OS Team22 MP4 Report

成員：郭蕙綺、趙仰生

貢獻：50% 50%

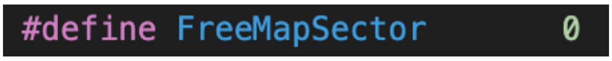
Part1. Understanding NachOS file system

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

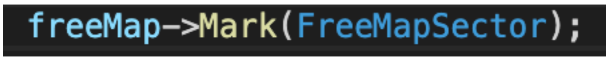
Manage: NachOS在bitmap.h裡定義一個Bitmap的class，在class裡宣告“*map”並以此來表示sector的使用狀況，更改map裡面的值則是利用“Mark()、Clear()”來做修改。

Find: 找尋free block則是利用“FindAndSet()”來做，呼叫這個function後會回傳clear bit的number，並set the bit，如果沒有任何clear bit的話則回傳-1。

Info stored: 在fliesys.cc的FileSystem::FileSystem()裡將map放到sector0。



```
#define FreeMapSector           0
```



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

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

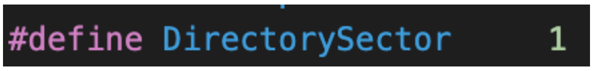
在dish.h裡頭宣告，總共有32個track，每個track有32個sector，而一個sector又是128Bytes，所以maximum disk size = 32 * 32 * 128 (Bytes) = 2**17 (Bytes) = 128KB。

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

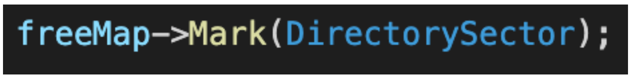
首先，Directory.h裡面maintain一個table，其中每個entry存取name、sector、inUse等資訊，sector為該file的header存在raw disk的sector位置，而Add和Remove用來新增或移除在目錄裡的file。

接著，filesys.cc裡面建立directoryFile，在constructor裡面new一個directory紀錄哪個sector有被使用，並且new一個dirHdr為了在disk存放directory file的資料，利用freeMap在directory上標記sector 1被file header使用，利用Allocate為file header dirHdr配置data blocks，利用WriteBack將dirHdr寫回去DirectorySector，利用OpenFile打開directoryFile並把directory寫進去，而只要directoryFile有被改變，就要利用WriteBack把它寫回去，最後要delete freeMap、directory、mapHdr以及dirHdr。

Info stored: 在fliesys.cc的FileSystem::FileSystem()裡將Directory放到sector1。



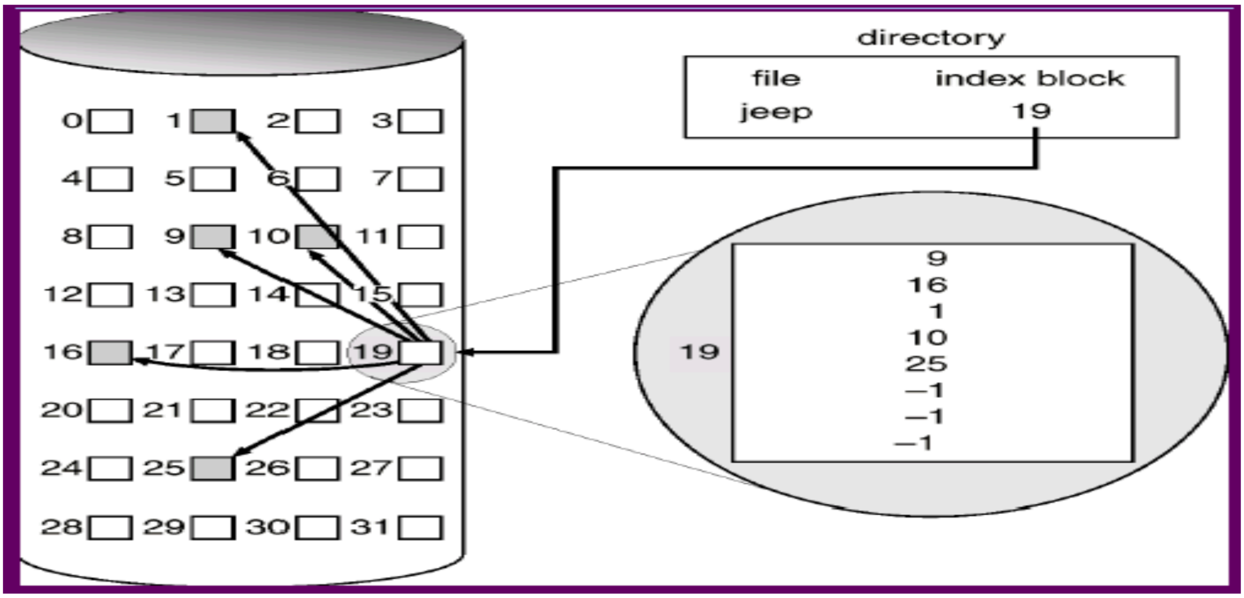
```
#define DirectorySector       1
```



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

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

Info stored in inode: numBytes(Number of bytes in the file)、numSectors(Number of data sectors in the file)、dataSector[](Disk sector numbers for each data block in the file)。
Disk allocation scheme: Indexed allocation with direct blocks。



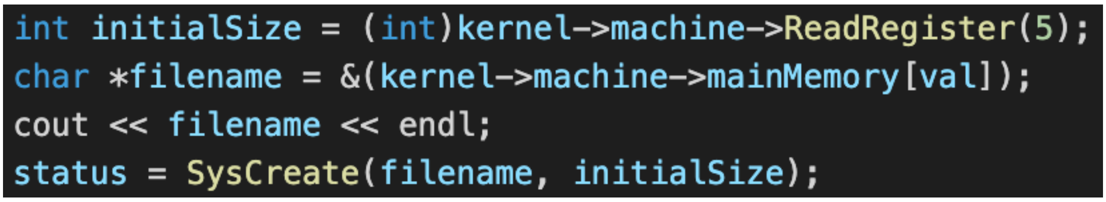
5. Why a file is limited to 4KB in the current implementation?

在dish.h裡頭宣告一個sector是128Bytes，而在filehdr.h裡頭define，NumDirect = ((SectorSize - 2 × sizeof(int)) / sizeof(int)) = (128 - 2 × 4) / 4 = 30，所以MaxFileSize = 30 × 128 = 3840 B = 3.75 KB。

Part II. Modify the file system code to support file I/O system call and larger file size

(1) Combine your MP1 file system call interface with NachOS FS

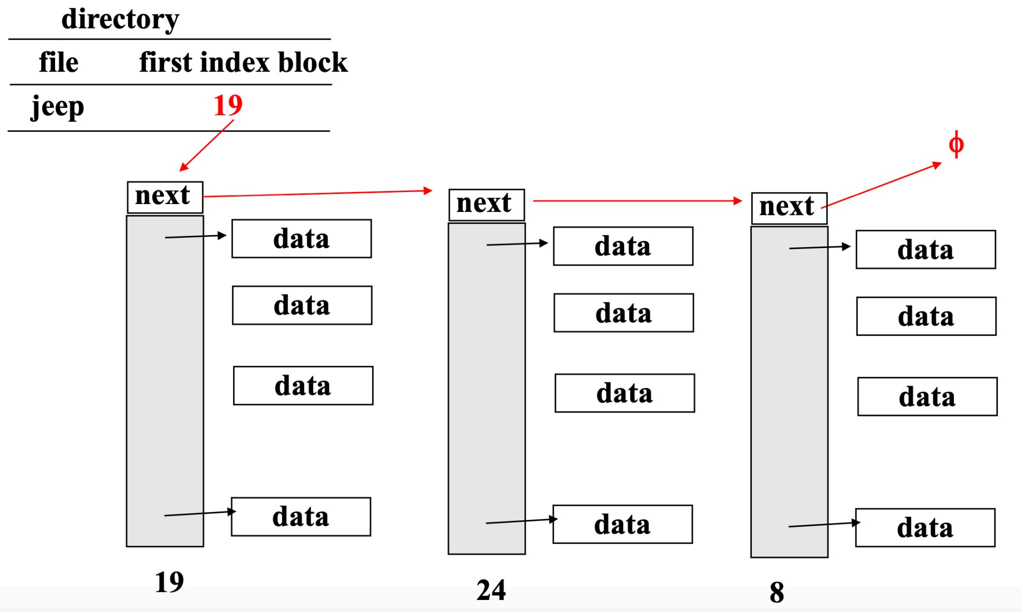
- 這部分我們基本上均套用我們在MP1的實作，只更動“Create()”，在exception.cc讀register5的值，並將值傳入SysCreate()，之後在實作關於Create system call的function都加入initialSize。



```
int initialSize = (int)kernel->machine->ReadRegister(5);
char *filename = &(kernel->machine->mainMemory[val]);
cout << filename << endl;
status = SysCreate(filename, initialSize);
```

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

- 在這部分我們使用“Linked Indexed Scheme”來實作(如圖)，共修改了在filesys檔案夾中的filehdr.h、filehdr.cc，以下一一說明。



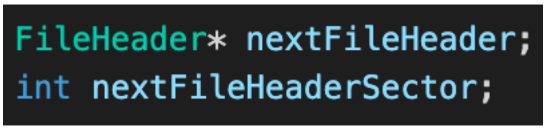
directory

file	first index block
jeep	19

1. filehdr.h:

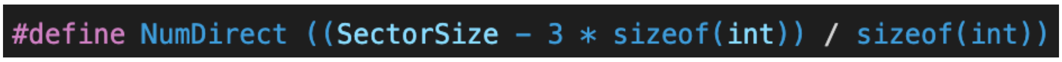
   - class FileHeader:我們在這裡新增了兩個在private的data structure，nextFileHeader(用來記錄下一個linked file的

pointer)、nextFileHeaderSector(用來記錄下一個fileHeader所在的sector)。



```
FileHeader* nextFileHeader;
int nextFileHeaderSector;
```

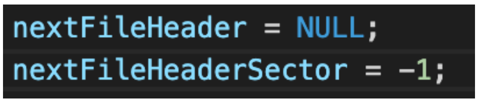
- Global:將原本的“- 2 * sizeof(int)”改為“- 3 * sizeof(int)”，原本只有扣掉2個integer的空間，由於我們多宣告了int nextFileHeaderSector所以需要多扣一個integer的空間。



```
#define NumDirect ((SectorSize - 3 * sizeof(int)) / sizeof(int))
```

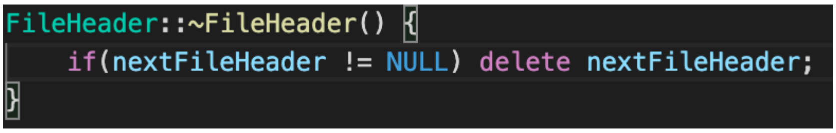
2. filehdr.cc:

- FileHeader::FileHeader():將我們新增的兩個data structure做初始化。



```
nextFileHeader = NULL;
nextFileHeaderSector = -1;
```

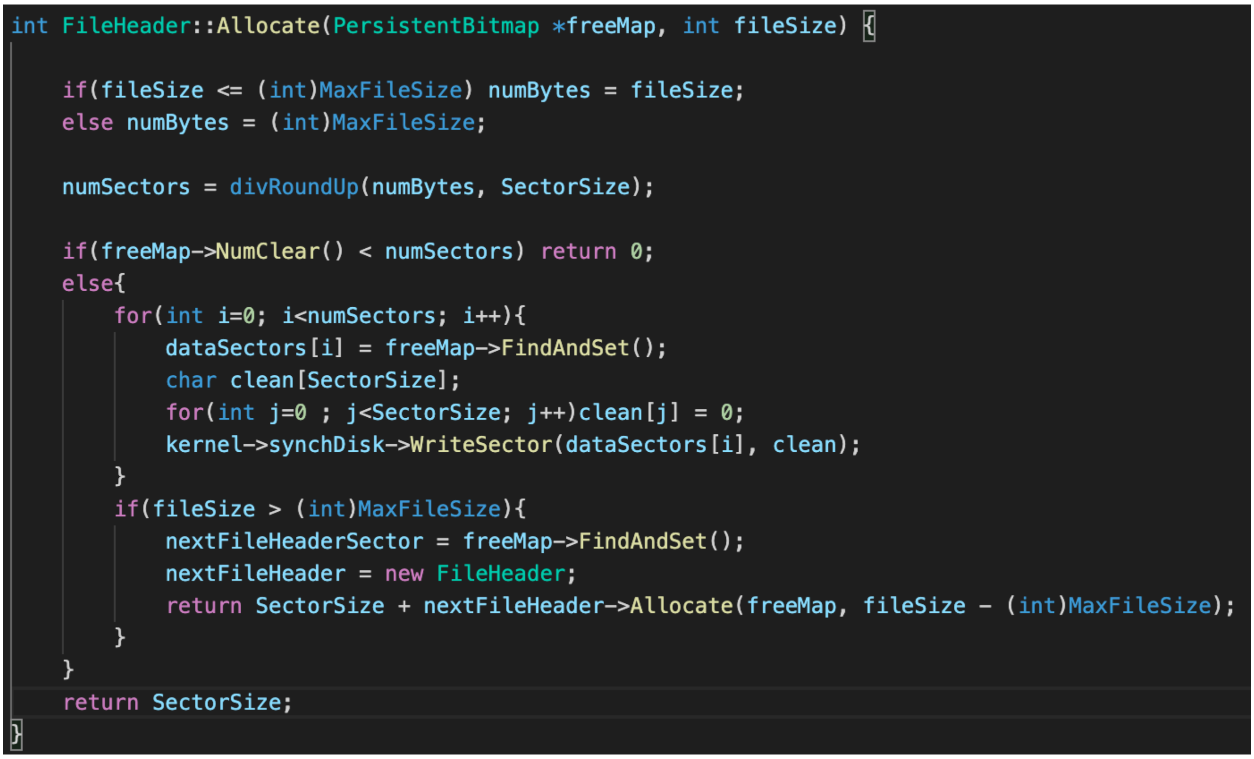
- FileHeader::~FileHeader():為了避免造成memory leakage，我們在這裡做nextFileHeader的delete。



```
FileHeader::~FileHeader() {
    if(nextFileHeader != NULL) delete nextFileHeader;
}
```

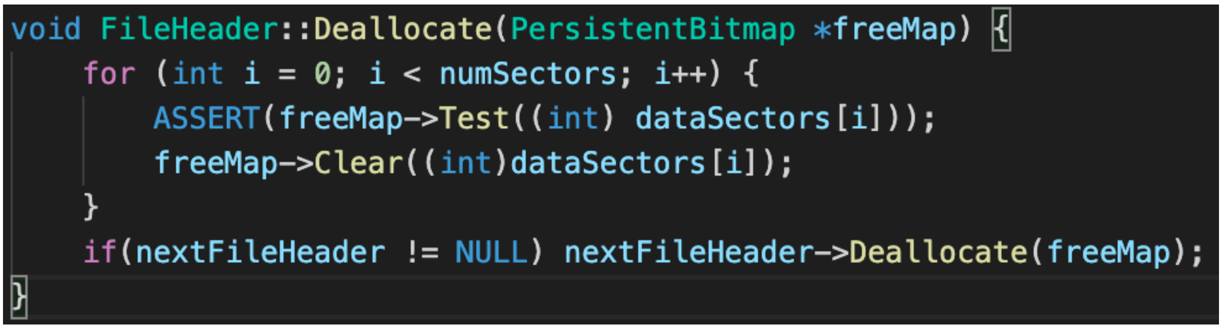
- int FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize):原本這個function的型態為布林值，但由於方便後面算出使用的byte總數，在這裡改為回傳integer，function裡首先判斷fileSize有沒有超過MaxFileSize，如果超過則讓numbytes = MaxFileSize(超過的後面會Allocate)，沒有超過則numbytes = fileSize，接著用divRoundUp算出data需要的sector數，如果sector數不夠則return 0表示無法Allocate，足夠則用一個for-loop跑numSector次，找到freeMap裡可用的Sector id記錄到其dataSector的陣列後將該sector mark起來，表示已用過，並將該sector清空(寫回disk)，最後判斷如果還有剩餘的

data則到freeMap找到空的sector給nextFileHeaderSector，new一個新的FileHeader為nextFileHeader，並return SectorSize + nextFileHeader->Allocate(freeMap, filesize-MaxFileSize)，遞迴將剩餘的data用一樣的方法做Allocate並累加使用的bytes數。



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

    if(fileSize <= (int)MaxFileSize) numBytes = fileSize;
    else numBytes = (int)MaxFileSize;

    numSectors = divRoundUp(numBytes, SectorSize);

    if(freeMap->NumClear() < numSectors) return 0;
    else{
        for(int i=0; i<numSectors; i++){
            dataSectors[i] = freeMap->FindAndSet();
            char clean[SectorSize];
            for(int j=0 ; j<SectorSize; j++)clean[j] = 0;
            kernel->synchDisk->WriteSector(dataSectors[i], clean);
        }
        if(fileSize > (int)MaxFileSize){
            nextFileHeaderSector = freeMap->FindAndSet();
            nextFileHeader = new FileHeader;
            return SectorSize + nextFileHeader->Allocate(freeMap, fileSize - (int)MaxFileSize);
        }
    }
    return SectorSize;
}
```

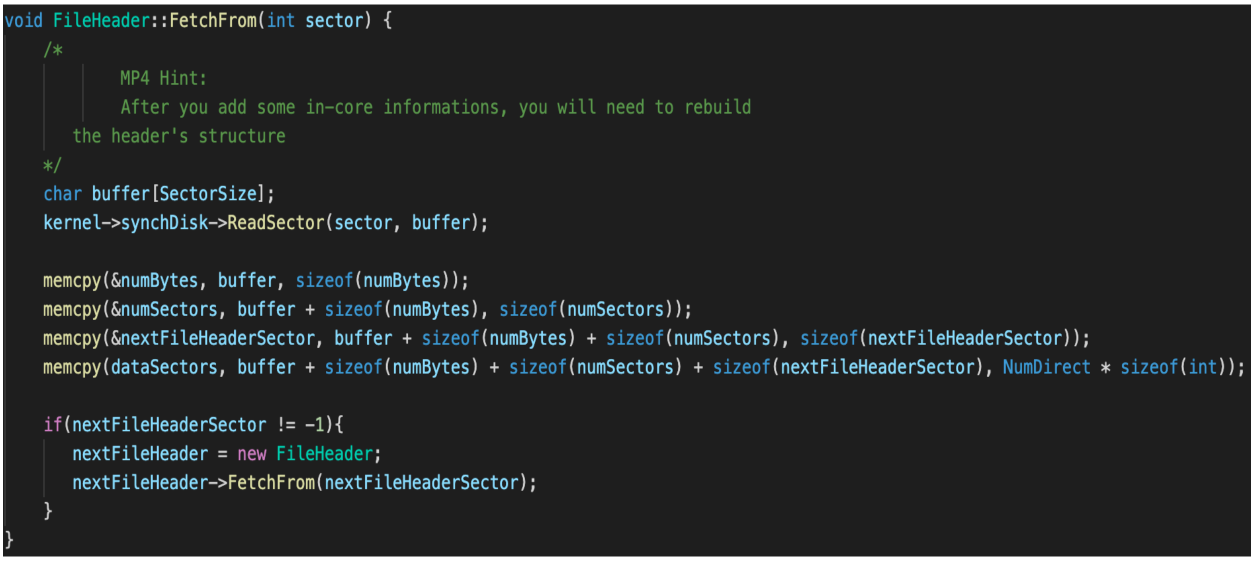
- FileHeader::Deallocate(PersistentBitmap *freemap):這個function與原本的大致相同，只差在最後用遞迴Deallocate 每個nextFileHeader。



```
void FileHeader::Deallocate(PersistentBitmap *freeMap) {
    for (int i = 0; i < numSectors; i++) {
        ASSERT(freeMap->Test((int) dataSectors[i]));
        freeMap->Clear((int)dataSectors[i]);
    }
    if(nextFileHeader != NULL) nextFileHeader->Deallocate(freeMap);
}
```

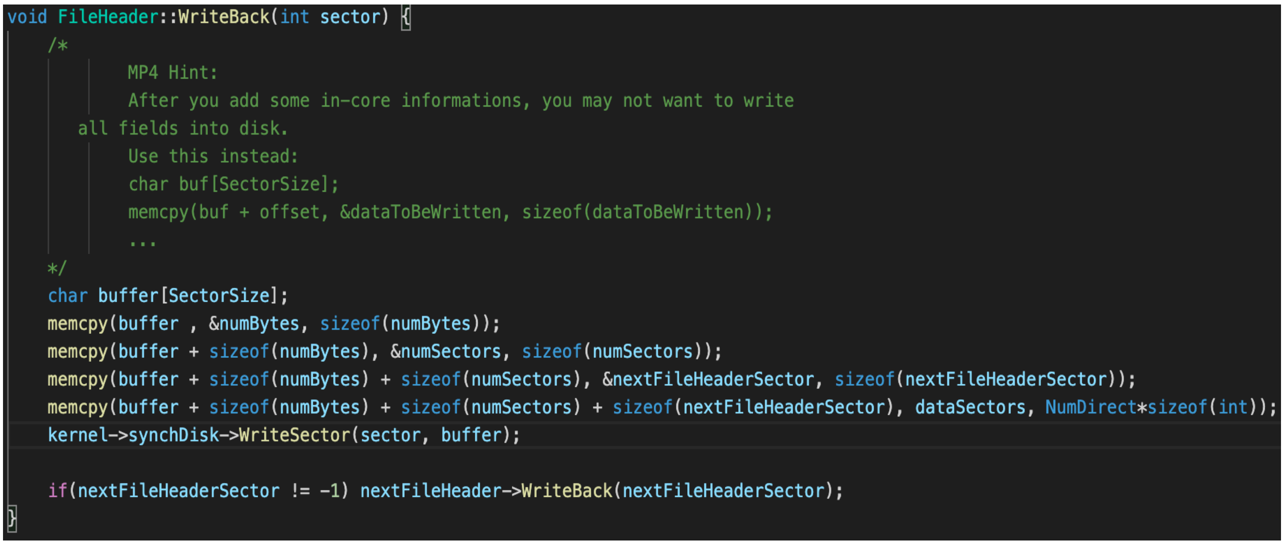
- FileHeader::FetchFrom(int sector):這個function將傳入的sector裡fileHeader的content從disk取出，我們先將sector的內容

read到buffer上，再將裡面的content 利用memcpy copy到其對應位置，最後用遞迴將所有FileHeader的資料取出。



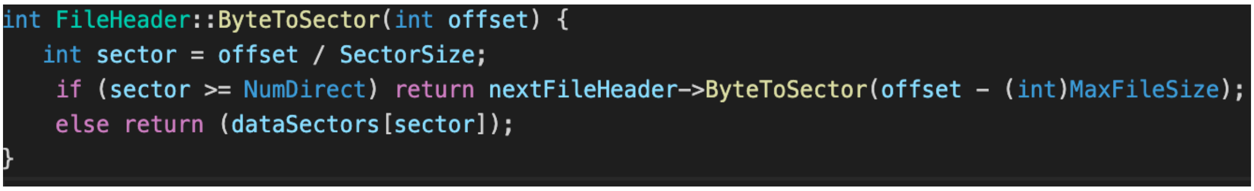
```
void FileHeader::FetchFrom(int sector) {
    /*
            MP4 Hint:
            After you add some in-core informations, you will need to rebuild
        the header's structure
    */
    char buffer[SectorSize];
    kernel->synchDisk->ReadSector(sector, buffer);

    memcpy(&numBytes, buffer, sizeof(numBytes));
    memcpy(&numSectors, buffer + sizeof(numBytes), sizeof(numSectors));
    memcpy(&nextFileHeaderSector, buffer + sizeof(numBytes) + sizeof(numSectors), sizeof(nextFileHeaderSector));
    memcpy(dataSectors, buffer + sizeof(numBytes) + sizeof(numSectors) + sizeof(nextFileHeaderSector), NumDirect * sizeof(int));

    if(nextFileHeaderSector != -1){
        nextFileHeader = new FileHeader;
        nextFileHeader->FetchFrom(nextFileHeaderSector);
    }
}
```

- FileHeader::WriteBack(int sector):這個function將更改過的fileHeader的contents寫回disk，做法和FetchFrom()相似，先將contents放到buffer上，再將其寫到disk上，最後用遞迴的方式將所有FileHeader的contents寫到disk。



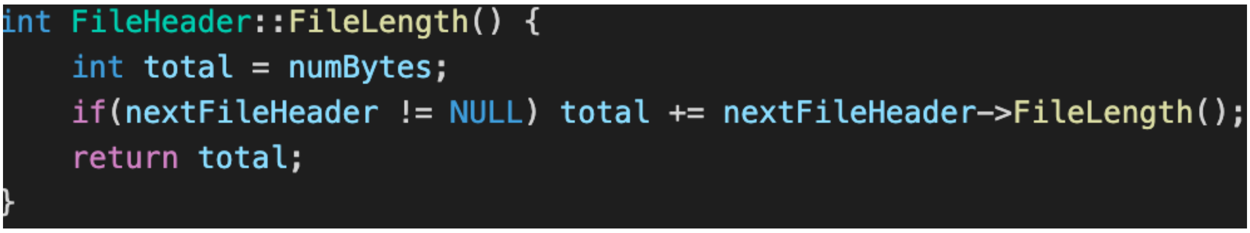
```
void FileHeader::WriteBack(int sector) {
    /*
            MP4 Hint:
            After you add some in-core informations, you may not want to write
        all fields into disk.
            Use this instead:
            char buf[SectorSize];
            memcpy(buf + offset, &dataToBeWritten, sizeof(dataToBeWritten));
            ...
    */
    char buffer[SectorSize];
    memcpy(buffer , &numBytes, sizeof(numBytes));
    memcpy(buffer + sizeof(numBytes), &numSectors, sizeof(numSectors));
    memcpy(buffer + sizeof(numBytes) + sizeof(numSectors), &nextFileHeaderSector, sizeof(nextFileHeaderSector));
    memcpy(buffer + sizeof(numBytes) + sizeof(numSectors) + sizeof(nextFileHeaderSector), dataSectors, NumDirect*sizeof(int));
    kernel->synchDisk->WriteSector(sector, buffer);

    if(nextFileHeaderSector != -1) nextFileHeader->WriteBack(nextFileHeaderSector);
}
```

- FileHeader::ByteToSector(int offset):這個function要找到指定byte所對應的sector，我們先算出sector，如果sector數大於NumDirect數(表示不在這個fileHeader所屬的block)，則往後找，直到在當個block時return dataSector[sector]。



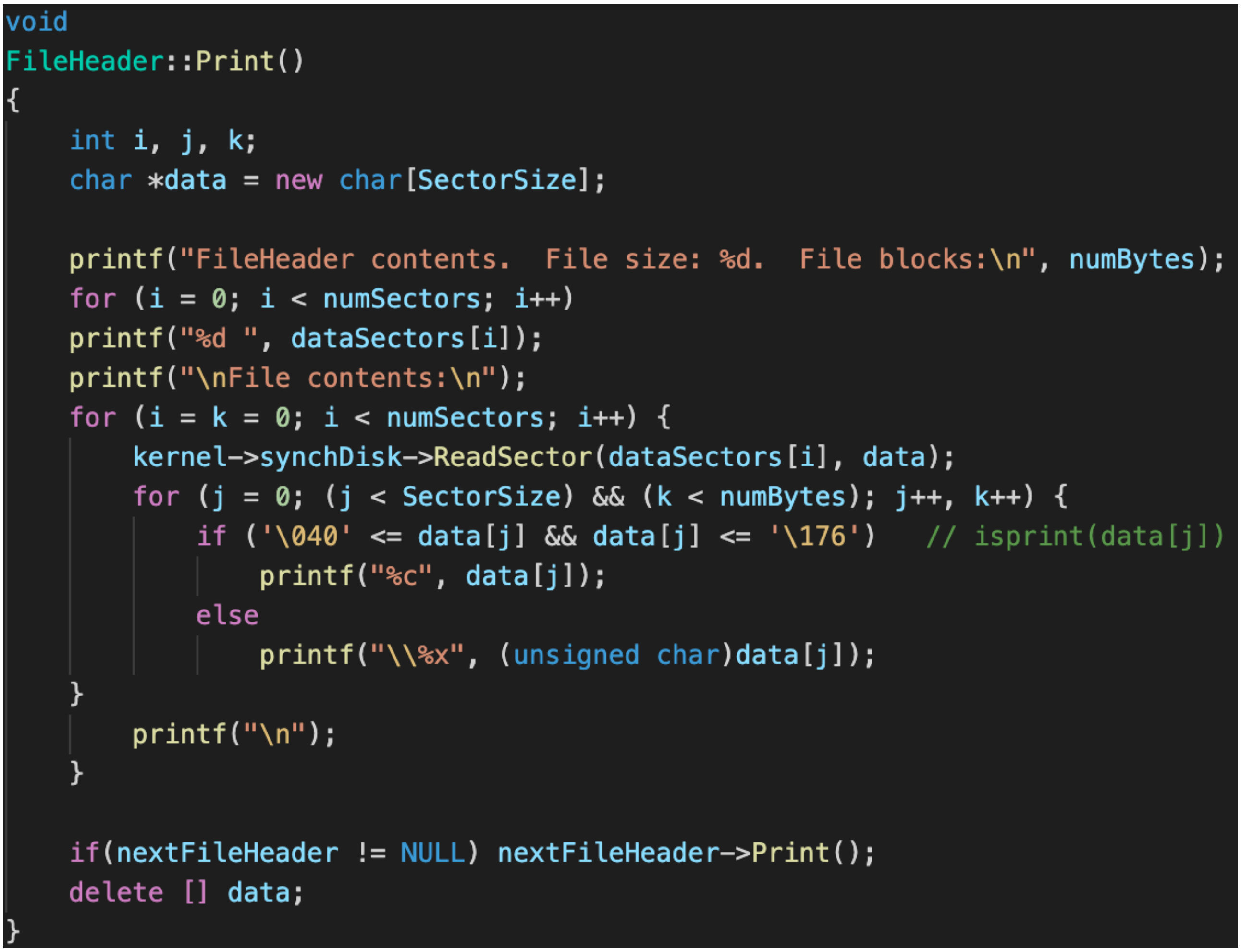
```
int FileHeader::ByteToSector(int offset) {
   int sector = offset / SectorSize;
    if (sector >= NumDirect) return nextFileHeader->ByteToSector(offset - (int)MaxFileSize);
    else return (dataSectors[sector]);
}
```

- FileHeader::FileLength():這個function回傳file的bytes數，用一個total來記，遞迴累加numBytes後return。



```
int FileHeader::FileLength() {
    int total = numBytes;
    if(nextFileHeader != NULL) total += nextFileHeader->FileLength();
    return total;
}
```

- FileHeader::Print():這個function用來print file header的contents還有其data blocks指到的contents。Print方式和原本相同，只有在最後的地方利用遞迴印出所有file header的contents。



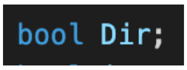
```
void
FileHeader::Print()
{
    int i, j, k;
    char *data = new char[SectorSize];

    printf("FileHeader contents.  File size: %d.  File blocks:\n", numBytes);
    for (i = 0; i < numSectors; i++)
    printf("%d ", dataSectors[i]);
    printf("\nFile contents:\n");
    for (i = k = 0; i < numSectors; i++) {
        kernel->synchDisk->ReadSector(dataSectors[i], data);
        for (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");
    }

    if(nextFileHeader != NULL) nextFileHeader->Print();
    delete [] data;
}
```

Part III. Modify the file system code to support subdirectory

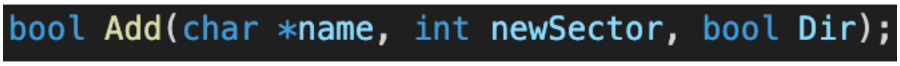
(1) Implement the subdirectory structure

- 實作subdirectory的部分我們先修改了directory.h、directory.cc來support subdirectory，再修改了main.cc、filesys.h、filesys.cc來support subdirectory的運作，以下會一一說明。

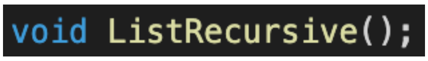
1. directory.h:
   - class DirectoryEntry:我們在public新增一個“bool Dir”來代表這個entry存放的是否為Directory，還有將Add function多傳入一個“bool Dir”判斷即將Add的Entry是否為directory，最後再新增一個ListRecursive()的function，用來Recursively list the file/directory in a directory。



```
bool Dir;
```

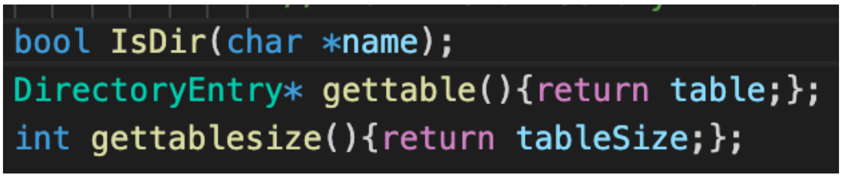


```
bool Add(char *name, int newSector, bool Dir);
```



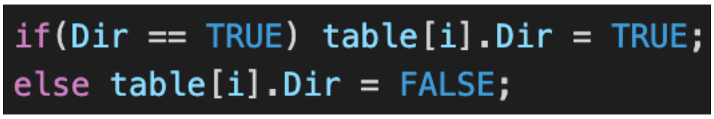
```
void ListRecursive();
```

   - class Directory:我們在public新增三個function，bool isDir(char* name)用來判斷傳入的name是否為directory，DirectoryEntry* gettable()用來回傳在private的table(filesys.cc需要用)，int getableSize()用來回傳在private的tableSize。



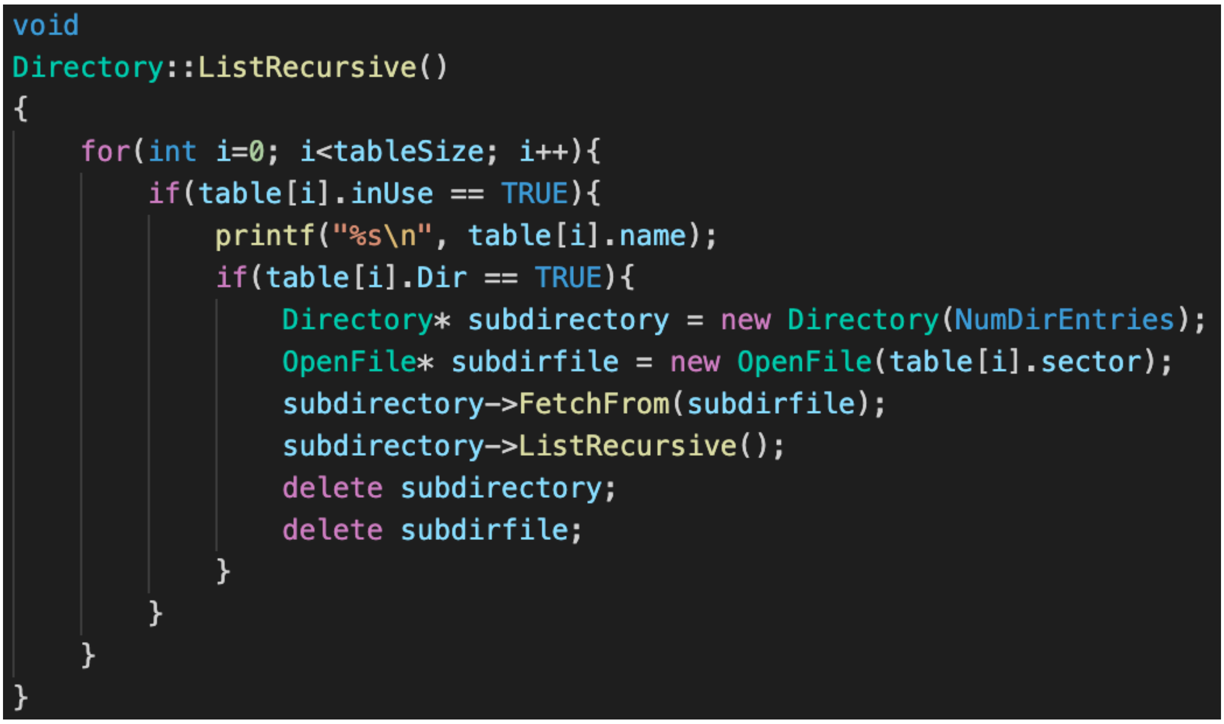
```
bool IsDir(char *name);
DirectoryEntry* gettable(){return table;};
int gettablesize(){return tableSize;};
```

2. directory.cc:
   - Directory::Add(char *name, int newSector, bool isDir):我們在Add時多判斷是否為directory，把該table位置的“Dir”設好。



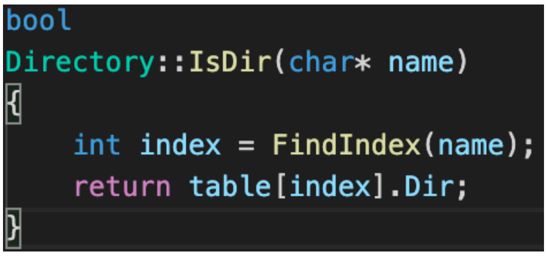
```
if(Dir == TRUE) table[i].Dir = TRUE;
else table[i].Dir = FALSE;
```

- Directory::ListRecursive():用一個for-loop跑tableSize次，如果是有用到的table印出filename，再判斷如果該file為subDirectory的話，new一個directory給subDirectory，打開該subDirectory的file，利用FetchFrom讀取file的內容後遞迴List出file name。



```
void
Directory::ListRecursive()
{
    for(int i=0; i<tableSize; i++){
        if(table[i].inUse == TRUE){
            printf("%s\n", table[i].name);
            if(table[i].Dir == TRUE){
                Directory* subdirectory = new Directory(NumDirEntries);
                OpenFile* subdirfile = new OpenFile(table[i].sector);
                subdirectory->FetchFrom(subdirfile);
                subdirectory->ListRecursive();
                delete subdirectory;
                delete subdirfile;
            }
        }
    }
}
```

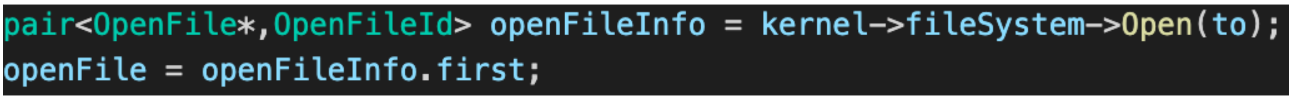
- Directory::IsDir(char *name):這個function回傳傳入的name是否為directory。



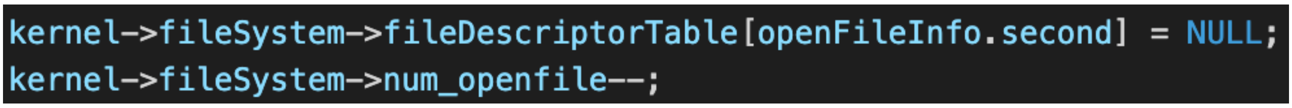
```
bool
Directory::IsDir(char* name)
{
    int index = FindIndex(name);
    return table[index].Dir;
}
```

3. main.cc:

- Copy(char *from, char *to):由於我們有更改openFile的型態，所以在create與close NachOS file做調整。

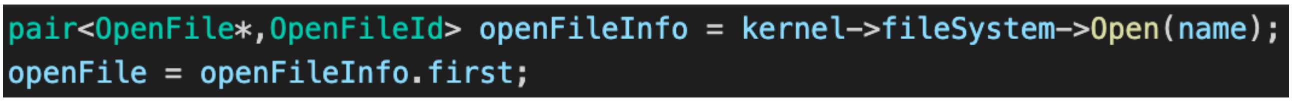


```
pair<OpenFile*,OpenFileId> openFileInfo = kernel->fileSystem->Open(to);
openFile = openFileInfo.first;
```

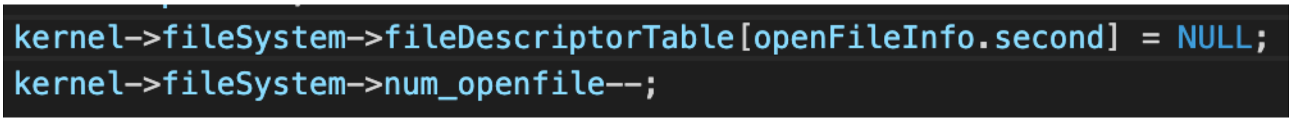


```
kernel->fileSystem->fileDescriptorTable[openFileInfo.second] = NULL;
kernel->fileSystem->num_openfile--;
```

- Print(char *name):與Copy一樣，做OpenFile的調整。

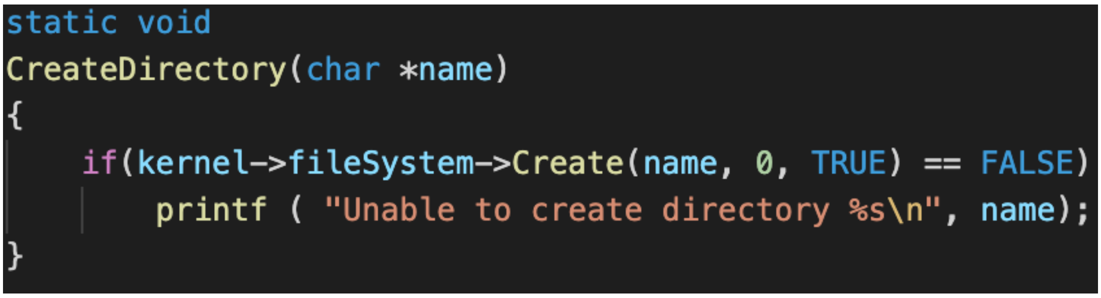


```
pair<OpenFile*,OpenFileId> openFileInfo = kernel->fileSystem->Open(name);
openFile = openFileInfo.first;
```



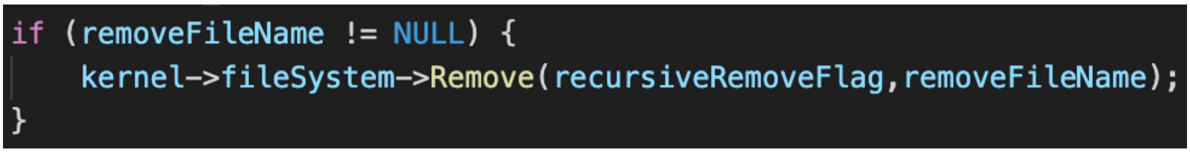
```
kernel->fileSystem->fileDescriptorTable[openFileInfo.second] = NULL;
kernel->fileSystem->num_openfile--;
```

- CreateDirectory(char *name):在這邊實作create directory的部分，我們call fileSystem的create，第三個傳入的變數為true，代表現在要create的為directory。

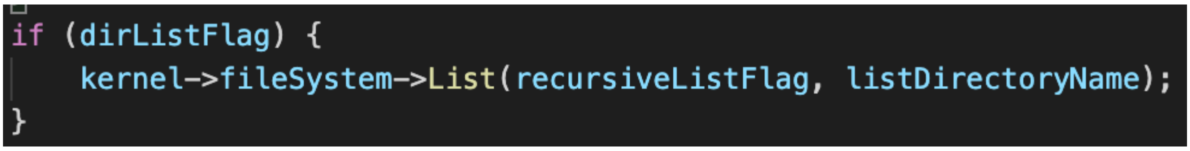


```
static void
CreateDirectory(char *name)
{
    if(kernel->fileSystem->Create(name, 0, TRUE) == FALSE)
        printf ( "Unable to create directory %s\n", name);
}
```

- int main(int argc, char **argv):在remove與list時多傳入recursiveRemoveFlag與recursiveListFlag。



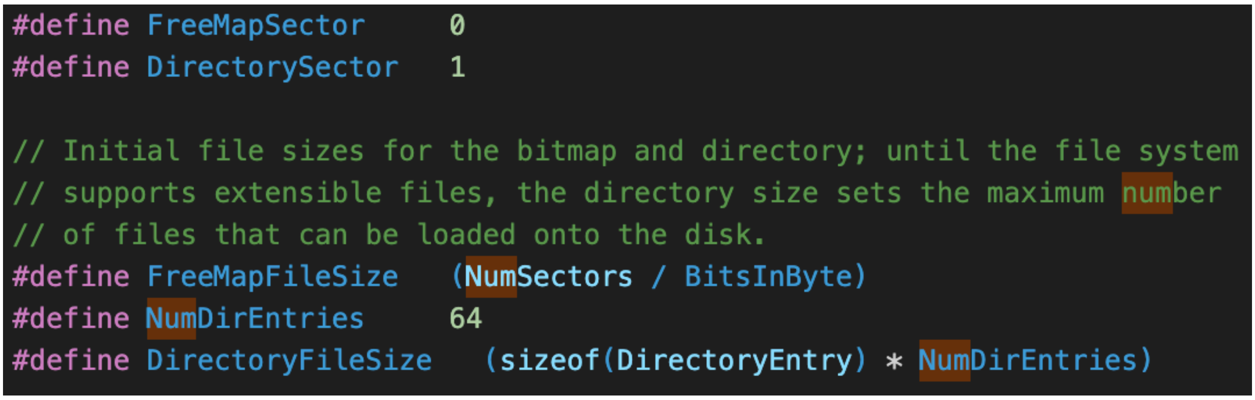
```
if (removeFileName != NULL) {
    kernel->fileSystem->Remove(recursiveRemoveFlag,removeFileName);
}
```



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

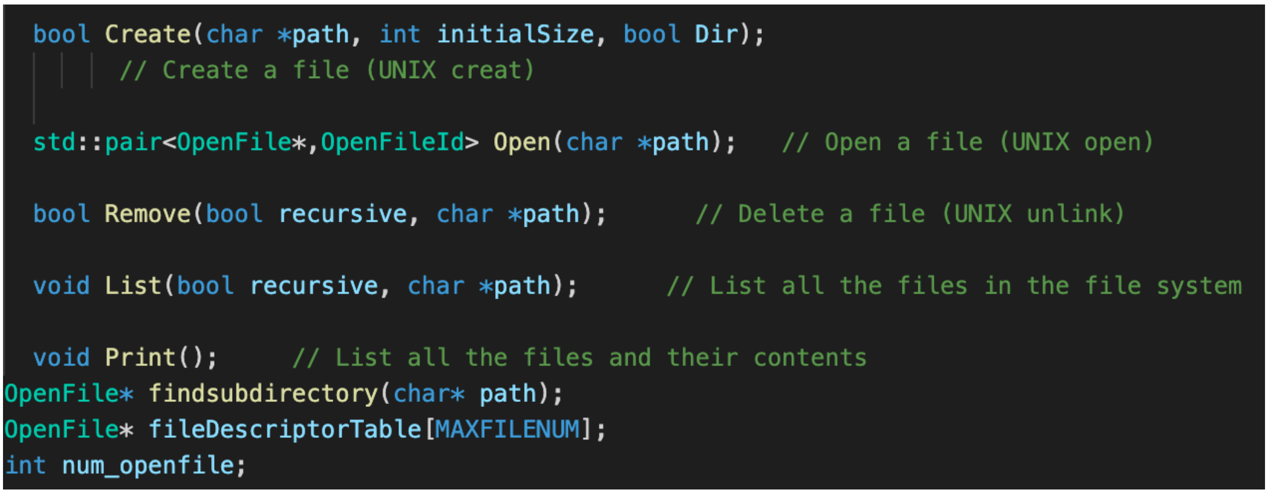
4. filesys.h:

- Global:將下圖的#define從cc檔放到h檔，方便其他cc檔使用。



```
#define FreeMapSector       0
#define DirectorySector     1

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

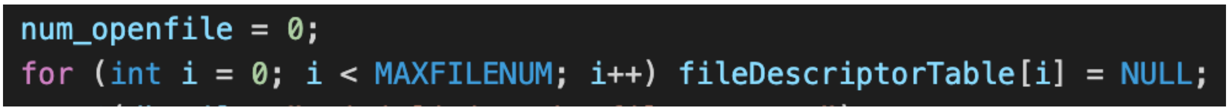
- class FileSystem:在Create()多傳入一個“bool Dir”，判斷create的file是否為Directory；將Open()的回傳型態改為pair，方便讀取OpenFileID；Remove()與List()多傳入一個“bool recursive”，判斷是否需要遞迴Remove、List；建findsubdirectory function，回傳該檔案的前一層directory，且將path修改為file name；fileDescriptorTable[]為MP1的實作部分；num_openfile用來記錄open過的file個數。



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

  std::pair<OpenFile*,OpenFileId> Open(char *path);   // Open a file (UNIX open)

  bool Remove(bool recursive, char *path);      // Delete a file (UNIX unlink)

  void List(bool recursive, char *path);      // List all the files in the file system

  void Print();     // List all the files and their contents
OpenFile* findsubdirectory(char* path);
OpenFile* fileDescriptorTable[MAXFILENUM];
int num_openfile;
```

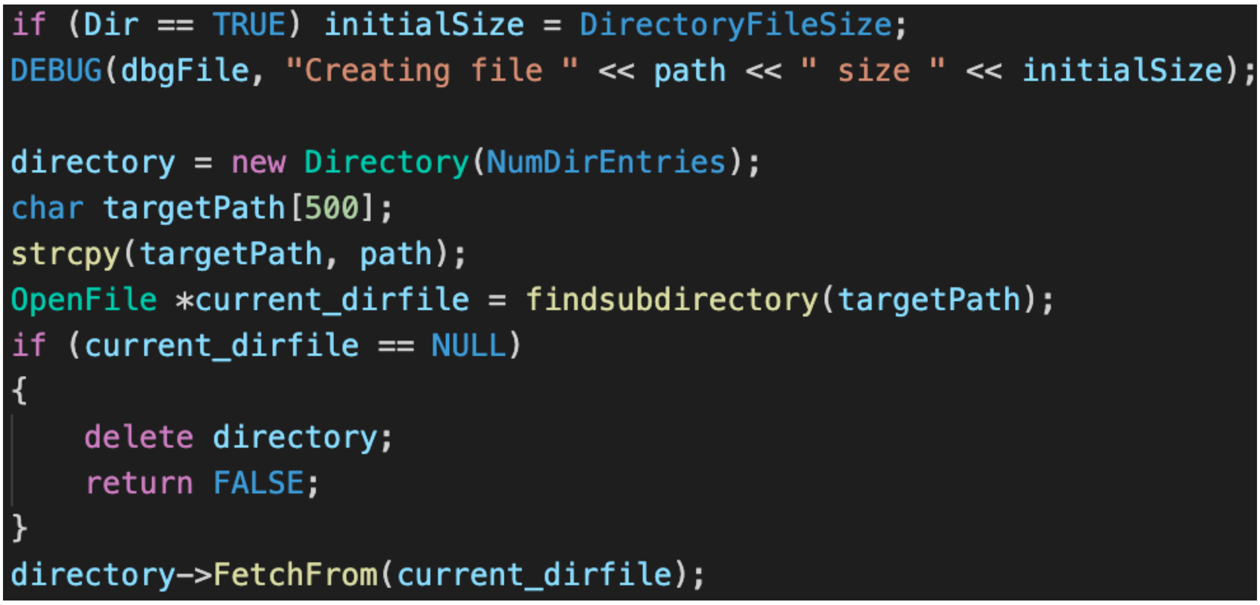
5. filesys.cc:

- FileSystem::FileSystem(bool format):我們在這裡初始化num_openfile、fileDescriptorTable[]。

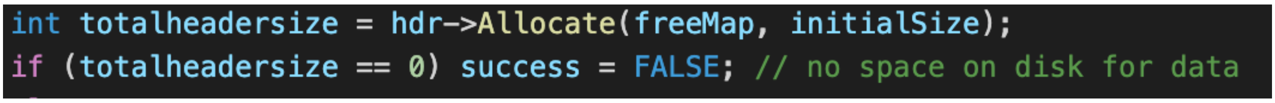


```
num_openfile = 0;
for (int i = 0; i < MAXFILENUM; i++) fileDescriptorTable[i] = NULL;
```

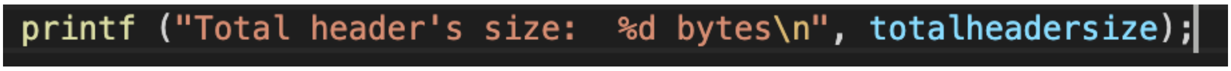
- FileSystem::Create(char *path, int initialSize, bool Dir):我們在這裡新增判斷式，如果要create的是Directory的話，將initialSize改為DirectoryFileSize，在之後Allocate需要用到；利用findsubdirectory來找到該檔案的前一層目錄，並將targetPath改為要create的file name，如果目錄不存在則直接return False，存在則去disk搬contents(FetchFrom)。接著則是稍微修改判斷是否可以Allocate的地方，由於我們已經將Allocate()的回傳值改為integer，所以我們在這裡用totalHeaderSize去接，如果回傳值為0代表Allocate失敗，最後印出totalHeaderSize。



```
if (Dir == TRUE) initialSize = DirectoryFileSize;
DEBUG(dbgFile, "Creating file " << path << " size " << initialSize);

directory = new Directory(NumDirEntries);
char targetPath[500];
strcpy(targetPath, path);
OpenFile *current_dirfile = findsubdirectory(targetPath);
if (current_dirfile == NULL)
{
    delete directory;
    return FALSE;
}
directory->FetchFrom(current_dirfile);
```

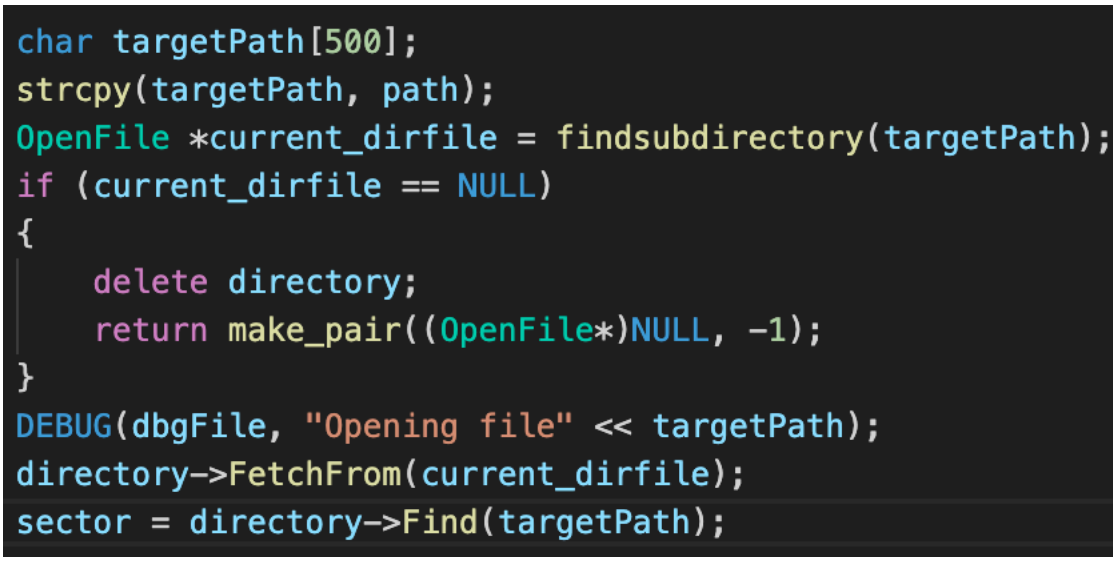


```
int totalheadersize = hdr->Allocate(freeMap, initialSize);
if (totalheadersize == 0) success = FALSE; // no space on disk for data
```

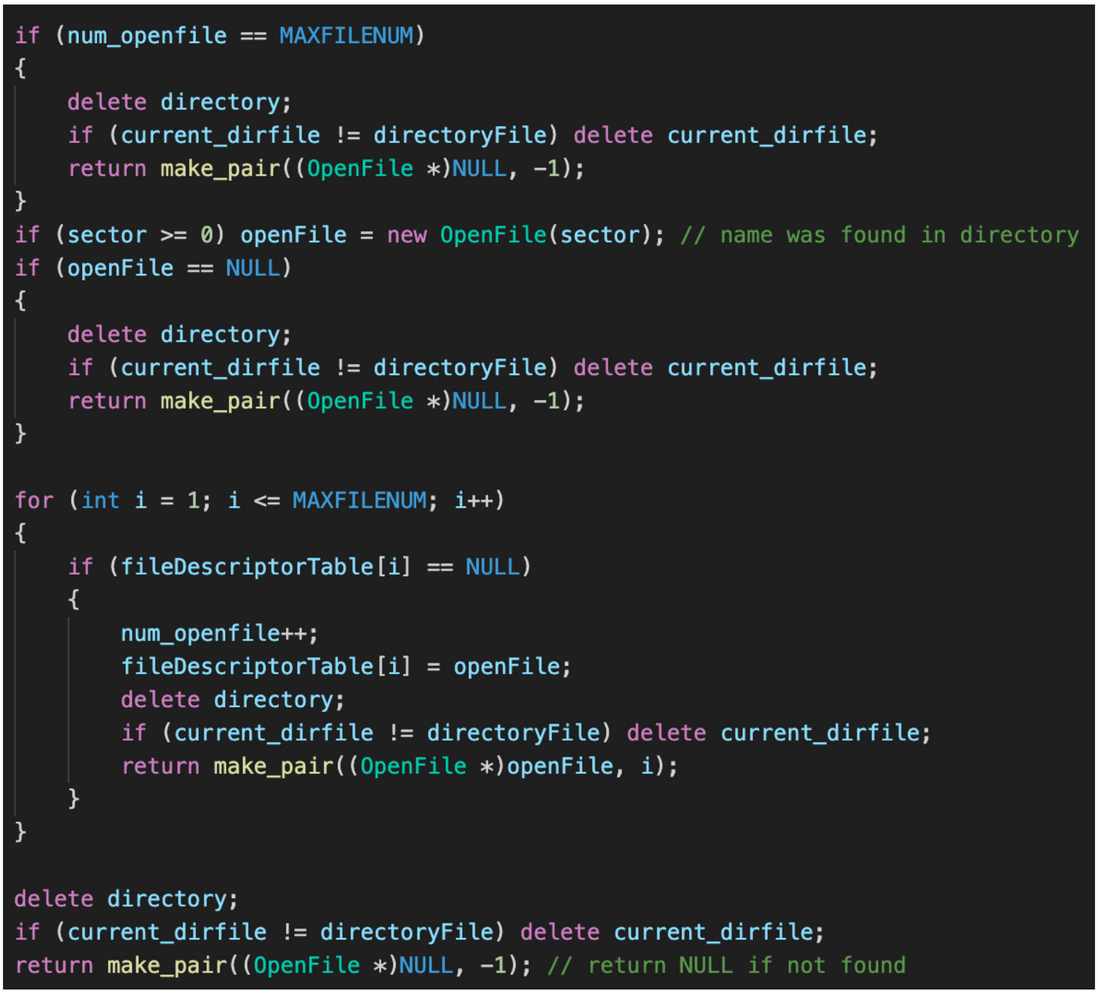


```
printf ("Total header's size:  %d bytes\n", totalheadersize);
```

- FileSystem::Open(char *path):先利用findsubdirectory找到前一層目錄(current_dirfile)並把target path改為file name，去搬目錄的content(FetchFrom)，再找到該file所在的sector。接著判斷如果開啟的file數已達MAXFILENUM則無法再開啟，如果可以成功開啟則用for-loop找尋空的fileDescriptor存放新的openFile後回傳OpenFile及其ID。

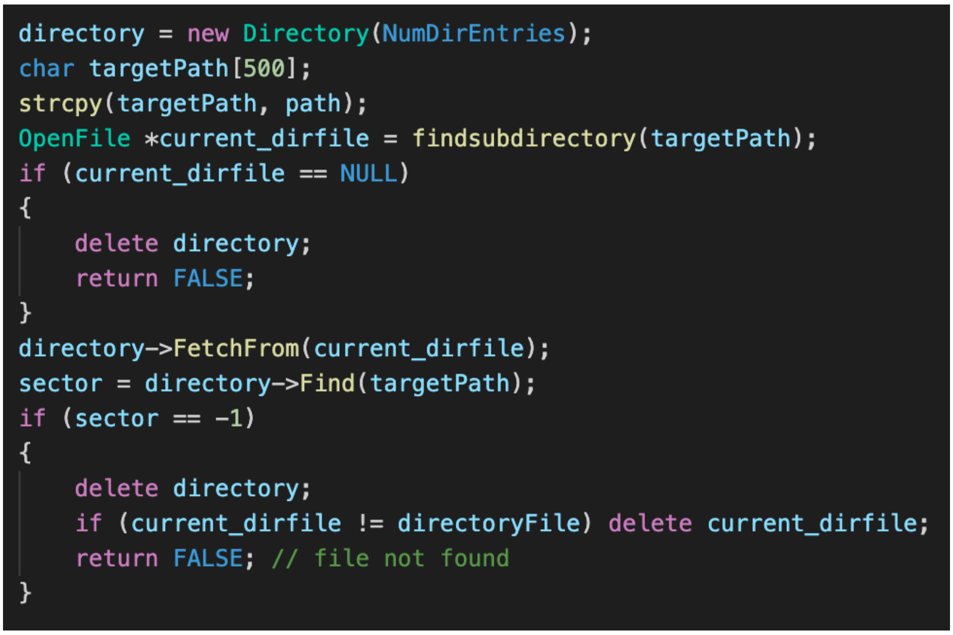


```
char targetPath[500];
strcpy(targetPath, path);
OpenFile *current_dirfile = findsubdirectory(targetPath);
if (current_dirfile == NULL)
{
    delete directory;
    return make_pair((OpenFile*)NULL, -1);
}
DEBUG(dbgFile, "Opening file" << targetPath);
directory->FetchFrom(current_dirfile);
sector = directory->Find(targetPath);
```



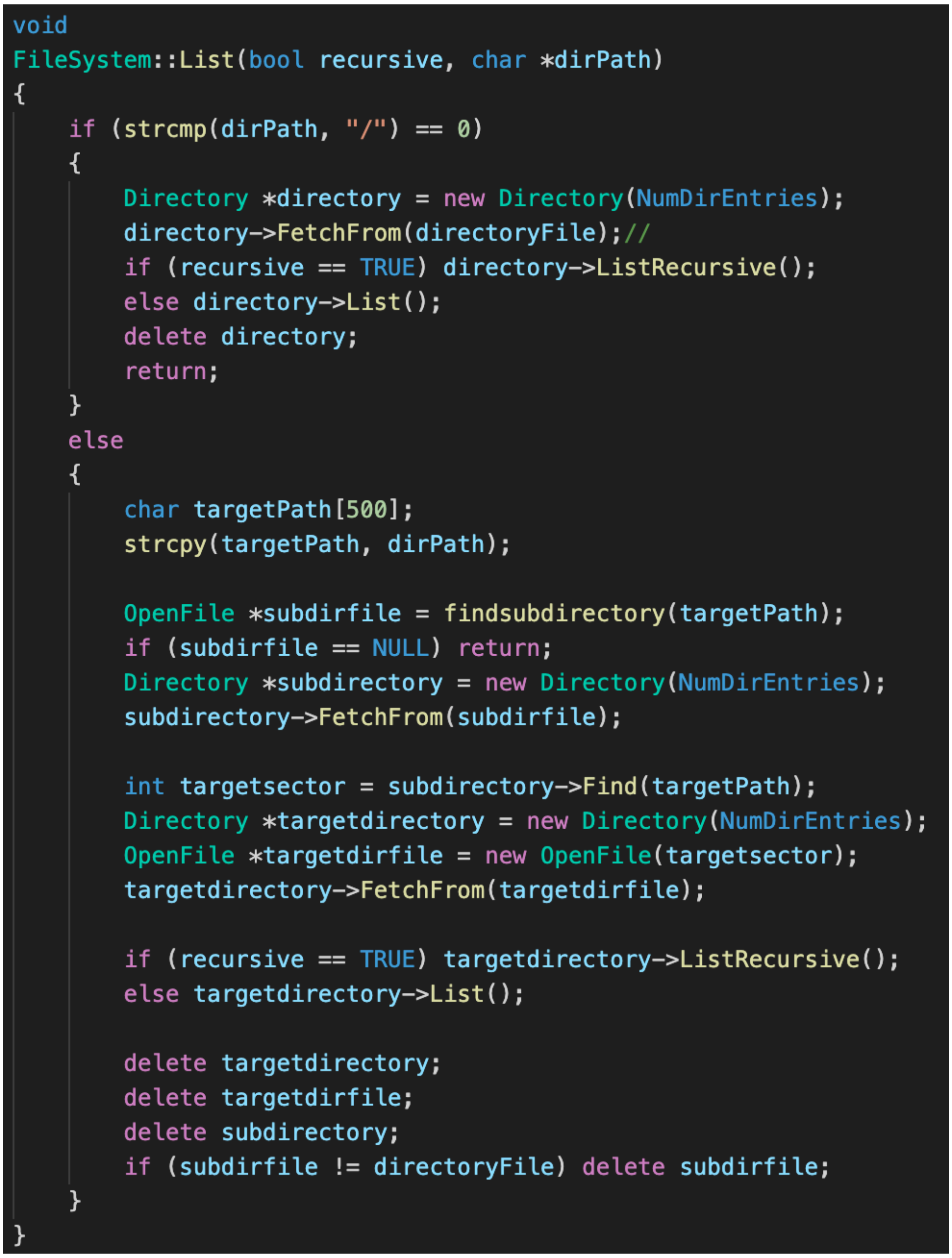
```
if (num_openfile == MAXFILENUM)
{
    delete directory;
    if (current_dirfile != directoryFile) delete current_dirfile;
    return make_pair((OpenFile *)NULL, -1);
}
if (sector >= 0) openFile = new OpenFile(sector); // name was found in directory
if (openFile == NULL)
{
    delete directory;
    if (current_dirfile != directoryFile) delete current_dirfile;
    return make_pair((OpenFile *)NULL, -1);
}

for (int i = 1; i <= MAXFILENUM; i++)
{
    if (fileDescriptorTable[i] == NULL)
    {
        num_openfile++;
        fileDescriptorTable[i] = openFile;
        delete directory;
        if (current_dirfile != directoryFile) delete current_dirfile;
        return make_pair((OpenFile *)openFile, i);
    }
}

delete directory;
if (current_dirfile != directoryFile) delete current_dirfile;
return make_pair((OpenFile *)NULL, -1); // return NULL if not found
```

- FileSystem::Remove(bool recursive, char *path):先利用findsubdirectory找到前一層目錄(current_dirfile)並把target path改為file name，去搬目錄的content(FetchFrom)，再找到該file所在的sector，fileHeader從sector fetch內容後Deallocate，清掉sector後將freeMap跟directory寫回disk。



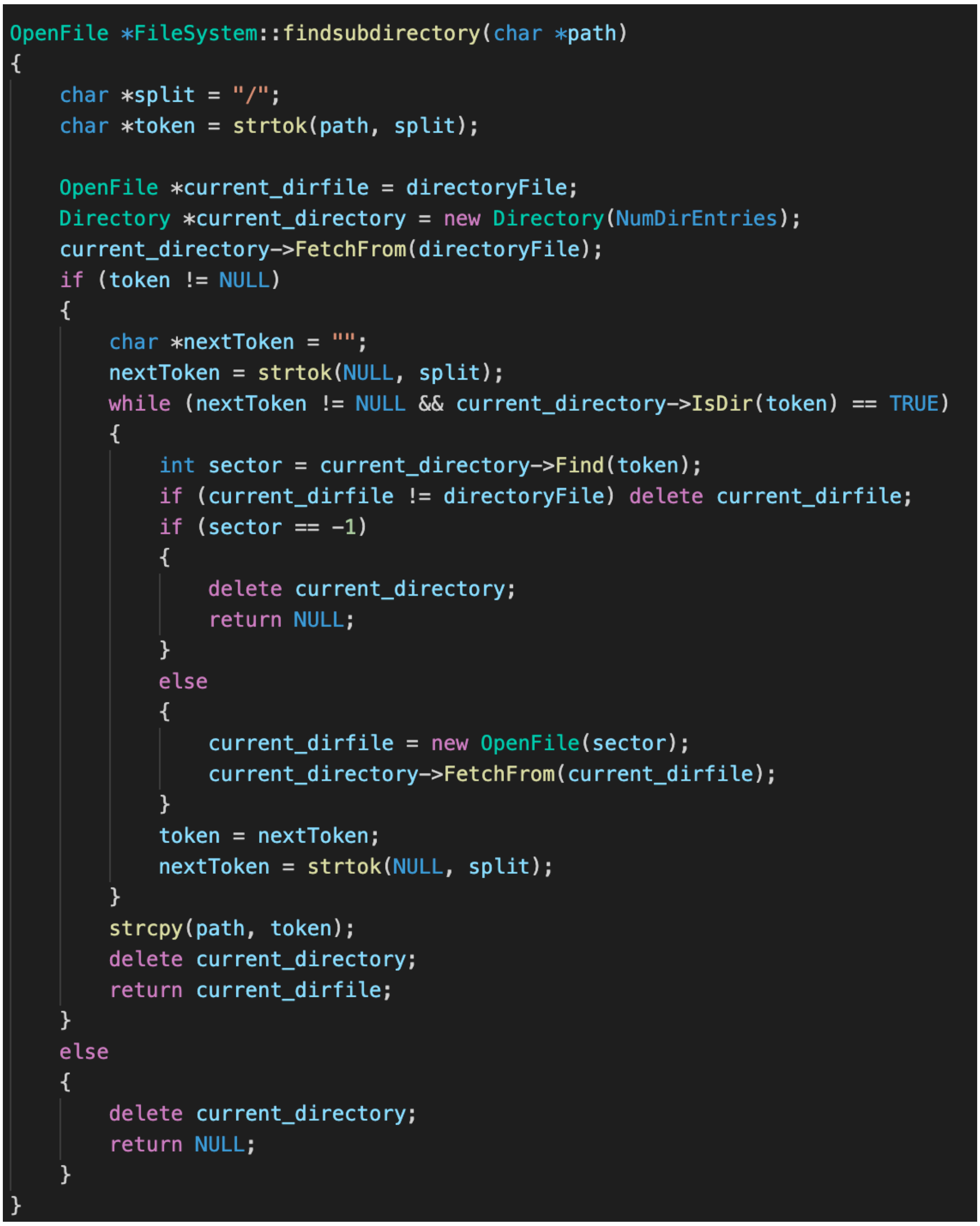
```
directory = new Directory(NumDirEntries);
char targetPath[500];
strcpy(targetPath, path);
OpenFile *current_dirfile = findsubdirectory(targetPath);
if (current_dirfile == NULL)
{
    delete directory;
    return FALSE;
}
directory->FetchFrom(current_dirfile);
sector = directory->Find(targetPath);
if (sector == -1)
{
    delete directory;
    if (current_dirfile != directoryFile) delete current_dirfile;
    return FALSE; // file not found
}
```

- FileSystem::List(bool recursive, char *path):先判斷如果是要List “/” 則直接將File Fetch出來，如果需要遞迴call directory的ListRecursive()，不需要的話call directory的List()；其他狀況的話先利用findsubdirectory找到前一層目錄(subdirfile)並把target path改為file name，去搬目錄的content(FetchFrom)，找到需要List的directory的sector，在該sector new一個openFile，並new一個directory將該openFile fetch出來，如果需要遞迴call directory的ListRecursive()，不需要的話call directory的List()。



```
void
FileSystem::List(bool recursive, char *dirPath)
{
    if (strcmp(dirPath, "/") == 0)
    {
        Directory *directory = new Directory(NumDirEntries);
        directory->FetchFrom(directoryFile);//
        if (recursive == TRUE) directory->ListRecursive();
        else directory->List();
        delete directory;
        return;
    }
    else
    {
        char targetPath[500];
        strcpy(targetPath, dirPath);

        OpenFile *subdirfile = findsubdirectory(targetPath);
        if (subdirfile == NULL) return;
        Directory *subdirectory = new Directory(NumDirEntries);
        subdirectory->FetchFrom(subdirfile);

        int targetsector = subdirectory->Find(targetPath);
        Directory *targetdirectory = new Directory(NumDirEntries);
        OpenFile *targetdirfile = new OpenFile(targetsector);
        targetdirectory->FetchFrom(targetdirfile);

        if (recursive == TRUE) targetdirectory->ListRecursive();
        else targetdirectory->List();

        delete targetdirectory;
        delete targetdirfile;
        delete subdirectory;
        if (subdirfile != directoryFile) delete subdirfile;
    }
}
```

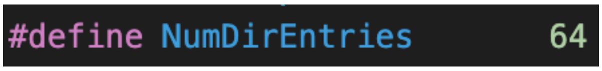
- FileSystem::findsubdirectory(char *path):利用”/”當作分段，將傳進來的路徑，利用strtok每次得到其中一個子字串，並且創造current_dirfile用來表示當前在哪一層目錄，然後將current_directory執行FetchFrom(directoryFile)，一開始的token紀錄了第一個子字串，接著利用nexttoken紀錄下一個子字串，從根目錄開始尋找傳進來路徑的subdirectory。在while迴圈裡，判斷如果nexttoken不是NULL且token是current_directory裡的一個directory file，才能繼續尋找，直到找到傳進來路徑的subdirectory，最後將路徑改為token，然後delete current_directory，並且回傳subdirectory也就是最後的current_dirfile。



```
OpenFile *FileSystem::findsubdirectory(char *path)
{
    char *split = "/";
    char *token = strtok(path, split);

    OpenFile *current_dirfile = directoryFile;
    Directory *current_directory = new Directory(NumDirEntries);
    current_directory->FetchFrom(directoryFile);
    if (token != NULL)
    {
        char *nextToken = "";
        nextToken = strtok(NULL, split);
        while (nextToken != NULL && current_directory->IsDir(token) == TRUE)
        {
            int sector = current_directory->Find(token);
            if (current_dirfile != directoryFile) delete current_dirfile;
            if (sector == -1)
            {
                delete current_directory;
                return NULL;
            }
            else
            {
                current_dirfile = new OpenFile(sector);
                current_directory->FetchFrom(current_dirfile);
            }
            token = nextToken;
            nextToken = strtok(NULL, split);
        }
        strcpy(path, token);
        delete current_directory;
        return current_dirfile;
    }
    else
    {
        delete current_directory;
        return NULL;
    }
}
```

(2) Support up to 64 files/subdirectories per directory

- 將filesys.h檔裡的 NumDirEntries改為64。



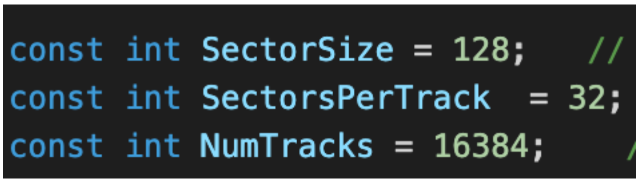
```
#define NumDirEntries       64
```

Bonus Assignment

Bonus I. Enhance the NachOS to support even larger file size

(1) Extend the disk from 128KB to 64MB

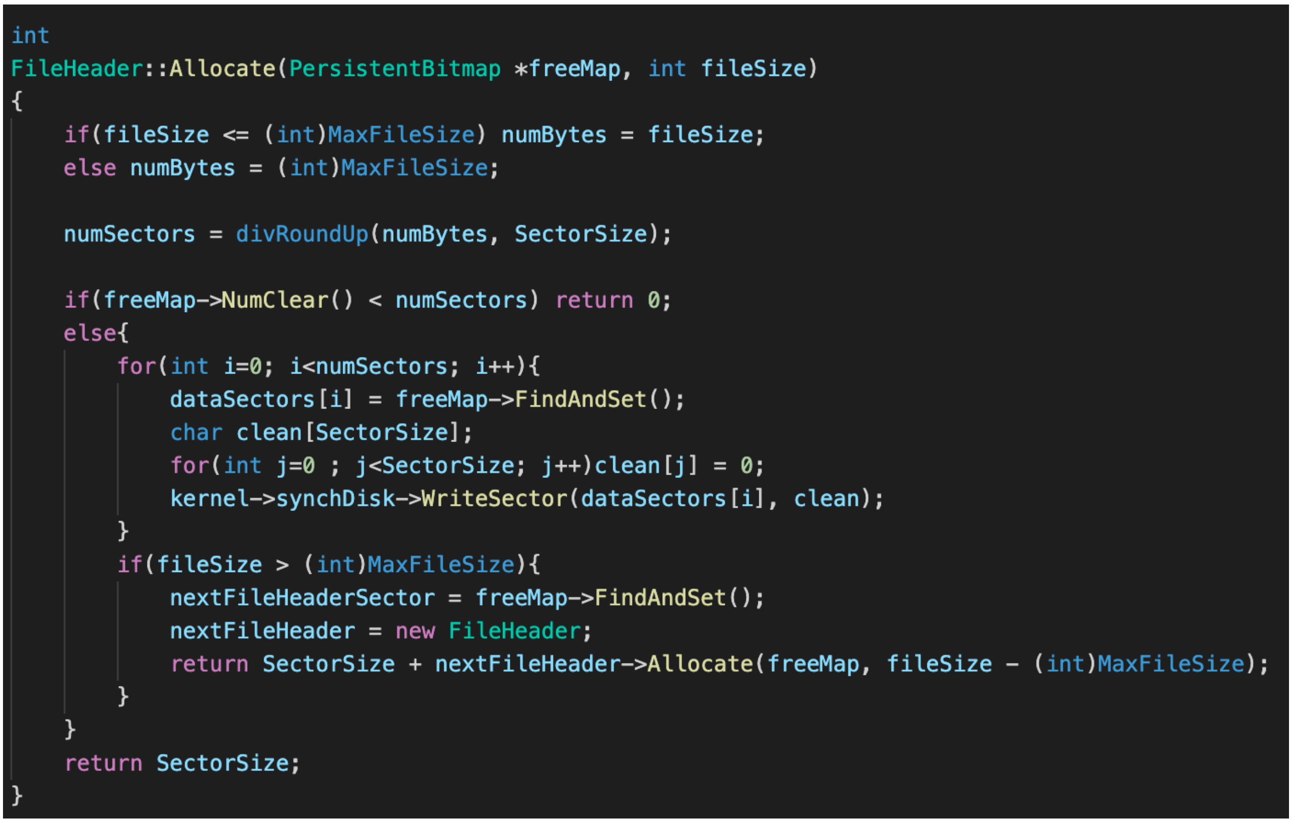
- 原本的SectorSize = 128bytes，SectorPerTrack = 32，NumTracks = 32，所以maximum disk size = 32 * 32 * 128 (Bytes) = 2**17 (Bytes) = 128KB，要提高disk size需更動這三個變數的值，由於SectorSize、SectorPerTrack是不能被更動的，於是我們將NumTracks改為16384(64MB/128B/32=16384)。



```
const int SectorSize = 128;    //
const int SectorsPerTrack  = 32;
const int NumTracks = 16384;      /
```

(2) Support up to 64MB single file

- 在filehdr.cc的Allocate()我們根據file的大小去遞迴Allocate，file如果是64MB的話，在空間允許的狀況下可以一直往下Allocate，所以我們的做法是可以Support 64MB file的。



```
int
FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
{
    if(fileSize <= (int)MaxFileSize) numBytes = fileSize;
    else numBytes = (int)MaxFileSize;

    numSectors = divRoundUp(numBytes, SectorSize);

    if(freeMap->NumClear() < numSectors) return 0;
    else{
        for(int i=0; i<numSectors; i++){
            dataSectors[i] = freeMap->FindAndSet();
            char clean[SectorSize];
            for(int j=0 ; j<SectorSize; j++)clean[j] = 0;
            kernel->synchDisk->WriteSector(dataSectors[i], clean);
        }
        if(fileSize > (int)MaxFileSize){
            nextFileHeaderSector = freeMap->FindAndSet();
            nextFileHeader = new FileHeader;
            return SectorSize + nextFileHeader->Allocate(freeMap, fileSize - (int)MaxFileSize);
        }
    }
    return SectorSize;
}
```

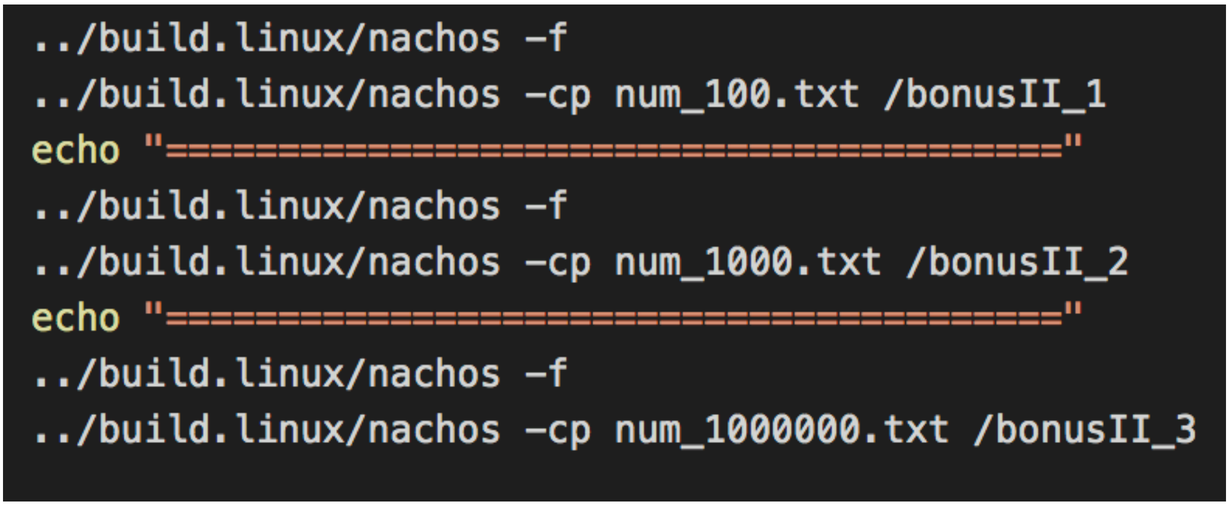
Bonus II. Multi-level header size

(1) Show that smaller file can have smaller header size.

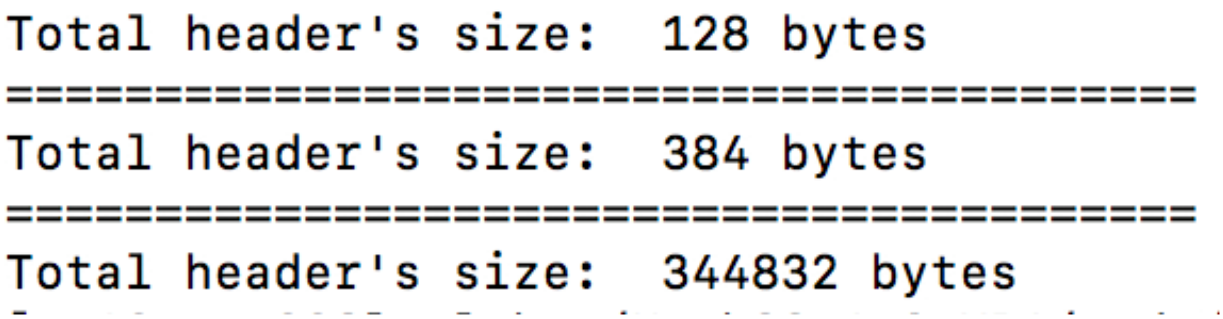
- 我們的實作方式是跟根據file的大小來配置fileheader的數量，總header size會因為fileheader數量不同而不同，也就是說，當file較小時，會有較小的header size。

(2) Implement at least 3 different size of headers for different size of files

- 以下是我們的測資，可以看出來，當在-cp較小的文件時，其擁有較小的header size，而在-cp較大的檔案時，其擁有較大的header size。



```
../build.linux/nachos -f
../build.linux/nachos -cp num_100.txt /bonusII_1
echo "========================================"
../build.linux/nachos -f
../build.linux/nachos -cp num_1000.txt /bonusII_2
echo "========================================"
../build.linux/nachos -f
../build.linux/nachos -cp num_1000000.txt /bonusII_3
```

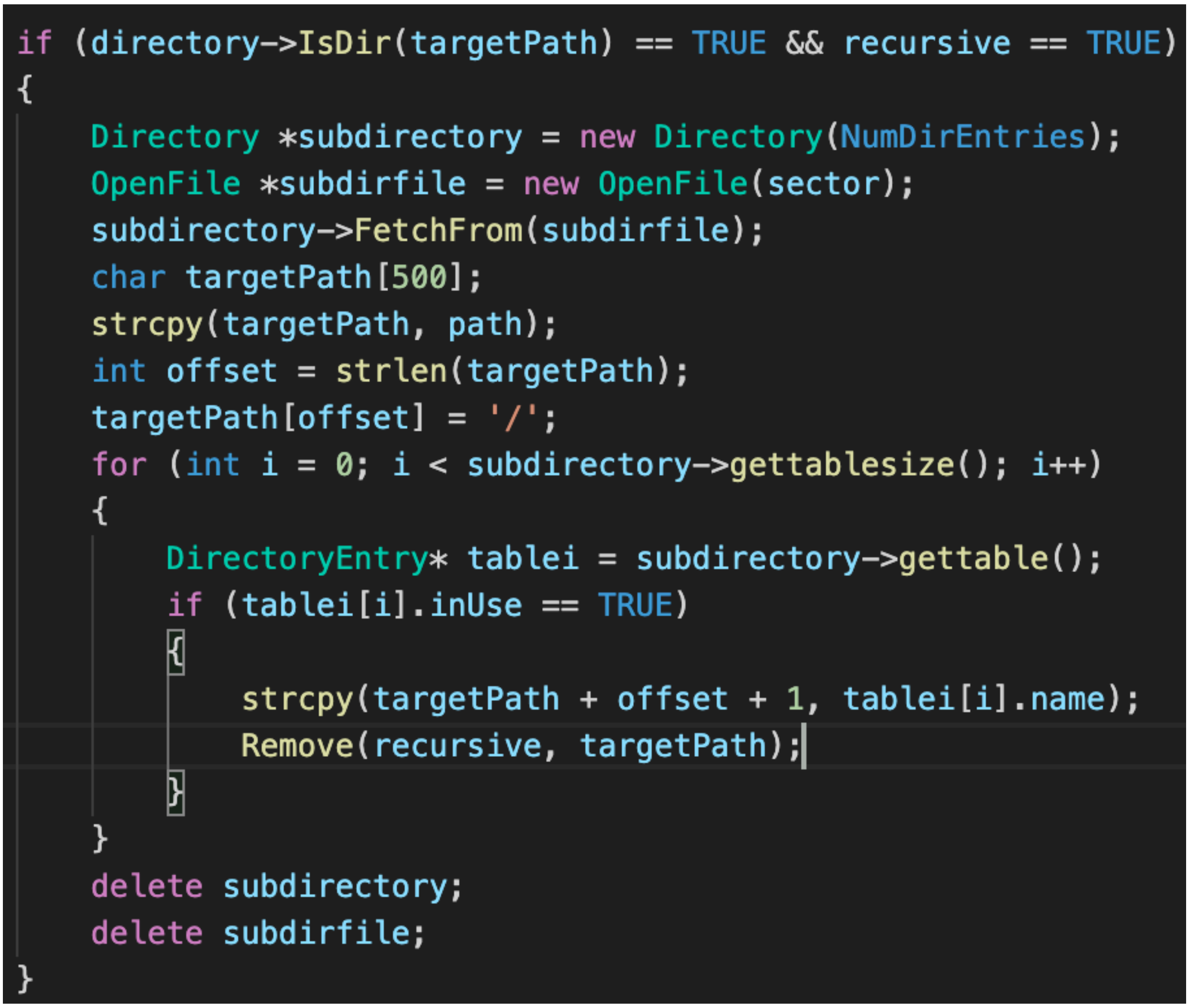


```
Total header's size:  128 bytes
========================================
Total header's size:  384 bytes
========================================
Total header's size:  344832 bytes
```

Bonus III. Recursive Operations on Directories

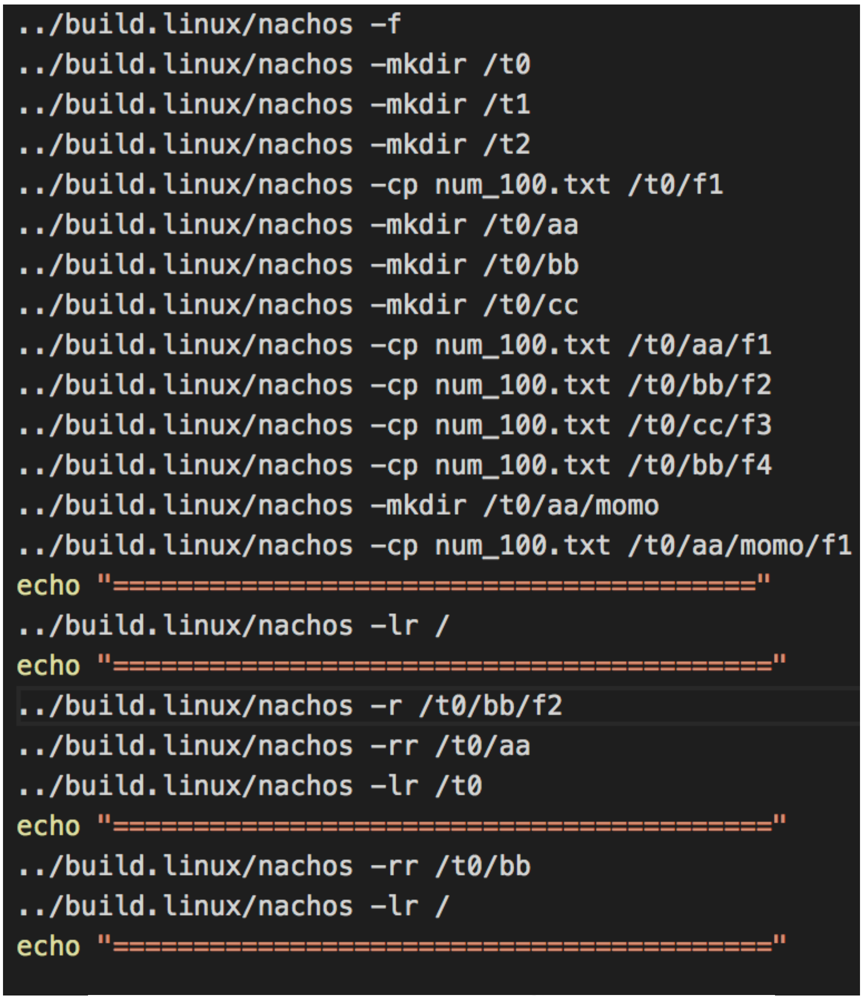
(1) Support recursive remove of a directory

- 在Remove裡面，我們多傳入了一個recursive，用來判斷是否需要遞迴刪除，如果要刪除的對象是一個目錄，且recursive這個flag是TRUE的話，那就必須將這個目錄裡面的file全部刪除，也就是執行recursive remove，這裡的方法是先將path後面接上一個 ’ / ’，然後利用我們在directory自定義的function把tablesize與table取出來，用for迴圈去針對table裡面inUse為TRUE的項目，將其name接在target後面，最後呼叫Remove遞迴刪除。

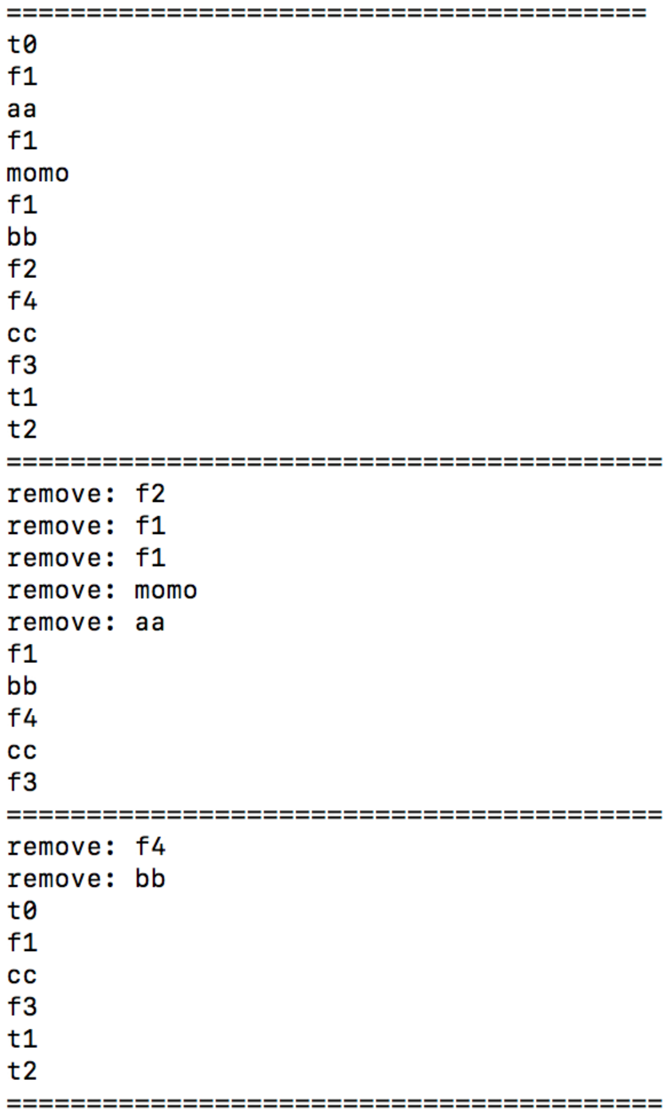


```
if (directory->IsDir(targetPath) == TRUE && recursive == TRUE)
{
    Directory *subdirectory = new Directory(NumDirEntries);
    OpenFile *subdirfile = new OpenFile(sector);
    subdirectory->FetchFrom(subdirfile);
    char targetPath[500];
    strcpy(targetPath, path);
    int offset = strlen(targetPath);
    targetPath[offset] = '/';
    for (int i = 0; i < subdirectory->gettablesize(); i++)
    {
        DirectoryEntry* tablei = subdirectory->gettable();
        if (tablei[i].inUse == TRUE)
        {
            strcpy(targetPath + offset + 1, tablei[i].name);
            Remove(recursive, targetPath);
        }
    }
    delete subdirectory;
    delete subdirfile;
}
```

下兩張圖為測資及run的結果。



```
../build.linux/nachos -f
../build.linux/nachos -mkdir /t0
../build.linux/nachos -mkdir /t1
../build.linux/nachos -mkdir /t2
../build.linux/nachos -cp num_100.txt /t0/f1
../build.linux/nachos -mkdir /t0/aa
../build.linux/nachos -mkdir /t0/bb
../build.linux/nachos -mkdir /t0/cc
../build.linux/nachos -cp num_100.txt /t0/aa/f1
../build.linux/nachos -cp num_100.txt /t0/bb/f2
../build.linux/nachos -cp num_100.txt /t0/cc/f3
../build.linux/nachos -cp num_100.txt /t0/bb/f4
../build.linux/nachos -mkdir /t0/aa/momo
../build.linux/nachos -cp num_100.txt /t0/aa/momo/f1
echo "========================================"
../build.linux/nachos -lr /
echo "========================================="
../build.linux/nachos -r /t0/bb/f2
../build.linux/nachos -rr /t0/aa
../build.linux/nachos -lr /t0
echo "========================================="
../build.linux/nachos -rr /t0/bb
../build.linux/nachos -lr /
echo "========================================="
```



```
========================================
t0
f1
aa
f1
momo
f1
bb
f2
f4
cc
f3
t1
t2
=========================================
remove: f2
remove: f1
remove: f1
remove: momo
remove: aa
f1
bb
f4
cc
f3
=========================================
remove: f4
remove: bb
t0
f1
cc
f3
t1
t2
=========================================
```

Feedback:

這次作業在trace code時就更加了解NachOS在Disk 的運作，了解了NachOS 怎麼管理free block，File System怎麼manage directory structure，還有disk size與file size的資訊，對後面的implement非常有幫助。在implement時，有許多小地方需要注意，像是FetchFrom的call法、recursive的寫法...，有時候改一個地方需要注意前前後後有哪些地方有call這個function及這個function call了哪些function，都是需要一起改的，做完之後更了解了上課所學的一些disk scheduling、management，也懂的如何應用在coding上，收穫許多!