Operating systems
MP4

Team 67

108062128 林育丞

108062135 呂佳恩

分工表

林育丞	Trace Code, Question 回答整理, Report 整理
呂佳恩	Trace Code, Implementation, Report 整理

PART I

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

FileHeader::Allocate 中，依據 file size 算出需要存在幾個 sector 裡面，並依據所需 sector 數量，呼叫 freeMap->FindAndSet()，尋找在 bitmap 中還空閒的 bit 編號，同時也就是空閒的 sector 編號。

在 FileSystem 的建構子中，可以看到 FreeMapSector，而這個常數的值為 0，也就是存在 sector 0。

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

32 track * 32 sectors/track = 1024 sectors.
1024 sectors * 128 bytes/sector = 128KB.
But files have a size limit of about 4KB, and current file system support only ten files, so lesser than 40KB will be actually used.

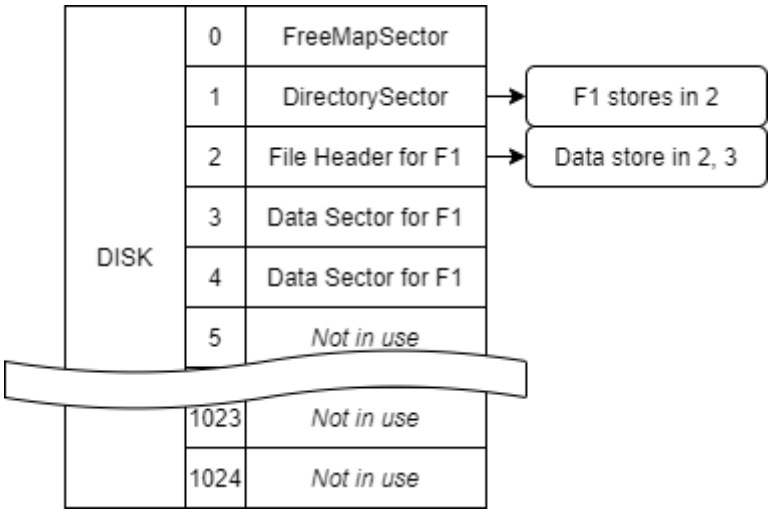
(3) Explain how the NachOS FS manage the directory data structure? Where is this information stored on the raw disk (which sector)?

在 FileSystem 的建構子中，如果格式化的 flag 為 true，則會呼叫 Directory 的建構子，並建立一個能存放 10 個檔案的簡單空資料夾。 如果沒有要格式化，則從 DirectorySector 讀取資料夾資訊。

DirectorySector = 1，所以是存在 sector 1。

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

FileHeader is the inode in NachOS, so the information stored in it are numBytes, numSectors, dataSectors.



▲ The figure for current implementation.

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

Since each File header will be stored in a sector, this means only 128 bytes of information can be stored in it.
numBytes, numSectors both take up 1 int space.
128 – sizeof(int) * 2 = 120 bytes.
120 bytes / sizeof(int) = 30.

So for each file, the number of sectors it occupied cannot be more than 30, which result in 30 * 128 bytes = 3840 bytes, about 4KB.

PART II

支援 NachOS FS system call：
複製 share 資料夾底下的 MP4 資料夾，並將 MP1 的 interface 接過去 (包含新增 exception.cc 裡的 SC_OPEN、SC_CLOSE 等，還有修改 ksyscall.h)
在 ksyscall.h 裡面，把 OpenFileTable 陣列改成單一 openFile 變數，並且把原本要回傳 OpenFile*的 SysOpen，改成回傳 OpenFileId，然後讓它總是回傳 true。

接著就把 FS_test1、FS_test2 依序放進 NachOS 執行，得出 ”Passed! ^_^” 的結果。

支援更大的 File Size：
在 File Header 加上 nextSector 和 nextHeader 屬性，如果單一 Header 存不下所有 data sectors，則將串接一個新的 File Header 在後面。
之後修改 Allocate, Deallocate, FetchFrom, WriteBack, ByteToSector, FileLength, Print 共七個函式，使 File Header 可以完整地支援 Linked 的結構。

```
// Read file header
kernel->synchDisk->ReadSector(sector, (char *)this);
FileHeader *ptr = this;
while (ptr->nextSector != -1)
{
    FileHeader *nextptr = new FileHeader();
    kernel->synchDisk->ReadSector(ptr->nextSector, (char *)nextptr);
    ptr->nextHeader = nextptr;
    ptr = nextptr;
}
```

▲ FetchFrom 的實作

```
FileHeader * ptr = this;
int count = 0;
while(ptr->nextSector != -1){
    count += ptr->numBytes;
    ptr = ptr->nextHeader;
}
count += ptr->numBytes;
return count;
```

▲ FileLength 的實作

PART III

在測試之前，把 code/test/DISK_0 檔案刪除，可以減少意外出錯的機率。

(1) Implement the subdirectory structure
在 main.cc 裡面的 CreateDirectory()，呼叫 FileSystem::Create()。
然後從 FIleSystem::Create() 開始修改。
我們把資料夾也視為一個檔案(為了方便開發，沿用了先前的架構)，所以得在函式裡加上一個 bool 參數，來記錄這個檔案是否為資料夾。
接著分析傳進來的 name 參數，一步一步將路徑拆解，並依據

1. 當前路徑
2. 目標路徑/檔案

來尋找路徑底下的檔案。最後再把資料寫回去該路徑所屬的 File 裡。

舉例說明： 已有/t0/t1 資料夾，要新增 f123 檔案。 (/t0/t1/f123)
目前資料夾：根目錄 → 搜尋目前資料夾底下的檔案是否有 t0 存在。
存在 → 將目前資料夾更新為 t0。
目前資料夾：t0 → 搜尋目前資料夾底下的檔案是否有 t1 存在。
存在 → 將目前資料夾更新為 t1。
目前資料夾：t1 → 搜尋目前資料夾底下的檔案是否有 f123 存在。
不存在 → 就在 t1 資料夾底下，建立 f123 檔案。

接著讓其他如 Remove、List、Open 等函式，都使用以上的搜尋法，來使其支援路徑。

```
currentDir :            Strlen : 0      tmp : /t0       target : /t0
entry name = /t0        target name = /t0
Successfully found Directory
currentDir : /t0        Strlen : 3      tmp : /bb       target : /bb
entry name = /f1        target name = /bb
entry name = /aa        target name = /bb
entry name = /bb        target name = /bb
Successfully found Directory
currentDir : /t0/bb     Strlen : 6      tmp : /f3       target : /f3
entry name = /f1        target name = /f3
entry name = /f2        target name = /f3
entry name = /f3        target name = /f3
Successfully found Directory
currentDir : /t0/bb/f3          Strlen : 9      target : SEARCH_COMPLETE
```

▲ 搜尋/t0/bb/f3 的過程

(2) Support up to 64 files/subdirectories per directory
直接將 NumDirEntries 改為 64 即可。由於 Part II 使用 Linked 的方式實作，所以不論是 Directory、FreeMap 或是一般的檔案，都可以取得足夠的空間，來存放其資料。

BONUS I

把 sectors per track 改成 512，num tracks 改成 1024，就可以擴充儲存空間至 64MB
128 * 512 * 1024 = 64MB
由於當前實作的 FileSystem 有 Overhead，要支援單一檔案 64MB，必須要有超過 64MB 的儲存空間，因此我們將 sectors per track 翻倍，改成 1024，這樣就可以支援 64MB 的單一檔案了。

使用 UNIX 指令來創建 64MB 的空檔案

*fallocate -l $((64*1024*1024)) 64MB.txt*

```
drwxr-xr-x.  2 os21team67 os2021 4.0K Jan 16 16:13 .
drwxr-xr-x. 12 os21team67 os2021  178 Jan 10 15:27 ..
-rw-r--r--.  1 os21team67 os2021  64M Jan 16 16:13 64MB.txt
-rw-r--r--.  1 os21team67 os2021   31 Jan 10 15:29 aaa.txt
-rw-r--r--.  1 os21team67 os2021  343 Jan 10 15:27 add.c
```

接著執行 bonusI.sh 可以檢測此過程，但即使我們將 stats.h 的時間都改為 1，仍然需要數十分鐘的時間才能完成整個過程。後來發現，我們卡住的地方並非寫入時間，而是為了找 bitmap 上的 free bit，每次都會跑迴圈檢查。

我們把 64MB.txt 在 FileHeader->Allocate 的時候，尚未 Allocate 的 sector 數量輸出出來，發現它越跑越慢，從 50 萬跑到 45 萬要 1 分鐘，但 45 萬到 43 萬卻要 5 分鐘。而 Allocate 裡面並沒有跟讀寫有關的函式，除了 O(1)的運算以外，只剩 FindAndSet。

```
75      while (currentSectors >= NumDirect)
76      {
77          // cout << currentSectors << " ";
78  >   O(30) for (int i = 0; i < NumDirect; i++) ···
83      O(1) {currentSectors -= NumDirect;
84           ptr->numBytes = NumDirect*SectorSize;
85           ptr->totalSectors = NumDirect;
86          ptr->nextSector = freeMap->FindAndSet();
87          // cout << ptr->nextSector << endl;
88      O(1) {ptr->nextHeader = new FileHeader();
89           ptr = ptr->nextHeader;        Empty Constructor
90      }
```

我們發現 FindAndSet 每次都會從 0 開始找 Free bit，對於需要存入 50 萬個 Sector 來說，越多 free bit 被占用，這個迴圈就要跑越久，才能找到下一個 free bit。
因此我們將 bitmap.cc 做了優化，每次 FindAndSet 的時候，都會記錄當下的位置，下次要找新的 bit，就可以直接從當前位置往下找，找不到才從 0 往上找。

```
114   int Bitmap::FindAndSet()
115   {
116   >     for (int i = currentBit; i < numBits; i++)···
125   >     for (int i = 0; i < currentBit; i++)···
134         return -1;
135   }
```

如此，之前卡好幾十分鐘的地方，現在只要不到一秒就解決。而後續在 WriteBack 的時候，也因為 stats.h 的時間都被改為 1，也能夠快速地完成。

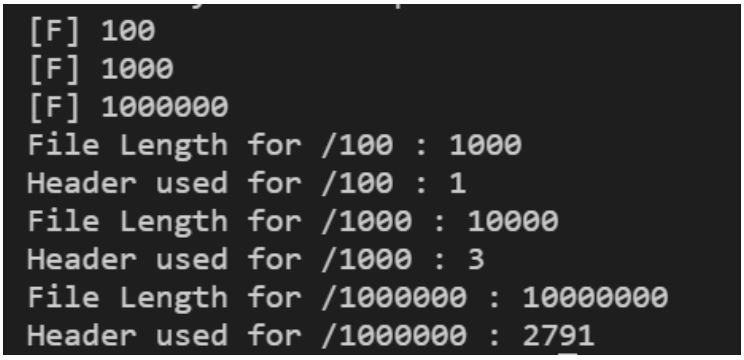
```
43883 543884 543885 543886 543887 543888 543889 543890 543891 543892 543893 543895 543896 543897 543898 543899 543900 543901 543902
543903 543904 543905 543906 543907 543908 543909 543910 543911 543912 543913 543914 543915 543916 543917 543918 543919 543920 543921
 543922 543924 543925 543926 543927 543928 543929 543930 543931 543932 543933 543934 543935 543936 543937 543938 543939 543940 54394
1 543942 543943 543944 543945 543946 543947 543948 543949 543950 543951 543953 543954 543955 543956 543957 543958 543959 543960 5439
61 543962 543963 543964 543965 543966 543967 543968 543969 543970 543971 543972 543973 543974 543975 543976 543977 543978 543979 543
980 543982 543983 543984 543985 543986 543987 543988 543989 543990 543991 543992 543993 543994 543995 543996 543997 543998 543999 54
4000 544001 544002 544003 544004 544005 544006 544007 544008 544009 544011 544012 544013 544014 544015 544016 544017 544018 544019 5
44020 544021 544022 544023 544024 544025 544026 544027 544028 544029 544030 544031 544032 544033 544034 544035 544036 544037 544038
544040 544041 544042 544043 544044 544045 544046 544047 544048 544049 544050 544051 544052 544053 544054 544055 544056 544057 544058
 544059 544060 544061 544062 544063 544064 544065 544066 544067 544069 544070 544071 544072 544073 544074 544075 544076 544077 54407
8 544079 544080 544081 544082 544083 544084 [F] 64MB
```

▲ 前面數字是 WriteBack 的 SectorNum，最後面是 -lr 所輸出的結果

同時我們可以以此數據計算 FileSystem 的 Overhead：
544084 sectors * 128 bytes = 69,642,752 bytes
64MB = 67,108,864 bytes
後者除以前者，得出大約 96%的空間是真正用來存資料的。

BONUS II

由於先前 FileHeader 是以 Linked 的結構實作，因此我們可以將每一個串接的 FileHeader 視為一個 Level，因此檔案越大，Fileheader 就越多，越小則反之。



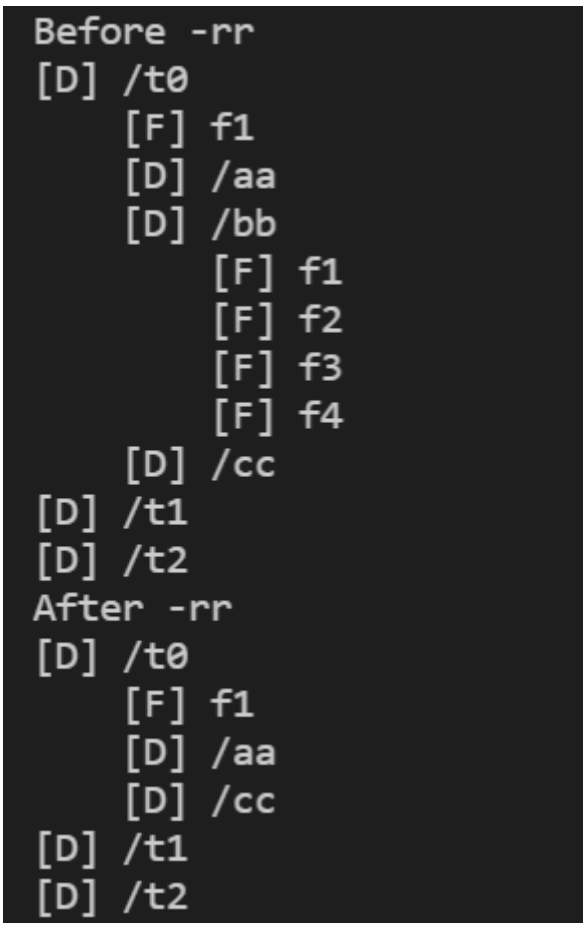
我們新增了一個 flag (-hdr)，用於輸出根目錄底下所有檔案的 FileLength，以及 Header 數量。
接著做了 bonusII.sh，將 num100、1000、1000000 三個檔案讀進去，並輸出，可以得到上圖的結果。可見 Header Size 會隨著檔案大小而有所不同。

BONUS III

FileSystem::RRemove() 遞迴式地逐一刪除所有此 directory 內的所有檔案。
其參數有二：
一、parentFile，用於後續刪除檔案時，更新父資料夾內的資訊。
二、target，也就是父資料夾內，需要被遞迴刪除的檔案名稱。

Directory::ReturnTable() 回傳 entry table。
有了 entry table 就能存取 sector 和檔案名稱，這時候遍歷所有 entry，判斷該 entry 是否為資料夾，是的話則進入下一層遞迴，否則直接刪除。

需要注意的是，只需要遞迴資料夾類型的檔案，一般檔案必須直接刪除。
所以我們在 directory 裡面加上 IsDirectory 函式，用來判斷 target 是否為資料夾。



使用 bonusIII.sh (與 FS_partIII.sh 一樣的測資，在最後會執行 -rr /t0/bb)