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 = Falue，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)

```
Bitmap::Bitmap(int numItems)
{
    int i;

    ASSERT(numItems > 0);

    numBits = numItems;
    numWords = divRoundUp(numBits, BitsInWord);
    map = new unsigned int[numWords];
    for (i = 0; i < numWords; i++) {
        map[i] = 0;     # initialize map to keep Purify happy
    }
    for (i = 0; i < numBits; i++) {
        Clear(i);
    }
}
```

- 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()

```
int Bitmap::FindAndSet()
{
    for (int i = 0; i < numBits; i++) {
        if (!Test(i)) { // Test會判斷第"i"bit是否被設定
            Mark(i);    // 將"i"bit設置成使用
            return i;
        }
    }
    return -1;
}
```

Test(int which)

```
bool
Bitmap::Test(int which) const
{
    ASSERT(which >= 0 && which < numBits);

    #檢查map[i]=1以及是合法的bitmap
    if (map[which / BitsInWord] & (1 << (which % BitsInWord))) {
        return TRUE;
    } else {
        return FALSE;
    }
}
```

Mark(int which)

```
void
Bitmap::Mark(int which)
{
    ASSERT(which >= 0 && which < numBits);

    #將第i個bit設置成1
    map[which / BitsInWord] |= 1 << (which % BitsInWord);

    ASSERT(Test(which));
}
```

- 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)

```
    ... ...
    FileHeader *mapHdr = new FileHeader;
    FileHeader *dirHdr = new FileHeader;
    ... ...
```

- 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。

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

```
#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)

```
#define FreeMapSector         0
#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

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

- 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)

```
const int MagicNumber = 0x456789ab;
const int MagicSize = sizeof(int);
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)

```
if (format) {
        PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);
        Directory *directory = new Directory(NumDirEntries);
        ... ...
```

- 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)

```
Directory::Directory(int size)
{
    table = new DirectoryEntry[size];

    // MP4 mod tag
    // dummy operation to keep valgrind happy
    memset(table, 0, sizeof(DirectoryEntry) * size);

    tableSize = size;
    for (int i = 0; i < tableSize; i++)
        table[i].inUse = FALSE;
}
```

- 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)

```
    ... ...
    FileHeader *mapHdr = new FileHeader;
    FileHeader *dirHdr = new FileHeader;
    ... ...
```

- 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)

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

```
#define NumDirEntries          10
#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)

```
#define FreeMapSector          0
#define DirectorySector        1
```

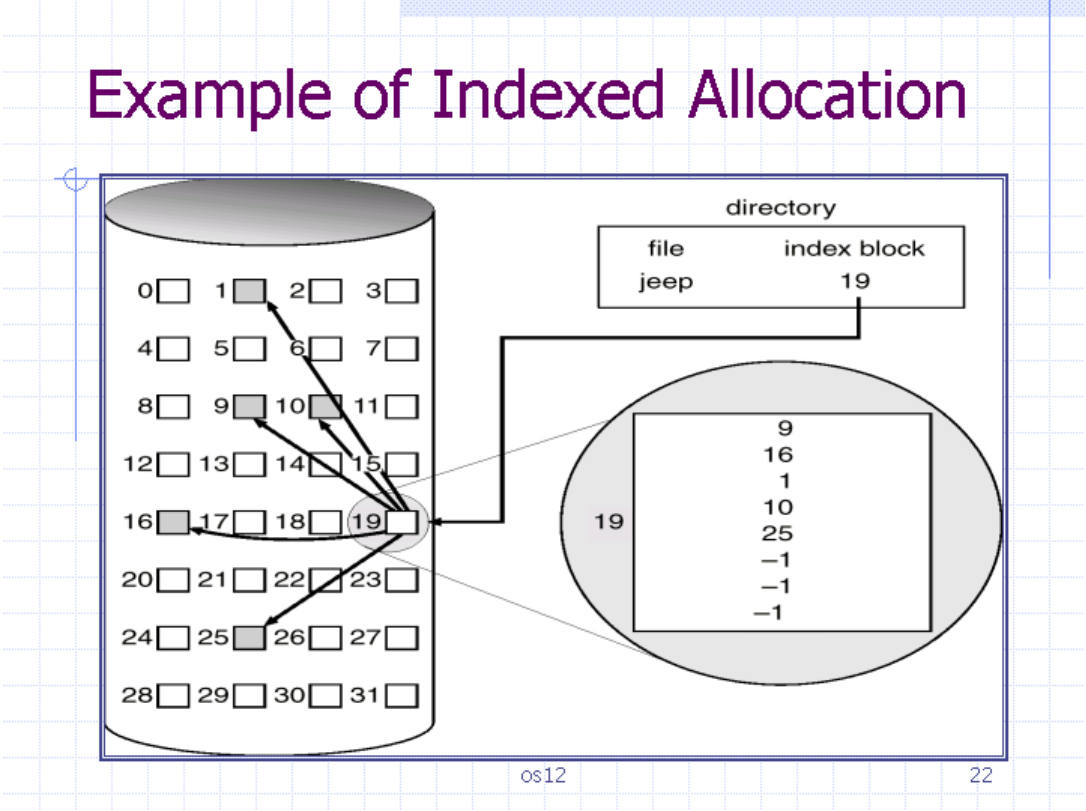
(4) Explain what information is stored in an inode, and use a figure to illustrate the disk allocation scheme of current implementation.

- 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

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

- 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:



(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

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

- 改一下function signature就好了

exception.c

```
#ifndef FILESYS_STUB
case SC_Create:
val = kernel->machine->ReadRegister(4);
{
    char *fileName = &(kernel->machine->mainMemory[val]);
    int initialSize = kernel->machine->ReadRegister(5);
    status = SysCreate(fileName, initialSize);
    kernel->machine->WriteRegister(2,(int)status);
}

kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
kernel->machine->WriteRegister(NextPCReg ,kernel->machine->ReadRegister(PCReg)+4
return;
ASSERTNOTREACHED();
break;

case SC_Open:
val = kernel->machine->ReadRegister(4);
{
    char *fileName = &(kernel->machine->mainMemory[val]);
    status = SysOpen(fileName);
    kernel->machine->WriteRegister(2,(int)status);
}

kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
kernel->machine->WriteRegister(NextPCReg ,kernel->machine->ReadRegister(PCReg)+4
return;
ASSERTNOTREACHED();
break;

case SC_Write:
val = kernel->machine->ReadRegister(4);
{
    char *buffer = &(kernel->machine->mainMemory[val]);
    int initialSize = kernel->machine->ReadRegister(5);
    int id = kernel->machine->ReadRegister(6);
    status = SysWrite(buffer, initialSize, id);
    kernel->machine->WriteRegister(2,(int)status);
}

kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
kernel->machine->WriteRegister(NextPCReg ,kernel->machine->ReadRegister(PCReg)+4
return;
ASSERTNOTREACHED();
break;

case SC_Read:
val = kernel->machine->ReadRegister(4);
{
    char *buffer = &(kernel->machine->mainMemory[val]);
    int initialSize = kernel->machine->ReadRegister(5);
    int id = kernel->machine->ReadRegister(6);
    status = SysRead(buffer ,initialSize ,id);
    kernel->machine->WriteRegister(2,(int)status);
}

kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
kernel->machine->WriteRegister(NextPCReg ,kernel->machine->ReadRegister(PCReg)+4
return;
ASSERTNOTREACHED();
break;

case SC_Close:
{
    int fd = kernel->machine->ReadRegister(4);
    status = SysClose(fd);
    kernel->machine->WriteRegister(2,(int)status);
}

kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
kernel->machine->WriteRegister(NextPCReg ,kernel->machine->ReadRegister(PCReg)+4
return;
ASSERTNOTREACHED();
```

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

filesys.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

filesys.c

```
int FileSystem::CreateFile0(char* name, int initialSize)
{
    return Create(name, initialSize);
}

OpenFileId FileSystem::OpenFile0(char* name)
{
    OpenFile* openPtr = Open(name);
    return PutFileDescriptor(openPtr);
}

int FileSystem::ReadFile0(char* buffer,int size, int id)
{
    if(id <= 0 || id >= MAXOPENFILES) return -1;
    if(fileDescriptorTable[id] == NULL) return -1;
    return fileDescriptorTable[id]->Read(buffer, size);
}

int FileSystem::WriteFile0(char* buffer,int size, int id)
{
    if(id <= 0 || id >= MAXOPENFILES) return -1;
    if(fileDescriptorTable[id] == NULL) return -1;
    return fileDescriptorTable[id]->Write(buffer, size);
}

int FileSystem::CloseFile0(int id)
{
    if(id < 0 || id >= MAXOPENFILES) return -1;
    if(fileDescriptorTable[id] == NULL) return -1;
    delete fileDescriptorTable[id];
    fileDescriptorTable[id] = NULL;
    return 1;
}

OpenFileId FileSystem::PutFileDescriptor(OpenFile* FileDesc)
{
    int cnt = 0;
    while( (++fileDescriptorIndex % MAXOPENFILES == 0)
    || fileDescriptorTable[fileDescriptorIndex] != NULL)
    {
        if(cnt < MAXOPENFILES)
            cnt++;
        else
            return 0;
    }
    fileDescriptorTable[fileDescriptorIndex] = FileDesc;
    return fileDescriptorIndex;
}
```

- 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

```
// 原本是 -2 ，改成-3 多存一個指標
#define NumDirect      ((SectorSize - 3 * sizeof(int)) / sizeof(int))

public:
    FileHeader* getNextFileHeader() { return nextFileHeader;}
    int HeaderLength(); // 新增，回傳Headr有幾格

prinvate:
    FileHeader* nextFileHeader;  // (in-core) 這個要永遠放在第一格!!!!!!
    ...
    int nextFileHeaderSector;    // (in-disk)
```

filehdr.c

```
FileHeader::FileHeader()
{
        nextFileHeader = NULL;
        nextFileHeaderSector = -1;
        numBytes = -1;
        numSectors = -1;
        memset(dataSectors, -1, sizeof(dataSectors));
}

FileHeader::~FileHeader()
{
        // nothing to do now
        if (nextFileHeader != NULL)
                delete nextFileHeader;
}

bool FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
{
// 本層FH要存的Bytes
numBytes = fileSize < MaxFileSize ? fileSize : MaxFileSize;
// 剩下要存的Bytes
fileSize -= numBytes;
// 幫本層分配Data Sectors
numSectors = divRoundUp(numBytes, SectorSize);

// not enough space
if (freeMap->NumClear() < numSectors)
    return FALSE;

for (int i = 0; i < numSectors; i++) {
    dataSectors[i] = freeMap->FindAndSet();
//we checked that there was enough free space, we expect this to succeed
    ASSERT(dataSectors[i] >= 0);
}
// 分配完本層，再多分配一個Link指向下一個FH
if (fileSize > 0) {
    // 為了確認有沒有Free Sector
    nextFileHeaderSector = freeMap->FindAndSet();
    if (nextFileHeaderSector == -1)  // 沒有Free Sector
              return FALSE;
      else {
              nextFileHeader = new FileHeader;
              return nextFileHeader->Allocate(freeMap, fileSize);
    }
  }
return TRUE;
}

void FileHeader::Deallocate(PersistentBitmap *freeMap)
{
    for (int i = 0; i < numSectors; i++) {
      ASSERT(freeMap->Test((int)dataSectors[i])); // ought to be marked!
      freeMap->Clear((int)dataSectors[i]);
        }
        if (nextFileHeaderSector != -1)
        {
                  ASSERT(nextFileHeader != NULL);
                  nextFileHeader->Deallocate(freeMap);
        }
}

void FileHeader::FetchFrom(int sector)
{
    //DEBUG(dbgFile, "Test[J]: kernel->synchDisk->ReadSector");
    kernel->synchDisk->ReadSector(sector, ((char *)this) + sizeof(FileHeader*))
    //DEBUG(dbgFile, "Test[J]: Done");

    if (nextFileHeaderSector != -1)
         {
                   nextFileHeader = new FileHeader;
                   nextFileHeader->FetchFrom(nextFileHeaderSector);
         }
/*
MP4 Hint:
After you add some in-core informations,
you will need to rebuild the header's structure
*/
}
```

```
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

filesys.c

```
#define NumDirEntries         64
```

- 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 $128 * 32 * (32 * 512) \text{Bytes} = 2^{26} \text{Bytes} = 64$ 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**

```
// number of bytes per disk sector
const int SectorSize = 128;
// number of sectors per disk track
const int SectorsPerTrack = 32;
// number of tracks per disk
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

```
../build.linux/nachos -f
echo "================== 1 ======================="
../build.linux/nachos -cp num_1000000.txt /1000000a
../build.linux/nachos -p /1000000a
echo "================== 2 ======================="
../build.linux/nachos -cp num_1000000.txt /1000000b
../build.linux/nachos -p /1000000b
echo "================== 3 ======================="
../build.linux/nachos -cp num_1000000.txt /1000000c
../build.linux/nachos -p /1000000c
echo "================== 4 ======================="
../build.linux/nachos -cp num_1000000.txt /1000000d
../build.linux/nachos -p /1000000d
echo "================== 5 ======================="
../build.linux/nachos -cp num_1000000.txt /1000000e
../build.linux/nachos -p /1000000e
echo "================== 6 ======================="
../build.linux/nachos -cp num_1000000.txt /1000000f
../build.linux/nachos -p /1000000f

echo "================ Last ========================="
../build.linux/nachos -l /
```

- 由於測試過程(cp)很費時，故直接附上截圖


```
[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
```