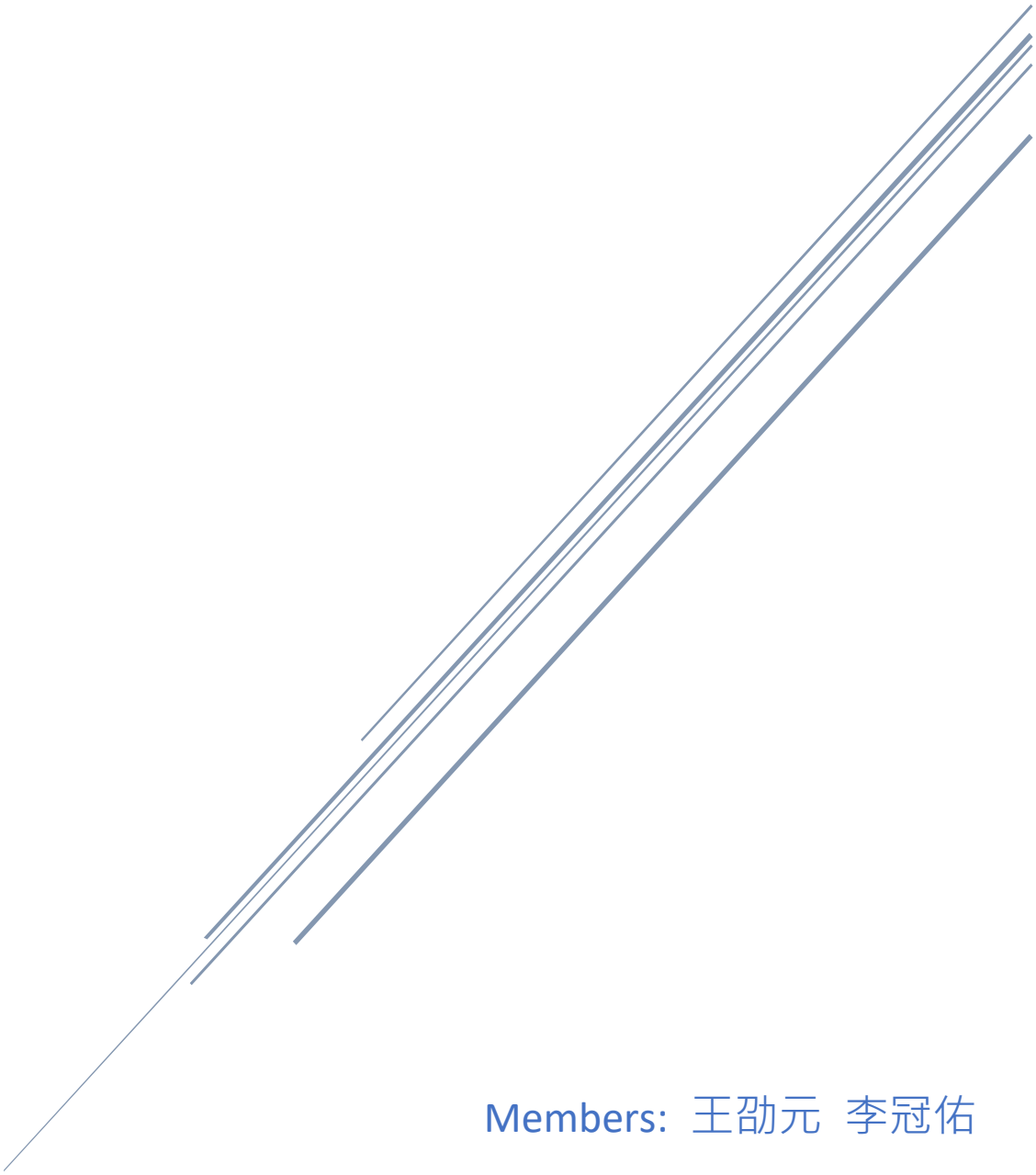
MP4: FILE SYSTEM

Report



Members: 王劭元 李冠佑

Contribution: 50 % 50 %

Part 1: Understanding NachOS file system

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

```
#define FreeMapSector 0
```

NachOS 是用一個 PersistentBitMap 去 handle 現在剩下的 sector。在 file system 的建構子中，就可以看到有一個 PersistentBitMap* freeMap 被 new 出來，並且可以看到在 define 的地方也定義好 FreeMapSector 為 0，因此我們知道這個 freeMap 的 Openfile 的存放位置是在 sector 0，當我們在在 file system 的建構子 format 的時候，由於剛開始會 initialize empty directory 和 bitmap，因此要先將 bitmap 裡 0 和 1（存放 bitmap free sector 和 directory of files 的 file header 的 sector）給 Mark 起來，以後在 allocate 時才不會再 allocate 到這兩個 sector，接下來要將更改後的 freeMap 寫回 disk 時，NachOS 會先以 FreeMapSector open 一個 file，然後再將更改後的 freeMap writeback 到剛剛 open 的 file，也就是寫回 disk 裡。而對 bitmap 的操作主要是在 bitmap.cc 當中，在 bitmap.cc 的建構子裡，它會先將陣列 map 清空，並把每個 bit 都初始成 0，代表都未使用，再來，剛剛提到的 Mark 是將傳入的 bit i 設為 1，代表這個 block 已被使用，除此之外，常用的還有 FindAndSet，Test，Clear 和 NumClear，FindAndSet 的用途就是去尋找第一個未使用的 bit，裡面會用 Test(i) 去檢查這個 bit 是否為 1，若為 0，即代表這個 bit 現在可以使用，NumClear 是會回傳現在縱共有幾個 bit 是尚未被使用的，Clear(i)就是將 bit i 給清空，讓它變回 unallocated。

接下來補充一下上面講過的 freeMap file header 的部分，NachOS 會用 header ->Allocate 去幫 file 裡的 data block allocate 空間，之後再將 header 寫回 disk 裡，舉例來說，在 filesystem 建構子的 format 裡，首先是新建一個 header

```
FileHeader *mapHdr = new FileHeader;
```

再來，要 allocate space 給 mapHdr 的 datablock，它會從 freeMap 裡去找空間存放

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

接下來，就可以將 header writeBack 到 disk 裡，這裡因為是 freeMap 所以是寫回到 FreeMapSector(sector 0)

```
mapHdr->WriteBack(FreeMapSector);
```

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

```
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
```

NumTracks 代表一共有 32 個 track，SectorsPerTrack 代表每個 track 有 32 個 sector，SectorSize 代表每個 sector 是 128 bytes，因此 Maximum disk size 為 32*32*128bytes

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

```
#define DirectorySector 1
```

從 filesystem.cc 可以得知，directory structure 的 header 存在 DirectorySector(sector 1)的位置，和 FreeMap 類似，在 filesystem 的建構子裡它就會先去以 NumDirEntries 去 new 一個 directory，同時也去 new 一個 file header，剛開始一樣要先在 freeMap 裡 Mark DirectorySector，這樣其他 file 才不會去用到這個 sector

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

再來就是要去 Allocate space 給 directory 的 data block，

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

接下來，就是要把 header writeBack 到 sector 1，和 FreeMap 一樣，之後也會用 DirectorySector 去 openfile，

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

這個 open 的 directory 就是我們的 root，往後對 root 下的檔案和資料夾的增減在這進行，然後把更動 directory 寫回這個 file 裡，

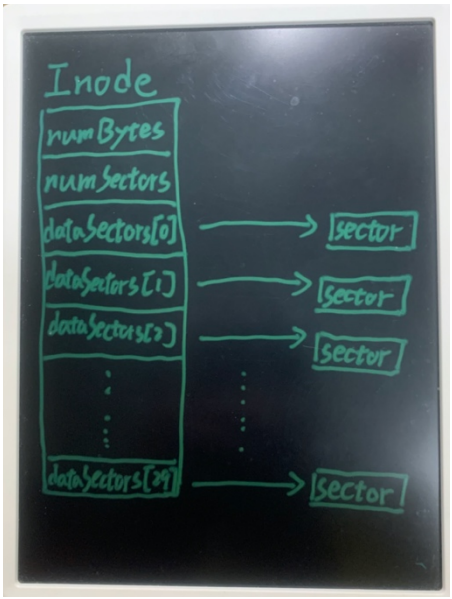
```
directory->WriteBack(directoryFile);
```

就完成了對 directory structure 的變動（在建構子裡會是空的，因為沒有新增任何檔案或資料夾）。

在 directory 裡，會有一個 table 去紀錄現在這個 directory 之下有多少個 file 或 sub directory，除此之外，還有三個變數去管控 file 的狀態，isDir 是來判斷在這個 entry 存的是 file 還是 directory，isUse 是來判斷這個 entry 是否有被使用，sector 則是這個 file 的 header 是存在哪個 sector，最後 name 就是這個 file 的 filename。

(4) Explain what information is stored in an inode, and use a figure to illustrate the disk allocation scheme of current implementation.

Inode 指的就是 fileheader，裡面存有 numBytes，numSectors 以及 dataSectors 的資訊，numBytes 存的是這個 file 的檔案大小(bytes)，numSectors 存的是這個 fileheader 總共使用幾個 data sector，dataSector 是一個陣列，裡面存有每個 data block 是存在哪個 sector 的資訊，dataSector 的陣列大小是 NumDirect，一個 sector 是 128 bytes，其中要留 8 個 bytes 去存 numBytes 和 numSectors，剩下的 120bytes 可以存 30 個 integer，所以 NumDirect = 30。



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

因為現在的 file header 只存在一個 single sector 裡，那一個 file 最多能用的空間就是 32 個 sector（扣除 header 的資訊後會少於 32），由於要留兩個 sizeof(int)存 numBytes 後 numSectors，那一個 file 最多就只會有 3840bytes 的空間能存 data，約 4KB。

Part 2: Modify code to support larger file size

1. Implement Five System calls

   首先由於大部分和 MP1 一樣，於是我先將 MP1 的架構貼上來再做修改。

   exception.cc 中不一樣的地方是 Create 的地方，現在要多給一個 size 進去，所以我有再從 register5 讀進一個 size 的 integer，並且也將它傳給 ksyscall。其他地方都是一樣的。

   而 ksyscall 的地方，不一樣的地方有：

   SysCreate 的地方，現在也會將 size 傳給 file system。

   SysOpen 的地方就比較特別了，因為在這次作業有保證一次只會有一個 file 被 open，所以我們就選擇不去 handle 一個 openFileTable，我們將現在被 Open 的 file，用 file system 中一個 public 的指標 activeFile 去指到它。因此我們也就判斷不會回傳 OpenFileId，我們就判斷若有成功打開，就回傳 1，反之，則回傳 0。

   而 SysWrite 與 Read 的地方我們現在就不會去呼叫 File System 的 Read, Write 了，我們會直接去找到 activeFile，用 OpenFile 的 Read, Write 來進行讀寫。

   而 SysClose 我們則是直接新加了一個 file system 的 function 叫 CloseFile()，而做的事情也就是直接把 activeFile delete 後回傳 1。

2. Enhance the FS to let it support 32KB file size

   我們在這邊選擇的方式是 combined index scheme，因為看到原本就是用 index scheme 了，並且也有看到 bonus 也要做更大，但若可以根據 filesize 去調整的話那會更好，所以選擇用適合 handle 較大型的檔案並且也有彈性的 combined index scheme。

   而在這邊我們只要分成兩層就夠了，第一層的 file header sector 的 dataSectors 中存的每一個 sector number，現在就不是真的 dataBlock 了，而是另一個 file

header，而在第二層的 file header 中，dataSectors 中的 sector number 才真的會是指到 dataBlocks。

我們修改的地方就是在 filehdr.cc 中：

Allocate()

第一個地方是在 Allocate()：Allocate 中我們會判斷，若 fileSize 大於 OneLevelSize(30*128)，那我們在 allocate 的時候就會用兩層的方式。

先找到一個 sector 給第一層的 file header，接著就去看若 fileSize 還大於 Onelevel 的話，那就 new 一個 file header 去 allocate 一個 OneLevel 大小的空間去存 data，若 Allocate 成功就會將 fileSize 往下減 OneLevel 的大小並繼續迴圈。

但若 fileSize 已經小於 OneLevel 的話，那 new 的 file header 就會只去 Allocate fileSize 大小的空間並且把 fileSize -= fileSize，接著就會跳出迴圈。

並且在每次 new 出來的 file header Allocate 完空間後，要把這個 new 的 header 存到 dataSectors[i]的地方，讓第一層的 file header 可以找到第二層的 file header 所在的位置。

```
else if(fileSize > OneLevelSize){
    int i = 0;
    while(fileSize > 0)
    {
        dataSectors[i] = freeMap->FindAndSet();
        ASSERT(dataSectors[i] >= 0);

        FileHeader *next_level_hdr = new FileHeader;

        if (fileSize > OneLevelSize)
        {
            bool success = next_level_hdr->Allocate(freeMap, OneLevelSize);
            if(!success) return FALSE; // space is not enough
            fileSize -= OneLevelSize;
        }
        else
        {
            bool success = next_level_hdr->Allocate(freeMap, fileSize);
            if(!success) return FALSE; // space is not enough
            fileSize -= fileSize;
        }
        next_level_hdr->WriteBack(dataSectors[i]);
        delete next_level_hdr;
        i++;
    }
    numSectors = i;
```

DeAllocate()

第二個地方是在 DeAllocate 的地方，若今天這個 header 中存的 numBytes 大於 OneLevelSize 的話，代表說這是第一層的 header，而 dataSectors 中存的 sector number 是其他也要被 deAllocate 的 file header，因此就會去拿到下一層的 file header 再去 call 他的 deAllocate，最後也把那個 file header 給 Clear 掉。

而若小於 OneLevel 的話就代表存放的是 datablock，故就跟原本 implement 的一樣。

```
void FileHeader::Deallocate(PersistentBitmap *freeMap)
{
    if(numBytes > OneLevelSize){
        for (int i = 0; i < numSectors; i++)
        {
            FileHeader *hdr = new FileHeader;
            hdr->FetchFrom(dataSectors[i]);
            hdr->Deallocate(freeMap);
            freeMap->Clear((int)dataSectors[i]);
            delete hdr;
        }
    }
    else{
        for (int i = 0; i < numSectors; i++)
        {
            ASSERT(freeMap->Test((int)dataSectors[i])); // ought to be marked!
            freeMap->Clear((int)dataSectors[i]);
        }
    }
}
```

ByteToSector()

第三個地方在 ByteToSector，由於若 offset 大於 OneLevel 的話，offset 就有可能超過一個 file header 能算出的 sector 數，故我們會先判斷若大於 Onelevel 的話，就會先 divRoundDown，算出他是在第二層的哪個 file header 中，去 fetch 出那個 file header 後，再用那個 file header 去 call ByteToSector，但這邊 offset 的值就要減掉剛剛算出他在第幾個 file header 去成上一個第二層的 file header 能存的 data 大小，也就是 OneLevelSize。

```
int FileHeader::ByteToSector(int offset)
{
    if (numBytes > OneLevelSize)
    {
        FileHeader *hdr = new FileHeader;
        int sector = divRoundDown(offset, OneLevelSize);
        hdr->FetchFrom(dataSectors[sector]);
        int value = hdr->ByteToSector(offset - sector * OneLevelSize);
        delete hdr;
        return value;
    }
    else
    {
        return (dataSectors[offset / SectorSize]);
    }
}
```

Part 3: Modify code to support subdirectory

1. Implement subdirectory structure

Createdirectory(name)

首先我們先去 implement mkdir，我們在 main.cc 裡面看到如果有下 mkdir 的指令，就會把後面跟著的名字也一起往下傳到 file system 裡面我們新寫的 CreateDirectory 的 function。

我們會用一個叫 strtok(str, target)的 function 來 parse，首先就用一個 token 去紀錄第一次的 strtok(name, "/"), 接著就是去跑一個 while 迴圈。在 while 迴圈裡面會判斷若當今天在這層 directory 找不到這個名字的 directroy 的話，那現在這個 token 就是我們要 create 的 directory 的 name，就會 break 出這個迴圈。因為我們不會去 create directory on non-existing directory。但若這層還有找到的話，那就會去 new 這層找到的 directory 的 Openfile，並且讓一個記錄目前在哪個 directory 的 Directory* directory 去 FetchFrom 這個 OpenFile。

等到 break 出來後，我們有的資訊有：要建這個新的 subDir 的 directory（以下稱它為 parent），和 parent 的 openfile(以下稱為 recur)，和我們要建的 subDir 的 name。

於是我們就可以找到一個 sector 來存放新的 subDir 的 header。找到後，把 name Add 進 parent 裡面，接著就可以 new 一個 header（以下稱 hdr），並且讓 hdr 去 Allocate 一個 directoryFileSize 的空間。

最後就把 hdr write 回剛剛找到的 free_sector，並用一個 openFile 打開，然後新建一個 Directory 存到 WriteBack 回剛剛新增的 openFile，並且把 parent write back 回 recur，因為在 parent 下新增了一個資料夾，等於 parent 的結構也變了，故也要重新把 parent 寫回 disk。

最後再把 freeMap 寫回 disk 去更新 freeSector

這樣中間若沒有空間不夠等等的問題，就完成了 mkdir。

Create(name, size)

當今天要新建一個 file 的時候，就會 call 到這邊。而我們在這邊做的修改就是，在基於原本的 implement 下，我們會先幫他們用上面類似 createDirectory 的方式 parse 到要建檔的資料夾。

一樣都會用 strtok(name, "/")去開始 parse 到要建檔的資料夾，但現在只是把 createDirectory 換成 CreateFile，所以就不需要去 new 出一個 directory，只要找到一個 free_sector，然後在上面 new 一個 file header，並讓 hdr 去 allocate 出一個 filesize 的空間。並把 directory 的結構也透過 Openfile 寫回 disk。

最後再把 freeMap 寫回 disk 去更新 freeSector

這樣就完成一個 file 的 create。

Open(name)

當今天要 open 一個 file 的時候，就會 call 到這個 function。

一開始也是用 strtok(name, "/")去 parse 路徑，但這邊不一樣的事，while 裡面 if 的判斷式跟上面不一樣。這邊是若在目前的 directory 中，找到的 token 是一個檔案的話，就會 break 出 while 迴圈。因為這樣就代表我們目前會處在要 Open 這個 file 的 directory，這樣 break 出來會比較好操作。

接下來我們就夠透過 directory->Find()這個 function 來找到這個 file 的 file header 在的 sector。若真的有找到這個 sector，就會用 Openfile 去打開，並讓 activeFile 去指到這個 file。

Remove(name)

當今天要 remove 一個 file 的時候，就會 call 到這個 function。

而這邊 Parse 路徑的方法與 Open 一樣，若碰到 file 就會停下來。

而刪除的部分，先透過 directory->Find()這個 function 來找到這個 file 的 file header 在的 sector。若沒找到就會 return。若有找到，就會 new 一個 file header 去 FetchFrom 那個 sector，接著就會 hdr->Deallocate(freeMap)。做完後也會用 freeMap 去 clear header 所在的 sector。再從 parent directory 中去 call directory->Remove(token)，把他從 directory 中 remove。

最後再把 parent directory 以及 freeMap 重新寫回 disk 去更新。

List(name)

當今天想要 List 出一個 directory 中有哪些 file and subdirectory 時，就會 call 到這個 function。

而這邊也是用 strtok(name, "/")去做 parsing，但就不會特別設判斷是讓他停在目標 directory 的 parent，因為我們就是希望去 List 出目標的 directory 中的檔案及資料夾，故就一直等到 token == NULL 後，break 出迴圈。

接下來就直接去 call 這個 directory->List()，這樣就可以列出所有 inUse 的空間了。

RecursiveList(name)

當今天想要印出目標檔案下的所有檔案及資料夾的結構，那就會 call 到這個 functoin。

一開始在 parsing 路徑的部分，也跟 List 一樣，到目標資料夾後，才會因為 token

== NULL 才被 break 出來。

下面這邊，我就直接透過 call directory->RecursiveList(0)去實作。而 directory 的 RecursiveList 為了讓他可以看出結構，所以我們會傳一個 num 進去代表要縮排縮幾格，而現在因為呼叫的 directory 不用縮排，故傳 0 進去。

directory->RecursiveList

在這裡面就會去掃這個 directory 的 table，若有 inUse 的話，就會根據傳進來的 num 先印出 num 個空格，接下來就會接著印出 table[i].name。

Print 完後就會去判斷這個 inUse 的是不是資料夾，若是的話，那就會去抓出他的 sector，並用 Openfile 去打開這個資料夾的 file header。接著再 new 一個 directory 並且去 FetchFrom 剛剛打開的 file header。再 call nextDir->RecursiveList(num+1)。這樣就能達到印出某個資料夾下的結構了。結果如下

```
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -lr /
{D}: t0
 {F}: f1
 {D}: aa
 {D}: bb
  {F}: f1
  {F}: f2
  {F}: f3
  {F}: f4
 {D}: cc
{D}: t1
{D}: t2
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -lr /t0
{F}: f1
{D}: aa
{D}: bb
 {F}: f1
 {F}: f2
 {F}: f3
 {F}: f4
{D}: cc
```

2. Support up to 64 files/subdirectories per directory

   在這邊我們就是將 directory.h 中的 NumDirEntries 改成 64。

```
// MP4 part3-2
#define NumDirEntries 64
#define DirFileSize (MaxDirNum * sizeof(DirectoryEntry))
```

Part 4 Bonus

(1) Bonus I

首先，要將 disk extend 到 64MB 的方法就是去 disk.h 裡將 NumTracks 從 32 改成 16384 (64MB / SectorSize / SectorPerTrack)，而 support up to 64MB single file 的部分就是將原本我們在 part2 implement 的兩層增加到四層，在 Allocate 和 ByteToSector 裡增加 TwoLevelSize(30*30*128)和 threeLevelSize(30*30*30*128)，其他的寫法就和 part2

implement 類似，判斷式改成從最大的 threeLevelSize 開始判斷，由 file size 大小去決定要去弄哪一層的，這樣就可以做到 support up to 64MB file。

```
if(fileSize > ThreeLevelSize){
    int i = 0;
    while(fileSize > 0)
    {
        dataSectors[i] = freeMap->FindAndSet();
        ASSERT(dataSectors[i] >= 0);

        FileHeader *next_level_hdr = new FileHeader;

        if (fileSize > ThreeLevelSize)
        {
            bool success = next_level_hdr->Allocate(freeMap, ThreeLevelSize);
            if(!success) return FALSE; // space is not enough
            fileSize -= ThreeLevelSize;
        }
        else
        {
            bool success = next_level_hdr->Allocate(freeMap, fileSize);
            if(!success) return FALSE; // space is not enough
            fileSize -= fileSize;
        }
        next_level_hdr->WriteBack(dataSectors[i]);
        delete next_level_hdr;
        i++;
    }
    numSectors = i;
}
```

```
else if(fileSize > TwoLevelSize){
    int i = 0;
    while(fileSize > 0)
    {
        dataSectors[i] = freeMap->FindAndSet();
        ASSERT(dataSectors[i] >= 0);

        FileHeader *next_level_hdr = new FileHeader;

        if (fileSize > TwoLevelSize)
        {
            bool success = next_level_hdr->Allocate(freeMap, TwoLevelSize);
            if(!success) return FALSE; // space is not enough
            fileSize -= TwoLevelSize;
        }
        else
        {
            bool success = next_level_hdr->Allocate(freeMap, fileSize);
            if(!success) return FALSE; // space is not enough
            fileSize -= fileSize;
        }
        next_level_hdr->WriteBack(dataSectors[i]);
        delete next_level_hdr;
        i++;
    }
    numSectors = i;
}
```

而驗證方法是，我們利用原本給的 num_1000000.txt，利用-cp 六次產生六個 10MB 的 file，再用-l 把檔案 list 出來，沒有 segmentation fault，然後剛剛的六個檔案都有成功 create 和 open，就代表我們的實作是正確的。

```
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -cp num_1000000.txt /1_2
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -cp num_1000000.txt /1_3
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -cp num_1000000.txt /1_4
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -cp num_1000000.txt /1_5
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -cp num_1000000.txt /1_6
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -l /
t0
t1
t2
1_1
1_2
1_3
1_4
1_5
1_6
```

(2) Bonus II

在 implement Allocate 的時候我們會依照檔案的大小去選擇用哪一個 level 的 allocate 方式，然後就會用遞迴去跑，並在每次遞迴前都會去 new 一個新的 header，這樣一來隨著檔案大小的不一樣，所用的 header size 也會跟著不一樣，驗證方法我們是去將 file 每個 header 所存在的 sector 印出來，如下圖：

```
[os20team42@lsalab ~/NachOS-4.0_MP4/code/build.linux]$ cd ../test
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -f
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -cp num_100.txt /100
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -cp num_1000.txt /1000
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -cp num_10000.txt /10000
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -cp num_20000.txt /20000
[os20team42@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -KYL
Name: 100, Sector: 542
Name: 1000, Sector: 551 552 583 614
Name: 10000, Sector: 634 635 666 697 728 759 790 821 852 883 914 945 976 1007 1038 1069 1100 1131 1162 1193 1224 1255 1286 1317 1348 1379 1410 1441 1472 1503
Name: 20000, Sector: 1524 1525 1526 1557 1588 1619 1650 1681 1712 1743 1774 1805 1836 1867 1898 1929 1960 1991 2022 2053 2084 2115 2146 2177 2208 2239 2270 2301 2332 23
63 2394 2425 2456 2457 2488 2519 2550 2581 2612 2643 2674 2705 2736 2767 2798 2829 2860 2891 2922 2953 2984 3015 3046 3077 3108 3139
```

可以看到隨著檔案大小的不一樣，header 數量也會不一樣，代表所佔的 header size 也會隨之改，小檔案如 100 就有小的 header size，大的檔案如 10000，20000，就有比較大的 header size，且 100001 和 20000 屬於同個 level，但印出來存 header 的 sector 數量明顯不一樣，可以證明 header size 有隨檔案大小改變。

印出這個資訊我們是在 main.cc 裡新增一個-KYL 的 command，它會將一個新增變數 showHeaderSize 設為 true，接著就會在下方 FILESYS_STUB 裡增加一個判斷式，在 showHeaderSize = true 時，它會去 call 到 filesystem 的 PrintHeaderUse，

```
else if (strcmp(argv[i], "-KYL") == 0)
{
    showHeaderSize = true;
}
```

```
if(showHeaderSize)
{
    kernel->fileSystem->PrintHeaderUse();
}
```

在 Filesystem::PrintHeaderUse 裡，我們會先去 fetch root directory structure，接下來就會去 call 到 Directory::PrintUse

```
void FileSystem::PrintHeaderUse()
{
    Directory *directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);
    directory->PrintUse();
}
```

在 Directory::PrintUse 裡，我們會掃過整個 directory table，若 table[i].inUse == true，就去判斷現在存在這個 table 位置的是 directory 還是 file，若是 directory，就去 fetch 下一層的 directory structure 後去遞迴，若 table 位置是 file，會先將他的 sector 印出來，接下來它會先從 table[i].sector 找到 header 存的 sector，然去把它從 disk fetch 出來，再去 call FileHeader::PrintUse

```
void Directory::PrintUse()
{
    FileHeader *hdr = new FileHeader;
    for (int i = 0; i < tableSize; i++){
        if (table[i].inUse)
        {
            if(table[i].isDir){
                OpenFile* tmp = new OpenFile(table[i].sector);
                Directory *directory = new Directory(NumDirEntries);
                directory->FetchFrom(tmp);
                directory->PrintUse();
                delete directory;
                delete tmp;
            }
            else{
                printf("Name: %s, Sector: %d ", table[i].name, table[i].sector);
                hdr->FetchFrom(table[i].sector);
                hdr->PrintUse();
                printf("\n");
            }

        }
    }
    delete hdr;
}
```

在 FileHeader::PrintUse 裡，只要這個 file 的大小大於 one level 的大小，就會去跑迴圈掃過每一個 datasector，並將 header 印出來，之後用遞迴去將其他的 header 也印出來。

```
void FileHeader::PrintUse(){
    int i;
    if (numBytes > OneLevelSize){
        for (i = 0; i < numSectors; i++)
        {
            printf("%d ", dataSectors[i]);
            FileHeader *hdr = new FileHeader;
            hdr->FetchFrom(dataSectors[i]);
            hdr->PrintUse();
        }
    }
}
```

(3) BonusIII

首先在 main.cc 裡，當 recursiveRemoveFlag == true 時，它就會去 call 到 Filesystem 裡的 RecursiveRemove，並把要 remove 的 filename 傳下去，在 Filesystem::RecursiveRemove 裡，它會先 new 一個 PersistentBitmap 的 freeMap，以及拿到 directory header 和 root 的 directory structure，一開始會用 strtok(name, “/”)去 parse 並將它存在 token 裡，接著就是去跑迴圈，迴圈裡會一層一層的找下去直到找到我們要 remove 的 file 或 directory，這裡會用一個 prev token 去記錄上一個 token（也就是上一層的資料夾），因為要有上一個 token，我們才可以去把最後跑到的 token remove 掉，當找到我們要找的 directory 或 file 時會 break 出迴圈，並且若我們找的的是 file 的話，TargetisFile 會等於 1，當 break 出迴圈後，若 TargetisFile == 1，我們就直接將剛剛傳入絕對路徑的 filename remove 掉，若 TargetisFile == 0，就代表它是 directory，因此我們要用遞迴去

remove 掉它下面所有的檔案。

首先我們會先透過 header 拿到當前的 directory structure，接著就會去 call Directory::RecursiveRemove 並將 freeMap 傳入，在 Directory::RecursiveRemove 裡，我們會跑過整個 table，若找到的是 file，就釋放出 freeMap 的空間後 Remove 掉檔案，若找到的是 directory，就去拿到這一層的 directory structure 再繼續一層一層遞迴找下去，return 回來時也要將 structure 寫回 disk，直到將所有檔案刪光，return 回 filesystem。

Return 回去後，會先將當前 structure 寫回 disk，再將之前的找到的 token（要刪掉檔案中的 root）Remove，將 structure 再寫回 disk 並將新的 freeMap 也寫回 disk，這樣就完成 RecursiveRemove 了，下面附上範例結果：