Operating System MP4

Report

Team 47

2021/01/17

Team Members:

Discussions between two members were held on both the implementation and code tracing.

廖允寬 106034071 - Completed most of the implementation and Part III and IV of the report.

洪紫珈 107062174 - Completed Part I and Part II of the report.

PART I - Understanding NachOS FS

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

The free block space in NachOS is managed by 'PersistentBitmap' which inherits from the 'Bitmap' class. In the filesys.h, there is a file object called 'freeMapFile' in FileSystem which stores all the free disk blocks of the system. When a file system is initialized, the constructor initializes the freeMapFile

```
        freeMapFile = new OpenFile(FreeMapSector);
```

where FreeMapSector has the index 0 in all sectors. Besides, the constructor of FileSystem also declares a PersistentBitMap variable 'freeMap' which records down the sectors that have been assigned or been used by a file.

A file can find free block space through FindAndSet(), this function will return a free sector that will assign to this file and mark it as used.

```
        sector = freeMap->FindAndSet();
```

And a sector can be returned to freeMap with Clear(), this function is called when deallocation happens and will mark the sector as unused.

```
        freeMap->Clear(sector);
```

(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
```

In the disk.h, the variables above have been declared which gives the maximum disk size possible.

So, NumTracks * SectorsPerTrack * SectorSize = 2^(5 + 5 + 7) = 128KB is the size of the disk.

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

Originally, no special structure was used to manage the directories because it did not support subdirectories. However, NachOS uses an ‘OpenFile’ object called directoryFile which acts as a root directory that stores a list of file names. When a file system is constructed, the directoryFile is initialized and stored in the index 1 sector.

```
    directoryFile = new OpenFile(DirectorySector);
```

Furthermore, under the Directory class, there are two variables as followed,

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

The tableSize indicates the number of entries it can hold, and the ‘table’ stores the filenames and their locations in the disk that are located under this directory.

With the following functions in the Directory class,

```
    void FetchFrom(OpenFile *file);
    void WriteBack(OpenFile *file);
    int Find(char *name);
    bool Add(char *name, int newSector, bool idDirectory);
    bool Remove(char *name);
    void List();
    void RecursiveList();
    void Print();
```

the interface is able to access and do operations on the DirectoryEntry table.

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

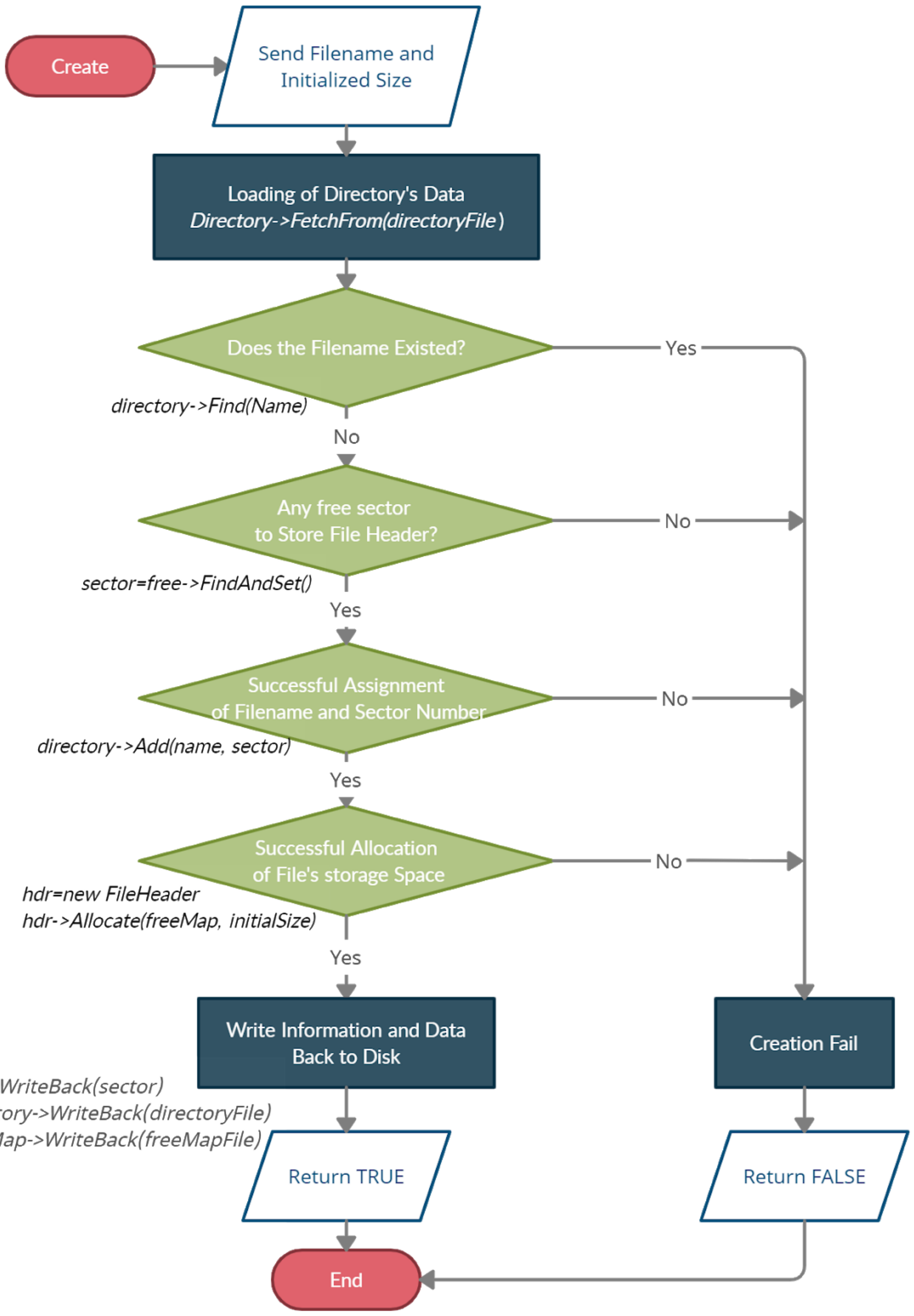
In the NachOS, an inode is implemented as a file header that stores the attributes and disk block locations of the object's data.

A file header has the following private variables,

```
  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 a file
```

They store the number of bytes a file occupies, the number of sectors used by a file, and the disk sector number of each data block in a file, respectively.

For the disk allocation scheme, please see below.


Create
Send Filename and Initialized Size
Loading of Directory's Data
Directory->FetchFrom(directoryFile)
Does the Filename Existed?
directory->Find(Name)
Yes
No
Any free sector to Store File Header?
sector=free->FindAndSet()
No
Yes
Successful Assignment of Filename and Sector Number
directory->Add(name, sector)
No
Yes
Successful Allocation of File's storage Space
hdr=new FileHeader
hdr->Allocate(freeMap, initialSize)
No
Yes
Write Information and Data Back to Disk
hdr->WriteBack(sector)
directory->WriteBack(directoryFile)
freeMap->WriteBack(freeMapFile)
Creation Fail
Return TRUE
Return FALSE
End

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

Since a file header should be fitted into a sector that holds 128bytes, those private variables like numBytes and numSectors should also be included in this 128KB. Thus, the space left in the file header will be available for storing the index of data sectors. Notice that numBytes and numSectors are both integers that take up 4 bytes, and each index of a data sector also needs 4 bytes.

128 bytes - (4 bytes * 2) = 120 bytes

120 bytes / 4 bytes = 30 indexes of sectors

30 indexes * 128 bytes = 3840 bytes

≒ 2^12 bytes = 4 KB

Therefore, we know that the size of a file is limited to 4KB without subdirectories.

PAR II - Modify the FS code to support file I/O system call and larger file size

To support file I/O system calls, we have implemented some functions in the following files.

ksyscall.h

```
OpenFileId SysOpen(char *name) {...OpenAFile(name)}
int SysRead(char *buffer, int size, OpenFileId id) {...ReadFile(buffer,size,id)}
int SysWrite(char *buffer,int size, OpenFileId id) {...WriteFile(buffer,size,id)}
int SysClose(OpenFileId id) {...CloseFile(id)}
int SysCreate(char *filename, int size) {...Create(filename, size)}
```

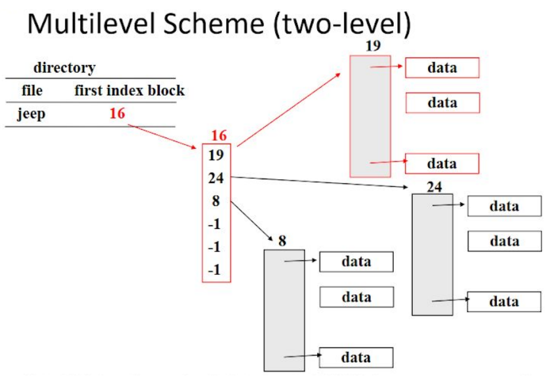
Here, each system call function calls a corresponding kernel operation to be done on a file. This acts as an interface from the user program to the kernel, so that ExceptionHandler(...) can call the following functions to perform operations.

Exception.cc

```
fileID = SysOpen(filename); // SC_Open

status = SysRead(buffer, val, fileID); // SC_Read

status = SysWrite(buffer, val, fileID); // SC_Write

status = SysClose(val); // SC_Close

status = SysCreate(filename, kernel->machine->ReadRegister(5)); // SC_Create
```

which are similar to the implementations in MP1.

Since the structure of the directory is single-layered, a file can hold up to 4KB only. To store a larger size file, it is necessary to change the structure of the directory. Here, we implement a new directory structure with the multilevel scheme learned in the class.



Since Bonus I wants to extend the file size to 64 MB, we will need to make a 4-level multilayer scheme.

LEVEL1 - 30*128 bytes ≒ 4 KB

LEVEL2 - 30*30*128 bytes ≒ 113 KB

LEVEL3 - 30*30*30*128 bytes ≒ 4 MB

LEVEL4 - 30* 30*30*30*128 bytes ≒ 104MB

filesys.h

New functions are declared in the FileSystem Class as shown below:

```
    int CreateAdirectory(char *name);
    bool RecursiveRemove(char *name);
    void RecursiveList(char *name);
    OpenFileId OpenAFile(char *name);
    int WriteFile(char *buffer, int size, OpenFileId id);
    int ReadFile(char *buffer, int size, OpenFileId id);
    int CloseFile(OpenFileId id);
```

filesys.cc

int FileSystem::Create(char *name, int initialSize)

1. Name and the filesize are passed as parameters, read the root directory called ‘directoryFile’.

2. Then with the strtok() function, we break down the absolute path into small parts which allow us to locate down through the directories until the last tmpName. By the time, we should be able to find the location for this file creation.
3. Check whether the filename exists or not through Find(). If the filename is unused, then continue with step 4, otherwise, return FALSE.
4. Find a free sector from freeMap, add this file to the directory. Continue if successful.
5. Allocate a free space for this file, continue if successful.
6. Flush all changes back to disk with writeback() function

int FileSystem::CreateAdirectory(char *name)

Most of the steps are identical to FileSystem::Create(), except step 4 where we add it as a directory, not as a file.

```
bool isAdd = directory->Add(Name, sector, TRUE);
```

This 'TRUE' indicates that this file object is a directory.

OpenFile *FileSystem::Open(char *name)

1. Break down the file name into small parts so we can locate the exact file requested. Each loop assigns directoryObj a file object which can be either a directory or a file.
2. Return the last found directoryObj which will be the file requested to open.

OpenFileId FileSystem::OpenAFile(char *name)

Most of the steps are identical to FileSystem::Open(), but this time, we return the index of the sector where the file locates, not the file itself.

int FileSystem::WriteFile(char* buffer, int size, OpenFileId id)

WriteFile calls the Write() function which will return the number actually written and increment position in file.

int FileSystem::ReadFile(char* buffer, int size, OpenFileId id)

ReadFile calls the Read() function which will return the number actually read and increment position in file.

int FileSystem::CloseFile(OpenFileId id)

This function simply deletes the fileDescriptor and assigns NULL to this pointer.

bool FileSystem::Remove(char *name)

1. With the given name, we break it down with strtok() function and find the sector which stores the file with remove request.
2. Return if the sector is not found, else continue.
3. Fetch sector data out, so we can remove the header and data blocks.
4. Mark the Directory Entry as unused on the Directory table where the file used to occupy.
5. Flush all changes back to disk with the writeback() function and delete the temporary variables used in the function.
6. Return TRUE when the file is successfully removed.

void FileSystem::List(char *name)

Again, with the strtok() function, we break down the absolute path and print each out. Then, with the

```
    directory->List();
```

we will print everything out on the directory.

void FileSystem::Print()

This function prints information on the freeMapSector, DirectorySector, the freeMap, and this root directory. Print() Function will be introduced below.

filehdr.cc

bool FileHeader::Allocate(PersistentBitmap *freeMap, int filesize)

Upon the file creation, we are to determine a suitable level of the multi-level scheme with the given file size. It is important to note that each block can only hold 30 dataSector's indexes.

1. We use divRoundUp to determine the number of sectors required by the file.
2. If there is not enough free sector in freeMap, then return FALSE, otherwise continue.

3. Determine whether the file size is smaller than 4KB, if so, then we can use the original implementation which simply assigns the free sectors to 1st level dataSectors. Else, we will recursively go through each subheader and assign free sectors for the file.
4. The variable 'fileSize' is decremented as free sectors are assigned to the file.
5. Notice that the dataSectors are writeBack to the disk of the file headers in every recursive loop.
6. The Recursive loop will terminate upon the completion of allocation.

void FileHeader::Deallocate(PersistentBitmap *freeMap)

1. With the numBytes of the fileHeader, we can determine the number of rounds of deallocation.
2. Deallocate() function will be called recursively until all sectors used by the file are returned to freeMap.

int FileHeader::ByteToSector (int offset)

1. Determine the level of the multi-level scheme used.
2. Use division to find which data sector the byte stores.
3. Use modules to find the position in the sector where the byte stores.
4. Recursive calls ByteToSector() function until we determine the exact location of the byte stores.
5. The function returns the disk sector that stores a particular byte which offset indicates.

void FileHeader::Print()

This function prints the contents of the file header and all the data blocks pointed by the file header.

1. Again we use the recursive methodology to go through each subheader and print the contents in it.
2. Notice that we also use the Levels to determine whether to continue the recursion or simply print the content. If numBytes are smaller than Level2 that means, it does not have an extended directory to go through.

PART III - Modify the FS code to support subdirectory

Implement the subdirectory structure:

1. For some functions we mentioned above, we split the absolute path by "/" using the strtok function.
2. We added isDirectory attribute in class DirectoryEntry to distinguish a file from a directory.
3. In filesys.h, the CreateADirectory function is declared for creating a directory and is similar to Create.
4. Filesystem::RecursiveList will split the absolute path and call Directory::RecursiveList, and Directory::RecursiveList which prints each filename and directory name under the current directory. When looping to a directory, call the Directory::RecursiveList again to print the content in this directory first.

Support up to 64 files/subdirectories per directory:

1. Modified 'NumDirEntries' from 10 to 64.
2. In CreateADirectory, when allocating space, we use `DirectoryFileSize` as the file size which is the value of NumDirEntries*sizeof(DirectoryEntry).

PART IV - Bonus Assignment

Bonus I:

We set the NumTracks to 16384, then the total size is SectorSize*(SectorPerTrack*NumTracks) = 128*(32*16384) = 67108864 ≒ 64MB.

Bonus II:

In FileHeader::Allocate, there are 4 level size bound according to the file size, if the current file size is larger than a block with 30 dataSectors, we allocate the current bound value recursively and minus file size by bound, and keep allocating until the file size is smaller than 0.

Bonus III:

bool FileSystem::RecursiveRemove(char *name)

1. Get the target file name by strtok function

2. Check if the target is a file or a directory, for the former, we deallocate the file header, clean the sector, and call directory->Remove to remove this file; For the latter, we loop each directoryTable and find which is inUse. Remove the in use one by calling RecursiveRemove again if it is a directory, or just remove it if it is a file.