CS342301: Operating System
MP4: File System

Team no. 46

Member Contribution:

陳敬中: Trace Code, Report, Implementation (50 %)

白宸安: Trace Code, Report, Implementation (50 %)

MP4

Part 1.

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

- NachOS file system manages free disk blocks with a persistent bitmap. The bitmap is represented as a file, which is stored in the disk. NachOS file system finds the bitmap by accessing the file header of the bitmap, which is stored in sector 0 on the disk.

kernel.cc

Kernel::Kernel(int argc, char **argv)

```
#ifndef FILESYS_STUB
    formatFlag = FALSE;
#endif
        //...
#ifndef FILESYS_STUB
        } else if (strcmp(argv[i], "-f") == 0) {
            formatFlag = TRUE;
#endif
        //...
```

Kernel::Initialize()

```
#ifdef FILESYS_STUB
    fileSystem = new FileSystem();
#else
    fileSystem = new FileSystem(formatFlag);
#endif // FILESYS_STUB
```

If not using the stub file system, the -f is set formatFlag to TRUE, and then you can use FileSystem (formatFlag) when initializing the file system.

filesys.cc

The bitmap and directory sectors are defined at the beginning. Use sector 0 for bitmap and sector 1 for directory.

In addition, the value of FreeMapFileSize is also defined.

```
#define FreeMapSector 0
#define DirectorySector 1

#define FreeMapFileSize (NumSectors / BitsInByte)
```

The following values are defined in disk.h

```
// MP4 Hint: DO NOT change the SectorSize, but other constants are allowed
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 di
```

FileSystem::FileSystem(bool format)

If format=TRUE, the disk will be empty, so the disk must be initialized so that it has a bitmap and an empty directory.

If format=False, use stub file system.

The initialization steps are as follows:

```
PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);
Directory *directory = new Directory(NumDirEntries);
FileHeader *mapHdr = new FileHeader;
FileHeader *dirHdr = new FileHeader;
```

Initialize the resources.

```
freeMap->Mark(FreeMapSector);
freeMap->Mark(DirectorySector);
```

Mark bitmap with sector 0 and directory with sector 1.

```
ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));
```

Allocate space to bitmap and directory.

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

Update the bitmap and directory back to the disk, which must be done before opening these two.

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

Keep bitmap and directory opened while NachOS is running.

```
DEBUG(dbgFile, "Writing bitmap and directory back to disk.");
freeMap->WriteBack(freeMapFile); // flush changes to disk
directory->WriteBack(directoryFile);
```

Write the initial version of the bitmap and directory to the disk. The directory will still be empty at this time, but the bitmap will record that sector 0 and sector 1 store both.

pbitmap.cc

```
PersistentBitmap::PersistentBitmap(int numItems) : Bitmap(numItems)
```

Call Bitmap(numItems) to initialize the bitmap with numItems bits.

bitmap.cc

Bitmap::Bitmap(int numItems)

```
Bitmap::Bitmap(int numItems)
{
    int i;

    ASSERT(numItems > 0);

    numBits = numItems;
    numWords = divRoundUp(numBits, BitsInWord);
    map = new unsigned int[numWords];
    for (i = 0; i < numWords; i++)
    {
        map[i] = 0; // initialize map to keep Purify happy
    }
    for (i = 0; i < numBits; i++)
    {
        Clear(i);
    }
}
```

NachOS uses bitmap to record where there is free block space.

The empty sectors are set to 0, and sectors with stored items are set to 1.

bool Bitmap::Test(int which)

Used to check whether the nth bit is set (whether the nth sector is in use).

1 → in use

0 → not in use

int Bitmap::FindAndSet()

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

Starting from 0, find the first free block space bit and return it. If not found, -1 is returned.

void Bitmap::Mark(int which)

Set the nth bit to 1.

void Bitmap::Clear(int which)

Set the nth bit to 0.

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

- In the current implementation, there are 32 tracks, 32 sectors per track, and the size of a sector is 128 bytes. Therefore, the maximum disk size is 128 KB.

disk.h

```
// MP4 Hint: DO NOT change the SectorSize, but other constants are allowed
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 di
```

As can be seen from disk.h, each sector has 128 bytes, each track has 32 sectors, and there are 32 tracks.

So there are 32 * 32 = 1024 sectors in total

Each sector is 128 bytes, so disk size = 1024 * 128 = 128 KB.

disk.cc

```
const int MagicNumber = 0x456789ab;
const int MagicSize = sizeof(int);
```

```
const int DiskSize = (MagicSize + (NumSectors * SectorSize));
```

從這邊也可以看出DiskSize的計算公式。

注意，這邊MagicNumber是用來避免將有用的file當成disk來用，並沒有對disk的實際size造成影響。

We can also see the calculation formula of DiskSize from here.

Note that the MagicNumber here is used to avoid using useful files as disks and does not affect the actual size of the disk.

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

- NachOS file system manages the directory data structure with a table (an array of directory entries). Each table entry contains information of a file including its filename and its location on disk. NachOS finds the directory structure by accessing the directory header (as a file) on disk, which is stored in sector 1.

filesys.cc

```
#define NumDirEntries       10
#define DirectoryFileSize   (sizeof(DirectoryEntry) * NumDirEntries)
```

The number of entries is defined at the beginning as 10, and the size of the directory.

FileSystem::FileSystem(bool format)

```
Directory *directory = new Directory(NumDirEntries);
```

The directory in NachOS is a file, which is managed by file.

As described in question 1.

If format=TRUE, the disk will be empty, so the disk must be initialized so that it has a bitmap and an empty directory.

If format=False, use stub file system.

The initialization steps are as follows:

```
PersistentBitmap *freeMap = new PersistentBitmap(NumSectors);
Directory *directory = new Directory(NumDirEntries);
FileHeader *mapHdr = new FileHeader;
FileHeader *dirHdr = new FileHeader;
```

Initialize the resources.

```
freeMap->Mark(FreeMapSector);
freeMap->Mark(DirectorySector);
```

Mark bitmap with sector 0 and directory with sector 1.

```
ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));
```

Allocate space to bitmap and directory.

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

Update the bitmap and directory back to the disk, which must be done before opening these two.

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

Keep bitmap and directory opened while NachOS is running.

```
DEBUG(dbgFile, "Writing bitmap and directory back to disk.");
freeMap->WriteBack(freeMapFile); // flush changes to disk
directory->WriteBack(directoryFile);
```

Write the initial version of the bitmap and directory to the disk. The directory will still be empty at this time, but the bitmap will record that sector 0 and sector 1 store both.

Directory.cc

Directory is a table with fixed-length entries. Each entry represents a single file and contains the file name and the location of the file header on the disk.

Because each entry has a fixed size, it means that the file name has an upper limit on the length.

Directory::Directory(int size)

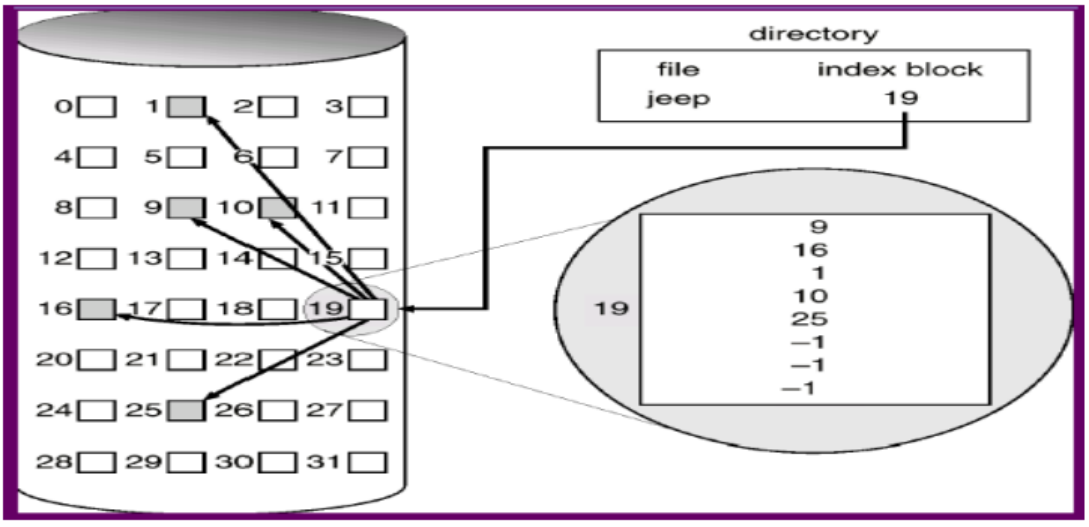
```
Directory::Directory(int size)
{
    table = new DirectoryEntry[size];

    // MP4 mod tag
    memset(table, 0, sizeof(DirectoryEntry) * size); // dummy operation to keep va

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

Initialize the directory, size means how many entries.

At the beginning, each entry inUse is set to false, which means that this directory does not currently have any entries.

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

- Inode contains information about a file including file size (in number of bytes and number of sectors) and pointers to each data block of the file. The pointers to data blocks are maintained in an array called `dataSecors`.



The inode here is the file header in NachOS. So go look in filehdr.h, filehdr.cc.

filehdr.h

```
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
```

Let's take a look inside filehdr.h.

numBytes represents how many bytes this file has.

numSectors represents how many sectors this file uses.

dataSectors stores the pointer to the data block, that is, which sectors on the disk are used by this file.

filehdr.cc

```
// filehdr.cc
//  Routines for managing the disk file header (in UNIX, this
//  would be called the i-node).
//
//  The file header is used to locate where on disk the
//  file's data is stored.  We implement this as a fixed size
//  table of pointers -- each entry in the table points to the
```

```
//  disk sector containing that portion of the file data
//  (in other words, there are no indirect or doubly indirect
//  blocks). The table size is chosen so that the file header
//  will be just big enough to fit in one disk sector,
```

The file header is used to find where the file data is placed on the disk.

The file header of NachOS is a table with fixed size pointers. The entry in each table points to the disk sector containing the portion file data.

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

Basically the initialization of the file header is also done here.

```
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
        ASSERT(dataSectors[i] >= 0);
    }
    return TRUE;
}
```

Set the values of numBytes and numSectors.

Then use a for loop to find numSectors sectors in the bitmap to use as data blocks, and store the numbers found in dataSectors[i].

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

- In the current implementation, the indexed table of a file is stored in a file header. Each file header is stored in one sector. Since the size of a file is limited by the size of the indexed table and the maximum size of the indexed table is $(128 - 2 * 4)/4 = 30$, the maximum size of a file is 30 sectors, i.e., 3840 bytes.

disk.h

From here we know that SectorSize is 128

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

filehdr.h

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

```
//...

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
```

Implement

Part II.

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

As we did in MP1, we provide the system call interface in `ksyscall.h` for the five system calls. Since there is at most one open file in the system, we maintain an open file pointer `OF` in the kernel.

```
int SysCreate(char *name, int size)
{
    kernel->fileSystem->Create(name, size);
    return 1;
}

OpenFileId SysOpen(char *name)
{
    kernel->OF = kernel->fileSystem->Open(name);
    return 1;                       // only one open file at a time, use 1 as open
}

int SysRead(char *buf, int size, OpenFileId id)
{
    return kernel->OF->Read(buf, size);
}

int SysWrite(char *buf, int size, OpenFileId id)
{
    return kernel->OF->Write(buf, size);
}

int SysClose(OpenFileId id)
{
    delete kernel->OF;
    return 1;
}
```

And we handle the system calls with `ExceptionHandler()` in `Exception.cc`, which is nearly the same as the implementation we did in MP1.

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

We use the linked scheme. First modify the NumDirect, to save an extra pointer point to the linked file header. The scheme can dynamically fit the file size, so it can support up to 32 KB file.

```
#define NumDirect ((SectorSize - 3 * sizeof(int)) / sizeof(int))
```

filehdr.h

```
//..
class FileHeader {
// ...
private:
        // in-core part
        FileHeader* nextFileHeader;
        //IndexTable **idxTable;
        int numTables;

        // Disk part
        int nextFileHeaderSector;
        //bool isDir;                 // set if the file is directory
        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
}
```

Add nextFileHeader FileHeader pointer, use to link another file header to save more data.

Add nextFileHeaderSector to store the sector on disk of the next file header.

filehdr.cc

FileHeader::FileHeader()

```
FileHeader::FileHeader()
{
    nextFileHeader = NULL;
    nextFileHeaderSector = -1;
    isDir = FALSE;
    numBytes = -1;
    numSectors = -1;
    memset(dataSectors, -1, sizeof(dataSectors));
}

FileHeader::~FileHeader()
{
    if (nextFileHeader != NULL) delete nextFileHeader;
}
```

Constructor：set nextFileHeader to NULL, and nextFileHeaderSector to -1.

Destructor: if the file header has a link, delete the link.

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

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

    // not enough space
    if (freeMap->NumClear() < numSectors)
        return FALSE;

    for (int i = 0; i < numSectors; i++) {
        dataSectors[i] = freeMap->FindAndSet();
        //we checked that there was enough free space, we expect this to succeed
        ASSERT(dataSectors[i] >= 0);
    }
    if (fileSize > 0) {
        nextFileHeaderSector = freeMap->FindAndSet();
        if (nextFileHeaderSector == -1)
                return FALSE;
        else {
            //cout << "linked another" << endl;
            nextFileHeader = new FileHeader;
            return nextFileHeader->Allocate(freeMap, fileSize);
        }
    }
    return TRUE;
}
```

`numBytes` is the size of the file

`localNumBytes` is the bytes that this file header block is going to store.

`numSectors` number of sectors needed by the file

the for loop fill the dataSectors with data, and if the fileSize is still > 0, create a new file header block to store, and call Allocate to recursively do the job, until the file size = 0.

Return TRUE if the allocation succeeds.

void FileHeader::Deallocate(PersistentBitmap *freeMap)

```
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 (nextFileHeaderSector != -1)
    {
        ASSERT(nextFileHeader != NULL);
        nextFileHeader->Deallocate(freeMap);
    }
}
```

First, deallocate all sectors within the file header.

Then, deallocate the data sectors of the linked file header block.

void FileHeader::FetchFrom(int sector)

```
void FileHeader::FetchFrom(int sector)
{
    // kernel->synchDisk->ReadSector(sector, (char *)this);
    char buf[SectorSize];
    kernel->synchDisk->ReadSector(sector, buf);

    memcpy(&nextFileHeaderSector, buf, sizeof(int));
    memcpy(&numBytes, buf + 1 * sizeof(int), sizeof(int));
    memcpy(&numSectors, buf + 2 * sizeof(int), sizeof(int));
    memcpy(dataSectors, buf + 3 * sizeof(int), sizeof(dataSectors));

    if (nextFileHeaderSector != -1)
    {
        nextFileHeader = new FileHeader;
        //ASSERT(nextFileHeader != NULL);
        nextFileHeader->FetchFrom(nextFileHeaderSector);
    }
}
```

Read the file header from disk. `sector` is the sector number of this file header.

Use `kernel->synchDisk->ReadSector()` to read.

Since there are in-core part and disk part, we have to use `memcpy()` to restore `nextFileHeaderSector,` `numBytes`, `numSectors`, and `dataSectors`.

Then, we reconstruct the in-core data `nextFileHeader`.

We then recursively fetch linked file header data.

void FileHeader::WriteBack(int sector)

```
void FileHeader::WriteBack(int sector)
{
    // kernel->synchDisk->WriteSector(sector, (char *)this);
    char buf[SectorSize];
    memcpy(buf, &nextFileHeaderSector, sizeof(int));
```

```
    memcpy(buf + 1 * sizeof(int), &numBytes, sizeof(int));
    memcpy(buf + 2 * sizeof(int), &numSectors, sizeof(int));
    memcpy(buf + 3 * sizeof(int), dataSectors, sizeof(dataSectors));
    kernel->synchDisk->WriteSector(sector, buf);

    if (nextFileHeaderSector != -1)
    {
        ASSERT(nextFileHeader != NULL);
        nextFileHeader->WriteBack(nextFileHeaderSector);
    }
}
```

Write the modified contents of the file header back to disk recursively.

int FileHeader::ByteToSector(int offset)

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

Return the sector number that stores the data at `offset` in the file.

Need to recursively trace the link to get to the sector of that offset data.

void FileHeader::Print()

```
void FileHeader::Print()
{
    printf("FileHeader contents.  File size: %d.  File blocks:\n", numBytes);
    for (i = 0; i < numSectors; i++)
        printf("%d ", dataSectors[i]);
    printf("\nFile contents:\n");
    for (i = k = 0; i < numSectors; i++)
    {
        kernel->synchDisk->ReadSector(dataSectors[i], data);
        for (j = 0; (j < SectorSize) && (k < numBytes); j++, k++)
        {
            if ('\040' <= data[j] && data[j] <= '\176') // isprint(data[j])
                printf("%c", data[j]);
            else
                printf("\\%x", (unsigned char)data[j]);
        }
```

```
        printf("\n");
    }

    if (nextFileHeaderSector != -1)
    {
        ASSERT(nextFileHeader != NULL);
        nextFileHeader->Print();
    }

    delete[] data;
}
```

Print the content in file recursively.

Part III

1. Implement the subdirectory structure

Here are some flags we need to modify to support subdirectory structure.

1. `-mkdir` — create a subdirectory.
2. `-cp` — copy a UNIX file and store it under the given directory in nachos.
3. `-l` — list all files/directories under the given directory.
4. `-lr` — recursively list all files/directories under the given directory.
5. `-p` — prints a file to stdout.
6. `-r` — remove a file

-mkdir

main.cc

```
else if (strcmp(argv[i], "-mkdir") == 0)
{
    // MP4 mod tag
    ASSERT(i + 1 < argc);
    createDirectoryName = argv[i + 1];
    mkdirFlag = true;
    i++;
}

//...

if (mkdirFlag)
{
    // MP4 mod tag
    CreateDirectory(createDirectoryName);
}
```

```
//...

static void CreateDirectory(char *name)
{
    // MP4 Assignment
    // cout << "main mkdir" << endl;
    kernel->fileSystem->CreateDir(name);
    return;
}
```

First, we handle the flag in `main.cc`. `mkdirFlag` is set if we are to create a new subdirectory. The the file system calls `CreateDir(name)`.

filesys.cc

bool FileSystem::CreateDir(char *dirName)

`dirName` is the input path name, e.g. `/t0/t1`

Initialization:

```
Directory *newDir, *curDir;
PersistentBitmap *freeMap;
FileHeader *hdr;
OpenFile *curDirFile = directoryFile;
char *path;
char *slash;
int sector;
bool success;
int curDirSector;

newDir = new Directory(NumDirEntries);
// root directory
curDir = new Directory(NumDirEntries);
```

Read root directory. Current directory is at root.

```
curDir->FetchFrom(directoryFile);
curDirSector = DirectorySector;
```

We skip the first slash, because it represents the root directory, and we are already there.

```
path = dirName + 1;
```

We traverse through the file system to find the right place to create the subdirectory.

In this while loop, we keep searching for a '/'. The string leading the slash is a directory. We then traverse to this directory (set `curDir` to this directory). Afterwards, we skip this slash and search for another one.

Take `/t0/t1` as an example. The first slash is skipped, `t0/t1`. `slash` splits the path into `t0` and `t1` . `t0` is the directory we traverse. `t1` is the remaining path we need to traverse.

Once there is no more slash in `path`, it contains the name of the directory we are going to create.

```
while ((slash = strchr(path, '/')) != NULL)    // find the next slash
{
    char curDirName[slash - path + 1];
    strncpy(curDirName, path, slash - path);
    curDirName[slash - path] = '\0';
    path = slash + 1;
    curDirSector = curDir->Find(curDirName);
    curDirFile = new OpenFile(curDirSector);
    curDir->FetchFrom(curDirFile);
    delete curDirFile;
}
```

After entering the directory you want to go to (such as `t0`), first check whether `t1` is already in `t0`. If it is already there, set success to FALSE.

Then read freeMap and find a sector for `t1` to store its file header.

Then create a new hdr FileHeader and assign how many sectors to use for hdr.

Finally, write all the content to be updated back to the disk, such as writing hdr back to the sector where the `t1` file header is stored, and updating curDir, newDir and freeMap.

```
if (curDir->Find(path) != -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 (!curDir->Add(path, sector, TRUE))
            success = FALSE; // no space in directory
        else {

            hdr = new FileHeader;
            if (!hdr->Allocate(freeMap, DirectoryFileSize))
                success = FALSE; // no space on disk for data
            else
            {

                // everthing worked, flush all changes back to disk
                hdr->WriteBack(sector);
                OpenFile *newDirFile = new OpenFile(sector);
                curDirFile = new OpenFile(curDirSector);
                newDir->WriteBack(newDirFile);
                curDir->WriteBack(curDirFile);
                freeMap->WriteBack(freeMapFile);
                delete newDirFile, curDirFile;
            }
            delete hdr;
```

```
        }
        delete freeMap;
    }
    delete newDir, curDir;
    return success;
```

-cp

main.cc

```
else if (strcmp(argv[i], "-cp") == 0)
{
    ASSERT(i + 2 < argc);
    copyUnixFileName = argv[i + 1];
    copyNachosFileName = argv[i + 2];
    i += 2;
}

//...

if (copyUnixFileName != NULL && copyNachosFileName != NULL)
{
    Copy(copyUnixFileName, copyNachosFileName);
}
```

main.cc will read the -cp flag and call the Copy function.

Copy()

```
//...
if (!kernel->fileSystem->Create(to, fileLength))
{ // Create Nachos file
    printf("Copy: couldn't create output file %s\n", to);
    Close(fd);
    return;
}

openFile = kernel->fileSystem->Open(to);
//...
```

The Copy function will call Create and Open function in the file system.

filesys.cc

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

The processing is basically the same as CreateDir. The different parts are:

```
path = name + 1;
char *parentPath;
```

```
char newFileName[strlen(name) + 1];
char *lastSlash = strrchr(path, '/');
if (lastSlash != NULL)
{
    parentPath = new char[lastSlash - path + 2];
    parentPath[0] = '/';
    strncpy(&parentPath[1], path, lastSlash - path);
    parentPath[lastSlash - path + 1] = '\0';
    // parentPath[lastSlash - path+2] = '\0';
    strcpy(newFileName, lastSlash + 1);
}
else
{
    parentPath = new char[1 + 1];
    parentPath[0] = '/';
    parentPath[1] = '\0';
    strcpy(newFileName, path);
}
```

When creating a file, the last one will be the file name to be created instead of the directory. For example, if you enter `/t1/f1`, `f1` will be the file name.

Therefore, we must first disassemble the path `/t1/f1` into `/t1` and `f1`. The former `/t1` is parentPath, and the latter `f1` is newFileName.

The subsequent steps are the same as CreateDir, except that the original path is replaced by parentPath.

Then when you finally update the disk, add the newly added `f1` to the `t1` directory.

OpenFile *FileSystem::Open(char *name)

There is also a modification to the Open function, which will affect `cp` and `p`.

The steps for processing parentPath are the same as Create above. The difference is below:

```
Directory *directory = new Directory(NumDirEntries);
directory->FetchFrom(directoryFile);
int curDirSector;
curDirSector = DirectorySector;

path = name + 1;
char *slash;
while ((slash = strchr(path, '/')) != NULL)    // find the next slash
{
    char curDirName[strlen(name) + 1];
    strncpy(curDirName, path, slash - path);
    curDirName[slash - path] = '\0';
    path = slash + 1;
    curDirSector = directory->Find(curDirName);
    curDirFile = new OpenFile(curDirSector);
    directory->FetchFrom(curDirFile);
}
```

```
sector = directory->Find(newFileName);
if (sector >= 0)
    openFile = new OpenFile(sector); // name was found in directory
delete directory;
return openFile; // return NULL if not found
```

Handle parentPath in a similar way, but return the openFile object of the found file.

-l

main.cc

```
else if (strcmp(argv[i], "-l") == 0)
{
    // MP4 mod tag
    ASSERT(i + 1 < argc);
    listDirectoryName = argv[i + 1];
    dirListFlag = true;
    i++;
}

//...

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

}
```

main.cc will parse the -l flag and call List function in the file system.

filesys.cc

void FileSystem::List(char *name)

```
void FileSystem::List(char *name)
{
    Directory *directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);
    if (strcmp(name, "/") == 0)
    {
        directory->List();
    }
    else
    {
```

```
        char *path = name + 1;
        char *slash;
        while ((slash = strchr(path, '/')) != NULL)
        {
            char curDirName[slash - path + 1];
            strncpy(curDirName, path, slash - path);
            curDirName[slash - path] = '\0';
                        path = slash + 1;
            int curDirSector = directory->Find(curDirName);
            OpenFile *curDirFile = new OpenFile(curDirSector);
            directory->FetchFrom(curDirFile);
            delete curDirFile;
        }
        int curDirSector = directory->Find(path);
        OpenFile *curDirFile = new OpenFile(curDirSector);
        directory->FetchFrom(curDirFile);
        directory->List();
        delete curDirFile;
    }
    delete directory;
}
```

The parsing part of the file path is very similar to the above method. After running to the last directory, read the file of the directory, and then call directory→List() on it.

For example, `-l /t0/t1` will open the file of `t1` and print out the files in `t1`.

-lr

The method is almost the same as -l, except that the call to directory→List() is replaced by directory->RecursiveList(0).

-p

Because Print() called by `-p` will only use the fileSystem function Open(), and the modification of Open has been described in `-cp`.

-r

main.cc

```
else if (strcmp(argv[i], "-r") == 0)
{
    ASSERT(i + 1 < argc);
    removeFileName = argv[i + 1];
    i++;
}

//...
```

```
if (removeFileName != NULL)
{
    if (recursiveRemoveFlag) {
        kernel->fileSystem->RecursiveRemove(removeFileName);
    } else {
        kernel->fileSystem->Remove(removeFileName);
    }
}
```

filesys.cc

bool FileSystem::Remove(char *name)

```
bool FileSystem::Remove(char *name)
{
    Directory *directory;
    PersistentBitmap *freeMap;
    OpenFile *dirFile;
    FileHeader *fileHdr;
    int curDirSector;
    int sector = DirectorySector;

    directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);

    char *path = name + 1;
    char *slash;
    while ((slash = strchr(path, '/')) != NULL)
    {
        char curDirName[slash - path + 1];
        strncpy(curDirName, path, slash - path);
        curDirName[slash - path] = '\0';
        path = slash + 1;
        curDirSector = directory->Find(curDirName);
        OpenFile *curDirFile = new OpenFile(curDirSector);
        directory->FetchFrom(curDirFile);
        delete curDirFile;
    }
    sector = directory->Find(path);

    if (sector == -1)
    {
        delete directory;
        return FALSE; // file not found
    }
    fileHdr = new FileHeader;
    fileHdr->FetchFrom(sector);

    freeMap = new PersistentBitmap(freeMapFile, NumSectors);
```

```
    dirFile = new OpenFile(curDirSector);

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

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

First, initialize resources and parse paths as usual.

If file is found, do the following actions.

1. Remove it from the directory
2. Delete the space for its header
3. Delete the space for its data blocks
4. Write changes to directory, bitmap back to disk

2. Support up to 64 files/subdirectories per directory

Change NumDirEntries to 64 so that each directory can have 64 files/subdirectories

```
///#define NumDirEntries 10
#define NumDirEntries 64
```

part 3. Bonus Assignment

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

```
const int SectorSize = 128;     // number of bytes per disk sector
const int SectorsPerTrack  = 32 * 2;    // number of sectors per disk track
const int NumTracks = 32 * 512;     // number of tracks per disk
const int NumSectors = (SectorsPerTrack * NumTracks); // total # of sectors per di
```

we modify the SectorsPerTrack and NumTracks, to support up to 64 MB file.

Because we use linked scheme, so it can dynamically fit the file size, the only problem here is that the copy speed is slow due to keeping reading the disk.

Bonus II: Multi-level header size

Because we use linked scheme, so if the file is large, it will link many file header block, causing the effect of multi-level header size.