

MP4 - File System

Cover Page

徐竣霆：50%

江承紘：50%

Trace Code

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

NachOS File System use Bit vector to manage the free spaces. It calls

`Bitmap::FindAndSet()` scan through `numBits` bits from bit 0 to bit `numBits-1`. Use `Test(i)` to check whether bit `i` is occupied. If not, then calls `Mark(i)` to make bit `i` be occupied and return the value `i`. Otherwise, there is no bit can be marked, we return `-1` meaning that there is no free block space.

The information is stored in the `freeMapFile` variable in `FileSystem` class. Once we create a file, we find the sector from `freeMap` in `freeMapFile`

```
205     freeMap = new PersistentBitmap(freeMapFile, NumSectors);
206     sector = freeMap->FindAndSet(); // find a sector to hold the file header
```

```
102 private:
103     OpenFile *freeMapFile; // Bit map of free disk blocks,
104     // represented as a file
105     OpenFile *directoryFile; // "Root" directory -- list of
106     // file names, represented as a file
```

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

$32 * 32 * 128 = 128 * 1024 \text{ bytes} = 128\text{Kbs}$

In `disk.h`

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

In disk.cc

```
27  const int MagicSize = sizeof(int);
28  const int DiskSize = (MagicSize + (NumSectors * SectorSize));
```

Note that the disk size is actually 4bytes + 128Kbs. The goal of the additional 4 bytes is to prevent us from accidentally treating a useful file as a disk so we prepend UNIX file with a magic number.

Kernel::Initialize() 會執行 synchDisk = new SynchDisk();

SynchDisk::SynchDisk() 會執行 disk = new Disk(this);

這裡我們使用 file 去模擬 disk.

Disk::Disk(CallBackObj *toCall) 會根據當前得到的 diskname 去 open file.

如果這個 file 存在，我們去確認 read 後存在 magicNum 的值是否和 MagicNumber 相同，不相同就直接 abort。

若不存在，我們建立這個 file，並將 MagicNumber 寫進這個 file. 最後將 tmp = 0 這個值寫進 file 的尾端，讓 read 不會讀到 EOF，而是我們寫的 0。

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

NachOS use a table to manage directory. The table contains: whether the current directory entry is used, which sector is the file stored in, and the file name.

```
76  private:
77      /*
78          MP4 Hint:
79          Directory is actually a "file", be careful of how it works w
80          Disk part: table
81          In-core part: tableSize
82      */
83
84      int tableSize;           // Number of directory entries
85      DirectoryEntry *table;   // Table of pairs:
86      | | | | | | | | | | | | // <file name, file header location>
```

```

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

```

Similar to (1), the information is stored in the `directoryFile` variable in `FileSystem` class.

Note that once we initialize a file system, no matter we need to format it or not, the `directoryFile` and `freeMapFile` will be specified.

```

117     freeMapFile = new OpenFile(FreeMapSector);
118     directoryFile = new OpenFile(DirectorySector);
119
120     // Once we have the files "open", we can write the initial version
121     // of each file back to disk. The directory at this point is completely
122     // empty; but the bitmap has been changed to reflect the fact that
123     // sectors on the disk have been allocated for the file headers and
124     // to hold the file data for the directory and bitmap.
125
126     DEBUG(dbgFile, "Writing bitmap and directory back to disk.");
127     freeMap->WriteBack(freeMapFile); // flush changes to disk
128     directory->WriteBack(directoryFile);
129
130     if (debug->IsEnabled('f'))
131     {
132         freeMap->Print();
133         directory->Print();
134     }
135     delete freeMap;
136     delete directory;
137     delete mapHdr;
138     delete dirHdr;
139 }
140 else
141 {
142     // if we are not formatting the disk, just open the files representing
143     // the bitmap and directory; these are left open while Nachos is running
144     freeMapFile = new OpenFile(FreeMapSector);
145     directoryFile = new OpenFile(DirectorySector);
146 }
147 }

```

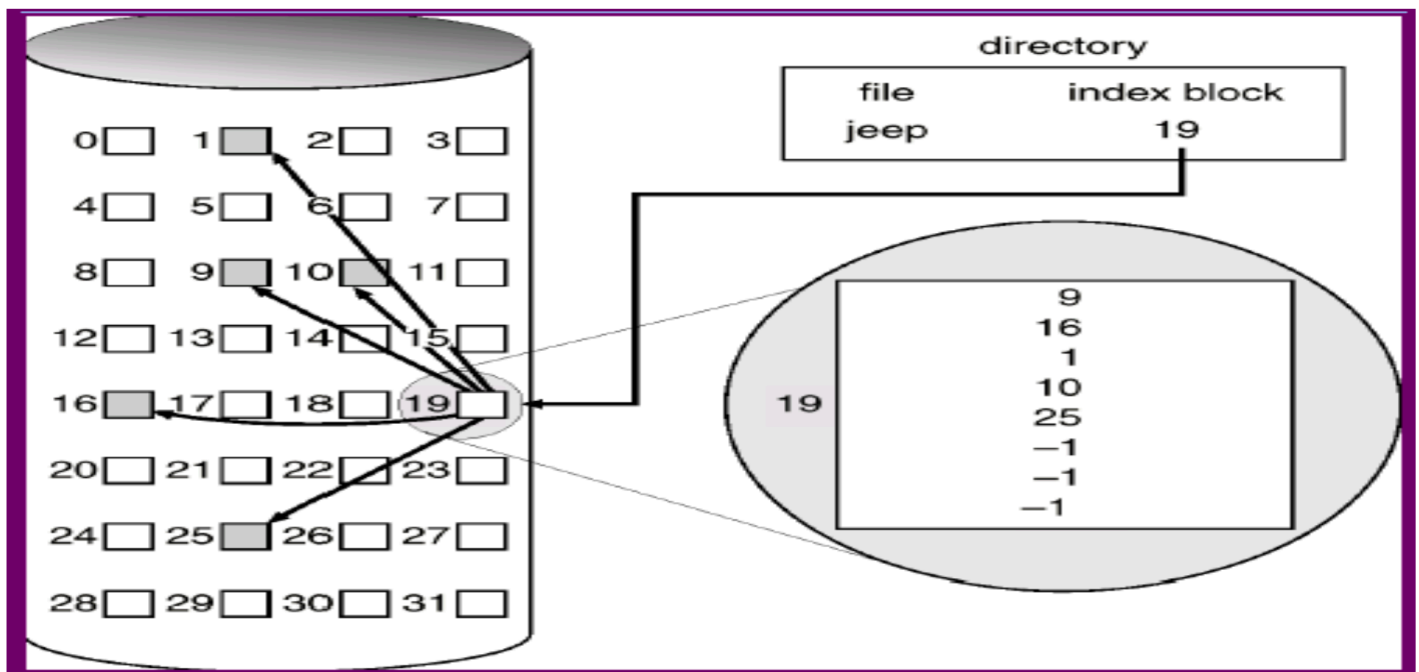
(4) What information is stored in an inode? Use a figure to illustrate

the disk allocation scheme of the current implementation.

The class `Fileheader` is the i-node. It contains: number of bytes, number of sectors and the disk sector number for each data block.

```
64 private:
65     /*
66     MP4 hint:
67     You will need a data structure to store more information in a header.
68     Fields in a class can be separated into disk part and in-core part.
69     Disk part are data that will be written into disk.
70     In-core part are data only lies in memory, and are used to maintain the data structure of this class.
71     In order to implement a data structure, you will need to add some "in-core" data
72     to maintain data structure.
73
74     Disk Part - numBytes, numSectors, dataSectors occupy exactly 128 bytes and will be
75     written to a sector on disk.
76     In-core part - none
77
78     */
79
80     int numBytes;           // Number of bytes in the file
81     int numSectors;         // Number of data sectors in the file
82     int dataSectors[NumDirect]; // Disk sector numbers for each data
83                               // block in the file
84 };
```

Since we have the variable `dataSectors`, the disk allocation scheme is index allocation.



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

In (4), `NumDirect` is $(128 - 2 * 4) / 4 = 30$. The `dataSectors` array has 30 entry. Each `FileHeader` also contains `numBytes = 4bytes` and `numSectors = 4bytes`. The sector size is 128 bytes defined in `disk.h`. Therefore, the maximum file size is $128 * (30 + 1 + 1) = 4KB$.

Implementation

Part II (1)

首先將 `SC_Create` , `SC_Open` , `SC_Read` , `SC_Write` 和 `SC_Close` 從 MP1 的 `exception.cc` 幾乎複製過來。這裡以 `SC_Create` 為例，要更改的地方是另外宣告一個變數 `sz` 代表要建立的檔案大小。

```
1  case SC_Create:
2      DEBUG(dbgSys, "Create File.\n");
3      val = kernel->machine->ReadRegister(4);
4      {
5          char *filename = &(kernel->machine->mainMemory[val]);
6          //cout << filename << endl;
7          int sz = kernel -> machine -> ReadRegister(5);
8          status = SysCreate(filename, sz);
9          kernel->machine->WriteRegister(2, (int) status);
10     }
11     kernel->machine->WriteRegister(PrevPCReg, kernel->machine
12                                     ->ReadRegister(PCReg));
13     kernel->machine->WriteRegister(PCReg, kernel->machine
14                                     ->ReadRegister(PCReg) + 4);
15     kernel->machine->WriteRegister(NextPCReg, kernel->machine
16                                     ->ReadRegister(PCReg)+4);
17     return;
18     ASSERTNOTREACHED();
19     break;
```

由於我們在 `exception.cc` 定義了 `SysCreate` , `SysOpen` , `SysRead` , `SysWrite` 和 `SysClose` . 由於這次我們要呼叫的是 `fileSystem` 的 `method`, 不同於 MP1 交給 UNIX `system` 管理，因此每個函數都要個別實作。

我們在 `filesys.h` 的 `public` 定義了 `OpenFile` 指標讓 `filesys.h` 可以直接呼叫存取。

```
1  | OpenFile *opfile;
```

- `SysCreate` 直接呼叫 `fileSystem` 的 `Create` 即可
- `SysOpen` 先透過 `opfile` 接收開啟的檔案。由於 `spec` 中指示一次最多只會開啟一個檔案，因此我們 `return 1` , 作為他的 `file descriptor`.
- `SysRead` 透過 `opfile` 做 `Read()` 並直接回傳值
- `SysWrite` 觀念同 `SysRead`
- `SysClose` 將 `opfile` `delete` 後將其值設為 `NULL`

```

1  int SysCreate(char* filename, int sz) {
2      kernel->fileSystem->Create(filename, sz);
3  }
4  OpenFileId SysOpen(char* name) {
5      kernel->fileSystem->opfile = kernel->fileSystem->Open(name);
6      return 1;
7  }
8  int SysRead(char* buffer, int sz, OpenFileId id) {
9      return kernel->fileSystem->opfile->Read(buffer, sz);
10 }
11 int SysWrite(char* buffer, int sz, OpenFileId id) {
12     return kernel->fileSystem->opfile->Write(buffer, sz);
13 }
14 int SysClose(int id) {
15     delete kernel->fileSystem->opfile;
16     kernel->fileSystem->opfile = NULL;
17     return 1;
18 }

```

Part II (2)

我們使用 linked list scheme 將 FS 增加到 32KB. 必需記錄下一個 sector 的 ID 使我們在 Fetch 時可以找到正確的 file 並將每個 file header 給 link 起來，所以在 filehdr.h 新增 nextSectorID 和 next 兩個參數。

```

1  private:
2      int nextSectorID;
3      FileHeader* next;

```

由於新增了 nextSectorID，dataSectors 少了一個 byte. 因此要修改 NumDirect 的值。不需特別為 next 分配空間是因為這個指標只有在 memory 才有用，沒必要把他一起存到 disk 中。

```

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

```

Constructor 為 filehdr.h 中的變數 nextSectorID = -1 和 next = NULL 初始化

```

1  FileHeader::FileHeader()
2  {
3      numBytes = -1;
4      numSectors = -1;
5      memset(dataSectors, -1, sizeof(dataSectors));
6      nextSectorID = -1;
7      next = NULL;
8  }

```

Destructor 將 next delete

```

1  FileHeader::~~FileHeader()
2  {
3      delete next;
4  }

```

Allocate() 因為這次收到的資料大小可能超過一個 file header 所提供最大的空間，因此要先對 numBytes 和 numSectors 取 min. 如果收到的 fileSize 大於一個 file header 可以儲存空間的最大值，則透過 next 指標遞迴呼叫 Allocate

```

1  bool FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
2  {
3      numBytes = fileSize < MaxFileSize ? fileSize : MaxFileSize;
4      numSectors = divRoundUp(fileSize, SectorSize) < NumDirect ?
5                      divRoundUp(fileSize, SectorSize) : NumDirect;
6      if (freeMap->NumClear() < numSectors)
7          return FALSE; // not enough space
8
9      for (int i = 0; i < numSectors; i++)
10     {
11         dataSectors[i] = freeMap->FindAndSet();
12         // since we checked that there was enough free space,
13         // we expect this to succeed
14         ASSERT(dataSectors[i] >= 0);
15     }
16     if (fileSize > numBytes) {
17         next = new FileHeader();
18         nextSectorID = freeMap->FindAndSet();
19         return next -> Allocate(freeMap, fileSize - numBytes);
20     }
21     return TRUE;
22 }

```

Deallocate() 的概念與 Allocate() 相似，只是將 Allocate() 反著做。

```

1 void FileHeader::Deallocate(PersistentBitmap *freeMap)
2 {
3     FileHeader *cur = this;
4     do {
5         for (int i = 0; i < cur -> numSectors; i++)
6         {
7             ASSERT(freeMap->Test((int)cur -> dataSectors[i]));
8             // ought to be marked!
9             freeMap->Clear((int)cur -> dataSectors[i]);
10        }
11        cur = cur -> next;
12    } while(cur != NULL);
13 }

```

FetchFrom() 透過 sector 一個個讀取 disk 的資料。值得注意的是 Line 10, 我們為了保證 sector 是 -1 時, cur 必為 NULL, 因此一但 sector == -1, 我們就不再 new 新的 FileHeader 給 cur

```

1 void FileHeader::FetchFrom(int sector)
2 {
3     FileHeader* cur = this;
4
5     while(true) {
6         // cout << "sector = " << sector << "\n";
7         kernel->synchDisk->ReadSector(sector, (char *)cur);
8         // cout << "nB = " << cur -> numBytes << "\n";
9         sector = cur -> nextSectorID;
10        if (sector == -1) break;
11        cur -> next = new FileHeader;
12        cur = cur -> next;
13    }
14 }

```

WriteBack() 的概念與 FetchFrom() 相似, 只是將 FetchFrom() 反著做。由於在 FetchFrom() 保證了 sector == -1 與 cur == NULL 的配對, 若將 while(sector != -1) 改成 while(cur != NULL) 的執行結果會相同。


```

1 void FileHeader::WriteBack(int sector)
2 {
3     FileHeader* cur = this;
4     while(sector != -1) {
5         kernel->synchDisk->WriteSector(sector, (char *)cur);
6         sector = cur -> nextSectorID;
7         cur = cur -> next;
8     }
9 }

```

ByteToSector() 根據 offset 的大小去找到資料應該在哪個 sector. 首先要先判斷 offset 長度應該對應到哪個 file header, 找到正確的 file header 再直接 return 對應的 dataSectors[offset / SectorSize]

```

1 int FileHeader::ByteToSector(int offset)
2 {
3     FileHeader* cur = this;
4     while (offset >= MaxFileSize) {
5         cur = cur -> next;
6         offset -= MaxFileSize;
7     }
8     return (cur -> dataSectors[offset / SectorSize]);
9 }

```

FileLength() 計算 file 總共有幾個 Bytes. 由於我們用 linked list scheme, 因此 go through 所有的 file header 才能計算出正確的大小。

```

1 int FileHeader::FileLength()
2 {
3     int res = 0;
4     FileHeader* cur = this;
5     while(cur != NULL) {
6         res += cur -> numBytes;
7         cur = cur -> next;
8     }
9     return res;
10 }

```

Part III

實作這段的主要想法是找出給定的 name 的倒數第二個 file 和最後一個 file. 倒數第二個 file 必為 directory, 而最後一個 file 在 mkdir 時為 directory, 其他則為 file. 如果我們能找到這兩個 file, 則可以把一切的操作視為倒數第二個 file 為 root, 而最後一個 file 則是我們要在

root 底下建立的 file 或 directory.

修改前的 code 預設是所有的操作都在 root 下運行，因為 directory 呼叫的 FetchFrom() 所帶的參數永遠是 directoryFile。我們只要尋找到倒數第二個 file 並呼叫 directory->FetchFrom()，同時正確地開啟最後一個 file 並將他存到 opFile 中即可。

底下介紹細部實作：

main.cc

在 CreateDirectory 呼叫一個自己實作的 method CreateDirectory(name)

```
1 static void CreateDirectory(char *name)
2 {
3     kernel -> fileSystem -> CreateDirectory(name);
4 }
```

為了支援使用者可能要 list 所有 subdirectory 與否，我們新增兩個參數到 List() method 中

```
1 if (dirListFlag) {
2     kernel->fileSystem->List(listDirectoryName, recursiveListFlag);
3 }
```

fileSYS.h

除了 fileSYS.h 中的 List()，同樣地我們修改 Create 為了判別是否要將 directory 屬性指派給當前建立的新檔案。

```
1 bool Create(char *name, int initialSize, bool isdir = 0);
2 void List(char* rootName, bool recursiveListFlag); // List all the fi
```

fileSYS.cc

為了讓每個 directory 底下可以有 64 個 entry 而非只有 10 個，我們先將 NumDirEntries 改為 64

```
1 #define NumDirEntries 64
```

若我們要建立 directory, 首先要先建立一個空的 general file 並將建立好的 general file 打開。因為要賦予這個 file directory 的屬性，我們 new 一個新的 Directory 並呼叫 WriteBack 將該 directory 喂給 general file, 如此一來，一個 directory file 就完成了。

```
1 void FileSystem::CreateDirectory(char *name) {  
2     Create(name, DirectoryFileSize, true);  
3     OpenFile* opFile = Open(name);  
4     Directory* dir = new Directory(NumDirEntries);  
5     dir -> WriteBack(opFile);  
6     delete opFile;  
7     delete dir;  
8 }
```

Create() 實作概念如前述提及，我們使用 FindParentAndChild() 找到倒數第二和最後一個 file 並在回傳前將 directory fetch 完畢以及 opFile 開好。回到 Create() 後即可繼續進行類似原本在 root 底下的操作。要注意的是 name 要改成最後一個 file 的名字 (nextname), WriteBack 則是要寫回我們開好的 opFile

可以注意到我們在 Add 新增了一個參數 isdir 去檢測目前要建立的檔案為普通檔案或 directory file

```

1  bool FileSystem::Create(char *name, int initialSize, bool isdir)
2  {
3      ...
4      OpenFile *opFile = directoryFile;
5      ...
6      char* nextName = new char[300];
7      ...
8      FindParentAndChild(name, directory, opFile, nextName);
9
10     if (directory->Find(nextName) != -1)
11         success = FALSE; // file is already in directory
12     else {
13         if (sector == -1)
14             success = FALSE;
15         else if (!directory->Add(nextName, sector, isdir))
16             success = FALSE; // no space in directory
17         else {
18             ...
19             directory->WriteBack(opFile);
20         }
21     }
22     ...
23     return success;
24 }
25

```

Open() 和 Create() 的概念相似，核心也是透過 FindParentAndChild() 找到對應的 directory 和 file 並 open file.

```

1  OpenFile * FileSystem::Open(char *name)
2  {
3      Directory *directory;
4      OpenFile *openFile = directoryFile;
5      int sector;
6      char* nextName;
7
8      DEBUG(dbgFile, "Opening file" << name);
9
10     FindParentAndChild(name, directory, openFile, nextName);
11
12     sector = directory->Find(nextName);
13     if (sector >= 0)
14         openFile = new OpenFile(sector); // name was found in directo
15     delete directory;
16     return openFile; // return NULL if not found
17 }

```

FindParentAndChild() 從 name 提取我們要的資訊：

- dir: fetch 完畢的倒數第二個 file
- opFile: open 完畢的最後一個 file
- nextName: 最後一個 file 的名字

table 存放 name parse 完畢的字串。例如：name = /t0/aa/f0，則 table[0] = "t0"，table[1] = "aa"，table[2] = "f0"

Line 20 的 while 迴圈就是不斷檢查目前 fetch 和 open 的是否為我們最終要的 directory 和 file.

```

1 void FindParentAndChild(char* name, Directory*& dir,
2                          OpenFile*& opFile, char*& nextName)
3 {
4     char table[128][20] = {0};
5     int idx = 0, curpos = 0;
6     for (int i = 1; name[i]; ++i) {
7         if (name[i] == '/') {
8             ++i;
9             ++idx;
10            curpos = 0;
11        }
12        table[idx][curpos++] = name[i];
13    }
14
15    nextName = table[0];
16
17    dir = new Directory(NumDirEntries);
18    dir -> FetchFrom(opFile);
19    int curidx = 0;
20    while(curidx < idx) {
21        int nextSec = dir -> Find(nextName);
22        opFile = new OpenFile(nextSec);
23        dir -> FetchFrom(opFile);
24        ++curidx;
25        nextName = table[curidx];
26    }
27    return;
28 }

```

List() 預設 recursiveListFlag 為 false 表示只會列出當前目錄底下的 file. if-else 特別將若當前要列出的目錄做特別判斷，若為 / 則照原本的 fetch 即可，否則呼叫 open 打開最後一個 file 再用 directory 去 fetch.

```

1 void FileSystem::List(char* rootName, bool recursiveListFlag = false)
2 {
3     Directory *directory = new Directory(NumDirEntries);
4     if (strcmp(rootName, "/") == 0)
5         directory->FetchFrom(directoryFile);
6     else {
7         OpenFile* opFile = Open(rootName);
8         directory -> FetchFrom(opFile);
9     }
10    directory->List(recursiveListFlag, 0);
11    delete directory;
12 }

```

directory.h

在 DirectoryEntry 新增 isdir 紀錄當前要建立的 file 是否為 directory.

```
1  class DirectoryEntry
2  {
3      public:
4          bool inUse;           // Is this directory entry in use?
5          int sector;           // Location on disk to find the
6                                // FileHeader for this file
7          char name[FileNameMaxLen + 1]; // Text name for file, with +1 for
8                                // the trailing '\0'
9          bool isdir;
10 };
```

為了判斷新增的 file 是否為 directory, 我們新增 isdir 在 (Add) 中。另外和 filesys.h 一樣, 我們新增兩個參數到 List() method 中, 其中 level 代表進度到第 level 層的 directory, 為了排版需要。

```
1  bool Add(char *name, int newSector, bool isdir); // Add a file name i
2  void List(bool recursiveListFlag, int level);    // Print the names of
```

directory.cc

Add() 中新增 isdir 參數和 table[i].isdir = isdir

```
1  bool Directory::Add(char *name, int newSector, bool isdir)
2  {
3      ...
4      for (int i = 0; i < tableSize; i++)
5          if (!table[i].inUse) {
6              ...
7              table[i].isdir = isdir;
8              return TRUE;
9          }
10     return FALSE; // no space.  Fix when we have extensible files.
11 }
```

List() 中的 offset 表示要空格的長度。下方透過 isdir 判斷要印出的是 [D] 或 [F]

```

1 void Directory::List(bool recursiveListFlag, int level = 0)
2 {
3     char offset[level*4 + 1];
4     for (int i = 0; i < level*4; ++i)
5         offset[i] = ' ';
6     offset[level*4] = 0;
7     if (recursiveListFlag == false) {
8         for (int i = 0; i < tableSize; i++) {
9             if (table[i].inUse)
10                printf("%s%s\n", offset, table[i].name);
11        }
12    } else {
13        for (int i = 0; i < tableSize; ++i) {
14            if (table[i].inUse) {
15                if (table[i].isdir)
16                    printf("%s[D] ", offset);
17                else
18                    printf("%s[F] ", offset);
19                printf("%s\n", table[i].name);
20                if (table[i].isdir) {
21                    Directory *nextdir = new Directory(tableSize);
22                    nextdir -> FetchFrom(&OpenFile(table[i].sector));
23                    nextdir -> List(recursiveListFlag, level + 1);
24                }
25            }
26        }
27    }
28 }
29

```

Bonus I

因為 $128\text{KB} * 512 = 64\text{MB}$, 所以我們將 SectorsPerTrack 乘上 512 即可。

```

1 const int SectorsPerTrack = 32 * 512;
2 // number of sectors per disk track

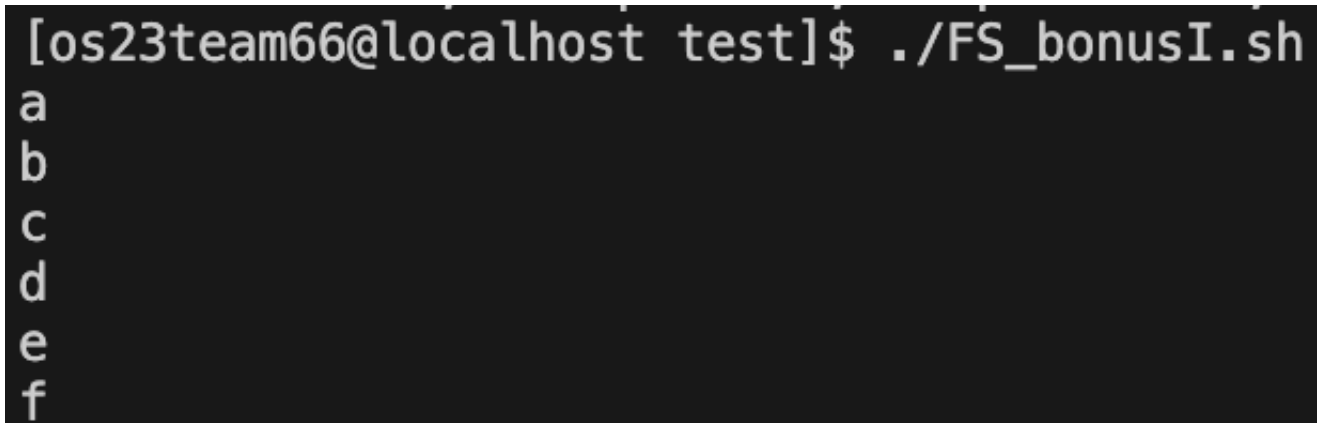
```

我們另外寫了一個 num_1000000.txt 包含 1 到 1000000 的數字約 9.6MB. 並將他複製 6 次後列出 / 底下的檔案確認為 6 個檔案(FS_bonusI.sh)。以此證明我們的改寫是成功的。


```

1  /* FS_bonusI.sh */
2
3  ../build.linux/nachos -f
4  ../build.linux/nachos -cp num_1000000.txt /a
5  ../build.linux/nachos -cp num_1000000.txt /b
6  ../build.linux/nachos -cp num_1000000.txt /c
7  ../build.linux/nachos -cp num_1000000.txt /d
8  ../build.linux/nachos -cp num_1000000.txt /e
9  ../build.linux/nachos -cp num_1000000.txt /f
10 ../build.linux/nachos -l /

```



```

[os23team66@localhost test]$ ./FS_bonusI.sh
a
b
c
d
e
f

```

Bonus II

我們寫了一個 shell script 將 num_100.txt, num_1000.txt 和 num_1000000.txt 複製到 nachOS FS 裡，同時我們微調 -D 會執行的指令。另外，我們在 filehdr.h 另外實作一個和 FileLength() 相似的 method int FileBlock() 計算該 file 使用了多少個 sector.

```

1  /* FS_bonusII.sh */
2
3  ../build.linux/nachos -f
4  ../build.linux/nachos -cp num_100.txt /a
5  ../build.linux/nachos -cp num_1000.txt /b
6  ../build.linux/nachos -cp num_1000000.txt /c
7  ../build.linux/nachos -l /
8  echo =====
9  ../build.linux/nachos -D

```

每個 file 所佔用的 total sector 數量愈多，total header size 的大小就愈大。由下方部分的結果可以得知愈小的檔案其 total header size 會愈小。

```
1  /* out.txt */
2
3  a
4  b
5  c
6  =====
7  Directory contents:
8  Name: a, First Sector: 541
9  FileHeader contents.  File size: 1000.
10 File blocks(Total of number of sectors): 8
11 Name: b, First Sector: 550
12 FileHeader contents.  File size: 10000.
13 File blocks(Total of number of sectors): 79
14 Name: c, First Sector: 632
15 FileHeader contents.  File size: 10000000.
16 File blocks(Total of number of sectors): 78125
17
18
```