

OS MP4 Report

Team 27 : 112062232賴允中、111070070林標松

分工表

組員	Trace Code	Report編寫	Implementation
賴允中	Y	Y	Y
林標松	Y	Y	Y

Table of Contents

Part I.	2
Part II.	9
Part III.	18
Bonus I.	32
Bonus II.	33
Bonus III.	34

Part I.

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

我們從整個 file system 的起點開始看。

- filesystem/filesys.cc

```
// The file system consists of several data structures:  
//   A bitmap of free disk sectors (cf. bitmap.h)  
//   A directory of file names and file headers  
//  
// Both the bitmap and the directory are represented as normal  
// files. Their file headers are located in specific sectors  
// (sector 0 and sector 1), so that the file system can find them  
// on bootup.
```

```
// Sectors containing the file headers for the bitmap of free sectors,  
// and the directory of files. These file headers are placed in well-known  
// sectors, so that they can be located on boot-up.  
#define FreeMapSector 0  
#define DirectorySector 1
```

從這裡我們可以知道，NachOS 是使用 bitmap 來管理和尋找空白的磁區。

NachOS 將其 bitmap 的 file header 存放於 **sector 0**，這樣開機的時候就可以馬上被讀取。

我們可以透過觀察 file system 在 formatting 時是如何分配空間給 bitmap 和 directory，了解整個管理空白磁區的流程。

- filesystem/filesys.cc ⇒ FileSystem::FileSystem()

```
FileSystem::FileSystem(bool format)  
{  
    DEBUG(dbgFile, "Initializing the file system.");  
    if (format)  
    {  
        PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);  
        Directory *directory = new Directory(NumDirEntries);  
        FileHeader *mapHdr = new FileHeader;  
        FileHeader *dirHdr = new FileHeader;
```

format 變數指示了是否要初始化這顆硬碟。如果 format 為真，則會建立新的 bitmap 和 directory，以及它們對應的 file header。（在 NachOS 中，這兩個物件也都是以檔案的形式儲存）

```
// First, allocate space for FileHeaders for the directory and bitmap
// (make sure no one else grabs these!)
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));
```

先將 bitmap 上 0 和 1 的位置標記為已使用（分配給 bitmap 和 directory 的 file header），然後使用 `Allocate()` 為 bitmap 和 directory 分配相對應的空間。

- `filesys/filehdr.cc` ⇒ `FileHeader::Allocate()`

```
bool FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
{
    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;
}
```

此函數可為新檔案的 file header 在 bitmap 上分配空間。首先根據 fileSize 計算需要的 sector 數量，判斷是否有足夠空間被分配。如果有，就會用迴圈循序在 bitmap 上透過 [FindAndSet\(\)](#) 尋找空白的 sector 並更新。

- lib(bitmap.cc ⇒ Bitmap::FindAndSet())

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

用迴圈遍歷整個 bitmap，將第一個找到為空白的 sector 更改為已使用並回傳其 index，沒找到就回傳 -1。

以上就是 NachOS 管理和尋找空白磁區的方式。

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

- **machine/disk.cc**

```
// 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).

const int MagicNumber = 0x456789ab;
const int MagicSize = sizeof(int);
const int DiskSize = (MagicSize + (NumSectors * SectorSize));
```

Disk class 是 NachOS 用以模擬實體硬碟操作的架構。在最上面，它明確的指出磁碟大小是如何計算的。(`sizeof(int) = 4 bytes`)

- **machine/disk.h**

```
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
```

這裡明確的定義了上面需要的幾個變數。

所以 $DiskSize = (4 + (32 \times 32) \times 128) = 131076$ bytes，扣掉 `MagicNumber` 要佔掉的空間 4 bytes，為 131072 bytes，也就是 128 KB。

故現階段 NachOS 的磁碟最大空間為 128 KB。

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

在第一題的最前面，我們可以看到 directory 也是一個檔案，而它的 file header 被存放於 sector 1。

- filesystem/directory.cc ⇒ Directory::Directory()

```
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;
}
```

Directory class 囊括了 NachOS 用來管理 directory 的各個物件和方法。

當一個 directory 被初始化時，會建立一個 table 用以儲存各個檔案的資訊
(現階段 NachOS 的設計為 single-level directory，也就是每一個 entry 對應
到一個檔案)。

- filesystem/directory.h

```
class DirectoryEntry
{
public:
    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'
};
```

可以看出 directory 中的每個 entry 記錄了三種資訊：

inUse 指示是否正在使用此 entry，sector 紀錄這個檔案的 file header 在哪，
name[] 則代表檔案名稱。

回到 directory 建構子，建構完 table 後把其分配到的記憶體先清空，最後把
每一個 entry 的 inUse 都設為 false，就完成了 directory 的初步建置。

以上即為 NachOS directory 資料結構的規劃方式。

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

- **filesys/filehdr.h**

```
// The following class defines the Nachos "file header" (in UNIX terms,  
// the "i-node"), describing where on disk to find all of the data in the file.  
// The file header is organized as a simple table of pointers to  
// data blocks.
```

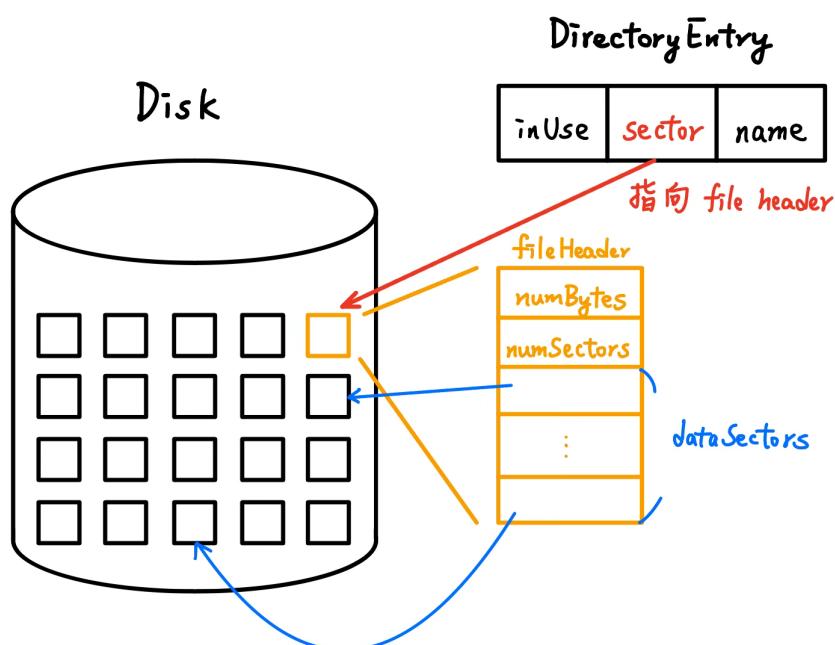
對 NachOS 來說，其 file header 就相當於 inode。

FileHeader class 中的 private 區塊包含以下三種資訊：

```
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
```

其中 `numBytes` 代表檔案的大小，`numSectors` 代表檔案佔用的磁區數量，`dataSectors[]` 紀錄了檔案的每個區塊被儲存在哪個磁區。

透過以上資訊，我們可以得知 NachOS 採用的磁碟分配策略為 **index allocation**。將其運作圖像化大概如下：



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

- **filesys/filehdr.h**

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

由於 NachOS 採用 index allocation scheme，所以一個檔案的最大容量取決於其 file header 中 dataSectors 的 entry 數量。

上一題中，我們知道一個 file header 最前面會有兩個 int 用來儲存 numBytes 和 numSectors，所以剩下留給 NumDirect 的空間就是 1 個 sector 的大小扣掉 2 個 int 的大小，最後除以 sizeof(int) 就得到可用 entry 數。

而 SectorSize 在第二題中已經知道是 128 bytes，

所以 $\text{MaxFileSize} = ((128 - 2 \times 4) \div 4) \times 128 = 3840 \text{ bytes}$ 。

Part II.

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

- userprog/ksyscall.h

```
// MP4
int SysCreate(char *filename, int size) {
    return kernel->fileSystem->Create(filename, size);
}

OpenFileId SysOpen(char *filename) {
    // Since only at most 1 file can be opened at a time, we can just return 1 as file id
    return (kernel->fileSystem->Open(filename) == NULL) ? -1 : 1;
}

int SysRead(char *buffer, int size, OpenFileId id) {
    return kernel->fileSystem->Read(buffer, size, id);
}

int SysWrite(char *buffer, int size, OpenFileId id) {
    return kernel->fileSystem->Write(buffer, size, id);
}

int SysClose(OpenFileId id) {
    return kernel->fileSystem->Close(id);
}
```

和之前 MP1 的做法一樣，由於 **ksyscall.h** 負責連結 system call 到 kernel 底下的各個組件做對應的操作，這裡我們要連結到 **fileSystem**。

除了 **SysOpen()** 的設計變得更簡單以外（因為本次作業已註明一次只會開啟一個檔案，故不需要處理 open file table，file ID 回傳 1 即可），其他的都和 MP1 做過的一樣。

- userprog/exception.cc

和 MP1 一樣，我們要在 [ExceptionHandler\(\)](#) 加上會用到的 syscall cases。

```
case SC_Create:
    val = kernel->machine->ReadRegister(4);
    size = kernel->machine->ReadRegister(5);
    {
        char *filename = &(kernel->machine->mainMemory[val]);
        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;
case SC_Open:
    val = kernel->machine->ReadRegister(4);
    {
        char *filename = &(kernel->machine->mainMemory[val]);
        fileID = SysOpen(filename);
        kernel->machine->WriteRegister(2, fileID);
    }
    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;
case SC_Read:
    val = kernel->machine->ReadRegister(4);
    size = kernel->machine->ReadRegister(5);
    fileID = kernel->machine->ReadRegister(6);
    {
        char *buffer = &(kernel->machine->mainMemory[val]);
        int result = SysRead(buffer, size, fileID);
        kernel->machine->WriteRegister(2, result);
    }
    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;
case SC_Write:
    val = kernel->machine->ReadRegister(4);
    size = kernel->machine->ReadRegister(5);
    fileID = kernel->machine->ReadRegister(6);
    {
        char *buffer = &(kernel->machine->mainMemory[val]);
        int result = SysWrite(buffer, size, fileID);
        kernel->machine->WriteRegister(2, result);
    }
    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;
case SC_Close:
    fileID = kernel->machine->ReadRegister(4);
    {
        int result = SysClose(fileID);
        kernel->machine->WriteRegister(2, result);
    }
    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;
```

- filesys/filesys.h

```
bool Create(char *name, int initialSize); // Create a file (UNIX creat)

OpenFile *Open(char *name); // Open a file (UNIX open)

int Read(char *buffer, int size, OpenFileId id);

int Write(char *buffer, int size, OpenFileId id);

int Close(OpenFileId id);
```

```
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

    OpenFile *openedFile; // The file that is currently opened
};
```

因應同時只會開啟一個檔案的設計，將 MP1 設計的 openFileTable 換成單一的 openedFile。等等的 `Open()` 也要做一點修改。

- filesystem/filesys.cc

Create() 和 Open() 原本就已經實作得差不多了。不過 Open() 中，要讓剛剛新加入的 openedFile 物件連結到開啟的檔案。

```
OpenFile * FileSystem::Open(char *name)
{
    Directory *directory = new Directory(NumDirEntries);
    OpenFile *openFile = NULL;
    int sector;

    DEBUG(dbgFile, "Opening file" << name);
    directory->FetchFrom(directoryFile);
    sector = directory->Find(name);
    if (sector >= 0)
        openFile = new OpenFile(sector); // name was found in directory
    delete directory;

    this->openedFile = openFile;

    return openFile; // return NULL if not found
}
```

剩下要寫的部分只有 Read(), Write(), 和 Close()，這些都和 MP1 的一樣。

```
// MP4
int FileSystem::Read(char *buffer, int size, OpenFileId id) {
    if(this->openedFile != NULL) {
        return this->openedFile->Read(buffer, size);
    }
}

int FileSystem::Write(char *buffer, int size, OpenFileId id) {
    if(this->openedFile != NULL) {
        return this->openedFile->Write(buffer, size);
    }
}

int FileSystem::Close(OpenFileId id) {
    if(this->openedFile != NULL) {
        delete this->openedFile;
        this->openedFile = NULL;
        return 1;
    }
    return 0;
}
```

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

為了可以將 Bonus I, II 的部分也一起達成，我選擇使用 combined scheme，也就是將 linked scheme 和 multilevel scheme 結合的方法。

我依據檔案大小的不同，來決定要使用幾層的 indirect 結構。

如果檔案的大小一個 FileHeader 就能處理的話，就使用 direct allocation；如果介於 direct 和 single indirect 可以處理的範圍內的話，就使用 single indirect allocation，以此類推。為了實作方便，不同的策略不混用。

為了後續可以分辨是第幾層的 block，在 FileHeader 中加入一個 level 變數，故每個 block 可用於連結 sector 的 entry 剩下 $(128 - 3 \times 4) \div 4 = 29$ 個。

以下分析本策略可以支援的最大檔案大小：

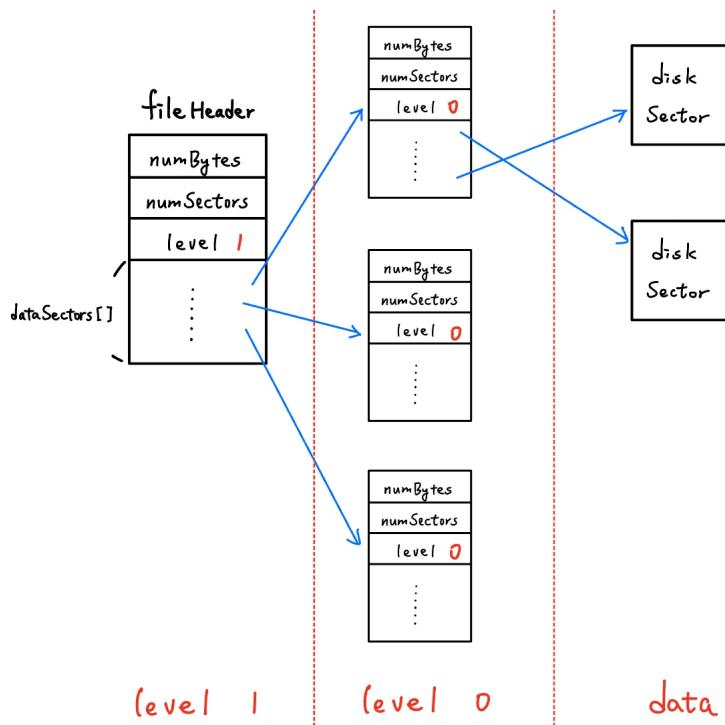
level 0 (direct allocation): $29 \times 128 = 3712$ bytes = 3.625 KB

level 1 (single indirect): $29 \times 3.625 \approx 105$ KB

level 2 (double indirect): $29 \times 105 \approx 3049$ KB ≈ 3 MB

level 3 (triple indirect): $29 \times 3 \approx 87$ MB

故 level 3 以後可以達到 bonus 的要求。



- filesys/filehdr.h

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

我更改了 FileHeader 的結構，多儲存了一個 level 變數，用來記錄目前的 FileHeader 屬於整個 multilevel 的第幾層。level 0 代表 direct，level 1 代表 single indirect，以此類推。

一樣為了實作方便，每一層的 block 我都直接使用 FileHeader 物件來做，並沒有設計新的物件。這樣的小問題就是每個 block 都要重複存一次 numBytes 跟 numSectors，有點浪費空間。

```
#define NumDirect ((SectorSize - 3 * sizeof(int)) / sizeof(int))
#define MaxDirectSize (NumDirect * SectorSize)
#define MaxSingleIndirectSize (NumDirect * NumDirect * SectorSize)
#define MaxDoubleIndirectSize (NumDirect * MaxSingleIndirectSize)
#define MaxTripleIndirectSize (NumDirect * MaxDoubleIndirectSize)
```

四種不同的檔案大小分別使用不同的分層結構，為了方便和易讀性，在這將他們先定義好。

- filesys/filehdr.cc ⇒ FileHeader::Allocate()

依據 fileSize 選擇不同的分層方式。

```
if(fileSize <= MaxDirectSize) {
    // Direct allocation
    level = 0;
    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);
    }
}
```

上面是檔案大小在 direct allocation 可以處理的範圍以內的 case。其實程式完全和原本的一樣，畢竟原本的方法就是 direct allocation。

```
else if(fileSize <= MaxSingleIndirectSize) {
    // Single indirect allocation
    level = 1;
    numSectors = divRoundUp(fileSize, MaxDirectSize);
    if(freeMap->NumClear() < numSectors)
        return FALSE; // not enough space

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

        FileHeader *nextHdr = new FileHeader;
        int nextLevelSize = (remainSize > MaxDirectSize) ? MaxDirectSize : remainSize;
        remainSize -= nextLevelSize;

        if(!nextHdr->Allocate(freeMap, nextLevelSize))
            return FALSE;
        else
            nextHdr->WriteBack(dataSectors[i]);

        delete nextHdr;
        if(remainSize <= 0) break;
    }
}
```

這是檔案大小介於 direct 和 single indirect 可以處理的範圍之間的 case。

首先計算 numSectors 的地方，這裡計算的是總共需要幾個 level 1 blocks，所以除數是 MaxDirectSize。接下來到紅框之前都和 base case 一樣。

紅框部分的邏輯是將 level 1 block 連接到 level 0，所以建立一個新的 FileHeader 物件 nextHdr，然後計算這個 level 0 block 需要分配多少空間。每連接一個 level 0 block，就可以把最多 MaxDirectSize 大小的資料放到 disk 上，然後把 remainSize 減掉我們準備放到硬碟上的大小。

綠框部分的邏輯是遞迴呼叫 `Allocate()`，為下一個 level 的 blocks 也在 bitmap 上分配空間，如果成功了就將這個新的 FileHeader 資訊寫到硬碟上。for 迴圈跑完後，也就把這個 level 1 的 block 底下的所有連結都處理好了。

剩下的 double indirect 和 triple indirect 邏輯和這裡都相同，就不贅述。

- **filesys/filehdr.cc** ⇒ **FileHeader::Deallocate()**

```
if(level == 0) {
    // Direct deallocation
    for(int i = 0; i < numSectors; i++) {
        ASSERT(freeMap->Test((int)dataSectors[i])); // ought to be marked!
        freeMap->Clear((int)dataSectors[i]);
    }
}
```

如果是 level 0 的 block，就遍歷整個 FileHeader 的 dataSector 陣列，將其所有欄位全部清空，也就是和原本一樣的做法。

```
else {
    // Indirect deallocation
    for(int i = 0; i < numSectors; i++) {
        FileHeader *nextHdr = new FileHeader;
        // Recursively deallocate the next level
        nextHdr->FetchFrom(dataSectors[i]);
        nextHdr->Deallocate(freeMap);
        delete nextHdr;
        ASSERT(freeMap->Test((int)dataSectors[i])); // ought to be marked!
        freeMap->Clear((int)dataSectors[i]);
    }
}
```

如果是非 level 0 的 block，就用遞迴 DFS 的方式處理。這裡跟上面的差別，在於非 level 0 的 block 下一個連接的必定也是 FileHeader，所以我們要先用 [FetchFrom\(\)](#) 將其以 FileHeader 的形式讀出，再對其做 [Deallocate\(\)](#)。

- filesys/filehdr.cc ⇒ FileHeader::ByteToSector()

此函數可將檔案中的某個 byte 轉換為其於硬碟上 sector 的位置。其實就跟之前的虛擬地址轉換到實體地址是一樣的邏輯。

```
if(level == 0) {  
    return dataSectors[offset / SectorSize];  
}
```

一樣根據 level 分成不同的 case 處理。如果是 level 0，就用跟預設一樣的換算方式即可。

```
else if(level == 1) {  
    int idx = offset / MaxDirectSize;  
    FileHeader *nextHdr = new FileHeader;  
    nextHdr->FetchFrom(dataSectors[idx]);  
    int ret = nextHdr->ByteToSector(offset % MaxDirectSize);  
    delete nextHdr;  
    return ret;  
}
```

如果不是 level 0，就要透過遞迴做多層的轉換。和上一個函數一樣，由於非 level 0 的 block 下一個連接的必定也是 FileHeader，所以我們要先用 [FetchFrom\(\)](#) 將其以 FileHeader 的形式讀出，再對其套用 [ByteToSector\(\)](#)。

其他 level 的邏輯也完全一樣，只差在紅框的 offset 後面取模的除數不同而已，故不贅述。

Part III.

- filesys/directory.h

- 首先，在 DirectoryEntry 裡新增一個變數 `isDirec`，來判斷這個 entry 存放的資料是 directory 還是 file，以便後面 recursive list 印出需要的資訊。

```
class DirectoryEntry
{
public:
    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'
    // MP4 add
    bool isDirec; //是否是directory
};
```

- 接著，因為 spec 規定一個 directory 裡要支援至多 64 個 file/directory，因此新增預定義 `#define` 變數 `NumDirEntries` 為 64。此外多定義一個變數 `DirectoryNameMaxLen` 為 9，因為 directory 的名字最長不會超過 9，此變數在後面 `Directory::GetDirecSector()` 裡會用到，用來初始化一個暫存 directory 名字的 buffer 陣列。

```
// MP4 add，因spec規定一個directory裡要支援64個file/directory
#define NumDirEntries 64
#define DirectoryNameMaxLen 9
```

- 因為需要初始化 DirectoryEntry 裡的 `isDirec`，所以在 `Add()` 函數裡，多傳入一個參數 `isDirec` 來初始化 entry 裡的 `isDirec` 欄位。除此之外，我還新增兩個函數，一個是 `RecursiveList()`，用來遞迴印出 directory 裡面的 file / directory 資訊；另外一個函數是 `GetDirecSector()`，此函數可以將傳入的絕對路徑拆解，並取得目標 directory 所在的 sector。

```
// MP4 add，增加傳入isDir變數來初始化directory entry
bool Add(char *name, int newSector, bool isDirec); // Add a file name into the directory
```

```
// MP4 add
void RecursiveList(int level); // recursuve地印出所有directory裡內容
int GetDirecSector(char* direcPath); //回傳絕對路徑最後的directory所在的sector
```

- filesys/directory.cc

- 在 `Add()` 函數裡，根據傳入的 `isDirec` 布林值，初始化 `DirectoryEntry`。

```
for (int i = 0; i < tableSize; i++)  
    if (!table[i].inUse)  
    {  
        table[i].inUse = TRUE;  
        strncpy(table[i].name, name, FileNameMaxLen);  
        table[i].sector = newSector;  
        // MP4 add  
        table[i].isDirec = isDirec; // 初始化isDirec  
        return TRUE;  
    }
```

- 實作 `RecursiveList()`，以新增 ”lr” 指令的功能。此函數會先去遍歷第 0 層 `directory` (即自己)，如果裡面還有 `directory` 存在 (`isDirec == true`)，則會對該 `directory` 遞迴呼叫 `RecursiveList()`，並傳入 `level + 1`，代表進入下一層。在印出每個 `entry` 以前，會先用 `level` 判斷現在是第幾層，並加上對應的縮排空格，再進行輸出。

```
void Directory::RecursiveList(int level)  
{  
    Directory* subDirectory = new Directory(NumDirEntries);  
    // 開始遍歷directory裡的entry  
    for(int i = 0; i < tableSize; i++)  
    {  
        if(table[i].inUse)  
        {  
            for(int j = 0; j < level; j++) printf("    "); // 依照spec規定空四格  
            printf("%c %s\n", (table[i].isDirec == true) ? 'D' : 'F', table[i].name);  
            if(table[i].isDirec)  
            {  
                // 從file header所在的sector讀進directory資料  
                OpenFile *file = new OpenFile(table[i].sector);  
                subDirectory->FetchFrom(file);  
                subDirectory->RecursiveList(level + 1); //遞迴呼叫  
                delete file;  
            }  
        }  
    }  
    delete subDirectory;  
}
```

在這邊可以看到，如果 `entry` 裡的資料形式為 `directory`，要先透過 `FetchFrom()` 將其以 `directory` 的形式讀出，再繼續遞迴遍歷。

- 實作 `GetDirecSector()`，此函數負責解析傳入的絕對路徑，並回傳路徑中最後的 directory 所在的 sector。舉個例子，例如傳入的路徑是 `"/t0/t1/t2"`，會先解析出 t0，接著去 root directory 找出 t0 所在 entry 裡的 sector 資料（呼叫 `Directory::Find()` 得到 t0 的 file header 所在的 sector），接著因為還沒碰到結尾 `\0`，所以建立 t0 的 directory，遞迴呼叫 `GetDirecSector()`，傳入 `"/t0"` 後面的字串 `"/t1/t2"` 繼續遞迴呼叫，直到找到字串結尾，回傳目標 directory 的 file header 所在的 sector (回傳 `sector_we_want`)。

```

int Directory::GetDirecSector(char* direcPath)
{
    if(!strcmp("/", direcPath)) return 1; //代表是根目錄，root directory存在sector 1
    int index = 1; // 從第二個字('/'的下一個)開始找directory
    while(direcPath[index] != '/' && direcPath[index] != '\0') index++;

    char dir[DirectoryNameMaxLen + 1];
    strncpy(dir, direcPath + 1, index - 1); //只需複製directory名字就好
    dir[index - 1] = '\0'; //在結尾補上'\0'

    int direc_sector = Find(dir); //去找去找dir的fileHeader所在的sector
    int sector_we_want = -1;
    if(direcPath[index] != '\0') //代表後面還有路徑要找
    {
        Directory* directory = new Directory(NumDirEntries);
        OpenFile* file = new OpenFile(direc_sector);
        directory->FetchFrom(file);
        sector_we_want = directory->GetDirecSector(direcPath + index); //遞迴呼叫，繼續往下找
        delete directory;
        delete file;
    }
    else sector_we_want = direc_sector;

    return sector_we_want;
}

```

- filesystem/filesys.h

- 接著，在 FileSystem 中增加 3 個函數，第一個是 `CreateDirectory()`，用來給 kernel 呼叫並建立 directory。第二個是 `RecursiveList()`，此函數將目標 directory 裡的所有 file 和 subdirectory 印出來。第三個是 `SplitPath()`，用來將絕對路徑做拆解，切成 directory path 和 file/directory name 兩部分，例如：“/t0/t1/t2”，會拆解成 directory path “/t0/t1” 和 file name “t2”，代表 t2 檔案或 directory 要在根目錄底下的 t0 目錄底下的 t1 目錄裡。

```
// MP4 add
bool CreateDirectory(char* name);
void RecursiveList(char* direcPath);
```

```
// MP4 add
void SplitPath(char* name, char* dirPath, char* fileName); //拆解絕對路徑
```

- filesystem/filesys.cc

- 先修改原本的 `Create()` 函數。在 `Create()` 中，因為傳入的 name 不一定是在 root directory 下，所以需要對傳入的絕對路徑進行解析。呼叫我定義的 `SplitPath()` 來將 name 切成 directory path 和 file name，並且將拆解完的 directory path 在 root directory 中呼叫上面定義的 `Directory::GetDirecSector()` 來找到目標 directory 的 file header 所在的 sector，並且將此目標 directory 資料抓進來，再依照原先步驟，在此目標 directory 下建立新的檔案。

```
bool FileSystem::Create(char *name, int initialSize)
{
    Directory *directory;
    PersistentBitmap *freeMap;
    FileHeader *hdr;
    int sector;
    bool success;
    // MP4 add
    OpenFile* file; //用來開啟最終directory的file header
    Directory* root; //存根目錄
    int direcSector; //最終directory的fileHeader所在的sector
    char dirPath[256], fileName[10]; //size多加一，來存'\0'在最後

    SplitPath(name, dirPath, fileName); //進行拆解動作

    DEBUG(dbgFile, "Creating file " << fileName << " size " << initialSize);
```

除了原本定義的一些變數，我增加了五個變數來使後面找 directory 及開啟目錄的動作能順利進行。第一個是 OpenFile 的變數 file，會用來儲存目標 directory 的 file header 開啟物件。第二個是一個 Directory 的變數 root，會用來儲存根目錄，來進行找尋目標 directory 的動作。第三個是 direcSector 變數，會用來儲存找到目標 directory 的 file header 所在的 sector。第四和五個是兩個 char 陣列，用來儲存拆解絕對路徑後的 directory path 以及 file name。

接著，會呼叫我們定義的 [SplitPath\(\)](#)，進行絕對路徑的拆解動作。（此函數實作稍後會提到）

```
// MP4 add
root = new Directory(NumDirEntries);
directory = new Directory(NumDirEntries);
root->FetchFrom(directoryFile); //將root directory讀進來
direcSector = root->GetDirecSector(dirPath); //找到最終directory的sector
file = new OpenFile(direcSector); //開啟最終directory檔案
directory->FetchFrom(file); //把directory抓進來
```

拆解完絕對路徑後，就可以利用拆解得到的 directory path 來找尋目標 directory 所在的 sector。首先，先將 root directory 資料抓進來並開啟，利用 root directory 來呼叫 [GetDirecSector\(\)](#)，將 directory path 傳入來進行找尋目標 directory，此函數會回傳找到的目標 directory 的 sector。接著，我們就能利用目標 directory 的 file header 所在的 sector (direcSector) 來開啟檔案，new一個 OpenFile 物件。最後，將目標 directory 資料抓進來，呼叫 [Directory::FetchFrom\(\)](#)。

以下繼續進行講解....

```

    if (directory->Find(fileName) != -1)
        success = FALSE; // 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 = FALSE; // no free block for file header
        else if (!directory->Add(fileName, sector, FALSE)) //此為file，isDirec是FALSE
            success = FALSE; // no space in directory
        else
        {
            hdr = new FileHeader;
            if (!hdr->Allocate(freeMap, initialSize))
                success = FALSE; // no space on disk for data
            else
            {
                success = TRUE;
                // everthing worked, flush all changes back to disk
                hdr->WriteBack(sector);
                directory->WriteBack(file); //更新directory
                freeMap->WriteBack(freeMapFile);
            }
            delete hdr;
        }
        delete freeMap;
    }
    delete directory;
    delete root;
    delete file;
    return success;
}

```

找到並開啟目標 directory 後，就能進行創建檔案的動作了。和原本的步驟大致上相同，只差在 `Directory::Add()` 函數要多傳入 `isDirec` 參數來初始化 directory entry，而因為這裡是在創建「檔案」，因此 `isDirec` 為 FALSE。最後釋放所有資源，包括我們新增加的指標變數。

- 實作 `CreateDirectory()` 函數，和上面講述的 `Create()` 大致相同。先解析絕對路徑得到 directory path 和 file name，得到目標 directory 並抓取其資料進來，接著在目標 directory 下創建一個 directory，directory 一樣也是檔案，只差在 `isDirec` 為 TRUE，因此在 `Add()` 裡傳入的 `isDirec` 為 TRUE，代表這是一個 directory，並透過 `Directory::WriteBack()` 將新創建初始化完的 directory 也寫回 disk。

```

bool FileSystem::.CreateDirectory(char* name)
{
    Directory *directory;
    PersistentBitmap *freeMap;
    FileHeader *hdr;
    int sector;
    bool success;
    OpenFile* file; //用來開啟最終directory的file header
    Directory* root; //存根目錄
    int direcSector; //最終directory的fileHeader所在的sector
    char dirPath[256], fileName[10]; //size多加一，來存'\0'在最後

    SplitPath(name, dirPath, fileName); //進行拆解動作

    DEBUG(dbgFile, "Creating file " <> fileName <> " size " <> DirectoryFileSize);

    root = new Directory(NumDirEntries);
    directory = new Directory(NumDirEntries);
    root->FetchFrom(directoryFile); //將root directory讀進來
    direcSector = root->GetDirecSector(dirPath); //找到最終directory的sector
    file = new OpenFile(direcSector); //開啟最終directory檔案
    directory->FetchFrom(file); //把directory抓進來
}

```

```

if (directory->Find(fileName) != -1)
    success = FALSE; // 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 = FALSE; // no free block for file header
    else if (!directory->Add(fileName, sector, TRUE)) //此為directory · isDirec是TRUE
        success = FALSE; // no space in directory
    else
    {
        hdr = new FileHeader;
        if (!hdr->Allocate(freeMap, DirectoryFileSize))
            success = FALSE; // no space on disk for data
        else
        {
            success = TRUE;
            // everthing worked, flush all changes back to disk
            hdr->WriteBack(sector);
            directory->WriteBack(file); //更新directory
            freeMap->WriteBack(freeMapFile);
            // 要將初始化完的directory也寫回disk
            OpenFile* f = new OpenFile(sector);
            Directory* d = new Directory(NumDirEntries);
            d->WriteBack(f);
        }
        delete hdr;
    }
    delete freeMap;
}
delete directory;
delete root;
delete file;
return success;
}

```

- 實作 `RecursiveList()` 函數。先 fetch root directory 的資料，再對其呼叫 `GetDirecSector()` 得到目標 directory 的 file header 所在的 sector。因為只要求列出 directory，所以不用解析絕對路徑，傳入的路徑已經是 directory path。找到目標 directory 後，一樣 fetch 其資料，並呼叫 `Directory::RecursiveList()` 遞迴印出其中的 file 和 subdirectory。傳入的 level 為 0，代表現在在第 0 層，即自己，從自己開始遍歷。

```
void FileSystem::RecursiveList(char* direcPath)
{
    Directory* root = new Directory(NumDirEntries);
    Directory* directoryToBeList = new Directory(NumDirEntries);
    root->FetchFrom(directoryFile);
    int sector = root->GetDirecSector(direcPath); //找到目標directory所在的sector
    OpenFile* file = new OpenFile(sector);
    directoryToBeList->FetchFrom(file);
    directoryToBeList->RecursiveList(0); //從第0層(自己)開始遍歷去list
    delete root;
    delete directoryToBeList;
    delete file;
}
```

- 實作 `SplitPath()` 函數。此函數可將絕對路徑字串進行解析，切成 directory path 以及 file name 兩個字串，並複製到 dirPath 和 fileName 裡。我的做法是從字串的最後面開始找，找到第一個 “/”，就代表找到 file name 了，剩下前面的都是 directory 的相對路徑。如果找到的第一個 “/” 是在 index 0 的位置，代表此檔案要在根目錄下面，例如：“/t0”，即代表 t0 檔案要在根目錄下，因此 directory path 要手動加上，使 directory path 為 “/”。此外，我們都需在字串結尾手動加上 ”\0”。

```
void FileSystem::SplitPath(char* name, char* dirPath, char* fileName)
{
    int len = strlen(name);
    int index;
    for(index = len - 1; index >= 0; index--)
    {
        if(name[index] == '/') break; //找到最後面的檔名了
    }
    strncpy(dirPath, name, index);
    strncpy(fileName, name + index + 1, len - index - 1);
    dirPath[index] = fileName[len - index - 1] = '\0'; //結尾補上'\0'
    if(index == 0) //如果只有一層根目錄
    {
        dirPath[0] = '/';
        dirPath[1] = '\0';
    }
}
```

- 修改 [List\(\)](#) 函數，和 [RecursiveList\(\)](#) 大致相同，都是先找到目標 directory 並抓取資料進來，只是差在不用遞迴呼叫，只需呼叫目標 directory 的 [Directory::List\(\)](#) 函數就好。

```
void FileSystem::List(char* direcPath)
{
    // MP4 add
    Directory* root = new Directory(NumDirEntries);
    Directory* directoryToBeList = new Directory(NumDirEntries);

    root->FetchFrom(directoryFile);
    int sector = root->GetDirecSector(direcPath); //找到目標directory所在的sector
    OpenFile* file = new OpenFile(sector);
    directoryToBeList->FetchFrom(file);
    directoryToBeList->List(); //開始遍歷去list
    delete root;
    delete directoryToBeList;
    delete file;
}
```

- threads/main.cc

- 實作完底層的 directory 以及 file system 後，就可以在 [main\(\)](#) 裡依據 “-l” 和 “-lr”，呼叫FileSystem裡剛剛定義的 [List\(\)](#) 以及 [RecursiveList\(\)](#)，依據 recursiveListFlag 是否為TRUE，來決定要呼叫哪種 list 方式。

```
if (dirListFlag)
{
    if(recursiveListFlag) kernel->fileSystem->RecursiveList(listDirectoryName);
    else kernel->fileSystem->List(listDirectoryName);
}
```

- 接著，若輸入 “-mkdir” 指令，代表要創建 directory，mkdirFlag 會被設為 TRUE，呼叫 main 裡的 [.CreateDirectory\(\)](#)，而我們需要在 [.CreateDirectory\(\)](#) 中呼叫 FileSystem 的 [.CreateDirectory\(\)](#)，並將絕對路徑傳入。

```
if (mkdirFlag)
{
    // MP4 mod tag
    CreateDirectory(createDirectoryName);
}
```

```
static void CreateDirectory(char *name)
{
    // MP4 Assignment
    kernel->fileSystem->CreateDirectory(name); //呼叫file system
}
```

- filesystem/filesys.cc 以及 filesystem/directory.h

- 因為 spec 規定要使一個 directory 支援 64 個 file/subdirectory，因此除了上面在 directory.h 裡有先定義 NumDirEntries 為 64 (可見p.18)，在 filesys.cc 裡的 NumDirEntries 也需改為 64。

```
#define FreeMapFileSize (NumSectors / BitsInByte)
#define NumDirEntries 64 // MP4 add, 因spec規定一個directory裡要支援64個file/directory
#define DirectoryFileSize (sizeof(DirectoryEntry) * NumDirEntries)
```

完成 Part 3 要求的實作後，還需要修改 “-r” 以及 “-p” 指令，以支援傳入絕對路徑，因為現在傳入的路徑有可能是多層的，例如：“/t0/t1”。

- 先修改 “-r” 指令，trace 後可以發現在 main() 裡面，因為 removeFileName 不為 NULL，所以會去呼叫 FileSystem 的 Remove() 函數，並將要刪除的檔案路徑傳入。

- threads/main.cc

- filesystem/filesys.cc

```
if (removeFileName != NULL)
{
    kernel->fileSystem->Remove(removeFileName);
}
```

修改 Remove()，使其可以拆解絕對路徑。做法和上面的 Create() 一樣，先呼叫 SplitPath() 來將絕對路徑拆成 directory path 和 file name，接著利用 root

directory 來呼叫 `GetDirecSector()`，取得要刪除的檔案所在的 directory 的 sector，並 fetch 其資料。在此目標 directory 下，呼叫 `Find()` 取得要刪除檔案的 file header 所在的 sector。利用刪除檔案的 sector 來把其 file header 抓進來，並執行刪除的動作，釋放其 data block 以及其 file header 的空間，並從 directory 裡移除，最後將更新後的 directory 以及 freeMap 寫回 disk 裡。

```
bool FileSystem::Remove(char *name)
{
    Directory *directory;
    PersistentBitmap *freeMap;
    FileHeader *fileHdr;
    int sector;
    // MP4 add
    Directory* root; //存根目錄
    OpenFile* file; //用來開啟檔案所在的directory
    int direcSector; //檔案所在的directory的fileHeader所在的sector
    char dirPath[256], fileName[10]; //size多加一，來存'\0'在最後

    SplitPath(name, dirPath, fileName); //進行拆解動作

    root = new Directory(NumDirEntries);
    directory = new Directory(NumDirEntries);
    root->FetchFrom(directoryFile);
    direcSector = root->GetDirecSector(dirPath);
    file = new OpenFile(direcSector);
    directory->FetchFrom(file);

    sector = directory->Find(fileName); //找到檔案所在的sector
```

這邊在進行找尋欲刪除檔案所在的 directory，以及在此 directory 下找到欲刪除檔案的 file header 所在的 sector。

以下繼續講解....

```

if (sector == -1)
{
    delete directory;
    return FALSE; // file not found
}

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

freeMap = new PersistentBitmap(freeMapFile, NumSectors);

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

freeMap->WriteBack(freeMapFile); // flush to disk
directory->WriteBack(file); // flush to disk
delete fileHdr;
delete directory;
delete freeMap;
return TRUE;
}

```

這裡的釋放空間以及移除檔案的動作，和原本的 code 一樣，只差在 `directory->Remove()` 傳入的是拆解完的 file name，以及 `directory->WriteBack()` 是傳入欲刪除檔案所在的 directory 的開啟檔案，不是根目錄，因為檔案可能在不同層 directory 裡。這樣就完成修改了。

- 再來修改 “-p” 指令，trace 後可以發現在 `main()` 裡面，因為 `PrintFileName` 不為 NULL，所以會去呼叫 `main.cc` 裡的 `Print()`，並將 `PrintFileName` 傳入。

```

if (printFileName != NULL)
{
    Print(printFileName);
}

```

- threads/main.cc

```
void Print(char *name)
{
    OpenFile *openFile;
    int i, amountRead;
    char *buffer;

    if (openFile = kernel->fileSystem->Open(name)) == NULL)
    {
        printf("Print: unable to open file %s\n", name);
        return;
    }
```

在 Print() 裡，會呼叫 FileSystem 的 Open() 開啟要印出的檔案。

- filesys/filesys.cc

```
OpenFile * FileSystem::Open(char *name)
{
    Directory *directory = new Directory(NumDirEntries);
    OpenFile *openFile = NULL;
    int sector;

    DEBUG(dbgFile, "Opening file" << name);
    directory->FetchFrom(directoryFile);
    // MP4 add
    sector = directory->GetDirecSector(name); // 傳入的路徑有可能是多層的
    if (sector >= 0)
        openFile = new OpenFile(sector); // name was found in directory
    delete directory;

    this->openedFile = openFile;

    return openFile; // return NULL if not found
}
```

修改 Open()，使其可以解析絕對路徑，並獲取目標檔案的 file header 所在的 sector。這裡比較不同的是，不用呼叫 SplitPath()，而是直接對 root directory 呼叫 GetDirecSector()，來獲取絕對路徑最後面檔案 file header 的 sector，因為我們定義的這個函數可以直接對傳入的路徑做遞迴呼叫，一直解析路徑直到最後面的檔案，並回傳最後面檔案的 file header 所在的 sector（詳見

p.20)。例如：“/t0/t1/t2”，可以直接獲取到 t2 的 file header 所在的sector。
並利用獲取到的 sector 來開啟檔案。

Bonus I.

- machine/disk.h

```
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
```

原本的硬碟大小為 128KB，將 SectorPerTrack 從 32 改成 16384（放大了 512 倍），這樣硬碟大小就變為 $128 \times 512 = 65536\text{ KB} = 64\text{ MB}$ 。

至於最大的檔案大小，在 Part II 也已經解釋過為約 87 MB。

Bonus II.

我們採用的 multi level index 設計，根據檔案大小的不同，會使用的 FileHeader block 的數量也不一樣。

例如，對於一個 fileSize 在 MaxDirectSize 以內的檔案來說，會用到的 FileHeader block 也就只有一個；而對於 fileSize 介於 MaxDirectSize 和 MaxSingleDirectSize 之間的檔案來說，會用到的 FileHeader block 至少大於 3 個（因為 level 1 的 block 有一個，其連接到的 level 0 的 block 至少有 2 個，至多有 29 個）。

這樣的設計符合越小的檔案對應越小的 file header，且超過三種不同的 file header 大小。

Bonus III.

原本的 `Remove()` 只會去把欲刪除的檔案刪掉，若欲刪除檔案室是 `directory`，會需要遞迴將其底下的所有資料都刪除，直到全都刪光光。因此這裡做法和 Part 3 類似，要先從底層 `Directory` 增加一些函數，再到上層的 `FileSystem` 的 `Remove()` 函數裡做修改。

- `filesys/directory.h`

首先，先在 `Directory` 裡新增三個函數，分別是 `IsDirec()`，此函數會判斷傳入欲搜尋的檔案，利用其檔案名稱去搜尋其 `isDirec` 欄位布林值，並回傳此檔案是否為 `directory`。第二個是 `GetTableSize()`，此函數會回傳此 `directory` 的 `table entry` 數量，第三個是 `GetTable()`，此函數會回傳此 `directory` 的 `table`，這兩個函數回傳的資訊可以讓我們在 recursive remove 時去遍歷 `directory` 裡的所有 `entry` 資料。

```
// recursive remove會用到的
bool IsDirec(char* fileName); //看此entry是否是directory
int GetTableSize() {return tableSize;} // 回傳table的entry數量
DirectoryEntry* GetTable() {return table;} //回傳table
```

- `filesys/directory.cc`

實作 `IsDirec()` 函數。利用 `FindIndex()` 來將傳入的欲搜尋的檔案名稱，去看其在此 `directory` 的 `table` 裡的 `index` 是多少，並去其對應的 `entry`，回傳其 `isDirec` 的布林值。

```
bool Directory::IsDirec(char* fileName)
{
    int index = FindIndex(fileName);
    return table[index].isDirec;
}
```

- filesys/filesys.h

實作完底層 Directory 後，回到上層 FileSystem。我在 Remove() 函數裡多傳入一個 bool 變數 recursive，來判斷是否需要執行遞迴 remove。

```
// MP4 add，判斷是否需要遞迴刪除  
bool Remove(char *name, bool recursive); // Delete a file (UNIX unlink)
```

- filesys/filesys.cc

在 Remove() 裡，和上面修改 “-r” 指令相同（詳見p.28）。一開始一樣先對傳入的絕對路徑做解析，找到欲刪除的檔案所在的 directory 以及欲刪除的檔案 (file或是directory)。這裡，多增加一個判斷區塊，如果需要遞迴 remove (recursive == TRUE)，且要刪除的對象是 directory (isDirec == TRUE)，就會先去讀取其 directoryEntry table，遍歷 table 並遞迴呼叫 Remove() 來移除每個 entry。等到遞迴結束後，會繼續執行最後面刪除此 directory 或是 file 的動作，將其 data block 和 file header 的空間釋放掉，並從其所在的 directory 中移除，最後將更新後的 freeMap 和 directory 資訊寫回 disk 裡。

```
bool FileSystem::Remove(char *name, bool recursive)  
{  
    Directory *directory;  
    PersistentBitmap *freeMap;  
    FileHeader *fileHdr;  
    int sector;  
    // MP4 add  
    Directory* root; //存根目錄  
    OpenFile* file; //用來開啟檔案所在的directory  
    int direcSector; //檔案所在的directory的fileHeader所在的sector  
    char dirPath[256], fileName[10]; //size多加一，來存'\0'在最後  
  
    SplitPath(name, dirPath, fileName); //進行拆解動作  
  
    root = new Directory(NumDirEntries);  
    directory = new Directory(NumDirEntries);  
    root->FetchFrom(directoryFile);  
    direcSector = root->GetDirecSector(dirPath);  
    file = new OpenFile(direcSector);  
    directory->FetchFrom(file);  
  
    sector = directory->Find(fileName); //找到檔案所在的sector
```

這裡和 p.28 修改後的 Remove() 相同，先解析絕對路徑，拆成 directory path 和 file name。接著去找到欲刪除檔案所在的 directory 並開啟，再去找欲刪除檔案的 file header 所在的 sector (呼叫 directory->Find(fileName))。

```
if (recursive)
{
    bool isDirec = directory->IsDirec(fileName); // 看看要刪除的entry是不是directory
    if(isDirec)
    {
        //開啟要刪除的directory，並遞迴呼叫Remove()，來刪除其table裡的entry
        OpenFile* f = new OpenFile(sector);
        Directory* d = new Directory(NumDirEntries);
        d->FetchFrom(f);
        int tableSize = d->GetTableSize(); //取得table的size
        DirectoryEntry* table = d->GetTable(); //取得table
        // 遍歷此directory，遞迴呼叫來刪除所有裡面的資料
        for(int i = 0; i < tableSize; i++)
        {
            if(table[i].inUse)
            {
                // 將絕對路徑做增加，來呼叫此檔案的Remove
                char nextName[256];
                strcpy(nextName, name);
                strcat(nextName, "/");
                strcat(nextName, table[i].name);
                Remove(nextName, TRUE); //遞迴呼叫
            }
        }
    }
}
```

這裡進行遞迴的動作，會先呼叫欲刪除檔案所在的 directory 下的 IsDirec() 函數，來判斷欲刪除檔案是否是 directory (directory->IsDirec(fileName))，如果需要進行遞迴 remove，且欲刪除檔案是 directory (recursive == TRUE && isDirec == TRUE)，就會進行遞迴動作。首先，先將欲刪除的 directory 開啟，呼叫 GetTableSize() 以及 GetTable() 來獲取此 directory 的 table 相關資訊。接著，用迴圈遍歷此 directory 底下的所有 entry，如果 entry 裡有資料，就會創建此資料的絕對路徑字串，先將當初傳入的絕對路徑 (name) 複製到 nextName 裡，然後利用 strcat() 將 "/" 以及此 entry 的檔案名稱貼到 nextName 後面，例如：一開始傳入的 name 是 “/t0/t1”，entry 裡的檔案名稱是

t2，此 entry 檔案的絕對路徑 nextName 就會變成 “/t0/t1/t2” 。並遞迴呼叫 Remove()，將此 entry 的絕對路徑傳入，以及 recursive flag 設為 TRUE。

```
// 遞迴結束，刪除目標directory
fileHdr = new FileHeader;
fileHdr->FetchFrom(sector);

freeMap = new PersistentBitmap(freeMapFile, NumSectors);

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

freeMap->WriteBack(freeMapFile); // flush to disk
directory->WriteBack(file); // 將更新完的directory寫回disk
delete fileHdr;
delete directory;
delete root;
delete freeMap;
delete file;
return TRUE;
}
```

當所有底下的 entry 都被刪除，遞迴結束。這裡和原先 Remove() 相同，將當初欲刪除的檔案進行刪除動作，釋放其 data block 以及 file header 的空間，並將其從所在的 directory 底下移除。最後將更新完的 freeMap 以及 directory 寫回 disk 裡。

如此，便完成遞迴 remove 的功能。