NACHOS-4.0

MP4

第 21 組

106062173 吳東弦(一起 trace code,負責整理 report)

106062271 郭子豪(一起 TRACE CODE,負責實作 CODING)

一、 Understanding NachOS file system

1.

NachOS 的虛擬機以 bit vector 的方式來管理 free block space，主要實作在 Bitmap class（bitmap.h / bitmap.cc）以及繼承它的 PersistentBitmap class（pbitmap.h / pbitmap.cc）中。

在 Bitmap class 中，Mark()會把傳進來的 sector 所對應的 bit 設為 1，Clear()會把傳進來的 sector 所對應的 bit 設為 0，FindAndSet()會根據 bitmap 找到 free sectors 並將這些 sectors 所對應的 bit 設為 1。

而 PersistentBitmap class 則是 NachOS 真正實例化的 bitmap，除了有 Bitmap class 的功能外，還能對虛擬機的 disk 進行讀寫，FetchFrom()會把 disk 上的 data block 讀進 memory 中，WriteBack()則是把 data 從 memory 寫回 disk 上。

在 disk 上的 file 有 file control block 和 data block，在 NachOS 中 FCB 即為 FileHeader，讀檔時會把 FileHeader 載入記憶體中，再據此找到 data block，而 free block space 實際上也是以 file 的形式存在 disk 上（freeMapFile）。

當 FileSystem（filesys.h / filesys.cc）在 NachOS 中被 new 出來時，如果 disk 沒有要被 format，就直接依照 free block space 之 FileHeader 所在的 sector（FreeMapSector）載入記憶體中，保持 open 的狀態。

如果 disk 被 format，則要重新建立 free block space，先在記憶體中 new 出其 FileHeader（mapHdr）以及對應的 data（freeMap），再將指定要放 mapHdr 的 sector 標記起成「使用中（Mark()）」，並分配 sector 給 freeMap，最後將兩者皆寫到 disk 上（WriteBack()），同時依照 free block space 之 FileHeader 所在的 sector（FreeMapSector）載入記憶體中，保持 open 的狀態。

由此可知，free block space 的 FileHeader 放在 FreeMapSector 中，也就是 sector 0，而由於 sector 1 放 directory 的 FCB，故 free block space 的 data block（freeMapFile）放在 sector 2。

2.

在 disk.h 中可知，此 disk 有 32 個 track，每個 track 有 32 個 sector，共 1024 個 sector，每個 sector 為 128 Byte，故此 disk 共 128 KB。

此外 1024 個 sector 對應 freeMapFile 為 1024 bits，即 128 Byte，由此可知 freeMapFile 恰放滿一個 sector，解釋了上題 freeMap 為什麼只需要 sector 2。

3.

directory 主要實作在 Directory class 和 DirectoryEntry class 中（directory.h / directory.cc）。Directory class 為一張表（table），一格為一個 DirectoryEntry，代表可放一個 file，紀錄該格是否已被使用（inUse），如果已被佔用，該檔案的 FileHeader 放在哪個 sector，檔名（name）為何。

在 Directory class 中，可以新增（Add()）、移除（Remove()）、表列（List()）檔案，以及找到檔案的 FileHeader 後（Find()），再將其讀進 memory 中（FetchFrom()），或是寫回 disk 上（WriteBack()）。

directory 和 free block space 同樣也是以 file 的形式存在 disk 上。當 FileSystem（filesys.h / filesys.cc）在 NachOS 中被 new 出來時，如果 disk 沒有要被 format，就直接依照 directory 之 FileHeader 所在的 sector（DirectorySector）載入記憶體中，保持 open 的狀態。

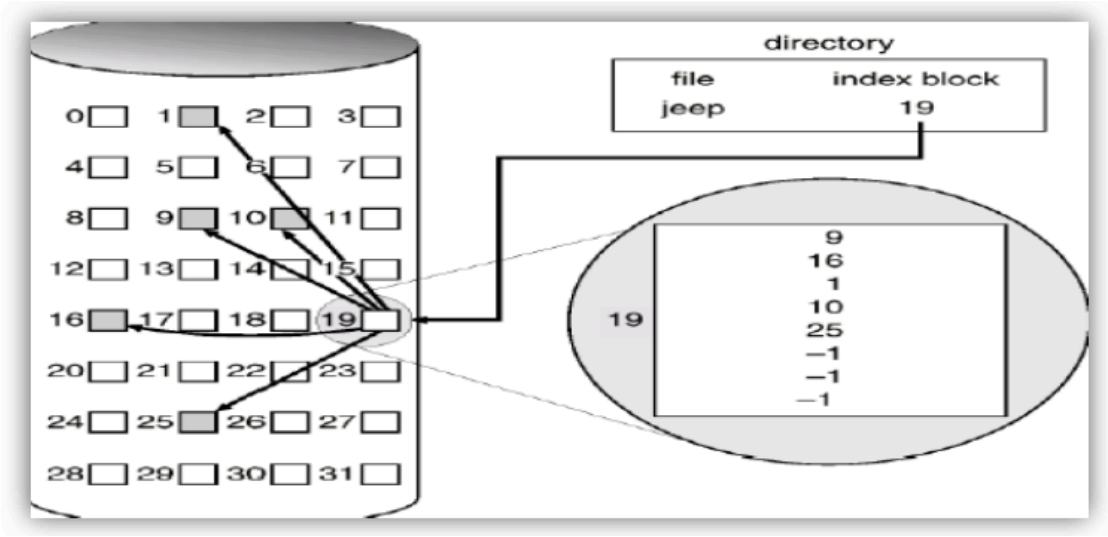
如果 disk 被 format，則要重新建立 directory，先在記憶體中 new 出其 FileHeader（dirHdr）以及對應的 data（directory），再將指定要放 dirHdr 的 sector 標記起成「使用中（Mark()）」，並分配 sector 給 directory，最後將兩者皆寫到 disk 上（WriteBack()），同時依照 directory 之 FileHeader 所在的 sector（DirectorySector）載入記憶體中，保持 open 的狀態。

由此可知，directory 的 FileHeader 放在 DirectorySector 中，也就是 sector 1，而由於 sector 2 放 free block space 的 data block（freeMapFile），故 directory 的 data block（directoryFile）放在 sector 3 和 sector 4。

4.

在 NachOS 中，inode 就是 FileHeader，存放檔案大小（numBytes）、檔案用了多少 sector（numSectors），以及指向 data block 每個 sector 的 entry（dataSectors[]）。

由於 dataSectors[] 為一個陣列，由此可知 NachOS 的檔案系統是以 Indexed Allocation 來存放檔案的。



5.

在 FileHeader 所在的 sector 中，以 4 個 Byte（sizeof (int)）為一個單位，前 4 個 Byte 放 file 大小（numBytes），次 4 個 Byte 放 file 的 data block 使用了幾個 sector（numSectors），剩下 120 個 byte 即 30 個 entry（NumDirect），最多對應 30 個 sector，因此一個 file 的 data block 最多為 3.75 KB，大約為 4 KB。

而由 filesys.cc 中可知，NachOS（唯一）的 directory 共 10 個 entry，即可放 10 個檔案，共 37.5 KB，表示目前的檔案系統只能映射 37.5 KB。

此外，directory 之 entry 的資料（DirectoryEntry），佔用了 20 Byte （inUse + sector + name[]），10 個 entry 共 200 Byle，會超過一個 sector 的大小（128 Byte），解釋了第 3 題 directoryFile 為什麼需要 sector 3 和 sector 4。

二、 Modify FS code to support I/O system call and larger file size

1.

I/O system call 呼叫的流程大致和 MP1 相同，Create 的部分多傳了一個表示檔案大小（size）的參數，並讀入暫存器$5。

呼叫流程以 file create 為例，case SC_Create（exception.*）呼叫 SysCreate()（ksyscall.h），SysCreate()呼叫 kernel→interrupt→CreateFile()（interrupt.*），CreateFile()呼叫 kernel→CreatFile()（kernel.*），CreateFile()呼叫 fileSystem→Create()（filesys.*）。

因此，實作部分主要放在 filesys.c 的 FileSystem class 中，各個 API 也與 MP1 大致相同，一樣多新增一張表（fileDescriptorTable[]）方便記錄開啟過的檔案。

2.

為了 data block 不增加額外的 pointer，以及使 FileHeader 本身保持洽使用一個 sector，以 two-level Indexed Allocation 來實作，主要實作在 filehdr.cc 中。

第一層 index table （LevelOneTable[]）有 30 個 entry（NumDirect），可對應到 30 張第二層 index table，每張表有 32 個 entry，實作上將其併成一張表（dataSectors[]），共對應 960 個 sector（NumDirect * NumberofSecondEntry），即最大可支援單檔 120KB。

分別修改了 Allocate()、Deallocate()、FetchFrom()、WriteBack()。

三、 Modify the file system code to support subdirectory

1.

在 directory.*中，為了要 recursively trace，需判斷當前資料夾的 entry（DirectoryEntry）為 file 還是 directory，因此新增了一個變數 isDir，同時改寫 List() 成能夠做 recursively list。

在 filesys.*中，由於資料夾變成多層結構，多實做了 Find_My_Directory()來判斷是否要繼續往下 trace，並回傳當前資料夾的 FileHeader。而原本的函數中，多傳了 isDir 判斷要創建的是資料夾還是檔案，並利用 Find_My_Directory()找到當前資料夾再進行操作，另外 List()多判斷是否要進行遞歸處理。

此外，在 create directory 時，在 directory.*多實做了一個 Clean()，先將 disk 上指定的位置清零，再從記憶體寫回去（WriteBack()），避免因為 disk 的殘留值，導致在判斷 DirectoryEntry 的 inUse 時發生錯誤。

2.

此題直接將 filesys.h 的#define NumDirEntries 改為 64 即可。

四、 Bonus

1.

```
const int SectorSize = 128;   // number of bytes per disk sector
const int SectorsPerTrack  = 32;  // number of sectors per disk track
//bonus1
const int NumTracks = 16384;    // number of tracks per disk
const int NumSectors = (SectorsPerTrack * NumTracks);
                    // total # of sectors per disk
```

- 先將 disk 變大成 16384*32*128 = 67108864 = 64MB
- 再將 fileheader 的 hierarchy 變成四層使其可記錄 30*32*32*32*128 = 125829120 = 120 MB 的檔案 只要將 disk 變大就可以紀錄 120MB 的檔案

2.

```
bool
FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
{
//todo bonus
    numBytes = fileSize;
    numSectors  = divRoundUp(fileSize, SectorSize);
    int numThrLevelTables=divRoundUp(numSectors,NumberofEntry);
    int numSecLevelTables=divRoundUp(numThrLevelTables,NumberofEntry);
    int numFirLevelTables=divRoundUp(numSecLevelTables,NumberofEntry);
    int total=numSectors+numThrLevelTables+numSecLevelTables+numFirLevelTables;
```

- 因為我們 fileheader 使用 hieracy 的結構紀錄其所占的 sector
- 故將 numFir + numSec + numThr + 1 就會是總 header 需要的大小，而這個數字由上圖可知是由 numsectors 決定的，也就是檔案大小決定了 header 的大小，因為 numsectors 是變數，故超過三種以上

3.

```
//todo bonus
    if (curdir->isDir(nameforfile) && recursive)//recursive & dirctory
    {
        char NextPathName[300];
        strcpy(NextPathName, pathname);
        int now_path_length = strlen(NextPathName);
        NextPathName[now_path_length] = '/';//to undertake next file name

        OpenFile* tardirFile = new OpenFile(sector);//Open target dirctory openfile
        Directory* tardir = new Directory(NumDirEntries);
        tardir->FetchFrom(tardirFile);

        //remove down this dirctory
        for (int i = 0; i < tardir->tableSize; i++)
        {
            if (tardir->table[i].inUse)
            {
                // after '/' offset is tarPathName+now_length+1
                strcpy(NextPathName+now_path_length+1, tardir->table[i].name);
                //recursive
                Remove(NextPathName, TRUE);
            }
        }
        delete tardir;
        delete tardirFile;
    }
```

- 這是多加在 remove 的部分，先判斷是否為資料夾（所以我們在 dirctory 新增了 isDir(char* name)來判斷要刪除的檔案是否為資料夾），接者如果是資料夾的話，我們找資料裡 inUse 的部份遞迴進去，就可以將這個資料夾內所有的資料刪除了。

五、 Summary & FeedBack

本次實作 2-1 大致與 MP1 相同；2-2 由於採用 Indexed Allocation 所以 bonus 單檔 64 MB 不好實作；3-1 為這次實作重點，概念並不難，但是很容易出 bug，要不斷觀察 disk 的資訊。

其一是創建檔案與創建資料夾是判斷不一樣的東西，前者看 free block space，後者看 DirectoryEntry 的 inUse，要把 directory 在 disk 部分先清空再寫入。

其二是 disk 的 format 並不會真正把 disk 清零，導致雖然做了格式化，卻還有殘留值，所以得把整個 disk 刪掉再重建。