MP4_Report_47

HackMD 好讀版：連結 (https://hackmd.io/@OJo2ruXGShKdpuewtwzZcQ/ryawfBira)

團隊分工表

團隊成員	Trace code	文件撰寫	功能實作
111065542 楊智堯	v	v	v
111065531 郭芳妤	v	v	v

Trace code

Part I. Understanding NachOS file system

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

- How to manage free block space?

Short answer: Use Bitmap to manage free block space

Bitmap 用來追蹤在 disk 中的 free blocks。當系統初始化時帶有 `-f` 指令，則此時將 `formatFlag` 設置成 `TRUE` ，意旨 FileSystem 需要 format the disk。

```
// in kernel.cc
Kernel::Kernel(int argc, char **argv)
{
    if (strcmp(argv[i], "-f") == 0) {
        formatFlag = TRUE;
    }
}

void
Kernel::Initialize()
{
    fileSystem = new FileSystem(formatFlag);
}

// in filesys.cc

FileSystem::FileSystem(bool format)
{
    if (format) {
        PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);
        Directory *directory = new Directory(NumDirEntries);
       FileHeader *mapHdr = new FileHeader;
       FileHeader *dirHdr = new FileHeader;
    }
}
```

若無需 formatting, 則系統會直接將檔案開啟在 sector 0 的位置，並且在 NachOS 執期間會存在 memory 中。

```
FileSystem::FileSystem(bool format)
{
    if (format)
    {
        ...
    }
    else
    {
        freeMapFile = new OpenFile(FreeMapSector);
        directoryFile = new OpenFile(DirectorySector);
    }
}
```

- How to find free block space?

Short Answer: 在初始化的時候將 Bitmap 全設成 0（0 代表未使用，1 代表已使用）。使用 `FindAndSet()` 找到 bit 為 0 的 index，`Mark()` 成 1（將被使用）後返回 index。如果找不到 bit 為 0 的 index 則傳回 -1。

此處使用的是 `PersistentBitmap`，繼承自 `Bitmap`，可視為多了 FS operation 的 Bitmap。

```
// in filesys.cc

FileSystem::FileSystem(bool format)
{
    if (format) {
        PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);
        Directory *directory = new Directory(NumDirEntries);
       FileHeader *mapHdr = new FileHeader;
       FileHeader *dirHdr = new FileHeader;
    }
}

// in pbitmap.h
class PersistentBitmap : public Bitmap {};

// in bitmap.h
class Bitmap {
  public:
    Bitmap(int numItems);
}
```

此時找到 `Bitmap` 位置，可看到此處初始化全部 `Bitmap` 的位置為 0。

```
// in bitmap.cc
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);
    }
}
```

此處以 `FileSystem::Create` 內的程式碼為例，當需要找到空的位置的時候

```
sector = freeMap->FindAndSet(); // find a sector to hold the file header
```

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

- Where is this information stored on the raw disk

Short answer: Sector 2
Bitmap (freeMapFile) 的 file header 存在 sector 0 (FreeMapSector). Sector 1 用來存 directory file's header。而真正的 Bitmap data(追蹤 disk 內每個 sector free 或 used 的狀態) 則存在其他 sector，通常是 sector 2。

細部分解的話：

1. 分配 File Header 的空間

在 FileSystem 的建構子中 (FileSystem::FileSystem(bool format)) 當 (format = true) 則需要做 formatting，此時會將空間分配給 `freeMapFile` 和 `directoryFile` 的 File headers

```
// New FileHeader
FileHeader *mapHdr = new FileHeader;

// Allocate space for FileHeaders for the directory and bitmap
freeMap->Mark(FreeMapSector); // 0 is marked

// Allocate space for the data blocks containing the contents
mapHdr->Allocate(freeMap, FreeMapFileSize)
```

2. Writing the Bitmap File Header to Disk:

`Bitmap` file header 寫回 Disk。

```
// Flush the bitmap and directory FileHeaders back to disk
mapHdr->WriteBack(FreeMapSector);
```

FreeMapSector 定義為 sector 0 (i.e., `#define FreeMapSector 0`), 意思是 bitmap file header(`freeMapFile`) 存在硬碟內 sector 0 的位置。

3. Opening and Using the freeMapFile:

The bitmap file (`freeMapFile`) is then opened from the FreeMapSector.

```
// OK to open the bitmap and directory files now
// The file system operations assume these two files are left open
// while Nachos is running.
freeMapFile = new OpenFile(FreeMapSector);
```

這個 file 表示 bitmap(track 使用或未使用過的 disk sector)。

4. Storage Location of the Bitmap Information:

```
#define FreeMapFileSize (NumSectors / BitsInByte)
```

FreeMapFileSize = 1024 / 8 = 128 Bytes 即一個 sector 即可儲存。

Sector 0 存 bitmap 的 fileheader
Sector 1 存 directory 的 fileheader
實際紀錄每個 sector 是否已使用的資訊則存到 Sector 2

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

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

Disk 的空間 = SectorSize * NumSectors = SectorSize * (SectorsPerTrack * NumTracks) = 128 * (32 * 32) = 128 KB

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

- How to manage directory data structure?

0. 如果不需要格式化，直接打開 sector 1 所在的檔案並保留在記憶體中。

```
FileSystem::FileSystem(bool format)
{
    if (format)
    {
        ...
    }
    else
    {
        freeMapFile = new OpenFile(FreeMapSector);
        directoryFile = new OpenFile(DirectorySector);
    }
}
```

1. 若需格式化 Directory，建立根目錄後初始化底下的 `DirectoryEntry`

```
// in filesys.cc
#define NumDirEntries 10

FileSystem::FileSystem(bool format)
{
    if (format) {
        Directory *directory = new Directory(NumDirEntries);
        FileHeader *dirHdr = new FileHeader;
    }
}

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

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

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

2. `DirectoryEntry` 內有 `inUse` , `sector` 及 `name` 變數，透過 `memset` 將其值設定為 0/false/empty，為了確保 `inUse` 設定成 `False` ，再跑一次 for-loop。

```
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'
};
```

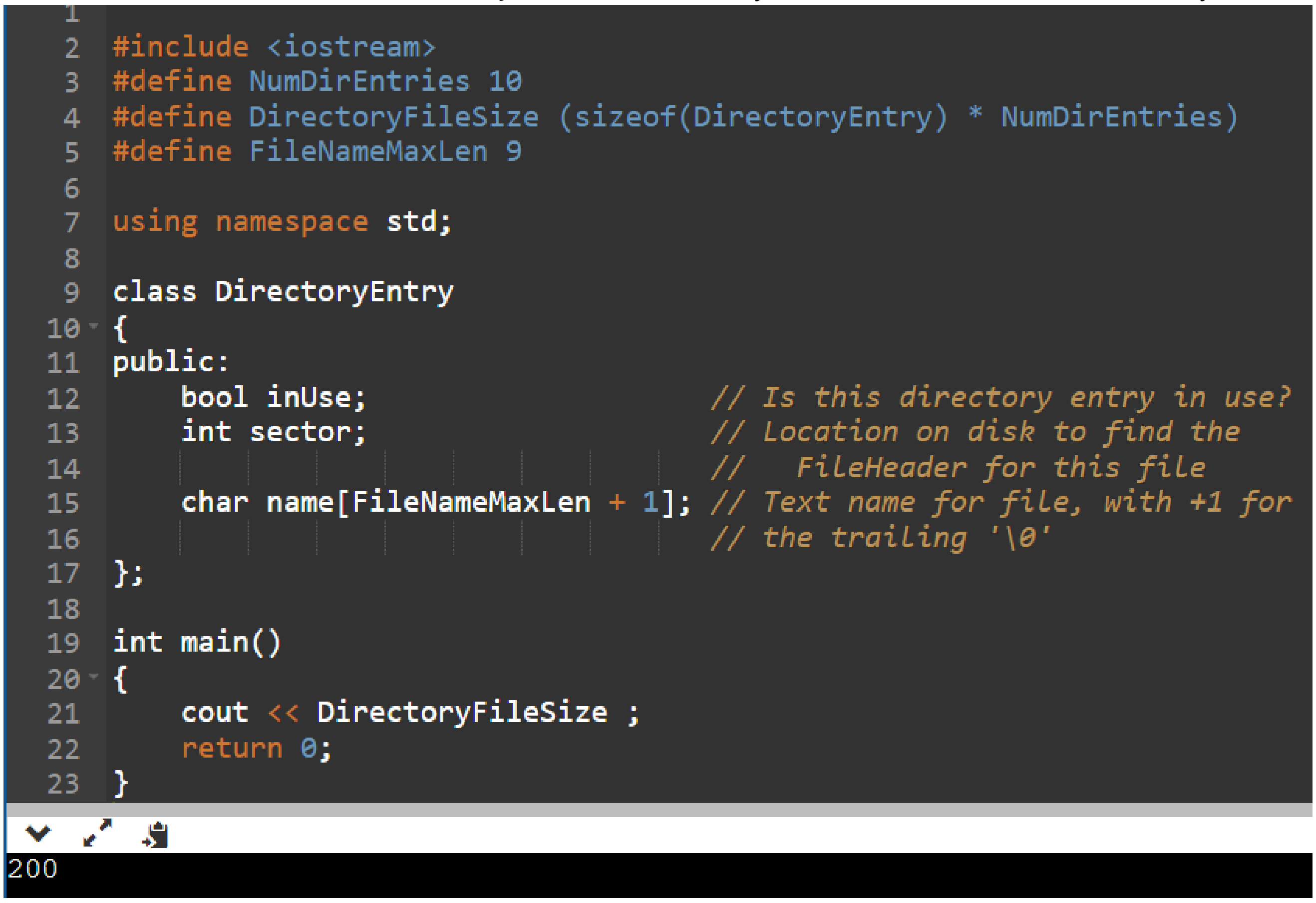
3. 在 FileSystem 的建構子 FileSystem::FileSystem(bool format) 中，我們開啟 `directoryFile` 儲存 filename

```
// New FileHeader
FileHeader *dirHdr = new FileHeader;

// Allocate space for FileHeaders for the directory and bitmap
freeMap->Mark(DirectorySector); // 1 is marked

// Allocate space for the data blocks containing the contents
dirHdr->Allocate(freeMap, DirectoryFileSize)

// Flush the bitmap and directory FileHeaders back to disk
dirHdr->WriteBack(DirectorySector);

// OK to open the bitmap and directory files now
// The file system operations assume these two files are left open
// while Nachos is running.
directoryFile = new OpenFile(DirectorySector);
```

- Where is directory data information stored on the raw disk (which sector)?

Short answer: Sector 3 and Sector 4
Sector 0 存 Bitmap (freeMapFile) 的 file header。
Sector 1 用來存 directory's file header。
Sector 2 存 Bitmap data。

實際需要兩個 Sector 原因是 `DirectoryFileSize` 是 200 Bytes，而一個 Sector Size 是 128 Bytes。



```
#include <iostream>
#define NumDirEntries 10
#define DirectoryFileSize (sizeof(DirectoryEntry) * NumDirEntries)
#define FileNameMaxLen 9

using namespace std;

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 main()
{
    cout << DirectoryFileSize ;
    return 0;
}
```

```
200
```

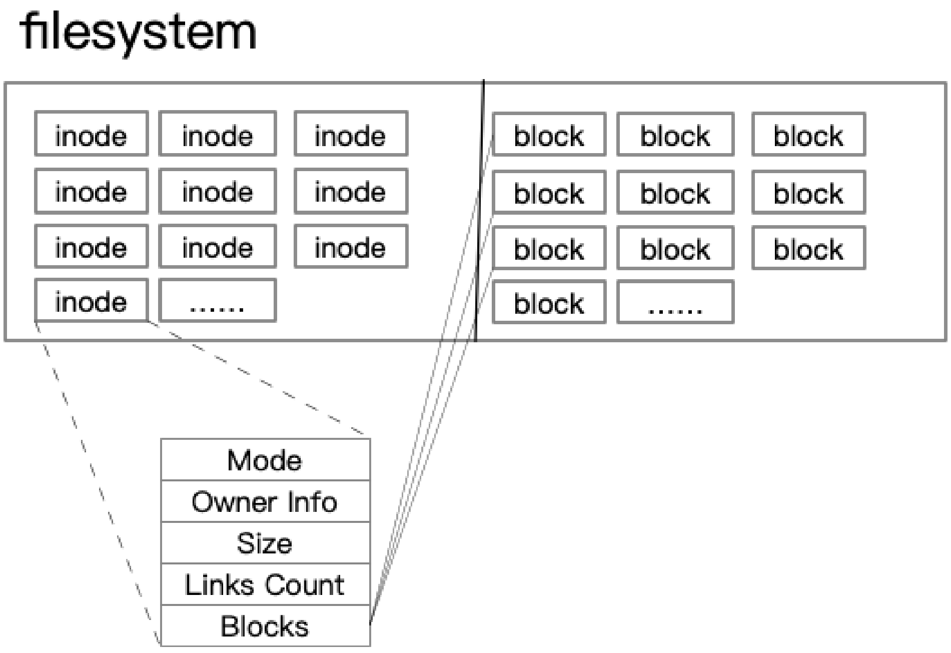
細部分解 `DirectoryFileSize`，以下是 `sizeof(DirectoryEntry)` 細節：
Total: 1 (bool) + 3 (padding) + 4 (int) + 10 (char array) + 2 (padding) = 20 bytes

其中最大的 type 是 `int` (4 Bytes)，所以會 padded 到 4 的倍數，而 `NumDirEntries` 是 10，故 `DirectoryFileSize` = 20 * 10 = 200(Bytes)。

(4) What information is stored in an inode? Use a figure to illustrate the disk allocation scheme of the current implementation.

- 背景知識

一般的 Linux System 會把 FileSystem 分成 `block` 以及 `inode`，文件的內容存在 `block`，而 metadata(owner, size 等) 會存在 `inode`



圖片來源:https://www.kawabangga.com/posts/3561 (https://www.kawabangga.com/posts/3561)

- 回到 NachOS

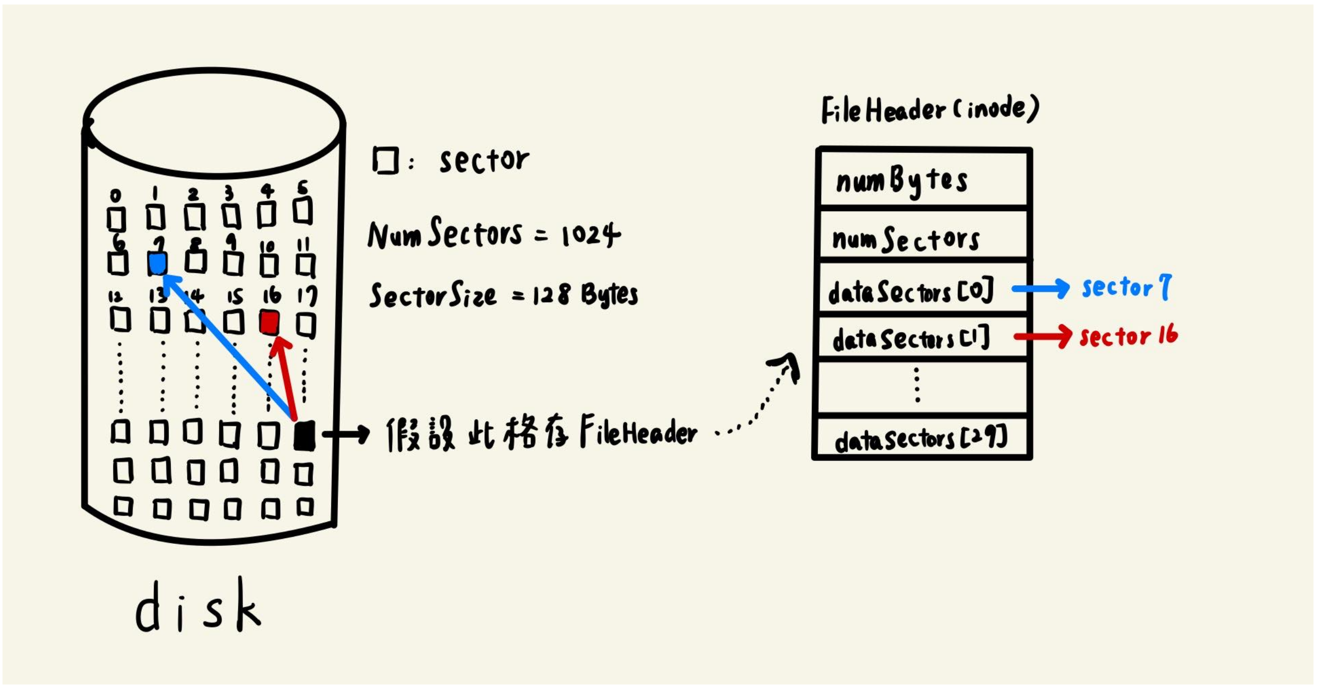
以此對應到 NachOS 的設計，會看到這些 metadata 存在 `class FileHeader`

```
- numBytes： 檔案的大小幾個 Bytes
- numSectors： 檔案用了幾個 Sectors
- dataSectors： 這個檔案連結的 Sectors 位置
```

dataSectors 可以存 `NumDirect` 個 sectors，
\[NumDirect = (128 - 2*4) / 4 = 30 \]

```
#define NumDirect ((SectorSize - 2 * sizeof(int)) / sizeof(int))
class FileHeader
{
private
        int numBytes;  // Number of bytes in the file
        int numSectors;  // Number of data sectors in the file
        int dataSectors[NumDirect]; // Disk sector numbers for each data
                        // block in the file
};
```

畫成圖如以下所示



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

呈上題，因一個 file 只能分配到一個 sector，且一個 sector 最多只會有 30 個 entry(sectors)，故一個 sector 佔 128 Bytes。

故 maximum file size = 30 * 128 Bytes = 3,840 Bytes = 3.75 KB(3,840 / 1,024)

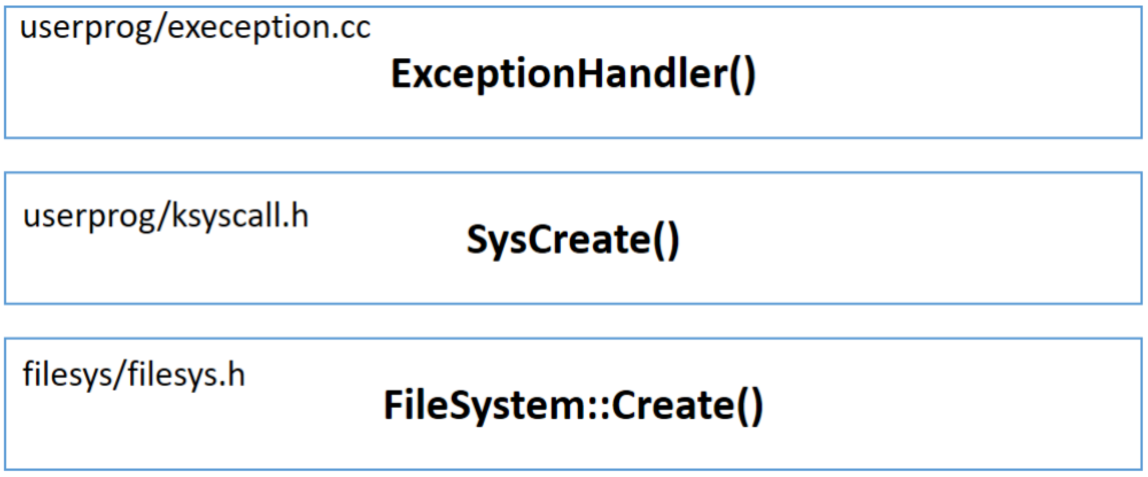
Implement File System

Part II. Modify the file system code to support file I/O system calls and larger file size

(1) Combine your MP1 file system call interface with NachOS FS to implement five system calls:

int Create(char *name, int size);

首先整個 Systemc call 的流程為



在 `exception.cc` 裡，更改 `SC_Create` 的部分

```
#ifdef FILESYS_STUB
...
#else
    case SC_Create:
        val = kernel->machine->ReadRegister(4);
        {
            int size = kernel->machine->ReadRegister(5);
            char* filename = &(kernel->machine->mainMemory[val]);
            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) +
        kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg
        return;
        ASSERTNOTREACHED();
        break;

#endif
```

找到原先 `SysCreate` 的位置(`ksyscall.h`)

```
#ifdef FILESYS_STUB
...
#else
int SysCreate(char* name, int size) {
    return kernel->fileSystem->Create(name, size);
}
#endif
```

發現該函數已經在 `filesys.h` 故可直接使用

```
bool Create(char *name, int initialSize);
```

OpenFileId Open(char *name);

在 `filesys.h` 新增

```
OpenFileId OpenAFile(char *name);
OpenFile *fileDescriptorTable[20];
```

在 `filesys.cc` 實作 `OpenAFile()`

```
OpenFileId
FileSystem::OpenAFile(char *name) {
    OpenFile* fd = Open(name);
    if(fd == NULL) return -1;

    for(int i = 0; i < 20; i++) {
        if(fileDescriptorTable[i] == NULL) {
            fileDescriptorTable[i] = fd;
            return i;
        }
    }

    delete fd;
    return -1;
}
```

int Read(char *buf, int size, OpenFileId id);

在 `filesys.h` 新增

```
int ReadAFile(char *buffer, int size, OpenFileId id)
```

在 `filesys.cc` 實作 `ReadAFile()`

```
int
FileSystem::ReadAFile(char *buffer, int size, OpenFileId id) {
    if(id < 0 || id > 19) return -1;
    if(size < 0) return -1;
    if(fileDescriptorTable[id] == NULL) return -1;

    return fileDescriptorTable[id]->Read(buffer, size);
}
```

int Write(char *buf, int size, OpenFileId id);

在 `filesys.h` 新增

```
int WriteAFile(char *buffer, int size, OpenFileId id)
```

在 `filesys.cc` 實作 `WriteAFile()`

```
int
FileSystem::WriteAFile(char *buffer, int size, OpenFileId id) {
    if(id < 0 || id > 19) return -1;
    if(size < 0) return -1;
    if(fileDescriptorTable[id] == NULL) return -1;

    return fileDescriptorTable[id]->Write(buffer, size);
}
```

int Close(OpenFileId id);

在 `filesys.h` 新增

```
int CloseAFile(char *buffer, int size, OpenFileId id)
```

在 `filesys.cc` 實作 `CloseAFile()`

```
int
FileSystem::CloseAFile(OpenFileId id) {
    if(id < 0 || id > 19) return -1;
    if(fileDescriptorTable[id] == NULL) return -1;
    delete fileDescriptorTable[id];
    fileDescriptorTable[id]= NULL;

    return 1;
}
```

在實作的時候發現有很多共同處，故小小重構一下

in `filesys.h`

```
class FileSystem
{
public:
        ...
private:
        bool IsValidFileId(OpenFileId id);
```

in `filesys.cc`

```
//----------------------------------------------------------------------
bool
FileSystem::IsValidFileId(OpenFileId id) {
    return !(id < 0 || id > 19 || fileDescriptorTable[id] == NULL);
}

//----------------------------------------------------------------------
int
FileSystem::ReadAFile(char *buffer, int size, OpenFileId id) {
    if (!IsValidFileId(id) || size < 0) return -1;
    return fileDescriptorTable[id]->Read(buffer, size);
}

//----------------------------------------------------------------------
int
FileSystem::WriteAFile(char *buffer, int size, OpenFileId id) {
    if (!IsValidFileId(id) || size < 0) return -1;
    return fileDescriptorTable[id]->Write(buffer, size);
}
//----------------------------------------------------------------------
int
FileSystem::CloseAFile(OpenFileId id) {
    if (!IsValidFileId(id)) return -1;
    delete fileDescriptorTable[id];
    fileDescriptorTable[id]= NULL;

    return 1;
}
```

後來發現 `/userprog/ksyscall.h` 及 `userprog/excpetion.cc` 需要修改

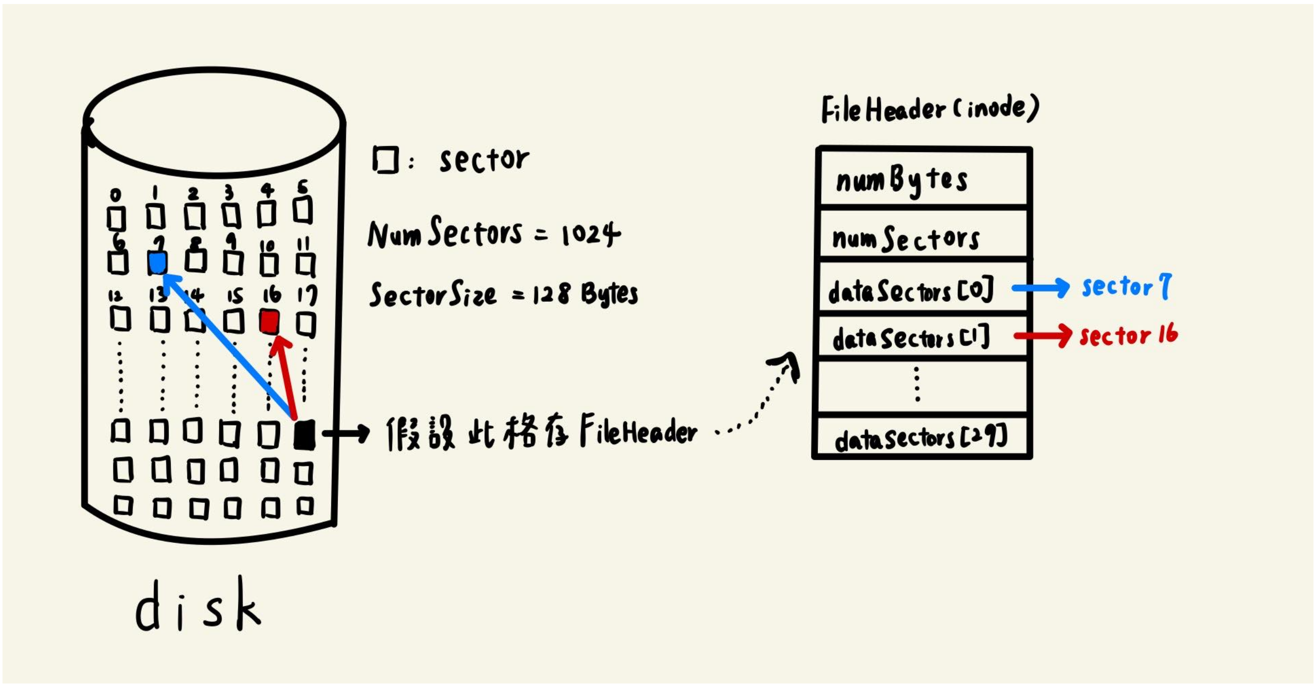
```
// /userprog/ksyscall.h
#ifdef FILESYS_STUB
...
#else
int SysCreate(char* name, int size) {
        return kernel->fileSystem->Create(name, size);
}

int SysOpen(char* name) {
        return kernel->fileSystem->OpenAFile(name);
}

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

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

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

```
// /userprog/excpetion.cc
case SyscallException:
    switch (type)
    {
    case SC_Open:
    val = kernel->machine->ReadRegister(4);
    {
        char *filename = &(kernel->machine->mainMemory[val]);
        status = SysOpen(filename);
        kernel->machine->WriteRegister(2, (int) status);
    }
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4)
    return;
    ASSERTNOTREACHED();
    break;
    case SC_Write:
    val = kernel->machine->ReadRegister(4);
    {
        char* fileBuffer = &(kernel->machine->mainMemory[val]);
        int fileSize = kernel->machine->ReadRegister(5);
        int fileID = kernel->machine->ReadRegister(6);
        status = SysWrite(fileBuffer, fileSize, (OpenFileId) fileID);
        kernel->machine->WriteRegister(2, (int) status);
    }
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4)
    return;
    ASSERTNOTREACHED();
    break;
    case SC_Read:
    val = kernel->machine->ReadRegister(4);
    {
        char* fileBuffer = &(kernel->machine->mainMemory[val]);
        int fileSize = kernel->machine->ReadRegister(5);
        int fileID = kernel->machine->ReadRegister(6);
        status = SysRead(fileBuffer, fileSize, (OpenFileId) fileID);
        kernel->machine->WriteRegister(2, (int) status);
    }
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4)
    return;
    ASSERTNOTREACHED();
    break;
    case SC_Close:
    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;
```

(2) Enhance the FS to let it support up to 32KB file size

我們此處將 `Indexed Allocation` 進一步用 `Linked Indexed Scheme` ，讓 File 能夠處理的檔案長度增加。

現在的實作圖解



目標實作如下
image

故 `FileHeader(inode)` 需要再空出一格指向下一個 Sector(圖中的 `next`) 指標的空間

```
// in filehdr.h
#define NumDirect ((SectorSize - 3 * sizeof(int)) / sizeof(int))
```

每一格 `FileHeader(inode)` 需要存以下兩個資訊

```
// in filehdr.h
class FileHeader
{
public:
    ...
private:
        FileHeader *nextFileHeader;
        int nextFileHeaderSector;
};
```

開始實作 `Fileheader` 的細節，在初始化以及解構時需要新增兩個新設變數的初值

```
//in filehdr.cc
FileHeader::FileHeader()
{
        nextFileHeader = NULL;
        nextFileHeaderSector = -1;
}

FileHeader::~FileHeader()
{
    if(nextFileHeader != NULL) delete nextFileHeader;
}
```

在 `Allocate` 時，用 `sectorsToAllocate` 去追蹤一個 `Fileheader` 是否可以裝滿，若可以 (`sectorsToAllocate == 0`) 則傳回 `True` ，若不行，則此時 `sectorsToAllocate > 0` ，設定新的 `nextFileHeader` 以及 `nextFileHeaderSector` 並再次 `Allocate` 空間

```
bool FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
{
    numBytes = fileSize;
    numSectors = divRoundUp(fileSize, SectorSize);
    if (freeMap->NumClear() < numSectors)
        return FALSE; // not enough space

    int sectorsToAllocate = numSectors;
    for(int i = 0; i < NumDirect; i++) {
        if(sectorsToAllocate == 0) break;
        dataSectors[i] = freeMap->FindAndSet();

        ASSERT(dataSectors[i] >= 0);
        sectorsToAllocate--;
    }

    // Check if we need another FileHeader for the remaining sectors
    if(sectorsToAllocate > 0) {
        nextFileHeader = new FileHeader();
        if(nextFileHeader == NULL) return FALSE;

        // Allocate a sector for the next file header
        nextFileHeaderSector = freeMap->FindAndSet();
        if(nextFileHeaderSector == -1) return FALSE;

        return nextFileHeader->Allocate(freeMap, sectorsToAllocate * SectorSize);
    }
    return TRUE;
}
```

並且更新 `Deallocate` 的 code

```
void FileHeader::Deallocate(PersistentBitmap *freeMap)
{
...
    if(nextFileHeader != NULL) nextFileHeader->Deallocate(freeMap);
}
```

> while Allocate sets up the in-memory structure when the file's size is set or extended, FetchFrom needs to independently reconstruct this structure when the file header is read from disk, ensuring that the file system works correctly in all scenarios where file headers are used.

根據提示完成 `WriteBack()`

> /*
> 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));
> ...
> */

```
void FileHeader::WriteBack(int sector)
{

    // Create a temporary buffer
    char buf[SectorSize];
    memset(buf, 0, SectorSize); // Initialize the buffer with zeros

    // Copy only the necessary fields into the buffer
    memcpy(buf, &numBytes, sizeof(numBytes));
    memcpy(buf + sizeof(numBytes), &numSectors, sizeof(numSectors));
    memcpy(buf + sizeof(numBytes) + sizeof(numSectors), dataSectors, sizeof(int) * Nu
    memcpy(buf + sizeof(numBytes) + sizeof(numSectors) + sizeof(int) * NumDirect, &ne

     // Write the buffer to the disk
    kernel->synchDisk->WriteSector(sector, buf);

    // Recursively write the next file header if it exists
    if(nextFileHeaderSector != -1 && nextFileHeader != NULL) {
        nextFileHeader->WriteBack(nextFileHeaderSector);
    }
}
```

緊接著更新 `FetchFrom` ，此處的目的是 `Fetch contents of file header from disk.` ，因為 disk 讀出的資料不包含 nextHeader，所以需要額外處理，詳細原因見時做後的解釋

```
void FileHeader::FetchFrom(int sector)
{
        char *buffer = new char[SectorSize]; // Ensure buffer is allocated
        kernel->synchDisk->ReadSector(sector, buffer);


         // Extract data from the buffer into the FileHeader fields
    int offset = 0;
    memcpy(&numBytes, buffer + offset, sizeof(numBytes));
    offset += sizeof(numBytes);
    memcpy(&numSectors, buffer + offset, sizeof(numSectors));
    offset += sizeof(numSectors);
    memcpy(dataSectors, buffer + offset, sizeof(int) * NumDirect);
    offset += sizeof(int) * NumDirect;
    memcpy(&nextFileHeaderSector, buffer + offset, sizeof(nextFileHeaderSector));
        /*
                MP4 Hint:
                After you add some in-core informations, you will need to rebuild the
        */
        // Check if there is a linked file header
        if(nextFileHeaderSector != -1) {
                if(nextFileHeader == NULL) {
                        // Allocate memory for the next file header
                        nextFileHeader = new FileHeader();
                }
                // Fetch the next file header from its sector
                nextFileHeader->FetchFrom(nextFileHeaderSector);
        }
         delete[] buffer; // Clean up allocated memory
}
```

在 `BytetoSector()` 的轉換只須注意是否超過一個 FileHeader 的空間，超過後只要指到下一個 FileHeader 即可

```
int FileHeader::ByteToSector(int offset) {
    int sectorOffset = offset / SectorSize;  // Calculate the sector offset relative

    // Check if the offset is within the range of the current file header
    if (sectorOffset < NumDirect) {
        return dataSectors[sectorOffset];
    } else {
        // If the offset is beyond the current file header, adjust the offset for the
        int adjustedOffset = offset - (NumDirect * SectorSize);
        if (nextFileHeader != NULL) {
            return nextFileHeader->ByteToSector(adjustedOffset);
        } else {
            // Error handling: offset is out of bounds of the file
            ASSERT(false);  // Offset is beyond the end of the file
            return -1;  // Return an invalid sector
        }
    }
}
```

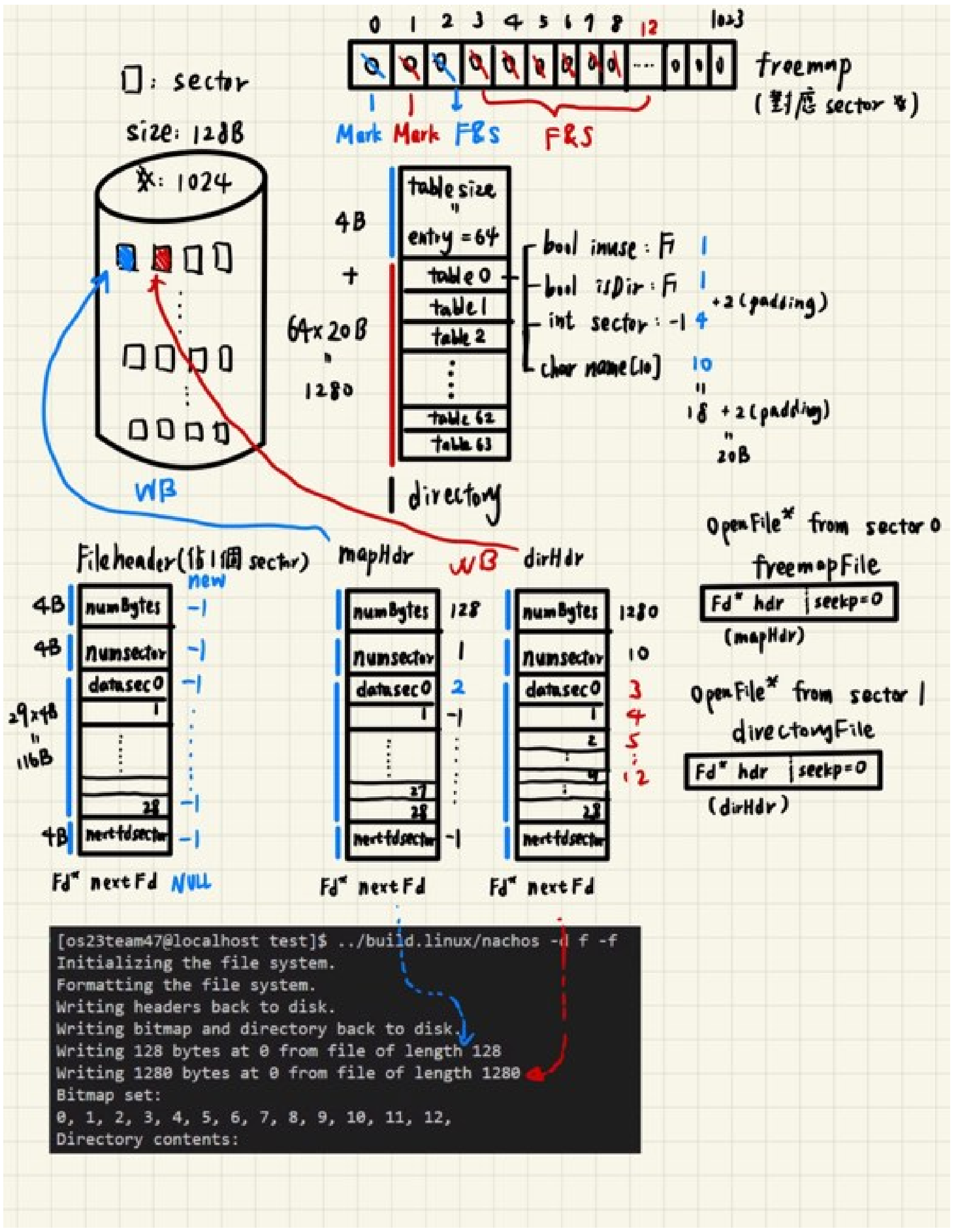
實作 FileLength

```
int FileHeader::FileLength()
{

    return numBytes;
}
```

```
void
FileHeader::Print() {
        char* data = new char[SectorSize];
        if(data == NULL) {
                printf("Unable to allocate memory for printing\n");
        return;
        }

        // Phase 1: Print all the blocks
    PrintBlocks();

        // Phase 2: Print the file contents
    printf("\nFile contents:\n");
    PrintContents(data, 0);

        delete[] data;
}

void
FileHeader::PrintBlocks() {
        printf("FileHeader contents.  File size: %d.  File blocks:\n", numBytes);
        for(int i = 0; i < NumDirect; i++) {
                if(dataSectors[i] == -1) {
                        break; // No more sectors to print
                }
                printf("%d, ", dataSectors[i]);
        }
        printf("\n");

        // Recursively print the blocks of the next file header if it exists
        if(nextFileHeaderSector != -1 && nextFileHeader != NULL) {
                nextFileHeader->PrintBlocks();
        }
}

void
FileHeader::PrintContents(char* data, int startByte) {
        int k = startByte;
        for(int i = 0; i < NumDirect && k < numBytes; i++) {
                if(dataSectors[i] == -1) {
                        break; // No more sectors to print
                }

                kernel->synchDisk->ReadSector(dataSectors[i], data);
                for(int 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");
        // Recursively print the contents of the next file header if it exists
    if (nextFileHeaderSector != -1 && nextFileHeader != NULL) {
        nextFileHeader->PrintContents(data, k);
    }
}
```

Part III. Modify the file system code to support the subdirectory

此處需要在 create 時就確定是檔案或是 `directory` ，於是我們順著 `format` 跟 建立檔案的邏輯更新報告。



```
[os23team47@localhost test]$ ../build.linux/nachos -d f -f
Initializing the file system.
Formatting the file system.
Writing headers back to disk.
Writing bitmap and directory back to disk.
Writing 128 bytes at 0 from file of length 128
Writing 1280 bytes at 0 from file of length 1280
Bitmap set:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
Directory contents:
```

目前建立檔案的邏輯會分為兩種

1. 建立新資料夾

```
../build.linux/nachos -mkdir /t1
```

2. 從 Unix 系統複製到 Nachos 建立檔案

```
../build.linux/nachos -cp num_100.txt /t0/f1
```

觀察 `-mkdir flag` ，發現最後是實作 `fileSystem::Create()`

```
else if (strcmp(argv[i], "-mkdir") == 0)
{
    // MP4 mod tag
    ASSERT(i + 1 < argc);
    createDirectoryName = argv[i + 1];
    mkdirFlag = true;
    i++;
}

if (mkdirFlag)
{
    // MP4 mod tag
    CreateDirectory(createDirectoryName);
}

static void CreateDirectory(char *name)
{
    // MP4 Assignment
    if (!kernel->fileSystem->Create(name, -1, TRUE))
    { // Create Nachos file
        printf("Mkdir: couldn't create directory %s\n", name);
        return;
    }
}
```

再觀察 `-cp flag` ，也發現最後是實作 `fileSystem::Create()`

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

if (copyUnixFileName != NULL && copyNachosFileName != NULL)
{
    Copy(copyUnixFileName, copyNachosFileName);
}
static void Copy(char *from, char *to)
{
    ...
    // Create a Nachos file of the same length
    if (!kernel->fileSystem->Create(to, fileLength, FALSE))
    { // Create Nachos file
        printf("Copy: couldn't create output file %s\n", to);
        Close(fd);
        return;
    }
}
```

進一步來觀察 `fileSystem::Create()` ，實作邏輯為下

1. 目前輸入的檔案會是絕對路徑，故無論是檔案或是資料夾傳入的格式皆為 (`/aaa/bbb/ccc` 或 `/aaa`) 故需要先將其切為各個部分(`/aaa/bbb/ccc -> /aaa, /bbb, /ccc`) 且在 Traverse 後順便紀錄有幾層資料。

以下 `SplitPath()` 如 `path` 傳入 `/aaa/bbb/ccc` 則
`dirs[0] = aaa, dirs[1] = bbb, dirs[2] = ccc, count = 3`

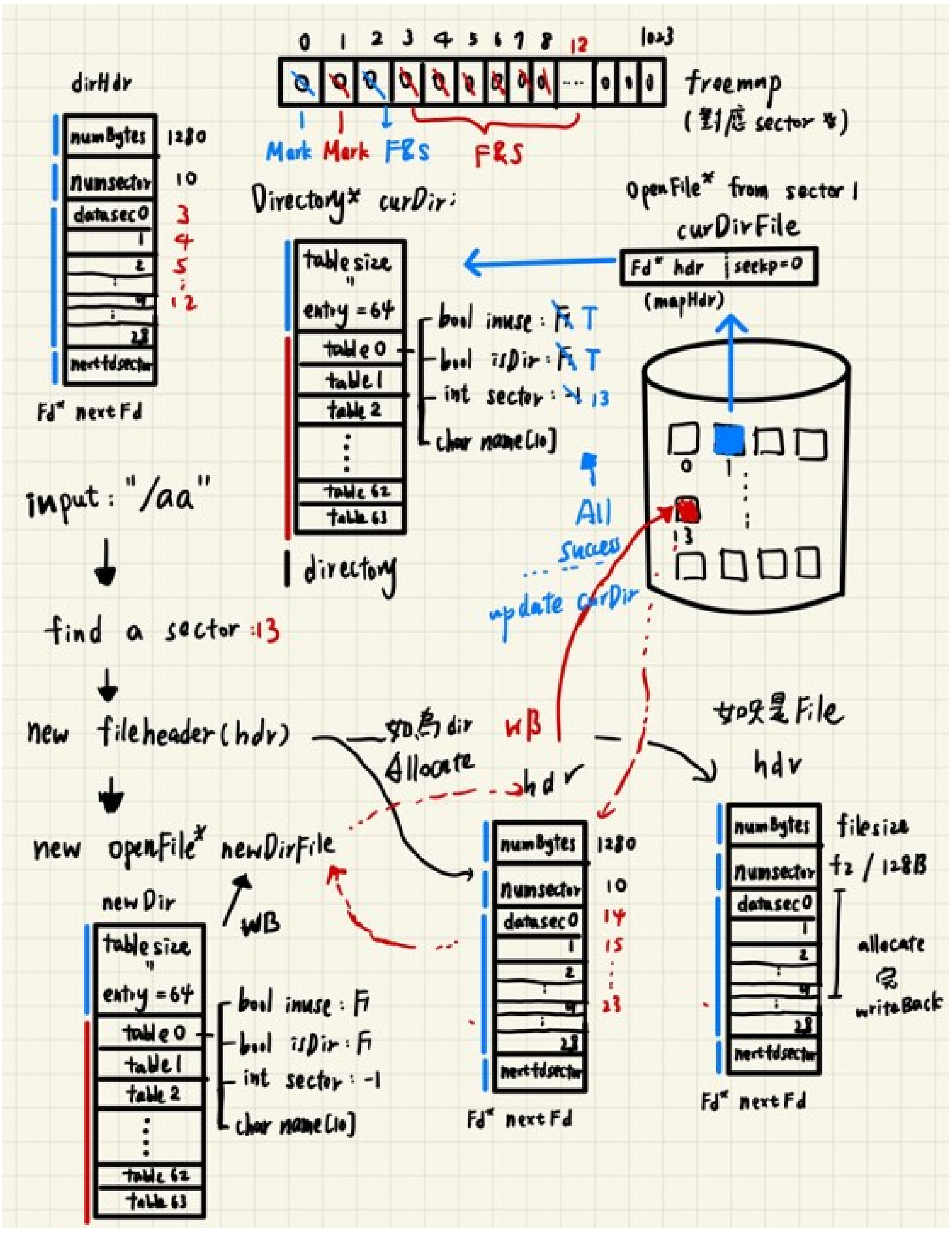
```
void SplitPath(const char *path, char dirs[NumDirEntries][MAX_PATH_LEN], int *count)
    char* token;
    char tempPath[MAX_PATH_LEN];

    strncpy(tempPath, path, MAX_PATH_LEN);
    tempPath[MAX_PATH_LEN - 1] = '\0';
    token = strtok(tempPath, "/");
    *count = 0;

    while(token != NULL && *count < NumDirEntries) {
        strncpy(dirs[*count], token, MAX_PATH_LEN);
        dirs[*count][MAX_PATH_LEN - 1] = '\0';
        (*count)++;
        token = strtok(NULL, "/");
    }
}
```

2. 假設目前要新建的是 `/ccc` ，我們需要檢查 `/aaa/bbb` 路徑是否已存在，若不存在則新增失敗

3. 檔案跟資料夾差別在，如果我們是要 Traverse 資料夾，則需要找到存在哪個 sector，讀出 DirectoryFile 再更新至 currentDirectory

```
bool FileSystem::Create(char *name, int initialSize, bool isDir) {
    Directory *currentDirectory;
    PersistentBitmap *freeMap;
    FileHeader *hdr;
    OpenFile *currentDirectoryFile;
    int sector;
    bool success = TRUE;


    char dirs[NumDirEntries][MAX_PATH_LEN];
    int dirCount;
    SplitPath(name, dirs, &dirCount);

    if (dirCount == 0) {
        return FALSE; // No directory name given
    }

    currentDirectoryFile = new OpenFile(DirectorySector);
    currentDirectory = new Directory(NumDirEntries);
    currentDirectory->FetchFrom(currentDirectoryFile);

    for(int i = 0; i < dirCount - 1; i++)
        char *dirName = dirs[i];

        sector = currentDirectory->Find(dirName);
        if(sector == -1) {
            success = FALSE;
            break;
        }

        delete currentDirectoryFile;
        delete currentDirectory;
        currentDirectoryFile = new OpenFile(sector);
        currentDirectory = new Directory(NumDirEntries);
        currentDirectory->FetchFrom(currentDirectoryFile);
    }


    if(!success) {
        delete currentDirectoryFile;
        delete currentDirectory;
        return success;
    }

    char* finalName = dirs[dirCount - 1];

    if(currentDirectory->Find(finalName) != -1) {
        // Final directory or file already exists
        delete currentDirectoryFile;
        delete currentDirectory;
        return FALSE;
    } else {
         // Create the final directory or file
        freeMap = new PersistentBitmap(freeMapFile, NumSectors);
        sector  = freeMap->FindAndSet();
        if(sector == -1) {

            success = FALSE;
        } else {
            hdr = new FileHeader;
            if(isDir) {
                if(!hdr->Allocate(freeMap, DirectoryFileSize)) {

                    success = FALSE;
                } else {
                    // Initialize the new directory
                    hdr->WriteBack(sector);
                    OpenFile *newDirFile = new OpenFile(sector);
                    Directory *newDirectory = new Directory(NumDirEntries);
                    newDirectory->WriteBack(newDirFile);

                    delete newDirFile;
                }
            } else {
                if(!hdr->Allocate(freeMap, initialSize)) {
                          success = FALSE;
                }
            }

            if(success) {
                hdr->WriteBack(sector);
                if(!currentDirectory->Add(finalName, sector, isDir)) {

                    success = FALSE;
                }
                currentDirectory->WriteBack(currentDirectoryFile);
                freeMap->WriteBack(freeMapFile);


            }

            delete hdr;
        }
        delete freeMap;
    }
    delete currentDirectoryFile;
    delete currentDirectory;
    return success;
}
```

情況 1: 如 `/aaa` 直接新增



情況 2: 如 `/aaa/bbb/ccc` 則檢查 `/aaa/bbb` 是否存在，若存在則新增 `/ccc` ，此時已成功實作，再來就仿造此處 Traverse 將 List 實作出來。

觀察 tag `l` 與 `lr` 的差別。

```
// in threads/main.cc
else if (strcmp(argv[i], "-l") == 0)
{
    // MP4 mod tag
    ASSERT(i + 1 < argc);
    listDirectoryName = argv[i + 1];
    dirListFlag = true;
    i++;
}
else if (strcmp(argv[i], "-lr") == 0)
{
    // MP4 mod tag
    // recursive list
    ASSERT(i + 1 < argc);
    listDirectoryName = argv[i + 1];
    dirListFlag = true;
    recursiveListFlag = true;
    i++;
}
```

可知 `l` 只需要列印當層 Directoy 的資料印出即可，而 `lr` 則需要繼續往下搜尋。

```
// in threads/main.cc
if (dirListFlag)
{
    if(recursiveListFlag) {
        kernel->fileSystem->RecursiveList(listDirectoryName);
    } else {
        kernel->fileSystem->List(listDirectoryName);
    }
}
```

我們先實作 `List -l`
先走到該層資料夾的位置(`FileSystem::List`)，後將該位置印出來(`Directory::List()`)

```
void FileSystem::List(char *listDirectoryName) {
    Directory *currentDirectory;
    OpenFile *currentDirectoryFile;
    int sector;

    char dirs[NumDirEntries][MAX_PATH_LEN];
    int dirCount;
    SplitPath(listDirectoryName, dirs, &dirCount);

    if (dirCount == 0 || strcmp(listDirectoryName, "/") == 0) {
        // List root directory
        currentDirectory = new Directory(NumDirEntries);
        currentDirectoryFile = new OpenFile(DirectorySector);
        currentDirectory->FetchFrom(currentDirectoryFile);
        currentDirectory->List();

        delete currentDirectoryFile;
        delete currentDirectory;
        return;
    } else {
        // Traverse to specified directory
        currentDirectoryFile = new OpenFile(DirectorySector);
        currentDirectory = new Directory(NumDirEntries);
        currentDirectory->FetchFrom(currentDirectoryFile);

        for (int i = 0; i < dirCount; i++) {
            char *dirName = dirs[i];
            sector = currentDirectory->Find(dirName);
            if (sector == -1) {
                printf("Directory not found: %s\n", dirName);
                delete currentDirectoryFile;
                delete currentDirectory;
                return;
            }

            delete currentDirectory;
            currentDirectoryFile = new OpenFile(sector);
            currentDirectory = new Directory(NumDirEntries);
            currentDirectory->FetchFrom(currentDirectoryFile);
        }
}

    // List the contents of the reached directory
    currentDirectory->List();

    delete currentDirectoryFile;
    delete currentDirectory;
}

void Directory::List()
{
for (int i = 0; i < tableSize; i++)
    if (table[i].inUse) {
        if(table[i].isDir) {
            printf("%s\n", table[i].name);
        } else {
            printf("%s\n", table[i].name);
        }
    }
}
```

再實作 `List recursive -lr`
先走到該層資料夾的位置(`FileSystem::RecursiveList`)，後將該位置印出來
(`Directory::RecursiveList()`)

```
void FileSystem::RecursiveList(char *listDirectoryName) {
    Directory *currentDirectory;
    OpenFile *currentDirectoryFile;
    int sector;

    char dirs[NumDirEntries][MAX_PATH_LEN];
    int dirCount;
    SplitPath(listDirectoryName, dirs, &dirCount);

    if (dirCount == 0 || strcmp(listDirectoryName, "/") == 0) {
        // List root directory
        currentDirectory = new Directory(NumDirEntries);
        currentDirectoryFile = new OpenFile(DirectorySector);
        currentDirectory->FetchFrom(currentDirectoryFile);
        currentDirectory->RecursiveList(0, currentDirectoryFile);

        delete currentDirectoryFile;
        delete currentDirectory;
        return;
    } else {
        // Traverse to specified directory
        currentDirectoryFile = new OpenFile(DirectorySector);
        currentDirectory = new Directory(NumDirEntries);
        currentDirectory->FetchFrom(currentDirectoryFile);

        for (int i = 0; i < dirCount; i++) {
            char *dirName = dirs[i];
            sector = currentDirectory->Find(dirName);
            if (sector == -1) {
                printf("Directory not found: %s\n", dirName);
                delete currentDirectoryFile;
                delete currentDirectory;
                return;
            }

            delete currentDirectory;
            currentDirectoryFile = new OpenFile(sector);
            currentDirectory = new Directory(NumDirEntries);
            currentDirectory->FetchFrom(currentDirectoryFile);
        }
}

    // List the contents of the reached directory
    currentDirectory->RecursiveList(0, currentDirectoryFile);

    delete currentDirectoryFile;
    delete currentDirectory;
}

void Directory::RecursiveList(int level, OpenFile *directoryFile) {
    Directory *directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);

    for(int i = 0; i < directory->tableSize; i++) {

        if(directory->table[i].inUse) {
            for(int j = 0; j < level; j++) {
                printf("    "); // Indentation for readability
            }
            if(directory->table[i].isDir) {
                printf("[D] %s\n", directory->table[i].name);
                OpenFile *subDirectoryFile = new OpenFile(directory->table[i].sector)
                RecursiveList(level + 1, subDirectoryFile);
                delete subDirectoryFile;
            } else {
                printf("[F] %s\n", directory->table[i].name);
            }
        }

    }
    delete directory;
}
```

Bonus Assignment

Bonus I: Enhance the NachOS to support even larger file size

- Extend the disk from 128KB to 64MB
- Support up to 64 MB single file

第二點因為我們採用 `linked index scheme` 所以單一檔案的大小已能夠支援 64 MB

而關於 Disk 的容量

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

Disk 目前的空間 = SectorSize * NumSectors = SectorSize * (SectorsPerTrack * NumTracks) = 128 * (32 * 32) = 128 KB

題目規定不能更改 Sector size，故我們從 `SectorsPerTrack` 及 `NumTracks` 下手。

將 `SectorsPerTrack` 改為 512，且 `NumTracks` 改為 1024。則新的配置

Disk 擴大後的空間 = SectorSize * NumSectors = SectorSize * (SectorsPerTrack * NumTracks) = 128 * (512 * 1024) = 64 MB

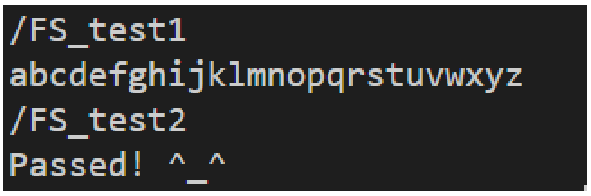
Bonus II: Multi-level header size

Bonus III. Recursive operations on directories

Result

FS_partII_a.sh

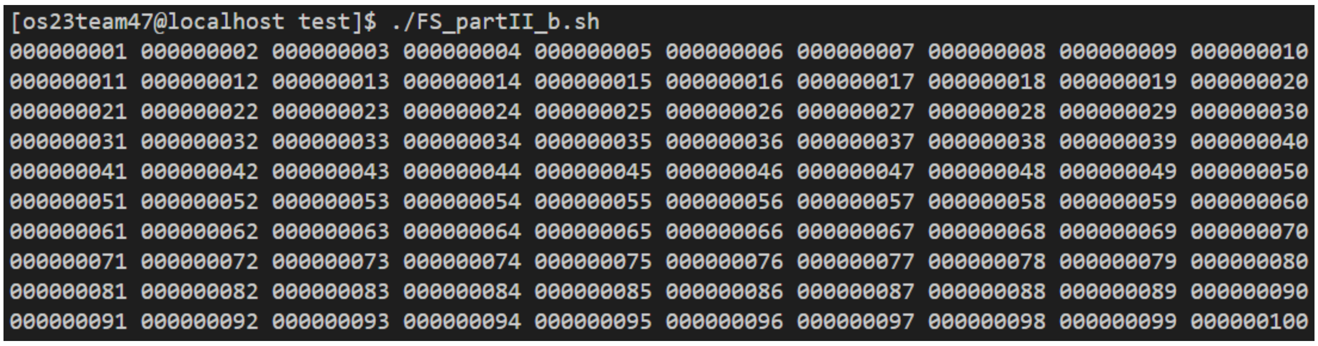
```
../build.linux/nachos -f
../build.linux/nachos -cp FS_test1 /FS_test1
../build.linux/nachos -e /FS_test1
../build.linux/nachos -p /file1
../build.linux/nachos -cp FS_test2 /FS_test2
../build.linux/nachos -e /FS_test2
```



```
/FS_test1
abcdefghijklmnopqrstuvwxyz
/FS_test2
Passed! ^_^
```

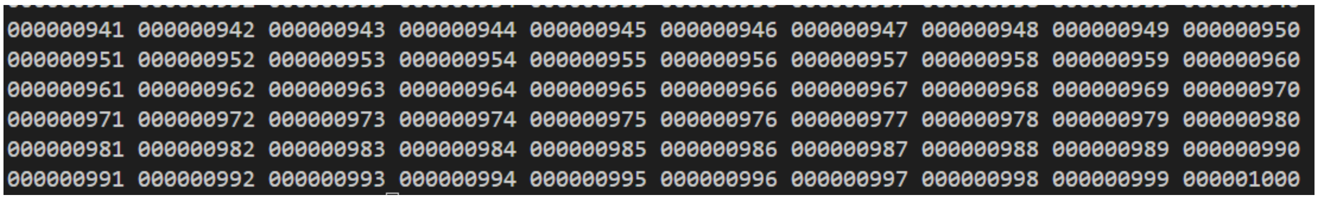
FS_partII_b.sh

```
../build.linux/nachos -f
../build.linux/nachos -cp num_1000.txt /1000
../build.linux/nachos -p /1000
```



```
[os23team47@localhost test]$ ./FS_partII_b.sh
000000001 000000002 000000003 000000004 000000005 000000006 000000007 000000008 000000009 000000010
000000011 000000012 000000013 000000014 000000015 000000016 000000017 000000018 000000019 000000020
000000021 000000022 000000023 000000024 000000025 000000026 000000027 000000028 000000029 000000030
000000031 000000032 000000033 000000034 000000035 000000036 000000037 000000038 000000039 000000040
000000041 000000042 000000043 000000044 000000045 000000046 000000047 000000048 000000049 000000050
000000051 000000052 000000053 000000054 000000055 000000056 000000057 000000058 000000059 000000060
000000061 000000062 000000063 000000064 000000065 000000066 000000067 000000068 000000069 000000070
000000071 000000072 000000073 000000074 000000075 000000076 000000077 000000078 000000079 000000080
000000081 000000082 000000083 000000084 000000085 000000086 000000087 000000088 000000089 000000090
000000091 000000092 000000093 000000094 000000095 000000096 000000097 000000098 000000099 000000100
```

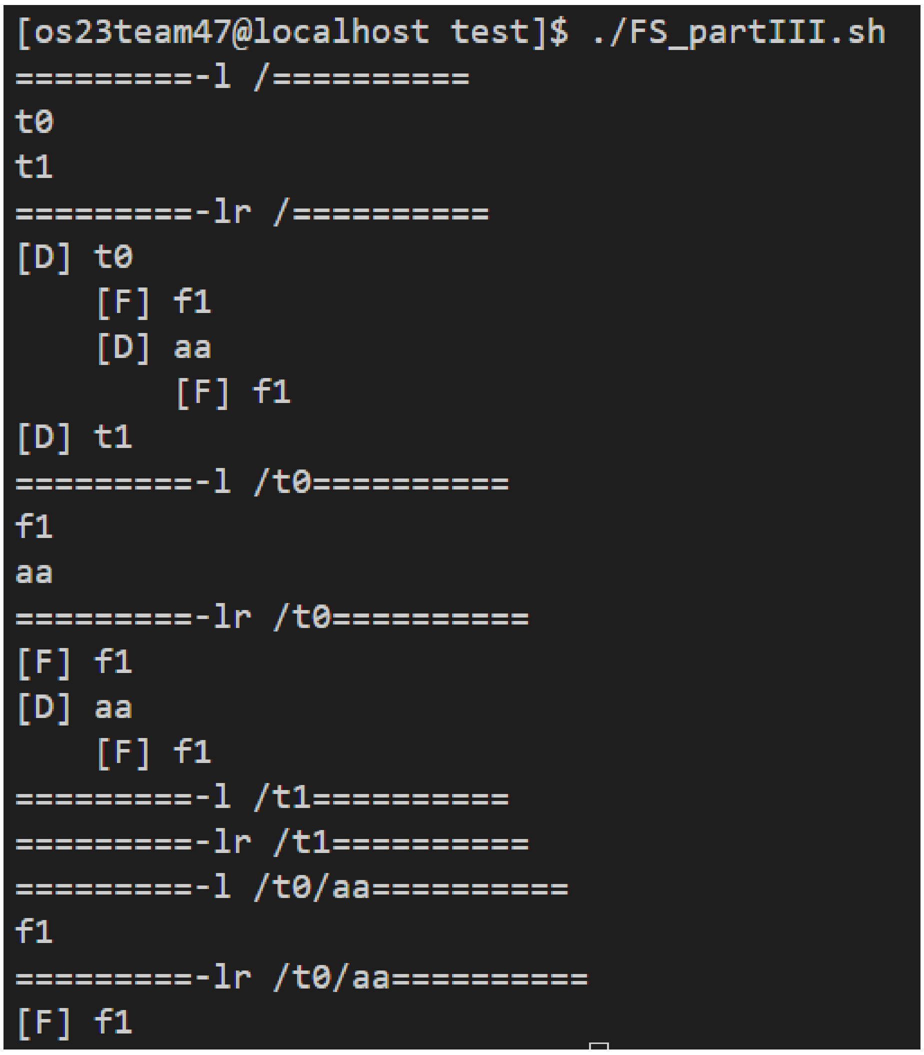
…



```
000000941 000000942 000000943 000000944 000000945 000000946 000000947 000000948 000000949 000000950
000000951 000000952 000000953 000000954 000000955 000000956 000000957 000000958 000000959 000000960
000000961 000000962 000000963 000000964 000000965 000000966 000000967 000000968 000000969 000000970
000000971 000000972 000000973 000000974 000000975 000000976 000000977 000000978 000000979 000000980
000000981 000000982 000000983 000000984 000000985 000000986 000000987 000000988 000000989 000000990
000000991 000000992 000000993 000000994 000000995 000000996 000000997 000000998 000000999 000001000
```

FS_partIII.sh

```
../build.linux/nachos -f
../build.linux/nachos -mkdir /t0
../build.linux/nachos -mkdir /t1
../build.linux/nachos -cp num_100.txt /t0/f1
../build.linux/nachos -mkdir /t0/aa
../build.linux/nachos -cp num_100.txt /t0/aa/f1
../build.linux/nachos -l /
../build.linux/nachos -lr /
../build.linux/nachos -l /t0
../build.linux/nachos -lr /t0
../build.linux/nachos -l /t1
../build.linux/nachos -lr /t1
../build.linux/nachos -l /t0/aa
../build.linux/nachos -lr /t0/aa
```



```
[os23team47@localhost test]$ ./FS_partIII.sh
==========-l /==========
t0
t1
==========-lr /==========
[D] t0
    [F] f1
    [D] aa
        [F] f1
[D] t1
==========-l /t0==========
f1
aa
==========-lr /t0==========
[F] f1
[D] aa
    [F] f1
==========-l /t1==========
==========-lr /t1==========
==========-l /t0/aa==========
f1
==========-lr /t0/aa==========
[F] f1
```