoire pour la première fois, ce nombre sera donc de 1 pour le fichier. Chose importante, quand vous ajoutez le répertoire `/b/a` dans le répertoire `/b`, il faut incrémenter `st_nlink` pour le répertoire `/b`. De même, lorsque vous retirez le répertoire `/b/a` du répertoire `/b`, vous devez décrémenter `st_nlink` pour le répertoire `/b`. Le champ `st_size` indique la taille des données du fichier (pas la taille des métadonnées). D'ailleurs, un fichier vide aura `st_size=0`.

2.3 Structure d'un répertoire

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 `st_block` qui contient le numéro du bloc où les données sont stockées. Pour le TP et étant donné que sur la VM du cours un string a une taille de 24 octets et un `size_t` (*unsigned int*) a une taille de 4 octets, chaque `DirEntry` a une taille de 28 octets (Attention : Ces valeurs peuvent changer si vous exécutez votre code sur un autre système):

```
struct dirEntry
{
    size_t m_iNode;           // numero de l'i-node
    std::string m_filename;   // nom du fichier ou du repertoire

    dirEntry(size_t i, std::string m) : m_iNode(i), m_filename(m) {}
};
```

Le champs `m_iNode` comprend 4 octets (`size_t`) et constitue le numéro de l'*i-node* d'un fichier ou d'un sous-répertoire. Le champ `m_filename` est une chaîne de caractères représentant le nom d'un fichier ou d'un sous-répertoire.

Par défaut, lorsque vous créez un répertoire, il devra contenir les deux répertoires suivants : « . » et « .. ». Le premier contient son propre *i-node*, et le deuxième l'*i-node* du répertoire parent (sauf pour le répertoire racine, qui sera lui-même son propre parent). Encore une fois, assurez-vous d'incrémenter les `st_nlink` pour ces répertoires « . » et « .. ».

Une façon rapide de lire l'entrée *n* d'un répertoire dont les données sont stockées par exemple dans le bloc 47 est la suivante :

```
std::cout << "Le nieme fichier du repertoire : inode "
           << m_blockDisque[47].m_dirEntry[n]->m_iNode
           << " avec nom " << m_blockDisque[47].m_dirEntry[n]->m_filename;
```

2.4 Manipulation de chaînes de caractères (string)

Pour savoir si deux strings sont parfaitement identiques, utilisez par exemple *compare* :

```
m_blockDisque[positionBlockData].m_dirEntry[0]
->m_filename.compare("/") == 0)
```

Le résultat sera *true* si *m_filename* contenait un string identique à *"/"*.

Pour chercher la position d'un caractère, utilisez par exemple *find_last_of()*. Pour retourner une sous-chaîne, utilisez *substr()*. Au besoin, consultez ce [lien](#).

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 :

UFS: Saisie bloc 25

UFS: 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 :

UFS: Saisie i-node 5

UFS: Relache i-node 5

où 25 et 5 sont respectivement le numéro du bloc et de l'i-node en question. Ces messages vous aideront à mieux analyser votre programme, et nous aiderons à corriger votre solution.

2.5 Méthodes principales à implémenter

Les méthodes principales que vous devez implémenter sont celles de la classe *DisqueVirtuel* :

```
int bd_FormatDisk();
```

La méthode de formatage vient effacer le disque virtuel et créer le système de fichier UFS. Cette méthode doit initialiser le bitmap des blocs libres dans le bloc 2. Initialiser le bitmap des *i-nodes* libres dans le bloc 3. Marquez tous les blocs de 0 à 23 comme non-libres, puisqu'ils seront utilisés par le système de fichier. Créer les *i-nodes* dans les blocs de 4 à 23 (1 par bloc). Marquez tous les *i-nodes* de 1 à 19 comme libres (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. Cette méthode retourne soit 1 pour succès soit 0 pour échec.

```
int bd_create(const std::string& p_FileName);
```

Cette méthode vient créer un fichier vide (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 allez créer le fichier *a.txt* dans le répertoire */doc/tmp*. Assurez-vous que ce répertoire existe, et que l'*i-node* correspondant au fichier est marqué comme fichier (*st_mode* est à *S_IFREG*). Assurez-vous aussi que ce fichier n'existe pas déjà. Cette méthode retourne soit 1 pour succès soit 0 pour échec.

```
int bd_mkdir(const std::string& p_DirName);
```

Cette méthode créer le répertoire *pDirName*. Si le chemin d'accès à *pDirName* est inexistant, ne faites rien et retournez 0, par exemple si on demande de créer le répertoire */doc/tmp/test* et que le répertoire */doc/tmp* n'existe pas. Assurez-vous que l'*i-node* correspondant au répertoire est marqué comme répertoire (*st_mode* est à *S_IFDIR*). Si le répertoire *pDirName* existe déjà, retournez avec 0. Assurez-vous aussi que le répertoire contiennent les deux répertoires suivants : « . » et « .. ». N'oubliez-pas d'incrémenter *st_nlink* pour le répertoire parent « .. ». Cette méthode retourne soit 1 pour succès soit 0 pour échec.

```
std::string bd_ls(const std::string& p_DirLocation);
```

Cette méthode liste tous les fichiers et répertoires présents dans le répertoire *pDirLocation*. La sortie à l'écran devra être similaire à ceci :

```
===== Commande ls / =====
/
d          . Size:  112 inode:      1 nlink:      3
d          .. Size:  112 inode:      1 nlink:      3
d          doc Size:   56 inode:      2 nlink:      2
-          a.txt Size:   0 inode:      3 nlink:      1
```

Le premier champ indique un répertoire (d) ou un fichier (-). Le deuxième champ est le nom du fichier ou du sous-répertoire. Ceci est suivi de la taille (en octets), le numéro d'*i-node* et le nombre de liens. La taille du répertoire racine est 112 octets, car on utilise 4 entrées *dirEntry* (4 * 28 pour les répertoires ., .., doc et le fichier a.txt). La taille du répertoire doc est 56 octets, car on utilise 2 entrées *dirEntry* (2 * 28 pour les répertoires . et ..). La taille du fichier a.txt est 0, car il est vide. Le nombre de liens (*nlink*) pour l'*i-node* 1 est 3, car elle est référencée deux fois dans le répertoire racine et une fois dans le répertoire doc (répertoire parent). Le nombre de liens pour l'*i-node* 2 est 2, car elle est référence une fois dans le répertoire racine et une fois dans le répertoire doc (répertoire courant). Le nombre de liens pour l'*i-node* 3 est 1, car elle référencée seulement une seule fois dans le répertoire racine concernant le fichier a.txt. Cette méthode retourne un string contenant le formatage du contenu à afficher.

```
int bd_rm(const std::string& p_Filename);
```

Cette méthode provoque la déletion d'un fichier ou d'un répertoire. Attention! Dans UFS, la déletion se fait en réduisant le nombre de lien (*st_nlink*) et en détruisant l'entrée dans un fichier de répertoire. Si *st_nlink* tombe à zéro, vous devez libérer les blocs de données associés à l'*i-node*, et libérer aussi ce dernier. Si après *bd_rm* le nombre de lien n'est pas zéro, vous ne devez pas libérer l'*i-node*, puisqu'il est utilisé ailleurs. Si le string *pFilename* correspond à un répertoire, vous ne pouvez le détruire que si il est vide, i.e. ne contient pas d'autres répertoires autres que « . » et « .. ». Si le répertoire n'est pas vide, ne faites rien et retournez 0. N'oubliez-pas de décrémenter *st_nlink* pour le répertoire parent « .. », si un répertoire est retiré. Cette méthode retourne soit 1 pour succès soit 0 pour échec.

Pour toutes ces méthodes, prenez pour acquis que le chemin d'accès du fichier est bien formé. Vous devez vous assurer, par contre, que le fichier ou le répertoire est présent sur le disque. Le caractère « / » est utilisé pour séparer les répertoires. Sinon, vous devez bien sûr implémenter les constructeurs et destructeurs des classes DisqueVirtuel et Block (le constructeur prenant un paramètre permet d'initialiser *m_type_donnees*). Il est à noter que la classe Block est amie avec la classe DisqueVirtuel afin de simplifier l'implémentation des méthodes demandées. De ce fait, vous pouvez accéder directement à la partie privée de la classe Block sans avoir besoin d'ajouter des accesseurs (*get*) ou des modificateurs (*set*).

2.6 Méthodes utilitaires

Nous vous demandons d'ajouter des méthodes utilitaires vous permettant d'implémenter les méthodes principales mentionnées auparavant. Libre à vous de choisir le prototype de ces méthodes. En fait, vous devez implémenter une méthode permettant de trouver le premier i-node libre et une méthode permettant de trouver le premier bloc de données libre. Vous devez également concevoir une méthode créant un répertoire vide contenant les entrées « . » et « .. » et une méthode permettant de vérifier si un répertoire existe.

Nous vous conseillons d'ajouter d'autres méthodes utilitaires, mais elles ne sont pas obligatoires. Il s'agit par exemple de méthodes permettant d'incrémenter ou de décrémenter le n-link dans l'i-node, d'augmenter ou de diminuer la taille inscrite dans l'i-node, de relâcher ou de saisir un bloc ou un i-node dans le FREE_BLOCK_BITMAP ou dans le FREE_INODE_BITMAP, etc.

2.7 Tester votre programme

Après avoir implémenter les classes DisqueVirtuel et Block dans les fichiers *disqueVirtuel.cpp* et *block.cpp* respectivement et pour construire le programme dans le terminal, faites *make ufs*. le programme *ufs* sera créé. Pour nettoyer le répertoire, faites *make clean*. Le programme *ufs* va afficher un menu afin de soit exécuter

une commande soit exécuter plusieurs commandes se trouvant dans un fichier texte (pour simuler l'exécuter d'un script). À chaque appel de vos méthodes *bd_**, le système va exécuter la commande de manipulation de fichier passée en argument, avec les arguments supplémentaires requis. Le parsing est fait pour vous dans *main.cpp*, et va appeler la méthode *bd_** appropriée. Par exemple, pour formater le disque, vous tapez la commande *format*, ce qui appellera votre méthode *bd_FormatDisk()*. Pour créer un nouveau répertoire, faites *mkdir nom_de_repertoire*, ce qui appellera votre méthode *bd_mkdir()*. Pour effacer un fichier ou un répertoire (vide), faites *rm nom* ce qui appellera votre méthode *bd_rm()*. Pour faire la liste des fichiers dans un répertoire, faites *ls nom_de_repertoire* ce qui appellera votre méthode *bd_ls()*. Pour créer un fichier, faites *create nom_de_repertoire* ce qui appellera votre méthode *bd_create()*.

Pour tester votre programme, utilisez les fichiers *Test1* et *Test2* fourni avec le TP. Capturez la sortie d'écran en faisant un copier-coller du texte (pas une saisie d'écran graphique). Pour vous aider, la sortie de votre programme pour le script *Test1* devrait être la suivante :

```
===== Commande format =====
UFS: saisir i-node 1
UFS: saisir bloc 24

===== Commande ls / =====
/
d          . Size: 56 inode: 1 nlink: 2
d          .. Size: 56 inode: 1 nlink: 2

===== Commande mkdir /doc =====
UFS: saisir i-node 2
UFS: saisir bloc 25

===== Commande ls / =====
/
d          . Size: 84 inode: 1 nlink: 3
d          .. Size: 84 inode: 1 nlink: 3
d          doc Size: 56 inode: 2 nlink: 2

===== Commande mkdir /tmp =====
UFS: saisir i-node 3
UFS: saisir bloc 26

===== Commande ls / =====
/
d          . Size: 112 inode: 1 nlink: 4
d          .. Size: 112 inode: 1 nlink: 4
d          doc Size: 56 inode: 2 nlink: 2
d          tmp Size: 56 inode: 3 nlink: 2

===== Commande mkdir /tmp/lib =====
UFS: saisir i-node 4
UFS: saisir bloc 27

===== Commande ls / =====
/
d          . Size: 112 inode: 1 nlink: 4
d          .. Size: 112 inode: 1 nlink: 4
```



```

d          doc Size:    56 inode:    2 nlink:    2
d          tmp Size:    84 inode:    3 nlink:    3

===== Commande ls /tmp =====
/tmp
d          . Size:      84 inode:    3 nlink:    3
d          .. Size:    112 inode:    1 nlink:    4
d          lib Size:    56 inode:    4 nlink:    2

===== Commande mkdir /tmp/lib/deep =====
UFS: saisir i-node 5
UFS: saisir bloc    28

===== Commande ls /tmp/lib/deep =====
/tmp/lib/deep
d          . Size:      56 inode:    5 nlink:    2
d          .. Size:    84 inode:    4 nlink:    3

===== Commande rm /tmp/lib/deep =====
UFS: Relache i-node 5
UFS: Relache bloc    28

===== Commande rm /tmp/lib =====
UFS: Relache i-node 4
UFS: Relache bloc    27

===== Commande rm /tmp =====
UFS: Relache i-node 3
UFS: Relache bloc    26

===== Commande ls / =====
/
d          . Size:      84 inode:    1 nlink:    3
d          .. Size:    84 inode:    1 nlink:    3
d          doc Size:    56 inode:    2 nlink:    2

===== Commande create /a.txt =====
UFS: saisir i-node 3

===== Commande ls / =====
/
d          . Size:    112 inode:    1 nlink:    3
d          .. Size:    112 inode:    1 nlink:    3
d          doc Size:    56 inode:    2 nlink:    2
-          a.txt Size:    0 inode:    3 nlink:    1

===== Commande rm /a.txt =====
UFS: Relache i-node 3

===== Commande ls / =====
/
d          . Size:      84 inode:    1 nlink:    3
d          .. Size:    84 inode:    1 nlink:    3
d          doc Size:    56 inode:    2 nlink:    2

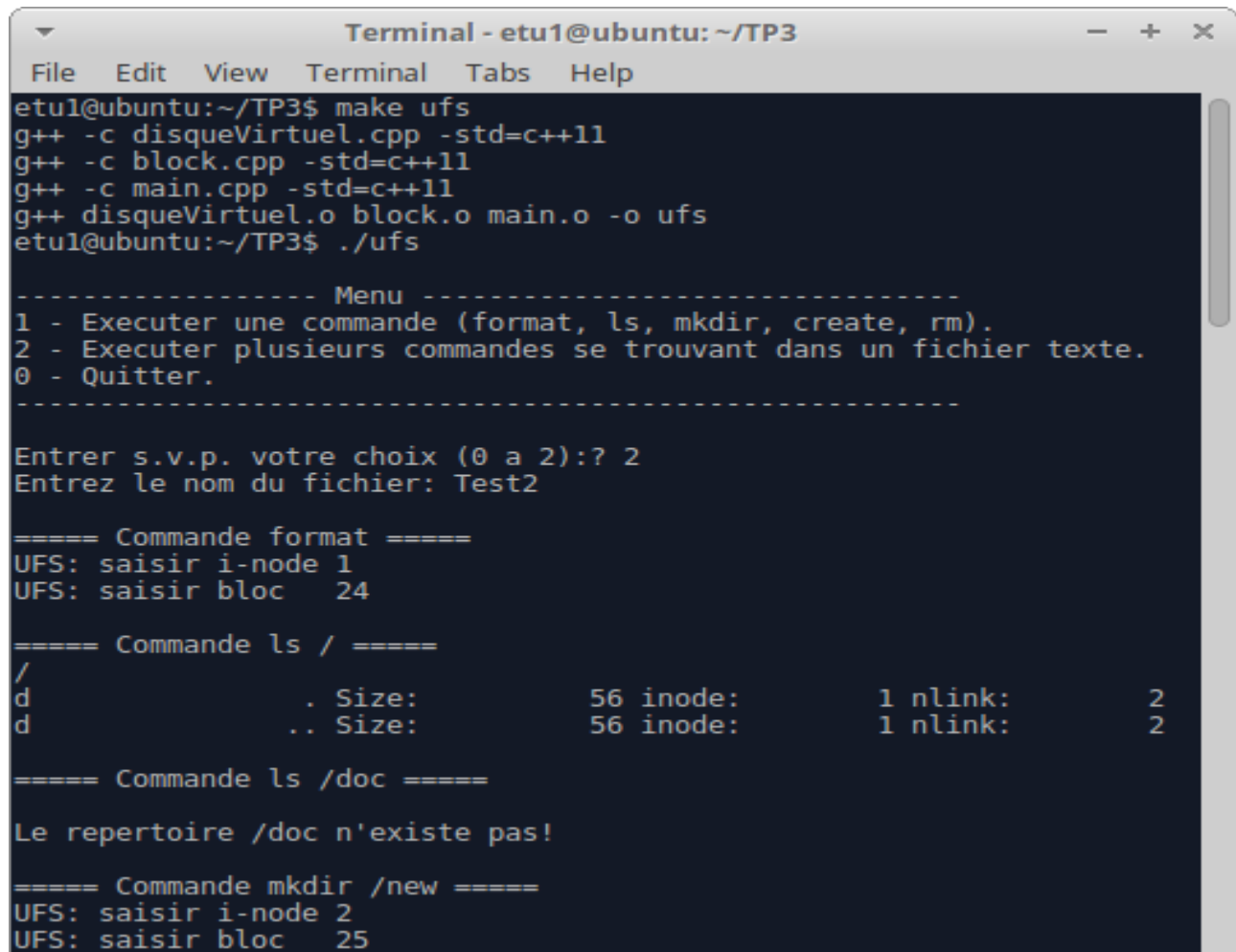
===== Commande format =====
UFS: saisir i-node 1
UFS: saisir bloc    24

===== Commande ls / =====
/

```

```
d          . Size:      56 inode:      1 nlink:      2
d          .. Size:     56 inode:      1 nlink:      2
```

Testez également vos méthodes à l'aide du scénario fourni dans le script *Test2*. Il faut bien sûr mettre la sortie d'écran (en texte) dans votre rapport. Commentez les résultats des tests, en un ou deux paragraphes. Si votre programme plante ou subit des échecs lors du déroulement, veuillez indiquer les causes probables.



```
Terminal - etu1@ubuntu: ~/TP3
File Edit View Terminal Tabs Help
etu1@ubuntu:~/TP3$ make ufs
g++ -c disqueVirtuel.cpp -std=c++11
g++ -c block.cpp -std=c++11
g++ -c main.cpp -std=c++11
g++ disqueVirtuel.o block.o main.o -o ufs
etu1@ubuntu:~/TP3$ ./ufs

----- Menu -----
1 - Executer une commande (format, ls, mkdir, create, rm).
2 - Executer plusieurs commandes se trouvant dans un fichier texte.
0 - Quitter.
-----

Entrez s.v.p. votre choix (0 a 2):? 2
Entrez le nom du fichier: Test2

===== Commande format =====
UFS: saisir i-node 1
UFS: saisir bloc 24

===== Commande ls / =====
/
d          . Size:      56 inode:      1 nlink:      2
d          .. Size:     56 inode:      1 nlink:      2

===== Commande ls /doc =====
Le repertoire /doc n'existe pas!

===== Commande mkdir /new =====
UFS: saisir i-node 2
UFS: saisir bloc 25
```

3 Ce que vous devez rendre

Vous devez rendre un fichier **.zip** comportant **uniquement** les fichiers suivants :

- Le fichier **RapportTP3.pdf** contenant votre rapport
- Les fichiers **disqueVirtuel.cpp** et **block.cpp** contenant votre code. Assurez-vous de commenter votre code raisonnablement. Un code mal documenté pourra être pénalisé.
- Les fichiers fournis en ajoutant bien sûr les prototypes des méthodes utilitaires: **main.cpp**, **disqueVirtuel.h**, **block.h**, **Makefile**, **Test1** et **Test2**.
- Le fichier **NoteTp3.xls** contenant le barème (ajoutez simplement les noms et matricules afin de faciliter le travail du correcteur).

Vous ne devez en aucun cas ajouter un autre fichier ou répertoire. N'incluez aucun exécutable, ni aucun fichier généré par votre environnement de développement. De plus, il est de votre responsabilité de vérifier après la remise que vous nous avez envoyé les bons fichiers (**non vides et non corrompus**), sinon vous pouvez avoir un zéro.

Tolérance zéro vis-à-vis des erreurs de compilation :

Vos programmes doivent contenir zéro erreur de syntaxe. Un programme qui ne compile pas, peu importe la raison, sera fortement pénalisé.

Plagiat : Tel que décrit dans le plan de cours, le plagiat est interdit. Une politique stricte de tolérance zéro est appliquée en tout temps et sous toutes circonstances.

Attention! Tout travail remis en retard se verra pénalisé. Le retard débute dès la limite de remise dépassée (dès la première minute). Un retard excédant une journée provoquera le rejet du travail pour la correction et la note de 0 pour ce travail.

Bon travail !