MP4: File System

Group 51
110000132 詹振暘 110021121陶威綸

詹振暘：code, report
陶威倫：code, report

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)?

NachOS使用包裝Bitmap的物件PersistentBitmap紀錄Sector的資訊。FileSystem呼叫freeMap->FindAndSet()尋找bitmap中為空的元素，藉此找到free block space，並且回傳該Sector的位址。資料結構部分，NachOS使用PersistentBitmap管理free block space。PersistentBitmap跟Bitmap相似，差別在他加上了兩個˙function FetchFrom, WriteBack，這兩個function會將Bitmap寫入並讀出DISK，藉此可達到Persistent的效果。

```
Kernel::Kernel(int argc, char **argv)
if (strcmp(argv[i], "-f") == 0) {
            formatFlag = TRUE;
}
Kernel::Initialize()
{
      fileSystem = new FileSystem(formatFlag);
}
```

```
FileSystem::FileSystem(bool format)
{
   DEBUG(dbgFile, "Initializing the file system.");
   if (format)
   {
       PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);
   }
}
```

每當使用者需要空間，PersistentBitmap會呼叫FindAndSet，迭代所有bit找出可用的部分，並且設為標記為使用中。Test負責確認該空間是否可用，Mark則會將該空間標記為使用中。

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

   if (map[which / BitsInWord] & (1 << (which % BitsInWord)))
```

```
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}
void Bitmap::Mark(int which)
{
    ASSERT(which >= 0 && which < numBits);

    map[which / BitsInWord] |= 1 << (which % BitsInWord);

    ASSERT(Test(which));
}
```

在 FileSystem::Create、FileSystem::Remove 和 FileSystem::Print 中，當需要用到 freeMap 時，會先建立一個新的 PersistentBitmap 物件，並使用全域變數 freeMapFile 對其進行初始化，使其內容與 freeMapFile 一致。freeMapFile 隨後會幫助尋找空閒空間(free space)。在操作期間(如 Create、Remove 或 Print)，會直接使用這個新的 freeMap 物件。若操作成功並更新了 freeMap，則將修改後的內容寫回 freeMapFile否則丟棄本次操作的變更。

在 Create 的情況下，會使用 freeMap->FindAndSet()找到放置檔頭(header)的位置並更新 freeMap，接著透過 fileHeader->Allocate(PersistentBitmap *freeMap, int fileSize) 找到放置資料(data)的位置並更新 freeMap。

在 Remove 的情況下，使用 FileHeader::Deallocate(PersistentBitmap *freeMap, int fileSize) 釋放 data 區域的空間，然後再釋放 fileHeader 的空間。

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

```
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);



disk.cc
```

```
const int MagicNumber = 0x456789ab;

const int MagicSize = sizeof(int);

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

NachOS 的最大磁碟大小大約是 128 KB。因為它的設計是每個磁碟有 32 個磁軌(NumTracks)，每個磁軌又有 32 個區段，每個區段的大小是 128 字節。所以總區段數就是 32 * 32 = 1024 個。再加上disk.cc中的公式，DiskSize = (MagicSize + (NumSectors * SectorSize))，其中 MagicSize=sizeof(int) 是 4 字節。把這些數字代入公式後，就能算出來磁碟的大小是 4 + (1024 * 128) = 131072 字節，大約就是 128 KB 。

因為 NachOS 的設計本身把磁軌數、區段數和區段大小都固定了，所以磁碟大小也會被限制在這個範圍內。

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

當建立 FileSystem 且不需要格式化時，直接打開位於DISK sector 1 的檔案，並將其載入記憶體中；如果需要格式化，則需初始化 root directory，設定底下的每個 DirectoryEntry，並將其 inUse 欄位設為 0。接著，分配空間並透過 Mark() 標記已使用的空間。DirectoryFileSize 的計算方式為 sizeof(DirectoryEntry) * NumDirEntries。

```
Directory::Directory(int size){

   table = new DirectoryEntry[size];

   memset(table, 0, sizeof(DirectoryEntry) * size);

   tableSize = size;

   for (int i = 0; i < tableSize; i++)

       table[i].inUse = FALSE;

}
```

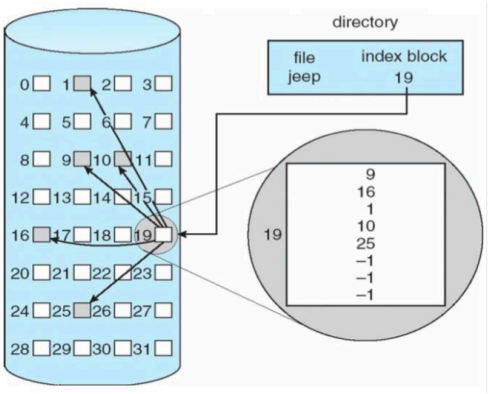
4. What information is stored in an inode? Use a figure to illustrate the disk allocation scheme of the current implementation.

NachOS在註解中說明了它將fileHeader作為iinode。inode中儲存了numBytes（檔案有幾個bytes），numSectors（這個檔案佔用了多少Sector）以及dataSector（儲存檔案位址的index）。

這個做法為講義中的direct index allocation。

```
// The following class defines the Nachos "file header" (in UNIX terms,

// the "i-node"), describing where on disk to find all of the data in the file.
```

```
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
```



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

目前為Direct index allocation，所以file header只能存在一個sector，所以能儲存的index有限。程式中，第一行扣掉兩個sizeof(int)分別為numBytes以及numSectors，只兩個變數會寫回DISK，所以剩下的空間為(128-2*4)可以紀錄 (128 - 2*4)/4 = 30 個SectorId，總共大小為30*128約為4KB。

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

#define MaxFileSize (NumDirect * SectorSize)
```

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:

第一個小題實作跟MP1非常相似，需要動到exception.cc, ksyscall.h以及filesys.cc, filesys.h。當SysCall呼叫時，程式透過syscall.h定義輸入，用exception.cc中的register會取傳入參數，並且呼叫ksyscall介接filesystem中的函數。filesystem中原本沒有Read以及Write，故需要額外撰寫程式。

exception.cc的程式很長，這邊用Read來做解釋，其餘概念類似。

因為Read有三個input（分別是buf, size, id）所以register4是buf，5跟6則對應到size和id。我們將三個參數直接傳遞到ksyscall的SysRead(buf, size, id)再把回反值寫回register2，就可以介接回原本呼叫的function。

而其他的函數，Create有兩個傳入參數，Close有一個，Write則跟Read類似。需要依此調整ReadRegister的數量。

```
case SC_Read:

            {

                val = kernel->machine->ReadRegister(4);

                char *buf = &(kernel->machine->mainMemory[val]);

                status = SysRead(

                    buf,

                    kernel->machine->ReadRegister(5),

                    kernel->machine->ReadRegister(6)

                );

                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);
```

ksyscall則是作為介接的橋樑，呼叫filesystem實作。

```
ksyscall.h

int SysCreate(char *filename, int filesize){

   return kernel->fileSystem->Create(filename, filesize);

}

OpenFileId SysOpen(char *filename){

   OpenFile* file = kernel->fileSystem->Open(filename);

   if (file) return 1;
```

```
    return 0;

}

int SysRead(char *buf, int size, OpenFileId id){

    return kernel->fileSystem->Read(buf, size, id);

}

int SysWrite(char *buf, int size, OpenFileId id){

    return kernel->fileSystem->Write(buf, size, id);

}

int SysClose(OpenFileId id){

    return kernel->fileSystem->Close(id);

}
```

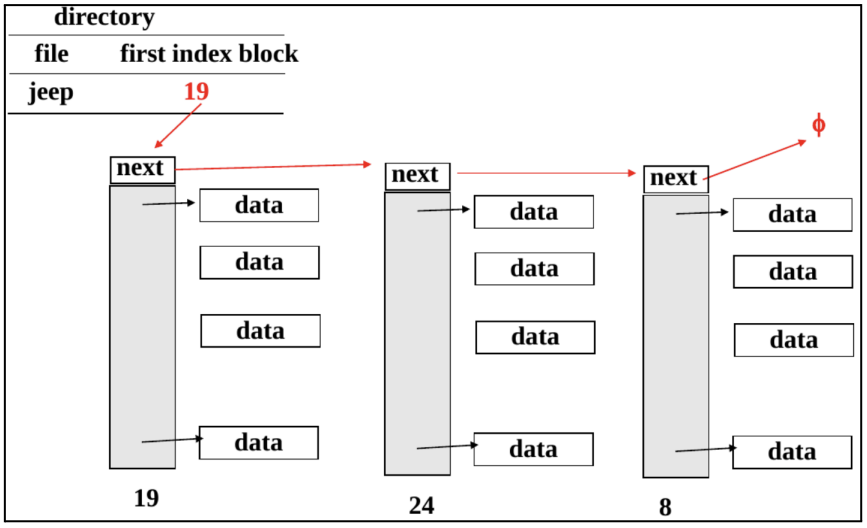
而filesystem的部分，因為一次只會有一個檔案被開啟，我們使用curOpen，curOpenName來紀錄被開啟的檔案資訊。所以在Open時會更新curOpen, curOpenName，並且在Close時會release掉。而Read及Write的實作則非常直覺，直接呼叫被開啟的file，執行read及write即可。

```
OpenFile * FileSystem::Open(char *name)
{
    directory->FetchFrom(directoryFile);
    sector = directory->Find(name);
    if (sector >= 0)
        openFile = new OpenFile(sector); // name was found in directory

    curOpen = openFile;
    curOpenName = name;
    return openFile; // return NULL if not found
}

int FileSystem::Read(char *buf, int size, OpenFileId id){
    return curOpen->Read(buf, size);
}
int FileSystem::Write(char *buf, int size, OpenFileId id){
    return curOpen->Write(buf, size);
}

int FileSystem::Close(OpenFileId id){
    curOpen = NULL;
```

```
    curOpenName = NULL;
    return 1;
}
```

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

第二題的實作較為複雜，現在的檔案為direct index allocation所以一個header無法儲存完所有(32KB)的資訊。我們選用用LinkedList實作新的allocation。這個做法需要改變原本header儲存的內容，寫入時需要用不同種方式Allocate，Read、Write問題也會要考慮在哪個black之中，所以會更動到filesys, filehdr, openfile.cc, .h六個檔案。



講義中Linked scheme圖示

先從Header部分說明，新的Header需要紀錄下個Header的位址，所以我們多放了nextHdr, nextHdrSector，前者紀錄該hdr在DISK中的位置，後者則使用資料結構方便紀錄使用。

```
class FileHeader
{
public:
    FileHeader(); // dummy constructor to keep valgrind happy
    ~FileHeader();
    bool Allocate(PersistentBitmap *bitMap, int fileSize); // Initialize a file
    void Deallocate(PersistentBitmap *bitMap);             // De-allocate this
    void FetchFrom(int sectorNumber); // Initialize file header from disk
    void WriteBack(int sectorNumber); // Write modifications to file header
    int ByteToSector(int offset); // Convert a byte offset into the file
    int FileLength(); // Return the length of the file
    int TotalFileLength();
private:
    int nextHdrSector;
    FileHeader* nextHdr;
```

```
    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
};
```

Allocate的部分，我們用recursion的方式實作分配，程式會不停地分配MaxFileSize的空間給檔案，在分配完後判斷是否還有需要bytes需要存取。如果有的話會新增下一個Header並第回呼叫，沒有則分配完成可以回傳True。如果FindAndSet發現沒有記憶體分給dataSector或是分給新的Header則會回傳False。

```
bool FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
{
    numBytes = fileSize <= MaxFileSize ? fileSize:MaxFileSize;
    DEBUG(dbgFile, "Allocate " << numBytes << " bytes " << fileSize << " in
total.");
    numSectors = divRoundUp(numBytes, SectorSize);

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

    // enough space
    for (int i = 0; i < numSectors; i++)
    {
        dataSectors[i] = freeMap->FindAndSet();
        ASSERT(dataSectors[i] >= 0);
    }

    // finish block allocate
    fileSize -= numBytes;
    if (fileSize <= 0){
        nextHdrSector = -1;
        nextHdr = NULL;
        DEBUG(dbgFile, "Allocate success.");
        return TRUE;
    }

    if ((nextHdrSector = freeMap->FindAndSet()) == -1){
        return FALSE;
    }else{
        nextHdr = new FileHeader();
        return nextHdr->Allocate(freeMap, fileSize);
    }
}
```

Deallocater就是把前面遞迴Allocate的bit收回，所以也是遞迴呼叫Clear收回所有空間。

```
void FileHeader::Deallocate(PersistentBitmap *freeMap)
```

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

WriteBack跟FetchFrom是類似的程式，一個寫入一個讀出。我們儲存資料的方式為：先numSectors再存numBytes跟nextHdrSector，最後才是儲存dataSector內的所有資訊。所以一個sector128bytes，有4個分給numSector, numBytes, nextHdrSector，剩下則29格整數空位則可以儲存dataSector的資訊。
而read write差別就是讀取跟寫入而已，基本上一樣。

```
void FileHeader::WriteBack(int sector){
    char buf[SectorSize];
    int offset = 0;
    memcpy(buf + offset, &numSectors, sizeof(int));
    offset += sizeof(int);
    memcpy(buf + offset, &numBytes, sizeof(int));
    offset += sizeof(int);
    memcpy(buf + offset, &nextHdrSector, sizeof(int));
    for (int i = 0;i < numSectors;i++){
        offset += sizeof(int);
        memcpy(buf + offset, &dataSectors[i], sizeof(int));
    }
    kernel->synchDisk->WriteSector(sector, buf);
    if (nextHdr) nextHdr->WriteBack(nextHdrSector);
}
```

```
void FileHeader::FetchFrom(int sector){
    char buf[SectorSize];
    int offset = 0;
    kernel->synchDisk->ReadSector(sector, buf);
    memcpy(&numSectors, buf, sizeof(int));
    offset += sizeof(int);
    memcpy(&numBytes, buf + offset, sizeof(int));
    offset += sizeof(int);
    memcpy(&nextHdrSector, buf + offset, sizeof(int));
    for (int i = 0;i < numSectors;i++){
        offset += sizeof(int);
        memcpy(&dataSectors[i], buf + offset, sizeof(int));
    }

    if (nextHdrSector == -1){
        nextHdr = NULL;
```

```
        return;
    }
    nextHdr = new FileHeader();
    nextHdr->FetchFrom(nextHdrSector);
}
```

當需要尋找某個位元組 (byte) 的資料位置時，可透過 ByteToSector 方法判斷該資料所在的磁區 (Sector)。我們採用遞迴方式實作：如果偏移量 (offset) 超過 NumDirect（即檔案可直接儲存的磁區數，通常為 29 個），則遞迴查找下一層的 Header，以定位正確的磁區位置。

```
int FileHeader::ByteToSector(int offset)
{
    int id = offset / SectorSize;
    if (id < NumDirect)
         return (dataSectors[id]);

    return nextHdr->ByteToSector(offset - MaxFileSize);

}
```

最後則是新增一個TotalFileLength回傳該Header後面連結檔案儲存的bytes總數

```
int FileHeader::TotalFileLength()
{
    if (!nextHdr){
        return numBytes;
    }
    return numBytes + nextHdr->TotalFileLength();
}
```

完成這些實作後，用於儲存檔案資料的檔案結構便已完成。接下來，只需在檔案系統（filesystem）和開啟檔案（openfile）的過程中正確調用相關函數，即可滿足需求。開始之前，先設定幾個重要參數：NumDirect 代表一個檔案可容納的磁區（Sector）數量，而 NumTrack 則是由於檔案變大，需對磁碟（DISK）的空間進行相應擴展的大小設定。

```
#define NumDirect ((SectorSize - 3 * sizeof(int)) / sizeof(int))
const int SectorsPerTrack  = 32;  // number of sectors per disk track
const int NumTracks = 64;    // number of tracks per disk
```

我們的目標是可以順利執行-f, -cp, -p所以要確保下列四個程式能執行。

```
fileSystem = new FileSystem(formatFlag);
kernel->fileSystem->Create(to, fileLength)
openFile = kernel->fileSystem->Open(to);
openFile->Read(buffer, amountRead);
openFile->Write(buffer, amountRead);
```

其中FileSystem init, Create, Open不需要做更動，我們只需要處理Read跟Write。

Read會呼叫ReadAt，我們只需要將fileLength修正，再透過更改好的ByteToSector索取資料，程式就會正確了。Write的概念也一樣。

```
int OpenFile::ReadAt(char *into, int numBytes, int position)
{
   int fileLength = hdr->TotalFileLength();
   // 省略中間部分...
   for (i = firstSector; i <= lastSector; i++)
       kernel->synchDisk->ReadSector(hdr->ByteToSector(i * SectorSize),
                                     &buf[(i - firstSector) * SectorSize]);
}
```

```
int OpenFile::WriteAt(char *from, int numBytes, int position)
{
   int fileLength = hdr->TotalFileLength();
   // 省略中間部分...
   for (i = firstSector; i <= lastSector; i++)
       kernel->synchDisk->WriteSector(hdr->ByteToSector(i * SectorSize),
                                      &buf[(i - firstSector) * SectorSize]);
}
```

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

要實作 subdirectory 的結構，必須對底層到上層的 **DirectoryEntry**、**Directory**、**FileSystem** 等類別進行相應修改。

首先在 DirectoryEntry 中新增一個叫做`isDir`的成員變數，作用是用來判斷該 entry 是目錄還是檔案。接著，在Directory中，修改 Add() 方法，新增一個`isDir`參數來標記新增的 entry 類型，並實作 RecurList() 以支援類似 “`-lr`”的功能。此外，實作 GetDirSector() 方法，其功能是接收一個相對路徑，並回傳對應檔案所在的 sector。在 Add() 方法的具體實作中，當找到目標 entry 後，會新增設定`isDir`的邏輯以確保正確更新資料。

```
class DirectoryEntry

{

public:

   bool inUse;                     // Is this directory entry in use?

   bool isDir;

   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'

};
```

```
bool Add(char *name, int newSector, bool isDir); // Add a file name into the
directory
   void RecurList(int layer);
   int GetDirSector(char *name);
```

```
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].inUse = TRUE;
           strncpy(table[i].name, name, FileNameMaxLen);
           table[i].sector = newSector;
           table[i].isDir = isDir;
           return TRUE;
       }
   return FALSE; // no space.  Fix when we have extensible files.
}
```

在一般的 List() 中，修改輸出格式，用`isDir`判斷每個 entry是目錄還是檔案，並根據判斷結果進行相應的格式化輸出。在 RecursiveList() 的做法中，每個 entry 輸出前，會根據當前層數`layer`判斷所處層級，並加上對應數量的indent，在輸出完成後，如果該 entry 是目錄，則繼續遞迴呼叫 RecursiveList()，同時將`layer`增加 1。

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

```
void Directory::RecurList(int layer)
{
   Directory *subDir = new Directory(NumDirEntries);
```

```
    for (int i = 0;i < tableSize;i++){
        if (table[i].inUse){
            for (int j = 0; j < layer;j++) printf("    ");
            char fg = table[i].isDir ? 'D':'F';
            printf("[%c] %s\n", fg, table[i].name);
            if (table[i].isDir){
                OpenFile* file = new OpenFile(table[i ].sector);
                subDir->FetchFrom(file);
                subDir->RecurList(layer+1);
            }
        }
    }
}
```

這裡的GetDirSector()的主要步驟是進行字串處理。例如當name = "/t0/t1"時，會先取出 /t0，找到對應的 entry index，然後建立 /t0 的directory，並遞迴呼叫GetDirSector()，此時傳入的name為/t1。這個過程會持續進行，直到處理完整個字串，最後回傳該檔案或目錄所在的 sector 位置。在這個做法中我們假設除了root 外，所有name的結尾不會包含/。因此，當name = "/"時，即表示 root，直接回傳 1，因為 root 的資料存放在 sector 1。

```
int Directory::GetDirSector(char *name) {
    // find first entry
    if (!strcmp(name, "/")) return 1;
    int idx = 1;
    while (name[idx] != '\0' && name[idx] != '/') {
        idx++;
    }

    char entry[256];
    // copy entry
    strncpy(entry, name + 1, idx - 1);
    entry[idx - 1] = '\0';  // 保证末尾是 null 字符

    // find same name and return sector
    int i = FindIndex(entry), sector = -1;
    if (name[idx] != '\0'){
        Directory *dir = new Directory(NumDirEntries);
        OpenFile *file = new OpenFile(table[i].sector);
        dir->FetchFrom(file);
        sector = dir->GetDirSector(name + idx);
        delete dir;
        delete file;
    }else{
```

```
        sector = table[i].sector;
    }
    return sector;
}
```

在 FileSystem 中新增了三個功能函式，CreateDir()：供 kernel 呼叫來建立目錄、RecurList()：同樣由kernel呼叫，用於遞迴列出directory內容、SplitPath()：將傳入的路徑字串分為兩部分 Crea

```
static void CreateDirectory(char *name)

{

    // MP4 Assignment

    DEBUG(dbgFile, "createDir called.");

    kernel->fileSystem->CreateDir(name);

}
```

```
void FileSystem::RecurList(char *name){

    Directory *root = new Directory(NumDirEntries);

    Directory *directory = new Directory(NumDirEntries);

    root->FetchFrom(directoryFile);

    int sector = root->GetDirSector(name);

    OpenFile *file = new OpenFile(sector);

    directory->FetchFrom(file);

    directory->RecurList(0);

    delete root;

    delete directory;

    delete file;

}
```

```
void SplitPath(char *path, char* dirName, char* fileName){
```

```
    int len = strlen(path);

    int i;

    for (i = len - 1;i >= 0;i--){

        if (path[i] == '/') break;

    }

    strncpy(dirName, path, i);

    strncpy(fileName, path+i+1, len-(i+1));

    dirName[i] = '\0';

    fileName[len-(i+1)] = '\0';

    if (i == 0) dirName[0] = '/', dirName[1] = '\0';

}
```

在 Create() 中，傳入的name不再限於root下的路徑。因此，需要先使用 SplitPath() 將 name分割為所在目錄名稱與檔案名稱。接著透過 root 呼叫GetDirSector()找到該目錄所在的sector，開啟該目錄後，再按照原先的邏輯，在此目錄下建立新的檔案。

```
bool FileSystem::Create(char *name, int initialSize)
{
    Directory *root, *directory;
    PersistentBitmap *freeMap;
    FileHeader *hdr;
    OpenFile *file;
    int sector, dirSector;
    bool success;
    char dirName[512], filename[16];
    SplitPath(name, dirName, filename);

    DEBUG(dbgFile, "Creating file " << name << " size " << initialSize);

    root = new Directory(NumDirEntries);
    directory = new Directory(NumDirEntries);
    root->FetchFrom(directoryFile);
    dirSector = root->GetDirSector(dirName);
    file = new OpenFile(dirSector);
    directory->FetchFrom(file);
```

```
    if (directory->Find(filename) != -1)
        success = FALSE; // file is already in directory
    else
    {
        freeMap = new PersistentBitmap(freeMapFile, NumSectors);
        sector = freeMap->FindAndSet(); // find a sector to hold the file header
        if (sector == -1)
            success = FALSE; // no free block for file header
        else if (!directory->Add(filename, sector, FALSE))
            success = FALSE; // no space in directory
        else
        {
            hdr = new FileHeader;
            if (!hdr->Allocate(freeMap, initialSize)){
                success = FALSE; // no space on disk for data
            }else{
                success = TRUE;
                // everthing worked, flush all changes back to disk
                hdr->WriteBack(sector);
                directory->WriteBack(file);
                freeMap->WriteBack(freeMapFile);
            }
            delete hdr;
        }
        delete freeMap;
    }

    delete directory;
    delete root;
    delete file;
    return success;
}
```

在 Open() 中，將原本使用的Find()改為GetDirSector()，因為路徑可能包含好幾層，需要支援多層次結構。在 CreateDirectory() 方法中，流程與 Create() 類似，同樣先使用 SplitPath()將路徑分開，找到對應的目錄。不同之處在於Add() 方法的 `isDir` 參數需要設為TRUE，表示這是一個目錄。此外，在第 408 至 411 行，需要將該 sector 的資料初始化為目錄結構，並將其存回磁碟中。在 RecursiveList() 方法中，同樣會先找到目錄所在的 sector，開啟該目錄後，呼叫該目錄的 RecursiveList() 方法，並傳入 `0` 表示目前處於第 0 層。

```
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->GetDirSector(name);

    if (sector >= 0)

        openFile = new OpenFile(sector); // name was found in directory

    delete directory;



    curOpen = openFile;

    curOpenName = name;

    return openFile; // return NULL if not found

}
```

```
bool FileSystem::CreateDir(char* path){

    Directory *root, *directory;

    PersistentBitmap *freeMap;

    OpenFile *file;

    FileHeader *hdr;

    char dirName[256], fileName[10];

    int sector, dirSector;

    bool success;

    SplitPath(path, dirName, fileName);

    DEBUG(dbgFile, "Creating directory ");

    root = new Directory(NumDirEntries);

    directory = new Directory(NumDirEntries);
```

```
root->FetchFrom(directoryFile);

dirSector = root->GetDirSector(dirName);

file = new OpenFile(dirSector);

directory->FetchFrom(file);

if (directory->Find(fileName) != -1){

    success = FALSE;

}else{

    freeMap = new PersistentBitmap(freeMapFile, NumSectors);

    sector = freeMap->FindAndSet();

    if (sector == -1){

        success = FALSE;

    }else if (!directory->Add(fileName, sector, TRUE)){

        success = FALSE;

    }else{

        hdr = new FileHeader;

        if (!hdr->Allocate(freeMap, DirectoryFileSize)){

            success = FALSE;

        }else{

            success = TRUE;

            hdr->WriteBack(sector);

            directory->WriteBack(file);

            freeMap->WriteBack(freeMapFile);

            OpenFile *f = new OpenFile(sector);

            Directory* d = new Directory(NumDirEntries);

            d->WriteBack(f);

        }
```

```
            delete hdr;

        }

        delete freeMap;

    }

    delete directory;

    delete root;

    delete file;

    return success;

}
```

Bonus

Enhance the NachOS to support even larger file size

在 Part 2 中，我們採用了 Linked Index Scheme 來進行文件分配。理論上，不論文件多大，文件的頭部資料都能夠裝得下，因為文件大小的限制並不是由文件頭決定的，而是由磁碟的大小決定的。

根據 disk.h 檔案中的定義，原本磁碟的大小是 128 * 32 * 32 = 128KB。因此，磁碟的容量是有限的，這就決定了文件的最大大小。如果想讓文件能夠更大，我們需要增加磁碟的容量。

為了解決這個問題，我們只需要修改 NumTracks 的大小，將其設定為 32 * 512 = 16384，這樣磁碟的總大小就變成了 64MB。透過這種方式，我們可以讓每個文件的最大大小擴展到 64MB。

```
[os24team51@localhost test]$ ../build.linux/nachos -cp bonus1.txt /20
[os24team51@localhost test]$ ../build.linux/nachos -p /20
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
```

```
000832251 000832252 000832253 000832254 000832255 000832256 000832257 000832258 000832259 000832260
000832261 000832262 000832263 000832264 000832265 000832266 000832267 000832268 000832269 000832270
000832271 000832272 000832273 000832274 000832275 000832276 000832277 000832278 000832279 000832280
000832281 000832282 000832283 000832284 000832285 000832286 000832287 000832288 000832289 000832290
000832291 000832292 000832293 000832294 000832295 000832296 000832297 000832298 000832299 000832300
000832301 000832302 000832303 000832304 000832305 000832306 000832307 00[os24team51@localhost test]$
```