Operating System

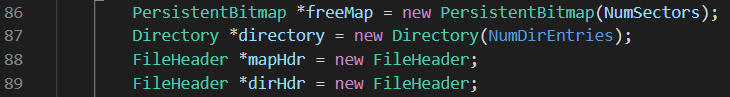
MP4 Report

組別：25

陳凱揚：Trace code、Implement、Report  
簡佩如：Trace code、Implement、Report

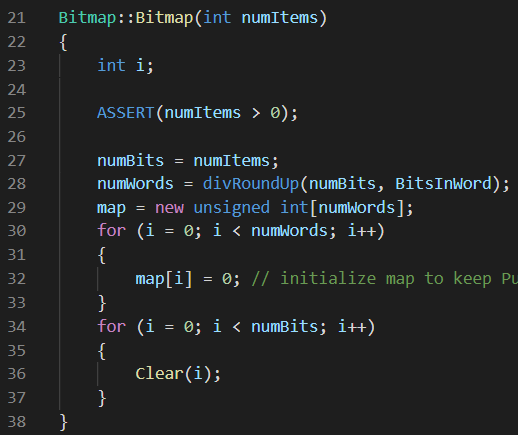
1. Understanding NachOS file system
   1. Explain how the NachOS FS manage and find free block space?

Nachos透過Bitmap來管理free blocks。首先在kernel中收到指令為 “-f” 時，formatFlag會設為TRUE，讓fileSystem建立時格式化disk。當沒有格式化時，會直接打開sector 0所在的檔案，在Nachos運行時保持開啟，存留在記憶體中；當需要格式化時，會去初始化NumSectors bits的PersistentBitmap，並new名為mapHdr、dirHdr的FileHeader，用來allocate空間給freeMap且寫回disk（圖1-1）。

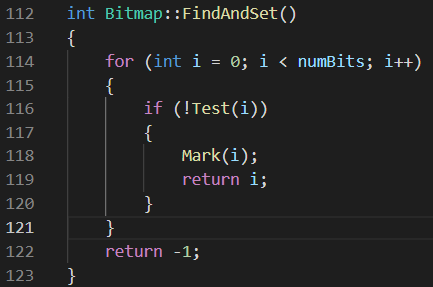


▲ 圖1-1

此外，PersistentBitmap繼承了Bitmap，可以透過Bitmap找到free block，以下可以看到管理方法會在一開始初始所有bit為0（0代表未使用、1代表使用中）（圖1-2），並且使用FindAndSet()找到一個bit為0的index（圖1-3），透過Mark()將其設為1，再回傳index，如果找不到救回傳 -1。



▲ 圖1-2



▲ 圖1-3

Where is this information stored on the raw disk (which sector)?

一開始會透過Mark()將FreeMapSector（sector 0）設為使用中，之後才不會再用到這個sector。接著mapHdr會allocate data block給freeMap，而FreeMapFileSize是freemap的大小，為NumSectors/BitsInByte（圖1-4）。



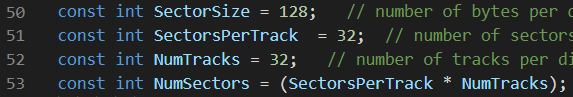




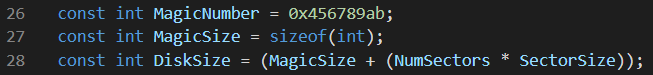
▲ 圖1-4

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

在dick.h中可以看到每個sector有128個bytes，每個track有32個sectors，disk有32個tracks（圖1-5）。而在disk.cc中可以看到DiskSize大小為4+(32\*32)\*128 = KB。其中MagicSize是用來驗證，防止不小心把有用的file當成disk（圖1-6）。



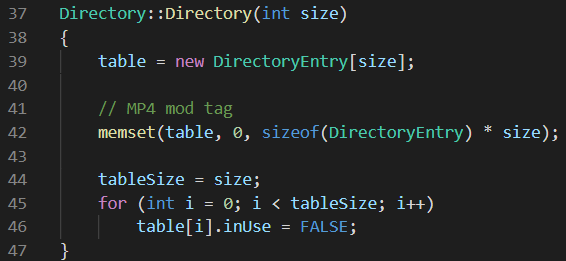
▲ 圖1-5



▲ 圖1-6

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

與freeMap很像，在建立FileSystem時，當不需格式化時，直接打開sector 1所在的檔案，保留在記憶體中；當需要格式化時，建立root directory初始化底下的DirectoryEntry，將每個entry的inUse設為0（圖1-7），並allocate空間且Mark()所使用的空間，DirectoryFileSize為sizeof(DirectoryEntry)\*NumDirEntries（圖1-8）。



▲ 圖1-7



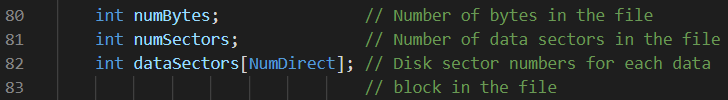




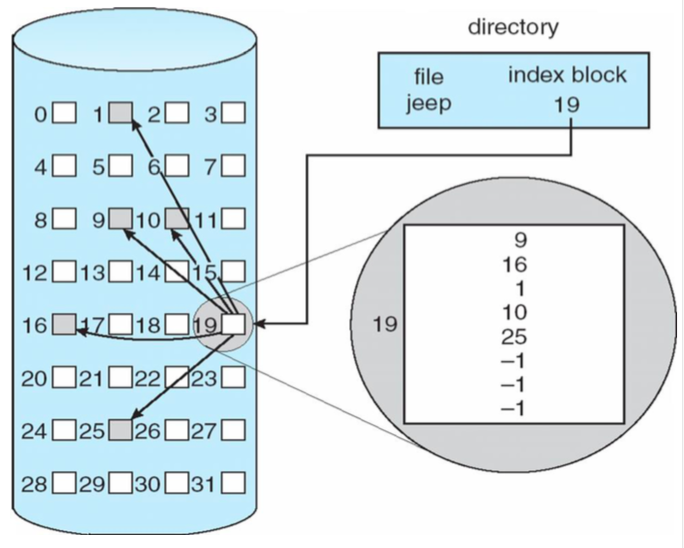
▲ 圖1-8

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

在filehdr.h的FileHeader這個class可以看到inode所儲存的相關資訊（圖1-9），其中numBytes為檔案的大小，numSectors為檔案使用了多少sector，dataSectors為儲存index的table，Nachos 使用direct index scheme（圖1-10）。



▲ 圖1-9



▲ 圖1-10

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

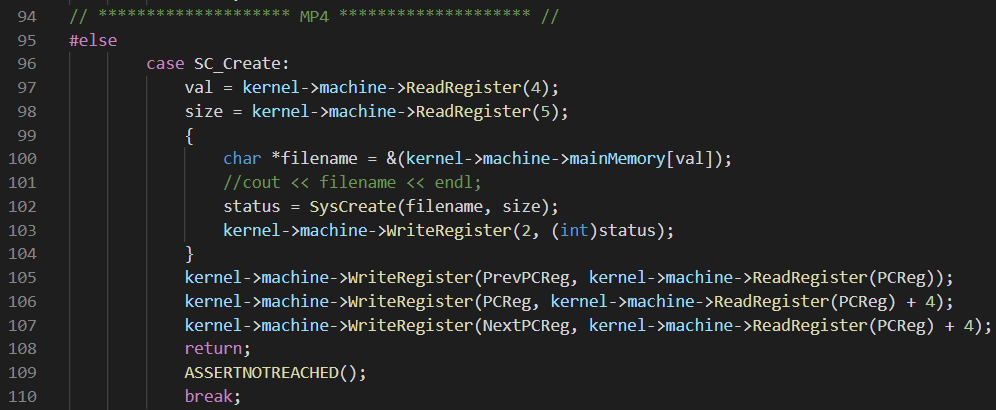
由於目前為direct index allocation，file header只能使用1個sector的大小，所以能儲存的index有限。而在filehdr.h中可以看到MaxFileSize為NumDirect\*SectorSize，其中NumDirect為(SectorSize-2\*sizeof(int))/sizeof(int)，扣掉的2個int為numBytes和numSectors（圖1-11）。NumDirect為(128-2\*4)/4 = 30，則MaxFileSize = 30\*128，約為4KB。



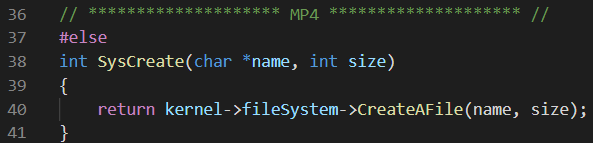
▲ 圖1-11

1. Modify the file system code to support file I/O system call and larger file size
   1. Combine your MP1 file system call interface with NachOS FS

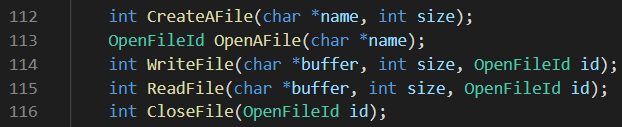
與MP1大致相同，user呼叫system call後，進入syscall.h（這裡原先就建好interface了），接著程式執行後，會進到exception.cc對應的case（圖2-1，以SC\_Create為例），再呼叫ksyscall.h中對應的function（圖2-2，以SysCreate()為例），最後呼叫fileSystem下對應的function（圖2-3）。



▲ 圖2-1



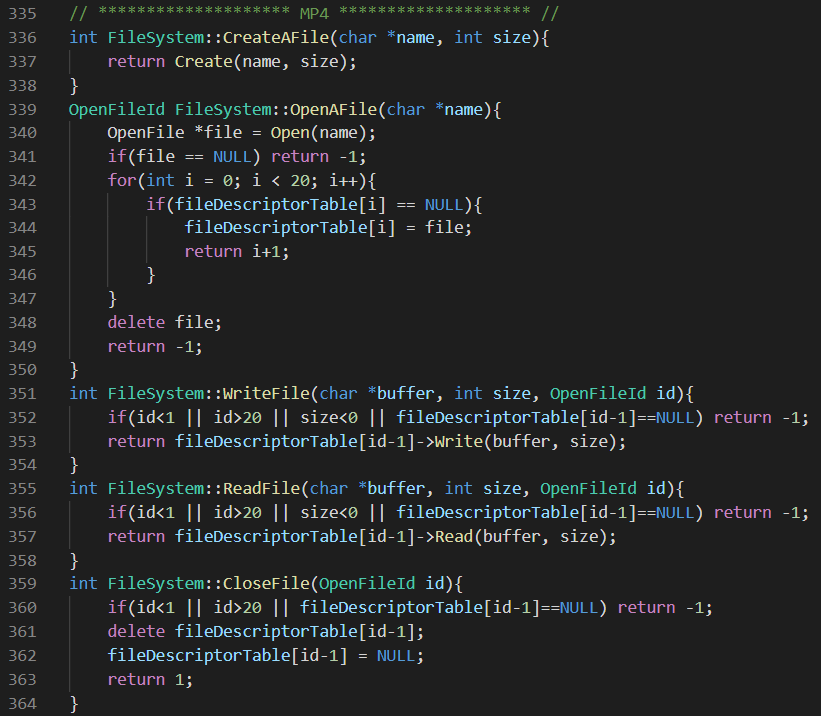
▲ 圖2-2



▲ 圖2-3

* 1. Implement five system calls

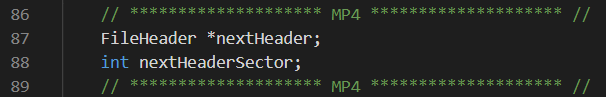
與MP1相同（圖2-4），只是多了CreateAFile()，且要在fileSystem下新增fileDescriptorTable來儲存開啟的檔案。



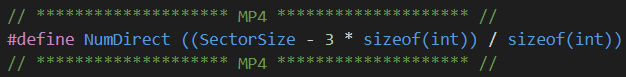
▲ 圖2-4

* 1. Enhance the FS to let it support up to 32KB file size

在這邊我將原先的index scheme，加強成linked index scheme，使檔案的長度不受限制。首先在filehdr.h下，我新增了nextHeader和nextHeaderSector等2個private變數（圖2-5），用以儲存下一個index block的資訊，其中注意的是nextHeader一定要寫在第一個，因為他不需要存回disk中，放在第一個用來之後WriteBack()時可以直接跳過，細節會在後面再做解釋。此外，NumDirect也必須更新（圖2-6），扣除nextHeaderSector的大小，使需要存回disk的大小與sector大小保持相同。

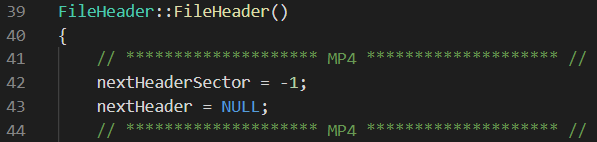


▲ 圖2-5

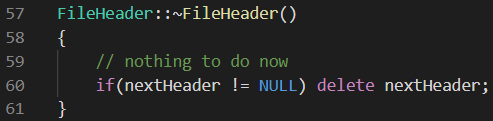


▲ 圖2-6

接著開始修改每個函式，首先在constructor初始化（圖2-7）、destructor delete掉（圖2-8）。

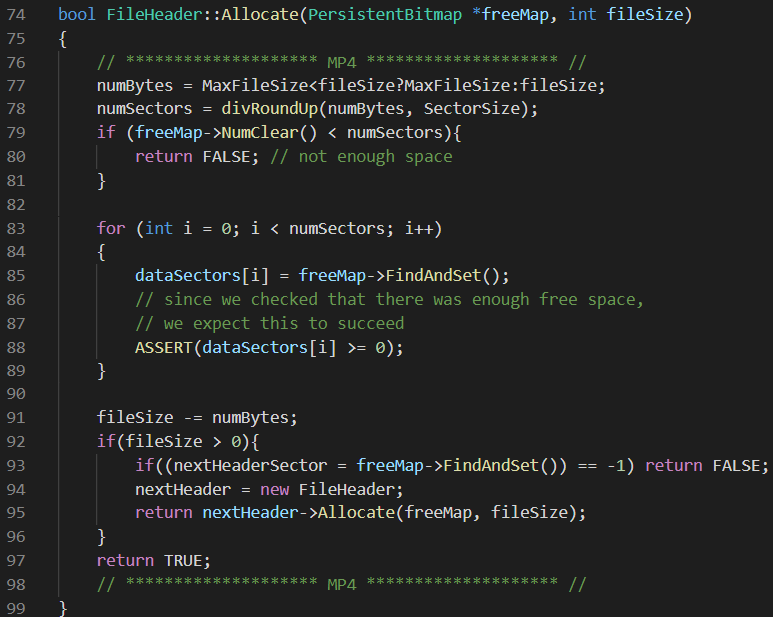


▲ 圖2-7

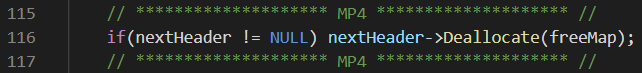


▲ 圖2-8

在Allocate()中（圖2-9），如果發現一個block存不下所有檔案的話，就會去尋找一個新的sector，儲存下一個file header，並繼續往下呼叫這個header的Allocate()儲存剩餘的檔案。Deallocate()則是會繼續往下一個file header呼叫Deallocate()（圖2-10）。

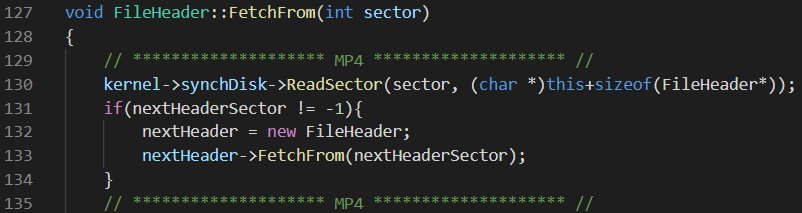


▲ 圖2-9

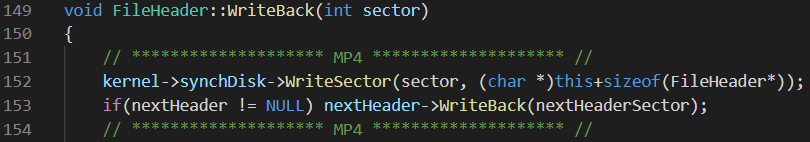


▲ 圖2-10

在FetchFrom()中（圖2-11），同樣會繼續遞迴呼叫下一個file header的FetchFrom()，其中注意的是這邊從disk讀出來的資料並不包含nextHeader，所以需要以nextHeaderSector判斷是否有下一個header，若有的話就new一個新的FileHeader。在WriteBack()時（圖2-12），會將this加上sizeof(FileHeader\*)用來跳過前面所提到的不需存回disk的nextHeader，因此在剛剛的FetchFrom()裡，this同樣會加上sizeof(FileHeader\*)，用來跳過這個位置。

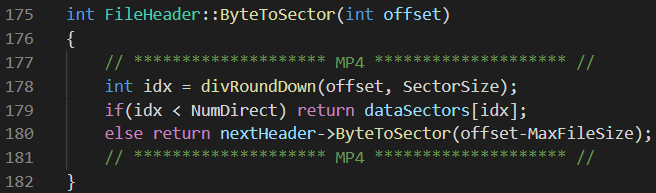


▲ 圖2-11

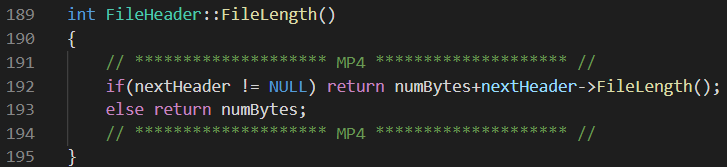


▲ 圖2-12

在ByteToSector()中（圖2-13），如果offset超過這個block儲存的上限，便會遞迴的向下尋找，並將傳入的offset減掉MaxFileSize。而FileLength()也是遞迴的加上後面的bytes再回傳（圖2-14）。

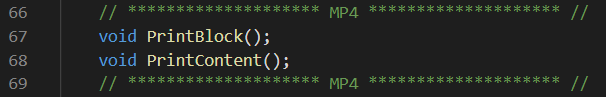


▲ 圖2-13



▲ 圖2-14

在Print()中，為了讓block資訊和content資訊能夠分開輸出，因此新增PrintBlock()和PrintContent()（圖2-15），用來分別在Print()裡呼叫遞迴。而這兩個function與原來的輸出方式皆相同，只是在最後加上遞迴呼叫（圖2-16）。



▲ 圖2-15

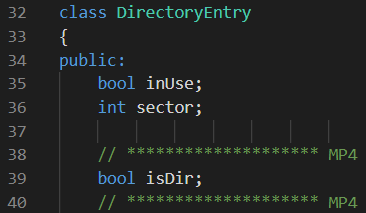


▲ 圖2-16

1. Modify the file system code to support subdirectory
   1. Implement the subdirectory structure

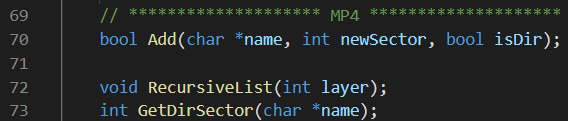
要實作出subdirectory的結構，從底層到上層的DirectoryEntry、Directory、FileSystem等都需要有對應的修改，這邊會從最底層開始解釋實作過程。

首先，在DirectoryEntry新增一個member variable為isDir，用來判斷此entry的資料是directory還是file（圖3-1）。

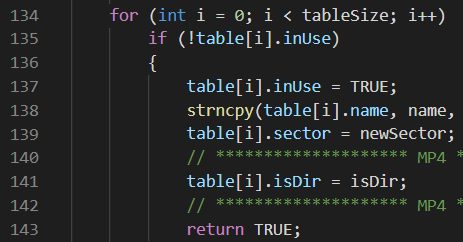


▲ 圖3-1

在Directory中（圖3-2），將原先Add()多傳入一個參數為isDir，並新增RecursiveList()來達到 “-lr” 的功能，而GetDirSector()的功能為傳入一個相對路徑，回傳此檔案所在的sector。而在Add()中（圖3-3），找到entry後，新增設定isDir的資料。



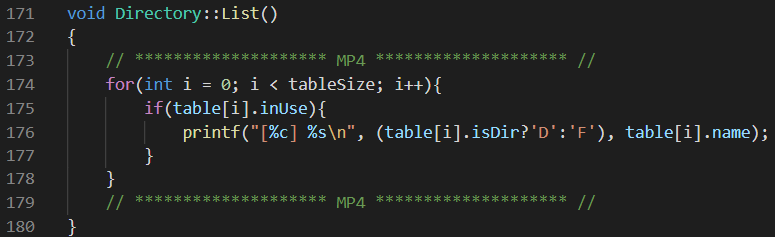
▲ 圖3-2



▲ 圖3-3

在一般的List()中（圖3-4），修改輸出為指定的格式，用isDir判斷是directory還是file。

在RecursiveList()中（圖3-5），在每個entry輸出前，會先以layer來判斷目前在第幾層，加上對應數量的indent，且在輸出後，若是directory的話，便繼續向下遞迴呼叫RecursiveList()，並將layer加1。

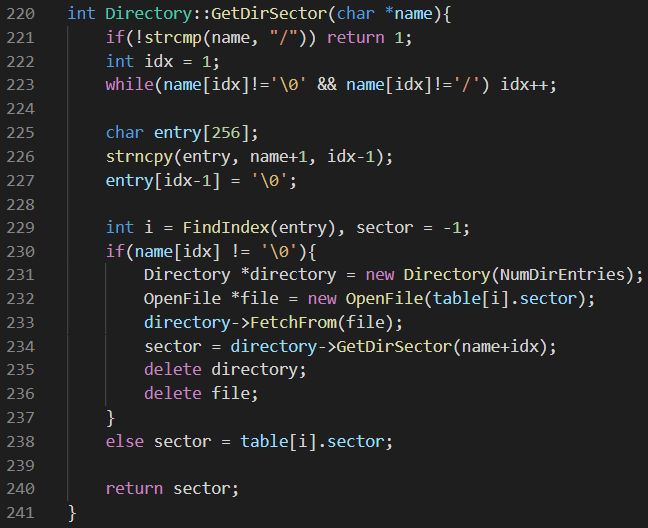


▲ 圖3-4



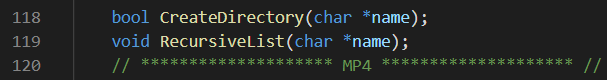
▲ 圖3-5

在GetDirSector()中的步驟主要為字串的處理（圖3-6），例如：name = “/t0/t1”，會先取出/t0，並找出其entry的index，接著建立t0的directory，遞迴呼叫GetDirSector()，此時傳入的name為/t1，直到整個字串結束後回傳sector位置。此外，在這邊會假設除了root外，所有的name結尾不會有 “/”，因此當name = “/”，即為root，回傳1代表root的資料存在sector 1。

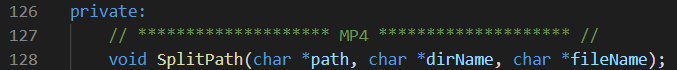


▲ 圖3-6

接著在FileSystem中新增3個function（圖3-7、圖3-8），CreateDirectory()用來使kernel呼叫並建立direcrtory，RecursiveList()也是由kernel所呼叫，SplitPath()的功能為將傳進的字串切成兩部分，如：name = “/t0/t1/t2，切成 ”/t0/t1” 和 ”t2”。

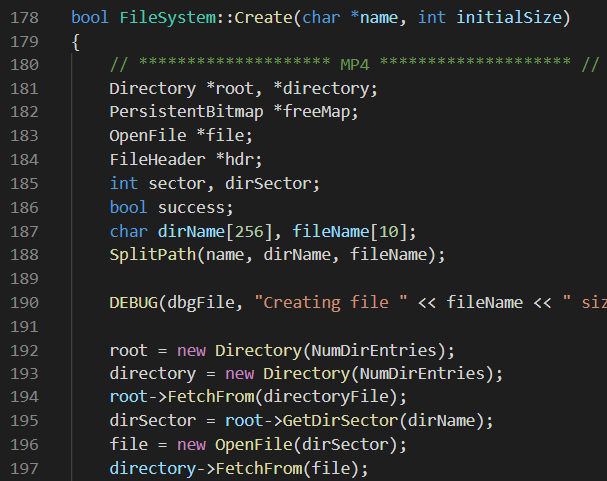


▲ 圖3-7

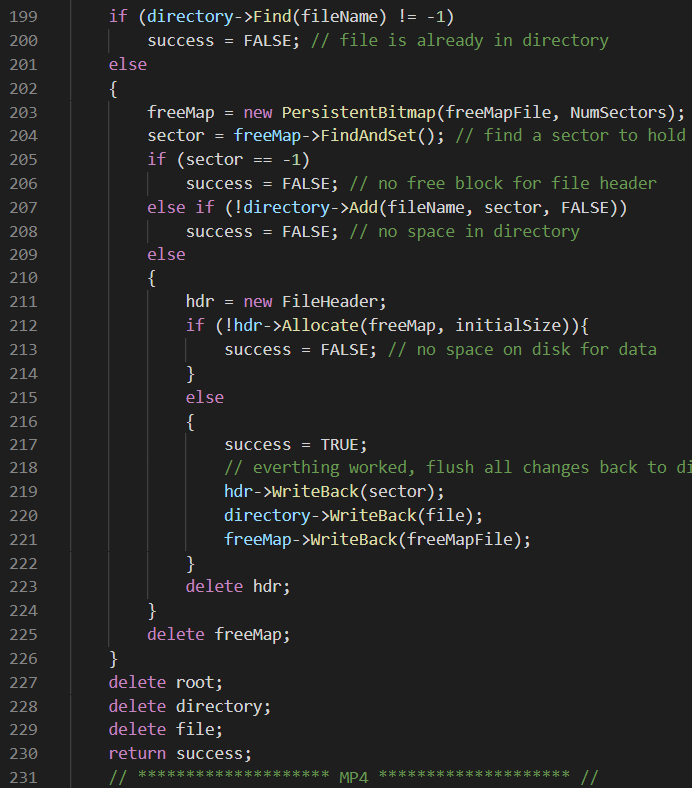


▲ 圖3-8

首先在Create()中（圖3-9、圖3-10），傳入的name不再是一定在root下了，因此需要先使用SplitPath()，將name切為所在directory名稱及file名稱，並且先從root中呼叫GetDirSector()來找到此directory所在的sector並開啟後，再依照原先的步驟，在這個directory下建立新的file。

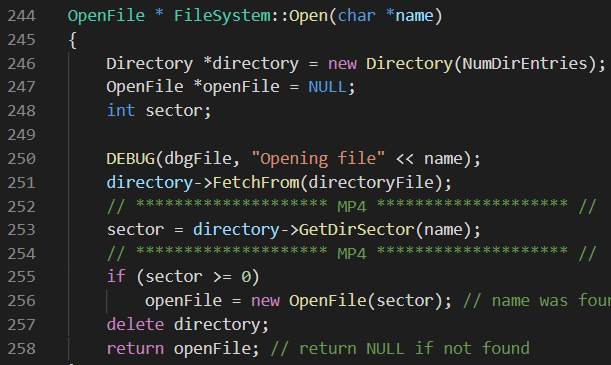


▲ 圖3-9



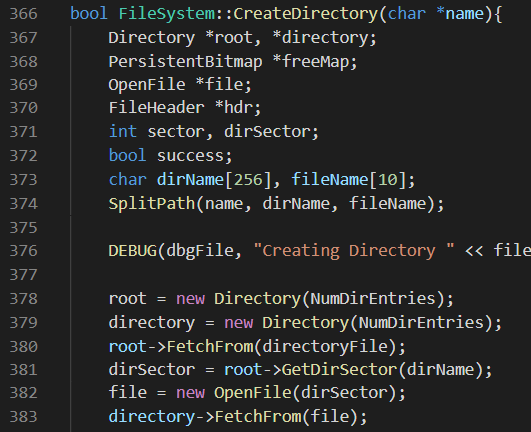
▲ 圖3-10

在Open()中（圖3-11），將原先的Find()改為使用GetDirSector()，因為路徑同樣可能是多層的。

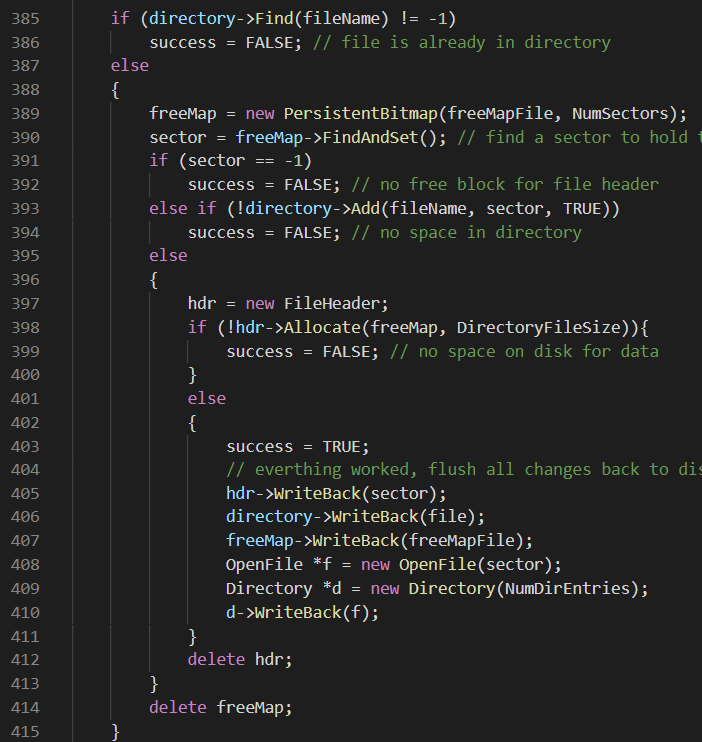


▲ 圖3-11

在CreateDirectory()中（圖3-12），與Create()相似，都是先切開路徑並找到directory，差別在於第393行時，Add()的參數isDir要設為TRUE，代表這是一個directory，還有第408~411行時，要將此sector的資料初始化為directory，並且存回disk裡（圖3-13）。

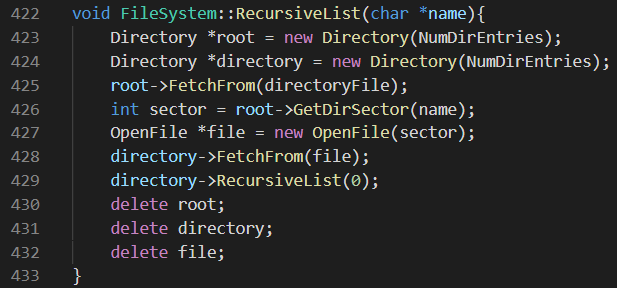


▲ 圖3-12



▲ 圖3-13

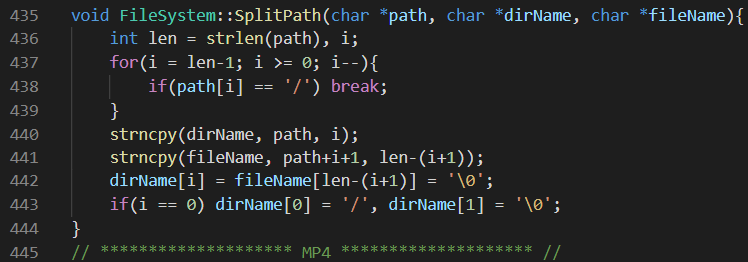
在RecursiveList()中（圖3-14），同樣先找到此directory的sector並開啟後，呼叫directory的RecursiveList()，傳入0代表現在在第0層。



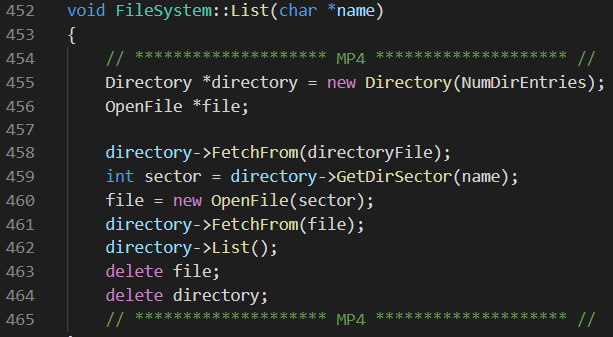
▲ 圖3-14

在SplitPath()中（圖3-15），主要在處理字串，直接將切開後的兩個字串，複製進dirName和fileName裡。

在List()中（圖3-16），與RecursiveList()大致相同，只是在最後改為呼叫此directory的List()就好，不需要recursive。

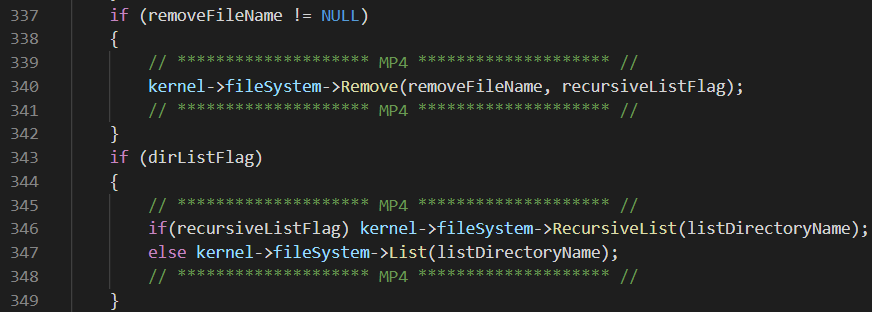


▲ 圖3-15

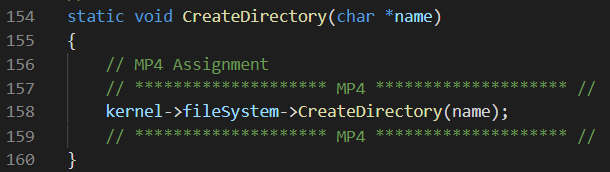


▲ 圖3-16

將以上FileSystem的功能都做好後，main()裡面就可以根據 “-l” 或 “lr”，呼叫List()或RecursiveList()（圖3-17），在CreateDirectory()中同樣呼叫fileSystem的CreateDirectory()（圖3-18）。



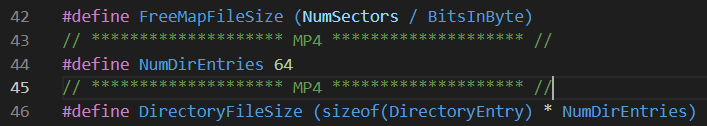
▲ 圖3-17



▲ 圖3-18

* 1. Support up to 64 files/subdirectories per directory

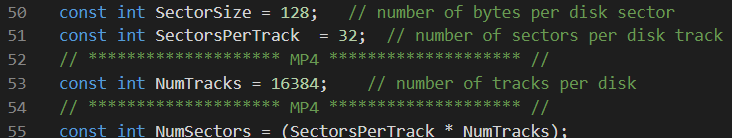
將NumDirEntries設為64，即可在directory下放入64個file。



▲ 圖3-19

1. Bonus
   1. Enhance the NachOS to support even larger file size

我們在part2所採用的是allocation方式為linked index scheme，理論上file size不管多大，header都能夠裝得下。而從disk.h中可以發現原本的disk大小為128\*32\*32 = 128KB，因此可以知道限制是在disk的大小，因此只需要改變NumTracks的大小為32\*512 = 16384（圖4-1），disk的大小即為64MB，就可以使一個file的大小最高可到64MB。

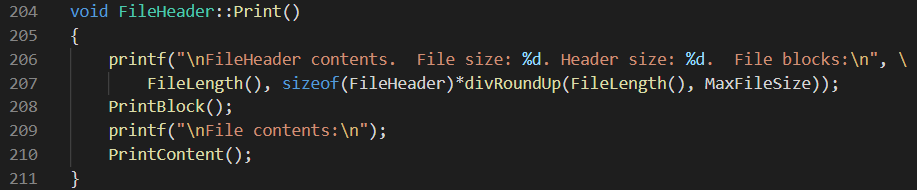


▲ 圖4-1

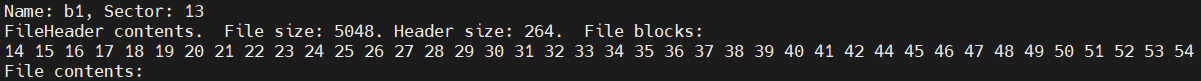
* 1. Multi-level header size

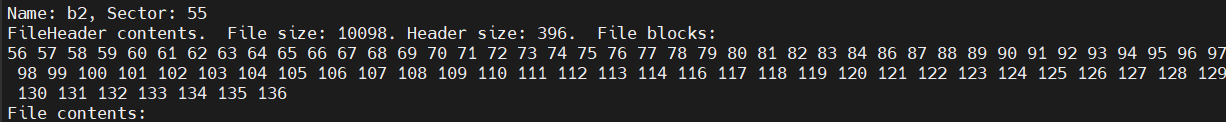
在FileHeader的Print()中（圖4-2），我新增了header size的顯示作為驗證，由於我們使用的是linked index scheme，越大的file size會有越大的header size。header size可由file size計算得到，每個header最多只能儲存MaxFileSize大小的檔案，算出所需幾個FileHeader後，再乘上sizeof(FileHeader)，即為檔案的header所佔據的空間。

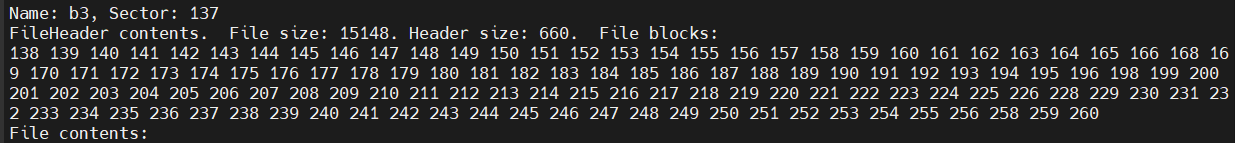
以下附上bonus2\_1.txt（5KB）、bonus2\_2.txt（10KB）、bonus2\_3.txt（15KB）等三個不同大小的檔案，複製進nachos後，可看出具有三種不同大小的header，分別為264B、396B、660B，也就是2個、3個、5個FileHeader（圖4-3）。



▲ 圖4-2



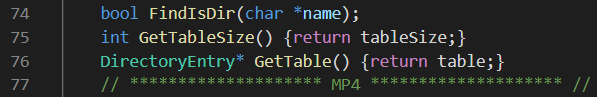




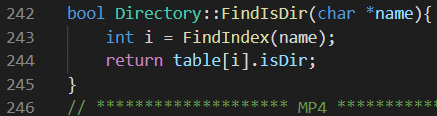
▲ 圖4-3

* 1. Recursive operations on directories

與part3一樣，需要從底層的Directory開始修改，到上層的FileSystem。首先在Directory新增FindIsDir()，功能為以檔案名稱搜尋，回傳此檔案是否為directory（圖4-5），GetTableSize()和GetTable()分別回傳對應的資訊（圖4-4）。

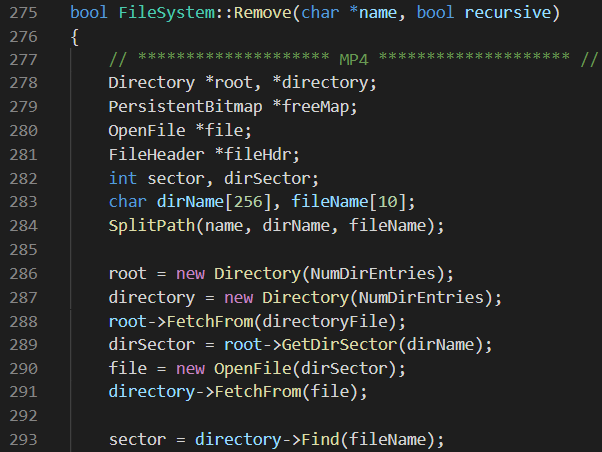


▲ 圖4-4

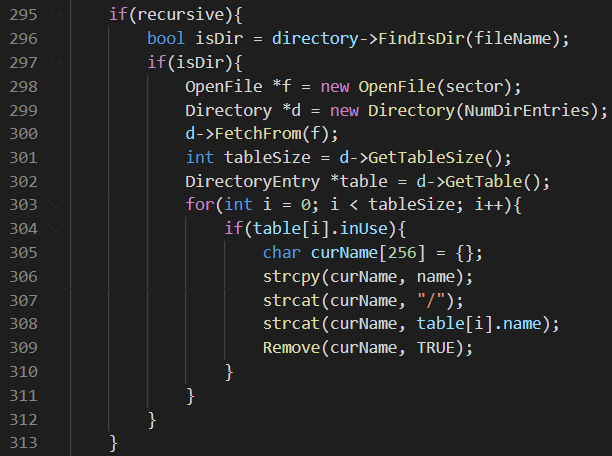


▲ 圖4-5

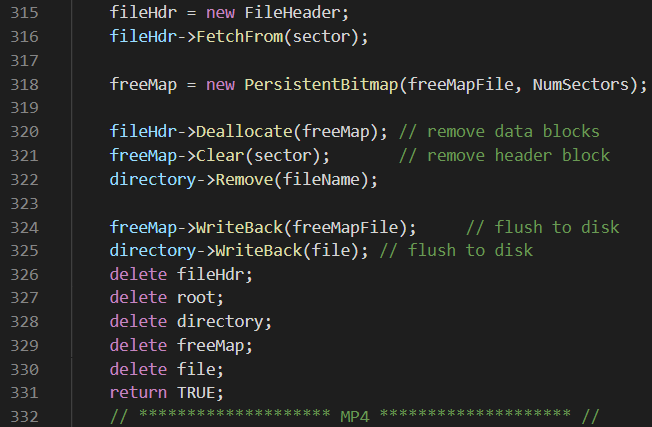
在FileSystem的Remove()中我新增了一個參數判斷是否需要recursive的remove（圖4-6），一開始與Create()相似，先對路徑做處理，找到directory及要remove的directory或file的名稱。接著如果需要recursive remove且為directory時，就先取得他的directoryEntry table，並開始遞迴呼叫Remove()，remove掉每個entry()（圖4-7）。遞迴結束後，即可開始刪除此directory或file，移除掉data block和header block（圖4-8）。



▲ 圖4-6

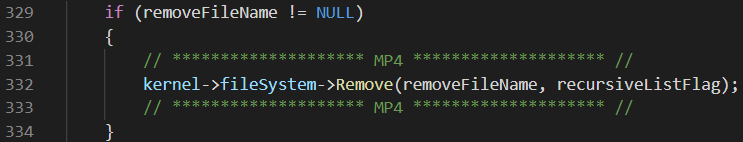


▲ 圖4-7



▲ 圖4-8

最後在main()裡，傳入recursiveListFlag，表示是否需要遞迴（圖4-9）。



▲ 圖4-9