MP4:File SYSTEM

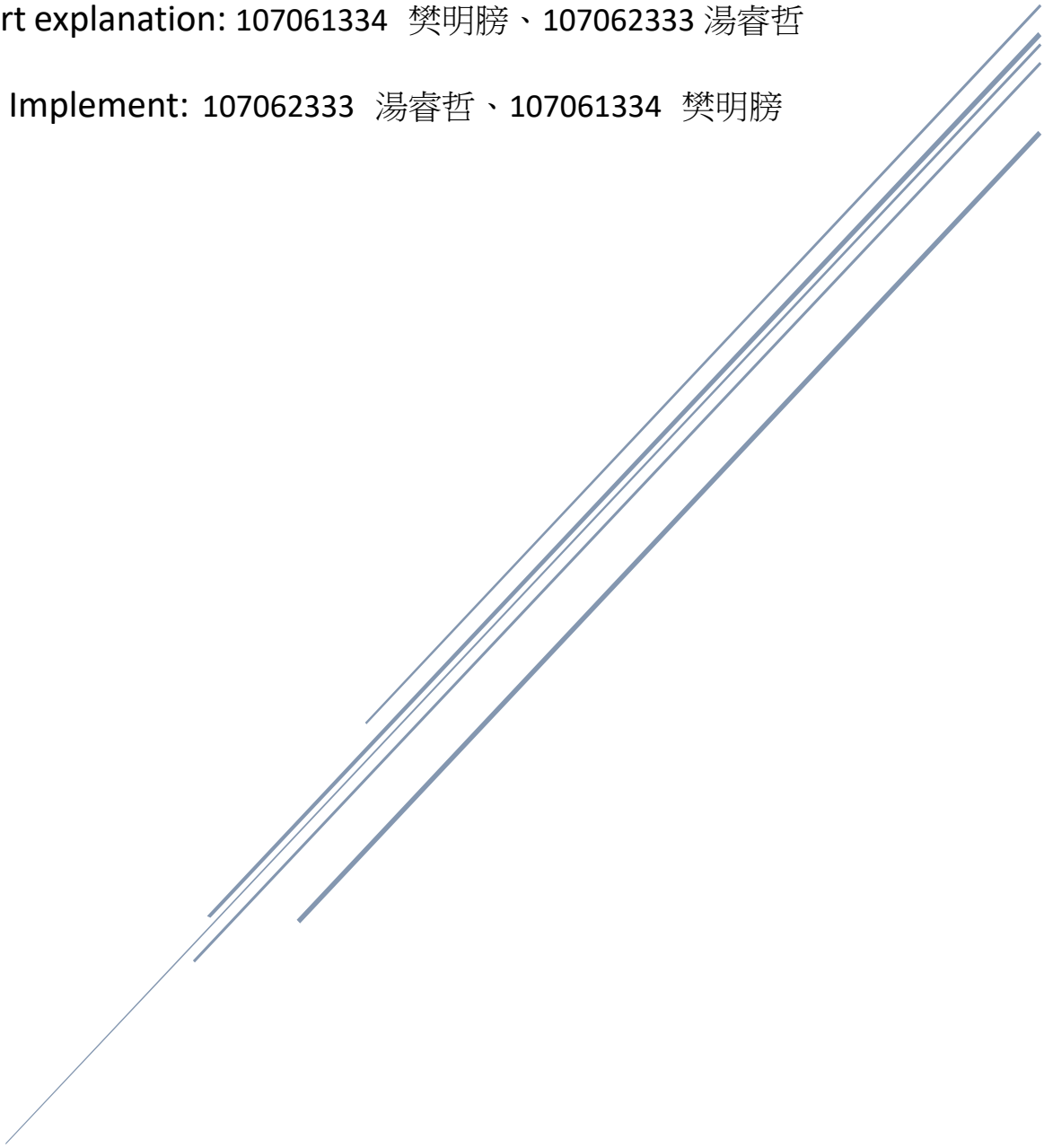
Team 21

Operating System Machine Problem 4 - File System

(a)Team List & Contibutions:

Report explanation: 107061334 樊明膀、107062333 湯睿哲

Code Implement: 107062333 湯睿哲、107061334 樊明膀



Part I. Understanding NachOS file system-1

這次沒有明確的說明說要從哪裏 trace code，我大致上從-DFILESYS_STUB 開始找，因為這次是拿掉了這個，讓 nachos 可以有 file system 的系統，這次沒有 Define DFILESYS_STUB，所以要注意的點是#ifndef FILESYS_STUB，大致上是這樣。

那先從 threads/main.cc 的 main function 開始，找#ifndef FILESYS_STUB 的地方，但後來 trace 到一半，覺得很多東西在 kernel init 的時候，很多都被創建了，所以覺得先解釋 kernel init 的地方。

那 kernel 的地方要從 main 的確認完指令的地方找

```
    kernel = new Kernel(argc, argv);


    kernel->Initialize();
```

因為有些是要設 kernel 的 parameter 所以就會傳進去

之後就做 initialize 的步驟

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

最主要是要講這個，因為 ndef FILESYS_STUB 的關係，所以 format flag 被拉成 TURE。

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

之後在 init 裡面做 filesystem 的 contruct

```
FileSystem::FileSystem(bool format)
{   DEBUG(dbgFile, "Initializing the file system.");
    if (format)
    {
        PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);
        Directory *directory = new Directory(NumDirEntries);
        FileHeader *mapHdr = new FileHeader;
        FileHeader *dirHdr = new FileHeader;
        DEBUG(dbgFile, "Formatting the file system.");
```

那因為 format 是 TURE 所以就開始創東西，像是 persistentbitmap、directory、mapHdr、dirHdr

```
const int NumSectors = (SectorsPerTrack * NumTracks);的數量 32*32
FileHeader::FileHeader()
{   numBytes = -1;
    numSectors = -1;
```

```
    memset(dataSectors, -1, sizeof(dataSectors));
}
```

那其實現在 fileheader 的 init 不是很重要，因為之後在 Allocate or FetchFrom 會再做一次

```
PersistentBitmap::PersistentBitmap(int numItems) : Bitmap(numItems)
{}
```

Persistentbitmap 裡面是空的，繼承了 Bitmap，

```
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);
    }
}
```

上面就是做一些 map 的設定

接著 filesystem 的 construct 繼續向下

```
        freeMap->Mark(FreeMapSector);
        freeMap->Mark(DirectorySector);
void Bitmap::Mark(int which)
{
    ASSERT(which >= 0 && which < numBits);
    map[which / BitsInWord] |= 1 << (which % BitsInWord);
    ASSERT(Test(which));
}
```

在 boot 的時候設定好這兩個位置

Freemapsector =0.Directorysector =1.是 free map 跟 directory 的 file header 所以先保留出來，然後 freemap 上面坐上標記代表用了

```
        ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
        ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize))
#define FreeMapFileSize (NumSectors(32*32)/ BitsInByte(=8))
```

```
#define NumDirEntries 10
#define DirectoryFileSize(sizeof(DirectoryEntry) * NumDirEntries(=10))
bool FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
{
    numBytes = fileSize;
    numSectors = divRoundUp(fileSize, SectorSize);
    if (freeMap->NumClear() < numSectors)
        return FALSE; // not enough space
    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);
    }
    return TRUE;
}
```

這邊將 freemap 上面的做 sector 的運算，看看有沒有空的位置夠不夠 (freeMap->NumClear() < numSectors)，如果有空的話(find and set)，就可以填入

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

FindAndSet 就是去試位置，看有沒有位置可以擺如果有的話就位置 mark 起來，然後 return 位置

繼續往下，因為都建好了可以取得 header 了

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

這一步是做一個 init 的動作，因為你目前不知道這兩個位置存什麼東西

```
void FileHeader::WriteBack(int sector)
{   kernel->synchDisk->WriteSector(sector, (char *)this);
```

```
}
```

Call 模擬的 disk 下去寫 sector

```
void SynchDisk::WriteSector(int sectorNumber, char *data)
{
    lock->Acquire(); // only one disk I/O at a time
    disk->WriteRequest(sectorNumber, data);
    semaphore->P(); // wait for interrupt
    lock->Release();
}
```

這邊跟 read sector 很像，先講 WriteRequest

```
void Disk::WriteRequest(int sectorNumber, char *data)
{
    int ticks = ComputeLatency(sectorNumber, TRUE);
    ASSERT(!active);
    ASSERT((sectorNumber >= 0) && (sectorNumber < NumSectors));
    DEBUG(dbgDisk, "Writing to sector " << sectorNumber);
    Lseek(fileno, SectorSize * sectorNumber + MagicSize, 0);
    WriteFile(fileno, data, SectorSize);
    if (debug->IsEnabled('d'))
        PrintSector(TRUE, sectorNumber, data);
    active = TRUE;
    UpdateLast(sectorNumber);
    kernel->stats->numDiskWrites++;
    kernel->interrupt->Schedule(this, ticks, DiskInt);
}
```

這個其實跟 ReadRequest 一樣

Read(fileno, data, SectorSize);換成 Write 這樣，那就一樣後面解釋

Header 拿到，可以打開了

```
freeMapFile = new OpenFile(FreeMapSector);
directoryFile = new OpenFile(DirectorySector);
```

```
OpenFile::OpenFile(int sector)
{   hdr = new FileHeader;
    hdr->FetchFrom(sector);
    seekPosition = 0;}
```

前面是在做 map 上面的維護，下面才是真正做 file 的設定，需要剛剛寫好的 header，然後有 sector 可以放

```
void FileHeader::FetchFrom(int sector)
```

```
{    kernel->synchDisk->ReadSector(sector, (char *)this);
}
```

那 Readsector 的部分之後會說，會跟 IO interrupt,linux system call 等等有關，那這個步驟會將要的地方讀出來，那這邊就只是打開。

之後都準備好後，把建好的 map 跟 directory 就可以放進去成一個 file

```
freeMap->WriteBack(freeMapFile); // flush changes to disk
directory->WriteBack(directoryFile);
```

這邊就大功告成了 freeMapFile、directoryFile 就是 filesystem init 留下來的東西

那值得一提的是

如果你 define FILESYS_STUB 的話，你會 call

```
        freeMapFile = new OpenFile(FreeMapSector);
        directoryFile = new OpenFile(DirectorySector);
```

因為不用 format disk，然後 linux 的 system call 直接支持 nachos 的檔案這樣子，後面也會再提到，在 readat 的地方

之後回到 main.cc 以一個指令-cp 來 trace

```
    else if (strcmp(argv[i], "-cp") == 0)
       {
           ASSERT(i + 2 < argc);
           copyUnixFileName = argv[i + 1];
           copyNachosFileName = argv[i + 2];
           i += 2;
       }
```

-cp 的指令，儲存了兩個

```
char *copyUnixFileName = NULL;    // UNIX file to be copied into Nachos
char *copyNachosFileName = NULL; // name of copied file in Nachos
```

一個是你要複製的檔案，一個是複製進去要叫什麼

```
if (copyUnixFileName != NULL && copyNachosFileName != NULL)
    {
        Copy(copyUnixFileName, copyNachosFileName);
    }
```

後面對應到 copy 的部分，這個 function 就在 main.cc 裡面，並傳入了兩個 parameter，from and to

```
static void Copy(char *from, char *to)
{
    int fd;
    OpenFile *openFile;
    int amountRead, fileLength;
    char *buffer;
if ((fd = OpenForReadWrite(from, FALSE)) < 0)
    {
        printf("Copy: couldn't open input file %s\n", from);
        return;
    }
```

先去取得 fd(File Descriptor)，用 OpenForReadWrite，這個 function 會是在 lib/sysdep.cc

```
Int OpenForReadWrite(char *name, bool crashOnError)
{
    int fd = open(name, O_RDWR, 0);
    ASSERT(!crashOnError || fd >= 0);
    return fd;
}
```

這個 open function 是系統呼叫，我想是#include <fcntl.h>或是 #include <sys/types.h>，那把 linux 上面的那個檔案(from)，在這邊用 open 裡面開 name，O_RDWR 的意思是: 以讀寫模式開啟，其他還有像
O_RDONLY: 以唯讀模式開啟
O_WRONLY: 以唯寫模式開啟
O_RDWR: 以讀寫模式開啟

後面那個 0，open 第三個引數 umode_t mode 做指定。如果沒帶 O_CREAT 旗標，該引數會被忽略，那因為 open function 的關係，會得到 fd，這邊一起講 read、write 的部分

```
ssize_t read(int fd, void *buf, size_t len);
```

read() 系統呼叫會從 fd 所參照檔案的當前位置讀取 len 個位元組到 buf，執行成功時會回傳寫進 buf 的位元組數，同時檔案位置也會前進所讀取的位元組數，執行失敗時會回傳 -1 並設定 errno。

那這邊先不列舉出執行失敗的狀況，nachos 都會用 assert 將他檔掉

```
ssize_t write(int fd, const void *buf, size_t count);
```

write() 系統呼叫會從 buf 中把 count 位元組的資料寫入 fd 所參照檔案的當前位置，執行成功時會回傳寫進入檔案的位元組數，檔案位置也會前進寫入的位元組數，執行失敗時會回傳 -1 並設定 errno。

之後回到 copy 的部分，往下

```
// Figure out length of UNIX file
    Lseek(fd, 0, 2);
    fileLength = Tell(fd);
    Lseek(fd, 0, 0);
```

Call 了 Lseek 跟 Tell 這兩個也是都在 lib/sysdep.cc

```
void  Lseek(int fd, int offset, int whence)
{
    int retVal = lseek(fd, offset, whence);
    ASSERT(retVal >= 0);
}
int Tell(int fd)
{#if defined(BSD) || defined(SOLARIS) || defined(LINUX)
    return lseek(fd,0,SEEK_CUR);
  #else
    return tell(fd);
#endif
}
```

lseek() 系統呼叫用來設定檔案的當前位置，本身不會產生任何 IO，成功時會回傳檔案的當前位置，失敗時回傳 -1 (所以 ASSERT 確定說 retval>=0)並設定 errn。lseek() 的 pos 可以是正、負或是零，實際的行為取決於 origin，可以是以下幾種的其中一個:

- SEEK_CUR: 位置設定為當前位置加上 offset
    - offset 是零時可以用來查詢目前檔案位置
- SEEK_END: 位置設定為當前的長度再加上 offset，也就是從結尾開始算的意思
    - offset 是零時可以把當前位置設為檔案結尾
- SEEK_SET: 位置設定為 offset，也就是從開頭開始算的意思
    - offset 是零時可以把當前位置設為檔案開頭

lseek() 也可以把檔案當前位置設定到檔案結尾以後。此時如果進行讀取的話，會回傳 EOF，若進行寫入的話，中間被跳過的部分會以 hole 的形式填補成 0 而變成 sparse file。

Tell // Report the current location within an open file.裡面是這樣寫

The tell() function reports the offset of the current byte relative to the beginning of the file associated with the file descriptor. 但我覺得這個解釋更好點

所以 Lseek(fd, 0, 2);設定當前位置(fd)為檔案開頭，tell 得到位置也就得到長度，Lseek(fd, 0, 0);得到目前檔案位置，這邊有寫// Figure out length of UNIX file，但我看除了 fileLength 其他都沒有存下來，不知道為什麼要做 lseek。

之後因為繼續，因為找到了所以要 create

```
if (!kernel->fileSystem->Create(to, fileLength))
    { // Create Nachos file
        printf("Copy: couldn't create output file %s\n", to);
        Close(fd);
        return;
    }
```

那如果 create 失敗就 printf 跟 close fd

```
bool FileSystem::Create(char *name, int initialSize)
{
    Directory *directory;
    PersistentBitmap *freeMap;
    FileHeader *hdr;
    int sector;
    bool success;
```

create 這邊

// "name" -- name of file to be created

// "initialSize" -- size of file to be created

回傳 bool 如果 create 成功就是 TRUE，否則就是 FALSE

```
    directory = new Directory(NumDirEntries);
```

define NumDirEntries = 10

```
Directory::Directory(int size)
{   table = new DirectoryEntry[size];
    // MP4 mod tag
    memset(table, 0, sizeof(DirectoryEntry) * size); // dummy operation
 to keep valgrind happy
    tableSize = size;
    for (int i = 0; i < tableSize; i++)
```

```
        table[i].inUse = FALSE;
}
```

Directory 裡面又先創 entry(結構在下面)，然後將這些空間的 bit 都設成 0，設定這個 directory 的 entry 的數量，裡面都還沒用->inuse 設成 false

```
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'
};
```

FileNameMaxLen = 9

會存使用的狀況跟 fileheader 在 disk 裡面的位置 sector，再下一步要去拿資料

```
    directory->FetchFrom(directoryFile);
```

上面是 create 繼續往下

```
void Directory::FetchFrom(OpenFile *file)
{
(void)file->ReadAt((char*)table, tableSize * sizeof(DirectoryEntry),0);
}
```

Fetchfrom 裡面又 call，file->open

進到 table 裡面，去讀 numbytes 的長度，以 position 為開頭

```
int OpenFile::ReadAt(char *into, int numBytes, int position)
{
    int fileLength = hdr->FileLength();
    int i, firstSector, lastSector, numSectors;
    char *buf;
```

先來看這個 hdr 是哪裏來的，這是每個 file 都有的 FileHeader *hdr，順帶一提，如果是 define FILESYS_STUB 的，readat 會 call 一個 lseek 之後就 call ReadPartial(file, into, numBytes);就直接從外部取檔案進來，跟在 nachos 模擬 system file 的不一樣

```
    if ((numBytes <= 0) || (position >= fileLength))
        return 0; // check request
    if ((position + numBytes) > fileLength)
        numBytes = fileLength - position;
```

```
    DEBUG(dbgFile, "Reading " << numBytes << " bytes at " << position <
< " from file of length " << fileLength);
    firstSector = divRoundDown(position, SectorSize);
    lastSector = divRoundDown(position + numBytes - 1, SectorSize);
    numSectors = 1 + lastSector - firstSector;
```

上面是在準備你要讀取的位置以及確認

```
    // read in all the full and partial sectors that we need
    buf = new char[numSectors * SectorSize];
    for (i = firstSector; i <= lastSector; i++)
        kernel->synchDisk->ReadSector(hdr->ByteToSector(i* SectorSize),
                                      &buf[(i - firstSector) * SectorSize]);
```

這邊用到

```
int FileHeader::ByteToSector(int offset)
{
    return (dataSectors[offset / SectorSize]);
}
int dataSectors[NumDirect]; // Disk sector numbers for each data
#define NumDirect ((SectorSize - 2 * sizeof(int)) / sizeof(int))
```

這個將要讀的 data sector 找出來

跟

```
void SynchDisk::ReadSector(int sectorNumber, char *data)
{
    lock->Acquire(); // only one disk I/O at a time
    disk->ReadRequest(sectorNumber, data);
    semaphore->P(); // wait for interrupt
    lock->Release();
}
```

因為會有 synchronize 的問題，會有上鎖的情形，接著用 disk->ReadRequest，那因為要等 IO 的關係，所以這邊做一個 condition variable waiting，之後等 interrupt 將他喚醒

```
void Disk::ReadRequest(int sectorNumber, char *data)
{
    int ticks = ComputeLatency(sectorNumber, FALSE);
    ASSERT(!active); // only one request at a time
    ASSERT((sectorNumber >= 0) && (sectorNumber < NumSectors));
    DEBUG(dbgDisk, "Reading from sector " << sectorNumber);
    Lseek(fileno, SectorSize * sectorNumber + MagicSize, 0);
    Read(fileno, data, SectorSize);
```

```
int fileno;       // UNIX file number for simulated disk
```

這個在 initial 的時候就會取得

```
sectorsize = 128
const int MagicNumber = 0x456789ab;就只是一串數字
```

上面只是在確認說 disk 的一些東西，fileno 是我們現在模擬的 disk 在 linux 上面的 fd，也是做 lseek 跟 read(上面解釋過在 sysdep 裡面)，read 的時候就會將你要的 data 搬到 buffer 裡面。

```
    if (debug->IsEnabled('d'))
        PrintSector(FALSE, sectorNumber, data);
    active = TRUE;
    UpdateLast(sectorNumber);
    kernel->stats->numDiskReads++;
    kernel->interrupt->Schedule(this, ticks, DiskInt);
}
```

上面這邊是在處理 debug 的訊息，然後計算出 disk 需要因為 seek 的時間，之後 call interrupt 回去到 semaphore->P()那邊

Readat 這邊繼續下去，將 buffer 裡面的東西搬出來到我要的地方，最後 return numByte

```
    // copy the part we want
    bcopy(&buf[position - (firstSector * SectorSize)], into, numBytes);
    delete[] buf;
    return numBytes;
}
```

後面這邊 create 這邊繼續下去

```
 if (directory->Find(name) != -1)
        success = FALSE; // file is already in directory
```

去找名子看看是不是已經創過了

```
int Directory::Find(char *name)
{   int i = FindIndex(name);
    if (i != -1)
        return table[i].sector;
    return -1;
}
```

裡面 call findindex 去用 file name 去找

```
int Directory::FindIndex(char *name)
{  for (int i = 0; i < tableSize; i++)
   if (table[i].inUse && !strncmp(table[i].name, name, FileNameMaxLen))
            return i;
    return -1; // name not in directory
}
```

就從 directory 的 file table(entry)下去找，那如果名子是有比對到的，然後如果又是 inuse 的話，就回傳這個 entry

不然都沒用過的話，就都會是傳-1 回去，那-1 回去的話 就會接到下面 create 的 code

```
else
    {
        freeMap = new PersistentBitmap(freeMapFile, NumSectors);
  sector = freeMap->FindAndSet ();find a sector to hold the file header
        if (sector == -1)
            success = FALSE; // no free block for file header
        else if (!directory->Add(name, sector))
            success = FALSE; // no space in directory
```

首先先創一個 persistentbitmap 這個 struct，但表面上是創，其實進去

```
PersistentBitmap::PersistentBitmap(OpenFile *file, int numItems) : Bitm
ap(numItems)
{   file->ReadAt((char *)map, numWords * sizeof(unsigned), 0);
}
```

他是將我原本的建好的 freemap 取出來，

確認一下 sector 有沒有 free 的 block(freeMap 維護好的)，然後看 directory 還能不能增加

```
bool Directory::Add(char *name, int newSector)
{
    if (FindIndex(name)!= -1)
        return FALSE;
    for (int i = 0; i < tableSize; i++)
        if (!table[i].inUse)
        {
            table[i].inUse = TRUE;
            strncpy(table[i].name, name, FileNameMaxLen);
            table[i].sector = newSector;
            return TRUE;
```

```
        }
    return FALSE; // no space.  Fix when we have extensible files.
}
```

一樣去找有沒有在裡面，沒有的話就可以把 directory 加進去，然後 table[i]設成設成 inuse

下面是 create 的最後了

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

最後一樣確認一下 header 能不能拿到空間，如果可以的話，那就可以往下做，從上面確認完了 freemap、directory、hdr 這三個確認可以了之後，才會真正的寫回去，否則 success 就會回傳 false。

那一直到這邊才完成了 nachos 裡面要創一個 file 的前置，之後再回到 main 那邊的 copy，往下 openFile

```
    openFile = kernel->fileSystem->Open(to);
    ASSERT(openFile != NULL);
```

```
OpenFile * FileSystem::Open(char *name)
{
    Directory *directory = new Directory(NumDirEntries);
    OpenFile *openFile = NULL;
    int sector;

    DEBUG(dbgFile, "Opening file" << name);
    directory->FetchFrom(directoryFile);
    sector = directory->Find(name);
    if (sector >= 0)
        openFile = new OpenFile(sector); // name was found in directory
```

```
    delete directory;
    return openFile; // return NULL if not found
}
```

前面一樣都是在做 read 的動作，像 directory 跟 sector，都找出來再去開 file，那 new OpenFile(sector);這邊也都解釋過了，所以就真的開起來一個 file 了。

接下來最後一步了，你用 linux 找到的檔案，跟你再 nachos 所開好的 file 都已經準備好了後面就是要做資料的轉移

```
// Copy the data in TransferSize chunks
    buffer = new char[TransferSize];
    while ((amountRead = ReadPartial(fd, buffer, sizeof(char) * Transfe
rSize)) > 0)
        openFile->Write(buffer, amountRead);
```

TransferSize 是=128

那 readpartial 是一個 linux 的 system call，他會將資料讀進 buffer 裡面，然後 openFile->Write(buffer, amountRead);裡面會再 call WriteAt(into, numBytes, seekPosition);這個跟前面解釋的 readat 很像，把裡面 readsector 的地方，換成 writesector，那 writesector 跟 readsector 很像，把 disk->ReadRequest 換成 disk->WriteRequest 這樣裡面就從 linux 的 read system call，換成 write，那這樣就可以寫進去了。

我們可以看到說，如果是 linux 的話，我們就直接打開 linux 就規劃好的檔案就可以了，但如果是我們自己要用 nachos 的話，就會先需要準備很多個步驟，因為 linux 支援幫你省略掉的步驟，除此之外我們的 sector 還要自己做劃分

Part I. Understanding NachOS file system-2

那經由分析目前這樣大概就清楚了，剛剛都在解釋 file system 的 boot 的流程，跟從 linux 拉檔案進來，這兩個動作，那下面我主要針對問題回答。

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

A:這個就是，前面在bootstrap的時候我們就可以看到說kernel init會去 call FileSystem(formatFlag)，這個時後最前面有兩個東西

PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);

Directory *directory = new Directory(NumDirEntries);

那freemap 就是在記錄說哪一個block 還沒有被用到的，Freemapsector =0.

可以看到說freemap被劃在了raw disk 0的位置。

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

Max disk的size的算式是這個樣子

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

magicsize = sizeof(int)=4B

這樣算下來，就是4+32*32*128 大概是 2^17也就是128KB

(3) Explain how the NachOS FS manage the directory data structure?

Where is this information stored on the raw disk (which sector)?

有以下的資訊"directory entry", 代表directory裡面的檔案，tablesize也就等於是directory所有檔案的大小，那findindex是給她自己來確認說有沒有創過這個檔案的如果有的化就可以找到index然後對應到table裡面就可以找到檔案

```
int tableSize;          // Number of directory entries
   DirectoryEntry *table; // Table of pairs:
                          // <file name, file header location>
   int FindIndex(char *name); // Find the index into the directory
```

```
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'
};
```

那entry裡面也會去看說，目前這個entry是不是有用到的因為，我是開到NumDirEntries這個數量(=10但後面有調)，然後從哪個sector可以找header，還有就是file的名子，可以供創建、read write的時候使用到。

this information stored on the raw disk (which sector)? 這個跟第一題一樣在kernel呼叫system創建的時候，但 Directorysector =1，所以是1的位置

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

A:這個就是在說file header的地方

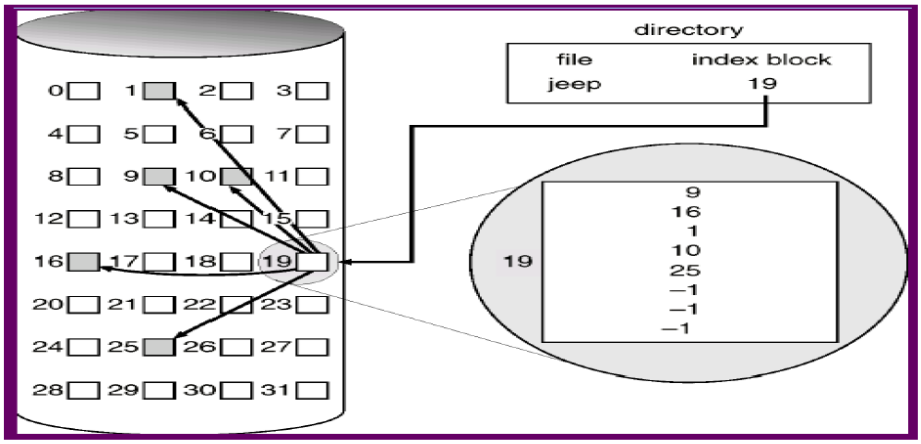
```
The following class defines the Nachos "file header" (in UNIX terms,
//the"inode"), describing where on disk to find all of the data in the
file.The file header is organized as a simple table of pointers to
data blocks.
```

```
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
```

存了 file 占了幾個 byte，占了幾個 sector，然後資料占的 sector(dataSectors)是哪幾個，因為有 dataSectors 的關係，所以是對應 block 的關係，是用 indexed allocation 的方式

Indexed Allocation Example

- The directory contains the address of the file index bloc
- Each file has its own index block
- Index block stores block # for file data



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

```
const int SectorSize = 128;   // number of bytes per disk sector
#define NumDirect ((SectorSize - 2 * sizeof(int)) / sizeof(int))這邊是
128/4
#define MaxFileSize (NumDirect * SectorSize)// *128*128/4
```

128*128B/4=2^12=4KB

```
// The file header data structure can be stored in memory or on disk.
// When it is on disk, it is stored in a single sector -- this means
// that we assume the size of this data structure to be the same
// as one disk sector.  Without indirect addressing, this
// limits the maximum file length to just under 4K bytes.
```

就因為他是 NumDirect 的數量去做直接的對應，所以大小才會被限制在 4KB，如果，做 directory 再做 subdirectory 的話，就會更多

Code Implements:

- Part_II:
  - 1. Combine your MP1 file system call interface with NachOS FS:
    這部分與 MP1 實作方式大致相同，流程主要是從 exception.cc -> ksyscall.h -> fileystem*，依此來執行不同的 instructions，而在 exception.cc 裡，為了 coding style 一致性，我會宣告一個 Sys***() 函示來進入 ksyscall，而在那呼叫 kernel->fileSystem->(instruction function)。
    下圖以 Open 為例:
    exceptions.cc:

```
case SC_Open: /// MP4 mod(2) Implement five system calls:
    val = kernel->machine->ReadRegister(4);
    {
    char *filename = &(kernel->machine->mainMemory[val]);

    fileID = SysOpen(filename);
    kernel->machine->WriteRegister(2, (int) fileID);
    }
    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;
```

    ksystem.h:

```
int SysOpen(char* filename){
    OpenFile* open_file;
    open_file = kernel->fileSystem->Open(filename);
    int index = -1;
    for(int i=0;i<20;i++){
        if(kernel->fileSystem->fileDescriptorTable[i] == NULL){
            index = i;
            break;
        }
    }
    //cout<<"ava index: "<<index<<"\n";
    if(index==-1)return -1; ///full
    else kernel->fileSystem->fileDescriptorTable[index] = open_file;
    return index+1; //since need larger than 0
}
```

    這邊要注意的是為處理有線個開檔數目，我會事先在 ksystem 裡調用到 fileSystem class 的 fileDescriptionTable[20]，來記錄開啟的檔案 ID，以及檢查還有沒有空間可以開檔案(假設最大為開 20 個不同檔案)
    fileSystem.cc: 詳細內容會在 2.解說

```
OpenFile* FileSystem::Open(char *name)
{
    Directory *directory = new Directory(NumDirEntries);
    OpenFile *openFile = NULL;

    char *file_name = get_file_name(name);
    char *dir_name = get_dir_name(name);

    DEBUG(dbgFile, "Opening file" << name);

    // Default root dir as parent
    directory->FetchFrom(directoryFile);
    // If not in root dir, change the parent directory
    if(dir_name != NULL){
        int dir_file_sector = directory->Find(dir_name, true);
        OpenFile *directory_file = new OpenFile(dir_file_sector);
        directory->FetchFrom(directory_file);

    }

    int sector = directory->Find(file_name, false);
    if (sector >= 0){
        openFile = new OpenFile(sector);    // name was found in directory
    }

    return openFile;                // return NULL if not found
}
```

- 2. Implement five system calls:
  接來將解說的是在 fileSystem 支援 Create, Open, Read, Write,Close 的實作部分:
  a. Read:

```
int Read(char *buf, int len, int id){
    int index = id-1;
    if(index<0 || index>19)return -1;
    if(fileDescriptorTable[index]==NULL)return -1;

    return fileDescriptorTable[index]->Read(buf, len);
}
```

  b. Write:

```
int Write(char *buf, int len, int id){
    int index = id-1;

    if(index<0)return -1;
    if(index>19)return -1;
    if(fileDescriptorTable[index]==NULL)return -1;

    return fileDescriptorTable[index]->Write(buf, len);
}
```

Read,Write 作法大同小異，都是先檢查 fileDescriptionTable 有沒有目標檔(是否有先 Open)以及 ID 是否有效(0~19)，都符合的話便能使用，即能接下去裡用 OpenFile.cc 支援的指令來 Read 或 Write 檔(原本就 define 好了)。

c. Close

```
int Close(int id){
    int index = id-1;
    if(index<0 || index>19)return 0;
    if(fileDescriptorTable[index]==NULL)return 0;
    fileDescriptorTable[index] = 0;
    return 1;
}
```

Close 的部分只需要將 fileDescriptionTable 對應目標檔案的 index 處清空(=NULL)，但依樣事先須檢查是否是有效的 ID。

d. 至於解釋 Open 與 Create 部分之前，先需解釋會使用到的額外 functions:

get_file_name():

```
char* get_file_name(char* name){
    char* file_name = strrchr(name, '/');
    file_name++;
    return file_name;
}
```

利用 strrchr()來切割出最後一個’/’後面的字串，即目標檔案名或目標資料夾名稱。

get_dir_name():

```
char* get_dir_name(char* name){ ///MP4
    int name_len = strlen(name)+1;
    char* tmp = new char[name_len];
    memset(tmp,0,name_len);
    memcpy(tmp, name, name_len);
    char* file_name = get_file_name(name);
    char* dir_name = strtok(name, "/");
    char *ans = new char[255];
    memset(ans,0,255);
    while(dir_name != file_name){
        memcpy(ans, dir_name, strlen(dir_name)+1);
        dir_name = strtok(NULL, "/");
    }
    memcpy(name, tmp, name_len);
    delete tmp;
    if(strlen(ans)==0)return NULL;
    return ans;
}
```

利用迴圈的方式以及 strtok，依據’/’ 的分隔來紀錄絕對路徑上，所有層遞關係父節點資料夾名稱。

Open:

```
OpenFile* FileSystem::Open(char *name){
    Directory *directory = new Directory(NumDirEntries);
    OpenFile *openFile = NULL;

    char *file_name = get_file_name(name);
    char *dir_name = get_dir_name(name);

    DEBUG(dbgFile, "Opening file" << name);

    // Default root dir as parent
    directory->FetchFrom(directoryFile);
    // If not in root dir, change the parent directory
    if(dir_name != NULL){
        int dir_file_sector = directory->Find(dir_name, true);
        OpenFile *directory_file = new OpenFile(dir_file_sector);
        directory->FetchFrom(directory_file);

    }

    int sector = directory->Find(file_name, false);
    if (sector >= 0){
        openFile = new OpenFile(sector);    // name was found in directory
    }


    return openFile;                // return NULL if not found
}
```

這裡連 Part3 部分一起說明，在利用 get file nam, get dir name 之餘，會先創建一個 Directory 型別的空指標，會先假設在 root 底下，用以之後能以 link list 方式走訪每層資料夾，直到目標檔案，而 FetchFrom()是從目前 Directory 那一層來搜尋目標資料(或資料夾)。從 if(dir_name !=NULL)來得知該目標是否在 root 層，若否，則須利用 Directory->Find()來已遞迴方式走訪並搜尋(後面的 true/false 來表示是否”可能”要進入更深層的資料夾)，當找尋完畢後即會回傳該目標的父資料夾在 disk 上 sector 的位置，隨後我另外宣告一個 OpenFile 型別來記錄，並讓 directory 用 FectchFrom()來找出真正的資料夾目錄。最後在從真正父資料夾 Find 出該目標在 disk sector 上的位置，並將檔案已 OpenFile 型別回傳，表示成功。

Create:

```
bool FileSystem::Create(char *name, int initialSize)
{
    if(!check_len(name))return false;

    Directory *directory;
    OpenFile *directory_file;
    PersistentBitmap *freeMap;
    FileHeader *hdr;
    int sector;
    bool success;

    //DEBUG(dbgFile, "[FileSystem::Create]\tCreating file " << name << " size " << initialSize);

    char* file_name = get_file_name(name);
    char* dir_name = get_dir_name(name);

    // Default put in root
    directory = new Directory(NumDirEntries);
    directory_file = directoryFile;

    // 不放在root, 切到目標dir
    if(dir_name!=NULL){
        Directory *root_dir = new Directory(NumDirEntries);
        root_dir->FetchFrom(directoryFile);
        int dir_file_sector = root_dir->Find(dir_name, true);
        if(dir_file_sector == -1)return false; ///dir 不存在

        directory_file = new OpenFile(dir_file_sector);
        //delete rootDirectory;
    }
```

Create 的部分會牽涉到 Allocation 框建，我是使用 linked-index 來操作，這部分會在 bonus1 解釋。而單純建檔流程，主要一樣會先 get 到目標檔案或資料夾的名稱，以及他的絕對路徑(不含自己)，接下來跟 Open 類似會先看 dir_name 來確定目標要創在哪裡，手法一樣先 Find 出父資料夾的位置，之後用 FetchFrom()進入父資料夾。

```
if (directory->Find(file_name, false) != -1){
    success = FALSE;            // file is already in directory
}
else {  ///create!
    freeMap = new PersistentBitmap(freeMapFile,NumSectors);
    sector = freeMap->FindAndSet(); // find a sector to hold the file header
    if (sector == -1) {
        success = FALSE;        // no free block for file header
    }
    else if (!directory->Add(file_name, sector, FILE)){
        success = FALSE;    // no space in directory
    }
    else {
        hdr = new FileHeader;
        if (!hdr->Allocate(freeMap, initialSize)){
            success = FALSE;    // no space on disk for data
        }
        else {
            success = TRUE;
            // everthing worked, flush all changes back to disk
            hdr->WriteBack(sector);
            directory->WriteBack(directory_file);
            freeMap->WriteBack(freeMapFile);
            DEBUG(dbgFile, "[FileSystem::Create]\tFile Created Success");
        }
```

另外可以用用 Find 順便檢查是否已經早被創建過。接下進入可以創建的階段，需調用到 FreeMap，來先看所有 sectors 是否滿了，以及檢查資料夾是否滿了(spec 規定最多放 64 個物件)，如果還能放就開始 Allocate()。放好後記得要 write back 回 disk 來記錄最終狀況。回傳 true(圖片中被切掉了)，完成一次 Create。

- Enhance the FS to let it support up to 32KB file size:

這裡一起連 bonus1 一起說明，目標是支援 60MB 的檔案大小，因此在 Alloction 上我採用 linked indexed Allocation，也因此我需要改動的資料結構變的很多，主要為:

a. 讓 FileHeader 能指向下一個 FileHeader 以及記錄下一個 FileHeader 在 sector 的位置，並新增一個 function 來回傳 next FileHeader，新的資料結構為下圖:

```
FileHeader* next_hdf;///MP4 mod
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
int next_hdf_sector;///MP4 mod
```

```
FileHeader* get_next_hdf(){return next_hdf;};////MP4 bonus1
```

b. OpenFile 在取得 file 總大小時，不能直接 call hdf->FileLength()，而須另行設計一個用 link_list traversal 方式計算總大小，方法如下圖:

```
int OpenFile::Length()
{
    //return hdr->FileLength();

    ///link list traversal
    int result = 0;
    FileHeader* next_hdf = hdr;
    while (next_hdf != NULL)
    {
        result += next_hdf->FileLength();
        next_hdf = next_hdf->get_next_hdf();
    }
    return result;
}
```

c. 所有有關 sector 的動皆須做更動: (in filedir)
主要概念都需變成 link_list 的方式走訪完完整資料:
WriteBack():

```
void FileHeader::WriteBack(int sector)
{
    kernel->synchDisk->WriteSector(sector, ((char *)this) + sizeof(FileHeader*));

    /*
        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));
        ...
    */
    if(next_hdf_sector!=-1){
        ASSERT(next_hdf != NULL);
        next_hdf->WriteBack(next_hdf_sector);
    }
}
```

FetchFrom():

```
void FileHeader::FetchFrom(int sector)
{
    kernel->synchDisk->ReadSector(sector, ((char *)this) + sizeof(FileHeader*));

    /*
        MP4 Hint:
        After you add some in-core informations, you will need to rebuild the header's structure
    */

    if(next_hdf_sector!=-1){
        next_hdf = new FileHeader;
        next_hdf->FetchFrom(next_hdf_sector);
    }
}
```

ByteToSector():

```
int FileHeader::ByteToSector(int offset)
{
    /*int target = offset/SectorSize;
    int index[32];
    kernel->synchDisk->ReadSector(dataSectors[target/32], (char*)index);
    return (index[target%32]);*/
    int index = offset / SectorSize;
    if (index < NumDirect)
        return (dataSectors[index]);
    else
    {
        ASSERT(next_hdf != NULL);
        return next_hdf->ByteToSector(offset - MaxFileSize);
    }
}
```

d. 有了需要的資料結構，接下來就能解說 Allocation 的部分:

```
/*=====================linked-index============================*/
if(fileSize<MaxFileSize)numBytes = fileSize;
else numBytes = MaxFileSize;
int remain_file_size = fileSize - numBytes;
numSectors = divRoundUp(numBytes,SectorSize);

if(freeMap->NumClear()<numSectors)return false;

for(int i=0;i<numSectors;i++){
    dataSectors[i] = freeMap->FindAndSet();
    ASSERT(dataSectors[i] >= 0);
}
if(remain_file_size>0){
    next_hdf_sector = freeMap->FindAndSet();
    if(next_hdf_sector==-1)return false;
    next_hdf = new FileHeader;
    return next_hdf->Allocate(freeMap,remain_file_size);
}
return true;
```

先檢查實體傳入的檔案大小是否超過該層最大能放的容量，若是則先處理能放塞滿的部分，這裡不用當心無法放下指向下一個的 pointer，因為在.h 檔時有做修改:

```
#define NumDirect ((SectorSize - 3 * sizeof(int)) / sizeof(int)) ///MP3 for bonus
#define MaxFileSize (NumDirect * SectorSize)
```

接下來判別完後，針對於一放在該層的資料，先檢查 freeMap 是否還有空位，若有，則利用 FindAndSet()找出能放的位置，並對 dataSectors 做 assign 來完成部分資料 allocate，接下來如果還有剩下的檔案還未被 allocate 的話，則在調用一次 alloction，傳入剩下為被 allocate 的檔案大小，並完成該檔案的 allocate。

另外需要注意的是，之所以會夠用，是因為我在 disk.h 裡

增加了 tracks 數目:

```
const int SectorSize = 128;   // number of bytes per disk sector
const int SectorsPerTrack  = 32;  // number of sectors per disk track
const int NumTracks = 32*512;   // number of tracks per disk ///MP4
const int NumSectors = (SectorsPerTrack * NumTracks);
```

因能滿足 60MB 的檔案的放置(只是我會跑很久，可能因為 link_list 吧)，60 MB <= 2^5 * 2^9(NumTracks) * 2^5(Sectors per Track) * 2^7(sector size) = 2^26

e. Deallocte():

一樣，由於 linked_index 的框架，我們需要 traversal 來優先 clear 最底(指的是最後被指到的 node)的 sector 方法實作如下:

```
void FileHeader::Deallocate(PersistentBitmap *freeMap)
{
    /*int index[32];
    for( int i=0; i<divRoundUp(numSectors,32); i++){
        kernel->synchDisk->ReadSector(dataSectors[i],(char*)index);
        for(int j=0; j<32; j++){
            if (index[j]!=-1)
                freeMap->Clear((int)index[j]);
        }
        ASSERT(freeMap->Test((int) dataSectors[i]));  // ought to be marked!
        freeMap->Clear((int) dataSectors[i]);
    }*/
    for (int i = 0; i < numSectors; i++) {
        ASSERT(freeMap->Test((int)dataSectors[i])); // ought to be marked!
        freeMap->Clear((int)dataSectors[i]);
    }
    if (next_hdf_sector != -1)
    {
        ASSERT(next_hdf != NULL);
        next_hdf->Deallocate(freeMap);
    }
}
```

◆ Support up to 64 files/subdirectories per directory:

很明顯只需改動 directory table 的陣列大小就行了，及更動 NumDirEntries 就行了

```
#define NumDirEntries       64  // MP4
```

■ Modify the file system code to support subdirectory:

這部分主要實作在 directory*檔案裡，概念是利用每個 directory 型別資料類會有一個 table[64]陣列，裡面紀錄該目錄(directory)下，甚麼 directory。

至於怎麼紀錄 File 呢?我的作法是直接修改 directory 的 class 資料，在裡面新增一個 type 變數，來判別是資料夾(DIR)還是檔案(FILE):

```
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'
    int type; //判別是FILE 還是 DIRECTORY
};
```

另外，從上圖，可以看見 inUse 負責確認該 index 是否正在使用，sector 則是該檔案(or 目錄)在 disk 上的 sector 位置。

創建 Directory:

```
void FileSystem::CreateDirectory(char *name){
    if(!check_len(name))return;

    char* file_name = get_file_name(name);
    char* dir_name = get_dir_name(name);
    ///找free block，並初始化
    PersistentBitmap* freeMap = new PersistentBitmap(freeMapFile,NumSectors);
    OpenFile* freeMapFile = new OpenFile(FreeMapSector);

    ///put to sector
    FileHeader* hdr = new FileHeader;
    hdr->Allocate(freeMap, DirectoryFileSize);

    Directory* directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);

    ///要創建的新資料夾
    Directory* inside_directory = new Directory(NumDirEntries);

    if(dir_name!=NULL){
        int dir_file_sector = directory->Find(dir_name, true);
        if(dir_file_sector==-1)return;///沒找到
        Directory* current_directory = new Directory(NumDirEntries);
        OpenFile* current_directory_files = new OpenFile(dir_file_sector);
        current_directory->FetchFrom(current_directory_files);

        int sector = freeMap->FindAndSet();
        hdr->WriteBack(sector);
        OpenFile* inside_directory_file = new OpenFile(sector);
```

```
        current_directory->FetchFrom(current_directory_files);

        int sector = freeMap->FindAndSet();
        hdr->WriteBack(sector);
        OpenFile* inside_directory_file = new OpenFile(sector);
        inside_directory->WriteBack(inside_directory_file);

        current_directory->Add(file_name, sector, DIR);
        current_directory->WriteBack(current_directory_files);
    }
    else{//放在root
        int sector = freeMap->FindAndSet();
        hdr->WriteBack(sector);
        OpenFile* inside_directory_file = new OpenFile(sector);
        inside_directory->WriteBack(inside_directory_file);
        directory->Add(file_name, sector, DIR);
        directory->WriteBack(directoryFile);
        //delete inside_directory_file;
    }

    //delete directory;
    //delete inside_directory;
    freeMap->WriteBack(freeMapFile);
    delete freeMapFile;
    delete freeMap;
    delete hdr;
    delete dir_name;
}
```

與創檔一樣會先 Call Allocate 來方配，之後要做的是看，是創在哪個資料夾底下，依據 dir_name 來判斷，用 Find()並 Fetch 方式來找出父資料夾位置，並自 FreeMap->FindAndSet()來找出能放的 Sector 位置，最後利用 Directory->ADD 放式來將目標資料夾放進父資料夾底下，然後 WriteBack 回 disk，及完成一次見資料夾的步驟。

而至於真正處理支援 Access 子資料夾，要解決的部分就是如何走訪，因此要修改的部分顯然為 Find(),List():

- Find(char* name, bool need_find_sub_dir):

```
int Directory::Find(char *name, bool need_find_sub_dir)
{
    int i = FindIndex(name);

    if (i != -1)return table[i].sector;
    else{
        //cout<<"part3\n";
        if(need_find_sub_dir){
            int result = -1;
            for(int j=0; j<tableSize; j++){
                if(table[j].inUse && (table[j].type==DIR)){
                    Directory *next_directory = new Directory(NumDirEntries);
                    OpenFile *next_directory_files = new OpenFile(table[j].sector);
                    next_directory->FetchFrom(next_directory_files);
                    result = next_directory->Find(name, true);
                    //delete next_directory;
                    //delete next_directory_files;
                }
                if(result != -1){
                    return result;
                }
            }
        }
    }
    return -1;
}
```

先用 FindIndex 來看是不是在當前目錄下，若是則直接回傳該檔案的 sector 位置，若否，則用迴圈暴力法，掃過所有 table index，並延伸深入往下找尋(一樣 call Find)，最後得出該檔案的 sector 位置(result)，並回傳表示找到。

- List(int depth, bool lr_flag):

```
void Directory::List(int depth, bool lr_flag)
{
    for (int i = 0; i < tableSize; i++){
        if (table[i].inUse){
            //printf("%s\n", table[i].name);
            if(table[i].type==FILE){
                for(int j=0;j<depth;j++)cout<<"    ";
                cout<<"[F] "<<i<<" "<<table[i].name<<"\n";
            }
            else if(table[i].type==DIR){
                for(int j=0;j<depth;j++)cout<<"    ";
                cout<<"[D] "<<i<<" "<<table[i].name<<"\n";
                if(lr_flag){
                    Directory* next_directory = new Directory(NumDirEntries);
                    OpenFile* next_directory_files = new OpenFile(table[i].sector);
                    next_directory->FetchFrom(next_directory_files);
                    int next_depth = depth+1;
                    next_directory->List(next_depth, true);
                    //delete next_directory;
                    //delete next_directory_files;
                }
            }
        }
    }
}
```

走訪方式很像 DFS 應用在多元樹，需要額外判斷的是走到的 node 可能是 FILE 或是 DIR，如果是後者則代表需要往下走訪，因此再

傳入的 depth 需+1，若為前者，則可以 print 出來他的名字與型別，並往下一個同層 node 走訪，從上述 DFS 方式走訪，即能完成整顆 directory Tree 的 traversal，值得注意的是 lr_flag 是指指令是要將該 dirctory 下的資料全部走訪還是只需走訪指定的目錄(使用者想要的目錄)底下同層的資料。

也因此在 Call directory 的 list 之前，一定要確認已經在指定的目錄下了。解決方法就是再 fileSystem:: List()解決，與 Open、Create 一樣，利用 Find 先找出該目標 sector 的位置，並利用 FetchFrom 來得到他的父 directory，完成以上步驟便能安心 Call Directory 的 List()了，實作如下圖:

```
void FileSystem::List(char *name, bool lr_flag)
{
    Directory *directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);
    int path_len = strlen(name);
    if(path_len<=1){
        int depth = 0;
        directory->List(depth,lr_flag);
    }
    else if(path_len>1){///not  root
        char* target_dir = get_file_name(name); ///last is dir
        int dir_file_sector = directory->Find(target_dir,true);
        OpenFile* next_directory_files = new OpenFile(dir_file_sector);
        Directory* next_directory = new Directory(NumDirEntries);
        next_directory->FetchFrom(next_directory_files);
        int depth = 0;
        next_directory->List(0,lr_flag);

    }

    delete directory;
}
```

- Bonus3 Recursive Remove:

  傳入指令為-rr，因此可藉該指令來知道是否需要 recursive 刪除，與 list()很像，可利用 DFS 走訪，並刪除，但需要注意的是，要 writeBack 回 disk，也因此需要更動的檔案為: fileSys.cc, directory*

  - fileSys.cc Remove():

```
else{
    //cout<<"just can delete a file\n";

    if(dir_name!=NULL){////要刪除的不在root那一層，在更深的
        //cout<<"want to delete not in root\n";
        delete_file = new OpenFile(directory->Find(dir_name,true));
        directory->FetchFrom(delete_file);
    }

    sector = directory->Find(file_name,false);
    if (sector == -1){
        delete directory;
        return FALSE; // file not found
    }
    fileHdr = new FileHeader;
    fileHdr->FetchFrom(sector);

    freeMap = new PersistentBitmap(freeMapFile, NumSectors);

    fileHdr->Deallocate(freeMap); // remove data blocks
    freeMap->Clear(sector);       // remove header block
    bool valid = directory->Remove(file_name,true);
    if(valid){ ///要求只能檔案
        freeMap->WriteBack(freeMapFile);     // flush to disk
        directory->WriteBack(delete_file); // flush to disk
    }
}
```

這部分為支援-r 的指令，只能刪除目標檔案，因此會須裡用到 Find()並 FetchFrom()得知該檔案的父目錄的位置，找到後，對該父目錄 Call Remove(裡面傳入的 true/false 代表是檔案還是目錄)，傳入目標檔案名，並刪除該檔案，並在完工後記得 writeBack 回 disk，directory 裡的 Remove 操作如下圖:

```
bool Directory::Remove(char *name, bool is_remove_file)
{
    int i = FindIndex(name);
    if(is_remove_file && table[i].type==DIR)return FALSE;
    //cout<<name<<"index: "<<i<<"\n";
    if (i == -1)
        return FALSE; // name not in directory
    table[i].inUse = FALSE;
    return TRUE;
}
```

```
if(rr_flag){//cout<<"recursive remove!\n";
    freeMap = new PersistentBitmap(freeMapFile,NumSectors);
    sector = directory->Find(file_name,true);
    if (sector == -1){
        delete directory;
        //cout<<file_name<<"not found\n";
        return FALSE; // file not found
    }
    Directory* tmp_dir = new Directory(NumDirEntries);
    OpenFile* tmp_dir_files = directoryFile;
    tmp_dir->FetchFrom(directoryFile);

    if(dir_name!=NULL){////要刪除的不在root那一層，在更深的
        //cout<<file_name<<" not in root\n";
        delete_file = new OpenFile(directory->Find(dir_name,true));
        directory->FetchFrom(tmp_dir_files);
    }

    delete_file = new OpenFile(sector);
    directory->FetchFrom(delete_file);
    //directory->List(0,true);
    directory->RemoveAll(freeMap,delete_file);


    fileHdr = new FileHeader;
    fileHdr->FetchFrom(sector);

    fileHdr->Deallocate(freeMap); // remove data blocks
    freeMap->Clear(sector);       // remove header block
```

```
tmp_dir->Remove(file_name,false);

freeMap->WriteBack(freeMapFile);     // flush to disk
tmp_dir->WriteBack(tmp_dir_files); // flush to disk
//tmp_dir->List(0,true);

delete tmp_dir;


////記得刪除目標dir
directory = new Directory(NumDirEntries);
directory->FetchFrom(directoryFile);
delete_file = directoryFile;
if(dir_name!=NULL){////要刪除的不在root那一層，在更深的
    //cout<<"next want to delete not in root\n";
    delete_file = new OpenFile(directory->Find(dir_name,true));
    directory->FetchFrom(delete_file);
}
int fi = tmp_dir->Find(file_name,true);
//cout<<fi<<"\n";
sector = fi;

freeMap->Clear(sector);       // remove header block
directory->Remove(file_name,false);

freeMap->WriteBack(freeMapFile);     // flush to disk
directory->WriteBack(delete_file); // flush to disk
```

接下來這部分(上圖)對於-rr 指令的處理，除了要刪除該目錄底下

的所有內容，該資料夾也一併須被刪除，因此得要先刪除子目錄才能最後刪除父目錄，者也不難聯想到 DFS 走訪方式。但在叫 directory 刪除之前，我們需要再 fileSys 處跳到目標目錄，一樣利用 Find()並 FetchFrom()來達成，但須事先紀錄該目錄的父目錄，才能在最後刪除該目錄，另外為了方便起見，可以額外用一個 Directory:: RemoveAll()來刪除目標目錄底下所有資料，實際操作如下圖:

```
bool Directory::RemoveAll(PersistentBitmap* freeMap, OpenFile *delete_file){
    for(int i=0;i<tableSize;i++){
        if(table[i].inUse && table[i].name){
            //cout<<table[i].name<<"--";
            if(table[i].type==DIR){
                ////delete inside dir first
                Directory* next_directory = new Directory(NumDirEntries);
                OpenFile* next_directory_files = new OpenFile(table[i].sector);
                next_directory->FetchFrom(next_directory_files);
                next_directory->RemoveAll(freeMap, next_directory_files);
                delete next_directory;
                delete next_directory_files;
            }

            FileHeader *hdr = new FileHeader;
            hdr->FetchFrom(table[i].sector);
            table[i].inUse = false;
            hdr->Deallocate(freeMap);
            freeMap->Clear(table[i].sector);
            delete hdr;
        }
    }
    this->WriteBack(delete_file);
}
```

跟 -lr 實的 list 很像，用迴圈掃過，如果掃到的物件是目錄，則對他 Call RemoveAll 來往下刪除，若是檔案，則可以直接刪除，但記得要 Deallocate 該檔案在 FreeMap 上以及 Clear 掉他在 disk 上的 sector。完成 Return 回 FileSys 後記得要刪除當下目錄，做法與刪除一個檔案一樣。

- Bonus2: multi_size head file:

  越大的檔案，他所用的 index blocks 一定會越多，以此，fileheder 的大小也會越來越大，從 OpenFile:: Length()操作方法就能得知:

  證明方式，可以在.sh 檔的利用’ D’ 指令及用來顯示出目前每個 Files 他們指向的所有 block，並 show 他們的位置，以下圖為例；

```
Directory contents:
Name: FS_test1, Sector: 543
FileHeader contents.  File header size: 948.  File blocks:
544 545 546 547 548 549 550 551
file  size: 948

Name: file1, Sector: 552
FileHeader contents.  File header size: 27.  File blocks:
553
file  size: 27

Name: f1, Sector: 554
FileHeader contents.  File header size: 1000.  File blocks:
555 556 557 558 559 560 561 562
file  size: 1000

Name: f2, Sector: 563
FileHeader contents.  File header size: 3712.  File blocks:
564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592
file  size: 10000


[F] 0 FS_test1
[F] 1 file1
[F] 2 f1
[F] 3 f2
```

- 最後簡單說明一下 main.cc
  對於 lr 跟 rr 都會傳入一個 true flag 來知道需要做 recursive 的動作，而 mkrdir，則會直接對 filesys call CreateDirectory 來創建資料夾，其他動作都如 MP1 實作一樣，需要注意的只是有些 function 需在 define，因為這次作業所有指令不再都是 system interrupt 囉。