

# MP4 report

## Part 1.

### Q1.

NachOS 將 free map 視為一個 file 來管理，裡面記錄著有哪些 sector 是可以使用的，在 file system 中會隨時開著這個 file 對他進行讀寫來管理 free map，在 filesys.h 裡可以看到 private 裡有一個 open file object：

```
OpenFile* freeMapFile; // Bit map of free disk blocks,  
// represented as a file
```

通常執行時會先 format file system，此時在 FileSystem 的建構式裡面會先建立一個 PersistentBitmap 的 object：

```
PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);
```

這個 PersistentBitmap 的資訊可以在 pbitmap.h 裡做一個全方面的深度了解，這裡可以觀察出他繼承 Bitmap，且比起 Bitmap 多了對 Disk 的讀取及寫回之操作：

```
class PersistentBitmap : public Bitmap {  
public:  
    PersistentBitmap(OpenFile *file, int numItems); // initialize bitmap from disk  
    PersistentBitmap(int numItems); // or don't...  
  
    ~PersistentBitmap(); // deallocate bitmap  
  
    void FetchFrom(OpenFile *file); // read bitmap from the disk  
    void WriteBack(OpenFile *file); // write bitmap contents to disk
```

而我們接下來抵達位於 lib 資料夾裡的 Bitmap.h 現場，對此做更深入的了解：

```

class Bitmap {
public:
    Bitmap(int numItems); // Initialize a bitmap, with "numItems" bits
    // initially, all bits are cleared.
    ~Bitmap();           // De-allocate bitmap

    void Mark(int which);      // Set the "nth" bit
    void Clear(int which);    // Clear the "nth" bit
    bool Test(int which) const; // Is the "nth" bit set?
    int FindAndSet();          // Return the # of a clear bit, and as a side
    // effect, set the bit.
    // If no bits are clear, return -1.
    int NumClear() const;     // Return the number of clear bits

    void Print() const;        // Print contents of bitmap
    void SelfTest();           // Test whether bitmap is working

protected:
    int numBits;   // number of bits in the bitmap
    int numWords;  // number of words of bitmap storage
    // (rounded up if numBits is not a
    // multiple of the number of bits in
    // a word)
    unsigned int *map; // bit storage
}

```

而剛剛提到 FileSystem 裡建立的 PersistentBitmap 型別的 object 便能藉由標示其本身的 integer array 為 0 or 1 來標示該 index 作為 sector number 對應到的 sector 是否被使用，0為 free sector，1為 sector 已被使用。之後我們將這個freeMap object寫到一個 file上。

### Free Map File的建立：

每個file都有file header記錄著這個file的資訊，以下是free map的file header建立過程：

```

#define FreeMapSector 0

freeMap->Mark(FreeMapSector);

FileHeader *mapHdr = new FileHeader;

mapHdr->WriteBack(FreeMapSector);

freeMapFile = new OpenFile(FreeMapSector);

```

這邊可以看到free map的file header會存在第0個sector(freeMap第0格sector被Mark為被使用)，之後會透過header打開free map file，然後將得到的OpenFile object存在file system裡，因此NachOS如果要尋找free map就可以直接對freeMapFile進行讀寫。

每個file的data block是由header提供的函式進行allocate，free map file也是：

```
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;
}
```

首先會去看free map裡有沒有足夠數量的free sector可以容納這個file需要的數量(透過NumClear())，之後透過一連串的FindAndSet()找到free sector將之標示為used並且記錄在header中的dataSectors[](用來記錄Data block的位置)。

### File Allocate：

一般file的Allocate方式，我們從Create file時來看，首先拿到free map，並且在free map上找到一個free sector存放這個file的header。

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

之後一樣由這個file的header來Allocate這個file的数据 block

```
    hdr = new FileHeader;
    if (!hdr->Allocate(freeMap, initialSize))
```

最後再將改變的free map寫回free map file，以及這個檔案的header寫到他所在的sector。

```
hdr->WriteBack(sector);
directory->WriteBack(directoryFile);
freeMap->WriteBack(freeMapFile);
```

## Q2.

在disk.h中定義了以下：

```
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
```

可以看到每個sector是128 bytes，每個track中有32個sector，總共有32個tracks。因此**disk的大小為 $128 * 32 * 32$  bytes = 128 KB**

## Q3.

首先看到NachOS的directory實作方式，他其實是一個table其中每一格都是一個DirectoryEntry:

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

每個entry記錄著一個file的file name和他的header所在的sector，以及是否正在使用中：

```
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'
};
```

如同free map，NachOS也將directory視為file來管理，file system內會一直開啟著directory這個file，用以下這個object操作directory

```
OpenFile* directoryFile;      // "Root" directory -- list of
                            // file names, represented as a file
```

Directory的建立：

如同free map，directory是在file system要format時所建立的：

```
#define NumDirEntries 10
```

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

這邊可以看到directory大小是寫死的，只有十個entry。

directory file header的建立：

```
#define DirectorySector 1
FileHeader *dirHdr = new FileHeader;
freeMap->Mark(DirectorySector);
```

之後Allocate directory所需要的data block，其所在位置會存在header中

```
ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));
```

將header寫回disk上的第一個sector

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

透過header將directory file打開，此時這個file仍然是空的，因此需要把最前面建立的directory寫到這個檔案中：

```
directoryFile = new OpenFile(DirectorySector);
directory->WriteBack(directoryFile);
```

之後我們便能透過directoryFile這個OpenFile object來對directory進行讀寫，而header會儲存在第1個sector上。

對 Directory 的操作：

以下是 directory.h 中提供的函式：

```
class Directory {
public:
    Directory(int size); // Initialize an empty directory
    // with space for "size" files
    ~Directory(); // De-allocate the directory

    void FetchFrom(OpenFile *file); // Init directory contents from disk
    void WriteBack(OpenFile *file); // Write modifications to
    // directory contents back to disk

    int Find(char *name); // Find the sector number of the
    // FileHeader for file: "name"

    bool Add(char *name, int newSector); // Add a file name into the directory

    bool Remove(char *name); // Remove a file from the directory

    void List(); // Print the names of all the files
    // in the directory
    void Print(); // Verbose print of the contents
    // of the directory -- all the file
    // names and their contents.
```

這些函式會在 `filesys.cc` 中各函式使用到，對 `file` 進行操作：

**Create()** 時：會利用 `FetchFrom()` 從 `directoryFile`(一個 `OpenFile`) 中拿出 `directory` 的資料，再利用 `Find()` 檢查 `file` 是否已在 `Directory` 內了，沒有就再檢查有沒有 free sector 能存放 `file header`，再檢查 `directory` 還有沒有空間並加入檔案資料，最後再檢查 `disk` 上有沒有足夠空間存放所有 `data` 並 `Allocate Data sector`，如果以上檢查都一一通過，那就將更新全部寫回：

```
success = TRUE;
// everything worked, flush all changes back to disk
    hdr->WriteBack(sector);
    directory->WriteBack(directoryFile);
    freeMap->WriteBack(freeMapFile);
```

**Open()** 時：藉由 `FetchFrom()` 從 `directoryFile` 拿到 `directory` 後，利用 `Find(name)`，給予檔名後拿到 `file header` 所在的 `sector`，就可以利用 `OpenFile(sector)` 建立負責該 `file` 的 `OpenFile object`。

**Remove()** 時：藉由 `FetchFrom()` 從 `directoryFile` 拿到 `directory` 後，利用 `Find(name)`，給予檔名後拿到 `file header` 所在的 `sector`，讀出 `file header` 之後，利用 `header` 的函式進行 `Deallocate data sector`，最後將 `header` 所在的 `sector` 清空，對 `directory` 進行修改，再將更新 `WriteBack` 到 `disk` 上。

**List()** 時：使用 `FetchFrom()` 拿到 `directory` 後使用 `List()`，將 `directory` 資料列出。

## Q4.

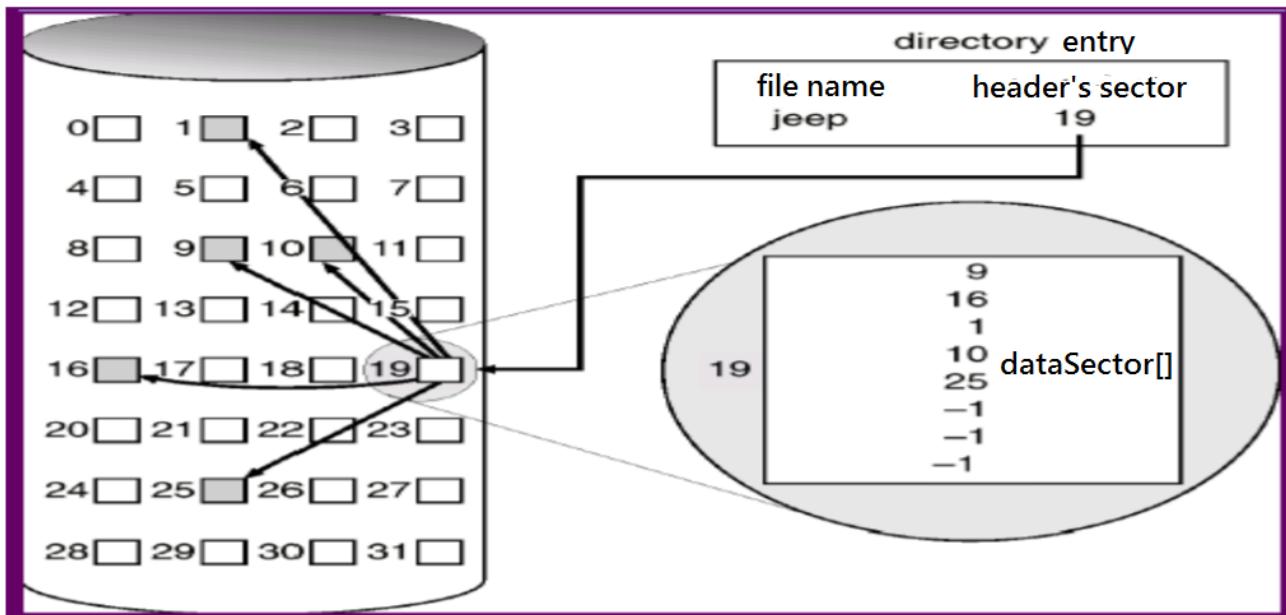
每個 `inode(file header)` 中擁有的資料可以在 `filedr.h` 中看到，註解則說明了他們的用途：

```

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

```

File system是採用類似index allocate scheme，但其 inode 本身就是 index block，概念上如下：



## Q5.

首先在disk.h中我們可以看到一個sector是128 bytes：

```
const int SectorSize = 128; // number of bytes per disk sector
```

NachOS在filehdr.h中定義max file size：

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

由於每個header是一個sector大小(128 bytes)，header中有numBytes和numSectors兩個資料，這兩個都是integer，所以SectorSize - 2 \* 4 bytes = 120 bytes，這剩下的120 bytes可以被dataSectors[]使用，dataSectors每一格都是一個integer，因此每個header最多可以有  $120/4 = 30$  個data sector，所能記錄的data為 $30 * 128 \text{ bytes} = 3840 \text{ bytes}$ ，差不多是4KB。而因為目前一個file只擁有一個file header，因此一個file上限為4KB。

# Part 2.

將 MP1 的 file system call interface 移到這次的project，由於ksyscall.h和start.S已經更改完成，因此只做以下的更改：

## 1. syscall.h

```
/* Create a Nachos file, with name "name" */
/* Note: Create does not open the file.  */
/* Return 1 on success, negative error code on failure */
int Create(char *name, int size);
```

這邊的create多傳了一個參數 size 進去。

## 2. exception.cc

多加了5個對應的case如下：

### a. open

```
/* MP1 */
case SC_Open:
    val = kernel->machine->ReadRegister(4);
    {
        char *filename = &(kernel->machine->mainMemory[val]);
        status = (int) 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;
```

將從 \$4 讀指向檔名的指標，接著去 nachOS 的 main memory 取出該檔名在底層 linux 中的位址，呼叫 SysOpen() 並傳入此位址，當 return 回來後再將 OpenFileID 寫回 \$2，最後 maintain PC 的資料。

## b. write

```
case SC_Write:  
{  
    int buffer = kernel->machine->ReadRegister(4);  
    int size = kernel->machine->ReadRegister(5);  
    int id = kernel->machine->ReadRegister(6);  
    char* cbuffer = &(kernel->machine->mainMemory[buffer]);  
    status = (int) SysWrite(cbuffer, size, 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;
```

write() 多了兩個參數，一樣取得該檔名在底層 linux 中的位址，並從 \$5, \$6 取得字串長度及 OpenFileID，呼叫 SysWrite() 並傳入這三個參數，而 SysWrite() return 回來會是寫入該檔的 bytes 數，將此值寫入 \$2，最後 maintain PC 值。

## c. read

```
case SC_Read:  
{  
    int buffer = kernel->machine->ReadRegister(4);  
    int size = kernel->machine->ReadRegister(5);  
    int id = kernel->machine->ReadRegister(6);  
    char* cbuffer = &(kernel->machine->mainMemory[buffer]);  
    status = (int) SysRead(cbuffer, size, 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;
```

read() 和 write() 差不多，不同的是從 \$4 得到的指標是要將 read 到的字串放入的位址，呼叫 SysRead() 並傳述這三筆參數，return 回來後同 Write()。

#### d. close

```
case SC_Close:
    val = kernel->machine->ReadRegister(4);
    status = SysClose(val);
    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;
```

Close() 稍微不同，從 \$4 讀出要關閉檔案的 OpenFileID，並呼叫 SysClose() 傳入 id，return 回來後將是否關閉成功的狀態寫入 \$2，最後 maintain PC 的資料。

#### e. create

這裡和之前 MP1 不一樣的地方就是多從 \$5 讀取多傳進的參數 size，其他的地方都一樣。

```
/* MP4 Using True file system */
case SC_Create:
    val = kernel->machine->ReadRegister(4);
    {
        int size = kernel->machine->ReadRegister(5); /* File size */
        char *filename = &(kernel->machine->mainMemory[val]); /* File name */
        //cout << filename << endl;
        status = SysCreate(filename, size);
        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;
```

### 3. ksyscall.h

ksyscall.h中，定義由system call產生interrupt的function

```
/* MP1 */
int SysClose(int id)
{
    return kernel->interrupt->Close(id);
}

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

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

int SysOpen(char *filename)
{
    return kernel->interrupt->Open(filename);
}

/* MP4 */
int SysCreate(char *filename, int size)
{
    // return value
    // 1: success
    // 0: failed
    return kernel->interrupt->CreateFile(filename, size);
}
```

#### 4. Interrupt.h:

```
/* MP1 */
int Open(char *filename);
int Write(char* buffer , int size , int id);
int Read(char* buffer , int size , int id);
int Close(int id);
/* MP4 */
int CreateFile(char *filename, int size);
```

#### 5. interrupt.cc

定義由interrupt呼叫kernel的Function。

```
/* MP1 */
int Interrupt::Open(char *filename)
{
    return kernel->Open(filename);
}

int Interrupt::Write(char* buffer , int size , int id)
{
    return kernel->Write(buffer, size, id);
}

int Interrupt::Read(char* buffer , int size , int id)
{
    return kernel->Read(buffer, size, id);
}

int Interrupt::Close(int id)
{
    return kernel->Close(id);
}

/* MP4 */
int Interrupt::CreateFile(char *filename, int size)
{
    return kernel->CreateFile(filename, int size);
}
```

## 6. kernel.h

```
/* MP1 */
int Open(char *filename);
int Write(char* buffer , int size , int id);
int Read(char* buffer , int size , int id);
int Close(int id);

/* MP4 */
int CreateFile(char* filename , int size);
```

## 7. kernel.cc

```
/* MP1 */
int Kernel::Open(char *filename)
{
    OpenFile* file = fileSystem->Open(filename);
    if(file == NULL) return -1;
    return (int)(file);
}

int Kernel::Write(char* buffer , int size , int id)
{
    OpenFile* file = (OpenFile*) id;
    for(int i=0 ; i < fileSystem->openFileTableTop ; i++)
        if(fileSystem->openFileTable[i] == file)
            return file->Write(buffer, size);
    return -1;
}
```

Open()：這邊直接使用 fileSystem 的 Open()，他會回傳一個 OpenFile 物件的指標，我們將該指標儲存位址當作 OpenFileID 回傳，若 fileSystem Open 失敗則回傳 -1。

Write()：首先將傳入的id轉型成OpenFile的指標，並去openFileTable中看是否有記錄這個指標，若沒有則代表該檔案沒有成功開啟，回傳-1，若有找到則呼叫該OpenFile物件裡的Write()並傳入字串及長度，openfile.h中定義的這個函式會一路call到底層linux的write()並回傳寫入了多少bytes，然後一路回傳到exception.cc。

```
int Kernel::Read(char* buffer , int size , int id)
{
    OpenFile* file = (OpenFile*) id;
    for(int i=0 ; i < fileSystem->openFileTableTop ; i++)
        if(fileSystem->openFileTable[i] == file)
            return file->Read(buffer, size);
    return -1;
}

int Kernel::Close(int id)
{
    OpenFile* file = (OpenFile*) id;
    for(int i=0 ; i < fileSystem->openFileTableTop ; i++)
    {
        if(fileSystem->openFileTable[i] == file)
        {
            fileSystem->openFileTable[i] = fileSystem->openFileTable[fileSystem->openFileTableTop-1];
            fileSystem->openFileTableTop--;
            delete file;
            return 1;
        }
    }
    return 0;
}
```

Read()：跟Write()一樣先將id的類型轉成OpenFile指標，然後去看看是否有成功開啟，沒有的話回傳-1，有的話呼叫該物件的Read()並傳入要read到的字串及大小，這函式一樣在openfile.h中定義，會一路call到底層linux的read()並回傳寫入了多少bytes，並一路回傳到exception.cc。

Close()：首先將傳入的id轉型成OpenFile指標，一樣去table檢查是否有成功開啟了，沒有的話就回傳0代表關閉失敗，如果有的話就將該OpenFile物件delete掉，並將這個指標從table中刪除，代表已經不是open的狀態，並回傳1回到exception.cc，值得注意的是delete OpenFile物件時，會啟動openfile.h中的解構式，這邊會一路call到底層linux的close()。

```
/* MP4 */
int Kernel::CreateFile(char *filename, int size)
{
    return fileSystem->Create(filename, size);
}
```

Create()：使用 kernel 的 fileSystem 中的 Create()，並傳入檔名跟檔案大小。

## 8. filesys.h

```
/* MP4 */
OpenFile *openFileTable[487];
int openFileTableTop;
```

我們新增原本 MP1 有的 openfileTable，是用來記錄所有User開啟的 OpenFile，代表該檔案有成功開啟，方便之後的Read(), Write(), Close()進行操作。

## 9. filesys.cc

```
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);

    /* MP4 */
    /* Too much open file? */
    if (openFileTableTop >= 487) return NULL;
    /* We find the file header? */
    if (sector >= 0)
    {
        openFile = new OpenFile(sector); // name was found in directory
        openFileTable[openFileTableTop++] = opened;
    }

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

再取得 file header 所在 sector 後，我們多加了用 openFileTable 來檢查目前打開的 file 是否太多，以免openFileTable塞不下。之後如果檔案開啟成功，就將得到的OpenFile object記錄在openFileTable中，並將 openFileTableTop++ 代表成功新開了一個file。

=====

以下是讓 File System support 32KB file 所做的更改，我們將使用**linked indexed scheme**，將 file header 串在一起，每個 header 可以管理少量的 Data Sector 。

## 10. filehdr.h

```
/* MP4 */
FileHeader* nextFileHeader; /* in core */
int nextFileHeaderSector; /* on disk */
```

在此 filehdr.h 新增這兩個變數， nextFileHeader 這個指標是用來指向下一個 header，nextFileHeaderSector 則紀錄下一個 header 在哪一個 sector。值得注意的是，nextFileHeader 是 in-core data，只會在memory中使用，並不會

跟著 write back 回 disk。因為每次使用 file header 時，是先建立一個新的 file header 然後從 disk 上抓取資料讀到這個 header 中，使用完以後，會把 header 刪掉，因此 nextFileHeader 所指向的地方已經被刪掉了，是沒有必要儲存 nextFileHeader。

我們還需要將 NumDirect 的計算稍做修正：

```
#define NumDirect ((SectorSize - 3 * sizeof(int)) / sizeof(int))
/* MP4 -3 is for numBytes, numSectors, nextFileHeaderSector to store in one sector */
```

SectorSize - 3\*sizeof(int) 代表一個 header 佔的一個 sector 要扣掉三個 int 的 size 後，剩下的才是放 dataSectors[NumDirect] 的實際空間大小。

## 11. filehdr.cc

建構式的初始：

```
FileHeader::FileHeader()
{
    /* MP4 */
    nextFileHeaderSector = -1;
    nextFileHeader = NULL;

    numBytes = -1;
    numSectors = -1;
    memset(dataSectors, -1, sizeof(dataSectors));
}
```

## 解構式：

```
FileHeader::~FileHeader()
{
    /* MP4 */
    if(nextFileHeader != NULL)
        delete nextFileHeader; /* invoke destructor of nextFileHeader recursively */
}
```

當一個 FileHeader 被 delete 便會觸發解構式，而其中如果 linked list 下能找到下一個 FileHeader 便 delete 它進而又出發它的解構式，並遞迴的 delete 掉整個 file 的一整串 FileHeader linked list。

## Allocate()：

```
FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
{
    /* MP4 */
    /* How much data bytes this header has? */
    int remainFileSize = fileSize - MaxFileSize;
    if(remainFileSize <= 0)/* We don't need next header */
        numBytes = fileSize;
    else /* We need next header */
        numBytes = MaxFileSize;

    numSectors = divRoundUp(numBytes, 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);
    }

    /* MP4 */
    /* Need next header */
    if(remainFileSize > 0)
    {
        /* Make a new header */
        nextFileHeaderSector = freeMap->FindAndSet();
        if(nextFileHeaderSector == -1) return FALSE;

        nextFileHeader = new FileHeader;
        return nextFileHeader->Allocate(freeMap, remainFileSize);
    }
}
```

這裡我們有傳入一個參數 fileSize，他是從 System call Create() 迭代傳進來的參數代表著現在這個 file 所需的空間大小。所以首先我們先判斷 fileSize 是不是超過了這個 header 所能擁有的資料量，如果超過了就代表我們需要下一個 header。

需要下一個就代表這個 FileHeader 的 Data sector 會放滿，因此 numBytes = MaxFileSize，不需要下一個的話就是 numBytes = fileSize。

然後去 freeMap 檢查有沒有足夠的 free sectors，不夠直接 return false，夠的話就用 freeMap->FindAndSet() 依次找 free sector 並記錄 sector number 到 dataSectors[ ] 裡。

最後如果需要下一個 FileHeader 就建立一個新的 FileHeader，用 FindAndSet() 找到一個 free sector 來存新的 FileHeader，記錄這個 sector number 到 nextFileHeaderSector 裡，之後用這個新的 FileHeader 來遞迴呼叫 Allocate() 來 Allocate 剩餘的 File 。

## Deallocate() :

```
void
FileHeader::Deallocate(PersistentBitmap *freeMap)
{
    /* MP4 */
    /* Deallocate recursively */
    if(nextFileHeader != NULL)  nextFileHeader->Deallocate(freeMap);

    for (int i = 0; i < numSectors; i++) {
        ASSERT(freeMap->Test((int) dataSectors[i])); // ought to be marked!
        freeMap->Clear((int) dataSectors[i]);
    }
}
```

這邊主要就是遞迴的釋放整個 linked list 上的 FileHeader 所持有的 dataSectors[ ]，讓那些 sectors 回歸成 free sector。

## FetchFrom() :

```
void
FileHeader::FetchFrom(int sector)
{
    /* MP4 */
    char buf[SectorSize];
    kernel->synchDisk->ReadSector(sector, buf);

    /* disk head */
    int offset = sizeof(numBytes);
    memcpy(&numBytes, buf, sizeof(numBytes));
    memcpy(&numSectors, buf + offset, sizeof(numSectors)); offset += sizeof(numSectors);
    memcpy(&nextFileHeaderSector, buf + offset, sizeof(nextFileHeaderSector)); offset += sizeof(nextFileHeaderSector);
    memcpy(dataSectors, buf + offset, NumDirect * sizeof(int));

    /* MP4 */
    /* Fetch nextFileHeader */
    if(nextFileHeaderSector != -1)
    {
        nextFileHeader = new FileHeader;
        nextFileHeader->FetchFrom(nextFileHeaderSector);
    }
}
```

這個函式就是要從 disk 中拿出 FileHeader 的資訊，而由於我們並沒有將 in-core information 放進 sector 中，因此我們這邊用 memcpy 就是按照原本放的順序拿出 numBytes, numSectors, nextFileHeaderSector 以及 dataSectors[ ]

而這邊由 nextFileHeaderSector 來判斷是否有 link 到下一個 FileHeader，有的話就會建立一個新的 FileHeader 並將 nextFileHeader 指向它，並透過它遞迴呼叫 FetchFrom(nextFileHeaderSector)，去 Disk 上將下一個 header 的資料取出。

## WriteBack() :

```
void
FileHeader::WriteBack(int sector)
{
    /* MP4 */
    char buf[SectorSize];
    int offset = sizeof(numBytes);

    /* We don't need to copy in-core data to disk */
    memcpy(buf, &numBytes, sizeof(numBytes));
    memcpy(buf + offset, &numSectors, sizeof(numSectors)); offset += sizeof(numSectors);
    memcpy(buf + offset, &nextFileHeaderSector, sizeof(nextFileHeaderSector)); offset += sizeof(nextFileHeaderSector);
    memcpy(buf + offset, dataSectors, NumDirect*sizeof(int));
    kernel->synchDisk->WriteSector(sector, buf);

    /* WriteBack nextFileHeader */
    if(nextFileHeaderSector != -1)
        nextFileHeader->WriteBack(nextFileHeaderSector);
}
```

我們在 writeback 時，不會把 in-core information 寫回 disk，因此依序用 memcpy() 將 numBytes, numSectors, nextFileHeaderSector 和 dataSectors[ ] 複製到 buf[ ] 裡，再用 WriteSector(sector, buf) 寫回 disk。最後也是用 nextFileHeaderSector 來遞回呼叫 WriteBack() 將所有 FileHeader 寫回。

## ByteToSector :

```
int
FileHeader::ByteToSector(int offset)
{
    /* MP4 */
    int index = offset / SectorSize;
    if(index < NumDirect)
        return(dataSectors[index]);
    else
        return nextFileHeader->ByteToSector(offset - MaxFileSize);
}
```

這個函式能回傳我們輸入的 File 的第 offset 個 byte 是在 disk 上的哪個 sector，如果 offset 算出是在第一個 FileHeader 的 dataSectors[ ] 範圍內，則回傳對應的 sector number。如果超出範圍之外，那麼就是在後面其他

FileHeader 的 dataSectors[ ] 上，因此遞迴呼叫ByteToSector() 繼續尋找對的 FileHeader 。

### FileLength :

藉由遞迴呼叫，將整個 linked list 的 FileHeader 中的 numBytes 累加在一起就是這個 File 的總 Size 了。

```
int
FileHeader::FileLength()
{
    /* MP4 */
    int totalNumBytes = numBytes;
    if(nextFileHeader != NULL)
        totalNumBytes += nextFileHeader->FileLength();
    return totalNumBytes;
}
```

### Print() :

```
void
FileHeader::Print()
{
    int i, j, k;
    char *data = new char[SectorSize];

    printf("FileHeader contents.  File size: %d.  File blocks:\n", numBytes);
    for (i = 0; i < numSectors; i++)
        printf("%d ", dataSectors[i]);
    printf("\nFile contents:\n");
    for (i = k = 0; i < numSectors; i++) {
        kernel->synchDisk->ReadSector(dataSectors[i], data);
        for (j = 0; (j < SectorSize) && (k < numBytes); j++, k++) {
            if ('\040' <= data[j] && data[j] <= '\176') // isprint(data[j])
                printf("%c", data[j]);
            else
                printf("\\%x", (unsigned char) data[j]);
        }
        printf("\n");
    }

    /* MP4 */
    if(nextFileHeader != NULL)
        nextFileHeader->Print();

    delete [] data;
}
```

主要是最下方的遞迴呼叫 linked list 每個 FileHeader 的 Print()。

到這邊的 implement 結束，完成第二部分 combine MP1 的 system call interface，以及利用 linked list 資料結構儲存過大的 FileHeader，因而能在不超過整體 disk 的狀況下 support 32KB 以上大小的 File。

## Part 3.

在 part 3 中，我們首先滿足 64 file/subdirectories in an directory，以及 file name and directory name will not exceed 9 characters，藉由以下的設定：

在 filesys.h 中，更改一個 directory 擁有的 entry 數量：

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

在 directory.h 中，原本已經寫好File Name的最長限制，因此不做更改：

```
#define FileNameMaxLen    9 // for simplicity, we assume  
                           // file names are <= 9 characters long
```

接下來要 support subdirectory structures，我們在 directory.h 中的 class DirectoryEntry 新增了一個 bool isDir，來表示這個 file 是不是 directory：

```
class DirectoryEntry {
public:
    bool isDir; /* MP4 */ /* Is this entry a sub dir ?*/

    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'
};
```

對 directory.h 做的一些小修改說明完後，我們看到 filesys.h 和 filesys.cc，這邊有對 “/” 的處理以分辨檔名跟 subdirectory 的重要函式，新增如下：

```
bool Create(char *pathName, int initialSize, bool isDir); /* MP4 */
// Create a file (UNIX creat)

OpenFile* Open(char *pathName); // Open a file (UNIX open)

bool Remove(char *pathName); // Delete a file (UNIX unlink)

/* MP4 */
void List(bool recursive, char* listDirectoryName);

void Print(); // List all the files and their contents

/* MP4 */
OpenFile *openFileTable[487];
int openFileTableTop;
OpenFile* FindSubDirectory(char* name);
```

我們將 **Create()** 新增了剛提到的在 **directory.h** 中新增的 **isDir**，並且在 **List()** 中新增了兩個變數傳入，以實現 recursively list the file/directory in a directory，最後剛說的重要函式便是最下面的 **OpenFile\*** **FindSubDirectory(char\* name)**，接下來便要詳細說明這些改變。

## filesys.cc :

### 1. **FindSubDirectory()** :

這函式主要是 **input** 一個字串指向輸入的一串包含 “/” 的 **path**，這個字串的尾端就是我們的 **file name** 。

這函式會回傳一個**OpenFile Object**，這個 **OpenFile** 就是包含這個檔名的 **directory file**，回傳以後可以提供其他函式利用它來**FetchFrom()**，來得到 **directory**。而剛開始**input**的那個字串則會被改為**file name** 。

首先透過**OpenFile\* curDirFile** 來紀錄目前的 **directory file**，透過它來從 **disk** 上讀取 **directory**，並放在 **Directory\* curDirectory** 上，初始狀態如下：

```
Directory *curDirectory = new Directory(NumDirEntries);
OpenFile* curDirFile = directoryFile; /* root file */
curDirectory->FetchFrom(directoryFile);
```

這裡的 **directoryFile** 就是 **root directory file** 。

接下來我們使用 **strtok()**函式來過濾掉 “/”，並拿到指向被分開的 **path name** 的 **char** 指標，這邊我們用了一個變數叫 **cut**，而在以迴圈找到 **subdirectory** 之前，我們先檢查了一個例外狀況，如果輸入的 **path** 只有一個 “/” 時，我們是回傳 **NULL**的，如下：

```
char* cut = strtok(name, "/");
if(cut == NULL)
{
    delete curDirectory;
    return NULL;
}
```

而在進入迴圈後，開了一個新的 char 指標 nextCut 來接受執行 strtok() 拿取過濾後的下一個 char 指標，並以 nextCut 來進行判斷，如果是 NULL 代表目前所拿取的 cut 已經是 path 的最後一段，也就是 target file/subdirectory，因此 return curDirFile；而如果不是 NULL，代表有下一層 subdirectory/file，因此我們就先讓 curDirectory 用 find(cut) 來檢查是否有 cut 代表的 subdirectory，如果沒有就回傳 NULL，如果找到了就代表我們能往下一層確認。

更新 curDirFile 為指向以 find(cut) 回傳的 sector 值所新建立的 OpenFile 物件的指標，而利用 FetchFrom(curDirFile) 來更新 curDirectory，並且將 nextCut assign 給 cut，程式如下：

```
char *nextCut = strtok(NULL, "/");
if(nextCut != NULL) /* Go deeper */
{
    //printf("Next cut exist(%s), go deeper\n", nextCut);
    /* Does sub-directory exist? */
    int sector = curDirectory->Find(cut);
    if(sector == -1)
    {
        printf("Sub-directory %s not found, return NULL\n\n",cut);
        delete curDirectory;
        if(curDirFile != directoryFile) delete curDirFile; /* Don't del root file */
        return NULL;
    }

    /* Deeper !!!! */
    if(curDirFile != directoryFile) delete curDirFile; /* Don't del root file */
    curDirFile = new OpenFile(sector);
    curDirectory->FetchFrom(curDirFile);
    printf("Change dir to %s\n",cut);
    //printf("Current Dir:\n");
    //curDirectory->List(FALSE, 0);
    cut = nextCut;
}
```

一直執行到 nextCut == NULL 時，由於一剛開始函式傳進來的 name 經過呼叫 strtok() 後，已經被改變的亂七八糟，因此最後我們將得到的 file name 覆蓋給 name 讓之後其他函式呼叫這個函式之後，傳入得 path 能被改成 target file name，然後 return curDirFile，程式如下：

```

else
{
    //printf("Next cut doesn't exist, stop \n");
    strcpy(name,cut); /* name will be modified to file name */
    delete curDirectory;
    return curDirFile;
}

```

這邊最後說明在這個函式中，我們在過程都有小心地將不用的物件 delete 掉，這是為了避免 memory leakage。而這個函式完成後，在接下來我們會說明 filesys.cc 的其他函式如何使用這個 FindSubDirectory()。

## 2. Create() :

多傳入的 bool isDir 用來在一剛開始判斷同樣傳入的 initialSize 是不是該改成 DirectoryFileSize。而接下來在執行原本的 code 之前，我們要先將傳進的 path name 處理過找出最後包含 target file 的 directory，才能讓下面原本的 code 順利照舊執行，透過FindSubDirectory()我們就可以拿到我們想要的 directory file，再透過FetchFrom讀取出 directory，之後的程式就不需改變，新增程式如下：

```

/* MP4 */ /* DirectoryFileSize */
if(isDir)    initialSize = DirectoryFileSize;

printf("Creating file, path:  %s, size %d bytes\n",pathName,initialSize);

/* MP4 */
/* Find the directory containing the target file */
char name[1000];
strcpy(name, pathName); /* prevent pathName being modified */
OpenFile* curDirFile = FindSubDirectory(name); /* name will be cut to file name */
if(curDirFile == NULL)  return FALSE; /* Directory file not found */
Directory *directory = new Directory(NumDirEntries);
directory->FetchFrom(curDirFile);

```

由於傳進 FindSubDirectory() 裡的字串會被 strtok() 更改，因此我們是複製 path name 到 name 這個字串，再把 name 傳進去，這樣當 FindSubDirectory() 結束以後，name 會被改成檔案的名字。

### 3. Open() :

一樣因為傳入的是 path name，因此也是要先經過一段處理後才能接著執行原本的 code，新加的程式如下：

```
char name[1000];
strcpy(name, pathName); /* prevent pathName being modified */
OpenFile* curDirFile = FindSubDirectory(name);
if(curDirFile == NULL)  return NULL; /* Directory file not found , return NULL */
Directory *directory = new Directory(NumDirEntries);
directory->FetchFrom(curDirFile);
```

一樣要傳入複製過後的 name 進去執行 FindSubDirectory()，找不到 directory 就回傳 NULL，找到就讀出這個 directory 後，照著原本的 code 接著執行。

### 4. Remove() :

跟上一個一樣，多加了尋找sub directory的地方，後面就是執行原本的 code：

```
char name[1000];
strcpy(name, pathName); /* prevent pathName being modified */
OpenFile* curDirFile = FindSubDirectory(name);
if(curDirFile == NULL)  return FALSE; /* Directory file not found */
Directory *directory = new Directory(NumDirEntries);
directory->FetchFrom(curDirFile);
```

### 5. List() :

在 List() 中，我們先判斷一下一種例外狀況就是如果傳進的 path name 就是 “ / ” 的話，我們就是要 list root directory，程式如下：

```

/* Special case : list root dir */
if(strcmp(listDirectoryName, "/") == 0)
{
    Directory *directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);
    directory->List(recursive,0);
    delete directory;
    return;
}

```

如果不是要 list root directory的話，就跟前面做的事一樣，獲取存有 target file 的 directory：

```

char listDirectoryName[1000];
strcpy(listDirectoryName, listDirectoryName); /* prevent pathName being modified */
OpenFile* curDirFile = FindSubDirectory(listDirectoryName);
if(curDirFile == NULL) return; /* Directory file not found */
Directory *directory = new Directory(NumDirEntries);
directory->FetchFrom(curDirFile);

```

在得到 target file(也是一個 directory)，將它讀取出來以後執行它的 List()，所以最後的程式如下：

```

/* Find the target dir */
int sector = directory->Find(listDirectoryName);
if (sector >= 0)
{
    OpenFile* tmpFile = new OpenFile(sector);
    Directory* tmpDir = new Directory(NumDirEntries);
    tmpDir->FetchFrom(tmpFile);
    tmpDir->List(recursive,0);
    delete tmpDir;
    delete tmpFile;
}

```

而最後這邊要特別在說明的是，FindSubDirectory()會建立一個 OpenFile，而所有使用 FindSubDirectory() 的地方，用完這個 OpenFile 要將他刪除。但有一個例外是這個 OpenFile 是 directoryFile( root dir file )，偵測程式如下：

```
/* Don't delete root directoryFile */
if(curDirFile != directoryFile && curDirFile != NULL) delete curDirFile;
```

## directory.h :

接下來，我們回到 directory.h 來說明，除了前面說明的更改 class DirectoryEntry 以外，我們在 class Directory 也新增 Add() 的 overload 版本，以及更改 List() 所傳入的參數：

```
/* MP4 */ /* overload */
bool Add(char *name, int newSector, bool isDir);

bool Remove(char *name);      // Remove a file from the directory

/* MP4 */ /* support recursive */
void List(bool recursive, int depth);      // Print the names of all the files
                                            // in the directory
```

## directory.cc :

### 1. Add() :

在原本的 Add() 中，新增在沒有傳入isDir參數的情況下，該 table entry的 isDir 皆設為 FALSE：

```
table[i].isDir = FALSE; /* MP4 */
```

而 overload 版本的 Add() 則會多傳入 isDir 值，讓我們依據他來設定，如下：

```
/* Overload */
bool Directory::Add(char *name, int newSector, bool isDir)
{
    if (FindIndex(name) != -1)
        return FALSE;

    for (int i = 0; i < tableSize; i++)
        if (!table[i].inUse)
        {
            table[i].isDir = isDir;
            table[i].inUse = TRUE;
            strcpy(table[i].name, name, FileNameMaxLen);
            table[i].sector = newSector;
            return TRUE;
        }
    return FALSE; // no space. Fix when we have extensible files.
}
```

## 2. List() :

多傳進了兩個參數(bool recursive , int depth) , recursive是讓我們判斷有無要 recursively list , depth則是讓我們知道目前進到第幾層的directory，方便output 格式的排版。

首先印出這個 directory 的所有 entry ，而當某個 entry 是 directory時，就會去檢查是否有要求要 recursive list，有的話就會把這個entry的資料從disk上讀出來，並遞迴呼叫List()來印出這個entry的資料。迴圈中的程式如下：

```

if (table[i].inUse)
{
    for(int j = 0 ; j < depth ; j++)    printf("    ");

    printf("<%d> ",num);
    printf("%s ", table[i].name);
    if(table[i].isDir)
    {
        printf("(dir)\n",table[i].sector);

        /* List recursively */
        if(recursive)
        {
            OpenFile* subDirFile = new OpenFile(table[i].sector);
            Directory* subDir = new Directory(NumDirEntries);
            subDir->FetchFrom(subDirFile);

            subDir->List(TRUE, depth+1);

            delete subDirFile;
            delete subDir;
        }
    }
    else printf("(file)\n",table[i].sector);
    num++;
}

```

而上面的排版及印出資訊執行結果如下：

```

Listing dir [/]...
<0> t0 (dir)
    <0> f1 (file)
    <1> aa (dir)
    <2> bb (dir)
        <0> f1 (file)
        <1> f2 (file)
        <2> f3 (file)
        <3> f4 (file)
    <3> cc (dir)
<1> t1 (dir)
<2> t2 (dir)

```

## Main.cc :

我們在讀輸入指令時，如果是“ -l ”或“ -lr ”時，會將 dirListFlag 設為 true，而“ -lr ”還會額外將 recursiveListFlag 設為 true，後頭呼叫 fileSystem 的 List() 更改如下：

```
if (dirListFlag) {  
    /* MP4 */  
    kernel->fileSystem->List(recursiveListFlag, listDirectoryName);  
}
```

而讀輸入指令時，如果是“ -mkdir ”時，會將 mkdirFlag 設為 true 進而呼叫 CreateDirectory()，而我們將 CreateDirectory() 更改成呼叫 fileSystem 的 Create()如下：

```
static void  
.CreateDirectory(char *name)  
{  
    /* MP4 */  
    /* Creating Dir is same as creating a file */  
    if(kernel->fileSystem->Create(name, 0, TRUE) ==FALSE)  
        printf("Unable to make directory %s\n",name);  
}
```

# Bonus.

## Bonus 1-1 :

Extend the disk from 128KB to 64MB :

更改了 disk.h 裡的這部分：

```
/* Extends disk from 128 KB ---> 64 MB */
const int NumTracks = 16384;      // number of tracks per disk
```

## Bonus 1-2 + 2 :

這三個 Bonus 要求在這裡一次解釋，就像前面提到的，我們是使用 linked indexed scheme，一個 file 可能會有一串的 FileHeader。

而看到在 FileHeader 的 Allocate() 中：

```
/* How much data bytes this header has? */
int remainFileSize = fileSize - MaxFileSize;
if(remainFileSize <= 0)/* We don't need next header */
    numBytes = fileSize;
else /* We need next header */
    numBytes = MaxFileSize;

numSectors = divRoundUp(numBytes, SectorSize);
if (freeMap->NumClear() < numSectors)
    return 0; // 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);

    /* MP4 */
    /* Clean this sector */
    char tmp[SectorSize]; for(int i=0 ; i<SectorSize ; i++) tmp[i] = 0;
    kernel->synchDisk->WriteSector(dataSectors[i], tmp);
}
```

前半依據 File 的大小來計算是否須要多於一個的 FileHeader，並且在分配 Free sector 時，我們用 kernel->synchDisk->WriteSector() 來將 sector 初始清空。

接下來後半：

```
/* Need next header */
if(remainFileSize > 0)
{
    /* Make a new header */
    nextFileHeaderSector = freeMap->FindAndSet();
    if(nextFileHeaderSector == -1) return FALSE;

    nextFileHeader = new FileHeader;
    return SectorSize + nextFileHeader->Allocate(freeMap, remainFileSize); /* Record the size of all total headers */
}

return SectorSize; /* 1 header 1 sector */
```

這邊就能看到我們是動態的根據 File 大小來配置 FileHeader 數量，也就是說 File 越大 FileHeader 數量就會依情況變多，利用 nextFileHeaderSector 來記錄下一個 FileHeader 的 sector number，並遞迴呼叫 Allocate()，因此我們不但能根據 File size 調整 FileHeader size，還能在 disk 總共大小允許下不斷 linked 下去，因此也可以儲存加 FileHeader size 後 64MB的 File 。

而這裡提供我們在不同大小的 File 時所配置的 FileHeader 大小結果圖：

```
Creating file path [/100]    size [1000 bytes]
Find desired directory.
Start creating [100]
Successfully allocated, total headers' size [128 bytes]

Opening File [100]

Creating file path [/3MB]    size [3145729 bytes]
Find desired directory.
Start creating [3MB]
Successfully allocated, total headers' size [108544 bytes]

Opening File [3MB]

Creating file path [/5MB]    size [5242881 bytes]
Find desired directory.
Start creating [5MB]
Successfully allocated, total headers' size [180864 bytes]

Opening File [5MB]

Listing dir [/]...
<0> 100 (file)
<1> 3MB (file)
<2> 5MB (file)
```

## Bonus 3 :

我們修改了 `fsys.cc` 中的 `Remove()`，多傳入了參數 `bool recursive`，用來判斷有沒有要 `recursive remove`。

之後在移除 target file 的時候，會去判斷這個 file 是不是 directory，如果是的話而且又有要求 recursive remove ，就會將這個 directory 從 disk 上讀出，並且刪除他所記錄的每一個檔案( 透過遞迴呼叫Remove() )

新增的這段程式如下：

```
/* Recursively remove , only if this file is a dir */
if(directory->isDir(name) && recursive)
{
    /* Read out the target dir */
    OpenFile* tarOpenFile = new OpenFile(sector);
    Directory* tarDirectory = new Directory(NumDirEntries);
    tarDirectory->FetchFrom(tarOpenFile);

    /* Creating Target's Path Name */
    char tarPathName[1000];
    strcpy(tarPathName, pathName); /* pathName: /a/b/c */
    int offset = strlen(tarPathName);
    tarPathName[offset] = '/'; /* tarPathName : /a/b/c/ */

    /* Remove all things in the target directory */
    for(int i=0 ; i<tarDirectory->tableSize ; i++)
    {
        if(tarDirectory->table[i].inUse)
        {
            strcpy(tarPathName+offset+1, tarDirectory->table[i].name); /* Append the file name */
            Remove(TRUE, tarPathName);
        }
    }

    delete tarOpenFile;
    delete tarDirectory;
}
```

為了讓 Remove() 可以判斷 target file 是不是 directory，在 directory.cc 裡新增 isDir()，如下：

```
/* Used for recursive remove */
bool Directory::isDir(char* name)
{
    int i = FindIndex(name);

    if (i == -1)    return FALSE;      // name not in directory
    return table[i].isDir;
}
```

而 main.cc 在讀取輸入指令時，如果是 ”-rr” 則會將 recursiveRemoveFlag 設為 true，並在後面判斷時呼叫 fileSystem->Remove()，並將 recursiveRemoveFlag 傳入已執行 recursively remove，如下：

```
if (removeFileName != NULL) {
    /* MP4 */
    /* Support recursive Remove */
    kernel->fileSystem->Remove(recursiveRemoveFlag, removeFileName);
}
```

組員貢獻：

103062121 劉亮廷：50% trace code, implement code, report  
103062238 林子淵：50% implement code, report