OS MP4 Report

Team member:

> **110062204 呂宜嫻:** part 2-1實作，part3實作 & report **110062239 侯茹文:** part 1、part 2-1 report、part 2-2實作、bonusI II

Part I. Understanding NachOS file system

(1) How does the NachOS FS manage and find free block space? Where is this information stored on the raw disk (which sector)?

```
#define FreeMapSector 0

PersistentBitmap *freeMap = new PersistentBitmap(NumSectors); //init 開新東西
freeMapFile = new OpenFile(FreeMapSector); //分配建立好空間之後就可以打開file了

freeMap->Mark(FreeMapSector); //代表對應的bitmap位置已經被占了 //傳入的是要被調為1的位置
freeMap->Mark(DirectorySector); //位置在前面有訂(0 1) -> for header
```

> Nachos使用了bitmap來記錄free block (Bitmap中有一個名為map的指標用來存該陣列)，存放在FreeMapSector，也就是sector 0，且map利用0/1來表示該sector是否被使用(而在一開始FreeMapSector以及DirectorySector就會被設為使用中)。
>
> 而利用繼承"Bitmap"的"PersistentBitmap"做宣告，使其可以運用Bitmap中的一些操作項目。

```
void Bitmap::Mark(int which)
{
    ASSERT(which >= 0 && which < numBits); //確保which值在可使用範圍內

    map[which / BitsInWord] |= 1 << (which % BitsInWord); //將對應位置調整為1，代表已經被使用

    ASSERT(Test(which));
}

void Bitmap::Clear(int which) //刪掉的註記
{
    ASSERT(which >= 0 && which < numBits);

    map[which / BitsInWord] &= ~(1 << (which % BitsInWord));

    ASSERT(!Test(which));
}
```

> 像是使用mark()，就可以標示對應的sector已經被使用了，而clear()就代表該sector被remove或是取消使用。最初也先將FreeMapSector以及DirectorySector的對應先標為1，他們分別使用sector 0、1。

```
int Bitmap::FindAndSet()
{
    for (int i = 0; i < numBits; i++) //嘗試可否使用 可以就讓他變成可以使用
    {
        if (!Test(i)) //測試可否用
        {
            Mark(i);
            return i;
        }
    }
    return -1; //沒有可以用的空間了
}
```

> 而使用FindAndSet()就可以遍歷map來找到是否有空的sector可以使用，找到後會把它設為1(已被使用)，並回傳可以使用的sector number；若沒有可以使用的，便會回傳-1。

(2) What is the maximum disk size that can be handled by the current implementation? Explain why.

```
const int DiskSize = (MagicSize + (NumSectors * SectorSize));
```

```
const int SectorSize = 128;     // number of bytes per disk sector
const int SectorsPerTrack  = 32;    // number of sectors per disk track
const int NumTracks = 32;       // number of tracks per disk
const int NumSectors = (SectorsPerTrack * NumTracks); // total # of sectors per
disk
```

> 在disk.cc中有宣告DiskSize，再加上disk.h中有定義他們的大小，因此可以知道disk size = MagicSize + 32 * 32 * 128 = 128KB (+ 4 bytes MagicSize)。

(3) How does the NachOS FS manage the directory data structure? Where is this information stored on the raw disk (which sector)?

```
#define DirectorySector 1

directoryFile = new OpenFile(DirectorySector);

//需要時:
directory->FetchFrom(directoryFile); //讀取目前的directory 讓新創建的direcory與現在的同步
```

```
//table的紀錄
Directory::Directory(int size)
{
    table = new DirectoryEntry[size]; //初始化一個table

    // MP4 mod tag
    memset(table, 0, sizeof(DirectoryEntry) * size); // dummy operation to keep
valgrind happy

    tableSize = size;
    for (int i = 0; i < tableSize; i++)
        table[i].inUse = FALSE; //將空間預設為未使用
}
```

> NachOS利用directory.h中的table(也是利用指標)來存是否被使用、file name以及其儲存位置等，並且將其header(dirHdr)存在sector 1中。

```
bool Directory::Add(char *name, int newSector) //將檔案放入directory
{
    if (FindIndex(name) != -1) //已經存在在檔案中了
        return FALSE;

    for (int i = 0; i < tableSize; i++)
        if (!table[i].inUse) //位置沒在用
        {
            table[i].inUse = TRUE; //放入!
            strncpy(table[i].name, name, FileNameMaxLen);
            table[i].sector = newSector;
            return TRUE;
        }
    return FALSE; // no space.  Fix when we have extensible files.
}

bool Directory::Remove(char *name)
{
    int i = FindIndex(name);

    if (i == -1)
        return FALSE; // name not in directory
    table[i].inUse = FALSE;
    return TRUE;
}
```

> 並且會透過Add()、Remove()來改寫table中的對象，描述是否正在被使用，具體是透過更改"inUse"的值，若是1就是正在被使用，反之就是沒被使用。

```
void Directory::FetchFrom(OpenFile *file) //讀取file(now directory)
{
```

```
    (void)file->ReadAt((char *)table, tableSize * sizeof(DirectoryEntry), 0);
}

void Directory::WriteBack(OpenFile *file) //寫入disk
{
    (void)file->WriteAt((char *)table, tableSize * sizeof(DirectoryEntry), 0);
}
```

> 並且透過FetchFrom()來read directory，然後利用WriteBack()來寫入。

```
int Directory::Find(char *name) //找到空的sector
{
    int i = FindIndex(name);

    if (i != -1)
        return table[i].sector;
    return -1;
}

int Directory::FindIndex(char *name)
{
    for (int i = 0; i < tableSize; i++) //找是否有這個檔名的檔案 有就回傳位置 無就回傳-1
        if (table[i].inUse && !strncmp(table[i].name, name, FileNameMaxLen))
            return i;
    return -1; // name not in directory
}
```

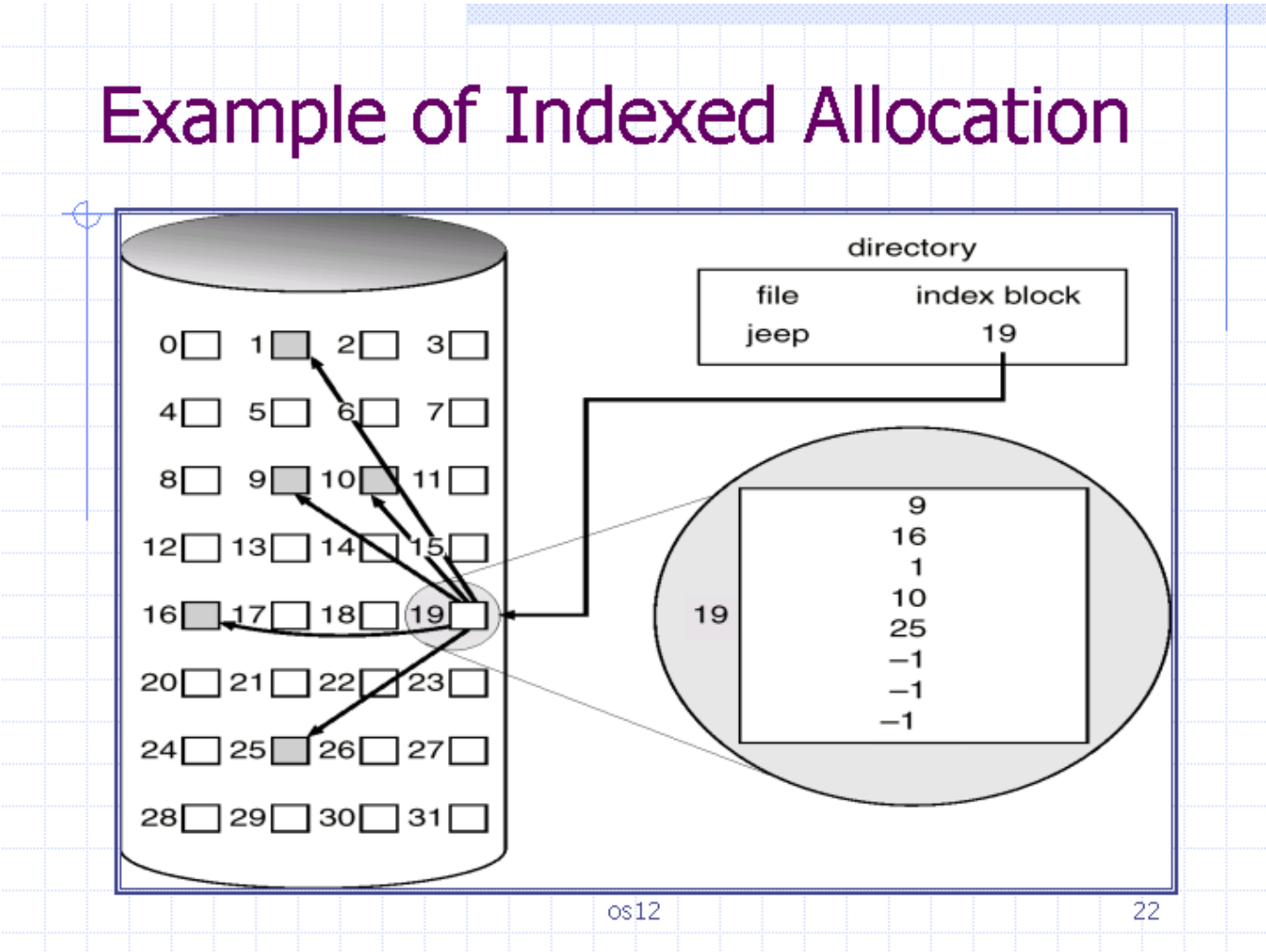
> 而使用Find([檔案名稱])可以進行找尋，具體會從directory.cc中的Find()進入FindIndex()進行遍歷，並比較檔案名稱是否有一樣的，有就會回傳其位置，沒有會回傳-1。

(4) What information is stored in an inode? Use a figure to illustrate the disk allocation scheme of the current implementation.

> 透過filehdr.h，我們可以知道inode中存放的如下:

```
int numBytes;       // Number of bytes in the file
int numSectors; // Number of data sectors in the file
int dataSectors[NumDirect]; // Disk sector numbers for each data
                // block in the file
```

可以了解到相關資訊，如存放著關於此file的Byte數量、sector數量以及每個block對應的sector number。而這樣allocate方法如下圖:



(5) What is the maximum file size that can be handled by the current implementation? Explain why.

> 因為目前使用的是index allocation，其可以使用的block數量取決於一個block中可以儲存多少index。

```
#define NumDirect ((SectorSize - 2 * sizeof(int)) / sizeof(int))
#define MaxFileSize (NumDirect * SectorSize)
```

```
const int SectorSize = 128;     // number of bytes per disk sector
const int SectorsPerTrack  = 32;    // number of sectors per disk track
const int NumTracks = 32;       // number of tracks per disk
const int NumSectors = (SectorsPerTrack * NumTracks); // total # of sectors per
disk
```

> 根據filehdr.h中的定義，可以知道現在可以使用的block為((SectorSize - 2 * sizeof(int)) / sizeof(int))，又總大小為(NumDirect * SectorSize)。我們可以算出((128 - 2 * 4) / 4) * 128 = 3840 byte，約為4KB。

Part II. Modify the file system code to support file I/O system calls and larger file size

(1) Combine your MP1 file system call interface with NachOS FS to implement five system calls:

exception.cc

```
case SC_Create:
    val = kernel->machine->ReadRegister(4);
    {
    char *filename = &(kernel->machine->mainMemory[val]);
    int size = kernel->machine->ReadRegister(5);
    //cout << filename << endl;
    status = SysCreate(filename, size);
    kernel->machine->WriteRegister(2, (int)status);
    }
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine-
>ReadRegister(PCReg));
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) +
4);
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)
+ 4);
    return;
    ASSERTNOTREACHED();
    break;
```

> 這部分與MP1相同，在exception.cc中處理相應的expection，並呼叫相應的函數。(這裡僅以Create()舉例)

ksyscall.h

```
int SysCreate(char *filename, int size)
{
    // return value
    // 1: success
    // 0: failed
    return kernel->fileSystem->Create(filename, size);
}
```

> 在ksyscall.h中新增對應函數，並call filesystem中的對應函數來處理，然後新放入了size。

(2) Enhance the FS to let it support up to 32KB file size

filehdr.h

```
//MP4-2
FileHeader* nextFileHeader; //指向下一個header //後來沒甚麼用到
int nextFileHeaderSector; //指向下一個header存的位置
```

> 在這裡新增了兩個變數，一個用來記錄next file的header (因為我們使用的是linked index)，一個指向儲存他的sector。

filehdr.cc

```
FileHeader::FileHeader()
{
    //MP4-2
    //nextFileHeader = NULL;
    nextFileHeaderSector = -1;

    numBytes = -1;
    numSectors = -1;
    memset(dataSectors, -1, sizeof(dataSectors));
}
```

> 而這裡對於nextFileHeaderSector進行初始化，並且因為我們後來nextFileHeader使用完就會delete，並不會存東西在裡面，所以這邊就沒有初始化了。

```
bool FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
{
    //MP4-2
    // numBytes = fileSize; //需要檢查numbytes是否小於現在的最大值
    if(fileSize <= MaxFileSize) numBytes = fileSize;
    else numBytes = MaxFileSize;

    numSectors = divRoundUp(numBytes, SectorSize);
    if (freeMap->NumClear() < numSectors)
        return FALSE; // not enough space

    for (int i = 0; i < numSectors; i++)
    {
        dataSectors[i] = freeMap->FindAndSet();

        ASSERT(dataSectors[i] >= 0);
    }

    //MP4-2
    if(fileSize > MaxFileSize){
        nextFileHeaderSector = freeMap->FindAndSet();

        if(nextFileHeaderSector == -1) return false;

                //若是有空間就宣告nextFileHeader 並繼續allocate
                //然後再將nextFileHeader所指向的空間寫入nextFileHeaderSector
        FileHeader* nextFileHeader = new FileHeader;
        bool success = nextFileHeader->Allocate(freeMap, fileSize-MaxFileSize);
        nextFileHeader->WriteBack(nextFileHeaderSector);
        delete nextFileHeader;
        return success;
    }
    return TRUE;
}
```

> allocate的部分除了原本就要做的事情以外，在一開始要先確認被放入的numByte小於max size。
>
> 另外再最後還需要檢查file size是否超過max size，若是超過，就需要再繼續往下呼叫allocate來儲存剩餘的檔案直到存完。
>
> 並且在存完之後將nextFileHeader指向的空間寫入nextFileHeaderSector，也就是剛剛在disk中找到的空間。

```
void FileHeader::Deallocate(PersistentBitmap *freeMap)
{
    for (int i = 0; i < numSectors; i++)
    {
        ASSERT(freeMap->Test((int)dataSectors[i])); // ought to be marked!
        freeMap->Clear((int)dataSectors[i]);
    }
    //MP4-2
    if(nextFileHeaderSector != -1) {
        FileHeader* nextFileHeader = new FileHeader;
        nextFileHeader->FetchFrom(nextFileHeaderSector);
        nextFileHeader->Deallocate(freeMap);
        delete nextFileHeader; //最後再將不需使用的nextFileHeader刪除
    }
}
```

> 在deallocate的最後需要檢查是否有nextFileHeaderSector的存在，若是有，就使用FetchFrom獲取其位址(也就是nextFileHeader)，然後再繼續遞迴將其deallocate。

```
int FileHeader::ByteToSector(int offset)
{
    int sector = offset / SectorSize;
    if (sector >= NumDirect) {
        FileHeader* nextFileHeader = new FileHeader;
        nextFileHeader->FetchFrom(nextFileHeaderSector);
        int b2sec = nextFileHeader->ByteToSector(offset - MaxFileSize);
        delete nextFileHeader;
        return b2sec;
    }
    else return (dataSectors[sector]);
}
```

> 先算出會在哪一個sector，如果超過目前的NumDirect，就需要遞迴往next header找ByteToSector()，如果沒有，就可以直接return，和原本一樣。
>
> nextFileHeader的獲取方式也和上面一樣，先用FetchFrom抓取，使用後再delete掉。

```
int FileHeader::FileLength()
{
```

```
    if(nextFileHeaderSector == -1) return numBytes;
    else {
        FileHeader* nextFileHeader = new FileHeader;
        nextFileHeader->FetchFrom(nextFileHeaderSector);
        int length = numBytes + nextFileHeader->FileLength();
        delete nextFileHeader;
        return length;
    }
}
```

> 若是沒有nextFileHeader，就直接return現在的numBytes；而若是有，就需要加上next的長度，因此需要遞迴往下。

```
void FileHeader::Print()
{
    int i, j, k;
    char *data = new char[SectorSize];

    printf("FileHeader contents.  File size: %d.  File blocks:\n", numBytes);
    for (i = 0; i < numSectors; i++)
        printf("%d ", dataSectors[i]);
        printf("\n");
    // printf("\nFile contents:\n");
    // for (i = k = 0; i < numSectors; i++)
    // {
    //  kernel->synchDisk->ReadSector(dataSectors[i], data);
    //  for (j = 0; (j < SectorSize) && (k < numBytes); j++, k++)
    //  {
    //      if ('\040' <= data[j] && data[j] <= '\176') // isprint(data[j])
    //          printf("%c", data[j]);
    //      else
    //          printf("\\%x", (unsigned char)data[j]);
    //  }
    //  printf("\n");
    // }
    if(nextFileHeaderSector != -1) {
        FileHeader* nextFileHeader = new FileHeader;
        nextFileHeader->FetchFrom(nextFileHeaderSector);
        nextFileHeader->Print();
        delete nextFileHeader;
    } //MP4-2

    delete[] data;
}
```

> 這裡只加上了print完後確認是否有nextFileHeader，若是有就需要繼續往下Print()。
>
> 而在bonusII的部分因為不需要print出中間的很多東西，所以就把他註解掉了，如此較為方便觀察。

Part III. Modify the file system code to support the subdirectory

(1) Implement the subdirectory structure

directory.h

```
class DirectoryEntry
{
public:
    //...
    // MP4
    bool isDir;                    // directory(1) or file(0)
};
```

> 將每個 subdirectory 視為一個 file，作為 DirectoryEntry 時，以 isDir 判斷為 file (false) 或 directory (true)。

Find file/directory Directory::Find

```
int Directory::Find(char *name)
{
    /// find first
    char *findCur;
    char cpyName[256];
    strcpy(cpyName, name);
    if (name[0] == '/') findCur = strtok(cpyName+1, "/");    // if with '/' at the
front, ignore it
    else findCur = strtok(cpyName, "/");
    if (findCur == NULL) findCur = cpyName; // last in path
    int i = FindIndex(findCur);

    if (i != -1) {
        char findNxt[256];
        if (strlen(name) - (strlen(findCur) + 1) == 0) return table[i].sector;
// return if the current is the last

        // *(findNxt+strlen(findNxt)) = '/';
        // while (strtok(NULL, "/"));
        // findNxt = name + strlen(findCur) + 1;
        strcpy(findNxt, name + strlen(findCur) + 1);
        // DEBUG('f', "Find file " << name << " directory " << findCur);
        if (!table[i].isDir) return -1;
        Directory* subDir = new Directory(NumDirEntries);
        OpenFile* dirFile = new OpenFile(table[i].sector);
        subDir->FetchFrom(dirFile);
        int findSec = subDir->Find(findNxt);
        delete dirFile;
        delete subDir;
        return findSec;
    }
```

```
    return -1;
}
```

> 先將收到的路徑依 '/' 分段，取第一個為當前 directory 要尋找的對象 (findCur)。在目前 directory 尋找名字相符的對象，若有則會得到對象在 table 中的 index，否則得到 -1。若 strlen(findCur) + 1 == name, 代表目前對象已是尋找的目標。 若不是且找到的目標為directory，則往該 subdirectory 繼續往下找。 若最後都沒有找到則回傳 -1。

Create file/directory main.cc

```
static void CreateDirectory(char *name)
{
    // MP4 Assignment
    kernel->fileSystem->Create(name, DirectoryFileSize, true);
}
```

> 此 function 在使用 -mkdir 指令時呼叫。 將 isDir 設定為 true 代表建立的是 directory，而非 file。

FileSystem::Create

```
int FileSystem::Create(char *name, int initialSize, bool isDir)
{
    // declare variables and get root-directory from directoryFile
    strncpy(tmpName, name, sizeof(char)*(strlen(name)+1));
    if (directory->Find(tmpName) != -1) {
        DEBUG(dbgFile, "File " << name << " is already in directory");
        success = 0; // file is already in directory
    }
    else
    {
        freeMap = new PersistentBitmap(freeMapFile, NumSectors);
        sector = freeMap->FindAndSet(); // find a sector to hold the file header
        if (sector == -1)
            success = 0; // no free block for file header
        else if (!directory->Add(name, sector, isDir))
            success = 0; // no space in directory
        else
        {
            hdr = new FileHeader;
            if (!hdr->Allocate(freeMap, initialSize))
                success = 0; // no space on disk for data
            else
            {
                success = 1;
                // everthing worked, flush all changes back to disk
                hdr->WriteBack(sector);
                //// key point for creating directory ////
                if (isDir) {
                    Directory* subDir = new Directory(NumDirEntries);
```

```
                    OpenFile* dirFile = new OpenFile(sector);
                    subDir->WriteBack(dirFile);
                    delete subDir;
                    delete dirFile;
                }
                ////////////////////////////////////////
                directory->WriteBack(directoryFile);
                freeMap->WriteBack(freeMapFile);
            }
            delete hdr;
        }
        delete freeMap;
    }
    delete directory;
    return success;
}
```

> 在 FileSystem 之下建立新的 file 或 directory，利用現存的 Create 加上 isDir 變數，設定要建立的是 file 還是 directory。如果建立的是 directory，則在 FileHeader 建立完成後將此 File 設定 Directory，讓此 File 可以做為 Directory 使用。 因為是使用 sector 來決定該 Directory file 的位置，不在建立 header 或加入 directory 時設定是沒有關係的，

Directory::Add

```
bool Directory::Add(char *name, int newSector, bool isDir)
{
    DEBUG('f', "Adding path " << name);
    char *findCur, cpyName[256];
    strcpy(cpyName, name);
    if (name[0] == '/') findCur = strtok(cpyName+1, "/");
    else findCur = strtok(cpyName, "/");
    if (findCur == NULL) findCur = cpyName; // last in path

    char findNxt[256];
    int i;
    if ((i = FindIndex(findCur)) != -1 && strlen(name) - (strlen(findCur) + 1) ==
0)) {
        DEBUG('f', "Already exist and can't fit target name anymore: " << name);
        return FALSE;
    }

    if (strlen(name) - (strlen(findCur) + 1) > 0) {
        strcpy(findNxt, name + strlen(findCur) + 1);
        Directory* subDir = new Directory(NumDirEntries);
        OpenFile* dirFile = new OpenFile(table[i].sector);
        subDir->FetchFrom(dirFile);
        bool success = subDir->Add(findNxt, newSector, isDir);
        subDir->WriteBack(dirFile);
        delete dirFile;
        delete subDir;
        return success;
```

```
    }

    for (int i = 0; i < tableSize; i++)
        if (!table[i].inUse)
        {
            table[i].inUse = TRUE;
            strncpy(table[i].name, findCur, FileNameMaxLen);
            table[i].sector = newSector;
            table[i].isDir = isDir;
            if (isDir) {
                DEBUG('f', "Create sub-dir " << table[i].name << " " <<
table[i].sector);
            }
            else DEBUG('f', "Create file " << table[i].name << " " <<
table[i].sector);
            // DEBUG('f', "Finish creating file " << table[i].name);

            return TRUE;
        }
    return FALSE; // no space.  Fix when we have extensible files.
}
```

> 將 name 依 '/' 分段，第一個為目前的目標 findCur。如果 findCur 存在於目前的 directory 且沒有接下來的目標則回傳，因為一個 directory 之下不能有相同名字的 file/directory。如果 findCur 存在但還有接下來的目標則往接下來的 subdirectory 繼續尋找。 若以上皆非，則在目前的 directory 以 findCur 作為名字建立新的 file/directory，並以傳入的 isDir 判斷新建的檔案為 file 還是 directory。

Remove file/directory Directory::Remove

```
bool Directory::Remove(char *name)
{
    char *findCur, cpyName[256];
    strcpy(cpyName, name);
    if (name[0] == '/') findCur = strtok(cpyName+1, "/");
    else findCur = strtok(cpyName, "/");
    if (findCur == NULL) findCur = cpyName; // last in path
    int i = FindIndex(findCur);

    if (i == -1 && strlen(name) - (strlen(findCur) + 1) == 0)
        return FALSE; // name not in directory

    char findNxt[256];
    if (strlen(name) - (strlen(findCur) + 1) > 0) {
        strcpy(findNxt, name + strlen(findCur) + 1);
        Directory* subDir = new Directory(NumDirEntries);
        OpenFile* dirFile = new OpenFile(table[i].sector);
        subDir->FetchFrom(dirFile);
        bool success = subDir->Remove(findNxt);
        subDir->WriteBack(dirFile);
        delete dirFile;
        delete subDir;
```

```
        return success;
    }

    table[i].inUse = FALSE;
    return TRUE;
}
```

> name 處理與 Add 和 Find 相同，在此不做贅述。 如果 findCur 存在但不是目標，則繼續往 subdirectory 搜尋，直到找到需要 Remove 的目標。

RecursiveList FileSystem::RecursiveList

```
void FileSystem::RecursiveList(char* name)
{
    Directory *directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);

    int dirSector = directory->Find(name);
    if (!strcmp(name, "/")) directory->RecursiveList(0);
    else {
        OpenFile *subDirFile = new OpenFile(dirSector);
        Directory *subDir = new Directory(NumDirEntries);
        subDir->FetchFrom(subDirFile);
        subDir->RecursiveList(0);
        delete subDir;
        delete subDirFile;
    }
    delete directory;
}
```

> 先尋找目標 directory，然後對該 directory 呼叫 RecursiveList。

Directory::RecursiveList

```
void Directory::RecursiveList(int lvl)
{
    for (int i = 0; i < tableSize; i++)
        if (table[i].inUse) {
            for (int j=0; j<lvl; j++) printf("    "); // padding
            if (table[i].isDir) {
                printf("[D] %s\n", table[i].name);
                OpenFile *subDirFile = new OpenFile(table[i].sector);
                Directory *subDir = new Directory(NumDirEntries);
                subDir->FetchFrom(subDirFile);
                subDir->RecursiveList(lvl+1);
                delete subDirFile;
                delete subDir;
            }
            else
```

```
                printf("[F] %s\n", table[i].name);
        }
}
```

> 以遞迴方式將目前 directory 之下的 file 和 subdirectory 全部 print 出來。 依 lvl 作為目前 subdirectory 的層數，在該 subdirectory 之下的 file 和 directory 前做 padding

(2) Support up to 64 files/subdirectories per directory

```
#define NumDirEntries 64
```

> 將 NumDirEntries 設定為 64。建立 Directory::table 時，大小是依據此變數。

Bonus Assignment

Bonus I: Enhance the NachOS to support even larger file size

> 因為我們使用了linked index，所以改過的code是可以support到64MB的。

```
const int SectorSize = 128;     // number of bytes per disk sector
const int SectorsPerTrack  = 32;    // number of sectors per disk track
const int NumTracks = 16384;        // number of tracks per disk
const int NumSectors = (SectorsPerTrack * NumTracks); // total # of sectors per
disk
```

> 而根據前述，可以透過修改NumTracks來達成目的(因為不能改sector)，在修改前NumTrack前是128KB, 而我們需要將他提升到64MB(512倍)。而32 * 512 = 16384。
>
> 並且因為我們在allocate時是使用遞迴一直往下allocate空間，所以我們的最大空間如果是64MB，單個檔案就可以到64MB。(詳細可以看上述FileHeader::Allocate()的實作)

Bonus II: Multi-level header size

> 因為我們是以linked index來實作，因此叫大的file，就需要較大的fileheader。以下會用test case證明:

```
Name: bonusII_1, Sector: 554
FileHeader contents.  File size: 1000.  File blocks:
555 556 557 558 559 560 561 562
Name: bonusII_2, Sector: 563
FileHeader contents.  File size: 3712.  File blocks:
564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591
592
FileHeader contents.  File size: 3712.  File blocks:
594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621
622
FileHeader contents.  File size: 2576.  File blocks:
624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644
Name: bonusII_3, Sector: 645
FileHeader contents.  File size: 3712.  File blocks:
646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673
674
FileHeader contents.  File size: 3712.  File blocks:
```

> 可以看到上述bonusII_1、bonusII_2、bonusII_3的出現，然後也可以觀察到他們都有開了許多header空間。除了1以外，都需要遞迴call Print()才有辦法全部print出，而call越多次的也就代表其header越大。
> (然後3的部分多到無法截圖，file block應該有到幾萬)