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:

```
#define FreeMapSector 0
#define DirectorySector 1
```

sector0(file header of bitmap)

(2)

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

A:

```
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);
```

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:

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

```
class DirectoryEntry
{
public:
    bool inUse;
    int sector;

    char name[FileNameMaxLen + 1];

};
```

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*.

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

    // MP4 mod tag
    memset(table, 0, sizeof(DirectoryEntry) * size);

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

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

```
void Directory::FetchFrom(OpenFile *file)
{
    (void)file->ReadAt((char *)table, tableSize * sizeof(DirectoryEntry), 0);
}
```

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

```
void Directory::WriteBack(OpenFile *file)
{
    (void)file->WriteAt((char *)table, tableSize * sizeof(DirectoryEntry), 0);
}
```

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

```
int Directory::FindIndex(char *name)
{
    for (int i = 0; i < tableSize; i++)
        if (table[i].inUse && !strncmp(table[i].name, name, FileNameMaxLen))
            return i;
    return -1; // name not in directory
}
```

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.

```
int Directory::Find(char *name)
{
    int i = FindIndex(name);

    if (i != -1)
        return table[i].sector;
    return -1;
}
```

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.

```
bool Directory::Add(char *name, int newSector)
{
    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;
            return TRUE;
        }
    return FALSE; // no space.  Fix when we have extensibl
}
```

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.

```
bool Directory::Remove(char *name)
{
    int i = FindIndex(name);

    if (i == -1)
        return FALSE; // name not in directory
    table[i].inUse = FALSE;
    return TRUE;
}
```

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.

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

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.

```
void Directory::Print()
{
    FileHeader *hdr = new FileHeader;

    printf("Directory contents:\n");
    for (int i = 0; i < tableSize; i++)
        if (table[i].inUse)
        {
            printf("Name: %s, Sector: %d\n", table[i].name, table[i].sector);
            hdr->FetchFrom(table[i].sector);
            hdr->Print();
        }
    printf("\n");
    delete hdr;
}
```

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:

```
#define FreeMapSector 0
#define DirectorySector 1
```

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:

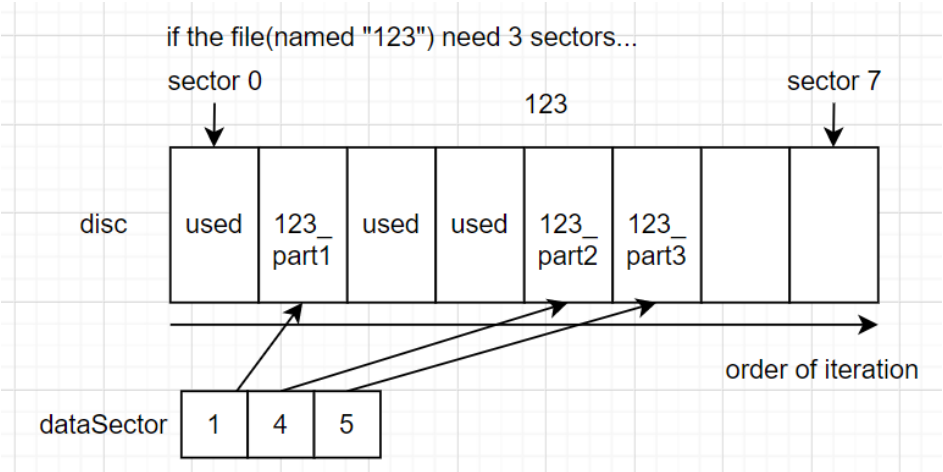
```
    int numBytes;              // Number of bytes in the file
    int numSectors;            // Number of data sectors in the file
    int dataSectors[NumDirect]; // Disk sector numbers for each data
                               // block in the file
```

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.

```
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;
}
```

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:

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

```
50    const int SectorSize = 128;   // number of bytes per disk sector
```

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

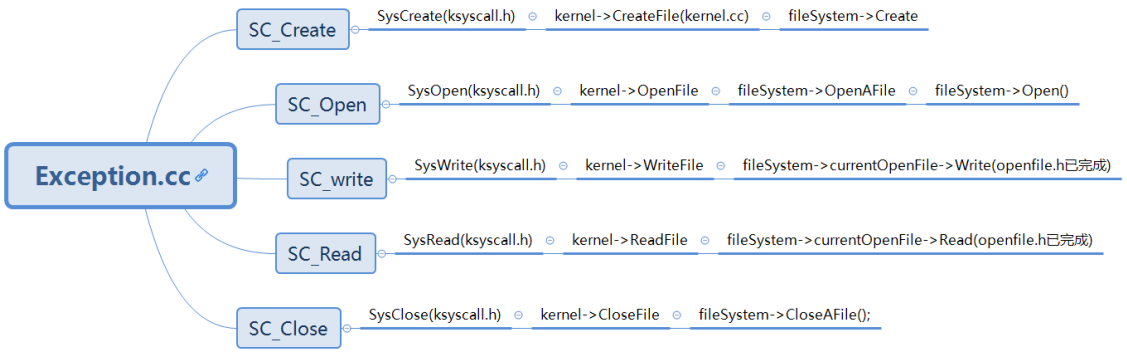
nachos -p <file_to_be_dumped> (print content of a file in the disc)

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.



```
case SC_Open:
    val = kernel->machine->ReadRegister(4);
    {
        filename = &(kernel->machine->mainMemory[val]);
        //cout << filename << endl;
        status = SysOpen(filename);
        kernel->machine->WriteRegister(2, 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_Write:
    val = kernel->machine->ReadRegister(4);
    {
        buffer = &(kernel->machine->mainMemory[val]);
        numChar = kernel->machine->ReadRegister(5);
        fileID = kernel->machine->ReadRegister(6);
        status = SysWrite(buffer, numChar, fileID);
        kernel->machine->WriteRegister(2, 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_Read:
    val = kernel->machine->ReadRegister(4);
    {
        buffer = &(kernel->machine->mainMemory[val]);
        numChar = kernel->machine->ReadRegister(5);
        fileID = kernel->machine->ReadRegister(6);
        status = SysRead(buffer, numChar, fileID);
        kernel->machine->WriteRegister(2, 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_Close:
    fileID = kernel->machine->ReadRegister(4);
    {
        status = SysClose(fileID);
        kernel->machine->WriteRegister(2, 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_Create:
    val = kernel->machine->ReadRegister(4);
    {
        filename = &(kernel->machine->mainMemory[val]);
        size = kernel->machine->ReadRegister(5);
        // TODO:MP4 size
        //cout << filename << endl;
        status = SysCreate(filename, size); // TODO:MP4 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;
```

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.

```
int SysCreate(char *filename, int size){
    return kernel->fileSystem->Create(filename, size); // TODO
}

int SysWrite(char *buffer, int size, OpenFileId id) { // v
    return kernel->fileSystem->nowOpenFile->Write(buffer, size); // ok
}

int SysClose(OpenFileId id) { // v
    return kernel->fileSystem->CloseFile(id); // TODO
}
int SysRead(char *buffer, int size, OpenFileId id) { // v
    return kernel->fileSystem->nowOpenFile->Read(buffer, size); // ok
}

OpenFileId SysOpen(char* filename){ // v
    kernel->fileSystem->nowOpenFile = kernel->fileSystem->Open(filename); // TODO
    return 1;
}
```

c. kernel.cc

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

```
int Kernel::CreateFile(char*filename, int size)
{
    return fileSystem->Create(filename, size);
}
OpenFileId Kernel::OpenFile(char *filename)
{
    // printf("hello from Kernel::OpenFile()");
    fileSystem->nowOpenFile = fileSystem->Open(filename);
    return 1;
}
int Kernel::WriteFile(char *buffer, int size, OpenFileId id)
{
    return fileSystem->nowOpenFile->Write(buffer, size);
}
int Kernel::ReadFile(char *buffer, int size, OpenFileId id)
{
    return fileSystem->nowOpenFile->Read(buffer, size);
}
int Kernel::CloseFile(OpenFileId id){
    return fileSystem->CloseFile(id);
}
```

directory.cc

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

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

```
bool Directory::checkIfDir(char* name){
    int entry_index = FindIndex(name);
    return entry_index >= 0 && table[entry_index].isDir;
}
```

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

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

```
void Directory::RecursiveList(int level) {
    Directory *subDirectory = new Directory(NumDirEntries);
    OpenFile *tempOpenFile;

    for (int i = 0; i < tableSize; i++) {
        if (table[i].inUse) {
            char type = table[i].isDir ? 'D' : 'F';
            for (int i = 0; i < level; i++) {
                printf("\t");
            }
            printf("[%c]: %s\n", type, table[i].name);
            if (table[i].isDir) {
                tempOpenFile = new OpenFile(table[i].sector);
                subDirectory->FetchFrom(tempOpenFile);
                subDirectory->RecursiveList(level+1);
            }
        }

    }
}
```

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

```
void FileHeader::MultiLayerAlloc(PersistentBitmap *freeMap, int fileSize, int maxFileSize) {
    int i;
    for (i = 0; i < numSectors; i++) {
        dataSectors[i] = freeMap->FindAndSet();
        FileHeader *subHdr = new FileHeader;
        int nextStorageSize = fileSize >= maxFileSize? maxFileSize : fileSize ;
        /*if (fileSize >= maxFileSize) {
            nextStorageSize = maxFileSize;
        } else {
            nextStorageSize = fileSize;
        }*/
        subHdr->Allocate(freeMap, nextStorageSize);
        fileSize -= nextStorageSize;
        subHdr->WriteBack(dataSectors[i]);

        if (fileSize <= 0) break;
    }
    numSectors = i+1; // Real allocated num
}
```

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.

```
bool FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
{
    numBytes = fileSize;
    int totalSectors;
    totalSectors =  divRoundUp(fileSize, SectorSize);
    numSectors = totalSectors > NumDirect ? NumDirect : totalSectors;
    if (freeMap->NumClear() < numSectors)
        return FALSE; // not enough space

    if (fileSize > MaxFileSize_4mb) {
        // need 4 layers to memory => 64 MB
        MultiLayerAlloc(freeMap, fileSize, MaxFileSize_4mb);
    }
    else if (fileSize > MaxFileSize_64kb) {
        // need 3 layers to memory => 4 MB
        MultiLayerAlloc(freeMap, fileSize, MaxFileSize_64kb);
    }
    else if (fileSize > MaxFileSize_4kb) {
        // need 2 layers to memory => 64 KB
        MultiLayerAlloc(freeMap, fileSize, MaxFileSize_4kb);
    } else {
        // need 1 layers to memory => 4 KB
        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;
}
```

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.

```
void FileHeader::Deallocate(PersistentBitmap *freeMap){
    if (numBytes > MaxFileSize_4kb) {
        // MaxFileSize1, MaxFileSize2, MaxFileSize3 are all need to go through it.
        for (int i = 0; i < numSectors; i++){
            DEBUG('f', "free: " << dataSectors[i]);
            FileHeader *subhdr = new FileHeader;
            subhdr->FetchFrom(dataSectors[i]);
            subhdr->Deallocate(freeMap);
        }
    } else {
        for (int i = 0; i < numSectors; i++) {
            ASSERT(freeMap->Test((int) dataSectors[i]));  // ought to be marked!
            freeMap->Clear((int) dataSectors[i]);
        }
    }
}
```

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.

```
int FileHeader::PerByteToSectorCalc(int offset, int maxFileSize) {
    int which = -1;
    which = divRoundDown(offset, maxFileSize);
    FileHeader *subhdr = new FileHeader;
    subhdr->FetchFrom(dataSectors[which]);
    return subhdr->ByteToSector(offset - maxFileSize*which);
}

int FileHeader::ByteToSector(int offset)
{
    if (numBytes > MaxFileSize_4mb) return PerByteToSectorCalc(offset, MaxFileSize_4mb);
    else if (numBytes > MaxFileSize_64kb) return PerByteToSectorCalc(offset, MaxFileSize_64kb);
    else if (numBytes > MaxFileSize_4kb) return PerByteToSectorCalc(offset, MaxFileSize_4kb);
    return (dataSectors[offset / SectorSize]);
}
```

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.

```
FileManager* FileSystem::GetFile(char* name){
    FileManager* found_file = new FileManager();
    Directory* dir = new Directory(NumDirEntries);
    OpenFile *tmp = directoryFile;
    int found_sector = DirectorySector;
    int dir_of_file_sector = DirectorySector;
    int dir_of_dir_sector;
    char* filename = "";
    dir->FetchFrom(tmp);

    char* now_name = strtok(name, "/");
    if(now_name == NULL){
        found_file->isDir = TRUE;
    }
    while(now_name != NULL){
        filename = now_name;
        found_sector = dir->Find(now_name);
        if(!dir->checkIfDir(now_name)){
            found_file->isDir = FALSE;
            break;
        }
        tmp = new OpenFile(found_sector);
        dir->FetchFrom(tmp);
        if(now_name == NULL){
            found_file->isDir = TRUE;
        }

        now_name = strtok(NULL, "/");
        dir_of_dir_sector = dir_of_file_sector;
        dir_of_file_sector = found_sector;
    }
    found_file->dir = dir;
    strcpy(found_file->lastName, filename);
    found_file->lastSector = found_sector;
    if (found_file->isDir)
        found_file->directorySector = dir_of_dir_sector;
    else
        found_file->directorySector = dir_of_file_sector;

    return found_file;
}
```

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.

```
bool FileSystem::CreateDirectory(char *name) {
    FileManager *found_file;
    Directory *directory;
    PersistentBitmap *freeMap;
    FileHeader *dirHdr = new FileHeader;
    int newSector;
    bool success = true;
    char *pch;

    // Get the root directory first && the filename in path (e.g. /a/b.png  => b.png)
    found_file = GetFile(name);
    directory = found_file->dir;
    pch = found_file->lastName;
    OpenFile *parent_dir = new OpenFile(found_file->directorySector);

    // Out from while loop, which means we're going to construct subDirectory
    // 1. Find free sector
    freeMap = new PersistentBitmap(freeMapFile, NumSectors);
    newSector = freeMap->FindAndSet();  // find a sector to hold the file header
    if (newSector == -1) success = FALSE;

    // 2. Link subDirectory to original directory
    if (!directory->Add(pch, newSector, true)) success = FALSE;

    // 3. Build up subDirectory <File Header>
    dirHdr->Allocate(freeMap, DirectoryFileSize);
    dirHdr->WriteBack(newSector);

    // 4. Build up subDirectory <Directory>
    Directory *subDirectory = new Directory(NumDirEntries);
    OpenFile* newDirectoryFile = new OpenFile(newSector);
    subDirectory->WriteBack(newDirectoryFile);

    // 5. Update directory / freeMap on disk
    directory->WriteBack(parent_dir);
    freeMap->WriteBack(freeMapFile);

    // 6. Free local storage
    delete directory;
    delete subDirectory;
    delete freeMap;
    delete dirHdr;
    delete newDirectoryFile;

    return success;
}
```

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.

```
bool FileSystem::Create(char *name, int initialSize)
{
    FileManager* found_file;
    Directory *directory;
    PersistentBitmap *freeMap;
    FileHeader *hdr;

    char *lastname;
    int sector;
    bool success;

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

    found_file = GetFile(name);
    directory = found_file->dir;
    lastname = found_file->lastName;
    OpenFile* parent_dir = new OpenFile(found_file->directorySector);
    if (directory->Find(lastname) != -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(lastname, sector, false))//same as name
            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(parent_dir);
                freeMap->WriteBack(freeMapFile);
            }
            delete hdr;
        }
        delete freeMap;
    }
    delete directory;
    return success;
}
```

Create is similar to CreateDirectory.

```
int FileSystem::CloseFile(int id)
{
    // TODO->Done
    nowOpenFile = NULL;
    return 1;
}
```

CloseFile function just set newOpenFile to NULL.

```
bool FileSystem::Remove(char *name, bool RecursiveRemoveFlag) // TODO
{
    Directory *directory;
    PersistentBitmap *freeMap;
    FileHeader *fileHdr;
    int sector;
    FileManager* found_file;
    char* lastname;
    char path[260], buffer[260];

    found_file = GetFile(name);
    directory = found_file->dir;

    if (RecursiveRemoveFlag && found_file->isDir){
        DirectoryEntry *table = directory->getTable();
        strcpy(path, name);
        for (int i=0; i<directory->getTableSize(); i++){
            if(table[i].inUse){
                sprintf(buffer,"%s/%s", path, table[i].name);
                Remove(buffer, true);
            }
        }
        // Open directory to goto file directory
        OpenFile *tmp = new OpenFile(found_file->directorySector);
        directory->FetchFrom(tmp);
    }
```

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

```
 sector = found_file->lastSector;
 if(sector == -1){
     delete directory;
     return FALSE;
 }
 lastname = found_file->lastName;
 OpenFile *parent_dir = new OpenFile(found_file->directorySector);

 fileHdr = new FileHeader;
 fileHdr->FetchFrom(sector);

 freeMap = new PersistentBitmap(freeMapFile, NumSectors);

 fileHdr->Deallocate(freeMap); // remove data blocks
 freeMap->Clear(sector);       // remove header block
 directory->Remove(lastname);

 freeMap->WriteBack(freeMapFile);     // flush to disk
 directory->WriteBack(directoryFile); // flush to disk
 delete fileHdr;
 delete directory;
 delete freeMap;
 return TRUE;
```

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]**\/|\/\/.*