MP4_report_36

109062233蘇裕恆：

50%trace code 50% implementation

109062320朱季葳：

50%trace code 50% implementation

Trace

File Create

參考自MP1 SC_Create，要create一個file會經過以下路徑

1. userprog/exception.cc ExceptionHandler()
2. userprog/ksyscall.h SysCreate()
3. filesys/filesys.h FileSystem::Create()
    - directory->FetchFrom(directoryFile):Fetch contents of file header from disk.
        - directoryFile:“Root” directory – list of file names, represented as a file
    - directory->Find(name):Find the sector number of the FileHeader for file: “name”
        - call FindIndex(name) --> Look up file name in directory, and return its location in the table of directory entries. Return -1 if the name isn’t in the directory.
        - DirectoryEntry:
            - bool inUse
            - int sector
            - char name[FileNameMaxLen + 1]
    - if the file is “Find” --> **create fail**
    - if the file is not found
        - freeMap = new PersistentBitmap(freeMapFile, NumSectors):“bitmap”, an array of bits,each of which can be independently set, cleared, and tested. Most useful for managing the allocation of the elements of an array for instance, disk sectors, or main memory pages.Each bit

represents whether the corresponding sector or page is in use or free.

- FindAndSet():Return the number of the first bit which is clear.
  - 每個bitmap有**numWords = numBits/BitsInWord(constant)** 個index
  - 在FindAndSet()中，會check (0 <= i < numBits)，並於Test(i)中找出i在map對應到的位置，並確認該位置是否為empty，是的話就mark他為non-empty，並作為被使用的space(**將用於new fileHeader的儲存空間**)。如果找到最後都沒有找到empty的位置就return False --> **create fail**
- directory->Add(name, sector):
  - check檔案名字沒有重複
  - check directory還有沒有空的entry可以給新檔案
  - 沒有的話return False --> **create fail**
- hdr->Allocate(freeMap, initialSize), hdr is a FileHeader
  - initialSize" – size of file to be created
  - numSectors：代表該file需要的sector數量
  - freeMap->NumClear()：Return the number of clear bits in the bitmap，當該回傳值小於numSector時，FileHeader::Allocate() return FALSE --> **create fail**
  - FileHeader object會有一個dataSectors table去儲存該file用到的每個data block在disk上是用到哪個sector(儲存該sector的number)，並用FindAndSet()去找空間並allocate給該file如下圖所示

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

for (int i = 0; i < numSectors; i++)
{
    dataSectors[i] = freeMap->FindAndSet();
    // since we checked that there was enough free space,
    // we expect this to succeed
    ASSERT(dataSectors[i] >= 0);
}
return TRUE;
```

以上可以總結出四個點會使得CREATE FILE失敗：

1. file already exist
2. 沒有空間儲存fileHeader
3. directory沒有空的entry
4. free space沒有足夠的空間給data blocks of the file

Notation (From NachOS)

Each file in the file system has
1. A file header, stored in a sector on disk (the size of the file header data structure is arranged to be precisely the size of 1 disk sector
2. A number of data blocks
3. An entry in the file system directory

The file system consists of several data structures:
1. A bitmap of free disk sectors (cf. bitmap.h)
2. A directory of file names and file headers

problems

1. Explain how the NachOS FS manages and finds free block space? Where is this information stored on the raw disk (which sector)?
   簡單來說
   manages and Find free block space-> 使用bitmap與bitmap的查找
   Information stored on the raw disk -> FreeMapSector
   基本上，bitmap為管理整個FS的free block的核心，不管是create一個file一或是create一個filesystem，我們都可以看到
   PersistentBitmap *freeMap = new PersistentBitmap(freeMapFile, NumSectors) 或是
   PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);
   (Persistentbitmap 是從bitmap 繼承下來)
   而在bitmap裡面，我們可清楚看到說

```
//----------------------------------------------------------------------
// Bitmap::FindAndSet
//  Return the number of the first bit which is clear.
//  As a side effect, set the bit (mark it as in use).
//  (In other words, find and allocate a bit.)
//
//  If no bits are clear, return -1.
//----------------------------------------------------------------------

int Bitmap::FindAndSet()
{
    for (int i = 0; i < numBits; i++)
    {
        if (!Test(i))
        {
            Mark(i);
            return i;
        }
    }
    return -1;
}
```

他會找到clear(unused)bit 然後並幫他設為一。因此可以得知她如何manage基本上就是看bitmap的map裡面的值。

```
Bitmap::Bitmap(int numItems)
{
    int i;

    ASSERT(numItems > 0);

    numBits = numItems;
    numWords = divRoundUp(numBits, BitsInWord);
    map = new unsigned int[numWords];
    for (i = 0; i < numWords; i++)
    {
        map[i] = 0; // initialize map to keep Purify happy
    }
    for (i = 0; i < numBits; i++)
    {
        Clear(i);
    }
}
```

(上圖為bitmap的constructor，可以見到他把map裡面的值都先設為0代表unused)

至於第二個問題，我們可以從整個filesystem得結構看出來，

```
private:
    OpenFile *freeMapFile;   // Bit map of free disk blocks,
                             // represented as a file
    OpenFile *directoryFile; // "Root" directory -- list of
                             // file names, represented as a file
};
```

他會將那些資訊以file的形式存在。而他會存在freemapsecotr

```
freeMap->Mark(FreeMapSector);
freeMap->Mark(DirectorySector);

// Second, allocate space for the data blocks containing the contents
// of the directory and bitmap files.  There better be enough space!

ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));

// Flush the bitmap and directory FileHeaders back to disk
// We need to do this before we can "Open" the file, since open
// reads the file header off of disk (and currently the disk has garbage
// on it!).

DEBUG(dbgFile, "Writing headers back to disk.");
mapHdr->WriteBack(FreeMapSector);
dirHdr->WriteBack(DirectorySector);
```

2. What is the maximum disk size that can be handled by the current implementation? Explain why.
   Ans::128 kb

```
const int MagicNumber = 0x456789ab;
const int MagicSize = sizeof(int);
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
```

   Total size = 128(number of bytes per disk sector) * 32 (SectorsPerTrack) * 32 (number of tracks per disk) + 4 bytes
   (We put a magic number at the front of the UNIX file representing the disk, to make it less likely we will accidentally treat a useful file as a disk (which would probably trash the file's contents) - NachOS

3. Explain how the NachOS FS manages the directory data structure? Where is this information stored on the raw disk (which sector)?
   Directory裡面有一個DirectoryEntry table，裡面記錄了該位置是否isUesd，以及file在的sector和file name。

```
Directory::Directory(int size)
{
    table = new DirectoryEntry[size];

    // 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;
}
```

   可以看到，他把table的值設為false

```
ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));

// Flush the bitmap and directory FileHeaders back to disk
// We need to do this before we can "Open" the file, since open
// reads the file header off of disk (and currently the disk has garbage
// on it!).

DEBUG(dbgFile, "Writing headers back to disk.");
mapHdr->WriteBack(FreeMapSector);
dirHdr->WriteBack(DirectorySector);

// OK to open the bitmap and directory files now
// The file system operations assume these two files are left open
// while Nachos is running.

freeMapFile = new OpenFile(FreeMapSector);
directoryFile = new OpenFile(DirectorySector);
```

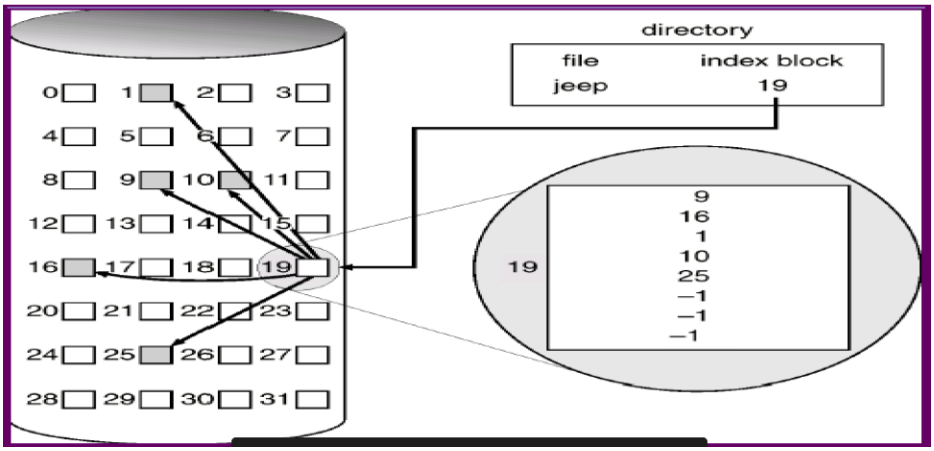
由之前提過的fs的constructor可以看到，他是存在**Directory sector**，順帶一提，若不是第一次創造，fs會讀取的就是FreeMapSector跟DirectorySector。
同時，他管理的方式包括了Fetch From (Read the contents of the directory from disk.), FindIndex(Look up file name in directory, and return its location in the table of // directory entries) 還有add,remove與find等function(api?)可以call。

4. Explain what information is stored in an inode, and use a figure to illustrate the disk allocation scheme of the current implementation.
   NachOs -> **filehdr.cc** (http://filehdr.cc) : Routines for managing the disk file header (in UNIX, this would be called the i-node).
   因此我們看到filehdr.h，

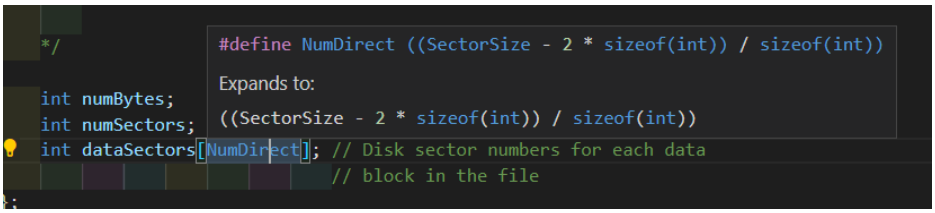
```
private:
    /*
        MP4 hint:
        You will need a data structure to store more information in a header.
        Fields in a class can be separated into disk part and in-core part.
        Disk part are data that will be written into disk.
        In-core part are data only lies in memory, and are used to maintain the data structure of this class.
        In order to implement a data structure, you will need to add some "in-core" data
        to maintain data structure.

        Disk Part - numBytes, numSectors, dataSectors occupy exactly 128 bytes and will be
        written to a sector on disk.
        In-core part - none
    */

    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
};
```

他共存有numBytes->這個file有幾個bytes，numsSectors(同理) 與dataSetcors(每個block的 Disk

sector numbers)
而他的儲存scheme為index allocation



5. Why is a file limited to 4KB in the current implementation?



在fileHeader裡面，我們可以看到一個file的sector數量最多為30。
再給定file sector大小為128bytes以後 我們可以得出結論大概為4kb

Implementation:

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

Combine your MP1 file system call interface with NachOS FS

USERPROG/EXCEPTION.CC

以SC_Create為例，我們把在MP1實作的部分貼過來，詳細實作如下圖

```
case SC_Create:
    DEBUG(dbgSys, "Call create.\n");
    val = kernel->machine->ReadRegister(4);
    {
    filename = &(kernel->machine->mainMemory[val]);
    //cout << filename << endl;
    int init = (int)kernel->machine->ReadRegister(5);
    status = SysCreate(filename , init);
    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;
```

在這裡我們會call到SysCreate，而這個function是定義在userprog/ksyscall.h裡面的。

USERPROG/KSYSCALL.H

在這裡同樣以SysCreate為例，我們會call到filesystem裡面的Create(filename , init)去進行一個file的creation。
而其他部分像是open file, read file, write file, close file也同樣是把MP1的code複製過來，並call到filesystem裡面相對應的實作function

Implement five system calls in FileSystem

在這個部分，除了原本template就有的Create function，我們額外定義了以下function來達到spec所要求的功能：

1. int OpenAFile(char* name)
2. int Read(char* buffer,int size, int id)
3. int Write(char* buffer, int size, int id)
4. int Close(int id)
   而我們將在以下逐步做介紹

INT CREATE(CHAR *NAME, INT SIZE)

在這裡主要的更動是在後面支援subdirectory的部分，其餘create a file並在freeSpace找空間的部分就如同Trace Create function所做的介紹一樣，而我們會在之後對支援subdirectory的部分做詳細的解說。

OPENFILEID OPEN(CHAR *NAME)

在這裡也有支援subdirectory的部分，不過就統一在後面做解說，而這個部分的實作，我們首先到該file所屬的directory做Fetch content from disk的動作，去把content of this directory load到Directory 的table，而table是一個DirectoryEntry object，他裡面的內容如下圖

```
class DirectoryEntry
{
public:
    bool isDir; // 109062233 MP4
    bool inUse;                     // Is this directory entry in use?
    int sector;                     // Location on disk to find the
                                    //   FileHeader for this file
    char name[FileNameMaxLen + 1]; // Text name for file, with +1 for
                                    // the trailing '\0'
};
```

其中包含

1. 之後為了支援subdirectory所設置的isDir(True is this is a directory，因為資料夾中可以有資料夾)
2. 標示該entry是否有content的inUse
3. 還有標示該file/subdirectory在disk上的位置的sector

4. 該file/subdirectory的名字(依照spec所要求，FileNameMaxLen = 9，而這個被define在directory.h)
   接下來，我們call Directory::Find(char *name)去lookup上述的table，而在Find()裡面，他會去call FindIndex(name)去traverse table並找尋到**inUse且name符合傳入的name的那個entry並回傳該entry的index** 如果沒有找到就回傳-1，所以我們接下來會確認回傳到Open()的sector是否為-1(是的話回傳NULL)，否則就new一個OpenFile，並傳入sector number，最後再把FileSystem::OpenFile fileDescriptor設為剛剛new的OpenFile，且因為spec上面講到**Only at most one file will be opened at the same time.** 所以原本在MP1時，filedescriptor是一個table，而在這裡就只是一個OpenFile object。

INT READ(CHAR *BUF, INT SIZE, OPENFILEID ID)

在這裡和MP1的實作很類似，基本上就是先check size, id是valid的，再來去check fileDescriptor是否為NULL，不是的話就call OpenFile::Read(buffer, size)去做read的動作。

INT WRITE(CHAR *BUF, INT SIZE, OPENFILEID ID)

這裡基本上和上述的Read很相似，先check size, id是valid的，再來去check fileDescriptor是否為NULL，不是的話就call OpenFile::Write(buffer, size)去做write的動作。

INT CLOSE(OPENFILEID ID)

spec中提到，the operation will always succeed且不會有messy operation，所以我們在這裡delete fileDescriptor,把他設置為NULL，並return 1。

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

因為在template所給定的空間之下，disk有128KB的空間(>32KB)所以在這裡我們使用linked list scheme的方式去使得file size可以增加，而我們首先新定義了兩個private member in FileHeader class:

1. FileHeader* nextFileHeader --> linked-list中，接在後面的FileHeader object pointer
2. int nextFileHeaderSector --> next fileheader的sector number
   接著我們把NumDirect從原本的((SectorSize - **2** * sizeof(int)) / sizeof(int))改為((SectorSize - **3** * sizeof(int)) / sizeof(int))，因為每個FileHeader object的dataSector的

size固定，而現在多存了一個**nextFileHeaderSector**，所以從-2變成-3。
而為了配合linked list scheme，我們對一些member functions做了更動，而接下來我們將會一一介紹更動的點。

FileHeader::FileHeader(), ~FileHeader()

分別增加initialization以及destruction of nextFileHeader and nextFileHeaderSector。

FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)

在這裡除了template原本的code，我們額外check fileSize是否大於MaxFileSize，如果是的話就把剩餘的部分交由nextFileHeader處理，並recursively Allocate space給current FileHeader的nextFileHeader直到所需的空間都已經allocate給這個file(在所剩空間足夠的情況下，否則return False)。

void FileHeader::Deallocate(PersistentBitmap *freeMap)

除了原本template的code，我們recursively call this function to deallocate the space used by next fileHeader。

void FileHeader::FetchFrom(int sector)

除了原本template的code，我們一樣在 **nextFileHeaderSector != -1** 的情況之下Recursively call FetchFrom(nextFileHeaderSector) to Fetch content of fileHeader from disk.

void FileHeader::WriteBack(int sector)

除了原本template的code，我們一樣在 **nextFileHeaderSector != -1** 的情況之下Recursively call WriteBack(nextFileHeaderSector) to write back modified content to the disk.

int FileHeader::ByteToSector(int offset)

在這裡分為兩個部分

1. offset在linked-list的head(offset < MaxFileSize，因為dataSectors的index是從0開始數)
   和原本template的一樣，return dataSectors[offset / SectorSize]
2. offset不在linked-list的head
   把offset減去MaxFileSize並recursively call

ByteToSector(offset - MaxFileSize)，直到offset < MaxFileSize。

int FileHeader::FileLength()

原本template只有numBytes代表number of bytes in the file，而在這禮因為支援linked-list scheme所以recursively call nextFileHeader->FileLength()並加上他們的值，直到nextFileHeader為NULL，並return遞迴加起來的總值。

void FileHeader::Print()

因為現在不只一個FileHeader for a file,所以我們把template中print data sector以及print file content的code分別貼到PrintDataSectors(), void PrintFileContents() function，並於兩個function裡面個別call recursive call untilnextFileHeader為NULL。如下圖所示。

```
void FileHeader::Print()
{
    int fileLen = FileLength();
    //Bonus2 109062320
    printf("header size : %d \n", sizeof(FileHeader) * divRoundUp(fileLen, MaxFileSize));

    printf("FileHeader contents.  File size: %d.  File blocks:\n", fileLen);

    this->PrintDataSectors();

    printf("\nFile contents:\n");

    this->PrintFileContents();

    //109062320 modify
}
```

Modify the file system code to support the subdirectory

Implement the subdirectory structure

以下將介紹一些因為要支援subdirectory而增加或者更改的data以及function

directory.*

- 新增
    1. 如同 **OpenFileId Open(char *name)**)所提及的，DirectoryEntry新增了**bool isDir** 來支援subdirectory，使得directory裡面可以有directory
    2. DirectoryEntry* GetTable()，在支援recursiveList的部分需要取得每個directory的table並去traverse它。
- 修改
    1. bool Add(char *name, int newSector , bool isDir)：因為資料夾裡面可以有資料夾，所以在加入檔案或資料

夾時，要再使用isDir做區隔

2. bool Remove(char *name):新增把要remove掉的object所屬的table entry的isDir設為false
3. void Directory::List():因為現在要支援subdirectory，所以在list出directory內的element時，依據spec所給的格式，在traverse table時，當table[i].isDir == true時，印出[D]以及directory name;當table[i].isDir == false時，印出[F]以及file name

filesys.*

1. bool CreateDirectory(char *name)
    - 配合mkdir的指令，可以在directory之中make a directory
    - 先找到parent directory，接下來再做一系列的check，基本上跟Create file的function很相似
    - 要check以下幾件事情
        - 如果不是root,則parent directory存在嗎
        - 是否能在parent directory找到同名的file/directory
        - parent directory是否有空間
        - 是否有足夠空間給header
        - 是否有足夠空間給directory(DirectoryFileSize)
    - 達成以上事情之後就new一個directory，並把剛剛找到的free sectors標記給該directory(writeBack)，而當此時create的directory為root or non-root時，會有以下不同的實作方式
        - directory->WriteBack(directoryFile) for root
        - directory->WriteBack(OpenDir(parent_path)) for non-root
2. void SplitPath(char* fullpath, char* parent_dir, char* target_name)
    - split the absolute path(full path into parent_dir and target_name)
        - e.g. aa/ab/cc.txt --> parent_dir = aa/ab, target_name = cc.txt
3. OpenFile* OpenDir(char* parent_path)
    - get the sector of the directory of parent
        - e.g. parent_path = a/b/c, this function will find the sector where store data of c

Support up to 64 files/subdirectories per directory

1. Main code segment for supporting subdirectory

```
/* -------- Main code segment ---------------- */
char* parent_path = new char[500];
char* target_name = new char[500];
char* temp_parent_path = new char[500];
SplitPath(name, parent_path, target_name);
strcpy(temp_parent_path, parent_path);
char* temp = strtok(temp_parent_path , "/");
/* -------- Main code segment ---------------- */
```

因為要支援subdirectory, 再加上傳入的name皆為absolute path，所以filesys.cc裡，以下的function會加入Main code segment來完成實作，並透過OpenDir(parent_dir)來fetch parent directory的content(table)from disk。
加入Main Code Segment的function如下

- bool FileSystem::CreateDirectory(char* name)
- bool FileSystem::Create(char *name, int initialSize)
- OpenFile * FileSystem::Open(char *name)
- bool FileSystem::Remove(char *name)
- void FileSystem::List(char* name , bool recursive)
- void FileSystem::recursiveList(char* name , int layer)

3. 64 files/subdirectories per directory

```
// sectors, so that they can be located on boot-up.
#define FreeMapSector 0
#define DirectorySector 1

// Initial file sizes for the bitmap and directory; until the file
// supports extensible files, the directory size sets the maximum
// of files that can be loaded onto the disk.
#define FreeMapFileSize (NumSectors / BitsInByte)
#define NumDirEntries 64 // 64 files/subdirectories per directory
#define DirectoryFileSize (sizeof(DirectoryEntry) * NumDirEntries)
```

mkdir

在main.cc裡面，當收到mkdir指令時會升起mkdirFlag
這時我們就call到FileSystem::CreateDirectory(char* name)
去實作他

delete the file

if removeFileName != NULL, call
FileSystem::Remove(removeFileName)

List the file/directory in a directory

```
if (dirListFlag)
{   //modified 109062320
    // 109062233 MP4
    if(!recursiveListFlag)
        kernel->fileSystem->List(listDirectoryName , FALSE);
    else
        kernel->fileSystem->recursiveList(listDirectoryName, 0);

}
```

因為只要有-l，無論是-l,-lr,dirListFlag都會被升起，所以放在一起判斷

- 當recursiveListFlag沒有被升起時，call FileSystem::List()
- 當recursiveListFlag被升起時，call FileSystem::recursiveList()

NORMAL LIST

- root directory:fetch directoryFile first, and call directory->list()
- non-root directory:call parent directory's list(因為在這裡路徑最結尾是'/'，所以for a/b/c/, the program will list the files/directories in directory c)
-
```
/*MP4 SPEC says Recursively list the file/directory in a
Directory *directory = new Directory(NumDirEntries);
/* -------- Main code segment ---------------- */
char* parent_path = new char[500];
char* target_name = new char[500];
//root
directory->FetchFrom(directoryFile);
char* temp_parent_path = new char[500];
SplitPath(name,parent_path,target_name);
strcpy(temp_parent_path, parent_path);
char* temp = strtok(temp_parent_path , "/");
/* -------- Main code segment ---------------- */
//non-root
if(temp){
    directory->FetchFrom(OpenDir(parent_path));
}
//modified 109062320
directory->List();
```

RECURSIVELY

在這裡我們分層進行實作，而第0層基本上跟normal list差不多，而在traverse directory的table時，如果遇到isDir==true的entry，就recursively call FileSystem::recursiveList()，並加一層layer，直到traverse完所有object為止。

Bonus

Bonus1

```
const int SectorSize = 128;             // number of bytes per disk sector
const int SectorsPerTrack  = 16384 ;    // 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
```

把disk.h最上方define的值進行修改，因為128x16384x32 = 64MB

所以disk的size即extend to 64MB

```
Last login: Sun Jan 15 20:40:33 2023 from 10.121.186.44
[os22team36@localhost ~]$ cd /home/os2022/os22team36/NachOS-4.0_MP4/code/test/
[os22team36@localhost test]$ ../build.linux/nachos -f
[os22team36@localhost test]$ ../build.linux/nachos -cp num_1000000.txt /a
[os22team36@localhost test]$ ../build.linux/nachos -cp num_1000000.txt /b

[os22team36@localhost test]$
[os22team36@localhost test]$ ../build.linux/nachos -cp num_1000000.txt /c
[os22team36@localhost test]$ ../build.linux/nachos -cp num_1000000.txt /d
[os22team36@localhost test]$ ../build.linux/nachos -l /
[F] a
[F] b
[F] c
[F] d
[os22team36@localhost test]$
```

而在上圖我們copy 4個10MB的檔案進去，並list他得到正確的答案，代表我們的實作成功，由於原本只有128KB，而我們現在成功擴充了。

Bonus2

```
void FileHeader::Print()
{
    int fileLen = FileLength();
    //Bonus2 109062320
    printf("header size : %d \n", sizeof(FileHeader) * divRoundUp(fileLen, MaxFileSize));
```

在FileHeader::Print()這裡，利用FileLength()所算出的bit數量來算出該file所用到的FileHeader object數量，因為每個FileHeader object只能夠容納容量為MaxFileSize的file。最後再乘上FileHeader object的size並印出來即為所求

首先，將num_100.txt, num_1000.txt, num_1000000.txt都放到root directory底下

```
[os22team36@localhost test]$ pwd
/home/os2022/os22team36/NachOS-4.0_MP4/code/test
[os22team36@localhost test]$ ../build.linux/nachos -f
[os22team36@localhost test]$ ../build.linux/nachos -cp num_100.txt /100
[os22team36@localhost test]$ ../build.linux/nachos -cp num_1000.txt /1000
[os22team36@localhost test]$ ../build.linux/nachos -cp num_1000000.txt /1000000
[os22team36@localhost test]$ ../build.linux/nachos -D
Bit map file header:
header size : 2376
Directory file header:
header size : 132
```

接著用 -D 查看資源使用，並可以發現其header size 依據放的檔案大小變大而變大

```
, 81432, 81433, 81434, 81435,
Directory contents:
Name: 100, Sector: 541
header size : 132
Name: 1000, Sector: 550
header size : 396
Name: 1000000, Sector: 632
header size : 355608
```