113-1 Operating System MP4

Group 45: 111062332 朱誼學, 111062333 高英耀

Contributions	**111062332**	**111062333**
Trace code	50%	50%
Implementation	50%	50%
Report	50%	50%
Explanation	We've done all our work in discord vc	

Part I - Understanding NachOS file system

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

1. In `FileSystem::FileSystem(bool format)`, we can see that there are two `OpenFile` instances called `freeMapFile` and `directoryFile`. These two `OpenFile`s save the data of free block space and directory information.
    - The codes below are going to be executed if the file system has already been initialized.
    - After knowing that we can get the free block space information from the file saved in `FreeMapSector`, we can trace the part how NachOS manage and find free block space.

```
// 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);
```

2. How to manage the free block spaces:
    - `*freeMap` is a pointer of `PersistentBitMap`, which inherits `BitMap`. So basically, it's managing the sector resources with `BitMap`. So, the total number of bits will be the same with number of sectors. Noted that in `BitMap`, every 32 bits are grouped into an integer, so it stores 1024 sectors using 32 integers.
    - That's also why the file header of `freeMapFile` needs to ask for allocating a space with size `FreeMapFileSize = NumSectors/bitsInByte (bytes)` -> To store the status of free blocks (with 32 integers)

```
// In FileSystem::FileSystem(bool format)
// Turn NumSectors (bits) into bytes
ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));

// In FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize)
// SectorSize is defined in bytes.
// So we have to pass the fileSize with bytes too.
numSectors = divRoundUp(fileSize, SectorSize);
```

3. How to find a free block space:
    - In `FileHeader::Allocate()`, a loop will get id of empty sectors and save into `FileHeader::dataSectors[]`.
    - And the way to get empty sector number is by calling `BitMap::FindAndSet()` iteratively.
    - As for the detail of `BitMap::FindAndSet()`, it will call `Test()` and `Mark()` inside the loop, where `Test()` is for checking whether the sector is occupied, and `Mark()` is for setting the sector to be occupied.

```
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;
}

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

4. Where is this information stored on the raw disk (which sector)?
    - For the conclusion, the `FileHeader` of `freeMapFile` will stores at `sector 0`, and the actual `freeMap` data is stored at `sector 2`. Following are the explanations.
    - Let's take a look at `FileSystem::FileSystem()` row by row. But only the part related to free sector management.
    - First, a `PersistentBitMap *freeMap` will be instantiated. And this is the structure which stores the free sector information.
    - Next, a `FileHeader *mapHdr` will be instantiate. This class stores the information of a file.
    - freeMap will first reserve a space for storing the `FileHeader` itself. At this moment, sector `FreeMapSector` (0) is occupied by the free map file header.
    - The next thing to do is, mapHdr will request an allocation for a space for storing the status of sectors. Which sector(s)?

        - Since `sector 0` and `sector 1` have already been used by `freeMapFileHeader` and `directoryFileHeader`, sectors allocated to `freeMapFile` data starts from `sector 2`.
        - With some computation, there are 1024 sectors and stored with 32 integers, so it needs total 1024 / 8 or 32 * 4 = 128 bytes. (Both statements are correct), we can know that `sector 2` is the only sector allocated to `freeMapFile`, because a `sectorSize` is exactly 128 bytes.
        - To validate our computation, let's trace the rest of the codes.
    - Next, since the header has updated all the data, it's time to flush it back to disk.
    - Later, open the file from `FreeMapSector` and return a `OpenFile` instance. `freeMapFile->hdr` is exactly the `mapHdr` we've mentioned before.
    - Last, flush the bitmap changes into disk by `WriteBack()` to `freeMapFile` (at `sector 2`).

```
FileSystem::FileSystem(bool format)
{
    if (format){
        PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);
        Directory *directory = new Directory(NumDirEntries);
        FileHeader *mapHdr = new FileHeader;
        FileHeader *dirHdr = new FileHeader;

        // 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
        mapHdr->WriteBack(FreeMapSector);
        dirHdr->WriteBack(DirectorySector);

        // OK to open the bitmap and directory files now
        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.
        freeMap->WriteBack(freeMapFile);
        directory->WriteBack(directoryFile);

    }
}
```

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

```
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
```

Disk size = `SectorSize` * `NumSectors` = `SectorSize` * (`SectorPerTrack` * `NumTracks`) = 128 * 32 * 32 = 128 KB

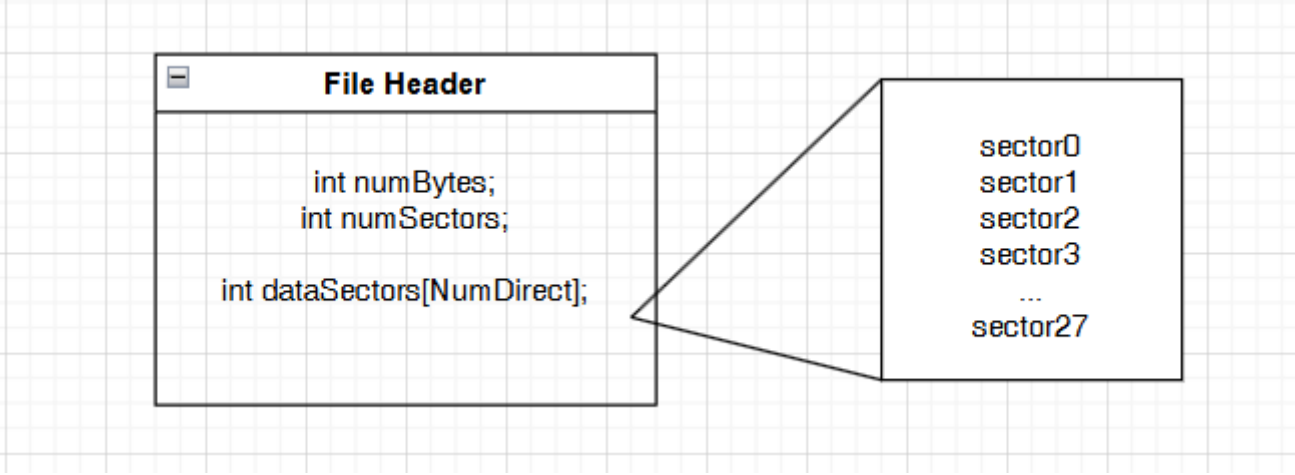
(3) How does the NachOS FS manage the directory data structure? Where is this information stored on the raw disk (which sector)?

1. Similar with the first question, a file `directoryFile` stores the file names.
2. How to manage the directory data structure.
    - Using `Directory` and `DirectoryEntry`
    - In a `Directory` instance, it consist of `int tableSize` and `DirectoryEntry *table`.
    - Each `DirectoryEntry` has a `inUse` boolean value. Besides, an entry gives the name of the file, and where the file's header is to be found on disk (which sector).
    - What if a file is created or is opened?
        - `FileSystem::Create()`, checks whether `file is already in directory`, `any free block for file header`, `any spaces in directory`. If every works successfully, flush all changes back to disk (includes new file header, directory file and free map file)
        - `FileSystem::Open()`, fetch the directory table from `directoryFile` and find the corresponding sector number in the table by filename. If founded (sector >= 0), return an `OpenFile` instance.
3. Where is this information stored on the raw disk (which sector)?
    - Tell the answer first, the `directoryFileHeader` is stored at `sector 1` and `directoryFile` data are stored at `sector 3` and `sector 4`. Following are the explanations.
    - Since Header and File of free map takes away 0 and 2, and 1 is for the `directoryFileHeader` due to `DirectorySector == 1`.
    - Next we are doing some computation for the sector numbers of `directoryFile`.
        - How many and which sectors are allocated to `directoryFileHeader` when directory is initialized?
        - Since `directoryFileHeader` asked for a `DirecotoryFileSize` space, which is `sizeof(DirectoryEntry) * NumDirEntries` = 20 * 10 = 200 bytes.
        - 200 bytes needs 2 sectors, which is `sector 3` and `sector 4`.

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

1. An inode is also known as a File Control Block (FCB). In NachOS, inode is `FileHeader`.
2. As we can see from the class structure, it contains 2 integers and 1 int array.

```
#define NumDirect ((SectorSize - 2 * sizeof(int)) / sizeof(int))
#define MaxFileSize (NumDirect * SectorSize)
class FileHeader{
    int numBytes;
    int numSectors;
    int dataSectors[NumDirect];
}
```



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

1. Following the previous question.
2. A file header costs 128 bytes (a sector size), so after storing 2 integers, there is only 120 bytes for the array, that is 120/4=30 integers in total.
3. So, `NumDirect` is 30 and `MaxFileSize` is 30 * 128 = 3840 bytes = 3.75 KB

Part II - Modify the file system code to support file I/O system calls and larger file size

We will explain our implementation by introducing what we've done to solve each tasks.

(1) Combine your MP1 file system call interface with NachOS FS to implement five system calls:

1. `int Create(char *name, int size)`:
    - Since the original `Create()` function has return value `bool`, so we simply changed the return type into `int`.
    - The content in this function stays the same, so we don't show the code here.
    - At this step, this change is already enough for partII-a.
2. `OpenFileId Open(char *name)`
    - Since there's already an `OpenFile *Open(char *name)` defined in `FileSystem` and are widely used, we defined a new function `OpenFileId OpenForFileId(char *name)` to solve this task.
    - In `OpenForFileId()`, it inferences `OpenFile *Open(char *name)` to get a `OpenFile` instance
    - And because we don't have to maintain OpenFileTable in this assignment, we can just return am arbitary id greater than 0. But we still have a `openfile` pointer for saving the opened file.

```
OpenFileId FileSystem::OpenForFileId(char *name){
    OpenFile* f = Open(name);
    int id = 1;
    openfile = f;
    return (OpenFileId) id;
}
```

3. `int Read(char *buf, int size, OpenFileId id)`
    - Get the opened file from `openfile`, and `OpenFile::Read()` to complete reset of the works.

```
int FileSystem::Read(char *buf, int size, OpenFileId id){
    OpenFile* f = openfile;
    if(f) return f->Read(buf, size);
    else return -1;
}
```

4. `int Write(char *buf, int size, OpenFileId id)`
    - Same as `Read()`

```
int FileSystem::Write(char *buf, int size, OpenFileId id){
    OpenFile* f = openfile;
    if(f) return f->Write(buf, size);
    else return -1;
}
```

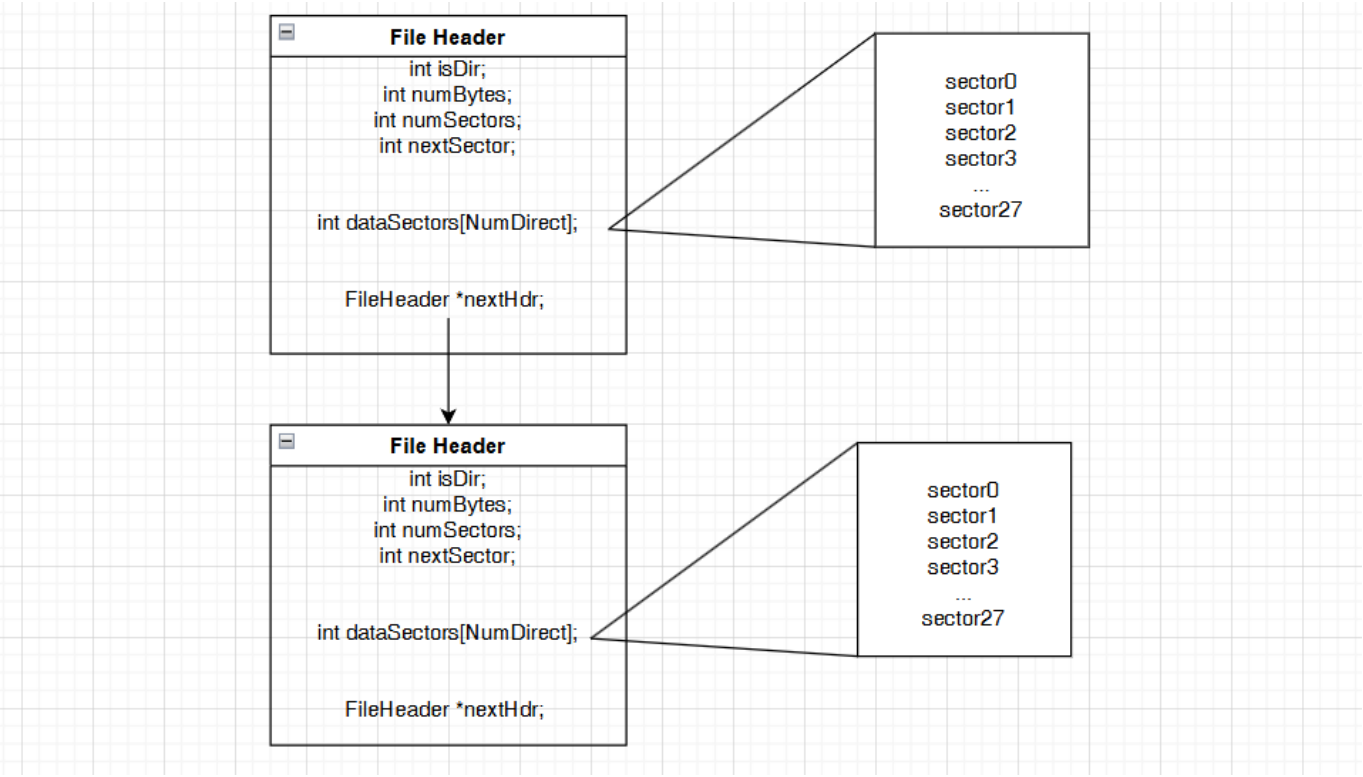
5. `int Close(OpenFileId id)`
    - Delete `openfile`, and simply return 1 (for success).

```
int FileSystem::Close(OpenFileId id){
    delete openfile;

    return 1;
}
```

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

1. To solve this question, we have to recall why the file size is restricted.
    - Because of the `FileHeader` is saved in one sector, so the number of allocated sectors is limited.
2. We extend the file size by using `Linked Index` approach.
3. For this approach, we have to adjust the structure of `FileHeader`.
    - Adding two On-disk data `nextSector` to stored the file header sector of the next linked node and `isDir` is used to marked the header as a directory file header or not (will be useful later).
    - And one in-core data `nextHdr` to improve the convenience when traversing the linked-index header list.
    - Due to the change of on disk data, we need to deduct `NumDirect` to make the total size still 128 bytes (sector size). For now, NumDirect changes from 30 to 28.

```
    int isDir;
    int numBytes;
    int numSectors;
    int nextSector;
    int dataSectors[NumDirect];

    FileHeader *nextHdr; // in-core
```



4. After that, we have to modify the related functions:
    - `FileHeader::FileHeader()`, initialization, nothing special
    - `FileHeader::~FileHeader()`, release the memory use of `FileHeader *nextHdr`
    - `FileHeader::Allocate(PersistentBitMap *, int)`, recursively allocate the sector.
        - If the `fileSize` is greater than `MaxFileSize` which is 3.75KB theoretically, allocate sectors with `MaxFileSize` = 28 sectors to the current header. If not, same as it used to be.
        - Next, since `fileSize` is originally greater than `MaxFileSize`, we have to extend the linked structure.
        - `FindAndSet()` a sector for `nextHdr` at the first, then recursively called `Allocate()` to comlete the allocation.
    - `FileHeader::Deallocate(PersistentBitMap *)`, recursively call `Deallocate()` for each file header.
    - `FileHeader::FetchFrom(int)`, read the on disk part data by using a 128 bytes buffer. And if `nextSector != -1`, recursively fetch data of `nextHdr`.
    - `FileHeader::WriteBack(int)`, similar to `FetchFrom()`, using buffer to identify the part is flushing back to disk.
    - `FileHeader::ByteToSector(int)`, recursively distribute the data size to the linked headers by passing the part `offset` exceeds `MaxFileSize` to the next header.

- Here is the code for `Allocate()` and `Deallocate()`

```
bool FileHeader::Allocate(PersistentBitmap *freeMap, int fileSize){
    numBytes = fileSize < MaxFileSize ? fileSize : MaxFileSize;
    fileSize -= numBytes;
    numSectors = divRoundUp(numBytes, SectorSize);

    if (freeMap->NumClear() < numSectors)
        return FALSE; // not enough space
    for (int i = 0; i < numSectors; i++){
        dataSectors[i] = freeMap->FindAndSet();
        ASSERT(dataSectors[i] >= 0);
    }
    if(fileSize > 0){
        nextSector = freeMap->FindAndSet();
        if(nextSector == -1)
            return FALSE;
        else{
            nextHdr = new FileHeader;
            nextHdr->Allocate(freeMap, fileSize);
        }
    }
    return TRUE;
}

void FileHeader::Deallocate(PersistentBitmap *freeMap){
    for (int i = 0; i < numSectors; i++){
        ASSERT(freeMap->Test((int)dataSectors[i])); // ought to be marked!
        freeMap->Clear((int)dataSectors[i]);
    }
    if(nextSector != -1){
        nextHdr->Deallocate(freeMap);
    }
}
```

- Here is the code for `FetchFrom()`, `WriteBack()`, and `ByteToSector()`.

```
void FileHeader::FetchFrom(int sector){
    ASSERT(sector>=0);
    char buf[SectorSize];
    memcpy(buf, (char*)this, SectorSize*sizeof(char));
    kernel->synchDisk->ReadSector(sector, (char *)this);
    if(nextSector != -1){
        nextHdr = new FileHeader;
        nextHdr->FetchFrom(nextSector);
    }
}

void FileHeader::WriteBack(int sector){
    ASSERT(sector>=0);
    char buf[SectorSize];
    memcpy(buf, (char*)this, SectorSize*sizeof(char));
    kernel->synchDisk->WriteSector(sector, (char *)buf);
    if(nextSector != -1){
        ASSERT(nextHdr != NULL);
        nextHdr->WriteBack(nextSector);
    }
}

int FileHeader::ByteToSector(int offset){
    int index = offset / SectorSize;
    if (index < NumDirect)
        return (dataSectors[index]);
    else{
        ASSERT(nextHdr != NULL);
        return nextHdr->ByteToSector(offset - MaxFileSize);
    }
}
```

5. Last, we still need to do a little change in `openfile.cc`.
    - make `OpenFile::Length()` calling `FileHeader::Length()` to recursively accumulate the file length. And change the `fileLength` in `OpenFile::ReadAt()`, `OpenFile::WriteAt()` from `FileHeader::Length()` into `OpenFile::Length()` so we can get the correct total file size.

```
int OpenFile::Length(){
    int length = 0;
    FileHeader *next_hdr = hdr;
    while(next_hdr != NULL){
        length += next_hdr->FileLength();
        next_hdr = next_hdr->GetNextFileHeader();
    }
    return length;
}
```

Part III - Modify the file system code to support the subdirectory

a. Implement the subdirectory structure

1. For this task, we can separate it into two subtasks, handle the path and do correct instructions with the path.
2. For the path, we split it with `/` and saved in a `vector<char*>`.
3. As for how to do instructions correctly with the splitted path, we sent the `vector<char*>` into a function called `TraverseDirectory(vector<char*> path, int parent, int level)`, where `parent` is the sector number of parent directory's file header, and level is the recursion level.
4. We will get the target files parent directory from `TraverseDirectory()`, so we can do all the instructions as usual with the parent directory and the file (no matter it's a file or a subdirectory).

```
int FileSystem::TraverseDirectory(vector<char*> path, int parent, int level){
    int res = -1;
    OpenFile *parentFile = new OpenFile(parent);
    Directory *dir = new Directory(NumDirEntries);
    dir->FetchFrom(parentFile);

    // base case
    // parent is root or the target is root itself
    if(path.size() <= 1)
        res = 1;
    //base case
    // reach the last item in path (target)
    else if(level == path.size()-1)
        res = parent;
    // recursive case
    else{
        int sector = dir->Find(path[level]);
        ASSERT(sector != -1);
        res = TraverseDirectory(path, sector, level+1);
    }
    delete parentFile;
    delete dir;
    return res;
}
```

b. Support up to 64 files/subdirectories per directory

1. Change `NumDirEntries` to 64 can complete this request.
2. We also have to print out the structure recursively.
3. In `main.cc`, we separated the list instruction into two cases (normal and recursive)

```
if (dirListFlag)
{
    if(!recursiveListFlag) kernel->fileSystem->List(listDirectoryName);
    else kernel->fileSystem->RecursiveList(listDirectoryName);
}
```

4. For the normal `List(char *name)`, we just have to show the files and subdiretories directly under a particular diretory.
    - It's all the same, split the path, and traverse to get the correct parent directory.
    - Justify whether the `path_list.size() < 1` (the target directory is root), if not, get the corresponding target directory.
    - Last, call `Directory::List()`.

```
void FileSystem::List(char *name){
    int parent_sector = -1, target_sector = -1;
    OpenFile *f;
    Directory *directory = new Directory(NumDirEntries);
    vector<char*> path_list = SplitPath(name);

    parent_sector = TraverseDirectory(path_list, DirectorySector, 0);
    ASSERT(parent_sector != -1);
    f = new OpenFile(parent_sector);
    directory->FetchFrom(f);

    if(path_list.size() > 0){
        target_sector = directory->Find(path_list[path_list.size()-1]);
        delete f;
        f = new OpenFile(target_sector);
        directory->FetchFrom(f);
    }

    directory->List();
    delete directory;
    delete f;
}
```

5. For the recursive `RecursiveList(char *name)`, do path split and directory traversal in this function, and pass the data into `recursion()`, where actually used to print information recursively.

```
void FileSystem::RecursiveList(char *name){
    int dir_sector = -1;

    vector<char*> path_list = SplitPath(name);
    dir_sector = TraverseDirectory(path_list, DirectorySector, 0);
    ASSERT(dir_sector != -1);

    recursion(path_list, dir_sector, 0);
}
```

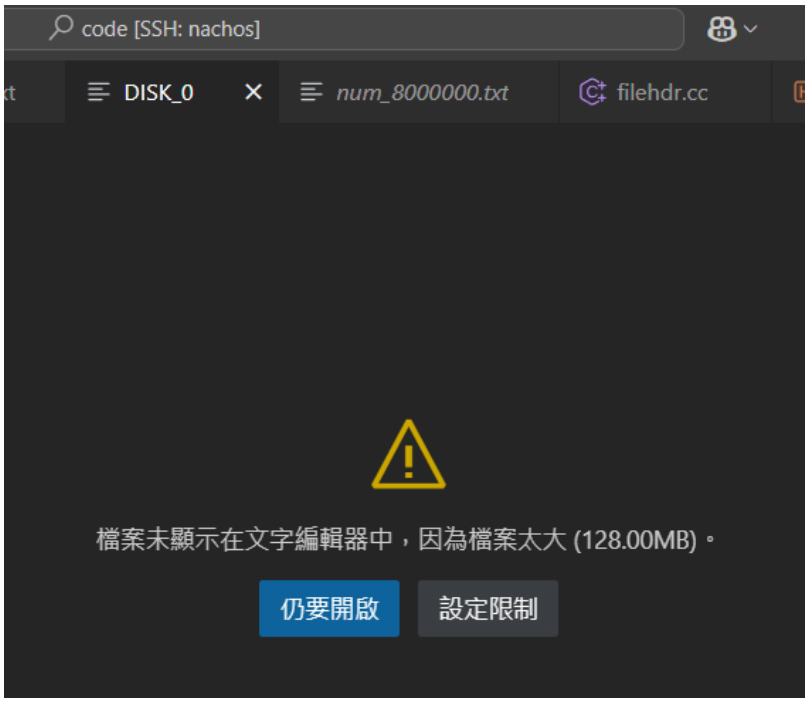
6. In `recursion(vector<char*>, int, int)`, same parameters with `TraverseDirectory()`.

- Get the current parent directory with sector number.
- Parse over the directory and print indentations with respect to the value of `level`.
- We have to fetch the file header for each entry, because we can get the `isDir` value from the header. Later, print the corresponding information.
- If the type of entry is a diretory, need to recusively to the next level.

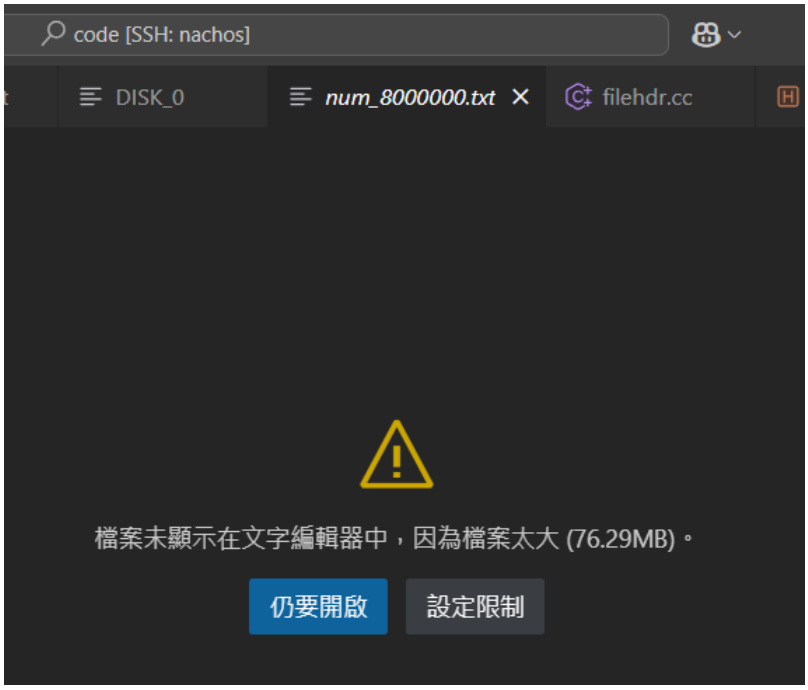
```
void FileSystem::recursion(vector<char*> path, int parent, int level){
    OpenFile *f;
    Directory *directory = new Directory(NumDirEntries);
    FileHeader *entryHdr = new FileHeader();

    f = new OpenFile(parent);
    directory->FetchFrom(f);

    for(int i=0; i<directory->getTableSize(); i++){
        DirectoryEntry *entry = directory->getEntry(i);
        if(entry->inUse){
            entryHdr->FetchFrom(entry->sector);
            for(int l=0; l<level; l++){
                printf("    ");
            }
            if(entryHdr->GetIsDir()){
                printf("[D] %s\n", entry->name);
                recursion(path, entry->sector, level+1);
            }
            else printf("[F] %s\n", entry->name);
        }
    }

    delete f;
    delete directory;
    delete entryHdr;
}
```

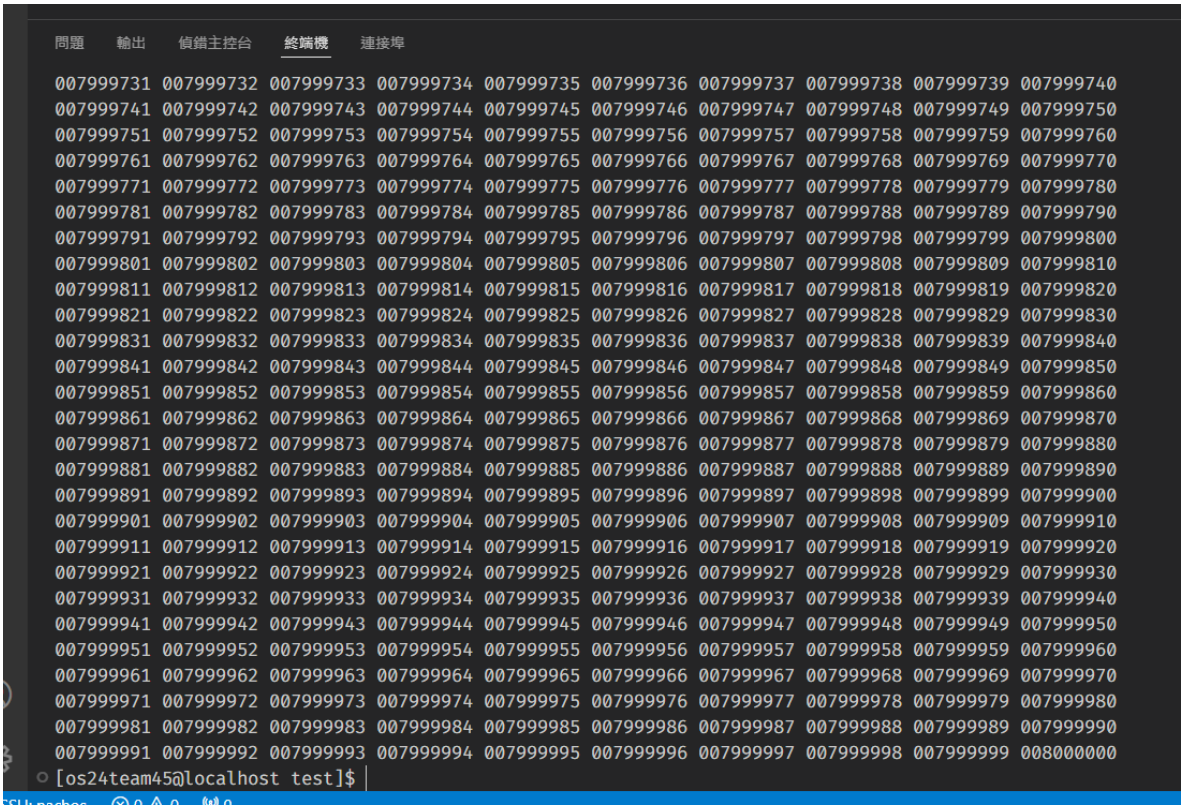
Bonus I - Enhance the NachOS to support even larger file size

1. As we mentioned before, `Disk size = SectorSize * NumSectors = SectorSize * (SectorPerTrack * NumTracks)`.
2. Because we can't change `SectorSize`, and we have to multiply current disk size with 64MB/128kB = 512.
3. So we decide to increase both `SectorPerTrack` and `NumTrack` from 32 to 32 * 32.
4. That is, `Disk size = 128 * (32 * 32) * (32 * 32) = 128MB`
5. **Can our file support up to 64MB single file ?** Since we use linked-index structure in our `FileHeader`, the file can infinitely find new linked-header to store the data exceeds the capacity of one header's `FileHeader::dataSector[NumDirect]`. To sum up, the maximum file size in our design is equal to disk size, which is `128MB` now.



6. For validation, we generated a text file called `num_8000000.txt` with size `76.29MB`, and replace it into `partII_b.sh`. After execution, it successfully print out all the numbers from 1 to 8,000,000. Following are the images of file size and result.





Bonus II - Multi-level header size.

1. Since we use the linked-index structure in PartII-b, our design's header size is already dynamic. And if the file needs more than a file header's `dataSector[]` to store the data, the file will have multiple headers eventually, and except for the last header, **all the fulled header will have size 3584 bytes** (28 integers).
2. To sum up, for all `single` file header, its file size is within interval (0, 3585).
3. Here shows three different kind of file header sizes.

```
Name: FS_test1, Sector: 1072
FileHeader contents.  File size: 948.  File blocks:
1073 1074 1075 1076 1077 1078 1079 1080
File contents:
```

```
Name: file1, Sector: 1081
FileHeader contents.  File size: 27.  File blocks:
1082
File contents:
```

```
Name: FS_test2, Sector: 1083
FileHeader contents.  File size: 980.  File blocks:
1084 1085 1086 1087 1088 1089 1090 1091
File contents:
```

That's all

We didn't work on Bonus 3