**Part I. Understanding NachOS file system**

**Trace** the file system call and answer the following questions:

(1)

Q: Explain how the NachOS FS **manage and find free block space**?

A:

FS uses a bitmap object to manage its free block space. If a block is used, 1 will be set on the corresponding location in the bitmap. Otherwise, 0 should be set. Whenever we want to create a new file, FS will find free block space according to this bitmap.

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

A:



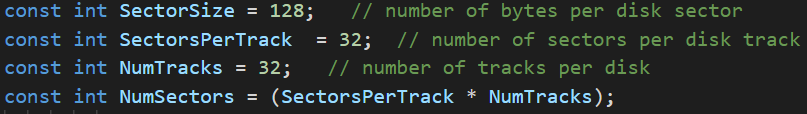
sector0(file header of bitmap)

(2)

Q: What is the **maximum disk size** that can be handled by the **current implementation**?

Explain **why**.

A:



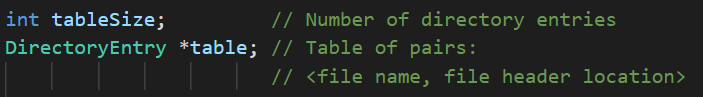
32\*32\*128Bytes = 128KB

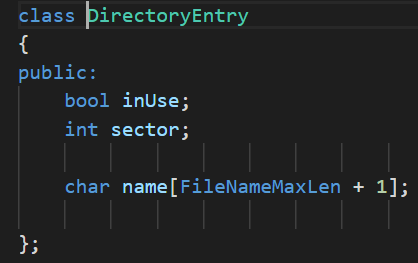
since we can see in *disc.h* that there are 128 bytes in a sector, 32 sectors in a track, and 32 tracks in a disc.

(3)

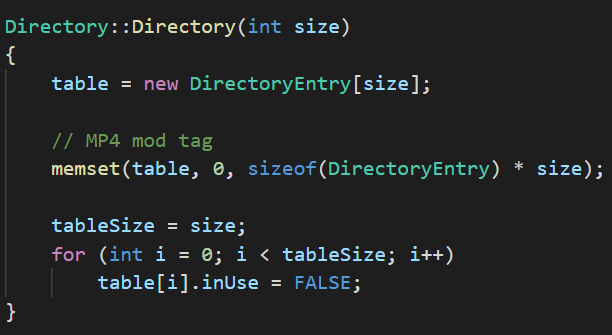
Q: Explain how the NachOS FS **manage the directory data structure**?

A:

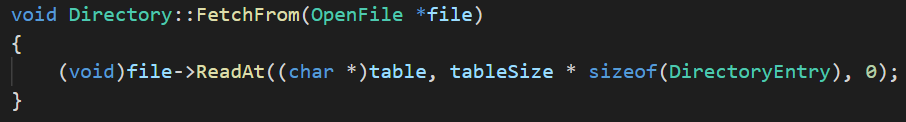




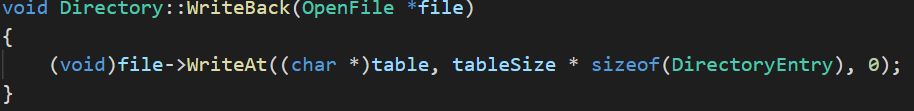
We can see in *directory.h* that *Class Directory* contains a table of *DirectoryEntry*, and each entry contains a file name and a sector number, where a file header locates in a disc. According to the current version of NachOS implementation, each *Directory* has 10 *DirectoryEntry*.



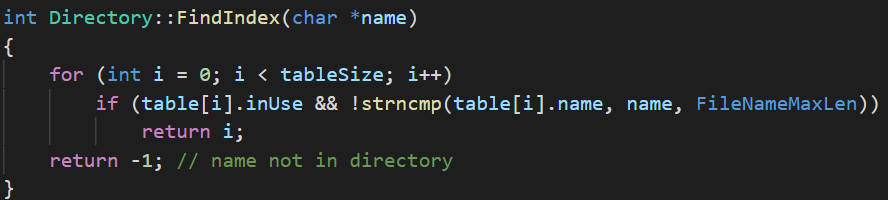
For the initialization function, we create a table, and set its contents to be 0. Then set every *DirectoryEntry*.inUse as False.



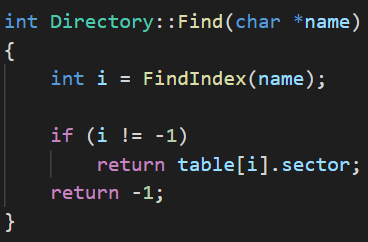
For this function, its purpose is to get *Directory* from an Openfile object, which has *Directory* contents stored in our disc previously.



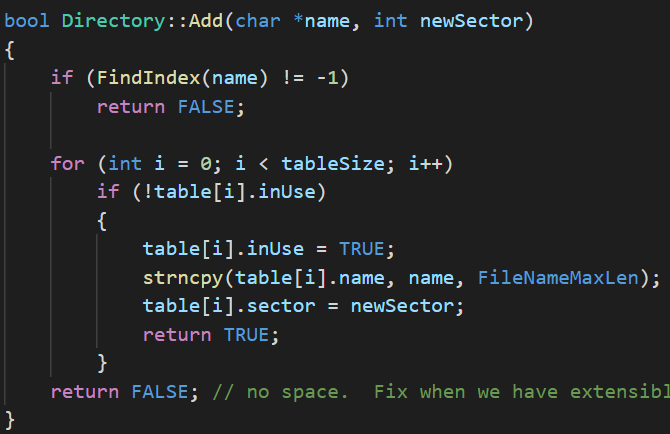
For this function, its purpose is to write itself back an Openfile object, which should be stored back into our disc later.



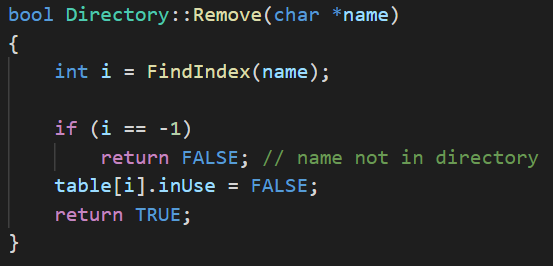
For this function, its purpose is that given a file name, we want to get its index in the *Directory* table. So we iterate through the whole table to find the index using the given name. If the file is not found in the table, this function will return -1; otherwise, it will return the index.



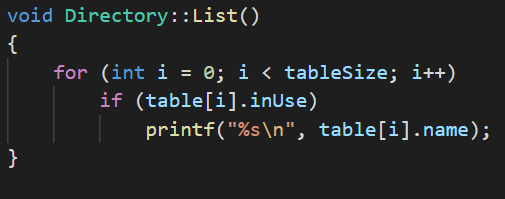
For this function, we will get an integer from the function FindIndex(), then we use it to find which sector the corresponding file is in, and return the location.



This function is to add a new file into a directory object. First, we need to find out if the upcoming file has already exist in the directory(return FALSE). If not, we will iterate through the directory table to find a space for the file. If we find a space for the file, copy the file name and set the entry as used, and recode the sector number of this file. If there is no space for the file, the function will return FALSE.



The function is to remove a specific file from a directory. First, we use its name to find the index of the file in the directory table. If we cannot find this file in the table, return FALSE. Otherwise, we release the entry by setting inUse as False, and return TRUE.



If we call this function, it can help us to see all files in this directory by listing out their names. The implementation is simple, we iterate through the table and list out all entries that the is using, then print out their name.



The function prints the content name of a directory by iterating through all the elements in it. If its table is being used then print the corresponding sector, then it fetches the directory from the table’s sector, and prints the contents of the file. Finally, delete the file header for memory management.

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

A:

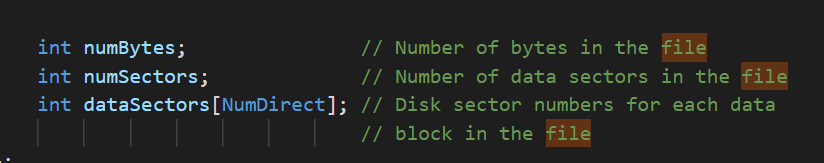


sector1

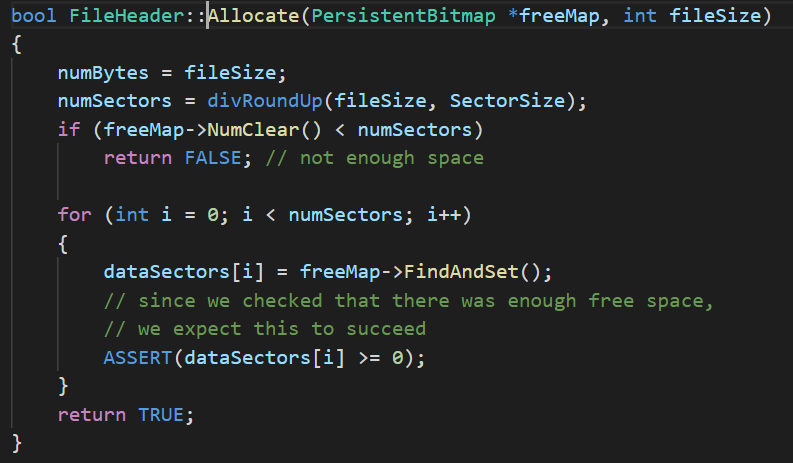
(4)

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

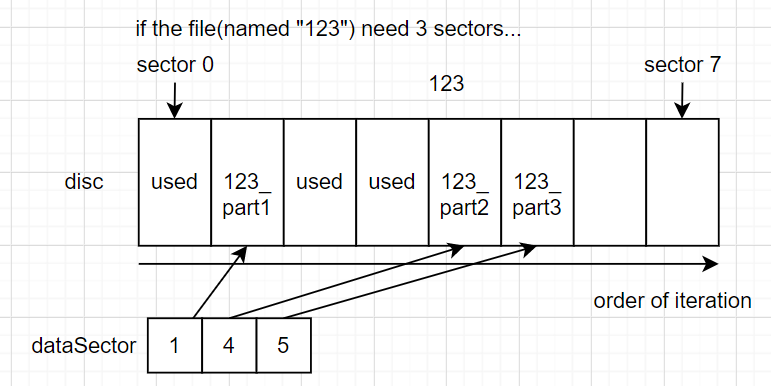
A:



From the current implementation, inode contains 3 elements. The first one is numBytes, which means how large is this file(bytes as its unit). The second one is numSectors, which means how many sectors are occupied by this file. The last one is dataSectors, which is an array storing 30 integers. It can be used to map a block to a corresponding sector.



As for the disc allocation scheme, we find out that this function handles it, and we use a figure as required to illustrate what’s going on if we call this function.

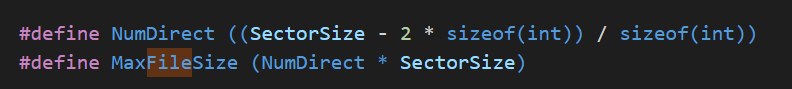


We will find spare sectors by iterating through the disc from the first sector to the last one. If we find a spare block, we store its number into the dataSector array. Therefore, we can load in a file by looking up the array, and get all of the corresponding sectors from our disc.

(5)

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

A:





4KB means 128Bytes\*32. Since SectorSize=128, there should be a maximum space (32Sectors) for each file.

In the above code, we can see the MaxFileSize is set to be “NumDirect \* SectorSize”, also SectorSize = 128 and NumDirect = (128 - 2\*4) / 4 = 30. Also, we use a sector to store file header, but we cannot find the usage of the last one sector in the code.

**實作的步驟濃縮版**

**Part II. Modify the file system code to support file I/O system call and larger file size**

**(30%)**

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

(2) Implement five system calls:

int **Create**(char \*name, int size);

OpenFileId **Open**(char \*name);

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

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

int **Close**(OpenFileId id);

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

Important: You ARE NOT allowed to change the sector size!!!

Verification

nachos -f (format the disc)

nachos -cp <file\_to\_be\_copied> <destination\_on\_NachOS\_FS>

* copy file from Linux to NachOS

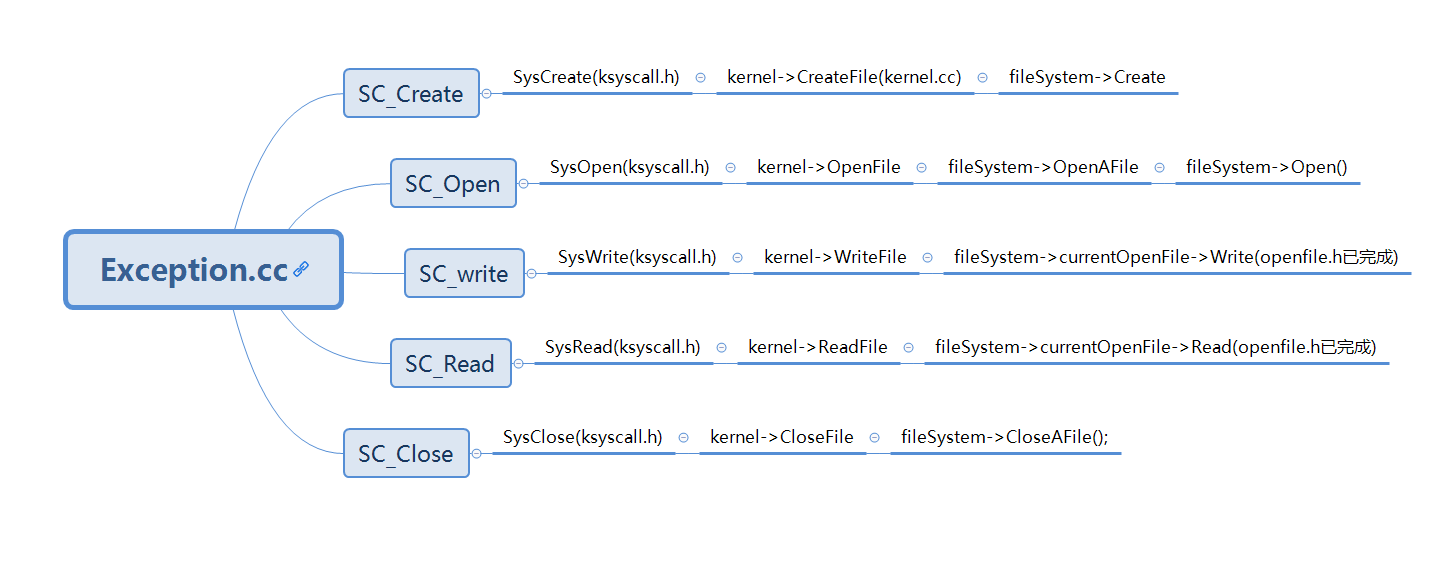
nachos -p <file\_to\_be\_dumped> (print content of a file in the disc)

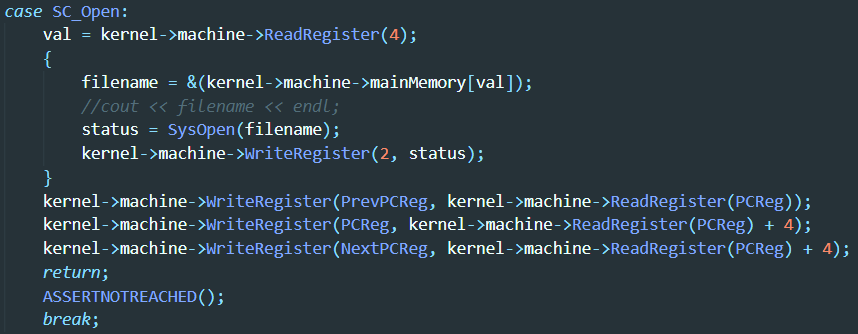
**Part III. Modify the file system code to support subdirectory**

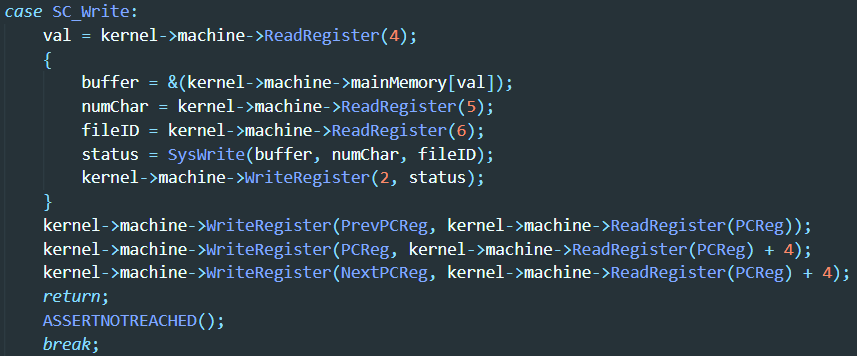
**(30%)**

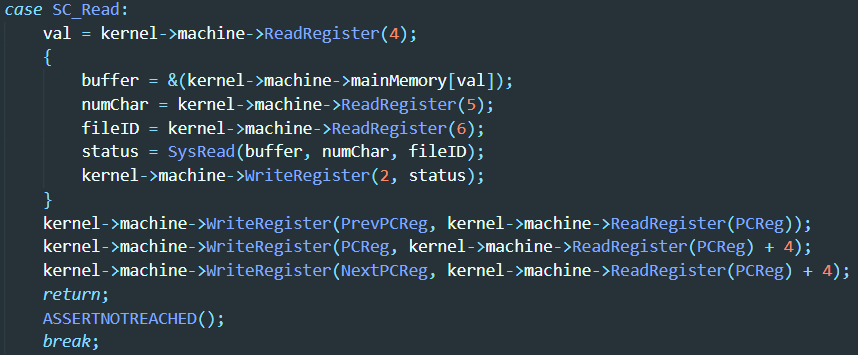
**Report**

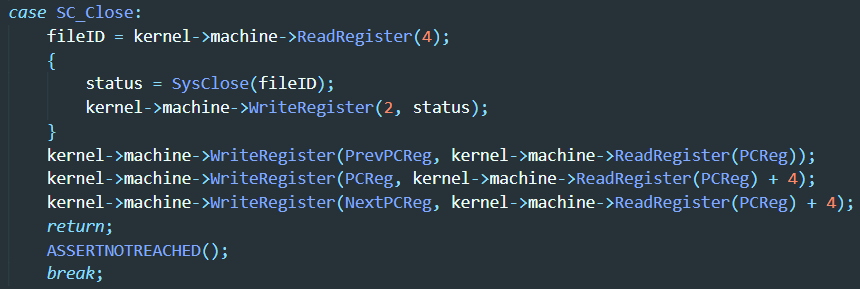
**1. SystemCall Implementation**

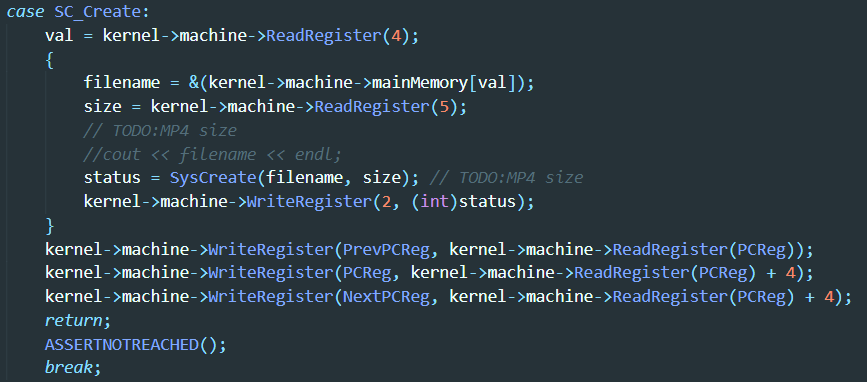
For this part, we use what we have implemented in MP1, and it works pretty well. The contents below are the structure of system calls and our code of system calls in exception.cc.****

****

****

****

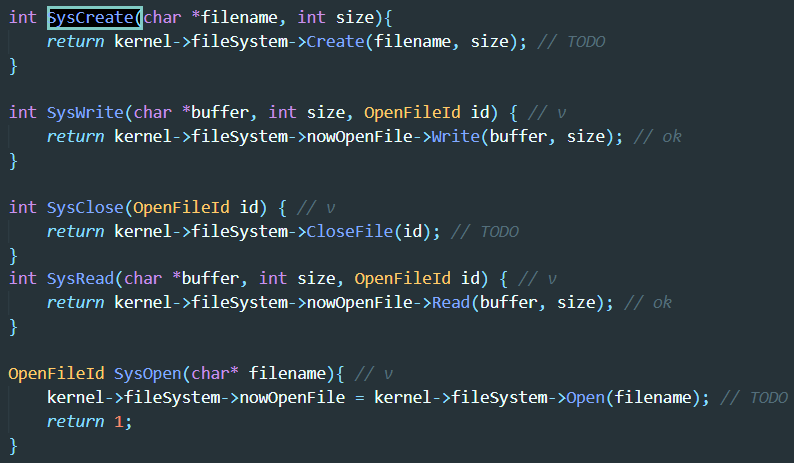
****

****

**a.** **Exeception.cc**

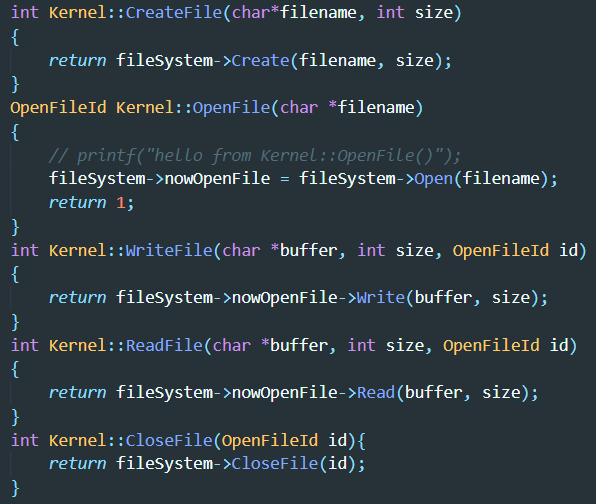
We add SC\_Create、SC\_Open、SC\_Read、SC\_Write、SC\_Close 5 functions, and link them to SysCreate、SysOpen、SysRead、SysWrite、SysClose 5 APIs.

**b.** **Ksyscall.h**

SysCreate、SysOpen、SysRead、SysWrite、SysClose are linked to CreateFile OpenFile, ReadFile, WriteFile, CloseFile in kernel.cc. 

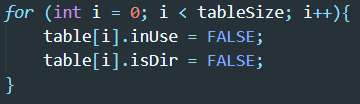
**c. kernel.cc**

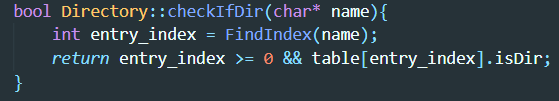
CreateFile, OpenFile, ReadFile, WriteFile, CloseFile are linked in Create, Open, Write, Read, CloseFile in filesys.cc.

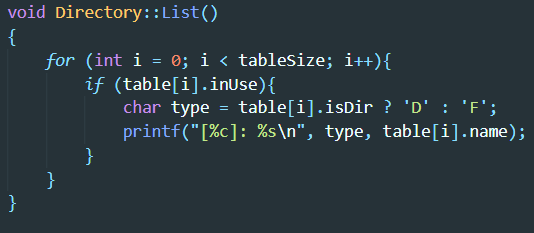


**directory.cc**

We create Isdir variable and checkIfDir to determine whether it is a directory or not .





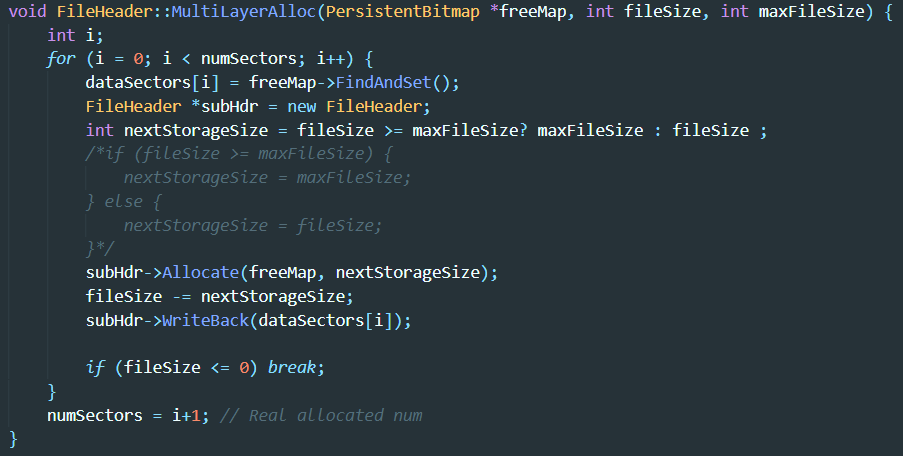


In List() function, we just simply list the directories in the table.

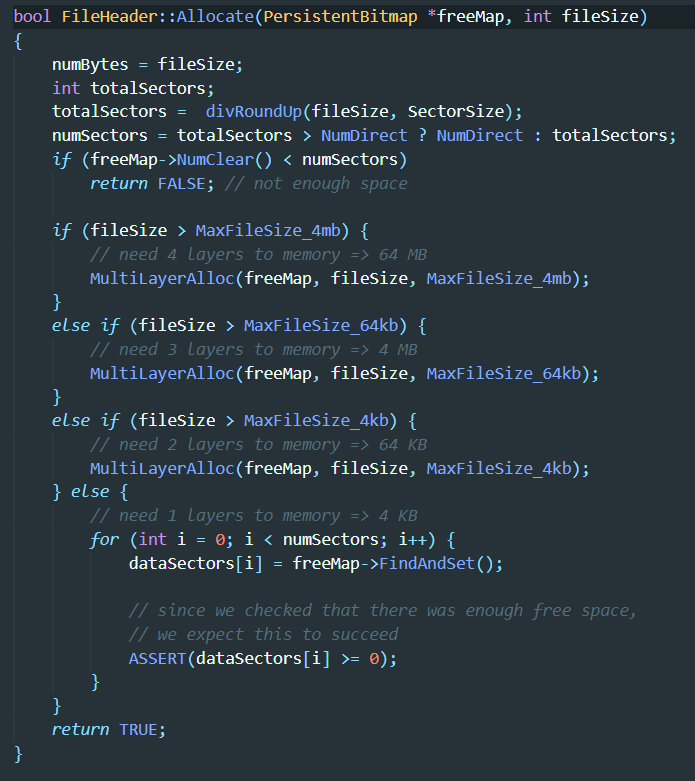


In Recursively list function, firstly we determine whether it is a directory or not ; then we print the number of tab according to its level, and finally we recursively enter into the next level of the directory.

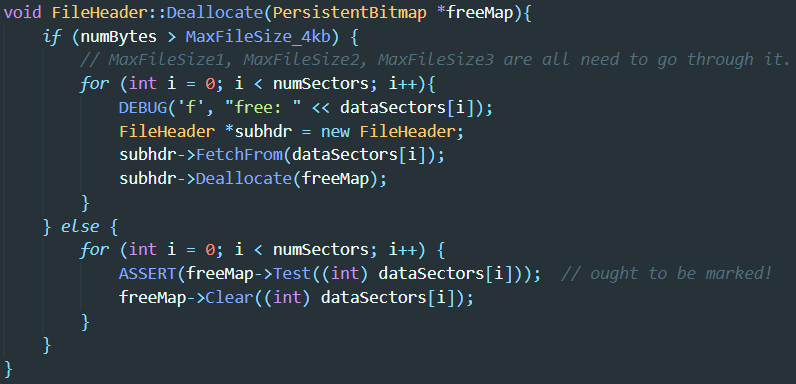
**filehdr.cc**

****

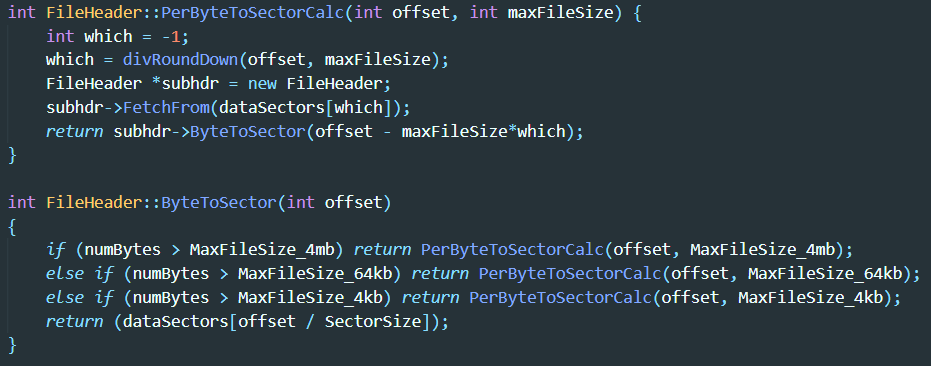
In MultiLayerAlloc function we have 3 parameters, freeMap for record the using condition fileSize for the size to allocate and maxFileSize to check if it exceeds the maximum allocate space. In the loop, we mark dataSectors[i] and allocates a new header file, and we set nextStorageSize to the minimum number of maxFileSize and fileSize. And we recursively allocate the data using MultiLayerAlloc and Allocate function, finally we write back to datasector[i] with what we have used. If fileSize <= 0, we have allocate enough space for the file.

****

In the Allocate function, we allocate the maximum space at the same level, as the same as MultiLayerAlloc, they recursively call the other function. At last, it writes back to the freemap.



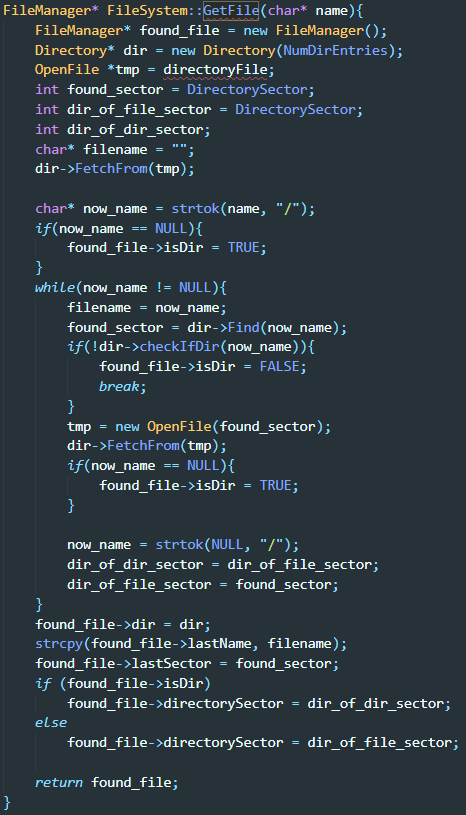
In Deallocate function, we determine what size should we clean for the freemap, if it exceeds 4KB then we read the datasectors[i] and recursively delete the data, if not, then we call Clear() function.



We create 2 recursively call PerByteToSectorCalc and ByteToSector to function to dynamically calculate data sectors with these two functions. We dividerounddown the offset with maxFileSize and create a subheader to fetch the information from data sectors, then we recursively call ByteToSector function to change the parameter of maxfilesize and finally return the data sectors[offset / sector size].

**filesys.cc**

In filesys.cc we create a function GetFile to get the file by iterating a linked list.



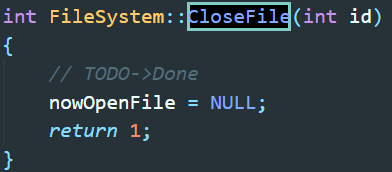
In this function, we allocate a new file manager and directory first, then we create a temporary file to record directory file. Later, we create 2 variables dir\_of\_file and dir\_of\_sector and we fetch(read) from the data from the directory. And we use now\_name to record the finally file name. In the while loop, we continuously check if the file is a directory, and if now\_name == NULL, then it is a directory. Then we can see that found\_file->dir = dir and we can also know that found\_file->lastName = found\_sector. If found\_file is a directory, then its directory sector is dir\_of\_dir\_sector else dir\_of\_file\_sector. At last, we return found\_file.



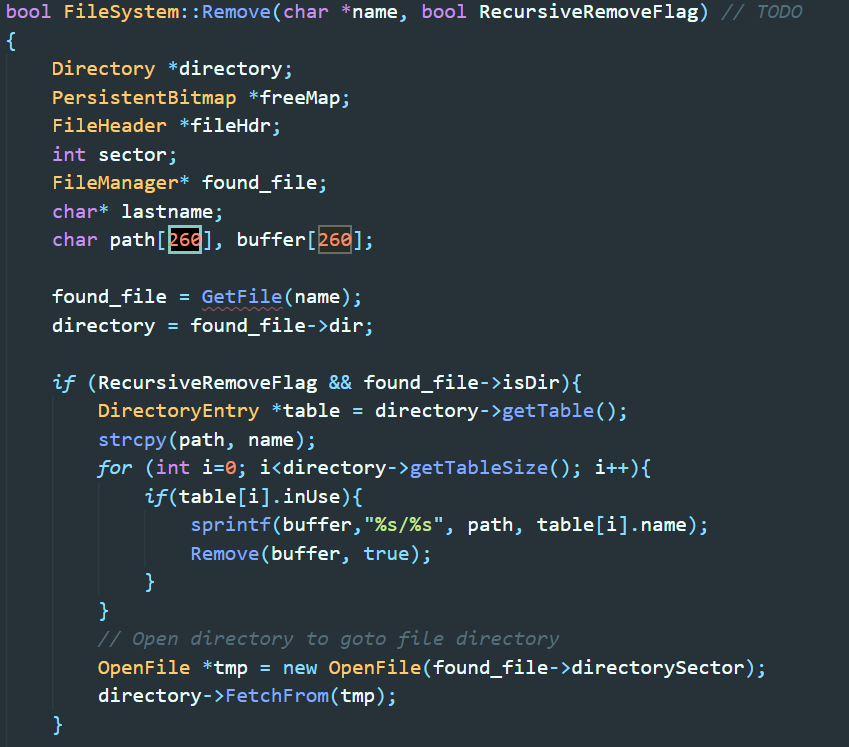
When creating a directory, we firstly get the file found\_file then we get its directory and its lastName. we create a freeMap using PersistentBitmap and newSector calling FindAndSet() function to find a sector to hold the file header. If not found, then success = FALSE; second, we link subdirectory to original directory; third, we build up a subdirectory to file head; fourth, we build up subdirectory; Fifth, we update the directory and free local storage.



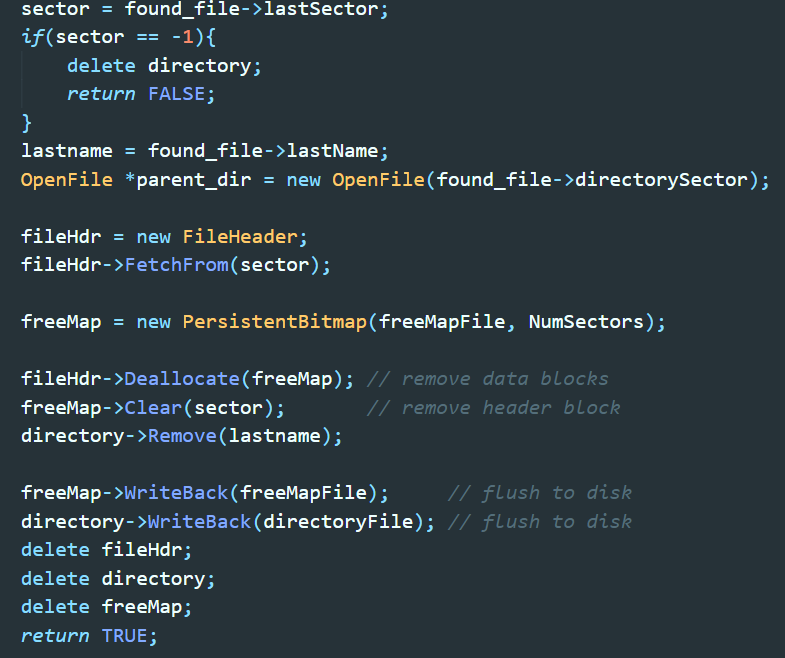
Create is similar to CreateDirectory.



CloseFile function just set newOpenFile to NULL.



In the first part of Remove function, found\_file = Get\_file(name), dir of found\_file is directory, if found\_file is a directory and RecursiveRemoveFlag = true, then we keep getting table of the directory and put the path and table name into buffer. And we open a new OpenFile called tmp and we fetch information from it, and



we set a tmp sector to found\_file->last sector, if sector == -1, return false; last name is equal to found\_file’s last file name. We create a file header and fetch some information and create a new freemap. And we deallocate and free it. Finally, we remove the data blocks and header blocks; We flush to disk with two files.

**Bonus**

iLMS提醒

| 這次作業需要改nachos的file system，因為已經拿掉build.linux/Makefile裡的-DFILESYS\_STUB，  現在nachos開檔、執行檔案的file system已經是由nachos自己管理了（內容物存在DISK\_0 檔案）  而非在linux的file system（例如 在test資料夾 下“ls”後看到的檔案）  所以同學想要測試時，必須先將binary複製入nachos的Filesystem  透過 **nachos -cp <file\_to\_be\_copied> <destination\_on\_NachOS\_FS>**  例如，若想測試的檔案是fileIO\_test1，那就必須下：**../build.linux/nachos -cp fileIO\_test1 /fileIO\_test1**  然後這時才能執行  **../build.linux/nachos -e /fileIO\_test1**  若無複製入nachos filesystem的話，會出現Unable to open file fileIO\_test1的error  新增測資要記得修改test下的Makefile  另外  /hone/os2020/share/NachOS-4.0\_MP4/code/test 內提供三個script方便大家測試。  (測資內只跑測資的make，若有修改nachos，還是需要到build.linux重新make)  使用方法：  ./FS\_partII\_a.sh  ./FS\_partII\_b.sh  ./FS\_partIII.sh  若同學想demo bonusII, 請準備測資與證明實作成功的方法。  filesys.cc  FileSystem::FileSystem(bool format)(X) //  FileManager\* FileSystem::GetFile(char\* name) //  bool FileSystem::CreateDirectory(char \*name) //  OpenFile \* FileSystem::Open(char \*name)(X) //  int FileSystem::CloseFile(int id)(X) //  bool FileSystem::Remove(char \*name, bool RecursiveRemoveFlag) //  void FileSystem::List(char \*listDirectoryName, bool RecursiveListFlag)(X)  FileManager//  main.cc 97 //  kernel.h order //  int Kernel::CreateFile(char \*filename) //  int FileHeader::PerByteToSectorCalc(int offset, int maxFileSize) QQ不能改  void FileHeader::Deallocate(PersistentBitmap \*freeMap) //  void FileHeader::MultiLayerAlloc(PersistentBitmap \*freeMap, int fileSize, int maxFileSize) //  bool Directory::Remove(char \*name)  \/\\*[\s\S]\*\\*\/|\/\/.\* |
| --- |