

MP4_report_37

tags: 筆記, 作業系統, OS

Team Member & Contributions

- 資應碩二 **107065522** 陳子潔
- 數學大四 **105021127** 徐迺茜

工作項目	分工
Trace Code & 寫報告	陳子潔 & 徐迺茜
功能實作 & 測試	陳子潔 & 徐迺茜

Part I. Trace code

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

- 在 Nachos 的FileSystem中，是通過Bitmap來管理Free blocks的。Nachos 的Physical Disk以Sectors為最小存取單位
- Sector從 0 開始編號。所謂管理，就是將這些編號填入一張Table
 - 表中為 0 的地方說明該Sector沒有被佔用，而1說明該Sector已被佔用
- 首先從kernal.cc開始觀察，在Initialize()函式裡，可以看到file system的初始化，如下所示:

void Kernel::Initialize()

```
1 | fileSystem = new FileSystem(formatFlag);
```

- 而formatFlag按照上面的Kernal()函式，當指令下達-f時，formatFlag = True，如下所示:

Kernel::Kernel(int argc, char **argv)

```
1 | #ifndef FILESYS_STUB
2 |     } else if (strcmp(argv[i], "-f") == 0) {
3 |         formatFlag = TRUE;
```

- 接著進入filesys.cc中觀察FileSystem(bool format)函式，可以發現當format = True，表示我們需要初始化磁碟來包含一個空的路徑，以及一個 bitmat 的磁區（幾乎但非全部的磁區被標示為可使用）。而format = False，就用freeMapFile = OpenFile(FreeMapSector)到FreeMapSector去讀header，把代表freeMap的file打開放在freeMapFile，而且這個file會在NachOS跑的期間一直存在。
- 當format = True，可以看到FileSystem(bool format)呼叫PersistentBitmap(NumSectors)初始化bitmap。

FileSystem::FileSystem(bool format)

```
1 | if (format) {
2 |     // 紀錄NumSectors個sectors的使用情況
3 |     PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);
4 |     ... ..
```

- 接著進到pbitmap.h觀察可以發現他大部分都繼承 bitmap.h。NachOS 主要是利用 bitmap 來記錄哪裡有free block space。
- 在bitmap.cc裡的Bitmap::Bitmap(int numItems)初始化bitmap，從下面的code可以看到，先開一個map，之後將使用情形紀錄在map裡，如果是0表示該位置的block還沒被使用，1則是已經被使用。因為是初始化的關係，先將numItems個sectors使用情形設置為0。

Bitmap::Bitmap(int numItems)

```

1  Bitmap::Bitmap(int numItems)
2  {
3      int i;
4
5      ASSERT(numItems > 0);
6
7      numBits = numItems;
8      numWords = divRoundUp(numBits, BitsInWord);
9      map = new unsigned int[numWords];
10     for (i = 0; i < numWords; i++) {
11         map[i] = 0;    # initialize map to keep Purify happy
12     }
13     for (i = 0; i < numBits; i++) {
14         Clear(i);
15     }
16 }

```

- 深入bitmap.h裡面觀察，有個用來找尋free block space的函式叫做FindAndSet()，第五行的Test(int i)也是bitmap.cc裡的函式，用來判斷第"i"個bit是否已被佔用，如果有就回傳True，反之就回傳False。
- 利用Test()搭配for迴圈，FindAndSet()會回傳第一個free block space的位置(即block[i] is free)。如果跑遍整個bitmap都找不到free block space，則回傳-1。
- 接著利用Mark()將第i個bit設置成用過(即block[i]要被使用)

FindAndSet()

```

1  int Bitmap::FindAndSet()
2  {
3      for (int i = 0; i < numBits; i++) {
4          if (!Test(i)) { // Test會判斷第"i"bit是否被設定
5              Mark(i);    // 將"i"bit設置成使用
6              return i;
7          }
8      }
9      return -1;
10 }

```

Test(int which)

```

1  bool
2  Bitmap::Test(int which) const
3  {
4      ASSERT(which >= 0 && which < numBits);
5
6      #檢查map[i]=1以及是合法的bitmap
7      if (map[which / BitsInWord] & (1 << (which % BitsInWord))) {
8          return TRUE;
9      } else {
10         return FALSE;
11     }
12 }

```

Mark(int which)

```

1 void
2 Bitmap::Mark(int which)
3 {
4     ASSERT(which >= 0 && which < numBits);
5
6     #將第i個bit設置成1
7     map[which / BitsInWord] |= 1 << (which % BitsInWord);
8
9     ASSERT(Test(which));
10 }

```

- 回過頭來看再繼續看filesys.cc的FileSystem(bool format)，還要new一個FileHeader叫mapHdr給它，如下所示：

FileSystem::FileSystem(bool format)

```

1     ... ..
2     FileHeader *mapHdr = new FileHeader;
3     FileHeader *dirHdr = new FileHeader;
4     ... ..

```

- 接著往下可以看到以下這段code。從Mark()裡面可以知道這一行的意思是bitmap header放在disk的第"FreeMapSector"個sector。
- 因為它的FileHeader要固定放在FreeMapSector，所以FreeMapSector只能被freeMap的header使用，要先把FreeMapSector標示成已被使用，再來去allocate freeMap的data時才不會用到FreeMapSector。
- 再來用mapHdr->Allocate(freeMap, FreeMapFileSize)決定freeMap的data要放在哪些sector。
- 其中freeMap是剛剛new而且已經改過的PersistentBitmap，FreeMapFileSize是freeMap的大小，算法是總共有幾個sectors除以BitInBytes。

```

1     freeMap->Mark(FreeMapSector);
2     ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));

```

```

1     #define FreeMapFileSize      (NumSectors / BitsInByte)

```

- 在filesys.cc裡有定義FreeMapSector跟DirectorySector的值，分別是0跟1。(如下所示)

```

1     #define FreeMapSector        0
2     #define DirectorySector     1

```

- 這裡再補充一下剩下還沒說到的FileSystem函式。在FileSystem::Create/Remove/Print 需要用到freeMap的時候，是先new一個PersistentBitmap，用freeMapFile初始化它，然後用這個一開始跟freeMapFile，之後FreeMapFile也是會一直開著用來找free space。在FileSystem::Create/Remove/Print需要用到freeMap的時候，是先new一個PersistentBitmap，用freeMapFile初始化它，然後用這個一開始跟freeMapFile一樣的新的freeMap去Create/Remove/Print，成功且有更新freeMap的話再寫回到freeMapFile，否則就丟掉這次的動作。
- Create的時候用freeMap->FindAndSet()找header要放的位置和更新freeMap，用fileHeader->Allocate(PersistentBitmap *freeMap, int fileSize) 找 data 放的位置和更新freeMap。
- Remove 用FileHeader::Deallocate(PersistentBitmap *freeMap, int fileSize) 更新 freeMap 把 data 空間釋放出來，再用Clear釋放fileHeader空間。

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

- 先觀察disk.h裡面可以看到，NachOS模擬的disk包含**NumTracks(32)** 個 tracks，每個track包含**SectorsPerTrack(32)** 個 sectors。各個sector的大小為**SectorSize(128)** bytes

disk.h

```

1  const int SectorSize = 128;           // number of bytes per disk sector
2  const int SectorsPerTrack = 32;       // number of sectors per disk track
3  const int NumTracks = 32;             // number of tracks per disk
4  const int NumSectors = (SectorsPerTrack * NumTracks);
5                                         // total # of sectors per disk

```

- 再來觀察disk.cc可以看到，NachOS **DiskSize = (MagicSize + (NumSectors * SectorSize))**。MagicNumber的用處是為了減少我們不小心將有用的文件視為磁盤的可能性(這可能會破壞文件的內容)。如果文件已經存在，Nachos會讀取前4個字節驗證他們是否包含預期的Nachos MagicNumber，如果檢查失敗則終止。這也是為甚麼DiskSize前面要加上MagicSize。

disk.cc (<http://disk.cc>)

```

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

```

- 總結來看，NachOS maximum disk size = (sizeof(int) + (32 * 32 * 128)) 大約等於 128KB

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

- NachOS FS 把 directory 當成一個file來管理，有header和dataSectors。Data是Directory，用來記錄檔案名稱和檔案fileHeader位置。
- 和Part(1)的free block space相似，這裡我們直接從filesys.cc的code開始trace起。
- 當format = False，就用directoryFile = OpenFile(DirectorySector)到DirectorySector去讀header，把代表directory的file打開放在directoryFile，而且這個file會再NachOS跑的期間一直存在，就可以用這個file去找fileHeader。
- 當format = True，可以看到FileSystem(bool format)new 一個 Directory 去紀錄 fileName和fileHeader location。(如下所示)

FileSystem(bool format)

```

1  if (format) {
2      PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);
3      Directory *directory = new Directory(NumDirEntries);
4      ... ...

```

- 接著進到directory.cc裡，觀察Directory(int size)函式，可以看到一開始directory的每個entry都還沒被用掉。

Directory::Directory(int size)

```

1  Directory::Directory(int size)
2  {
3      table = new DirectoryEntry[size];
4
5      // MP4 mod tag
6      // dummy operation to keep valgrind happy
7      memset(table, 0, sizeof(DirectoryEntry) * size);
8
9      tableSize = size;
10     for (int i = 0; i < tableSize; i++)
11         table[i].inUse = FALSE;
12 }

```

- 回過頭來看再繼續看filesys.cc的FileSystem(bool format)，還要new一個FileHeader叫dirHdr給它，如下所示:

FileSystem::FileSystem(bool format)

```

1     ... ..
2     FileHeader *mapHdr = new FileHeader;
3     FileHeader *dirHdr = new FileHeader;
4     ... ..

```

- 繼續看可以看到裡面有個Mark()函式(在Part1(1)已經描述過此函式，這裡不再重述)，如下所示:

FileSystem::FileSystem(bool format)

```

1     freeMap->Mark(DirectorySector);
2     ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));

```

- 從Mark()裡面可以知道這1行的意思是directory占用第"DirectorySector"個sector。
- 因為它的FileHeader要固定放在DirectorySector，所以DirectorySector只能被directory的header使用，要先把DirectorySector標示成已被使用，再來去allocate directory的data時才不會用到DirectorySector。
- 再來用mapHdr->Allocate(freeMap, DirectoryFileSize)決定directory的data要放在哪些sector。
- 其中freeMap是剛剛new而且已經改過的PersistentBitmap，DirectoryFileSize是directory的大小，算法是總共有幾個Directory乘以DirectoryEntry的大小。(如下所示)

```

1     #define NumDirEntries      10
2     #define DirectoryFileSize  (sizeof(DirectoryEntry) * NumDirEntries)

```

- 在filesys.cc裡有定義FreeMapSector跟DirectorySector的值，分別是0跟1。(如下所示)

```

1     #define FreeMapSector      0
2     #define DirectorySector    1

```

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

- 有關inode(即用來管理disk file header)，我們要觀察filehrd.cc和filehrd.h
- 要知道inode存甚麼資訊，要觀察filehrd.h的**class FileHeader**的private(public主要是各種函式)，如下所示:

class FileHeader

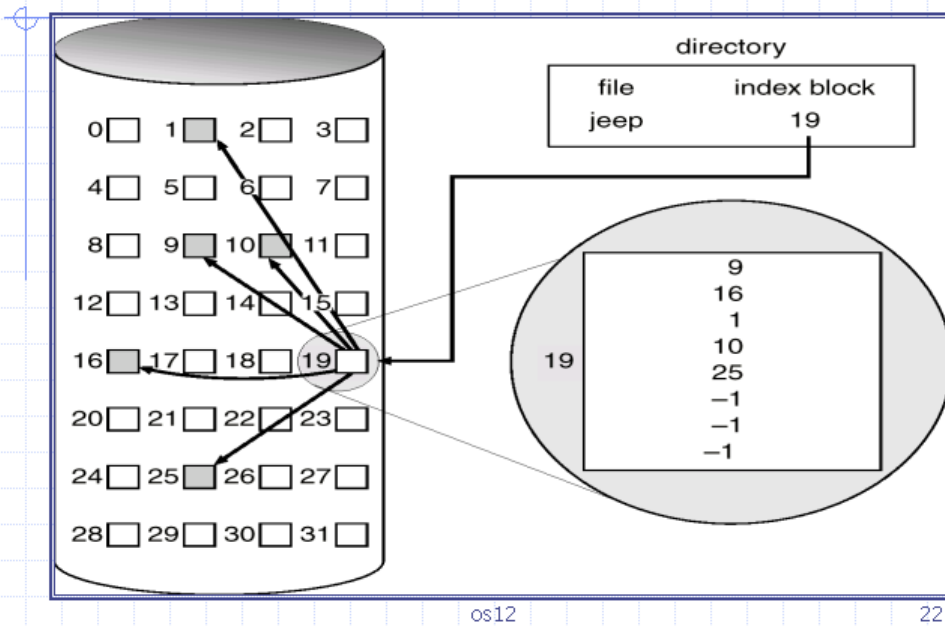
```

1     class FileHeader {
2     private:
3         int numBytes;        // Number of bytes in the file
4         int numSectors;      // Number of data sectors in the file
5         int dataSectors[NumDirect]; // Disk sector numbers for each data
6                                     // block in the file
7     };

```

- 可以看到在inode(fileHeader)裡，存了numBytes: 這個file有幾個bytes，numSectors:這個file要用到幾個dataSector，和dataSectors[NumDirect]: 每個dataBlock在哪個sector。
- 示意圖:

Example of Indexed Allocation



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

- 觀察filehrd.cc裡的Allocate(), 從下面第三行可以看出現在的file fileHeader只有一層, 一個fileHeader能用的空間就只有32個sector的大小

FileHeader::Allocate()

```
1  ... ...
2  for (int i = 0; i < numSectors; i++) {
3      dataSectors[i] = freeMap->FindAndSet();
4      // since we checked that there was enough free space,
5      // we expect this to succeed
6      ASSERT(dataSectors[i] >= 0);
7  }
8  return TRUE;
9  }
```

- 從filehrd.h可以知道(如下所示), 扣掉numBytes、numSectors, 就只能記錄NumDirect = ((SectorSize - 2 * sizeof(int))/sizeof(int)) = 30 這麼多的dataSector位置, 一個file也就只能有MaxFileSize = (NumDirect * SectorSize) = 30*128 bytes 的資料。

filehrd.h

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

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

- 主要有修改到的檔案為: Syscall, Ksyscall, exception.cc (<http://exception.cc>), filesystem
- 流程大致與MP1差不多, 故直接上CODE了

syscall.h

```
1 int Create(char *name, int size);
```

- 改一下function signature就好了

exception.c

```

1  #ifndef FILESYS_STUB
2  case SC_Create:
3  val = kernel->machine->ReadRegister(4);
4  {
5      char *fileName = &(kernel->machine->mainMemory[val]);
6      int initialSize = kernel->machine->ReadRegister(5);
7      status = SysCreate(fileName, initialSize);
8      kernel->machine->WriteRegister(2,(int)status);
9  }
10
11 kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
12 kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
13 kernel->machine->WriteRegister(NextPCReg ,kernel->machine->ReadRegister(PCReg)+4
14 return;
15 ASSERTNOTREACHED();
16 break;
17
18 case SC_Open:
19 val = kernel->machine->ReadRegister(4);
20 {
21     char *fileName = &(kernel->machine->mainMemory[val]);
22     status = SysOpen(fileName);
23     kernel->machine->WriteRegister(2,(int)status);
24 }
25
26 kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
27 kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
28 kernel->machine->WriteRegister(NextPCReg ,kernel->machine->ReadRegister(PCReg)+4
29 return;
30 ASSERTNOTREACHED();
31 break;
32
33 case SC_Write:
34 val = kernel->machine->ReadRegister(4);
35 {
36     char *buffer = &(kernel->machine->mainMemory[val]);
37     int initialSize = kernel->machine->ReadRegister(5);
38     int id = kernel->machine->ReadRegister(6);
39     status = SysWrite(buffer, initialSize, id);
40     kernel->machine->WriteRegister(2,(int)status);
41 }
42
43 kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
44 kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
45 kernel->machine->WriteRegister(NextPCReg ,kernel->machine->ReadRegister(PCReg)+4
46 return;
47 ASSERTNOTREACHED();
48 break;
49
50 case SC_Read:
51 val = kernel->machine->ReadRegister(4);
52 {
53     char *buffer = &(kernel->machine->mainMemory[val]);
54     int initialSize = kernel->machine->ReadRegister(5);
55     int id = kernel->machine->ReadRegister(6);
56     status = SysRead(buffer ,initialSize ,id);
57     kernel->machine->WriteRegister(2,(int)status);
58 }
59
60 kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
61 kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
62 kernel->machine->WriteRegister(NextPCReg ,kernel->machine->ReadRegister(PCReg)+4
63 return;
64 ASSERTNOTREACHED();
65 break;
66
67 case SC_Close:
68 {
69     int fd = kernel->machine->ReadRegister(4);
70     status = SysClose(fd);
71     kernel->machine->WriteRegister(2,(int)status);
72 }
73
74 kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
75 kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
76 kernel->machine->WriteRegister(NextPCReg ,kernel->machine->ReadRegister(PCReg)+4
77 return;
78 ASSERTNOTREACHED();

```



```
79     break;
80 #endif
```

filesystem.h

```
1 // 跟下面的CreateFile0幾乎一樣，主要是給OS內部用的
2 bool Create(char *name, int initialSize);
3 // 跟下面的OpenFile0幾乎一樣，主要是給OS內部用的
4 OpenFile* Open(char *name);
5
6 // Create a file (UNIX creat)
7 int CreateFile0(char* name, int initialSize);
8 // Open a file (UNIX open)
9 OpenFileId OpenFile0(char* name);
10 int ReadFile0(char* buffer, int initialSize, int id);
11 int WriteFile0(char* buffer, int initialSize, int id);
12 int CloseFile0(int id);
13 OpenFileId PutFileDescriptor(OpenFile* fileDescriptor);
14
15 int fileDescriptorIndex;
16 OpenFile* fileDescriptorTable[MAXOPENFILES];
```

- fileDescriptorTable就是OS的System-Wide Open file table
- Function後面有0是因為有些函式會跟sysdep裡面的功能有命名衝突，我懶，就全部都改成0結尾了

filesystem.c

```

1  int FileSystem::CreateFile0(char* name, int initialSize)
2  {
3      return Create(name, initialSize);
4  }
5
6  OpenFileId FileSystem::OpenFile0(char* name)
7  {
8      OpenFile* openPtr = Open(name);
9      return PutFileDescriptor(openPtr);
10 }
11
12 int FileSystem::ReadFile0(char* buffer, int size, int id)
13 {
14     if(id <= 0 || id >= MAXOPENFILES) return -1;
15     if(fileDescriptorTable[id] == NULL) return -1;
16     return fileDescriptorTable[id]->Read(buffer, size);
17 }
18
19 int FileSystem::WriteFile0(char* buffer, int size, int id)
20 {
21     if(id <= 0 || id >= MAXOPENFILES) return -1;
22     if(fileDescriptorTable[id] == NULL) return -1;
23     return fileDescriptorTable[id]->Write(buffer, size);
24 }
25
26 int FileSystem::CloseFile0(int id)
27 {
28     if(id < 0 || id >= MAXOPENFILES) return -1;
29     if(fileDescriptorTable[id] == NULL) return -1;
30     delete fileDescriptorTable[id];
31     fileDescriptorTable[id] = NULL;
32     return 1;
33 }
34
35 OpenFileId FileSystem::PutFileDescriptor(OpenFile* FileDesc)
36 {
37     int cnt = 0;
38     while( (++fileDescriptorIndex % MAXOPENFILES == 0)
39         || fileDescriptorTable[fileDescriptorIndex] != NULL)
40     {
41         if(cnt < MAXOPENFILES)
42             cnt++;
43         else
44             return 0;
45     }
46     fileDescriptorTable[fileDescriptorIndex] = FileDesc;
47     return fileDescriptorIndex;
48 }

```

- PutFileDescriptor()此函數會將Create的File放入fileDescriptorTable當中，並回傳放入Table的index位置

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

- 於filehdr.h中新增資料結構:
 - nextFileHeaderSector
 - nextFileHeader
- 接著修改以下函數:
 - FileHeader::Destructor
 - FileHeader::FetchFrom
 - FileHeader::WriteBack
 - FileHeader::Allocate
 - FileHeader::Deallocate
 - FileHeader::ByteToSector
 - OpenFile::Length
 - 改定義為所有連接在一起的 FileHeader

- OpenFile::ReadAt
 - 修改取得 FileLength所呼叫的函數
- OpenFile::WriteAt
- 修改取得FileLength的函數

filehdr.h

```
1 // 原本是 -2 , 改成-3 多存一個指標
2 #define NumDirect      ((SectorSize - 3 * sizeof(int)) / sizeof(int))
3
4 public:
5     FileHeader* getNextFileHeader() { return nextFileHeader;}
6     int HeaderLength(); // 新增, 回傳Headr有幾格
7
8 private:
9     FileHeader* nextFileHeader; // (in-core) 這個要永遠放在第一格!!!!!!
10    ...
11    int nextFileHeaderSector;    // (in-disk)
```

filehdr.c

```

1  FileHeader::FileHeader()
2  {
3      nextFileHeader = NULL;
4      nextFileHeaderSector = -1;
5      numBytes = -1;
6      numSectors = -1;
7      memset(dataSectors, -1, sizeof(dataSectors));
8  }
9
10 FileHeader::~FileHeader()
11 {
12     // nothing to do now
13     if (nextFileHeader != NULL)
14         delete nextFileHeader;
15 }
16
17 bool FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
18 {
19     // 本層FH要存的Bytes
20     numBytes = fileSize < MaxFileSize ? fileSize : MaxFileSize;
21     // 剩下要存的Bytes
22     fileSize -= numBytes;
23     // 幫本層分配Data Sectors
24     numSectors = divRoundUp(numBytes, SectorSize);
25
26     // not enough space
27     if (freeMap->NumClear() < numSectors)
28         return FALSE;
29
30     for (int i = 0; i < numSectors; i++) {
31         dataSectors[i] = freeMap->FindAndSet();
32         //we checked that there was enough free space, we expect this to succeed
33         ASSERT(dataSectors[i] >= 0);
34     }
35     // 分配完本層，再多分配一個Link指向下一個FH
36     if (fileSize > 0) {
37         // 為了確認有沒有Free Sector
38         nextFileHeaderSector = freeMap->FindAndSet();
39         if (nextFileHeaderSector == -1) // 沒有Free Sector
40             return FALSE;
41         else {
42             nextFileHeader = new FileHeader;
43             return nextFileHeader->Allocate(freeMap, fileSize);
44         }
45     }
46     return TRUE;
47 }
48
49 void FileHeader::Deallocate(PersistentBitmap *freeMap)
50 {
51     for (int i = 0; i < numSectors; i++) {
52         ASSERT(freeMap->Test((int)dataSectors[i])); // ought to be marked!
53         freeMap->Clear((int)dataSectors[i]);
54     }
55     if (nextFileHeaderSector != -1)
56     {
57         ASSERT(nextFileHeader != NULL);
58         nextFileHeader->Deallocate(freeMap);
59     }
60 }
61
62 void FileHeader::FetchFrom(int sector)
63 {
64     //DEBUG(dbgFile, "Test[J]: kernel->synchDisk->ReadSector");
65     kernel->synchDisk->ReadSector(sector, ((char *)this) + sizeof(FileHeader*))
66     //DEBUG(dbgFile, "Test[J]: Done");
67
68     if (nextFileHeaderSector != -1)
69     {
70         nextFileHeader = new FileHeader;
71         nextFileHeader->FetchFrom(nextFileHeaderSector);
72     }
73     /*
74     MP4 Hint:
75     After you add some in-core informations,
76     you will need to rebuild the header's structure
77     */
78 }

```

```

79
80 void FileHeader::WriteBack(int sector)
81 {
82     kernel->synchDisk->WriteSector(sector, ((char *)this) + sizeof(FileHeader*))
83
84     if (nextFileHeaderSector != -1)
85     {
86         ASSERT(nextFileHeader != NULL);
87         nextFileHeader->WriteBack(nextFileHeaderSector);
88     }
89     /*
90 MP4 Hint:
91 After you add some in-core informations,
92 you may not want to write all fields into disk.
93 Use this instead:
94 char buf[SectorSize];
95 memcpy(buf + offset, &dataToBeWritten, sizeof(dataToBeWritten));
96 ...
97 */
98 }
99
100 int FileHeader::ByteToSector(int offset)
101 {
102     int index = offset / SectorSize;
103     if (index < NumDirect)
104         return (dataSectors[index]);
105     else
106     {
107         ASSERT(nextFileHeader != NULL);
108         return nextFileHeader->ByteToSector(offset - MaxFileSize);
109     }
110 }

```

openfile.c

```

1  int OpenFile::ReadAt(char *into, int numBytes, int position)
2  {
3      int fileLength = Length();
4      ...
5  }
6
7  int OpenFile::WriteAt(char *from, int numBytes, int position)
8  {
9      int fileLength = Length();
10     ...
11 }
12
13 int OpenFile::Length()
14 {
15     int length = 0;
16     FileHeader *nextHdr = hdr;
17     while (nextHdr != NULL)
18     {
19         length += nextHdr->FileLength();
20         nextHdr = nextHdr->getNextFileHeader();
21     }
22     return length;
23 }

```

Part III. Modify the file system code to support subdirectory

(1) Implement the subdirectory structure

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

filesystem.c

- 從10改成64，應該是這樣吧...

Bonus Assignment

1. Enhance the NachOS to support even larger file size

Extend the disk from 128KB to 64MB

- 於Disk.h中可以發現以下三個變數會影響整個DISK的大小
- 而從ILMS上的討論串得知：
 - 不應修改的參數：
 - SectorSize
 - SectorsPerTrack
 - 可修改的參數：
 - NumTracks
- 於是把NumTracks從32改為32*512...
 - 此時的DISK就能支援 $128 * 32 * (32 * 512) \text{Bytes} = 2^{26} \text{Bytes} = 64 \text{ MB}$ 的容量了
- 而在Part II(b)當中，所採用的是Linked-Index的方式來做File Sector的配置，故可以**Support up to 64MB single file**

```

1 // number of bytes per disk sector
2 const int SectorSize = 128;
3 // number of sectors per disk track
4 const int SectorsPerTrack = 32;
5 // number of tracks per disk
6 const int NumTracks = 32 * 512;

```

- 測試用Bash，我把6個10MB的檔案存進去DISK裡面，並且把他Print出來
- 最後使用 "nachos -l /" 指令列出DISK內的FILE們，確實有6個FILE，且DISK大小被充滿為60MB左右

```

1 ../build.linux/nachos -f
2 echo "===== 1 ====="
3 ../build.linux/nachos -cp num_1000000.txt /1000000a
4 ../build.linux/nachos -p /1000000a
5 echo "===== 2 ====="
6 ../build.linux/nachos -cp num_1000000.txt /1000000b
7 ../build.linux/nachos -p /1000000b
8 echo "===== 3 ====="
9 ../build.linux/nachos -cp num_1000000.txt /1000000c
10 ../build.linux/nachos -p /1000000c
11 echo "===== 4 ====="
12 ../build.linux/nachos -cp num_1000000.txt /1000000d
13 ../build.linux/nachos -p /1000000d
14 echo "===== 5 ====="
15 ../build.linux/nachos -cp num_1000000.txt /1000000e
16 ../build.linux/nachos -p /1000000e
17 echo "===== 6 ====="
18 ../build.linux/nachos -cp num_1000000.txt /1000000f
19 ../build.linux/nachos -p /1000000f
20
21 echo "===== Last ====="
22 ../build.linux/nachos -l /

```

- 由於測試過程(cp)很費時，故直接附上截圖


```

18. os19team37
000999611 000999612 000999613 000999614 000999615 000999616 000999617 000999618 000999619 000999620
000999621 000999622 000999623 000999624 000999625 000999626 000999627 000999628 000999629 000999630
000999631 000999632 000999633 000999634 000999635 000999636 000999637 000999638 000999639 000999640
000999641 000999642 000999643 000999644 000999645 000999646 000999647 000999648 000999649 000999650
000999651 000999652 000999653 000999654 000999655 000999656 000999657 000999658 000999659 000999660
000999661 000999662 000999663 000999664 000999665 000999666 000999667 000999668 000999669 000999670
000999671 000999672 000999673 000999674 000999675 000999676 000999677 000999678 000999679 000999680
000999681 000999682 000999683 000999684 000999685 000999686 000999687 000999688 000999689 000999690
000999691 000999692 000999693 000999694 000999695 000999696 000999697 000999698 000999699 000999700
000999701 000999702 000999703 000999704 000999705 000999706 000999707 000999708 000999709 000999710
000999711 000999712 000999713 000999714 000999715 000999716 000999717 000999718 000999719 000999720
000999721 000999722 000999723 000999724 000999725 000999726 000999727 000999728 000999729 000999730
000999731 000999732 000999733 000999734 000999735 000999736 000999737 000999738 000999739 000999740
000999741 000999742 000999743 000999744 000999745 000999746 000999747 000999748 000999749 000999750
000999751 000999752 000999753 000999754 000999755 000999756 000999757 000999758 000999759 000999760
000999761 000999762 000999763 000999764 000999765 000999766 000999767 000999768 000999769 000999770
000999771 000999772 000999773 000999774 000999775 000999776 000999777 000999778 000999779 000999780
000999781 000999782 000999783 000999784 000999785 000999786 000999787 000999788 000999789 000999790
000999791 000999792 000999793 000999794 000999795 000999796 000999797 000999798 000999799 000999800
000999801 000999802 000999803 000999804 000999805 000999806 000999807 000999808 000999809 000999810
000999811 000999812 000999813 000999814 000999815 000999816 000999817 000999818 000999819 000999820
000999821 000999822 000999823 000999824 000999825 000999826 000999827 000999828 000999829 000999830
000999831 000999832 000999833 000999834 000999835 000999836 000999837 000999838 000999839 000999840
000999841 000999842 000999843 000999844 000999845 000999846 000999847 000999848 000999849 000999850
000999851 000999852 000999853 000999854 000999855 000999856 000999857 000999858 000999859 000999860
000999861 000999862 000999863 000999864 000999865 000999866 000999867 000999868 000999869 000999870
000999871 000999872 000999873 000999874 000999875 000999876 000999877 000999878 000999879 000999880
000999881 000999882 000999883 000999884 000999885 000999886 000999887 000999888 000999889 000999890
000999891 000999892 000999893 000999894 000999895 000999896 000999897 000999898 000999899 000999900
000999901 000999902 000999903 000999904 000999905 000999906 000999907 000999908 000999909 000999910
000999911 000999912 000999913 000999914 000999915 000999916 000999917 000999918 000999919 000999920
000999921 000999922 000999923 000999924 000999925 000999926 000999927 000999928 000999929 000999930
000999931 000999932 000999933 000999934 000999935 000999936 000999937 000999938 000999939 000999940
000999941 000999942 000999943 000999944 000999945 000999946 000999947 000999948 000999949 000999950
000999951 000999952 000999953 000999954 000999955 000999956 000999957 000999958 000999959 000999960
000999961 000999962 000999963 000999964 000999965 000999966 000999967 000999968 000999969 000999970
000999971 000999972 000999973 000999974 000999975 000999976 000999977 000999978 000999979 000999980
000999981 000999982 000999983 000999984 000999985 000999986 000999987 000999988 000999989 000999990
000999991 000999992 000999993 000999994 000999995 000999996 000999997 000999998 000999999 001000000

```

```

[os19team37@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -l /
/1000000a
/1000000b
/1000000c
/1000000d
/1000000e
/1000000f

```

```

[os19team37@lsalab ~/NachOS-4.0_MP4/code/test]$ stat DISK_0
  File: 'DISK_0'
  Size: 62137220      Blocks: 121376      IO Block: 4096   regular file
Device: 833h/2099d   Inode: 50472634    Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1038/os19team37)   Gid: ( 1002/ os2019)
Context: unconfined_u:object_r:user_home_t:s0
Access: 2020-01-13 21:47:21.441622134 +0800
Modify: 2020-01-13 21:47:21.439622177 +0800
Change: 2020-01-13 21:47:21.439622177 +0800

```

2. Multi-level header size

根據 Part II(b)中的實作, 我們使用 Linked List 的方式將所有 Index Sector 串接起來, 因此直接導致「越大的檔案, 越大的 Header Size」

```

[os19team37@lsalab ~/NachOS-4.0_MP4/code/test]$ ../build.linux/nachos -l /
/FS_test1
/file1
/FS_test2

```

```

Name: /FS_test2, Sector: 544
FileHeader contents.  File size: 980.  File blocks:
545 546 547 548 549 550 551 552

```

```

Name: /file1, Sector: 542
FileHeader contents.  File size: 27.  File blocks:
543

```

```

Name: /FS_test1, Sector: 533
FileHeader contents.  File size: 948.  File blocks:
534 535 536 537 538 539 540 541

```