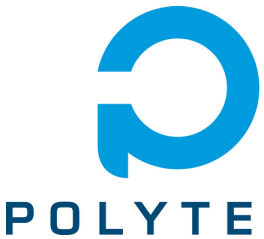

Polytech' Grenoble

RICM 4ème année

NachOS
Etape 5: Gestion des fichiers

Étudiants:
Elizabeth Paz
Salem Harrache

Enseignant:
Vania Marangozova

Avril 2012

1 Test de l'étape 5

Pour cette étape, nous avons mis au point un shell pour tester notre implémentation des fichiers. Ainsi on dispose de quelques commandes basiques pour copier, supprimer ou encore lister le contenu d'un dossier. Pour lancer le shell, il faut lancer le script de lancement automatique *runshell.sh* qui se trouve dans le dossier *code*.

Si le shell est pratique pour tester de façon intéractive, il est intéressant de noter qu'il comporte lui-même une commande pour lancer des tests automatiquement. C'est la commande test. Elle commence par formater le disque dur nachos, puis effectue un ensemble d'opérations dont la copie d'un gros fichier (100ko) de linux vers nachos.

```
$ ./runshell.sh
Compilation en cours...
Lancement du Shell
--------------------
./build-origin/nachos-filesys -shell
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times

Commandes disponibles :
exit
ls [<path>]
cd [<path>]
touch <filename>
print <filename>
cp <src> <dest>
rm <path>
mkdir <dirname>
mkdir -p <path>
pwd
test  : lance les tests
format
-------------------------

nachos / $ test
nachos / $ ls
.
..
nachos / $ mkdir foo
nachos / $ ls foo
.
..
nachos / $ mkdir foo/bar
nachos / $ cd foo/bar
nachos /foo/bar/ $ touch zero
nachos /foo/bar/ $ print zero
nachos /foo/bar/ $ cp ./filesys/test/small small
nachos /foo/bar/ $ print small
This is the spring of our discontent.
nachos /foo/bar/ $ cp ./filesys/test/big big
nachos /foo/bar/ $ ls
.
..
zero
small
big
nachos /foo/bar/ $ ls big
Name : big  Length : 106787 Bytes
nachos /foo/bar/ $ cd ..
nachos /foo/ $ rm bar
rm : le dossier n est pas vide
nachos /foo/ $ rm bar/zero
nachos /foo/ $ rm bar/small
nachos /foo/ $ rm bar/big
nachos /foo/ $ ls bar
.
..
nachos /foo/ $ rm bar
nachos /foo/ $ rm .
nachos / $ rm .
rm: impossible supprimer le r pertoire   /
nachos / $ mkdir -p this/is/a/simple/test/by/liz/and/salem
nachos / $ cd this/is/../is/../is/./../a/simple/test/by/liz/and/salem
```

```
nachos /this/is/a/simple/test/by/liz/and/salem/ $ pwd
/this/is/a/simple/test/by/liz/and/salem/
nachos /this/is/a/simple/test/by/liz/and/salem/ $ cd
nachos / $
```

2 Augmentation de la taille maximale des fichiers

La limitation de la taille des fichiers est de **FileHeader**. En effet, un FileHeader dispose au maximum de $32 - 2$ secteurs d'entrée pour les données. Ce qui fait une taille maximale de 3.75 Kb pour un fichier. Pour l'augmenter nous introduisons un niveau de redirection. C'est-à-dire qu'au lieu de disposer d'un tableau de n secteurs, nous allons avoir un tableau de n tableaux de secteurs. On peut avoir 30 tableaux de redirections. Chaque tableau de redirection comporte 32 secteurs. Chaque secteur a une taille de 128bits.

Ce qui fait une taille max de : $32 * 30 * 128bits = 122880bits =$ **120 KB**

Cette modification est complètement indépendante de la suite, c'est pour cette raison qu'on s'y intéresse en premier. Il faut modifier **FileHeader** pour qu'au moment de l'allocation et désallocation, on utilise bien ce niveau d'indirection. De plus il faut modifier la fonction **ByteToSector(int offset)** pour faire la translation.

```
#define NumIndirect     ((SectorSize) / sizeof(int))

bool FileHeader::Allocate(BitMap *freeMap, int fileSize)
{
    numBytes = fileSize;
    numSectors  = divRoundUp(FileLength(), SectorSize);
    if (freeMap->NumClear() < numSectors)
        return FALSE;// not enough space

    int * indirectList;
    int allocatedSectors = 0;
    int i;
    int j;
    for (i = 0; i < (int) NumDirect && allocatedSectors < (int) numSectors; i++) {
        dataSectors[i] = freeMap->Find();
        indirectList = new int[NumIndirect];
        for (j=0; (j < (int) NumIndirect) && (allocatedSectors < numSectors); j++) {
            indirectList[j] = freeMap->Find();
            allocatedSectors++;
        }
        synchDisk->WriteSector(dataSectors[i], (char *)indirectList);
    }

    return TRUE;
}
```

De la même manière on applique ce changement à **Deallocate(BitMap *freeMap)**

```
void FileHeader::Deallocate(BitMap *freeMap)
{
    int * indirectList;
    int deallocatedSectors = 0;
    int i;
    int j;
    for (i = 0; i < (int) NumDirect && deallocatedSectors < (int) numSectors; i++) {
        ASSERT(freeMap->Test((int) dataSectors[i]));  // ought to be marked!

        indirectList = new int[NumIndirect];
        synchDisk->ReadSector(dataSectors[i], (char *)indirectList);

        for (j=0; (j < (int) NumIndirect) && (deallocatedSectors < numSectors); j++) {
            ASSERT(freeMap->Test((int) indirectList[j]));
            deallocatedSectors++;
        }
        freeMap->Clear((int) dataSectors[i]);
    }
}
```

Maintenant il ne reste plus qu'à calculer le bon secteur en appliquant une petite translation

```
int FileHeader::ByteToSector(int offset)
{
    int sector = offset / SectorSize;
```

```
    int numList = sector / NumIndirect;
    int posInList = sector % NumIndirect;

    int * indirectList = new int[NumIndirect];
    synchDisk->ReadSector(dataSectors[numList], (char *)indirectList);

    return(indirectList[posInList]);
}
```

Finalement la manipulation consiste à implémenter une matrice au niveau du **FileHeader**. La taille peut varier d'un fichier à un autre, l'allocation se fait selon le besoin en espace/secteur.

3 Implantation d'une hiérarchie de répertoires

3.1 Distinction entre fichiers et repertoires

Pour distinquer un dossier d'un fichier, il faut modifier la structure du **FileHeader**. On pourrait éventuellement ajouter un booléen, seulement le **FileHeader** ne pourra plus tenir sur 1 seul secteur. La méthode que nous employons est d'utiliser la partie négative de la variable **numBytes** : Ainsi un dossier aura un **numBytes** négatif. Ca ne change strictement rien, il faut simplement s'assurer que partout où la variable est utilisée, on renvoie la valeur abolue.

```
int FileHeader::FileLength()
{
    return abs(numBytes);
}
```

Ansi on peut ajouter une nouvelle fonction pour savoir si c'est l'entête d'un dossier dont il s'agit.

```
bool FileHeader::isDirectoryHeader()
{
    return (numBytes < 0);
}
```

3.2 Les dossiers "." et ".."

Pour l'instant un **Directory** comporte un tableau de **DirectoryEntry** (secteurs) non distingué. Il faut donc avant tout allouer systématiquement deux de ces entrées aux répertoires "." et "..".

Un dossier se construit de la façon suivante :

```
Directory::Directory(int size, int sector, int parentSector) {
    table = new DirectoryEntry[size];
    tableSize = size;
    for (int i = 2; i < tableSize; i++)
        table[i].inUse = false;
    makeDirHierarchy(sector, parentSector);
}
```

```
void Directory::makeDirHierarchy(int sector, int parentSector) {
    // Ajout les dossiers "." et ".."
    table[0].inUse = true;
    table[0].sector = sector;
    strcpy(table[0].name, ".");

    table[1].inUse = true;
    table[1].sector = parentSector;
    strcpy(table[1].name, "..");
}
```

A l'exception du dossier racine, qui a pour secteur parent le secteur racine lui-même (= 1).

3.3 Création d'un dossier

La création du dossier se fait dans le dossier en cours. Nous avons une méthode dans la classe **FileSystem** qui construit systématiquement le dossier courant à partir du fichier **directoryFile**

On prend soin de vérifier que le nom n'est pas déjà utilisé dans ce répertoire. Ensuite on essaye d'alouer un secteur pour le **FileHeader** du dossier. Le plus important dans cette manipulation, est de bien allouer la zone des données (avec **FileHeader:Allocate**) avec taille négative pour bien marquer ce fichier comme étant un dossier. Une fois ajouté dans les entrées du dossier courant, il ne reste plus qu'à écrire en mémoire persistante.

```
int FileSystem::MakeDir(char *name) {
    bool error = false;

    Directory *currentDir = this->CurrentDir();

    if(strlen(name) > FileNameMaxLen) {
        printf("mkdir: nom de fichier trop long\n");
        error = true;
    }

    if (!error && currentDir->Find(name) != -1) {
       printf("mkdir: impossible de cr er le r pertoire  %s%s : Le fichier existe\n", currentDir->↩
           getDirName(), name);
       error = true;
    }

    if (!error && currentDir->isFull()) {
        printf("mkdir: le dossier %s est plein\n", currentDir->getDirName());
        error = true;
    }

    BitMap *freeMap = new BitMap(NumSectors);
    int freeSector;
    if (!error) {
        freeMap->FetchFrom(freeMapFile);
        freeSector = freeMap->Find();
        if (freeSector == -1) {
            printf("mkdir: plus de secteurs libres\n");
            error = true;
        }
    }

    if (!error) {
        int currentSector = currentDir->getCurrentSector();
        // Ajout du dossier dans le dossier courant (le parent)
        currentDir->Add(name, freeSector);

        // Cr ation du header du nouveau dossier
        FileHeader *newDirHeader = new FileHeader;
        // - DirectoryFileSize pour detecter que c'est un dossier
        ASSERT(newDirHeader->Allocate(freeMap, -1 * DirectoryFileSize));
        // Ecriture en m moire
        newDirHeader->WriteBack(freeSector);

        // creation du dossier avec le bon parent
        Directory *newDir = new Directory(NumDirEntries, freeSector, currentSector);
        // Ouverture du DirFile pour savegarder le dossier
        OpenFile *newDirFile = new OpenFile(freeSector);
        // Ecriture en m moire
        newDir->WriteBack(newDirFile);
        // Sauvegarde du dossier courant
        currentDir->WriteBack(directoryFile);
        // Sauvegarde de la freemap
        freeMap->WriteBack(freeMapFile);

        delete freeMap;
        delete newDirHeader;
        delete newDirFile;
        delete newDir;
    }
    delete currentDir;
    return 0;
}
```

3.4 Supression d'un dossier

La suppression est presque équivalente à celle d'un dossier. On s'assure simplement dans le cas d'un dossier, qu'il est vide, qu'il n'est pas à la racine et qu'on ne se trouve pas dans le dossier qu'on veut supprimer (Si c'est le cas on se déplace dans le dossier parent)

```
bool FileSystem::Remove(char *name)
{
    bool error = false;
    Directory *directory = this->CurrentDir();
    int currentSector = directory->getCurrentSector()

    BitMap *freeMap;
    FileHeader *fileHdr;
    int sector;

    sector = directory->Find(name);
    if (sector == -1) {
       printf("rm: le fichier ou dossier  %s  n'existe pas\n", name);
       error = true;
    }

    fileHdr = new FileHeader;
    fileHdr->FetchFrom(sector);

    if (!error) {
        // Si le fichier a supprimer est un dossier
        if (fileHdr->isDirectoryHeader()) {
            OpenFile * removeDirFile = new OpenFile(sector);
            Directory * removeDir = new Directory(NumDirEntries);
            removeDir->FetchFrom(removeDirFile);
            if (!removeDir->isEmpty()) {
                delete removeDirFile;
                delete removeDir;
                printf("rm : le dossier n'est pas vide\n");
                error = true;
            } else if (removeDir->isRoot()) {
                delete removeDirFile;
                delete removeDir;
                printf("rm: impossible supprimer le r pertoire  /  \n");
                error = true;
            }
            // si je me retrouve dans le dossier actuellement je remonte au parent
            // rm .
            else if (sector == currentSector) {
                currentSector = removeDir->getParentSector();
                MoveToSector(currentSector);
                directory = this->CurrentDir();
                delete removeDirFile;
                delete removeDir;
            }
        }
    }
    if (!error) {
        freeMap = new BitMap(NumSectors);
        freeMap->FetchFrom(freeMapFile);

        // suppression
        fileHdr->Deallocate(freeMap);          // remove data blocks
        freeMap->Clear(sector);            // remove header block
        directory->Remove(sector);

        // sauvegarde en m moire persistante
        freeMap->WriteBack(freeMapFile);        // flush to disk
        directory->WriteBack(directoryFile);        // flush to disk
        delete freeMap;
    }
    delete fileHdr;
    delete directory;

    return TRUE;
}
```

3.5 Changement de dossier

Pour changer de dossier il suffit de charger le bon fichier d'entête dans directoryFile. Il s'agit là d'un simple changement de contexte. Tout d'abord on cherche le secteur à charger en faisant une recherche dans le dossier courant (*currentDir-¿Find(name)*). Puis ensuite, on ouvre le nouveau bon fichier associé à ce secteur (OpenFile) et on l'enregistre comme nouvel entête du dossier courant.

```
int FileSystem::MoveToDir(char *name) {
    Directory *currentDir = this->CurrentDir();
    int dirSector = currentDir->Find(name);
    if (dirSector == -1) {
```

```
        printf("Le dossier %s%s n'existe pas\n", CurrentDir()->getDirName(), name);
        delete currentDir;
        return -1;
    }
    int result = MoveToSector(dirSector);
    if (result == -1)
        printf("%s%s n'est pas un dossier\n", CurrentDir()->getDirName(), name);
    delete currentDir;
    return result;
}

int FileSystem::MoveToSector (int dirSector) {
    OpenFile * newDirectoryFile = new OpenFile(dirSector);
    if (!newDirectoryFile->isDirectoryFile()) {
        delete newDirectoryFile;
        return -1;
    }
    directoryFile = newDirectoryFile;
    return 0;
}
```

4 Implantation des noms de chemin

Au-delà du parseur de chemin, ce mécanisme se résume à un changement temporaire de contexte (dossier courant) avant d'effectuer une opération.

Par exemple :

```
mkdir /foo/bar
```

Implique de se déplacer au dossier */foo*, de créer le dossier |bar pour ensuite restaurer le dossier courant original.

Sur ce principe, nous avons mis en place une fonction qui permet de se déplacer itérativement à l'avant-dernier dossier (selon les cas).

```
int FileSystem::MoveToLastDir(char * name)
{
    if (strcmp(name, "/") == 0) {
        this->MoveToRoot(); // == MoveToSector(1)
        strcpy(name, "\0");
        return 0;
    }
    if (name[0] == '/') {
        this->MoveToRoot();
    }
    char *paths[MAX_PATH_DEPTH];
    int npath;
    int i;
    parse_path(name, paths, &npath);
    if (npath > 0) {
        for(i = 0; i < npath-1; i++) {
            // On ignore les deplacement dans "." pour "optimiser"
            if (strcmp(paths[i], ".") != 0) {
                if (this->MoveToDir(paths[i]) != 0) {
                    return -1;
                }
            }
        }
        strcpy(name, paths[i]);
    }
    return 0;
}
```

Il suffit maintenant d'utiliser cette fonction partout où on désire utiliser les chemins, par exemple pour **MakeDir**

```
int FileSystem::MakeDir(char *name) {
    bool error = false;
    int originalSector = this->CurrentDir()->getCurrentSector();
    if (this->MoveToLastDir(name) == -1)
        return -1;
    // si je fais mkdir / > name = "\0"
    if (strcmp(name, "\0") == 0) {
        printf("mkdir: impossible de cr er le r pertoire   /  : Le fichier existe\n");
        return -1;
```

```
    }
    [...]
    this->MoveToSector(originalSector);
    return 0;
}
```

Sur ce principe nous avons implémentés la commande **mkdir -p** pour créer des dossiers récursivement (si les dossiers parents n'existent pas ils sont créés)