Operating System

MP4: File System

Team 73

Trace Code & Implementation: 1121036s 郭逸洪

Contents

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)?

FileSystem建構式中，如果需要format，會建立PersistentBitmap *freeMap物件以記錄sector的使用情形，並建立FileHeader *mapHdr配置freeMap所需空間，以一個bit表示一個block的是否被使用，1024個blocks共1024個bits，正好為一個sector的大小，FileHeader::Allocate配置時呼叫Bitmap::NumClear檢查是否有足夠的sector可用，如果有，在迴圈中使用Bitmap::FindAndSet取得第一個可用的sector，將此block的位置紀錄mapHdr的dataSectors中，配置成功後將mapHdr寫入硬碟上索引0的sector，mapHdr寫入成功後，從硬碟開啟freeMapFile，將freeMap的變更寫入該檔案。

最終，mapHdr存放在sector 0、dirHdr存放在sector 1，而freeMap的data block則在使用mapHdr配置的sector 2之中。

FileSystem建構式中，如果不需要format，則直接從sector 0開啟freeMapFile。

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

FileSystem建構式中建立的freeMap共有1024個sectors（32 sectors/track * 32 tracks），每個sector大小為128 bytes（定義在disk.h），故模擬的硬碟大小為128 KB。

值得注意的是，disk.cc定義的DiskSize是128 KB加上額外的4 bytes，原因是為了避免誤認其他檔案為模擬的硬碟，然而disk.cc中的操作並不會使用這4 bytes，故實際能儲存的空間仍是128 KB。

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

FileSystem建構式中，如果需要format，會建立Directory *directory，其中包含10個DirectoryEntry，每個DirectoryEntry紀錄這個entry是否在使用中、在硬碟上哪個sector與其檔名，隨後建立FileHeader *dirHdr以配置directory所需空間，每個DirectoryEntry大小為20 bytes，directory大小為200 bytes，需要配置2個sectors，配置成功後dirHdr寫入硬碟上索引1的sector，dirHdr寫入成功後，從硬碟開啟directoryFile，將directory的變更寫入該檔案。

最終，dirHdr存放在sector 1，而directory的data block則在使用dirHdr配置的sector 3、4之中。

FileSystem建構式中，如果不需要format，則直接從sector 1開啟directoryFile。

FileSystem的公開方法都與directory管理相關，以下是各個公開方法如何使用操作directory的細節：

a. bool FileSystem::Create(char *name, int initialSize)
    - 從directoryFile讀取directory物件
    - 如果該directory有同名的檔案，創建檔案失敗
    - 從freeMapFile讀取freeMap物件
    - 嘗試從freeMap取得第一個可用的sector存放file header，若沒有可用的sector，則創建檔案失敗
    - 嘗試將檔名及file header的sector位置資訊寫入directory，若沒有可用的空間（預設一個directory只能放10個檔案），創建檔案失敗
    - 建立FileHeader物件，嘗試從freeMap配置檔案所需空間，如果沒有足夠的空間，創建檔案失敗
    - 如果上述操作都成功，將file header、directory、freeMap寫回硬碟，回傳檔案創建成功
b. OpenFile *FileSystem::Open(char *name)
    - 從directoryFile讀取directory物件

- 尋找該檔案的header在directory下哪個sector，若找不到，回傳NULL，否則使用找到的sector開啟檔案並回傳

c. bool FileSystem::Remove(char *name)

- 從directoryFile讀取directory物件
- 尋找該檔案的header在directory下哪個sector，若找不到，回傳檔案刪除失敗
- 建立fileHdr及freeMap物件，將header及data blocks歸還freeMap，從directory移除該檔案
- 將directory、freeMap寫回硬碟，回傳檔案刪除成功

d. void FileSystem::List()

- 從directoryFile讀取directory物件
- 迴圈中列出directory每一個使用中的DirectoryEntry的檔名

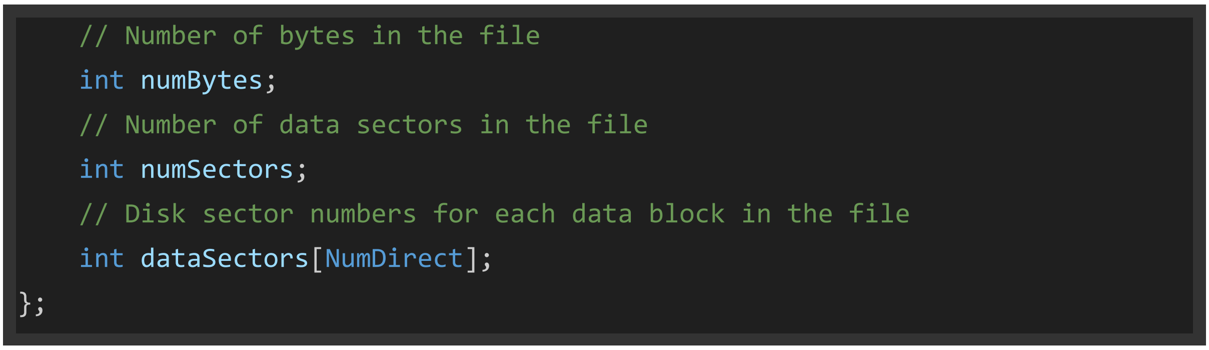
e. void FileSystem::Print()

- 從directoryFile與freeMapFile讀取directory和freeMap物件
- 從sector 0讀取freeMap的file header並印出
- 從sector 1讀取directory的file header並印出
- 印出freeMap中每個bit的使用狀況
- 迴圈中針對directory每一個使用中的DirectoryEntry，印出其file header與檔案內容

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

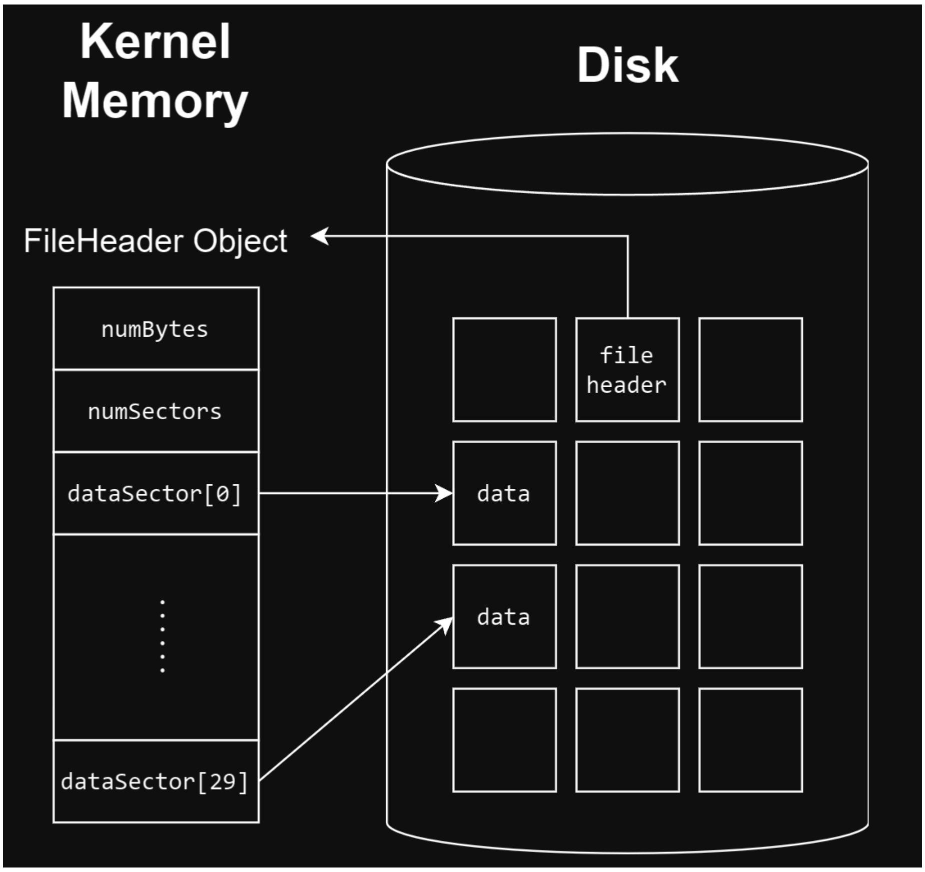
FileHeader即為inode，以下為FileHeader的私有成員，紀錄檔案大小、檔案所使用的sector數量及data blocks使用的sectors，FileHeader的大小為128 bytes，正好可放進一個sector，扣除numBytes和numSectors後，剩餘120 bytes用於存放長度為30的dataSectors，這表示一個檔案只能有30個data blocks，即3840 bytes。

```
#define NumDirect ((SectorSize - 2 * sizeof(int)) / sizeof(int))
#define MaxFileSize (NumDirect * SectorSize)
class FileHeader
{
    // ......
private:
```



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

FileHeader使用下圖方式存取data blocks



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

承上題, 一個檔案最大3840 bytes（3.75 KB）。

Part II - Implementation

1. File I/O system calls

首先比照MP1，在ksyscall.h加上System call介面

a. code/userprog/ksyscall.h

```
int SysCreate(char *filename, int initialSize)
{
    return kernel->fileSystem->Create(filename, initialSize);
}
OpenFileId SysOpen(char *name)
{
    return kernel->fileSystem->OpenAFile(name);
}
int SysWrite(char *buffer, int size, OpenFileId id)
{
    return kernel->fileSystem->WriteFile_(buffer, size, id);
}
int SysRead(char *buffer, int size, OpenFileId id)
{
    return kernel->fileSystem->ReadFile(buffer, size, id);
}
int SysClose(OpenFileId id)
{
    return kernel->fileSystem->CloseFile(id);
}
```

在filesys.h加入OpenFileTable操作方法及相關參數宣告

b. code/filesys/filesys.h

```
//  ......
#define FILE_OPEN_LIMIT 20

class FileSystem
{
public:
    // ......
    OpenFileId OpenAFile(char *name);
    int WriteFile_(char *buffer, int size, OpenFileId id);
    int ReadFile(char *buffer, int size, OpenFileId id);
    int CloseFile(OpenFileId id);
private:
    OpenFile *OpenFileTable[FILE_OPEN_LIMIT];
    bool isValidFileId(OpenFileId id);
};
```

在filesys.cc加上OpenFileTable操作方法，實作基本與MP1相同，都會先檢驗OpenFileTable和OpenFileId的狀態是否合法再進行操作

c. code/filesys/filesys.cc

```
OpenFileId FileSystem::OpenAFile(char *name)
{
    OpenFileId id;
    for (id = 0; id < 20; ++id)
    {
        if (OpenFileTable[id] == NULL)
        {
            break;
        }
    }
    // exceed the opened file limit
    if (id == 20)
    {
        return -1;
    }
    OpenFileTable[id] = Open(name);
    return id;
}
int FileSystem::WriteFile_(char *buffer, int size, OpenFileId id)
{
    if (buffer != NULL && size >= 0 && isValidFileId(id))
    {
        return OpenFileTable[id]->Write(buffer, size);
    }
    return -1;
}
```

```
int FileSystem::ReadFile(char *buffer, int size, OpenFileId id)
{
    if (buffer != NULL && size >= 0 && isValidFileId(id))
    {
        return OpenFileTable[id]->Read(buffer, size);
    }
    return -1;
}

int FileSystem::CloseFile(OpenFileId id)
{
    if (isValidFileId(id))
    {
        delete OpenFileTable[id];
        OpenFileTable[id] = NULL;
        return 1;
    }
    return -1;
}

bool FileSystem::isValidFileId(OpenFileId id)
{
    return id >= 0 && id < FILE_OPEN_LIMIT && OpenFileTable[id] != NULL;
}
```

比照MP1，加上System call的定義，除了SC_Create多帶了一個檔案大小的參數之外，其他方法都與MP1的實作一模一樣，為節省版面，以下僅列出SC_Create的switch case

d. code/userprog/exception.cc

```
case SC_Create:
    val = kernel->machine->ReadRegister(4);
    initialSize = kernel->machine->ReadRegister(5);
    {
        char *filename = &(kernel->machine->mainMemory[val]);
        // cout << filename << endl;
        status = SysCreate(filename, initialSize);
        kernel->machine->WriteRegister(2, (int)status);
    }
    // move PC and return...
```

2. File System

為了支援較大的檔案，必須先增加disk.h中Nachos模擬硬碟的容量，將SectorsPerTrack及NumTracks都從原先的32調整為1024，調整後，模擬硬碟大小從128 KB提升至128 MB。

$1024\ tracks\ \times\ 1024\ sectors/track\ \times\ 128\ bytes/sector\ =\ 128MB$

a. code/machine/disk.h

```
const int SectorsPerTrack  = 1024;  // number of sectors per disk track
const int NumTracks = 1024;    // number of tracks per disk
```

為支援至多64 MB的大型檔案，我重新將FileHeader::dataSectors設計為30-way tree的結構，至多支援4層（檔案大小約98 MB），為增加可讀性並簡化實作，在filehdr.h定義LEVEL_LIMIT表示此結構至多支援4層、定義MAX_SIZE表示每層的單一檔案大小上限。

新增私有成員dataSectorMapping及children，前者能在ByteToSector方法中直接用索引算出offset位置在哪一個physical sector上，不用遍歷整個樹狀結構，後者則方便我們遍歷樹狀結構進行Allocate、Deallocate等操作，兩者都屬於in-core資料，不會寫回硬碟中，每次使用FetchFrom從硬碟讀取資料，必須重建這兩者，而為了在遍歷樹狀結構時重建dataSectorMapping，新增Allocate及FetchFrom的私有方法，幫助公開方法以遞迴方式重建dataSectorMapping。

除了上述變更外，也新增私有方法whichLv幫助計算該檔案屬於哪一階層，新增私有方法clear協助清理記憶體、回復私有成員的狀態，原有Print方法加上printContent的旗標，以滿足Bonus部分只印出file header的需求。

b. code/filesys/filehdr.h

```
#define INVALID_SECTOR -1
#define LEVEL_LIMIT 4
const int NUM_DIRECT = (SectorSize - 2 * sizeof(int)) / sizeof(int);
// 3840 bytes = 3.75 KB (30 sectors)
const int MAX_SIZE_L0 = NUM_DIRECT * SectorSize;
// 115200 bytes = 112.5 KB (900 sectors)
const int MAX_SIZE_L1 = NUM_DIRECT * NUM_DIRECT * SectorSize;
// 3456000 bytes = 3375 KB (27000 sectors)
const int MAX_SIZE_L2 = NUM_DIRECT * NUM_DIRECT * NUM_DIRECT * SectorSize;
```

```
// 103680000 bytes = 101250 KB = 98.876953125 MB (810000 sectors)
const int MAX_SIZE_L3 = NUM_DIRECT * NUM_DIRECT * NUM_DIRECT * NUM_DIRECT *
SectorSize;
const int MAX_SIZE[LEVEL_LIMIT] = {MAX_SIZE_L0, MAX_SIZE_L1, MAX_SIZE_L2,
MAX_SIZE_L3};

class FileHeader
{
public:
    // ......
    void Print(bool printContent = TRUE);

private:
    // ===================disk part===================
    // Number of bytes in the file
    int numBytes;
    // Number of data sectors in the file
    int numDataSectors;
    // Disk sector numbers for each data block in the file
    int dataSectors[NUM_DIRECT];
    // ===================disk part===================
    // ===================in-core part===================
    // index: logical sector, value: physical sector
    vector<int> dataSectorMapping;
    FileHeader *children[NUM_DIRECT];
    // ===================in-core part===================
    int whichLv(int fileSize);
    void clear();
    bool Allocate(PersistentBitmap *bitMap, int fileSize, vector<int>
&pSectors);
    void FetchFrom(int sectorNumber, vector<int> &pSectors);
};
```

FileHeader建構和解構時，都會呼叫clear()初始化成員的狀態，clear()將numBytes、numDataSectors設為-1，將dataSectors全部寫為INVALID_SECTOR（-1），清除dataSectorMapping，並將children中非NULL的指標釋放，此處會遞迴觸發child FileHeader的解構式，釋放整棵樹的heap空間。

whichLv方法簡單地遍歷MAX_SIZE陣列，找到該檔案屬於哪個階層，此處假設不會有檔案超過第四層可容納的上限(約98 MB)。

ByteToSector方法直接使用dataSectorMapping轉換logical sector與physical sector，在轉換途中，也順便檢查私有成員numDataSectors和dataSectorMapping的大小是否一致，確保這兩個私有成員被正確地維護。

c. code/filesys/filehdr.cc 建構式、解構式、clear、whichLv及ByteToSector方法

```
FileHeader::FileHeader()
{
    memset(children, NULL, sizeof(children));
    clear();
}
FileHeader::~FileHeader() { clear(); }
int FileHeader::whichLv(int fileSize)
{
    for (int i = 0; i < LEVEL_LIMIT; ++i)
    {
        if (fileSize <= MAX_SIZE[i])
        {
            return i;
        }
    }
    ASSERTNOTREACHED(); // don't support file larger than MAX_SIZE_L3
}
void FileHeader::clear()
{
    numBytes = -1;
    numDataSectors = -1;
    memset(dataSectors, INVALID_SECTOR, sizeof(dataSectors));
    dataSectorMapping.resize(0);
    for (int i = 0; i < NUM_DIRECT; ++i)
    {
        if (children[i])
        {
            delete children[i];
            children[i] = NULL;
        }
        else
        {
            break;
        }
    }
}
int FileHeader::ByteToSector(int offset)
{
    int logicalSector = offset / SectorSize;
    int sectorsSz = static_cast<int>(dataSectorMapping.size());
    ASSERT(logicalSector >= 0 && logicalSector < sectorsSz && sectorsSz ==
numDataSectors);
    return dataSectorMapping[logicalSector];
}
```

公開的FileHeader::Allocate方法呼叫私有的Allocate方法，並傳入自己的dataSectorMapping以蒐集子FileHeader的data sector。Allocate先判斷freeMap是否有足夠的空間，之後根據FileHeader所屬層級分為兩種案例，第一種為葉節點，和Nachos原本的實作相同，不過要將取得的data sector寫入自己的dataSectorMapping，第二種為內部節點，內部節點在迴圈中將未配置的fileSize交給子節點配置，子節點配置的檔案大小不超過該層級所能容納的最大限制，值得注意的是，內部節點所取得的sector並非用於儲存data，故不加入dataSectorMapping之中。當此節點完成配置後，將此節點配置的data sectors寫入parent的dataSectorMapping之中，如此一來，根節點的dataSectorMapping就有整個檔案的data sectors資料。

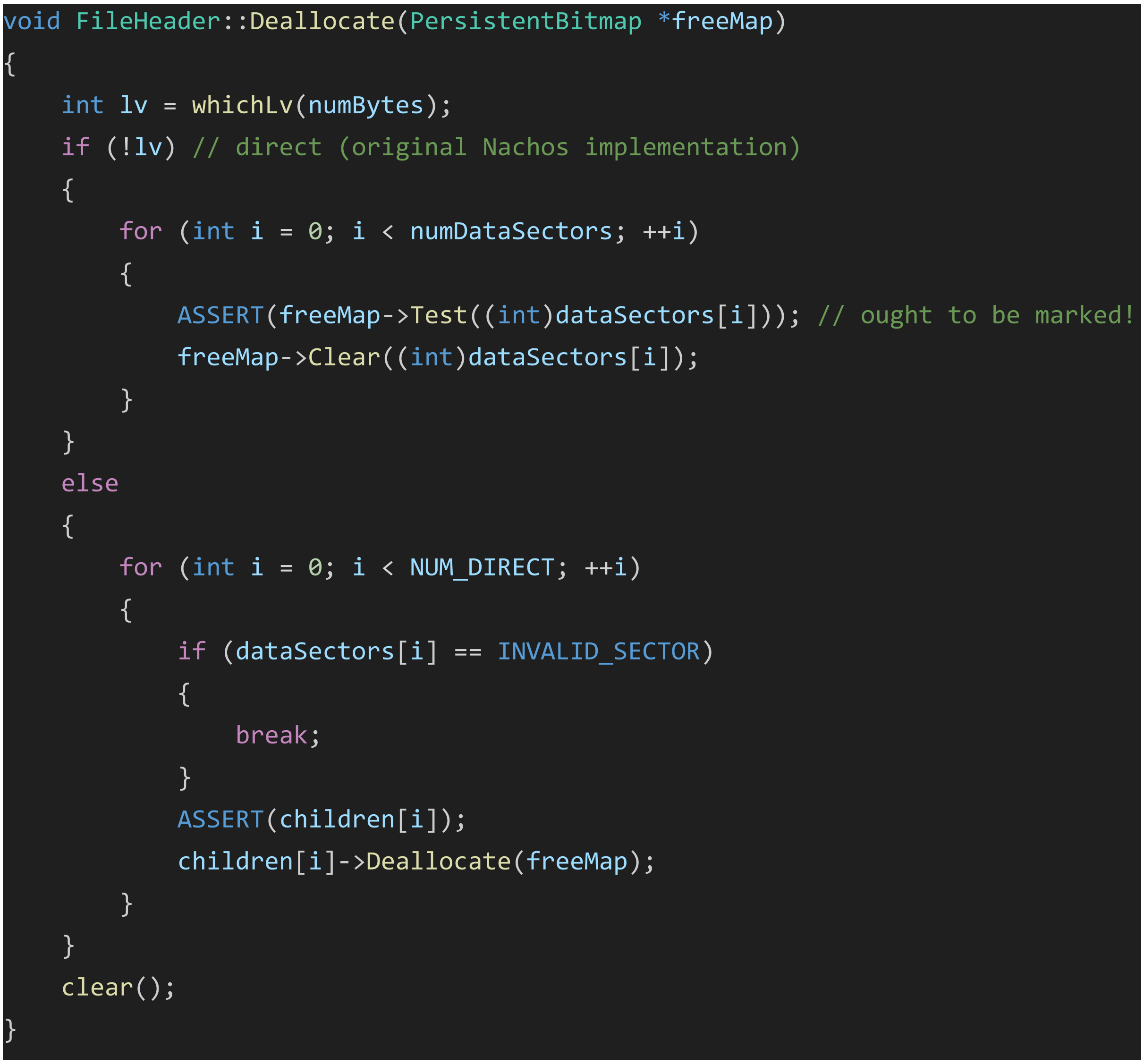
d. code/filesys/filehdr.cc Allocate

```
bool FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
{
    return Allocate(freeMap, fileSize, this->dataSectorMapping);
}

bool FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize, vector<int>
&pSectors)
{
    numBytes = fileSize;
    numDataSectors = divRoundUp(fileSize, SectorSize);
    if (freeMap->NumClear() < numDataSectors)
    {
        return FALSE; // not enough space
    }
    int lv = whichLv(fileSize);
    // DEBUG(dbgMp4, "allocate lv " << lv << " file header which requires " <<
fileSize << " bytes");
    if (!lv) // direct (original Nachos implementation)
    {
        for (int i = 0; i < numDataSectors; i++)
        {
            dataSectors[i] = freeMap->FindAndSet();
            // since we checked that there was enough free space, we expect
this to succeed
            ASSERT(dataSectors[i] >= 0);
            this->dataSectorMapping.push_back(dataSectors[i]);
        }
    }
    else
    {
        int i = 0;
        while (fileSize > 0)
        {
            dataSectors[i] = freeMap->FindAndSet();
            ASSERT(dataSectors[i] >= 0);
            this->children[i] = new FileHeader();
            int subHdrSize = min(fileSize, MAX_SIZE[lv - 1]);
            this->children[i]->Allocate(freeMap, subHdrSize,
this->dataSectorMapping); // recursive
            fileSize -= subHdrSize;
            ++i;
        }
    }
    if (this->dataSectorMapping != pSectors)
    {
        pSectors.insert(pSectors.end(), this->dataSectorMapping.begin(),
this->dataSectorMapping.end());
    }
    return TRUE;
}
```

FileHeader::Deallocate和Allocate方法十分類似，都是先判斷FileHeader所屬層級，再分為葉節點和內部節點兩個案例，葉節點Deallocate的方法和Nachos原本的實作相同，內部節

點則遍歷有配置的data sectors，此時，因為Allocate和FetchFrom在配置或讀取資料時，維護了children中的指標，故只要data sector並非INVALID_SECTOR，就能確定children[i]一定指向子節點，只需要遞迴呼叫子節點的Deallocate方法，就能釋放整個樹的sectors。最後當這個節點及其下的所有節點都釋放完sectors時，呼叫clear()清理自身的私有成員並遞迴釋放非NULL的children指標。

e. code/filesys/filehdr.cc Deallocate



```
void FileHeader::Deallocate(PersistentBitmap *freeMap)
{
    int lv = whichLv(numBytes);
    if (!lv) // direct (original Nachos implementation)
    {
        for (int i = 0; i < numDataSectors; ++i)
        {
            ASSERT(freeMap->Test((int)dataSectors[i])); // ought to be marked!
            freeMap->Clear((int)dataSectors[i]);
        }
    }
    else
    {
        for (int i = 0; i < NUM_DIRECT; ++i)
        {
            if (dataSectors[i] == INVALID_SECTOR)
            {
                break;
            }
            ASSERT(children[i]);
            children[i]->Deallocate(freeMap);
        }
    }
    clear();
}
```

公開的FileHeader::FetchFrom方法呼叫呼叫私有的FetchFrom方法，和Allocate方法類似，都傳入自己的dataSectorMapping以蒐集子FileHeader的data sector。私有的FetchFrom使用大小為128 bytes的buffer讀取disk part的私有成員numBytes、numDataSectors、dataSectors資料，和Allocate相似，接下來根據FileHeader所屬層級分為葉節點和內部節點兩個案例，葉節點直接將dataSectors放入自身的dataSectorMapping中，內部節點則遍歷

有效的dataSectors，這些dataSectors為子FileHeader所在的sector，建立子節點指標，並遞迴呼叫子節點的FetchFrom將整棵樹的資料從硬碟讀入kernel。當這個節點及其下的所有節點都讀取完成時，將此節點配置的dataSectors寫入parent的dataSectorMapping之中。

f. code/filesys/filehdr.cc FetchFrom

```
void FileHeader::FetchFrom(int sector)
{
    ASSERT(this->dataSectorMapping.empty());
    FetchFrom(sector, this->dataSectorMapping);
}
void FileHeader::FetchFrom(int sector, vector<int> &pSectors)
{
    int buf[SectorSize / sizeof(int)];
    kernel->synchDisk->ReadSector(sector, (char *)buf);
    numBytes = buf[0];
    numDataSectors = buf[1];
    memcpy(dataSectors, buf + 2, sizeof(dataSectors));
    // rebuild in-core part
    int lv = whichLv(numBytes);
    if (!lv) // leaf
    {
        for (int i = 0; i < numDataSectors; i++)
        {
            this->dataSectorMapping.push_back(dataSectors[i]);
        }
    }
    else
    {
        for (int i = 0; i < NUM_DIRECT; ++i)
        {
            if (dataSectors[i] == INVALID_SECTOR)
            {
                break;
            }
            this->children[i] = new FileHeader();
            this->children[i]->FetchFrom(dataSectors[i],
this->dataSectorMapping);
        }
    }
    if (this->dataSectorMapping != pSectors)
    {
        pSectors.insert(pSectors.end(), this->dataSectorMapping.begin(), 
this->dataSectorMapping.end());
    }
}
```

FileHeader::WriteBack方法因不需要建立、維護in-core的私有成員，較FetchFrom簡單。WriteBack使用大小為128 bytes的buffer將disk part的私有成員numBytes、numDataSectors、dataSectors寫入傳入的sector中，如果這個FileHeader有子節點，則遍歷有效的dataSectors，遞迴呼叫子節點的WriteBack方法，將整棵FileHeader樹的disk part都寫入對應的sector中。

g. code/filesys/filehdr.cc WriteBack

```
void FileHeader::WriteBack(int sector)
{
    int buf[SectorSize / sizeof(int)];
    buf[0] = numBytes;
    buf[1] = numDataSectors;
    memcpy(buf + 2, dataSectors, sizeof(dataSectors));
    kernel->synchDisk->WriteSector(sector, (char *)buf);
    // write disk part
    int lv = whichLv(numBytes);
    if (lv)
    {
        for (int i = 0; i < NUM_DIRECT; ++i)
        {
            if (dataSectors[i] == INVALID_SECTOR)
            {
                break;
            }
            ASSERT(children[i]);
            children[i]->WriteBack(dataSectors[i]);
        }
    }
}
```

FileHeader::Print印出該FileHeader的disk part和in-core part的大小，如果需要印出檔案內容，則使用dataSectorMapping印出該節點及其下的所有data sectors，接下來根據

FileHeader的層級分為葉節點和內部節點兩個案例，葉節點的部分和Nachos的原始實作相同，但會先判斷是否需要印出檔案內容，內部節點則遍歷有效的dataSectors，遞迴呼叫子節點的Print方法。

h. code/filesys/filehdr.cc Print

```
void FileHeader::Print(bool printContent)
{
    cout << "FileHeader contents:" << endl
         << "1. File size: " << numBytes << " bytes (" << numDataSectors << "
sectors)" << endl
         << "2. FileHeader size in disk: " << (sizeof(FileHeader) -
sizeof(dataSectorMapping) - sizeof(children)) << " bytes" << endl
         << "3. FileHeader size in memory: " << (sizeof(FileHeader) +
dataSectorMapping.size() * sizeof(int) + sizeof(children)) << " bytes" <<
endl;
    if (printContent)
    {
        cout << "4. Data blocks: " << endl;
        for (int i = 0; i < numDataSectors; ++i)
        {
            ASSERT(dataSectorMapping[i] != INVALID_SECTOR);
            cout << dataSectorMapping[i] << " ";
        }
        printf("\nFile contents:\n");
    }

    int lv = whichLv(numBytes);
    if (!lv) // leaf
    {
        if (!printContent)
        {
            return;
        }
        char *data = new char[SectorSize];
        for (int i = 0, k = 0; i < numDataSectors; ++i)
        {
            kernel->synchDisk->ReadSector(dataSectors[i], data);
            for (int 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");
        }
        delete[] data;
    }
    else
    {
        for (int i = 0; i < NUM_DIRECT; ++i)
        {
            if (dataSectors[i] == INVALID_SECTOR)
            {
                break;
            }
            ASSERT(children[i]);
            children[i]->Print(printContent);
        }
    }
}
```

將NumDirEntries調整為64，使一個目錄能容納64個檔案和目錄。為支援子目錄，DirectoryEntry新增isDir屬性，以區分該DirectoryEntry為檔案還是目錄，同時調整Directory原先Find、FindIndex、Add、Remove的方法宣告，加入isDir參數以區分要操作、尋找的DirectoryEntry。Directory加上RemoveAll及RecursivelyList的宣告以支援-rr和-lr指令。

i. code/filesys/directory.h

```
#define NumDirEntries 64
#define FILE false
#define DIR true

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

class Directory
{
public:
    // ......
    // Find the sector number of the FileHeader for file: "name"
    int Find(const char *name, bool isDir);
    // Add a file name into the directory
    bool Add(const char *name, int newSector, bool isDir);
    // Remove a file from the directory
    bool Remove(const char *name, bool isDir);
    // Remove all files and directories under this directory
    bool RemoveAll(PersistentBitmap *freeMap);
    // Print the names of all the files in the directory (command -l)
    void List();
    // command -lr
    void RecursivelyList(int depth);
private:
    //---disk part---//
    // Table of pairs: <file name, file header location>
    DirectoryEntry *table;
    //---disk part---//
    //---in-core part---//
    // Number of directory entries
    int tableSize;
    //---in-core part---//
    int FindIndex(const char *name, bool isDir);
};
```

Directory建構式在初始化迴圈中設定每個table entry的isDir屬性。

j. code/filesys/directory.cc 建構式

```
Directory::Directory(int size)
{
    table = new DirectoryEntry[size];
    memset(table, 0, sizeof(DirectoryEntry) * size); // dummy operation to 
keep valgrind happy
    tableSize = size;
    for (int i = 0; i < tableSize; i++)
    {
        table[i].isDir = FALSE;
        table[i].inUse = FALSE;
    }
}
```

Directory::FindIndex方法增加判斷isDir屬性是否和要找的目標相容，Find、Add和Remove則是將外部傳入的isDir參數直接傳給FindIndex，其中Add要記得設定新table entry的isDir屬性，其餘都和原實作相同

k. code/filesys/directory.cc FindIndex、Find、Add及Remove

```
int Directory::FindIndex(const char *name, bool isDir)
{
    for (int i = 0; i < tableSize; i++)
    {
        if (table[i].inUse && table[i].isDir == isDir && 
!strncmp(table[i].name, name, FileNameMaxLen))
        {
            return i;
        }
    }
    return -1; // name not in directory
}

int Directory::Find(const char *name, bool isDir)
{
    int i = FindIndex(name, isDir);
    if (i != -1)
    {
        return table[i].sector;
    }
    return INVALID_SECTOR;
}

bool Directory::Add(const char *name, int newSector, bool isDir)
{
    if (FindIndex(name, isDir) != -1)
    {
        return FALSE;
    }

    for (int i = 0; i < tableSize; i++)
    {
        if (!table[i].inUse)
        {
            table[i].isDir = isDir;
            table[i].inUse = TRUE;
            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(const char *name, bool isDir)
{
    int i = FindIndex(name, isDir);
    if (i == -1)
    {
        return FALSE; // name not in directory
    }
    table[i].inUse = FALSE;
    DEBUG(dbgMp4, "remove " << name << " (dir)");
    return TRUE;
}
```

Directory::RemoveAll遍歷每個inUse的table entry，如果該entry是子目錄，則使用entry中紀錄的sector開啟子目錄，並遞迴呼叫子目錄的RemoveAll，清除子目錄下的所有檔案和目錄（不含清除子目錄），完成後，或者該entry為檔案，將該entry的FileHeader及data占用的

sectors歸還freeMap。因為Directory物件並未記錄自身的FileHeader sector資訊，此方法的呼叫者必須負責將此Directory的變更寫回硬碟中。

l. code/filesys/directory.cc RemoveAll

```
bool Directory::RemoveAll(PersistentBitmap *freeMap)
{
    for (int i = 0; i < tableSize; i++)
    {
        if (!table[i].inUse)
        {
            continue;
        }
        if (table[i].isDir)
        {
            OpenFile *openFh = new OpenFile(table[i].sector);
            Directory *dir = new Directory(NumDirEntries);
            dir->FetchFrom(openFh);
            dir->RemoveAll(freeMap);
            delete openFh;
            delete dir;
        }
        DEBUG(dbgMp4, "remove " << table[i].name << " (dir or file)");
        ASSERT(freeMap->Test(table[i].sector));
        freeMap->Clear(table[i].sector); // return header sector
        FileHeader *fh = new FileHeader();
        fh->FetchFrom(table[i].sector);
        fh->Deallocate(freeMap); // return data sectors
        delete fh;

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

Directory::RecursivelyList遍歷每個inUse的table entry，根據深度階層印出空格縮排後，印出自身為檔案或目錄以及自身的名字，如果該table entry為子目錄，開啟子目錄並遞迴呼叫子目錄的RecursivelyList方法，傳入的深度階層參數往上+1，以正確印出縮排。

m. code/filesys/directory.cc RecursivelyList

```
void Directory::RecursivelyList(int depth)
```

```
{
    for (int i = 0; i < tableSize; i++)
    {
        if (!table[i].inUse)
        {
            continue;
        }
        // indentation
        for (int j = 0; j < depth * 4; ++j)
        {
            cout << " ";
        }
        cout << "[" << (table[i].isDir ? "D" : "F") << "] " << table[i].name << endl;
        if (table[i].isDir)
        {
            OpenFile *f = new OpenFile(table[i].sector);
            Directory *dir = new Directory(NumDirEntries);
            dir->FetchFrom(f);
            delete f;
            dir->RecursivelyList(depth + 1);
            delete dir;
        }
    }
}
```

為了使尋找檔案和目錄的操作變得輕鬆，在filesys.h新增class FileFinder，負責解析路徑、尋找檔案，尋找途中會將parent FileHeader sector寫入pFhSector屬性、將自身FileHeader sector寫入fhSector屬性、將檔名（路徑字串中，最後一個 '/' 後的部分）寫入filename屬性，並將搜尋結果寫入exist屬性。

n. code/filesys/filesys.h FileFinder

```
#define PATH_NAME_MAX_LEN 256
#define FILE_OPEN_LIMIT 20

class FileFinder
{
    friend class FileSystem;
    // is the file exist?
    bool exist;
    // sector number of the parent dir
    int pFhSector;
    // sector number of the file header
    int fhSector;
    string filename;

    FileFinder();
    static vector<string> splitPath(const char *name, bool
&exceedPathLenLimit);
    // find the file by path (name) and set the result to private fields
    void find(const char *name, bool isDir, OpenFile *root);
    void deleteOpenFile(OpenFile *openfile, OpenFile *root);
};
```

FileFinder建構式初始化表示搜尋結果的屬性，避免使用時誤解搜尋結果。splitPath將傳入的路徑字串依'/'符號進行分割，並回傳結果，分割途中順便檢查路徑字串是否超過PATH_NAME_MAX_LEN（256）的限制。deleteOpenFile則是為了簡化FileFinder::find中重複的刪除操作而存在的方法。

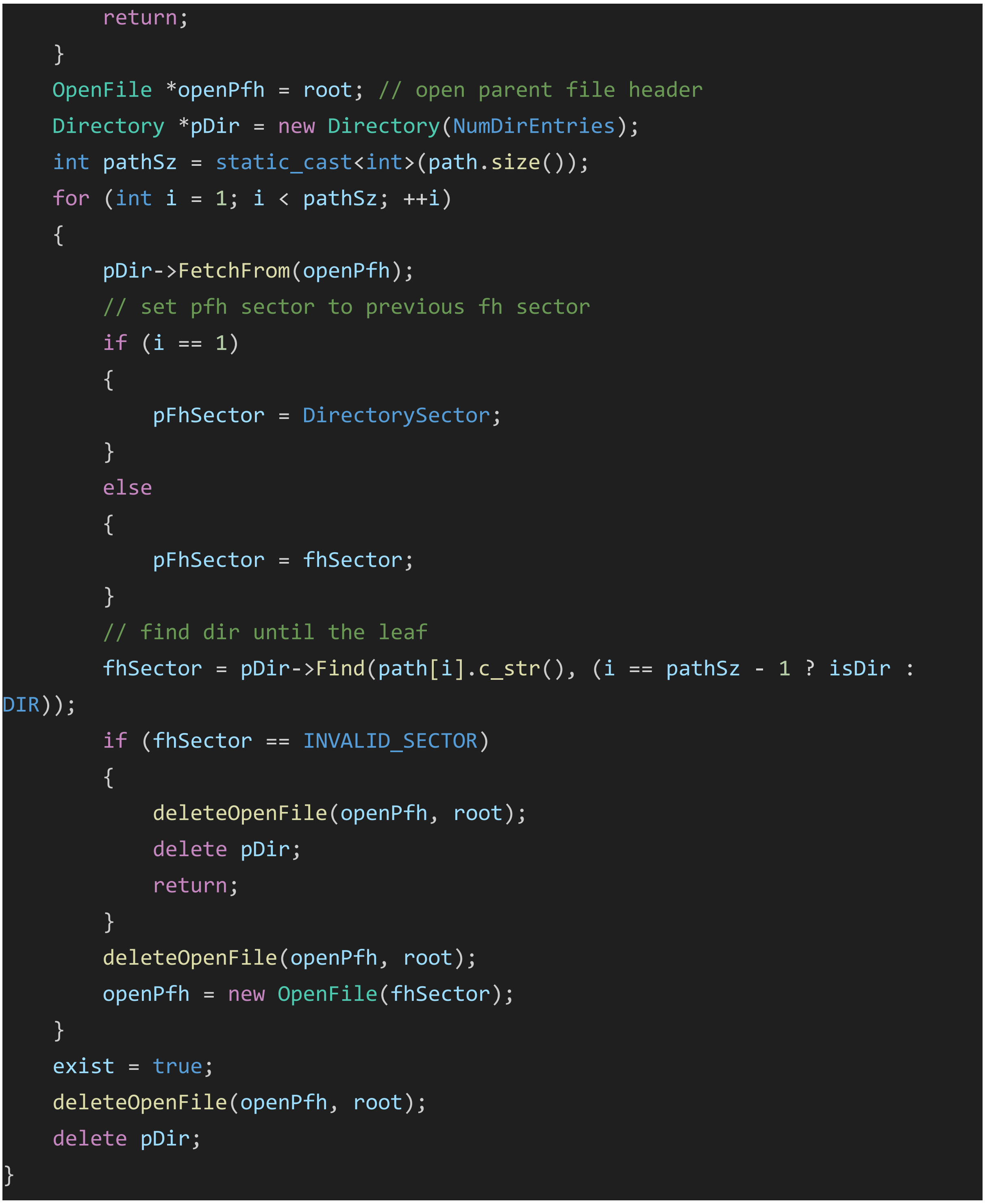
o. code/filesys/filesys.cc FileFinder建構式、splitPath及deleteOpenFile

```
FileFinder::FileFinder() : exist(FALSE), pFhSector(INVALID_SECTOR),
fhSector(INVALID_SECTOR) {}

vector<string> FileFinder::splitPath(const char *str, bool
&exceedPathLenLimit)
{
    vector<string> result;
    string token;
    int i = 0;
    while (*str)
    {
        if (*str == '/')
        {
            result.push_back(token);
            token.clear();
        }
        else
        {
            token += *str;
        }
        ++str;
        ++i;
    }
    exceedPathLenLimit = i >= PATH_NAME_MAX_LEN;
    result.push_back(token);
    return result;
}

void FileFinder::deleteOpenFile(OpenFile *openfile, OpenFile *root)
{
    if (openfile != NULL && openfile != root)
    {
        delete openfile;
    }
}
```

FileFinder::find方法先調用splitPath處理路徑字串，並檢查路徑長度是否超過上限，如果路徑字串為根目錄，將fhSector設定為DirectorySector、exist設定為true後返回表示尋找成功，否則遍歷分割的路徑字串，從根目錄開始尋找檔案，尋找的途中維護pFhSector及fhSector，除了最後一層尋找的目標型態由傳入的isDir參數決定，途中都是尋找目錄，如果中途找不到對應的目錄，或最後一層不存在目標型態、名稱相同的檔案或目錄，清除指標後返回，表示尋找失敗，否則遍歷到最後，表示找到目標，設定exist為true後返回。

p. code/filesys/filesys.cc FileFinder find

```
void FileFinder::find(const char *name, bool isDir, OpenFile *root)
{
    bool exceedPathLenLimit = FALSE;
    vector<string> path = splitPath(name, exceedPathLenLimit);
    filename = path.back();
    if (exceedPathLenLimit)
    {
        return;
    }
    // the file is the root dir
    if (!strcmp(name, "/"))
    {
        ASSERT(isDir);
        exist = true;
        fhSector = DirectorySector;
```



```
        return;
    }
    OpenFile *openPfh = root; // open parent file header
    Directory *pDir = new Directory(NumDirEntries);
    int pathSz = static_cast<int>(path.size());
    for (int i = 1; i < pathSz; ++i)
    {
        pDir->FetchFrom(openPfh);
        // set pfh sector to previous fh sector
        if (i == 1)
        {
            pFhSector = DirectorySector;
        }
        else
        {
            pFhSector = fhSector;
        }
        // find dir until the leaf
        fhSector = pDir->Find(path[i].c_str(), (i == pathSz - 1 ? isDir :
DIR));
        if (fhSector == INVALID_SECTOR)
        {
            deleteOpenFile(openPfh, root);
            delete pDir;
            return;
        }
        deleteOpenFile(openPfh, root);
        openPfh = new OpenFile(fhSector);
    }
    exist = true;
    deleteOpenFile(openPfh, root);
    delete pDir;
}
```

FileSystem除了新增System call的宣告外，既有的Remove及List方法加上recursive選項，以支援-rr和-lr指令，新增PrintHeader公開方法對應Bonus印出不同大小的FileHeader指令（新的自訂指令-h），新增Mkdir公開方法以支援新增目錄的指令（-mkdir），新增私有方法createFileOrDir以整合Create和Mkdir兩個公開方法的共通邏輯，新增私有方法

recursivelyRemove以實作-rr指令，新增私有方法returnSectorsToFreeMap以簡化將sectors歸還freeMap的操作。

q. code/filesys/filesys.h FileSystem

```
class FileSystem
{
    friend class FileFinder;

public:
    // ......
    // Create a file (UNIX creat)
    bool Create(char *name, int initialSize);
    // Open a file (UNIX open)
    OpenFile *Open(char *name);
    // This function is used for kernel open system call
    OpenFileId OpenAFile(char *name);
    int WriteFile_(char *buffer, int size, OpenFileId id);
    int ReadFile(char *buffer, int size, OpenFileId id);
    int CloseFile(OpenFileId id);
    // Delete a file (UNIX unlink)
    bool Remove(const char *name, bool recursive);
    // List all the files in the file system
    void List(char *name, bool recursive);
    void PrintHeader(char *name);
    bool Mkdir(char *name);
    // ......
private:
    // ......
    OpenFile *OpenFileTable[FILE_OPEN_LIMIT];
    bool isValidFileId(OpenFileId id);
    /**
     * @brief Create a File Or Dir
     *
     * @param name absolute path
     * @param isDir is this a dir or a file
     * @param initialSize file size (will be ignored if this is a dir)
     * @return true success
     * @return false fail
     */
    bool createFileOrDir(char *name, bool isDir, int initialSize);
    bool recursivelyRemove(const char *name);
    // Return data and header sectors to freeMap
    void returnSectorsToFreeMap(int fhSector, PersistentBitmap *freeMap);
};
```

Create和Mkdir兩者都是直接呼叫私有方法createFileOrDir完成實作，其中Create對應的是檔案的新增，Mkdir是目錄的新增。createFileOrDir的實作可分為4個部分，第一，根據路徑和屬性尋找檔案，如果檔案已存在，返回無法新增檔案，否則parent目錄必定存在，進入第二部分，將檔名寫入parent目錄的其中一個entry，我們假設測試案例總是有足夠的空間可以寫入，第三部分，配置data sectors，如果要新增的是目錄，則大小為NumDirEntries * sizeof(DirectoryEntry)，否則為傳入的initialSize，建立FileHeader配置data sectors後，第四部份，將FileHeader、parent目錄、freeMap的變更寫回硬碟，如果要新增的是目錄，也將目錄的table寫回硬碟，清除指標後返回新增成功。

r. code/filesys/filesys.cc Create、Mkdir及createFileOrDir

```
bool FileSystem::Create(char *name, int initialSize)
{
    return createFileOrDir(name, FILE, initialSize);
}
bool FileSystem::Mkdir(char *name)
{
    return createFileOrDir(name, DIR, -1);
}
bool FileSystem::createFileOrDir(char *name, bool isDir, int initialSize)
{
    // 1. find the parent dir
    FileFinder finder = FileFinder();
    finder.find(name, isDir, directoryFile);
    if (finder.exist)
    {
        DEBUG(dbgMp4, "File or dir exist, cannot create: " << name);
        return FALSE;
    }
    ASSERT(finder.pFhSector != INVALID_SECTOR); // parent dir should exist

    // 2. add file header to the parent dir
    PersistentBitmap *freeMap = new PersistentBitmap(freeMapFile, NumSectors);
    int sector = freeMap->FindAndSet();
    ASSERT(sector >= 0);
    OpenFile *openPfh = new OpenFile(finder.pFhSector);
    Directory *pDir = new Directory(NumDirEntries);
    pDir->FetchFrom(openPfh);

    ASSERT(pDir->Add(finder.filename.c_str(), sector, isDir)) // assume always
have space

    // 3. allocate data blocks
    int size = isDir ? NumDirEntries * sizeof(DirectoryEntry) : initialSize;
    ASSERT(size >= 0);
    FileHeader *fh = new FileHeader();
    ASSERT(fh->Allocate(freeMap, size));

    // 4. write back
    fh->WriteBack(sector);
    pDir->WriteBack(openPfh);
    freeMap->WriteBack(freeMapFile);
    if (isDir)
    {
        Directory *dir = new Directory(NumDirEntries);
        OpenFile *openFh = new OpenFile(sector);
        dir->WriteBack(openFh);
        delete dir;
        delete openFh;
    }
    delete freeMap;
    delete openPfh;
    delete pDir;
    delete fh;
    return TRUE;
}
```

FileSystem::Open使用FileFinder嘗試在傳入的路徑下尋找檔案或目錄，測試案例假設要開啟的檔案必定存在，故finder.exist必定為true，且FileHeader sector必定為非負整數，回傳開啟的檔案指標。

s. code/filesys/filesys.cc Open

```
OpenFile *FileSystem::Open(char *name)
{
    FileFinder finder = FileFinder();
    finder.find(name, FILE, directoryFile);
    if (!finder.exist)
    {
        finder.find(name, DIR, directoryFile);
    }
    ASSERT(finder.exist && finder.fhSector >= 0);
    return new OpenFile(finder.fhSector);
}
```

FileSystem::Remove增加遞迴刪除的選項，如果要遞迴刪除，直接交給recursivelyRemove方法，否則為單一檔案的刪除，使用FileFinder尋找檔案，不存在則回傳刪除失敗，否則將該檔案data和header占用的sectors歸還freeMap，開啟parent目錄，將檔名從parent目錄中移除，最後將freeMap和parent目錄的變更寫回硬碟。

FileSystem::recursivelyRemove先使用FileFinder尋找目錄，如果找不到，表示為檔案，呼叫Remove進行單檔刪除，否則呼叫Directory::RemoveAll方法，將該目錄下所有檔案和目錄移除(不包含該目錄)，成功之後將該目錄從其parent目錄刪除(如果為根目錄則不需要進行這一步)，並將freeMap和目錄的變更寫回硬碟。

t. code/filesys/filesys.cc Remove和recursivelyRemove

```
bool FileSystem::Remove(const char *name, bool recursive)
{
    if (recursive)
    {
        return recursivelyRemove(name);
    }
    FileFinder finder = FileFinder();
    finder.find(name, FILE, directoryFile);
    if (!finder.exist)
    {
        return FALSE;
    }
    PersistentBitmap *freeMap = new PersistentBitmap(freeMapFile, NumSectors);
    returnSectorsToFreeMap(finder.fhSector, freeMap);
    OpenFile *openPfh = new OpenFile(finder.pFhSector);
    Directory *pDir = new Directory(NumDirEntries);
    pDir->FetchFrom(openPfh);
    ASSERT(pDir->Remove(finder.filename.c_str(), FILE));

    freeMap->WriteBack(freeMapFile);
    pDir->WriteBack(openPfh);
    DEBUG(dbgMp4, "remove " << name << " (single file)");
    delete freeMap;
    delete openPfh;
    delete pDir;
    return TRUE;
}

bool FileSystem::recursivelyRemove(const char *name)
{
    FileFinder finder = FileFinder();
    finder.find(name, DIR, directoryFile);
    if (!finder.exist)
    {
        return Remove(name, FALSE); // remove single file
    };
    // remove all files/dirs in this dir
    PersistentBitmap *freeMap = new PersistentBitmap(freeMapFile, NumSectors);
    OpenFile *openfh = new OpenFile(finder.fhSector);
    Directory *dir = new Directory(NumDirEntries);
    dir->FetchFrom(openfh);
    ASSERT(dir->RemoveAll(freeMap));
    dir->WriteBack(openfh);
    freeMap->WriteBack(freeMapFile);
    // remove the dir
    if (finder.pFhSector != INVALID_SECTOR)
    {
        returnSectorsToFreeMap(finder.fhSector, freeMap);
        OpenFile *openPfh = new OpenFile(finder.pFhSector);
        Directory *pDir = new Directory(NumDirEntries);
        pDir->FetchFrom(openPfh);
        pDir->Remove(finder.filename.c_str(), DIR);
        pDir->WriteBack(openPfh);
        delete openPfh;
        delete pDir;
    }
    delete freeMap;
    delete openfh;
    delete dir;
    return TRUE;
}
```

FileSystem::List使用FileFinder尋找目錄，如果不存在則直接回傳，否則開啟目錄，並根據傳入的recursive選項決定呼叫Directory::RecursivelyList或Directory::List。

u. code/filesys/filesys.cc List

```
void FileSystem::List(char *name, bool recursive)
{
    FileFinder finder = FileFinder();
    finder.find(name, DIR, directoryFile);
    // file not exist
    if (!finder.exist)
    {
        return;
    }
    ASSERT(finder.fhSector != INVALID_SECTOR);
    OpenFile *f = new OpenFile(finder.fhSector);
    Directory *dir = new Directory(NumDirEntries);
    dir->FetchFrom(f);
    delete f;
    if (recursive)
    {
        dir->RecursivelyList(0);
    }
    else
    {
        dir->List();
    }
    delete dir;
}
```

FileSystem::PrintHeader對應新增的自訂指令-h，找到對應的檔案或目錄後，呼叫FileHeader::Print並傳入FALSE，表示只需印出header，不需印出檔案內容。

v. code/filesys/filesys.cc PrintHeader

```
void FileSystem::PrintHeader(char *name)
{
    FileFinder finder = FileFinder();
    finder.find(name, DIR, directoryFile);
    if (!finder.exist)
    {
        finder.find(name, FILE, directoryFile);
    }
    ASSERT(finder.fhSector != INVALID_SECTOR);
    FileHeader *hdr = new FileHeader();
    hdr->FetchFrom(finder.fhSector);
    hdr->Print(FALSE);
    delete hdr;
}
```

FileSystem::returnSectorsToFreeMap簡化將header和data sectors歸還freeMap的操作，呼叫者必須自行將freeMap的變更寫回硬碟。

w. code/filesys/filesys.cc returnSectorsToFreeMap

```
void FileSystem::returnSectorsToFreeMap(int fhSector, PersistentBitmap
*freeMap)
{
    ASSERT(freeMap->Test(fhSector));
    freeMap->Clear(fhSector); // return header sector
    FileHeader *fh = new FileHeader();
    fh->FetchFrom(fhSector);
    fh->Deallocate(freeMap); // return data sectors
    delete fh;
}
```

main方法新增-h指令以支援Bonus僅印出FileHeader的部分，對應-rr和-lr指令，將原本的Remove和List方法加上recursive選項，新增目錄則改為直接呼叫FileSystem::Mkdir方法。

x. code/threads/main.cc

```
// ......
int main(int argc, char **argv)
{
    // ......
    char *printHeaderName = NULL;
    bool mkdirFlag = false;
    bool recursiveListFlag = false;
    bool recursiveRemoveFlag = false;
    bool printHeaderFlag = false;
    for (i = 1; i < argc; i++)
    {
        // ......
        else if (strcmp(argv[i], "-h") == 0)
        {
            ASSERT(i + 1 < argc);
            printHeaderFlag = true;
            printHeaderName = argv[i + 1];
            i++;
        }
        // ......
    }

    if (removeFileName != NULL)
    {
        kernel->fileSystem->Remove(removeFileName, recursiveRemoveFlag);
    }
    // ......
    if (dirListFlag)
    {
        kernel->fileSystem->List(listDirectoryName, recursiveListFlag);
    }
    if (mkdirFlag)
    {
        kernel->fileSystem->Mkdir(createDirectoryName);
    }
    // ......
    if (printHeaderFlag)
    {
        kernel->fileSystem->PrintHeader(printHeaderName);
    }
}
```

因模擬硬碟大小提升至128 MB（1048576 sectors），freeMap大小增加為1048576 bits，為加速Bitmap::FindAndSet和Bitmap::NumClear方法，加入兩個新的私有成員numClear和cur，前者表示目前剩下多少仍是0的bits，後者表示下次從此處開始尋找可用的bit。

y. code/lib/bitmap.h

```
class Bitmap
```

```
{
// ......
protected:
    // Number of clear bits
    int numClear;
    // Start looking for clear bit here. Assuming only the first clear bit
will be marked.
    int cur;
// ......
};
```

Bitmap建構時，將numClear初始化為numItems，cur初始化為0。

Mark時，如果這個bit原本為0，將cur設到此處，並減少numClear。Clear時，如果這個bit位置比cur小，則將cur設到此處，表示下次從此處開始尋找可用的bit，同時，如果這個bit原先為1，則增加numClear。

FindAndSet直接使用numClear判斷目前是否有可用的bit，如果有，則從cur位置開始尋找可用的bit，最後，因為外部都使用FindAndSet和Clear來操作BitMap，不可能發生有可用的0 bit小於cur的狀況，故斷言不會到達最後一行（有剩餘的0 bit但在大於等於cur的部分找不到）。

NumClear改為直接回傳numClear。修改SelfTest，將原先SelfTest中隨機Mark bit的測試案例改掉，因為除了FileSystem建構式中使用Mark(FreeMapSector)和Mark(DirectorySector)，其他都使用FindAndSet和Clear來操作BitMap，而且FreeMapSector為0、DirectorySector為1，並不存在隨機Mark bit的狀況，故以上調整能大幅增加BitMap的效能，且不違反目前的使用案例。

z. code/lib/bitmap.cc

```
Bitmap::Bitmap(int numItems)
{
    // ......
    numClear = numItems;
    cur = 0;
    // ......
}
```

```
void Bitmap::Mark(int which)
{
    ASSERT(which >= 0 && which < numBits);
    if (!Test(which))
    {
        cur = which;
        --numClear;
    }
    map[which / BitsInWord] |= 1 << (which % BitsInWord);
    ASSERT(Test(which));
}

void Bitmap::Clear(int which)
{
    ASSERT(which >= 0 && which < numBits);
    cur = min(cur, which);
    if (Test(which))
    {
        ++numClear;
    }
    map[which / BitsInWord] &= ~(1 << (which % BitsInWord));
    ASSERT(!Test(which));
}

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

int Bitmap::NumClear() const
```

```
{
    return numClear;
}

void Bitmap::SelfTest()
{
    int i;

    ASSERT(numBits >= BitsInWord); // bitmap must be big enough

    ASSERT(NumClear() == numBits); // bitmap must be empty
    ASSERT(FindAndSet() == 0);
    Mark(1);
    ASSERT(Test(0) && Test(1));

    ASSERT(FindAndSet() == 2);
    Clear(0);
    Clear(1);
    Clear(2);

    for (i = 0; i < numBits; i++)
    {
        Mark(i);
    }
    ASSERT(FindAndSet() == -1); // bitmap should be full!
    for (i = 0; i < numBits; i++)
    {
        Clear(i);
    }
}
```

使用以下指令執行測試腳本，能在FS_fileheader_cout0.log中看到不同file header在硬碟和在記憶體所占用的大小。

```
bash FS_fileheader.sh > FS_fileheader_cout0.log 2> FS_fileheader_cerr0.log
```

aa. Bonus測試腳本（位於~/NachOS-4.0_MP4/code/test/FS_fileheader.sh）

```
../build.linux/nachos -f
../build.linux/nachos -mkdir /t0
../build.linux/nachos -mkdir /t1
../build.linux/nachos -cp num_100.txt /t0/f1
```

```
../build.linux/nachos -cp num_1000.txt /t0/f2
../build.linux/nachos -mkdir /t0/aa
../build.linux/nachos -cp num_100.txt /t0/aa/f1
../build.linux/nachos -cp 64MB.txt /t0/aa/64MB
../build.linux/nachos -l /
echo ===================
../build.linux/nachos -lr /
../build.linux/nachos -h /t0/aa/64MB
```

Part III - Feedback

本次的實作和除錯的時間都遠超前面3個報告，確實學到很多東西，如果有更多的時間，應該還有很多可以優化的部分(註解文件、變數和方法命名、報告用字)，但很可惜，將Bonus全做完後，目前剩下44個小時要完成Pthread實作和報告，明天還要出門投票，十分極限。