CS342301: Operating System
MP4: File System
Team Number: 71

Team Members & Contributions

- Members: 108022138 楊宗諺、111062649 許峻源
- Contributions: 合力製作MP4 report、討論如何實作multilevel indexed scheme以及subdirectory structure。

Trace Codes

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

➔ NachOS的file system是透過bit map的方式來管理free blocks，disk上physically以sector作為最小的storage unit。sectors從0開始編號，在bit map中記錄著所有sectors的使用情況，0代表目前sector尚未被使用，1則代表sector已被使用。

➔ 而在disk format時，我們可以觀察到NachOS的file system實際上是把bit map當成是file來管理，畢竟bit map的資訊也要存在disk上。在一開始就為bit map創建了file header，用bit map的file header去allocate data sectors來存儲bit map的資訊，而bit map的file header則是存儲在predefine好的sector上 (sector 0)。

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

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

➔ 而在需要尋找free sectors的空間時，NachOS的file system會先把bit map的資訊從disk fetch到memory裡，用變數freeMap進行操作。而class Bitmap同時也定義了FindAndSet()函數，主要是用來掃過所有的sector bit，找尋有沒有尚未被使用的sector，如果有則assign sector給需要的檔案，然後flip bit。

```
int Bitmap::FindAndSet()
{
    for (int i = 0; i < numBits; i++)
    {
        if (!Test(i))
        {
            Mark(i);
            return i;
        }
    }
    return -1;
}
```

➔ 最後操作完freeMap之後，會再把修改過後的資訊flush回disk上，避免下次從disk fetch出bit map的資訊的時候有資料不一致的狀況。

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

➔ 從disk.h可以知道，NachOS的disk上主要有32個tracks，每圈track上共有32個sectors，sector大小為128 bytes。

```
50 const int SectorSize = 128;      // number of bytes per
51 const int SectorsPerTrack  = 32;     // number of sector
52 const int NumTracks = 32;        // number of tracks per
53 const int NumSectors = (SectorsPerTrack * NumTracks);
```

➔ 而在disk.cc中有明確定義了disk的大小，其定義方式如下：

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

➔ MagicNumber的用處主要是為了避免將有用的檔案視作是disk，否則可能會破壞檔案的內容。因此， NachOS中定義disk的大小也包含了MagicNumber的大小， 全部加在一起總共是 4+32*32*128 bytes (大約是128KB)。

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

➔ NachOS的file system主要是將directory當作是file來管理。其管理方式跟管理檔案的方式相同，一樣會有file header跟data sectors。directory的file header中存儲的資訊為其data sectors的位置，這點跟一般檔案的file header相同，只是在其data sectors上存儲的並不是檔案的內容， 而是當前directory下所有檔案的file headers (我們可以從建立檔案的過程中觀察到這件事)。

```
hdr = new FileHeader;
if (!hdr->Allocate(freeMap, initialSize))
    success = FALSE; // no space on disk for
else
{
    success = TRUE;
    // everthing worked, flush all changes b
    hdr->WriteBack(sector);
    directory->WriteBack(subDirFile);
    freeMap->WriteBack(freeMapFile);
}
delete hdr;
```

➔ 從上圖可以發現，在檔案的file header allocate完其data sectors的位置後，有將檔案的file header的資訊flush回directory的data sector上 (因為file header的資訊有做更動)，所以我們可以知道directory的data sector上存儲的會是檔案的file header的資訊。

➔ 和所有檔案的file header一樣，directory的file header也必須存儲在某個sector上，而從disk的format中我們可以知道directory的file header在disk format的時候已被allocate在特定的sector上 (sector 1)。

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

```
mapHdr->WriteBack(FreeMapSector);
dirHdr->WriteBack(DirectorySector);
```

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

➔ i-node也就是file header，上面存儲的內容有檔案的大小、用了多少data sectors以及data sectors的位置。而從file header的allocate以及deallocate，我們也可以明顯觀察到NachOS的file system在尚未修改之前採取的是one level indexed allocation scheme。示意圖如下：

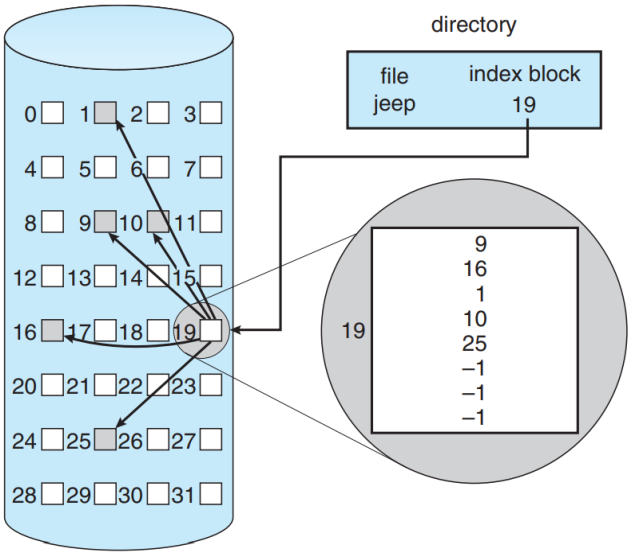

Figure 12.8 Indexed allocation of disk space.

➔ 藉由file header，或是說index sector上specify好的data sectors的位置的資訊， 我們可以找到檔案的內容是存儲在哪幾個data sectors上，進而可以對檔案進行讀寫的操作。

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

➔ 因為目前file system是採取one level indexed allocation scheme，也因為一個file header的大小被限制在一個sector上，所以扣除掉要存儲檔案大小以及用了幾個data sectors的資訊的空間之後，只能夠再存儲(128-4*2)/4=30個data sectors的位置，這部分從filehdr.h中對NumDirect的定義可以知道。

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

➔ 因此，實際用來儲存檔案內容的data sectors的數量最多只有30個， 所以在當前allocation scheme下允許的最大檔案大小即為30*128bytes=3.75KB。

Implementation Details

1. Files Modified

- NachOS-4.0_MP4/code/threads/main.cc
- NachOS-4.0_MP4/code/userprog/exception.cc
- NachOS-4.0_MP4/code/userprog/ksyscall.h
- NachOS-4.0_MP4/code/userprog/syscall.h
- NachOS-4.0_MP4/code/filesys/filesys.h

- NachOS-4.0_MP4/code/filesys/filesys.cc
- NachOS-4.0_MP4/code/filesys/filehdr.h
- NachOS-4.0_MP4/code/filesys/filehdr.cc
- NachOS-4.0_MP4/code/filesys/directory.h
- NachOS-4.0_MP4/code/filesys/directory.cc

2. File System Calls

- **Exception.cc — modified**
  我們在ExceptionHandler()中多加入了SC_Open, SC_Read, SC_Write, SC_Close的case，能夠分別呼叫ksyscall.h中不同的system call的API。同時，我們也在SC_Create中新增了file size的argument。

- **Ksyscall.h — modified**
  這邊我們新增了五個system call的API，分別可以呼叫file system中相關file操作的函數。

```
#ifndef FILESYS_STUB
int SysCreate(char *filename, int size) {
    // return value
    // 1: success
    // 0: failed
    return kernel->fileSystem->Create(filename, size);
}

OpenFileId SysOpen(char* filename) {
    if(kernel->fileSystem->Open(filename) == NULL)
        return 0;
    return 1;
}

int SysRead(char* buffer, int size, OpenFileId id) {
    return kernel->fileSystem->getCurrentFile()->Read(buffer, size);
}

int SysWrite(char* buffer, int size, OpenFileId id) {
    return kernel->fileSystem->getCurrentFile()->Write(buffer, size);
}

int SysClose(OpenFileId id) {
    kernel->fileSystem->deleteCurrentFile();
    return 1;
}
#endif
```

其中因為spec上有表明說同一時間最多只會有一個file被開啟，所以在file system的class中我們新增了OpenFile* currentFile，用來紀錄當前是哪一個file被開啟。同時我們也在class中新增了兩個函數用來存取currentFile (getCurrentFile()和deleteCurrentFile())。接著所有的file operations都是直接透過OpenFile的API去執行 (OpenFile::Read(), OpenFile::Write())。

```
OpenFile *currentFile;
```

```
OpenFile* getCurrentFile() {
    return currentFile;
}

void deleteCurrentFile() {
    delete currentFile;
    currentFile = NULL;
}
```

3. Increase the Maximum Allowed File Size to 32KB.

➔ 因為當前one level的allocation scheme限制file size不能超過4KB，所以我們這邊的實作主要是將one level的allocation scheme轉換成multilevel indexed allocation scheme，讓單一file可以取得更多的data sectors。

- **Filehdr.h — modified**
  這邊除了原先define好的MaxFileSize，我們多define了LevelTwoSize，代表如果使用了two level的indexed allocation，最多能允許的file size。

```
#define NumDirect ((SectorSize - 2 * sizeof(int)) / sizeof(int))
#define LevelOneSize (NumDirect * SectorSize)
#define LevelTwoSize (NumDirect * NumDirect * SectorSize)
```

- **FileHeader::Allocate(PersistentBitmap* freeMap, int fileSize) — modified**
  為了支援更大的file size，我們在Allocate()函數中實作了multilevel indexed scheme。原先one level的實作，index sector中存儲的均為file的data sectors的實際位置資訊 (在哪個sector上) 。而在multilevel indexed scheme中，第一層index sector存儲的會是下一層index sector的位置，直到最後一層index sector才會實際記錄file data sectors的位置。

  實作方式如下：

```
if(fileSize <= LevelOneSize) {
    for (int i = 0; i < numSectors; i++)
    {
        dataSectors[i] = freeMap->FindAndSet();
        // since we checked that there was enough free space,
        // we expect this to succeed
        ASSERT(dataSectors[i] >= 0);
    }
```

➔ 檔案大小小於LevelOneSize，代表說檔案大小不超過4KB， 直接分配data sectors給檔案內容。

```
} else if(fileSize > LevelOneSize
                && fileSize <= LevelTwoSize) {
    for(int i=0; fileSize; ++i) {
        dataSectors[i] = freeMap->FindAndSet();
        ASSERT(dataSectors[i] >= 0);

        FileHeader* Hdr = new FileHeader();

        if(fileSize > LevelOneSize) {
            Hdr->Allocate(freeMap, LevelOneSize);
            fileSize -= LevelOneSize;
        } else {
            Hdr->Allocate(freeMap, fileSize);
            fileSize = 0;
        }

        numSectors = i;
        Hdr->WriteBack(dataSectors[i]);
        delete Hdr;
    }
```

➔ 檔案大小超過4KB，分配data sectors紀錄第二層的index sector的位置。值得注意的是，在第二層index sector分配完其data sectors後，第一層index sector會把第二層index sector的資訊寫回其對應的sector上。

```
} else if(fileSize > LevelTwoSize) {
    for(int i=0; fileSize; ++i) {
        dataSectors[i] = freeMap->FindAndSet();
        ASSERT(dataSectors[i] >= 0);

        FileHeader* Hdr = new FileHeader();

        if(fileSize > LevelTwoSize) {
            Hdr->Allocate(freeMap, LevelTwoSize);
            fileSize -= LevelTwoSize;
        } else {
            Hdr->Allocate(freeMap, fileSize);
            fileSize = 0;
        }

        numSectors = i;
        Hdr->WriteBack(dataSectors[i]);
        delete Hdr;
    }
}
```

➔ 檔案大小超過LevelTwoSize，再多新增一層index sector的level，概念跟一層變兩層的狀況一樣。

- **FileHeader::Deallocate(PersistentBitmap* freeMap) — modified**
  因為allocation的scheme改變，deallocate這邊也必須做出相對應的調整。對於two level以上的allocation，會先讓下一層的index sector先去deallocate，最後再利用freeMap去clear下一層index sector所在的sector。而對於最後一層index sector或是one level的allocation，就直接用freeMap去clear其data sectors即可。

```
134 void FileHeader::Deallocate(PersistentBitmap *freeMap)
135 {
136     if(numBytes > LevelOneSize) {
137         for(int i=0; i<numSectors; ++i) {
138             FileHeader* Hdr = new FileHeader();
139             Hdr->FetchFrom(dataSectors[i]);
140             Hdr->Deallocate(freeMap);
141             freeMap->Clear((int)dataSectors[i]);
142             delete Hdr;
143         }
144     } else {
145         for (int i = 0; i < numSectors; i++)
146         {
147             ASSERT(freeMap->Test((int)dataSectors[i]));
148             freeMap->Clear((int)dataSectors[i]);
149         }
150     }
151 }
```

- **FileHeader::ByteToSector(int offset) — modified**
  此函數主要是負責virtual address和physical address的轉換。而因為allocation scheme的改變，這邊主要就是順著index sector specify的位置下去找出實際的physical sector。

```
199 int FileHeader::ByteToSector(int offset)
200 {
201     FileHeader* Hdr = new FileHeader();
202     int sector;
203 
204     if(numBytes <= LevelOneSize) {
205         return dataSectors[offset/SectorSize];
206     } else if(numBytes > LevelOneSize && numBytes <= LevelTwoSize) {
207         sector = offset/LevelOneSize;
208         Hdr->FetchFrom(dataSectors[sector]);
209         return Hdr->ByteToSector(offset-sector*LevelOneSize);
210     } else if(numBytes > LevelTwoSize) {
211         sector = offset/LevelTwoSize;
212         Hdr->FetchFrom(dataSectors[sector]);
213         return Hdr->ByteToSector(offset-sector*LevelTwoSize);
214     }
215     return -1;
216 }
```

4. Subdirectory Structure

➔ 因為在NachOS的file system中，directory是用file的形式來implement的，所以我們這邊實作subdirectory structure主要是讓directory的file header也可以存儲subdirectory的file header的資訊 (原先只有存儲檔案的file header)。

- **Filesys.cc — modified**
  為了讓每一個directory都可以儲存最多64個檔案或是subdirectories的資訊，我們先在Filesys.cc中將directory entry的數量從10增加到64。

```
65 #define NumDirEntries 64
```

- **FileSystem::Create(char* name) — modified**
  這邊因為開始有了subdirectories的概念，所以我們在create一個檔案的時候，必須先根據path找到對應的directory的位置，然後再把檔案建立在當前的directory之下。

```
char* temp = new char[MaxPathLength];
strcpy(temp, name);
char* subDirName = strtok(temp, "/");

while(subDirName) {
    sector = directory->Find(subDirName);
    if(sector == -1)
        break;
    subDirFile = new OpenFile(sector);
    directory->FetchFrom(subDirFile);
    subDirName = strtok(NULL, "/");
}
```

  這邊我們使用了strtok函數來把path中的directory依序分割出來，而while loop中實作的內容主要是負責找到並開啟要建立檔案的directory。找到之後，建立檔案的過程就跟原先只有一個root directory的過程相同，新增檔案名稱與sector位置到directory、建立與allocate檔案、把有修改過的資訊flush回disk。

- **FileSystem::Open(char* name), FileSystem::Remove(char* name), FileSystem::List(char* name) — modified**
  這幾個對於檔案的操作一樣要先順著path找到檔案所處的sector，然後再根據找到的sector去對檔案進行操作。基本上這部分跟尚未修改的時候相同，只是多加了traverse directory的過程。

```
char* temp = new char[MaxPathLength];
strcpy(temp, name);
char* subDirName = strtok(temp, "/");

while(subDirName) {
    sector = directory->Find(subDirName);
    //cout << sector << endl;
    //cout << subDirName << endl;
    if(!directory->isDir(subDirName)) {
        //cout << "Check" << endl;
        break;
    }
    subDirFile = new OpenFile(sector);
    directory->FetchFrom(subDirFile);
    subDirName = strtok(NULL, "/");
    delete subDirFile;
}
```

- **FileSystem::mkdir(char* name) — newly added**
  新增此函數用來負責創建新的directory，因為NachOS的file system把directory當成file來管理，創建directory的過程和創建檔案的過程其實大同小異。唯一的差異在於file header要allocate的是一個directory的大小，同時也要在directory的entry中和一般的檔案做出區別，這邊我們在class DirectoryEntry中新增了一個check的boolean變數 (bool isDir)，用來判別說當前的file header是屬於檔案的還是subdirectory的。

```
sector = freeMap->FindAndSet();

ASSERT(sector>=0);
ASSERT(dir->Add(subDirName, sector, 1)); // add subd

FileHeader* Hdr = new FileHeader(); // subdir file h

ASSERT(Hdr->Allocate(freeMap, DirectoryFileSize));

Hdr->WriteBack(sector); // flush back

OpenFile* newDirFile = new OpenFile(sector);
Directory* newDir = new Directory(NumDirEntries);

newDir->WriteBack(newDirFile);
dir->WriteBack(subDirFile);
freeMap->WriteBack(freeMapFile);
```

```
33 class DirectoryEntry
34 {
35 public:
36     bool inUse;
37     bool isDir;
38     int sector;
39
40     char name[FileNameMaxLen + 1];
41
42 };
```

- **Directory::Add(char* name, int newSector, bool isDir) — modified**
  我們在此多新增了一個boolean變數的argument，用來判別說要加入到當前directory的是檔案還是subdirectory。

```
128 bool Directory::Add(char *name, int newSector, bool isDir)
129 {
130     if (FindIndex(name) != -1)
131         return FALSE;
132
133     for (int i = 0; i < tableSize; i++)
134         if (!table[i].inUse)
135         {
136             table[i].inUse = TRUE;
137             table[i].isDir = isDir;
138             strncpy(table[i].name, name, FileNameMaxLen);
139             table[i].sector = newSector;
140             return TRUE;
141         }
142     return FALSE; // no space.  Fix when we have extensible
143 }
```

- **Directory::isDir(char* name) — newly added**
  新增此函數用來判別在當前directory下特定name的file是檔案還是subdirectory。

```
197 bool Directory::isDir(char* name) {
198     int idx = FindIndex(name);
199     if(idx == -1)
200         return 0;
201     return table[idx].isDir;
202 }
```

5. Recursively List the Directory

- **Main.cc — modified**
  這邊我們多加入了recursiveListFlag的判定，如果recursiveListFlag有被flip起來的話，就呼叫class filesystem中定義的RecursiveList()函數。反之則呼叫一般的List()函數。

```
if(recursiveListFlag)
    kernel->fileSystem->RecursiveList(listDirectoryName);
else
    kernel->fileSystem->List(listDirectoryName);
```

- **FileSystem::RecursiveList(char* name) — newly added**
  新增此函數先順著path找到要recursively list的directory，然後再利用class Directory中定義的RecursiveList()函數，來list出指定directory的所有內容。

```
char* temp = new char[MaxPathLength];
strcpy(temp, name);
char* subDirName = strtok(temp, "/");

while(subDirName) {
    int sector = dir->Find(subDirName);
    ASSERT(sector>=0);
    subDirFile = new OpenFile(sector);
    dir->FetchFrom(subDirFile);
    subDirName = strtok(NULL, "/");
    delete subDirFile;
}

dir->RecursiveList(0);
```

- **Directory::RecursiveList(int layer) — newly added**
  新增此函數用來recursive list出指定directory的內容，概念基本上就是掃過一次當前directory所擁有的file和subdirectories。如果掃到的是subdirectories，就recursively去呼叫subdirectories的RecursiveList()函數；而如果掃到的是檔案，就直接print出檔案的名稱即可。

  同時這邊我們也加入了一個整數變數的argument: layer，用來specify說目前在recursion的第幾層，方便我們實作indentation。

```
204 void Directory::RecursiveList(int layer) {
205     for(int i=0; i<tableSize; ++i)
206         if(table[i].inUse) {
207             if(table[i].isDir) {
208                 for(int j=0; j<layer; ++j)
209                     printf("    ");
210                 printf("[D] %s\n", table[i].name);
211                 Directory* subdir = new Directory(NumDirEntries);
212                 OpenFile* subDirFile = new OpenFile(table[i].sector);
213                 subdir->FetchFrom(subDirFile);
214                 subdir->RecursiveList(layer+1);
215                 delete subdir;
216                 delete subDirFile;
217             } else {
218                 for(int j=0; j<layer; ++j)
219                     printf("    ");
220                 printf("[F] %s\n", table[i].name);
221             }
222         }
223 }
```