Operating System

MP4: File System

Team Member(Team 10): 林子謙 (112065517)、陳奕潔 (110000212)

Contributions: Both contributed to implementation and report.

Contents

1 Part I. Questions - NachOS File System

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

```
NachOS-4.0_MP4_original > code > filesys > filesys.h > ...
#ifndef FS_H
#else // FILESYS
class FileSystem

private:
    OpenFile *freeMapFile;    // Bit map of free disk blocks,
                              // represented as a file
    OpenFile *directoryFile;  // "Root" directory -- list of
                              // file names, represented as a file
};
```

Figure 1: filesys/filesys.h

In `filesys/filesys.h`, the class `FileSystem` defines the attribute `freeMapFile`. This suggests that NachOS FS manages free block space similarly to managing file content.

```
NachOS-4.0_MP4_original > code > threads > kernel.cc > Kernel(int, char **)
Kernel::Kernel(int argc, char **argv)
    for (int i = 1; i < argc; i++) {
        } else if (strcmp(argv[i], "-co") == 0) {
#ifndef FILESYS_STUB
        } else if (strcmp(argv[i], "-f") == 0) {
            formatFlag = TRUE;
#endif
```

(a) Kernel::Kernel()

```
Kernel::Initialize()
{
    // We didn't explicitly allocate the current thread we are running in.
    // But if it ever tries to give up the CPU, we better have a Thread
    // object to save its state.


    currentThread = new Thread("main", threadNum++);
    currentThread->setStatus(RUNNING);

    stats = new Statistics();       // collect statistics
    interrupt = new Interrupt;      // start up interrupt handling
    scheduler = new Scheduler();    // initialize the ready queue
    alarm = new Alarm(randomSlice); // start up time slicing
    machine = new Machine(debugUserProg);
    synchConsoleIn = new SynchConsoleInput(consoleIn); // input from stdin
    synchConsoleOut = new SynchConsoleOutput(consoleOut); // output to stdout
    synchDisk = new SynchDisk();    //
#ifdef FILESYS_STUB
    fileSystem = new FileSystem();
#else
    fileSystem = new FileSystem(formatFlag);
#endif // FILESYS_STUB
```

(b) Kernel::Initialize()

Figure 2: `formatFlag`

In `Kernel::Kernel()`, the `formatFlag` is set to `TRUE` if the command line includes `-f`. In `Kernel::Initialize()`, `formatFlag` is passed to `FileSystem` during NachOS FS initialization.

In `FileSystem::FileSystem()` (Figure 3), if `formatFlag` is true, this function creates new `PersistentBitmap` and `FileHeader` objects, named `freeMap`

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

        DEBUG(dbgFile, "Formatting the file system.");

        // First, allocate space for FileHeaders for the directory and bitmap
        // (make sure no one else grabs these!)
        freeMap->Mark(FreeMapSector);
        freeMap->Mark(DirectorySector);

        // Second, allocate space for the data blocks containing the contents
        // of the directory and bitmap files.  There better be enough space!

        ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
        ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));

        // Flush the bitmap and directory FileHeaders back to disk
        // We need to do this before we can "Open" the file, since open
        // reads the file header off of disk (and currently the disk has garbage
        // on it!).

        DEBUG(dbgFile, "Writing headers back to disk.");
        mapHdr->WriteBack(FreeMapSector);
        dirHdr->WriteBack(DirectorySector);
```

```cpp
        // 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);
        directoryFile = new OpenFile(DirectorySector);

        // Once we have the files "open", we can write the initial version
        // of each file back to disk.  The directory at this point is completely
        // empty; but the bitmap has been changed to reflect the fact that
        // sectors on the disk have been allocated for the file headers and
        // to hold the file data for the directory and bitmap.

        DEBUG(dbgFile, "Writing bitmap and directory back to disk.");
        freeMap->WriteBack(freeMapFile); // flush changes to disk
        directory->WriteBack(directoryFile);

        if (debug->IsEnabled('f'))
        {
            freeMap->Print();
            directory->Print();
        }
        delete freeMap;
        delete directory;
        delete mapHdr;
        delete dirHdr;
    }
    else
    {
        // if we are not formatting the disk, just open the files representing
        // the bitmap and directory; these are left open while Nachos is running
        freeMapFile = new OpenFile(FreeMapSector);
        directoryFile = new OpenFile(DirectorySector);
    }
}
```

Figure 3: FileSystem::FileSystem()

and `mapHdr`, respectively.

```cpp
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
};

#endif // PBITMAP_H
```

(a) filesys/pbitmap.h

```cpp
void Bitmap::Mark(int which)
{
    ASSERT(which >= 0 && which < numBits);

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

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

(b) Bitmap::Mark()

Figure 4: PersistentBitmap and Bitmap

`PersistentBitmap` inherits from `Bitmap`, which uses `0` and `1` to indicate whether a sector is empty or in use. The function `Bitmap::Mark()` sets the specified bitmap index to `1`.

Hence, `freeMap->Mark()` marks its `FreeMapSector` (sector `0`) as in use.

Next, `mapHdr` is initialized and allocated in the `freeMap` data block using `mapHdr->Allocate()`. The updated contents are then written back to the disk with `mapHdr->WriteBack()`.

`FileHeader::Allocate()` first ensures that the input `freeMap` has sufficient space for allocation. If so, it uses `freeMap->FindAndSet()` to locate empty bits; otherwise, it returns `FALSE`.

`Bitmap::FindAndSet()` use `Bitmap::Test()` to find a empty bit one by one.

`Bitmap::Test()` checks each index in the bitmap. If the value is `0`, indicating the block is empty, it returns `TRUE`; otherwise, it returns `FALSE`.

```
// Sectors containing the file headers for the bitmap of free sectors,
// and the directory of files.  These file headers are placed in well-known
// sectors, so that they can be located on boot-up.
#define FreeMapSector 0
#define DirectorySector 1
```

Figure 5: `FreeMapSector` and `DirectorySector`

```
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;
}
```

(a) FileHeader::Allocate()

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

    /*
        MP4 Hint:
        After you add some in-core informations, you may not want to write all fie
        Use this instead:
        char buf[SectorSize];
        memcpy(buf + offset, &dataToBeWritten, sizeof(dataToBeWritten));
        ...
    */
}
```

(b) FileHeader::WriteBak()

Figure 6

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

Figure 7: Bitmap::FindAndSet()

```
bool Bitmap::Test(int which) const
{
    ASSERT(which >= 0 && which < numBits);

    if (map[which / BitsInWord] & (1 << (which % BitsInWord)))
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}
```

Figure 8: Bitmap::Test()

```
void PersistentBitmap::WriteBack(OpenFile *file)
{
    file->WriteAt((char *)map, numWords * sizeof(unsigned), 0);
}
```

Figure 9: PersistentBitmap::WriteBack()

Finally, the function creates a new `OpenFile` object to access `freeMapFile`, which stores information about free disk blocks. It then uses `freeMap->WriteBack()` to update the modified contents of `freeMap` into `freeMapFile`.

This process illustrates how NachOS manages and locates free block space, with the information stored in sector `0`.

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

NachOS-4.0_MP4_original > code > machine > disk.h > ...

```
#ifndef 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
```

Figure 10: machine/disk.h

NachOS-4.0_MP4_original > code > machine > disk.cc > ...

```
const int MagicNumber = 0x456789ab;
const int MagicSize = sizeof(int);
const int DiskSize = (MagicSize + (NumSectors * SectorSize));
```

Figure 11: machine/disk.cc

Max disk size is `DiskSize` = 4 + `SectorSize` * `NumSectors` ≈ 128 * 32 * 32 = 128 KB

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

In `filesys/directory.h`, `Directory` uses a table to manage `DirectoryEntry`. Each `DirectoryEntry` has attributes: `inUse` to indicate whether the entry is active, `sector` to store the file header's location on disk, and `name` to record the file name. Thus, it currently supports a single-level directory.

```
class Directory
private:
    /*
        MP4 Hint:
        Directory is actually a "file", be careful of how it works with OpenFile and Fi
        Disk part: table
        In-core part: tableSize
    */

    int tableSize;          // Number of directory entries
    DirectoryEntry *table; // Table of pairs:
                           // <file name, file header location>

    int FindIndex(char *name); // Find the index into the directory
                               //  table corresponding to "name"
};
```

```
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'
};
```

Figure 12: filesus/directory.h

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

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

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

Figure 13: Directory::Directory()

In Figure 3, if the system needs formatting, the function creates `Directory` and `FileHeader` objects, named `directory` and `dirHdr`, respectively.

`freeMap` marks its `DirectorySector` (sector `1`) as in use with `Mark()`. Similarly, `dirHdr` calls `Allocate` and `WriteBack` to initialize its file header and write it back to the disk.

Finally, `FileSystem::FileSystem()` creates a new `OpenFile` object, `directoryFile`, to store file metadata and update the modified `directory` into `directoryFile`.

Based on this process, the directory data structure is stored in sector `1`.

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

```
class FileHeader
    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
};
```

Figure 14: filesys/filehdr.h

Inodes act as FCBs in UFS. In NachOS, an FCB is represented by a `FileHeader`, as shown in Figure 14. Each `FileHeader` stores the file size (`numBytes`), the number of data blocks used (`numSectors`), and the sectors for each data block (`dataSectors[NumDirect]`). Currently, `NumDirect` is calculated as $(128 - 2 \times 4)/4 = 30$.

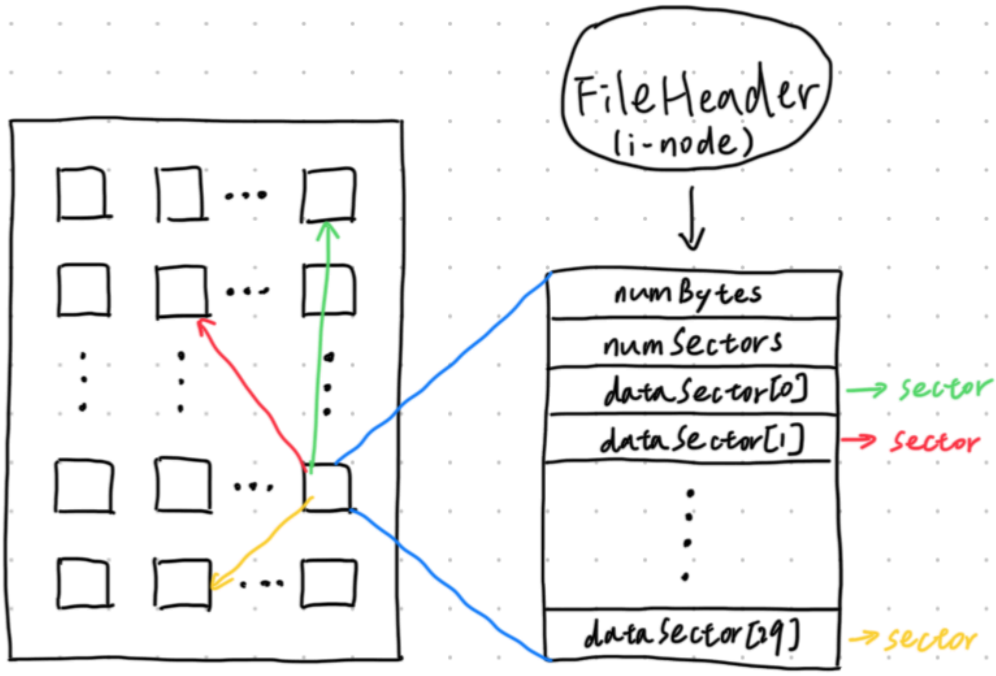

Figure 15: Direct Indexed Scheme

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

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

Figure 16: filesys/filehdr.h

`MaxFileSize` = (`NumDirect * SectorSize`) = 30 (computed as mentioned above) * 128 Bytes (shown in Figure 10) = 3.75 * 2 * 2 * 2 * 2^7 = 3.75 KB ≈ 4KB.

2 Part II. Implementation - File I/O System Calls & Larger File Size

2.1 Combine your MP1 file system call interface with NachOS FS to implement five system calls

如同 MP1，當 user program 使用 system call API 後，系統將之處理為 exception，並切換為 kernel mode 執行 kernel systeme calls。因此我們在 `userprog/exception.cc` 的 `ExceptionHandler()` 函式裡新增對應的 file system call handle cases，如 Figure 17 所示的 `case SC_Open` 與 `case S_Create`。`case SC_Read`、`case SC_Write`，和 `case SC_Close` 的實作邏輯也類似。其中唯一和 MP1 不同的是，`case S_Create` 多了檔案大小 `size` 的參數。

```
void ExceptionHandler(ExceptionType which)
    switch (which) {
        switch (type) {
            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;
            case SC_Open:
                /*
                #define SC_Open 6
                Open a file for read & write.
                */
                DEBUG(dbgFile, "In ExceptionHandler:case SC_Open.");
                val = kernel->machine->ReadRegister(
                    4);  // Retrieve file name address.
                {
                    char *filename =
                        &(kernel->machine
                              ->mainMemory[val]);  // Retrive file name.
                    DEBUG(
                        dbgFile,
                        "In ExceptionHandler:case SC_Open, into SysOpen.");
                    fileID = SysOpen(
                        filename);  // Success: get file ID / Fail: get -1
                    DEBUG(dbgFile,
                          "In ExceptionHandler:case SC_Open, return from "
                          "SysOpen.");
                    kernel->machine->WriteRegister(
                        2, fileID);  // Write file ID into register.
                }
```

Figure 17: ExceptionHandler()

```c
#endif
/* MP4 */
int SysCreate(char *filename, int size) {
    // return value
    // 1: success
    // 0: failed
    return kernel->fileSystem->Create(filename, size);
}

OpenFileId SysOpen(char *name) { return kernel->fileSystem->IdOpen(name); }

int SysRaed(char *name, int size, OpenFileId id) {
    return kernel->fileSystem->Read(name, size, id);
}

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

int SysClose(OpenFileId id) { return kernel->fileSystem->Close(id); }
#endif /* ! __USERPROG_KSYSCALL_H__ */
```

Figure 18: userprog/ksyscall.h

userprog/ksyscall.h 將 system calls 對接到 NachOS FS，如 Figure 18 所示。其中，此處的 SysOpen() 依照 Spec 規定要回傳 OpenFileId，因此會另外新建函式 FileSystem::IdOpen() 來處理，其餘的 system calls 則可以直接呼叫現存的函式。

另外，將系統改成支援 subdirectory 後，FileSystem::Create()、FileSystem::Open()，及其他 FileSystem 的 methods，包含 FileSystem::Remove() 和 FileSystem::List() 會因為處理路徑問題（可能有多層資料夾，要取得目標資料夾及檔案位置）而需要改動程式碼。詳細內容會於 Part III 說明。

由於 Spec 保證只有一個檔案會被開啟，我們新增了一個 attribute 名為 openedFile 的 OpenFile object，紀錄目前被打開的檔案，以 FileSystem::Read()、FileSystem::Write() 及 FileSystem::Close() 方便取用。

FileSystem::IdOpen() 呼叫 FileSystem::Open()，並將結果存到 openedFile，如果 openedFile 不是空的（NULL）代表成功開啟而回傳 1，否則回傳 -1，如 Figure 19 所示。

FileSystem::Read() 與 FileSystem::Write() 的實作會對接到 OpenFile 的 Read 與 Write method；FileSystem::Close() 則將 openedFile delete 後回傳 1 代表成功刪除檔案。如 Figure 20 所示。

```cpp
    OpenFileId IdOpen(char *name);

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

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

    int Close(OpenFileId id);

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

    void List(char *name, bool isRecursive);  // List all the files in the file sy

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


    char* FilenameCapture(char* name);
    char* CreateTraverse(Directory* dir, char* name);



  private:
    OpenFile *freeMapFile;     // Bit map of free disk blocks,
                               // represented as a file
    OpenFile *directoryFile;   // "Root" directory -- list of
                               // file names, represented as a file
    OpenFile *openedFile;
};
```

(a) in filesys/filesys.h

```cpp
OpenFileId FileSystem::IdOpen(char *name) {
    openedFile = Open(name);
    if (openedFile != NULL) {
        return 1;
    } else {
        return -1;
    }
}
```

(b) in filesys/filesys.cc

Figure 19: openedFile 與 FileSystem::IdOpen()

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

在 machine/disk.h 裡，我們把 NumTracks 改成 16384 讓 disk 能存到 64MB 的檔案大小（Bonus I 會說明），如 Figure 21 所示。

我們將 allocation scheme 從 direct indexed 改成 linked indexed scheme，使檔案長度不受 dataSectors table 大小的限制。我們在 dataSectors[0] 存放下一個 FileHeader，因此，每一個 dataSectors 能存的 data blocks 數量也就減少一個，即 29 個。

由於改變了 dataSectors 可存放 data blocks 數量，FileHeader 的管理方式需要修改，包含 FileHeader::Allocate()、FileHeader::Deallocate()、FileHeader::ByteToSector()，及 FileHeader::Print()。以下說明。

```cpp
//----------------------------------------------------------------------
// FileSystem::Read
//----------------------------------------------------------------------
int FileSystem::Read(char *buf, int size, OpenFileId id) {
    if (openedFile != NULL && id != -1) {
        DEBUG(dbgFile, "In FileSystem::Read, into OpenFile::Read()");
        int val = openedFile->Read(buf, size);
        DEBUG(dbgFile, "In FileSystem::Read, return from OpenFile::Read()");
        return val;
        // return openedFile->Read(buf, size);
    } else {
        return 0;
    }
}

//----------------------------------------------------------------------
// FileSystem::Write
//----------------------------------------------------------------------
int FileSystem::Write(char *buf, int size, OpenFileId id) {
    if (openedFile != NULL && id != -1) {
        return openedFile->Write(buf, size);
    } else {
        return 0;
    }
}

//----------------------------------------------------------------------
// FileSystem::Close
//----------------------------------------------------------------------
int FileSystem::Close(OpenFileId id) {
    delete openedFile;
    DEBUG(dbgFile,"In FileSystem::Close, after delete openedFile: " << openedFile);
    return 1;
}
```

Figure 20: FileSystem::Read(), Write() 及 Close()

```cpp
// MP4 Hint: DO NOT change the SectorSize, but other constants are allowed
const int SectorSize = 128;        // number of bytes per disk sector
const int SectorsPerTrack  = 32;  // number of sectors per disk track
const int NumTracks = 16384;
// const int NumTracks = 32;        // number of tracks per disk
const int NumSectors = (SectorsPerTrack * NumTracks); // total # of sectors per di
```

Figure 21: machine/disk.h

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

    if (numSectors < NumDirect) {
        /* Case 1: numSectors < NumDirect (30) */
        for (int i = 0; i < numSectors + 1; i++) {
            dataSectors[i] = freeMap->FindAndSet();
            // since we checked that there was enough free space,
            // we expect this to succeed
            ASSERT(dataSectors[i] >= 0);

            char *clean =new char[SectorSize]();
            kernel->synchDisk->WriteSector(dataSectors[i], clean);
            delete clean;
        }
        // FileHeader *nextIndexBlock = new FileHeader();
        // FileHeader *nextIndexBlock = NULL;
        // nextIndexBlock->WriteBack(dataSectors[0]);
        // delete nextIndexBlock;
    } else {
        /* Case 2: numSectors >= NumDirect (30) */
        for (int i = 0; i < NumDirect; i++) {
            dataSectors[i] = freeMap->FindAndSet();
            // since we checked that there was enough free space,
            // we expect this to succeed
            ASSERT(dataSectors[i] >= 0);

            char *clean =new char[SectorSize]();
            kernel->synchDisk->WriteSector(dataSectors[i], clean);
            delete clean;
        }
        FileHeader *nextIndexBlock = new FileHeader();
        nextIndexBlock->Allocate(freeMap, fileSize - (NumDirect - 1) * SectorSize);
        nextIndexBlock->WriteBack(dataSectors[0]);  // Put next i-node into the first entry
        delete nextIndexBlock;
    }
```

Figure 22: FileHeader::Allocate()

在 FileHeader::Allocate() 中，會判斷此檔案所需的 data blocks 數量（numSectors）。若在 29 以內（numSectors < NumDirect），便只 allocate numSectors + 1 的數量並 return。之所以加一，是因為我們規定，無論會不會用到兩個以上的 dataSectors table，每一 dataSectors 都會保留存放下一個 FileHeader 的 block。

若此檔案所需的 data blocks 數量若超過 29，會先 allocate 整個 dataSectors table 所需的空間，接著建立 FileHeader：nextIndexBlock，透過它遞迴 allocate 下一層 dataSectors table 的空間，遞迴回來後，將 nextIndexBlock 資料寫回 dataSectors[0]。

在每一次遞迴中，numBytes 和 numSectors 會依據輸入的檔案大小而更新，而這可作為其他 methods 是否要遞迴（有無下一層 dataSectors table）

```
void FileHeader::Deallocate(PersistentBitmap *freeMap) {
    // DEBUG(dbgFile, "In FileHeader::Deallocate()");
    FileHeader *nextIndexBlock = new FileHeader();
    nextIndexBlock->FetchFrom(this->dataSectors[0]);

    if (numSectors > NumDirect - 1) {
        nextIndexBlock->Deallocate(freeMap);
    }

    int numBlock = (numSectors + 1 < NumDirect) ? numSectors : NumDirect;
    for (int i = 0; i < numBlock; i++) {
        ASSERT(freeMap->Test((int)dataSectors[i]));  // ought to be marked!
        freeMap->Clear((int)dataSectors[i]);
    }
    delete nextIndexBlock;
    // DEBUG(dbgFile, ":O");
}
```

Figure 23: FileHeader::Deallocate()

的判斷條件。

FileHeader::Deallocate()（Figure 23）中，我們用來檢測目前 FileHeader 所管理的 dataSectors table 是否有下一層的方式是，依據目前 FileHeader 的 numSectors 數值是否超過 29。若超過 29，代表還有下一層，因此進入遞迴，回傳後再清除自己所 allocate 的 dataSectors table 空間；否則直接清除自己所 allocate 的 dataSectors table 空間。

FileHeader::ByteToSector()（Figure 24）中，檢測目前 FileHeader 所管理的 dataSectors table 是否有下一層的方式則是，檢查輸入進來的 offset 是否超過一個 dataSectors table 中可存 data blocks 的大小（(NumDirect - 1) * SectorSize）。若沒有超過，則回傳對應的 sector（由於 index 0 存放下一個 FileHeader 的位置，所以要加一）；超過的話，拿到下一個 FileHeader，由此進入遞迴。

```
int FileHeader::ByteToSector(int offset) {
    // DEBUG(dbgFile,"In FileHeader::ByteToSector(), offset: " << offset);

    if (offset < (NumDirect - 1) * SectorSize) {
        // DEBUG(dbgFile,"In FileHeader::ByteToSector(), index (offset / SectorSize +
        //                 << ", dataSectors[offset / SectorSize]: " << dataSectors[off
        //                 << ", dataSectors[offset / SectorSize + 1]: " << dataSectors
        return (dataSectors[offset / SectorSize + 1]);
    }
    FileHeader *nextIndexBlock = new FileHeader();
    nextIndexBlock->FetchFrom(dataSectors[0]);
    // DEBUG(dbgFile,"In FileHeader::ByteToSector(), nextIndexBlock's numBytes: " <<
    int where = nextIndexBlock->ByteToSector(offset - (NumDirect - 1) * SectorSize);
    delete nextIndexBlock;

    return where;
}
```

Figure 24: FileHeader::ByteToSector()

FileHeader::Print()（Figure 25）中，檢測目前 FileHeader 所管理的 dataSectors table 是否有下一層的方式則是，檢查目前 FileHeader 所存的 numBytes 是否超過一個 dataSectors table 中可存 data blocks 的大小。

若沒有超過，則印出目前 FileHeader 管理的 dataSectors table 內容（index 從 1 開始，data block sectors 及其 contents）；否則，印完目前這層的 dataSectors table 內容後，進入遞迴印出下一層的 dataSectors table 內容。

```
void FileHeader::Print() {
    int i, j, k;
    char *data = new char[SectorSize];
    FileHeader *nextIndexBlock = new FileHeader();
    nextIndexBlock->FetchFrom(dataSectors[0]);

    if (numBytes <= (NumDirect - 1) * SectorSize) {
        printf("FileHeader contents.  File size: %d.  File blocks:\n", numBytes);
        for (i = 1; i < numSectors; i++) printf("%d ", dataSectors[i]);
        printf("\nFile contents:\n");
        for (i = 1, 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");
        }
    } else {
        printf("FileHeader contents.  File size: %d.  File blocks:\n", numBytes);
        for (i = 1; i < NumDirect; i++) printf("%d ", dataSectors[i]);
        printf("\nFile contents:\n");
        for (i = 1, k = 0; i < NumDirect; 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");
        }
        nextIndexBlock->Print();
    }

    delete[] data;
    delete nextIndexBlock;
}
```

Figure 25: FileHeader::Print()

3 Part III. Implementation - Subdirectory

Subdirectory 的部分會分為四個部分講解，分別是 make directory (-mkdir)、list directory & file (-l)、recursively list directory & file (-lr) 和修改 file system create, open, remove file 絕對路徑處理的部分。

這裡先說明如何將 NachOS 修改成 support up to 64 files/subdirectories per directory 的版本。如 Figure 26所示，只要將 file.cc 內的 define NumDirEntries 修改成 64 就可以了。

```
// #define NumDirEntries 10
#define NumDirEntries 64
```

Figure 26: filesys.cc: NumDirEntries

3.1 Make Directory

首先-mkdir 的部分會由 main.cc 先將-mkdir 參數吃進去，並且將 mkdirFlag 設為 true，接著呼叫 CreateDirectory() 再觸發 filesys.cc 的 CreateDir()。如 Figure 27至 Figure 29所示。

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

Figure 27: main.cc: Get Argument -mkdir

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

Figure 28: main.cc: Call CreateDirectory()

接著是我們在 filesys.cc 新增的函式 CreateDir()，它會將需要建立的 directory 絕對路徑作為參數吃進來，Figure 30 是在 filesys.h 的宣告。然後在 filesys.cc 的實作如 Figure 31 至 Figure 33 所示。在 Figure 31 中我們先將等等需要用到的變數做宣告與初始化，包含創造一個空間把 root directory 從 disk load 進 memory 中。接著 Figure 32 展示 traverse directories 到需要建立 new directory 的那一層 directory。我們使用 strtok() 將路徑根據"/" 切成一段一段的 string，然後用 while loop 一層一層的從 root directory 開始尋找，直到找不到了就代表已經到達了需要創建 new directory

```
static void CreateDirectory(char *name)
{
    // MP4 Assignment
    kernel->fileSystem->CreateDir(name);
}
```

Figure 29: ain.cc: Invoke `CreateDir()`

```
    bool CreateDir(char *name);
```

Figure 30: filesys.h: Declaration of `CreateDir()`

的那一層，在 line 296 將 directory 名稱設定好後 break loop。在 Figure 33 中我們開始建立 directory。先做 directory name duplication 的檢查，由於 NachOs 會將這些與檔案操作有關的 object 作為 file 去管理，因此我們標記一個空間給這個實際存放 directory structure 位址的 file header structure，然後呼叫 `directory->Add(dirName, sector, FALSE)` 在 directory 結構中新增一個新的 directiry structure(`Add()` 函式我們有稍作修改，等等會提及)。檢查完以上條件後，我們新建一個 `FileHeader` object 並讓它去 allocate 一個 directory file size 的 disk 空間存放這個新 directory 的資料內容。最後將新增加的 file header、directory 結構、使用的空間標記寫回 disk。

```
bool FileSystem::CreateDir(char *name) {
    DEBUG(dbgFile, "In FileSystem::CreateDir()");
    Directory *directory;
    PersistentBitmap *freeMap;
    FileHeader *hdr;
    int sector;
    bool success;
    char *dirName;
    char *copyName;
    char *token;
    OpenFile *dirFile = directoryFile;

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

Figure 31: filesys.cc: Declaration of Variables & Get Root Directory from Disk in `CreateDir()`

由於在 directory 中的 `Add()` 函式我們有稍作修改，支援不只 file 還包含 directory 的新增操作。因此，我們必須要去記錄一個存在於 directory 結構中的構造的類型，所以我們增加了一個 `isAFile` 布林變數去設定這個新增加的構造是 file 還是 directory。如 Figure 34 至 Figure 35 所示。

3.2 List Directory & File

列出一個 directory 中所包含的 directory 或 file 是由在 terminal 輸入`-l` 參數所控制，因此如同`-mkdir`，我們需要從 `main.cc` 開始著手，請參考 Figure 36 至 Figure 37。`List()` 這裡我們新增一個參數 `recursiveListFlag`

```
    // Get directory
    copyName = new char[strlen(name)];
    strcpy(copyName, name);
    token = strtok(copyName, "/");
    while (token != NULL) {
        sector = directory->Find(token);
        DEBUG(dbgFile,"    In FileSystem::CreateDir(), token: " << token << ", found secotr: " << sector);

        // Check non exited file or directory
        if (sector == -1) {
            DEBUG(dbgFile, "    0L0 token: " << token);
            dirName = token;
            break;
        }
        dirFile = new OpenFile(sector);
        directory->FetchFrom(dirFile);
        token = strtok(NULL, "/");
    }
```

Figure 32: filesys.cc: Traverse Directories in `CreateDir()`

```
    // Start creating
    if (directory->Find(dirName) != -1) {
        DEBUG(dbgFile, "directory is already in directory");
        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) {
            DEBUG(dbgFile, "no free block for file header");
            success = FALSE;  // no free block for file header
        }
        else if (!directory->Add(dirName, sector, FALSE)) {
            DEBUG(dbgFile, "no space in directory");
            success = FALSE;  // no space in directory
        }
        else {
            hdr = new FileHeader;
            // if (!hdr->Allocate(freeMap, initialSize)) {
            if (!hdr->Allocate(freeMap, DirectoryFileSize)) {
                DEBUG(dbgFile, "no space on disk for data");
                success = FALSE;  // no space on disk for data
            }
            else {
                success = TRUE;
                // everthing worked, flush all changes back to disk
                DEBUG(dbgFile, "In FileSystem::CreateDir(), call WriteBack()");
                hdr->WriteBack(sector);
                // directory->WriteBack(directoryFile);
                // ===== MP4 =====
                directory->WriteBack(dirFile);
                // ===== MP4 =====
                freeMap->WriteBack(freeMapFile);
                DEBUG(dbgFile, "In FileSystem::CreateDir(), end WriteBack()");
            }
            delete hdr;
        }
        delete freeMap;
        DEBUG(dbgFile, "    >>> To create directory: " << dirName << ", in directory: " << directory << ", sector=" << sector);
    }
    delete directory;
    return success;
}
```

Figure 33: filesys.cc: Create New Directory in `CreateDir()`

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

Figure 34: directory.h: New Argument, `isAFile`, of `Add()` Function

```
bool Directory::Add(char *name, int newSector, bool isAFile)
{
    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].isFile = isAFile;
            return TRUE;
        }
    return FALSE; // no space.  Fix when we have extensible files.
}
```

Figure 35: directory.cc: `Add()`

來傳遞是否要使用 recursive list 的資訊 (這裡先介紹非 recursive 的 list 的實作，因此傳遞的參數值為 false)。

```
        else if (strcmp(argv[i], "-l") == 0)
        {
            // MP4 mod tag
            ASSERT(i + 1 < argc);
            listDirectoryName = argv[i + 1];
            dirListFlag = true;
            recursiveListFlag = false;
            i++;
        }
```

Figure 36: main.cc: Get Argument `-l`

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

Figure 37: main.cc: Call `List()`

接著在 `filesys.cc` 中，我們將原本的 `List()` 修改成可以吃絕對路徑和是否執行 recursive list 的版本。以下說明請參考 Figure 38 至 Figure 41。如同 Figure 39所呈現的，和 make directory 一樣，我們要先將 root directory load 進 memory。接著執行 traverse，找到要 List 的那一層的 directory。這裡的 traverse 和 make directory 的 traverse 有些不一樣，因為 spec 有保證不會有 messey operation，因此這裡必須找到最後一層名字才會是要 list 的 directory，要把這最後一層的 directory 結構抓進來 (make directory 不會將最後一層結構抓進來，因為不存在，在抓進來之前會先 break) 才 break loop，請參考 Figure 40。最後請看 Figure 41，這裡依據是否執行 recursively list 的參數 `isRecursive` 來決定要呼叫 directory.cc 的 `List()` 或 `RecursiveList()`。非 recursively list 的部分就直接呼叫原本就在 directory.cc 的 `List()` 就好。

```
    void List(char *name, bool isRecursive);  // List all the files in the file system

```

Figure 38: filesys.cc: Declaration of `List()` in `List()`

```
void FileSystem::List(char *name, bool isRecursive) {
    char *dirName;
    char *copyName;
    char *token;

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

```

Figure 39: filesys.cc: Declaration of Variables & Get Root Directory from Disk in `List()`

```
    // Get directory
    copyName = new char[strlen(name)];
    strcpy(copyName, name);
    token = strtok(copyName, "/");
    while (token != NULL) {
        int sector = directory->Find(token);
        DEBUG(dbgFile,"    In FileSystem::List(), token: " << token << ", found secotr: " << sector);

        // Check exited directory
        dirName = token;

        // if((token = strtok(NULL, "/")) == NULL){
        //     break;
        // }

        OpenFile *dirFile = new OpenFile(sector);  // to read content
        directory->FetchFrom(dirFile);
        // token = strtok(NULL, "/");
        if((token = strtok(NULL, "/")) == NULL){
            break;
        }
        DEBUG(dbgFile,"PPP");
    }
```

Figure 40: filesys.cc: Traverse Directories in `List()`

```
    DEBUG(dbgFile,"    >>> In FileSystem::List(), dirName: " << dirName << ", in directory: " << directory);
    if (isRecursive) {
        directory->RecursiveList(0, NumDirEntries);
    } else {
        directory->List();
    }

    delete directory;
```

Figure 41: filesys.cc: List Directories in `List()`

3.3 Recursively List Directory & File

Recursively list 的部分和非 recursively list 部分大同小異。不同的地方在於一開始在 main.cc 就會先把參數 `recursiveListFlag` 設為 `true`，如 Figure 42所示。中間一樣會呼叫 filesys.cc 中的 `List()`，只是最後會進入 directory.cc 中的 `RecursiveList()`。

```
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++;
}
```

Figure 42: main.cc: Get Argument `-lr`

`RecursiveList()` 的部分實作在 directory.cc 底下，請參考 Figure 43至 Figure 44。這裡由於必須要列印縮排，因此會吃參數 `level` 來計算排要印多少組，並且因為還要去 DFS through 這一個 directory 底下的所有 directory，所以也要將 entry 數量傳入。接下來就是去判斷這個存在 directory 內的結構是 file 還是 directory，若是 directory 的話還要再遞迴列印裡面的內容。

```
    void RecursiveList(int level, int NumDirEntries);
```

Figure 43: directory.h: Declaration of `RecursiveList()`

```
void Directory::RecursiveList(int level, int NumDirEntries) {
    // printf("level: %d: ", level);
    for (int i = 0; i < tableSize; i++) {
        if (table[i].inUse) {
            for (int j = 0; j < level; j++) {
                printf("    ");
            }
            if (table[i].isFile) {
                printf("[F] %s\n", table[i].name);
            } else {
                printf("[D] %s\n", table[i].name);
                OpenFile *nextLevelDirFile = new OpenFile(table[i].sector);
                Directory *nextLevelDir = new Directory(NumDirEntries);
                nextLevelDir->FetchFrom(nextLevelDirFile);
                nextLevelDir->RecursiveList(level + 1, NumDirEntries);
            }
        }
    }
}
```

Figure 44: directory.cc: `RecursiveList()`

3.4 Absolute File Path

我們在 Part III 的部分新增了可以建立 subdirectory 的功能，因此這時候的 file create, open, remove 要考慮到絕對路徑的切割。我們在 filesys.cc 內的 `Create()`、`Open()`、`Remove()` 中加了可以 traverse 絕對路徑的程式碼，請參考 Figure 45至 Figure 47。這些 traverse 的邏輯其實都一樣，只是寫法不同而已。因為是針對 file 做操作，不用再繼續 fetch 下一層就可以 break loop 了，並把修改後的內容寫回所在的資料夾（`directory->WriteBack(dirFile)`），其餘剩下的 create, open, remove 動作和原本的是一模一樣的。

```cpp
bool FileSystem::Create(char *name, int initialSize) {
    copyName = new char[strlen(name)];
    strcpy(copyName, name);
    token = strtok(copyName, "/");
    while (token != NULL) {
        sector = directory->Find(token);
        DEBUG(dbgFile,"    In FileSystem::Create(), token: " << token << ", found secotr: " << sector);

        // Check non exited file or directory
        if (sector == -1) {
            DEBUG(dbgFile, "    0L0 token: " << token);
            fileName = token;
            break;
        }
        dirFile = new OpenFile(sector);
        directory->FetchFrom(dirFile);
        token = strtok(NULL, "/");
    }
```

Figure 45: filesys.cc: Absolute Path in `Create()`

```cpp
OpenFile *FileSystem::Open(char *name) {
    copyName = new char[strlen(name)];
    strcpy(copyName, name);
    token = strtok(copyName, "/");
    while (1) {
        sector = directory->Find(token);
        DEBUG(dbgFile, "    In FileSystem::Open(), token: " << token << ", found secotr: " << sector);

        // Check exited file or directory
        fileName = token;
        if((token = strtok(NULL, "/")) == NULL){
            break;
        }
        OpenFile *dirFile = new OpenFile(sector);
        directory->FetchFrom(dirFile);
    }
```

Figure 46: filesys.cc: Absolute Path in `Open()`

```
bool FileSystem::Remove(char *name) {

    // Get directory
    copyName = new char[strlen(name)];
    strcpy(copyName, name);
    token = strtok(copyName, "/");
    while (1) {
        sector = directory->Find(token);
        DEBUG(dbgFile,"    In FileSystem::Remove(), token: " << token << ", found secotr: " << sector);

        // Check existed file or directory
        fileName = token;
        if((token = strtok(NULL, "/")) == NULL){
            break;
        }
        dirFile = new OpenFile(sector);  // to read content
        directory->FetchFrom(dirFile);
    }
```

Figure 47: filesys.cc: Absolute Path in `Remove()`

4 Bonus

4.1 Bonus I

為了使 NachOS 可以支援 up to 64MB 的 single file size，我們將模擬硬體的 disk.h 內定義的 `NumTracks` 改成了 $2^{26-7-5} = 2^{19-5} = 2^{14} = 16364$（除掉 `SectorSize` 及 `SectorsPerTrack`），請參考 Figure 48。如此一來，一個 file 就可以使用超過 64MB。

```
const int SectorSize = 128;         // number of bytes per disk sector
const int SectorsPerTrack  = 32;    // number of sectors per disk track
const int NumTracks = 16384;
// const int NumTracks = 32;        // number of tracks per disk
const int NumSectors = (SectorsPerTrack * NumTracks); // total # of sectors per disk
```

Figure 48: disk.h: `NumTracks`

4.2 Bonus II

我們分別建立了有 200、800，及 2000 個數字的檔案。每一個檔案所印出的第一行 `FileHeader contents. File size: . File blocks:` 中的 `File size` 是該檔案的大小。由於我們實作成 linked indexed scheme，不同的檔案大小會有不同的 FileHeader 長度 (`Next FileHeader sector`)，故檔案 size 比較小，其 FileHeader size 也較小；檔案 size 比較大，其 FileHeader size 也較大。

```
os24team10@nachos:~/NachOS-4.0_MP4/code/test$ ../build.linux/nachos -cp num_200.txt /200
FileHeader contents.  File size: 2000.  File blocks:
os24team10@nachos:~/NachOS-4.0_MP4/code/test$ ../build.linux/nachos -cp num_800.txt /800
FileHeader contents.  File size: 8000.  File blocks:
Next FileHeader sector: 562
FileHeader contents.  File size: 4288.  File blocks:
Next FileHeader sector: 592
FileHeader contents.  File size: 576.  File blocks:
os24team10@nachos:~/NachOS-4.0_MP4/code/test$ ../build.linux/nachos -cp num_2000.txt /2000
FileHeader contents.  File size: 19999.  File blocks:
Next FileHeader sector: 629
FileHeader contents.  File size: 16287.  File blocks:
Next FileHeader sector: 659
FileHeader contents.  File size: 12575.  File blocks:
Next FileHeader sector: 689
FileHeader contents.  File size: 8863.  File blocks:
Next FileHeader sector: 719
FileHeader contents.  File size: 5151.  File blocks:
Next FileHeader sector: 749
FileHeader contents.  File size: 1439.  File blocks:
```

Figure 49: Multilevel header size