Part I NachOS Problems

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

NachOS用 PersistentBitmap *freeMap 來記錄 free block space 。它會用一個長度為NumSectors的陣列來記錄每個block的資訊，0為尚未被使用，１則是被使用。而當需要尋找free space的時候，會去call FindAndSet()，將未被使用的block設為１，然後回傳sector的索引值。

FreeMap的資訊儲存在`#define FreeMapSector 0`

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

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

DiskSize = 4 + (32*32) * 128 = 131076B

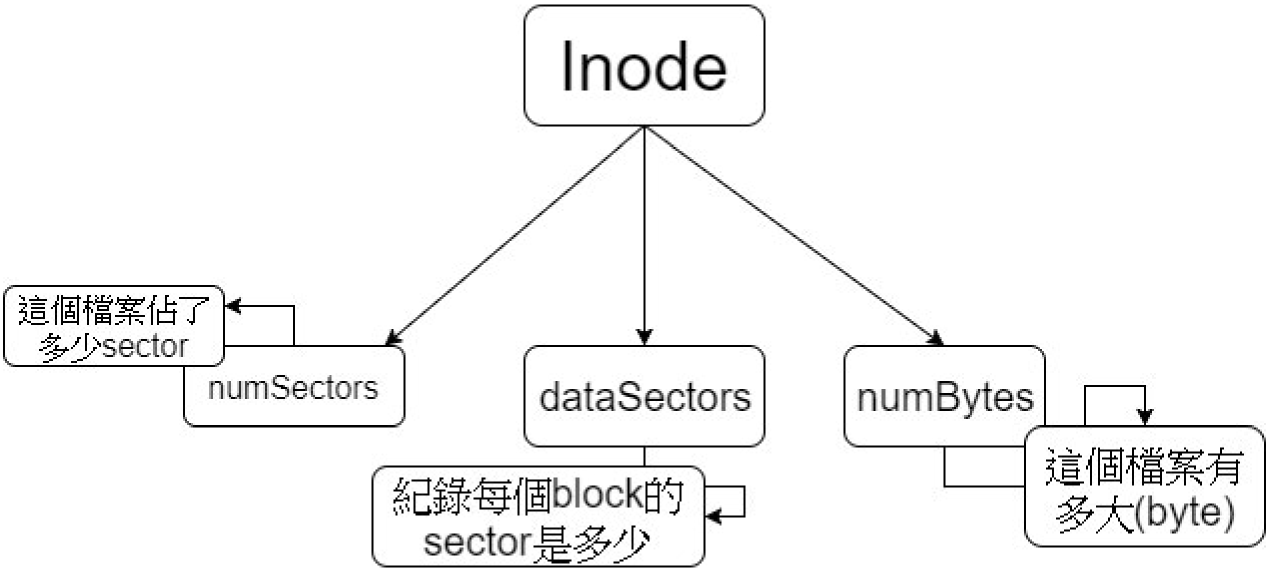
Explain how does the NachOS FS manage the directory data structure? Where is this information stored on the raw disk (which sector)?

當建立Directory，會有資料結構DirectoryEntry的table來記錄information。 裡面的inUse用來查看是否entry被使用，sector用來尋找fileheader在disk的位置。 如果需要一個資料時，會去table中尋找有沒有一樣的filename，然後將它的sector number回傳。

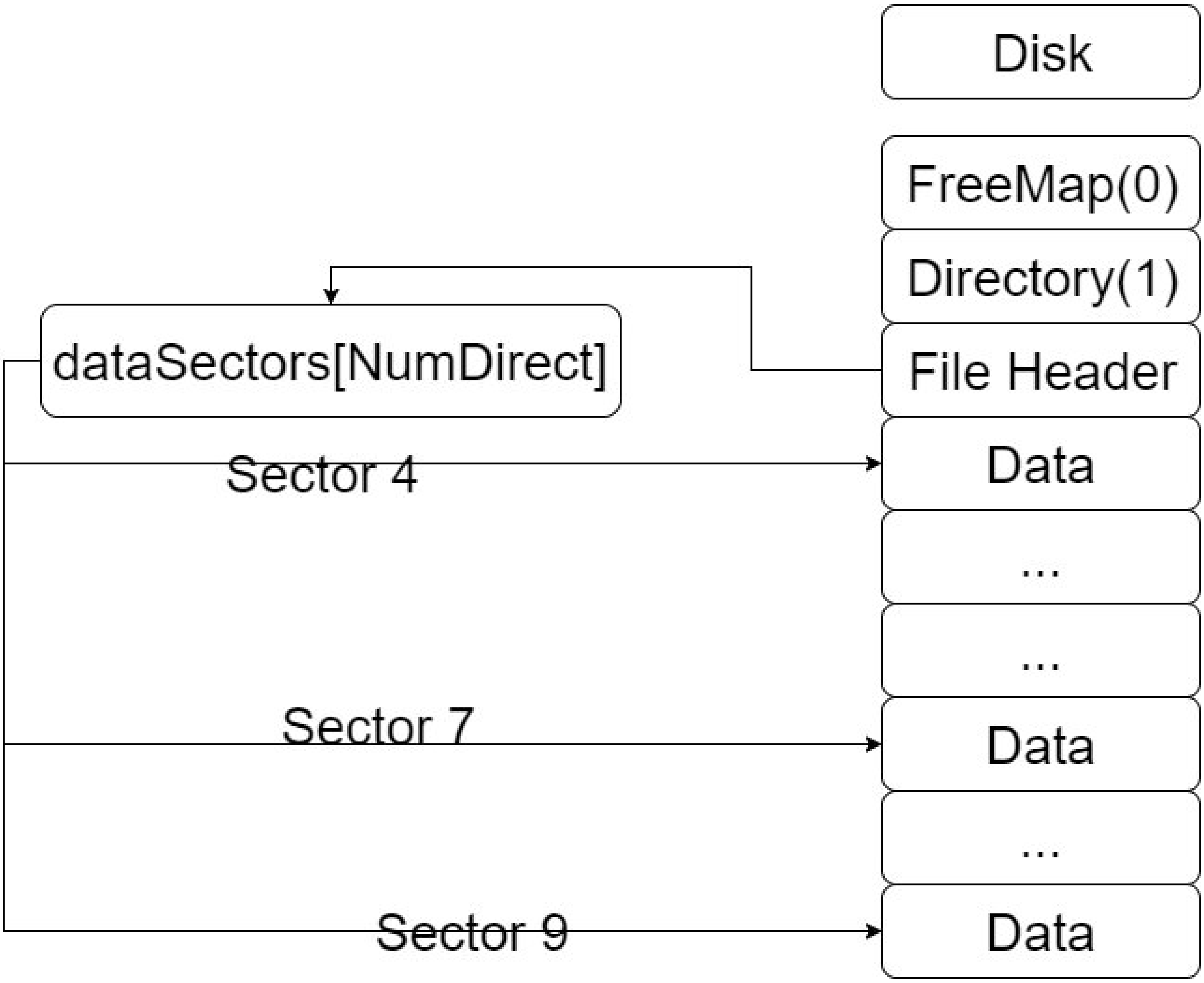
Directory information記錄在 `#define DirectorySector 1`

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

Inode Structure



Disk Allocation Scheme



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

因為 **NumDirect=30**，而 **MaxFileSize** = (**NumDirect** * **SectorSize**) = 30 * 128B = 3840B，大約是4KB。

PartII

(1) Combine your MP1 file system call interface with NachOS FS

- int Create(char *name, int size)

在這裡我們多增加了size變數傳入SysCreate()，其餘皆一樣。

```
case SC_Create:
    val = kernel->machine->ReadRegister(4);
    size = kernel->machine->ReadRegister(5);
    {
    char *filename = &(kernel->machine->mainMemory[val]);
    //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;
```

之後的皆與MP1實作方法一樣 ksyscall -> interrupt -> kernel -> filesys 而在filesys的Create()我們直接利用老師所提供的function。

```
int SysCreate(char *filename,int initialSize)
{
    return kernel->interrupt->CreateFile(filename,initialSize);
}
```

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

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

```
bool FileSystem::Create(char *name, int initialSize)
```

- OpenFileId Open(char *name)
  與MP1相同
  ksyscall -> interrupt -> kernel -> filesys
  Open()一樣使用老師提供的。

```
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;
```

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

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

```
int Kernel::Open(char *filename)
{
    int fileID = (int)fileSystem->Open(filename);
    if(fileID!=0) return fileID;
    return -1;
}
```

```
OpenFile *FileSystem::Open(char *name)
```

- int Read(char *buf, int size, OpenFileId id)
- int Write(char *buf, int size, OpenFileId id)
- int Close(OpenFileId id);
  exception -> ksyscall -> interrupt -> kernel -> filesys

```
case SC_Read:
    val = kernel->machine->ReadRegister(4);
    {
    buffer = &(kernel->machine->mainMemory[val]);
    size = kernel->machine->ReadRegister(5);
    id =  kernel->machine->ReadRegister(6);
    status = SysRead(buffer,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;
case SC_Write:
    val = kernel->machine->ReadRegister(4);
    {
    buffer = &(kernel->machine->mainMemory[val]);
    size = kernel->machine->ReadRegister(5);
    id =  kernel->machine->ReadRegister(6);
    status = SysWrite(buffer,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;
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;
```

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

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

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

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

```
int Kernel::Write(char *buffer, int size, int id)
{
    return fileSystem->Write(buffer, size, id);
}

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

int Kernel::Close(int id)
{
    return fileSystem->Close(id);
}
```

透過Read(),Write()會去call ReadAt與WriteAt去實作更底層的Read,Write。 而Close則是把指標給指向NULL，然後回傳1。

```
int FileSystem::Read(char *buffer, int size, int id)
{
    return opfile->Read(buffer, size);
}

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

int FileSystem::Close(int id)
{
    opfile = NULL;
    return 1;
}
```

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

我們主要修改的地方為 filehdr.h/.cc

原本的`MaxFileSize=30*128B`，於是在不改變SectorSize的情況下，我們為了要能support 32KB大小的File，我們將Sectors每32個單位綁在一起，然後再將它串起來，如此一來便能讀寫更大的File。

```
#define MaxListNum 29
#define SectorsPerList 32
#define MaxFileSize MaxListNum * SectorsPerList * SectorSize
```

接著，我們在.h新增一個numLists紀錄有幾個list串起來。

```
private:
    int numLists;
```

一開始假如沒有足夠的空間可以Allocate便return FALSE，而不再繼續執行下面的動作。 假如有空間可以擺放的話，便利用Lists的概念看看有多少Lists，還有最後一個Lists有多少sectors，然後以sector為單位寫回disk。

```
bool
FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
{
    char empty[128] = {0};
    numSectors  = divRoundUp(fileSize, SectorSize);
    numBytes = fileSize;
    numLists = divRoundUp(numSectors, SectorsPerList);

    if (freeMap->NumClear() < numSectors)
        return FALSE;       // not enough space

    int SectorsRead = 0;
    for (int i = 0; i < numLists; i++, SectorsRead += SectorsPerList) {
        dataSectorLists[i] = freeMap->FindAndSet();
        ASSERT(dataSectorLists[i] >= 0);

        int lastSector;
        if (SectorsRead + SectorsPerList > NumSectors)
            lastSector = NumSectors;
        else
            lastSector = SectorsRead + SectorsPerList;

        int *buffer = new int[SectorsPerList];
        memset(buffer, 0, sizeof(int)*SectorsPerList);
        for (int j = 0; j < lastSector - SectorsRead; j++) {
            buffer[j] = freeMap->FindAndSet();
            kernel->synchDisk->WriteSector(buffer[j], empty);
            ASSERT(buffer[j] >= 0);
        }
        kernel->synchDisk->WriteSector(dataSectorLists[i], (char*) buffer);
        delete [] buffer;
    }
    return true;
}
```

用for迴圈尋找file所涵蓋住的Lists，接著再利用for迴圈將每個Lists的sector(以及最後一個Lists剩下的sector)以sector為單位做Clear()的動作。

```
void
FileHeader::Deallocate(PersistentBitmap *freeMap)
{
    int SectorsRead = 0;
    for (int i = 0; i < numLists; i++, SectorsRead += SectorsPerList) {
        int lastSector;
        if (SectorsRead + SectorsPerList > numSectors)
            lastSector = numSectors;
        else
            lastSector = SectorsRead + SectorsPerList;

        int* buffer = new int[SectorsPerList];
        kernel->synchDisk->ReadSector(dataSectorLists[i], (char*) buffer);
        for (int j = 0; j < lastSector; j++) {
            ASSERT(freeMap->Test((int) buffer[j]));
            freeMap->Clear((int)buffer[j]);
        }
        delete [] buffer;
    }
}

```

我們利用sectorID去尋找資料放在第幾個sector，之後再將sectorID / SectorsPerList與sectorID % SectorsPerList。目的是要找出資料位在第幾個list的第幾個sector。 之後，將位在某個list裡的東西撈到buffer上，再找尋buffer[idInList]，這樣就能找到我們所要的資料。

```
int
FileHeader::ByteToSector(int offset)
{
    //return(dataSectors[offset / SectorSize]);
     int sectorID = offset / SectorSize;
     int listID = sectorID / SectorsPerList, idInList = sectorID % SectorsPerList;
     int *buffer = new int[SectorsPerList];
     kernel->synchDisk->ReadSector(dataSectorLists[listID], (char *) buffer);
     // get the SectorNum
     int retVal = buffer[idInList];
     delete [] buffer;
     return retVal;
}

```

Part III Implementation

How to support subdirectory

1. Entering FileSysyem API

我們使用以下這個function來實作subdirectory。當NachOS收到mkdir指令時會將mkdirflag設為真，程式本身依靠這個flag來判斷該創造一個檔案(FileSystem::Create)或是資料夾(FileSystem::CreateDir)

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

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

    if (directory->Find(name) != -1)
        success = FALSE;
    else
    {
        freeMap = new PersistentBitmap(freeMapFile, NumSectors);
        sector = freeMap->FindAndSet();
        if (sector == -1)
            success = FALSE;
        else if(!directory->Add(name, sector, 'D'))
            success = FALSE;
        else
        {
            hdr = new FileHeader;
            if (!hdr->Allocate(freeMap, DirectoryFileSize))
                success = FALSE;
            else
            {
                success = TRUE;
                hdr->WriteBack(sector);
                directory->WriteBack(directoryFile);
                freeMap->WriteBack(freeMapFile);
            }
        }
    }
    delete freeMap;
    delete hdr;
    delete directory;
    return success;
}
```

我們在DirectoryEntry新增了type來判斷這個檔案真的是檔案或者是資料夾

2. Parsing Path

原本的Directory.* 不支援subdirectory的查找，我們修改了下列function ***1. int Directory::Find(char name)***

```
 1  int Directory::Find(char *name)
 2  {
 3      name++;
 4      char localName[256] = {0};
 5      char localID = 0;
 6      bool findNext = false;
 7      while (name[0] != '\0') {
 8          if (name[0] == '/') {
 9              findNext = true;
10              break;
11          }
12          localName[localID++] = name[0];
13          name++;
14      }
15      int i = FindIndex(localName);
16      if (i != -1) {
17          if (findNext) {
18              OpenFile* nextDirectory = new OpenFile(table[i].sector);
19              Directory* nextDir = new Directory(NumDirEntries);
20              nextDir->FetchFrom(nextDirectory);
21              int result = nextDir->Find(name);
22              delete nextDirectory;
23              delete nextDir;
24              return result;
25          } else {
26              return table[i].sector;
27          }
28      } else {
29          return -1;
30      }
31  }
```

1. 一開始把name加一，因為我們不想把slash放進檔案名稱裡儲存
2. 程式7~14行將Path做parsing來判斷需不需要subdirectory visiting。一個字元一個字元的讀直到讀到slash或是結尾，如果有slash代表這是subdirectory，將findNext拉起來準備做recursion進入子資料夾內。
3. localName紀錄直到slash前的path，也就是第一個需要進入的資料夾。
4. 如果有子資料夾需要進入的話，就打開它並call它的Find(name)來做recursion
5. 如果不需要就直接return檔案的sector

2. bool Directory::Add(char name, int newSector, char inType)

```
bool Directory::Add(char *name, int newSector, char inType)
{
    if (Find(name) != -1)
        return FALSE;

    char Path[256] = {0};
    char File[9] = {0};
    int len = strlen(name), slash, tmpID = 0;
    for (int i = len - 1; i >= 0; i--) {
        if (name[i] == '/') {
            slash = i;
            break;
        }
    }

    for (int i = slash+1; i < len; i++) {
        File[tmpID++] = name[i];
    }
    for (int i = 0; i < slash; i++) {
        Path[i] = name[i];
    }

    if (Path[0] != 0) {
        int sector = Find(Path);
        OpenFile* nextDirectory = new OpenFile(sector);
        Directory* nextDir = new Directory(NumDirEntries);
        nextDir->FetchFrom(nextDirectory);

        for (int i = 0; i < tableSize; i++) {
            if (!nextDir->table[i].inUse) {
                nextDir->table[i].inUse = true;
                strncpy(nextDir->table[i].name, File, FileNameMaxLen);
                nextDir->table[i].sector = newSector;
                nextDir->table[i].type = inType;
                nextDir->WriteBack(nextDirectory);

                delete nextDirectory;
                delete nextDir;
                return true;
            }
        }
    } else {
        for (int i = 0; i < tableSize; i++) {
            if (!table[i].inUse) {
                table[i].inUse = true;
                strncpy(table[i].name, File, FileNameMaxLen);
                table[i].sector = newSector;
                table[i].type = inType;
                return true;
            }
        }
    }
    return false;   // no space.  Fix when we have extensible files.
}
```

1. 和Find一樣需要Parsing，不同的是這次是從後面Parse回來直到找到第一個slash，slash後面的Name就是檔案名稱，前面的就是Path。如果沒有subdirectory，Path就會是空的。
2. 如果Path[] != 0，就call Find()來拿到該寫進的正確directory，並且寫入disk內。
3. 如果Path == 0，代表自己就是該寫進去的正確資料夾，直接寫到裡面，因為會return回去FileSystem寫入disk，所以這邊不需要寫入。

3. bool Directory::Remove(char name)

```
bool Directory::Remove(char *name)
{
    if (Find(name) == -1)
        return false;

    char Path[256] = {0};
    char File[9] = {0};
    int len = strlen(name), tmpID = 0, slash;
    for (int i = len-1; i >= 0; i--) {
        if (name[i] == '/') {
            slash = i;
            break;
        }
    }

    for (int i = slash + 1; i < len; i++) {
        File[tmpID++] = name[i];
    }
    for (int i = 0; i < slash; i++) {
        Path[i] = name[i];
    }

    if (Path[0] != 0) {
        int sector = Find(Path);
        OpenFile* nextDirectory = new OpenFile(sector);
        Directory* nextDir = new Directory(NumDirEntries);
        nextDir->FetchFrom(nextDirectory);

        int id = nextDir->FindIndex(File);
        if (id == -1)
            return false;

        nextDir->table[id].inUse = false;
        nextDir->WriteBack(nextDirectory);
        delete nextDirectory;
        delete nextDir;
        return true;
    } else {
        int id = this->FindIndex(name);
        if (id == -1)
            return false;
        this->table[id].inUse = false;
        return true;
    }
}
```

大抵上和Add()很像，只是差別在於一個是Remove一個是Add

Support up to 64 entries in a directory

蠻簡單的，把NumdirEntries從10改成64即可

Support recursively list the file/directory in a directory

```
void Directory::RecursiveList(int depth)
{
    for (int i = 0; i < tableSize; i++) {
        if (table[i].inUse) {
            for (int j = 0; j < depth*8; j++)
                putchar(' ');
            printf("[Entry No.%d]: %s %c\n", i, table[i].name, table[i].type);
            if (table[i].type == 'D') {
                OpenFile* nextDirectory = new OpenFile(table[i].sector);
                Directory* nextDir = new Directory(NumDirEntries);
                nextDir->FetchFrom(nextDirectory);

                nextDir->RecursiveList(depth+1);
                delete nextDirectory;
                delete nextDir;
            }
        }
    }
}
```

呼叫上面的RecursiveList()，印出來時順便判斷是Directory還是File，如果是D就進去traverse一遍把東西都印出來。

Bonus II：Remove a file or recursively remove the directory

我們的Recursive Remove走和正常remove一樣的flow，Directory::remove()會自己parse Path並刪除。

Contribution

- 陳麒懋：Part I、Part II、Report Part I、II
- 鍾昀誼：Part II、PartIII、Report PartIII