西安交通大学

**操作系统专题实验报告**

班级： 计算机86

学号： xxxxxxxxxx

姓名： 李程浩

2021年3月14日

**目 录**

[3 类EXT2文件系统的设计 61](#_Toc61213526)

[3.1实验目的 61](#_Toc61213527)

[3.2实验内容 61](#_Toc61213528)

[3.3实验思想（或流程图） 61](#_Toc61213529)

[3.4实验步骤 62](#_Toc61213530)

[3.5测试数据设计 64](#_Toc61213531)

[3.6程序运行初值及运行结果分析 64](#_Toc61213532)

[3.7实验总结 71](#_Toc61213533)

[3.7.1实验中的问题与解决过程 71](#_Toc61213534)

[3.7.2实验收获 72](#_Toc61213535)

[3.8附件 72](#_Toc61213536)

[3.8.1附件1 程序 72](#_Toc61213537)

[3.8.2附件2 Readme 110](#_Toc61213538)

# 3 类EXT2文件系统的设计

## 3.1实验目的

通过一个简单文件系统的设计，加深理解文件系统的内部实现原理.

## 3.2实验内容

模拟EXT2文件系统原理设计实现一个类EXT2文件系统.

## 3.3实验思想（或流程图）

使用分层的面向对象思想, 进行文件系统的设计和实现:



## 3.4实验步骤

**1. 理解Ext2文件系统的设计思想, 自顶向下进行设计.**

## 理解需求

"硬盘"中包括两部分:

1. "控制部分"包括:

> 兼并了超级块的组描述符, 用于记录整个文件系统的一些信息

数据块位图, 用二进制位表示哪些数据块用过了

索引节点位图, 用二进制位表示在索引节点表中, 哪些位置被索引节点占用了

索引节点表, 用于存放一定数量的索引节点(inode)

索引节点(inode), 一个inode包含了一个文件的属性信息和在磁盘上的位置. 其中采用直接索引, 一级索引, 二级索引想结合的方式来记录位置.

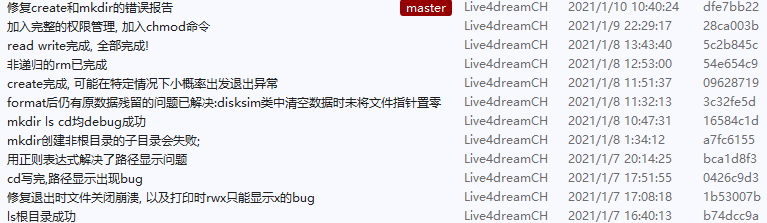
2. "数据部分"包括一定数量的数据块. 数据块是存储文件内容的地方, 而文件又分为数据文件和目录文件:

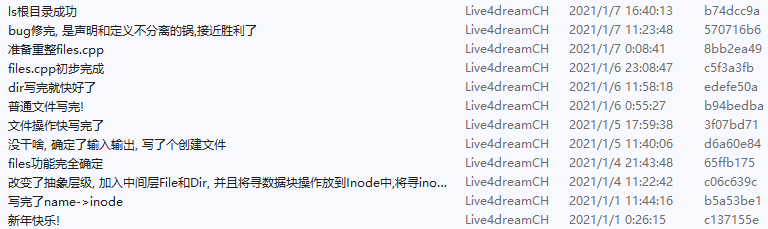
1. 数据文件所占据的数据块全部用来存储文件内容

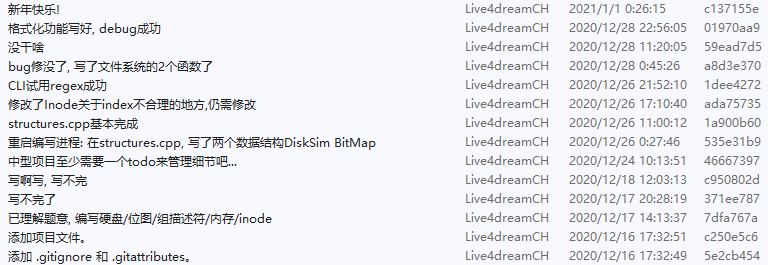
2. 目录文件的内容由若干个目录项构成. 目录项的数量等于此目录下的文件/文件夹数量+2(还有./和../两个默认文件夹). 每个目录项的内容由dir\_entry结构给出.

**2. 自底向上编写代码, 实现文件系统:**

以下为git的提交记录, 最终编写了超过1900行的代码, 也是我目前的最大项目。







## 3.5测试数据设计

由于采用命令行输入，故测试命令尽可能覆盖次系统的所有功能。

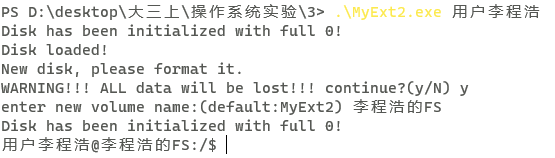
## 3.6程序运行初值及运行结果分析

**初值：**

将FS.txt删除，从零开始运行程序



**运行结果及分析：**

****

程序的运行参数是用户名，运行后检测不到文件，提示格式化。格式化时确定卷名。

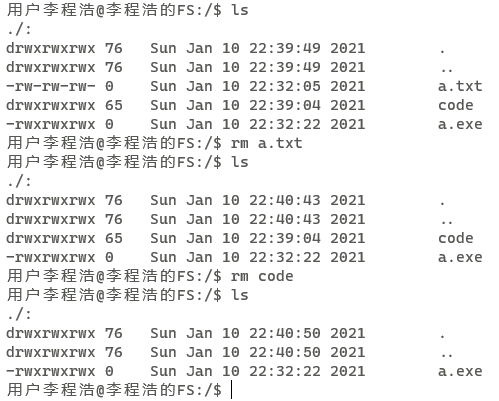


ls不加参数可以列出当前目录，create创建文件，mkdir创建文件夹，ls验证得创建成功。

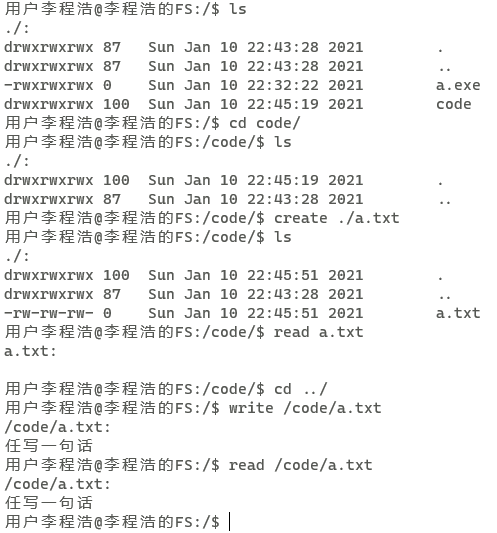
ls可以一次性列出多个文件夹，这种“批量”操作对大部分其余指令都有效。

ls可以列出普通文件的信息，ls遇到错误的路径会报错。

create可以根据文件的扩展名，正确地为其添加x执行权限。



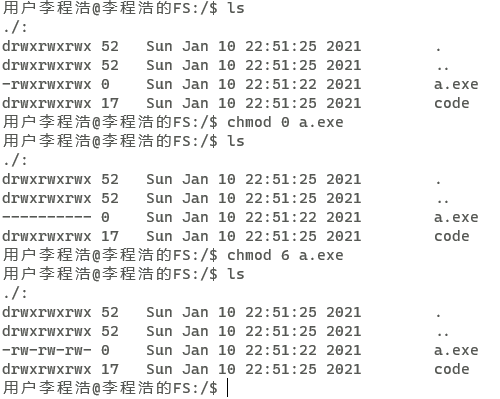
分别rm文件和文件夹，均成功。



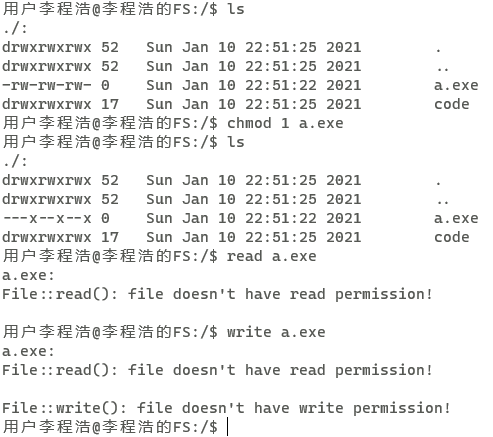
用cd进入code文件夹，在其下新建a.txt，读取其内容，为空，再使用cd ../回到根目录

在根目录利用完整路径，对a.txt进行写和读操作，均正常运作。

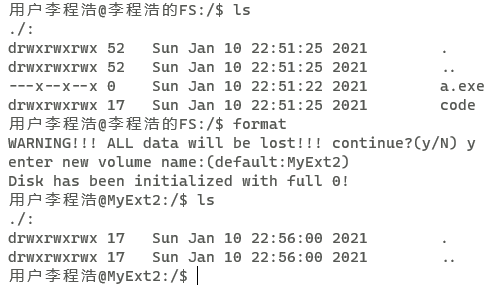
cd过程中，$符号左边的当前路径展示也正常。



使用chmod对a.exe的权限进行修改。



用chmod 1剥夺a.exe的读写权限，然后对其进行读写，均无法正常进行。



使用format格式化系统，格式化后仅剩根目录，其余一切信息全部丢失。

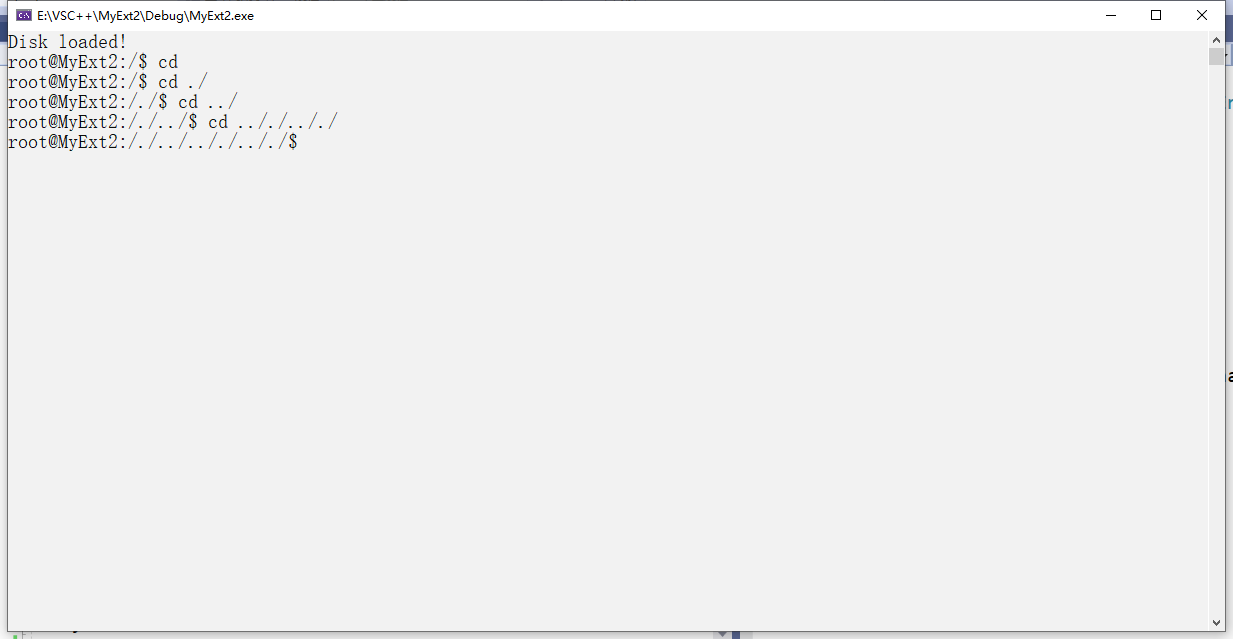


新建文件a.txt，并向内写入一句话，然后使用exit退出程序，关闭文件系统。

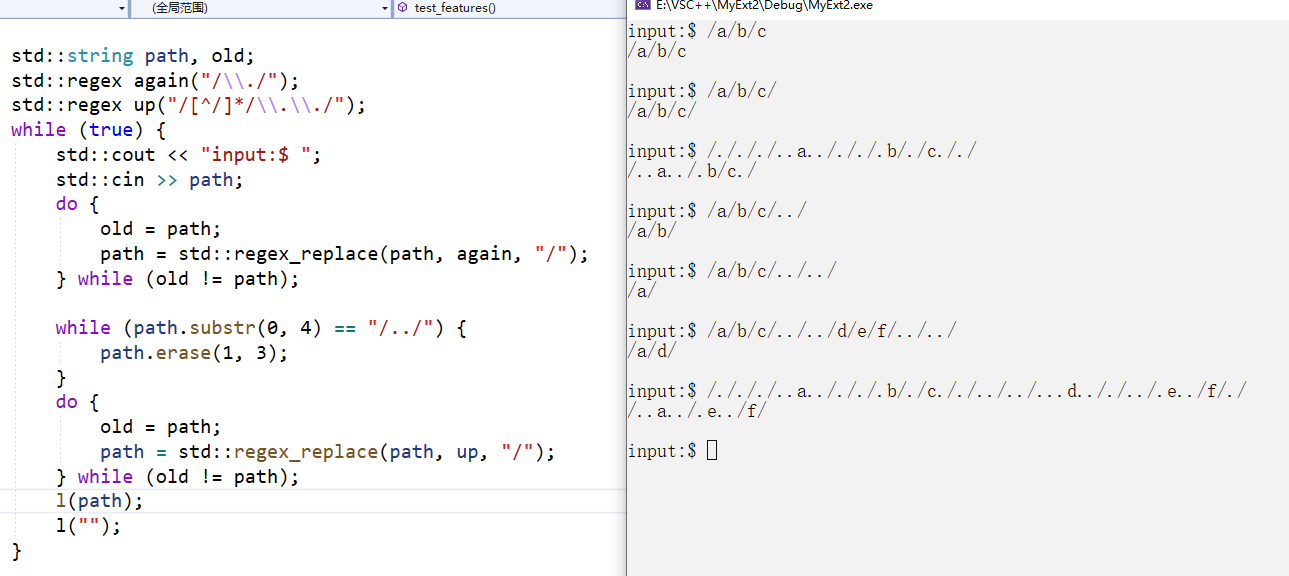
再次打开文件系统，数据依然存在，这证明数据是存储在硬盘中的，无电易失性。

## 3.7实验总结

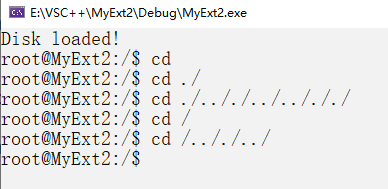
### 3.7.1实验中的问题与解决过程



**问题:路径显示不正确**



关于路径显示的bug修复: 使用正则表达式

效果: 

**问题: mkdir无效, ls不显示新建的文件夹**

解决:路径解析函数 目录项查找函数 均出现问题,已debug; 还有一个原因是添加新的目录项成功, 但是目录文件的长度没有更改, 导致下次读取时还是只能读取到之前的. ..两个目录项(共17字节, 后来添加了文件长度修改的代码后, 能够正常ls显示出来)

20余天中的问题几乎难以列举，这里只列举两个最具代表性的问题及解决。

### 3.7.2实验收获

近1个月的编写过程非常艰难，中途多次重构，颠覆自己对项目的理解，然后把一部分推倒重来。是前两次实验验收时，老师对我的认可与鼓励，才让我坚持了下来，最终形成了这样一个文件系统。但我的信心也因此得到提升，不再惧怕大量代码和较复杂的需求，并且也对Ext2这一经典的文件系统有了十分深刻的理解。

## 3.8附件

### 3.8.1附件1 程序

**pch.h**

#pragma once

#include <iostream>

#include <string>

typedef unsigned short int u16;

typedef unsigned int u32;

typedef unsigned long long int u64;

constexpr u16 BlockSize = 512;//块大小

constexpr u16 InodeSize = 64;//inode大小

constexpr u64 FS\_Size = BlockSize \* (1 + 1 + 1 + InodeSize \* 8 + BlockSize \* 8);//整个文件系统所占磁盘大小

constexpr u16 InodePerBlock = BlockSize / 64;

constexpr u16 DataBlockOffset = 1 + 1 + 1 + InodeSize \* 8;

void l(std::string log);

//定义:磁盘块号从0开始, 如组描述符的块号就为0

//索引节点(inode)号从1开始, 如根目录的inode就为1

//数据块号从0开始, 如根目录的目录文件就在第0块

//实际编写中发现数据块号从0开始很不方便, 后来人建议从1开始

**pch.cpp**

#pragma once

#include "pch.h"

void l(std::string log) {

std::cerr << log << '\n';

}

**Main.cpp**

// Main.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。

// C++17

#include "pch.h"

#include <iostream>

#include <string>

#include "MyExt2.cpp"

#include <regex>

//定义:磁盘块号从0开始, 如组描述符的块号就为0

//索引节点(inode)号从1开始, 如根目录的inode就为1

//数据块号从0开始, 如根目录的目录文件就在第0块

void format(MyExt2& fs) {

std::cout << "WARNING!!! ALL data will be lost!!! continue?(y/N) ";

std::string res;

std::getline(std::cin, res);

if (res == "y" || res == "Y") {

std::cout << "enter new volume name:(default:MyExt2) ";

std::string vn;

std::getline(std::cin, vn);

if (vn == "")

vn = "MyExt2";

if (vn.size() > 15) {

std::cout << "volume name too long! < 16 chars.\n";

}

fs.format(vn);

}

}

void test\_features() {

//std::cout << sizeof(Group\_Descriptor) << ' ' << sizeof(Inode) << ' ' << sizeof(BitMap);

//std::string a("Hi!");

//char b[16] = "123456789012";

//a.copy(b, sizeof b);

//std::cout << b;

//std::string command, token;

//std::regex whites("/");

//std::sregex\_token\_iterator end;

//while (true)

//{

// std::cout << "$ ";

// std::getline(std::cin, command);

// std::sregex\_token\_iterator it(command.begin(), command.end(), whites, -1);

// while (it != end)

// {

// std::cout << it->str() << '\n';

// it++;

// }

//}

//std::string sn = "Hello!";

//char cn[100] = "Hello!";

//char\* pt = cn;

//std::cout << (sn == pt);

//std::string path, old;

//std::regex again("/\\./");

//std::regex up("/[^/]\*/\\.\\./");

//while (true) {

// std::cout << "input:$ ";

// std::cin >> path;

// do {

// old = path;

// path = std::regex\_replace(path, again, "/");

// } while (old != path);

// while (path.substr(0, 4) == "/../") {

// path.erase(1, 3);

// }

// do {

// old = path;

// path = std::regex\_replace(path, up, "/");

// } while (old != path);

// l(path);

// l("");

//}

std::string name;

std::regex ext(".\*(\\.exe|\\.bin|\\.com)");

while (true)

{

std::cout << "$ ";

std::getline(std::cin, name);

if (name.find('.') == std::string::npos)

l("true");

else if (std::regex\_match(name, ext)) {

l("true");

}

else

l("flase");

}

}

int main(int argc, char\* argv[])

{

//test\_features();

MyExt2 test;

while (!test.is\_formatted()) {

std::cout << "New disk, please format it.\n";

format(test);

}

std::string command, user("root"), token;

if (argc > 1)

user = argv[1];

std::regex whites("\\s+");

std::sregex\_token\_iterator end;

while (true)

{

std::cout << user + "@" + test.volume\_name() + ":" + test.curr\_path() + "$ ";

std::getline(std::cin, command);

std::sregex\_token\_iterator it(command.begin(), command.end(), whites, -1);

if (it != end)

token = it->str();

if (token == "") {

if (++it != end)

token = it->str();

if (token == "")

continue;

}

if (token == "ls") {

if (++it == end) {

test.ls("./");

}

else {

do {

token = it->str();

test.ls(token);

} while (++it != end);

}

}

else if (token == "cd") {

if (++it == end) {

continue;

}

token = it->str();

test.cd(token);

}

else if (token == "mkdir") {

if (++it == end) {

l("mkdir: missing operand");

continue;

}

do {

token = it->str();

test.mkdir(token);

} while (++it != end);

}

else if (token == "create") {

if (++it == end) {

l("create: missing operand");

continue;

}

do {

token = it->str();

test.create(token);

} while (++it != end);

}

else if (token == "rm") {

if (++it == end) {

l("rm: missing operand");

continue;

}

do {

token = it->str();

test.rm(token);

} while (++it != end);

}

else if (token == "read") {

if (++it == end) {

l("read: missing operand");

continue;

}

do {

token = it->str();

test.read(token);

} while (++it != end);

}

else if (token == "write") {

if (++it == end) {

l("write: missing operand");

continue;

}

token = it->str();

test.write(token);

}

else if (token == "format") {

format(test);

}

else if (token == "chmod") {

if (++it == end) {

l("chmod: missing mode");

continue;

}

token = it->str();

if (++it == end) {

l("chmod: missing file");

continue;

}

char anum = token[0] - '0';

if (anum > 7) {

l("chmod: not a correct mode!");

continue;

}

std::string file;

do {

file = it->str();

test.chmod(anum, file);

} while (++it != end);

}

else if (token == "exit" || token == "quit") {

std::cout << "OK, exit.\n";

break;

}

else {

std::cout << token + ": command not found\n";

}

}

return 0;

}

**MyExt2.cpp**

#pragma once

#include "pch.h"

#include "structures.cpp"

#include <map>

#include <string>

#include <regex>

#include "files.h"

struct Res

{

bool succ = false;

u16 nodei = 0;

std::string name = "";

u16 parent = 0;

bool dir = false;

std::string path = "";

};

//文件系统的内存数据结构及操作

//"动态的"文件系统

class MyExt2

{

DiskSim disk;

std::map<u16, File\*> fopen\_table;//文件打开表

u16 current\_dir = 0;//当前目录(索引结点）

std::string current\_path = "";//当前路径(字符串)

Group\_Descriptor gdcache;//组描述符的内存缓存

BitMap inode\_map, block\_map;//位图的内存缓存

bool is\_fmt;//是否已格式化

Dir\* rootdir = nullptr, \* parent = nullptr;

File\* file = nullptr;

//将一个path转换为inode号, 此path不能为空串

Res path2inode(std::string path, bool silent = false) {

if (path[0] != '/')

path = current\_path + path;

std::regex split("/");

std::sregex\_token\_iterator end;

std::sregex\_token\_iterator it(path.begin(), path.end(), split, -1);

Dir pp = \*rootdir;

Res re;

bool is\_dir = true;

std::pair<bool, DirEntry> n;

std::string name;

\*parent = \*rootdir;

if (!parent->open("/", 1, rootdir)) {

l("path: open fail!");

return re;

}

if (++it == end) {

name = "/";

n.second.inode = 1;

}

while (it != end)

{

name = it->str();

if (!is\_dir) {

l("path: " + name + " is not a dir!");

return re;

}

parent->read();

n = parent->find(name, silent);

if (!n.first) {

if (!silent)

l("path: no file named " + name);

return re;

}

if (n.second.file\_type != 2) {

is\_dir = false;

if (!file->open(name, n.second.inode, parent)) {

l("path: open fail!");

return re;

}

}

else {

pp = \*parent;

parent->close();

if (!parent->open(name, n.second.inode, &pp)) {

l("path: open fail!");

return re;

}

}

++it;

}

re.name = name;

re.nodei = n.second.inode;

if (is\_dir) {

if (path.back() != '/')

path.push\_back('/');

re.parent = pp.get\_nodei();

}

else {

if (path.back() == '/')

path.pop\_back();

re.parent = parent->get\_nodei();

}

std::string old;

std::regex again("/\\./");

std::regex up("/[^/]\*/\\.\\./");

do {

old = path;

path = std::regex\_replace(path, again, "/");

} while (old != path);

while (path.substr(0, 4) == "/../") {

path.erase(1, 3);

}

do {

old = path;

path = std::regex\_replace(path, up, "/");

} while (old != path);

re.path = path;

re.dir = is\_dir;

re.succ = true;

return re;

}

public:

MyExt2()

:inode\_map(true), block\_map(false) {

is\_fmt = !disk.is\_new();

if (is\_fmt) {

disk.read(0, (char\*)&gdcache);

disk.read(gdcache.block\_bitmap, block\_map.pointer());

disk.read(gdcache.inode\_bitmap, inode\_map.pointer());

current\_dir = 1;

current\_path = "/";

rootdir = new Dir(&disk, &inode\_map, &block\_map, &gdcache, nullptr, &fopen\_table);

rootdir->open("/", 1, rootdir);

parent = new Dir(&disk, &inode\_map, &block\_map, &gdcache, rootdir, &fopen\_table);

file = new File(&disk, &inode\_map, &block\_map, &gdcache, rootdir, &fopen\_table);

}

}

std::string curr\_path() const {

return current\_path;

}

std::string volume\_name() {

return gdcache.volume\_name;

}

bool is\_formatted() {

return is\_fmt;

}

//格式化

//数据全部清零, 重置控制字段, 并初始化根目录, 及其他杂项

void format(std::string vn) {

disk.clear();

fopen\_table.clear();

Group\_Descriptor gd;

vn.copy(gd.volume\_name, sizeof(gd.volume\_name)-1);

gdcache = gd;

//gdcache.used\_dirs\_count++;

//gdcache.free\_blocks\_count--;

//gdcache.free\_inodes\_count--;

BitMap db(false), in(true);

block\_map = db;

inode\_map = in;

//新建根目录

Inode root\_inode(2, 7);

root\_inode.i\_blocks = 1;

root\_inode.i\_size = 17;

root\_inode.i\_block[0] = 0;

//this->set\_inode(root\_inode, 1);

char block[BlockSize] = { 0 };

\*((Inode\*)block) = root\_inode;

disk.write(gdcache.inode\_table, block);

DirEntry p(1,".",2), pp(1, "..", 2);

char temp[BlockSize] = { 0 }, \* pt = temp;

\*((DirEntry\*)pt) = p;

pt += p.rec\_len;

\*((DirEntry\*)pt) = pp;

disk.write(0 + DataBlockOffset, temp);

current\_dir = 1;

current\_path = "/";

gdcache.free\_blocks\_count--;

gdcache.free\_inodes\_count--;

gdcache.used\_dirs\_count++;

block\_map.set\_bit(0);

inode\_map.set\_bit(1);

disk.write(0, (const char\*)&gdcache);

disk.write(gdcache.block\_bitmap, block\_map.pointer());

disk.write(gdcache.inode\_bitmap, inode\_map.pointer());

is\_fmt = true;

rootdir = new Dir(&disk, &inode\_map, &block\_map, &gdcache, nullptr, &fopen\_table);

rootdir->open("/", 1, rootdir);

parent = new Dir(&disk, &inode\_map, &block\_map, &gdcache, rootdir, &fopen\_table);

file = new File(&disk, &inode\_map, &block\_map, &gdcache, rootdir, &fopen\_table);

}

void ls(std::string path) {

Res in = path2inode(path);

if (!in.succ) {

std::cout << "ls: cannot access \'" + path + "\': No such file or directory\n";

}

else {

std::cout << path + ":\n";

if (in.dir) {

parent->read();

parent->print();

}

else {

file->print();

}

}

}

void cd(std::string path) {

Res in = path2inode(path);

if (!in.succ) {

std::cout << "cd: cannot access \'" + path + "\': No such file or directory\n";

}

else {

if (in.dir) {

current\_dir = in.nodei;

current\_path = in.path;

}

else {

l("cd: \'" + path + "\': not a directory");

}

}

}

void mkdir(std::string path) {

if (path2inode(path, true).succ) {

l("mkdir: " + path + " already exist!");

return;

}

if (path[0] != '/')

path = current\_path + path;

std::regex split("/");

std::sregex\_token\_iterator end;

std::sregex\_token\_iterator it(path.begin(), path.end(), split, -1);

std::string name;

while (it != end) {

name = it->str();

it++;

}

if (path.back() == '/') {

path.pop\_back();

}

path.resize(path.size() - name.size());

Res in = path2inode(path);

if (!in.succ) {

l("mkdir: dir path fail!");

return;

}

if (!in.dir) {

l("mkdir: cannot mkdir under a file!");

return;

}

parent->read();

Dir mk(&disk, &inode\_map, &block\_map, &gdcache, parent, &fopen\_table);

Inode ino(2);

mk.create(name, ino);

//mk.close();

parent->close();

}

void create(std::string path) {

if (path2inode(path, true).succ) {

l("create: " + path + " already exist!");

return;

}

if (path[0] != '/')

path = current\_path + path;

std::regex split("/");

std::sregex\_token\_iterator end;

std::sregex\_token\_iterator it(path.begin(), path.end(), split, -1);

std::string name;

while (it != end) {

name = it->str();

it++;

}

if (path.back() == '/') {

path.pop\_back();

}

path.resize(path.size() - name.size());

Res in = path2inode(path);

if (!in.succ) {

l("create: dir path fail!");

return;

}

if (!in.dir) {

l("create: cannot create under a file!");

return;

}

parent->read();

File mk(&disk, &inode\_map, &block\_map, &gdcache, parent, &fopen\_table);

Inode ino(1, 6);

std::regex ext(".\*(\\.exe|\\.bin|\\.com)");

if (name.find('.') == std::string::npos)

ino.set\_access(7);

else if (std::regex\_match(name, ext)) {

ino.set\_access(7);

}

mk.create(name, ino);

//mk.close();

parent->close();

}

void rm(std::string path) {

Res full = path2inode(path);

if (!full.succ) {

l("rm: " + path + " did not exist!");

return;

}

if (path[0] != '/')

path = current\_path + path;

std::regex split("/");

std::sregex\_token\_iterator end;

std::sregex\_token\_iterator it(path.begin(), path.end(), split, -1);

std::string name;

while (it != end) {

name = it->str();

it++;

}

if (path.back() == '/') {

path.pop\_back();

}

path.resize(path.size() - name.size());

Res in = path2inode(path);

if (!in.succ) {

l("rm: dir path fail!");

return;

}

if (!in.dir) {

l("rm: cannot rm under a file!");

return;

}

parent->read();

if (!full.dir) {

File ff(&disk, &inode\_map, &block\_map, &gdcache, parent, &fopen\_table);

ff.open(full.name, full.nodei, parent);

ff.del();

}

else {

Dir dir(&disk, &inode\_map, &block\_map, &gdcache, parent, &fopen\_table);

dir.open(full.name, full.nodei, parent);

dir.del();

}

//mk.close();

parent->close();

}

void read(std::string path) {

Res in = path2inode(path);

if (!in.succ) {

std::cout << "read: cannot access \'" + path + "\': No such file or directory\n";

}

else {

std::cout << path + ":\n";

if (in.dir) {

l("read: cannot read a dir! consider ls instead!");

}

else {

auto con = file->read();

std::string skr;

skr.assign(con.first, con.second);

std::cout << skr << '\n';

file->close();

}

}

}

void write(std::string path) {

Res in = path2inode(path);

if (!in.succ) {

std::cout << "write: cannot access \'" + path + "\': No such file or directory\n";

}

else {

std::cout << path + ":\n";

if (in.dir) {

l("write: cannot write a dir!!");

}

else {

auto con = file->read();

std::string skr;

std::getline(std::cin, skr);

char\* midd = new char[skr.size() + 1];

strcpy\_s(midd, skr.size() + 1, skr.c\_str());

file->write(midd, skr.size() + 1);

delete[] midd;

file->close();

}

}

}

void chmod(char mode, std::string path) {

Res in = path2inode(path);

if (!in.succ) {

std::cout << "chmod: cannot access \'" + path + "\': No such file or directory\n";

}

else {

if (in.dir) {

l("chmod: change a dir's mode may influence all files under it!!");

parent->chmod(mode);

parent->close();

}

else {

file->chmod(mode);

file->close();

}

}

}

~MyExt2()

{

while (fopen\_table.size() > 0) {

auto it = fopen\_table.begin();

if (!it->second->close())

l("close fail!may lose data!");

}

disk.write(0, (const char\*)&gdcache);

disk.write(gdcache.block\_bitmap, block\_map.pointer());

disk.write(gdcache.inode\_bitmap, inode\_map.pointer());

}

};

**Files.h**

#pragma once

#include "pch.h"

#include "structures.cpp"

#include <string>

#include <utility>

#include <map>

#include <iostream>

class Dir;

class File {

protected:

DiskSim\* disk = nullptr;

BitMap\* inode\_map = nullptr, \* block\_map = nullptr;

Group\_Descriptor\* gd = nullptr;

Dir\* parent = nullptr;

std::map<u16, File\*>\* fopen\_table;

u16 node\_index = 0;

Inode inode;

std::string name = "";

char\* buffer = nullptr;

u32 len = 0, buflen = 0;

bool has\_open = false, dirty = false, has\_read = false;

//读取index号对应的inode索引结构

bool read\_inode(u16 nodei);

//写入此inode

bool write\_inode();

//对某Inode节点node, 在索引号为node.i\_blocks处, 添加一个数据块索引, block是数据块号

//由于过程中可能会申请间接索引块,磁盘满后可能失败,故返回是否成功

//若同时申请一二级间接索引块,二级失败, 会进行回退,避免1级索引块变为"死块"

bool add\_block(u16 block);

//减去一个索引号为node.i\_blocks的数据块索引

bool sub\_block();

//对某Inode节点node, 用索引号index, 获取一个数据块号

u16 get\_block(u16 index);

public:

//构造函数待定

File(DiskSim\* dsk, BitMap\* ino\_map, BitMap\* blk\_map, Group\_Descriptor\* gdc, Dir\* par, std::map<u16, File\*>\* fot);

/\*新建文件时要做什么:

MyExt2::inode\_map找一个空的nodei并占用

MyExt2::gd.free\_inodes\_count--

(若是目录文件, MyExt2::gd.used\_dirs\_count++, 并设. ..两项)

构建一个inode并写入磁盘

在父目录添加一个目录项

\*/

bool create(std::string nm, Inode ino);

/\*

打开文件要做什么:

插入MyExt2::fopen\_table

从磁盘获取inode

修改访问时间

\*/

bool open(std::string nm, u16 nodei, Dir\* pa);

/\*

读文件:

将文件分散在各数据块的内容连接起来, 放在缓冲区里

返回头指针和长度

\*/

std::pair<const char\*, u32> read();

/\*

写文件:

输入字符指针和长度

将字符流写入缓冲区

\*/

bool write(char\* str, u32 strlen);

bool append(char\* str, u32 applen);

bool change(char\* str, u32 begin, u32 end);

/\*

关闭:

为写入缓冲区至硬盘, 在MyExt2::block\_map, gd中申请/释放数据块

将缓冲区写入各分散数据块中

并改变inode.size, i\_blocks, 修改时间等

写入inode

删除MyExt2::fopen\_table

\*/

bool close();

/\*

删除:

释放MyExt2::inode\_map, block\_map, 修改gd

重置此inode

在父目录中删除此目录项

\*/

bool del();

operator bool()const;

bool print();

bool chmod(char mode);

u16 get\_nodei() {

return this->node\_index;

}

};

//管理DirEntry

class Dir :public File {

protected:

//目录项指针

int offset = 0, end = -1;

DirEntry temp;

public:

//对继承方法的修改

bool create(std::string nm, Inode ino);

bool del();

protected:

//低级api

bool ready();

bool head();

bool next();

bool alive();

DirEntry get\_this();

//change后如果原位不能存放,则可能移动其位置

bool set\_this(DirEntry de);

bool del\_this();

bool \_find(u16 nodei);

bool \_find(std::string nm, bool silent = false);

public:

Dir(DiskSim\* dsk, BitMap\* ino\_map, BitMap\* blk\_map, Group\_Descriptor\* gdc, Dir\* par, std::map<u16, File\*>\* fot);

//更高级的api

bool add(DirEntry de);

bool remove(u16 nodei);

bool remove(std::string nm);

std::pair<bool, DirEntry> find(u16 nodei);

std::pair<bool, DirEntry> find(std::string nm, bool silent = false);

bool change\_de(std::string nm, DirEntry de);

bool change\_de(u16 nodei, DirEntry de);

bool print();

//void get\_all() {}

};

**files.cpp**

#pragma once

#include "files.h"

class Dir;

template<class T>

T max(T a, T b) {

return (a > b) ? a : b;

}

u32 ceiling(u32 a) {

return (a % BlockSize == 0) ? a : a - a % BlockSize + BlockSize;

}

bool File::read\_inode(u16 nodei) {

Inode\* pt;

if (nodei == 0 || nodei >= BlockSize \* 8) {

std::cerr << "File::read\_inode(u16 nodei) out of range\n";

return false;

}

if (!inode\_map->get\_bit(nodei)) {

std::cerr << "File::read\_inode(u16 nodei) inode not used\n";

return false;

}

char block[BlockSize] = { 0 };

disk->read((nodei - 1) / InodePerBlock + 3, block);

pt = ((Inode\*)block) + ((nodei - 1) % InodePerBlock);

this->inode = \*pt;

return true;

}

bool File::write\_inode() {

if (this->node\_index == 0 || this->node\_index >= BlockSize \* 8) {

std::cerr << "void write\_inode() out of range\n";

return false;

}

Inode\* pt;

char block[BlockSize] = { 0 };

disk->read((this->node\_index - 1) / InodePerBlock + 3, block);

pt = ((Inode\*)block) + ((this->node\_index - 1) % InodePerBlock);

\*pt = this->inode;

disk->write((this->node\_index - 1) / InodePerBlock + 3, block);

return true;

}

bool File::add\_block(u16 block) {

if (this->has\_open) {

u16 index = this->inode.i\_blocks;

if (index < 6) {

this->inode.i\_block[index] = block;

}

//这里假设间接索引块已经建立, 实际上不行,下同且有两次

//比较复杂的是数据块号从0起始,如何判断间接索引块是否存在?可能需要利用0号必为根目录的目录内容来做/this->inode.i\_blocks限制+及时设置

//可能在设置一个数据块索引的同时,需要再多次分配给路径上的索引块以可用块号,这需要块位图,所以应怎么做?

//解决方法:只允许在末尾this->inode.i\_blocks添加一个索引,用到间接索引块时立即分配,用this->inode.i\_blocks来判断间接索引块是否已建立

//因为需要用到dsk,bmp等MyExt2的成员,这个方法应该挪过去

//并且分配间接索引块后也要修改组描述符 块位图等

//已解决, 以上供纪念

else if (index < 6 + BlockSize / 2) {

IndexBlock i1;

index -= 6;

if (index == 0) {

int ib = this->block\_map->find\_zeros(0, 1);

if (ib < 0)

return false;

this->gd->free\_blocks\_count--;

this->inode.i\_block[6] = ib;

i1.index[index] = block;

this->disk->write(this->inode.i\_block[6] + DataBlockOffset, (char\*)&i1);

}

else {

this->disk->read(this->inode.i\_block[6] + DataBlockOffset, (char\*)&i1);

i1.index[index] = block;

this->disk->write(this->inode.i\_block[6] + DataBlockOffset, (char\*)&i1);

}

}

else if (index < 6 + BlockSize / 2 + BlockSize \* BlockSize / 4) {

IndexBlock i1, i2;

index -= 6 + BlockSize / 2;

if (index == 0) {

int ib1 = this->block\_map->find\_zeros(0, 1);

int ib2 = this->block\_map->find\_zeros(0, 1);

if (ib1 < 0) {

return false;

}

else if (ib2 < 0) {

this->block\_map->reset\_bit(ib1);

return false;

}

this->gd->free\_blocks\_count -= 2;

this->inode.i\_block[7] = ib1;

i1.index[0] = ib2;

i2.index[0] = block;

this->disk->write(this->inode.i\_block[7] + DataBlockOffset, (char\*)&i1);

this->disk->write(i1.index[0] + DataBlockOffset, (char\*)&i2);

}

else if (index % (BlockSize / 2) == 0) {

int ib2 = this->block\_map->find\_zeros(0, 1);

if (ib2 < 0) {

return false;

}

this->gd->free\_blocks\_count--;

this->disk->read(this->inode.i\_block[7] + DataBlockOffset, (char\*)&i1);

i1.index[index / (BlockSize / 2)] = ib2;

i2.index[0] = block;

this->disk->write(this->inode.i\_block[7] + DataBlockOffset, (char\*)&i1);

this->disk->write(i1.index[index / (BlockSize / 2)] + DataBlockOffset, (char\*)&i2);

}

else {

this->disk->read(this->inode.i\_block[7] + DataBlockOffset, (char\*)&i1);

this->disk->read(i1.index[index / (BlockSize / 2)] + DataBlockOffset, (char\*)&i2);

i2.index[index % (BlockSize / 2)] = block;

this->disk->write(i1.index[index / (BlockSize / 2)] + DataBlockOffset, (char\*)&i2);

}

}

else {

std::cerr << "add\_block(u16 block) out of range\n";

return false;

}

this->inode.i\_blocks++;

return true;

}

else {

std::cerr << "add\_block(u16 block) not open yet\n";

return false;

}

}

bool File::sub\_block() {

if (this->inode.i\_blocks == 0) {

std::cerr << "sub\_block() no block already!\n";

return false;

}

if (!this->has\_open) {

l("sub\_block(u16 block) not open yet!");

return false;

}

u16 index = this->inode.i\_blocks - 1, blk;

if (index < 6) {

blk = this->inode.i\_block[index];

}

else if (index < 6 + BlockSize / 2) {

IndexBlock i1;

index -= 6;

this->disk->read(this->inode.i\_block[6] + DataBlockOffset, (char\*)&i1);

blk = i1.index[index];

if (index == 0) {

this->block\_map->reset\_bit(this->inode.i\_block[6]);

this->gd->free\_blocks\_count++;

}

}

else if (index < 6 + BlockSize / 2 + BlockSize \* BlockSize / 4) {

IndexBlock i1, i2;

index -= 6 + BlockSize / 2;

this->disk->read(this->inode.i\_block[7] + DataBlockOffset, (char\*)&i1);

this->disk->read(i1.index[index / (BlockSize / 2)] + DataBlockOffset, (char\*)&i2);

blk = i2.index[index % (BlockSize / 2)];

if (index == 0) {

this->block\_map->reset\_bit(i1.index[index / (BlockSize / 2)]);

this->block\_map->reset\_bit(this->inode.i\_block[7]);

this->gd->free\_blocks\_count += 2;

}

else if (index % (BlockSize / 2) == 0) {

this->block\_map->reset\_bit(i1.index[index / (BlockSize / 2)]);

this->gd->free\_blocks\_count++;

}

}

else {

std::cerr << "sub\_block(u16 block) out of range!\n";

return false;

}

this->block\_map->reset\_bit(blk);

this->gd->free\_blocks\_count++;

this->inode.i\_blocks--;

return true;

}

u16 File::get\_block(u16 index) {

if (has\_open) {

if (index > this->inode.i\_blocks) {

std::cerr << "index in i\_node.get\_block() out of range!please check\n";

return 0;

}

if (index < 6) {

return this->inode.i\_block[index];

}

else if (index < 6 + BlockSize / 2) {

IndexBlock i1;

index -= 6;

this->disk->read(this->inode.i\_block[6] + DataBlockOffset, (char\*)&i1);

return i1.index[index];

}

else {

IndexBlock i1, i2;

index -= 6 + BlockSize / 2;

this->disk->read(this->inode.i\_block[7] + DataBlockOffset, (char\*)&i1);

this->disk->read(i1.index[index / (BlockSize / 2)] + DataBlockOffset, (char\*)&i2);

return i2.index[index % (BlockSize / 2)];

}

}

else {

std::cerr << "u16 get\_block(u16 index) not open yet\n";

return false;

}

}

File::File(DiskSim\* dsk, BitMap\* ino\_map, BitMap\* blk\_map, Group\_Descriptor\* gdc, Dir\* par, std::map<u16, File\*>\* fot)

:disk(dsk), inode\_map(ino\_map), block\_map(blk\_map), gd(gdc), parent(par), fopen\_table(fot) {}

bool File::create(std::string nm, Inode ino) {

int nodei = this->inode\_map->find\_zeros(1, 1);

if (nodei <= 0) {

std::cerr << "inode full! create fail\n";

return false;

}

this->node\_index = nodei;

this->gd->free\_inodes\_count--;

this->inode = ino;

this->name = nm;

this->write\_inode();

DirEntry de(this->node\_index, this->name, this->inode.get\_type());

return this->parent->add(de);

}

bool File::open(std::string nm, u16 nodei, Dir\* pa) {

if (nodei == 0 || nodei >= BlockSize \* 8) {

std::cerr << "bool open() out of range\n";

return false;

}

this->fopen\_table->insert({ nodei, this });

if (!this->read\_inode(nodei)) {

std::cerr << "bool open() read\_inode fail\n";

return false;

}

has\_open = true;

has\_read = false;

dirty = false;

this->node\_index = nodei;

this->name = nm;

this->parent = pa;

this->inode.access();

return true;

}

std::pair<const char\*, u32> File::read() {

char mode = this->inode.get\_access();

if (((mode >> 2) & 0x01) == 0) {

l("File::read(): file doesn't have read permission!");

return { nullptr, 0 };

}

if (has\_open) {

if (this->dirty) {

return { (const char\*)this->buffer, this->len };

}

else {

this->len = this->inode.i\_size;

this->buflen = max<u32>(this->inode.i\_blocks \* BlockSize, this->len \* 3 / 2);

char\* buf = new char[this->buflen];

char\* curr = buf;

u16 block = 0;

for (int i = 0; i < this->inode.i\_blocks; i++) {

block = this->get\_block(i);

this->disk->read(block + DataBlockOffset, curr);

curr += BlockSize;

}

std::pair<const char\*, u32> res = { (const char\*)buf, this->len };

this->buffer = buf;

this->dirty = false;

this->has\_read = true;

return res;

}

}

else {

std::cerr << "read() not open yet\n";

return { nullptr, 0 };

}

}

bool File::write(char\* str, u32 strlen) {

char mode = this->inode.get\_access();

if (((mode >> 1) & 0x01) == 0) {

l("File::write(): file doesn't have write permission!");

return false;

}

if (this->has\_open) {

if (!this->has\_read) {

l("write() not read yet");

return false;

}

if (strlen > this->buflen) {

u32 new\_buflen = max<u32>(strlen \* 3 / 2, ceiling(strlen));

if (this->buflen)

delete this->buffer;

this->buffer = new char[new\_buflen];

this->buflen = new\_buflen;

}

for (int i = 0; i < strlen; i++) {

\*(this->buffer + i) = \*(str + i);

}

this->len = strlen;

dirty = true;

return true;

}

else {

std::cerr << "write() not open yet\n";

return false;

}

}

bool File::append(char\* str, u32 applen) {

char mode = this->inode.get\_access();

if (((mode >> 1) & 0x01) == 0) {

l("File::write(): file doesn't have write permission!");

return false;

}

if (has\_open) {

if (!this->has\_read) {

l("append() not read yet");

return false;

}

u32 strlen = applen + this->len;

if (strlen > this->buflen) {

u32 new\_buflen = max<u32>(strlen \* 3 / 2, ceiling(strlen));

char\* new\_buf = new char[new\_buflen];

for (int i = 0; i < this->len; i++) {

\*(new\_buf + i) = \*(this->buffer + i);

}

for (int i = 0; i < applen; i++) {

\*(new\_buf + this->len + i) = \*(str + i);

}

if (this->buflen)

delete this->buffer;

this->buflen = new\_buflen;

this->buffer = new\_buf;

}

else {

for (int i = 0; i < applen; i++) {

\*(this->buffer + this->len + i) = \*(str + i);

}

}

this->len = strlen;

dirty = true;

return true;

}

else {

std::cerr << "write() not open yet\n";

return false;

}

}

bool File::change(char\* str, u32 begin, u32 end) {

char mode = this->inode.get\_access();

if (((mode >> 1) & 0x01) == 0) {

l("File::write(): file doesn't have write permission!");

return false;

}

if (!this->has\_read) {

l("change() not read yet");

return false;

}

if (begin > end) {

u32 temp = begin;

begin = end;

end = temp;

}

if (end <= this->len) {

for (int i = 0; i < end - begin; i++) {

\*(this->buffer + begin + i) = \*(str + i);

}

this->dirty = true;

return true;

}

else if (begin == this->len) {

if (!this->append(str, end - begin))

return false;

return true;

}

else if (begin > this->len) {

char\* zeros = new char[begin - this->len];

for (int i = 0; i < begin - this->len; i++) {

\*(zeros + i) = 0;

}

if (!this->append(zeros, begin - this->len))

return false;

if (!this->append(str, end - begin))

return false;

return true;

}

else {

if (!this->change(str, begin, this->len))

return false;

if (!this->append(str + this->len - begin, end - this->len))

return false;

return true;

}

}

bool File::close() {

if (this->has\_open) {

if (this->dirty) {

u32 buf\_blocks = ceiling(this->len) / BlockSize;

if (this->gd->free\_blocks\_count + this->inode.i\_blocks < buf\_blocks) {

std::cerr << "bool File::close() no extra space!\n";

delete this->buffer;

return false;

}

if (buf\_blocks > this->inode.i\_blocks) {

int blk = 0, need = buf\_blocks - this->inode.i\_blocks;

u16 last\_pos = 0, query = need;

if (this->inode.i\_blocks != 0)

last\_pos = this->get\_block(this->inode.i\_blocks - 1) + 1;

do {

blk = this->block\_map->find\_zeros(last\_pos, query);

if (blk >= 0) {

for (int i = 0; i < query; i++) {

if (!this->add\_block(i + blk)) {

std::cerr << "add\_block() failed\n";

delete this->buffer;

return false;

}

}

need -= query;

last\_pos = blk + query;

query = (query > need) ? need : query;

}

else {

query = -blk - 1;

}

} while (need > 0);

}

else if (buf\_blocks < this->inode.i\_blocks) {

for (int i = 0; i < this->inode.i\_blocks - buf\_blocks; i++) {

this->sub\_block();

}

}

if (this->inode.i\_blocks != buf\_blocks) {

l("bool file::close() did not set the right inode.i\_blocks!");

delete this->buffer;

return false;

}

//写入

char\* curr = this->buffer;

for (int i = 0; i < buf\_blocks; i++) {

this->disk->write(this->get\_block(i) + DataBlockOffset, curr);

curr += BlockSize;

}

this->dirty = false;

this->inode.modify(this->len);

}

delete this->buffer;

this->len = 0;

this->buflen = 0;

this->has\_read = false;

this->write\_inode();

this->fopen\_table->erase(this->node\_index);

this->has\_open = false;

return true;

}

else {

std::cerr << "bool File::close() not open yet!\n";

this->fopen\_table->erase(this->node\_index);

return false;

}

}

bool File::del() {

if (!\*this) {

l("nodei out of range");

return false;

}

if (!this->has\_open) {

l("File::del need to rm blocks, please open first!");

return false;

}

while (this->inode.i\_blocks > 0) {

this->sub\_block();

}

delete this->buffer;

this->fopen\_table->erase(this->node\_index);

this->inode\_map->reset\_bit(this->node\_index);

this->gd->free\_inodes\_count++;

this->has\_open = false;

this->parent->remove(this->node\_index);

return true;

}

File::operator bool() const {

return !(this->node\_index == 0 || this->node\_index >= BlockSize \* 8);

}

bool File::print() {

if (!this->has\_open) {

std::cerr << "bool File::print() not open yet!\n";

return false;

}

this->inode.print();

std::cout << '\n';

return true;

}

bool File::chmod(char mode) {

if (!this->has\_open) {

std::cerr << "bool File::chmod() not open yet!\n";

return false;

}

this->inode.set\_access(mode);

return true;

}

bool Dir::create(std::string nm, Inode ino) {

if (!File::create(nm, ino))

return false;

this->gd->used\_dirs\_count++;

int blk;

blk = this->block\_map->find\_zeros(0, 1);

if (blk < 0) {

l("create ./ ../ fail:no block!");

return false;

}

DirEntry p(this->node\_index, ".", 2), pp(this->parent->node\_index, "..", 2);

char b[BlockSize] = { 0 };

char\* bp = b;

\*((DirEntry\*)bp) = p;

bp += p.rec\_len;

\*((DirEntry\*)bp) = pp;

this->disk->write(blk + DataBlockOffset, b);

this->inode.i\_size = p.rec\_len + pp.rec\_len;

this->inode.i\_blocks = 1;

this->inode.i\_block[0] = blk;

this->write\_inode();

return true;

}

bool Dir::del() {

if (!File::del())

return false;

this->gd->used\_dirs\_count--;

return true;

}

bool Dir::ready() {

if (!this->has\_open) {

l("dir not open yet!");

return false;

}

if (!this->has\_read) {

l("dir not read yet!");

return false;

}

return true;

}

bool Dir::head() {

if (!this->ready())

return false;

this->offset = 0;

if (!this->temp.is\_alive(this->buffer).first) {

l("a dir without ./");

return false;

}

return true;

}

bool Dir::next() {

if (!this->ready())

return false;

if (end != -1 && offset == end)

return false;

while (true) {

char\* n = this->temp.next\_head(this->buffer + this->offset, (u64)this->len - this->offset);

if (n == nullptr) {

this->end = this->offset;

return false;

}

auto r = this->temp.is\_alive(n);

if (r.second < 7 || r.second>261) {

this->end = this->offset;

return false;

}

this->offset = n - this->buffer;

if (r.first) {

return true;

}

}

}

bool Dir::alive() {

if (!this->ready())

return false;

auto r = temp.is\_alive(buffer + offset);

if (r.second < 7 || r.second>261) {

return false;

}

return r.first;

}

DirEntry Dir::get\_this() {

temp.init(buffer + offset);

return temp;

}

bool Dir::set\_this(DirEntry de) {

if (!this->ready())

return false;

temp.init(buffer + offset);

if (de.rec\_len <= temp.rec\_len) {

de.rec\_len = temp.rec\_len;

change((char\*)&de, offset, offset + de.rec\_len);

return true;

}

else {

del\_this();

return add(de);

}

}

bool Dir::del\_this() {

if (!this->ready())

return false;

int off = this->offset;

if (!next())

end = -1;

offset = off;

u16\* nodi = (u16\*)(buffer + offset);

if (\*nodi == node\_index || \*nodi == parent->node\_index) {

l("cannot remove ./ or ../");

return false;

}

\*nodi = 0;

this->dirty = true;

offset = 0;

return true;

}

bool Dir::\_find(u16 nodei) {

if (!this->ready())

return false;

head();

do {

temp.init(buffer + offset);

if (temp.inode == nodei) {

return true;

}

} while (next());

l("no such de!");

return false;

}

bool Dir::\_find(std::string nm, bool silent) {

if (!this->ready())

return false;

head();

do {

temp.init(buffer + offset);

if (nm == temp.name)

return true;

} while (next());

if (!silent)

l("no such de!");

return false;

}

Dir::Dir(DiskSim\* dsk, BitMap\* ino\_map, BitMap\* blk\_map, Group\_Descriptor\* gdc, Dir\* par, std::map<u16, File\*>\* fot)

:File(dsk, ino\_map, blk\_map, gdc, par, fot) {}

bool Dir::add(DirEntry de) {

if (!this->ready())

return false;

if (end == -1) {

while (next());

}

temp.init(buffer + end);

return change((char\*)&de, end + temp.rec\_len, end + temp.rec\_len + de.rec\_len);

}

bool Dir::remove(u16 nodei) {

if (!\_find(nodei))

return false;

del\_this();

return true;

}

bool Dir::remove(std::string nm) {

if (!\_find(nm))

return false;

del\_this();

return true;

}

std::pair<bool, DirEntry> Dir::find(u16 nodei) {

if (!\_find(nodei))

return { false, DirEntry() };

return { true,get\_this() };

}

std::pair<bool, DirEntry> Dir::find(std::string nm, bool silent) {

if (!\_find(nm, silent))

return { false, DirEntry() };

return { true,get\_this() };

}

bool Dir::change\_de(std::string nm, DirEntry de) {

if (!\_find(nm))

return false;

set\_this(de);

return true;

}

bool Dir::change\_de(u16 nodei, DirEntry de) {

if (!\_find(nodei))

return false;

set\_this(de);

return true;

}

bool Dir::print() {

if (!this->ready())

return false;

Inode store = inode;

head();

do {

temp.init(buffer + offset);

read\_inode(temp.inode);

inode.print();

std::cout << '\t' << temp.name << '\n';

} while (next());

inode = store;

return true;

}

**structures.cpp**

#pragma once

#include "pch.h"

#include <iostream>

#include <ctime>

#include <fstream>

#include <utility>

//文件系统的数据结构及其相关操作

//硬盘上的"静态的"文件系统

//用当前系统下的一个文件FS.txt, 模拟一个FS\_Size大小的硬盘.

//每次读写BlockSize大小的一块数据

class DiskSim

{

std::fstream disk;//连接文件的流

bool old = true;//由于文件系统首次启动和再次启动的行为可能不同, 所以需要保留,传递判断结果

char buffer[BlockSize] = { 0 };//缓冲区

u16 bufpos = 0;

bool buf\_dirty = true;

public:

//(若无文件则)创建文件, 打开文件并连接到流

DiskSim()

{

std::fstream exist;

exist.open("FS.txt", std::ios::in | std::ios::binary);

if (!exist.is\_open())

{

old = false;

exist.clear();

//exist.close();

exist.open("FS.txt", std::ios::out | std::ios::binary);

if (exist.is\_open())

{

for (int i = 0; i < FS\_Size / BlockSize; i++)

{

exist.write(buffer, BlockSize);

}

std::cout << "Disk has been initialized with full 0!\n";

}

else

{

std::cerr << "Create disk(FS.txt) failed!\n";

exit(-1);

}

}

exist.close();

disk.open("FS.txt", std::ios::out | std::ios::in | std::ios::binary);

if (disk.is\_open()) {

std::cout << "Disk loaded!\n";

}

else

{

std::cerr << "Failed to load the disk(FS.txt)!\n";

exit(-1);

}

}

//从硬盘读到缓冲区,如有指针再复制过去

//block\_num是硬盘块号, 使用数据块号时注意+DataBlockOffset

void read(u16 block\_num, char\* s = nullptr)

{

if (buf\_dirty || block\_num != bufpos)

{

disk.seekg((u64)BlockSize \* block\_num);

disk.read(buffer, BlockSize);

bufpos = block\_num;

buf\_dirty = false;

}

if (s != nullptr) {

for (int i = 0; i < BlockSize; i++)

{

s[i] = buffer[i];

}

}

}

//将缓冲区写入硬盘; 如有指针, 先从指针写到缓冲区

//block\_num是硬盘块号, 使用数据块号时注意+DataBlockOffset

void write(u16 block\_num, const char\* s = nullptr)

{

if (s != nullptr) {

for (int i = 0; i < BlockSize; i++)

{

buffer[i] = s[i];

}

buf\_dirty = true;

}

if (buf\_dirty || block\_num != bufpos)

{

disk.seekp((u64)BlockSize \* block\_num);

disk.write(buffer, BlockSize);

buf\_dirty = false;

bufpos = block\_num;

}

}

//硬盘是否是新建的

bool is\_new() {

return !old;

}

//数据全部清零

void clear() {

char zerob[BlockSize] = { 0 };

for (int i = 0; i < FS\_Size / BlockSize; i++)

{

disk.seekp(0);

disk.write(zerob, BlockSize);

}

std::cout << "Disk has been initialized with full 0!\n";

}

~DiskSim()

{

disk.close();

}

};

//位图块结构

class BitMap

{

u64 bits[BlockSize / 8] = { 0 };

bool start;

public:

BitMap(bool is\_inode) {

this->start = is\_inode;

}

//置pos位为1

void set\_bit(u16 pos) {

pos -= start;

if (pos >=4096)

l("-1 for uint!");

u64 t = 0x01;

bits[pos / 64] |= t << (pos % 64);

}

//重置pos位为0

void reset\_bit(u16 pos) {

pos -= start;

if (pos >= 4096)

l("-1 for uint!");

u64 t = 0x01;

bits[pos / 64] &= ~(t << (pos % 64));

}

//获取位图首地址,用于读写磁盘

char\* pointer() {

return (char\*)bits;

}

//取某一位的值

bool get\_bit(u16 pos) {

pos -= start;

if (pos >= 4096)

l("-1 for uint!");

u64 t = (u64)0x01 << (pos % 64);

return bool(bits[pos / 64] & t);

}

//寻找空位: 从某一位(pos,含)向后寻找

//若有连续的n位为空, 则返回这n位的首位位置, 并置这n位为1

//若无连续n位, 则返回最大的连续空位数的相反数-1(避免0的二义性)

int find\_zeros(u16 pos, u16 n) {

if (n == 0)

return pos;

pos -= start;

if (pos == 65535)

l("-1 for uint!");

int curr = 0, max = 0;

for (u16 i = pos; i < BlockSize \* 8; i++)

{

if (this->get\_bit(i+start)) {

if (max < curr)

max = curr;

curr = 0;

}

else {

curr++;

if (curr == n) {

for (int j = i - n + 1; j <= i; j++) {

this->set\_bit(j + start);

}

return i - n + 1 + start;

}

}

}

curr = 0;

for (u16 i = 0; i < pos; i++)

{

if (this->get\_bit(i + start)) {

if (max < curr)

max = curr;

curr = 0;

}

else {

curr++;

if (curr == n) {

for (int j = i - n + 1; j <= i; j++) {

this->set\_bit(j + start);

}

return i - n + 1 + start;

}

}

}

return (curr > max) ? -curr - 1 : -max - 1;

}

};

//组描述符, 保存文件系统的基础信息

//sizeof(Group\_Descriptor) = 512 Bytes

struct Group\_Descriptor

{

char volume\_name[16] = { 0 };//卷名

u16 block\_bitmap = 1;//数据块位图所在的磁盘块号

u16 inode\_bitmap = 2;//索引结点位图的磁盘块号

u16 inode\_table = 3;//索引结点表的起始磁盘块号

//针对这几个值有一系列操作,如判零 自减等.封装它们很没意思,直接外部修改吧

u16 free\_blocks\_count = BlockSize \* 8;//空闲块的个数(指数据块)

u16 free\_inodes\_count = BlockSize \* 8;//空闲索引结点的个数

u16 used\_dirs\_count = 0;//目录的个数

char pad[4];//填充

char remain\_padding[480] = { 0 };//继续填充至512字节

Group\_Descriptor()

{

for (int i = 0; i < 4; i++)

{

pad[i] = (char)0xff;

}

}

};

//用于间接寻址的数据块内容

struct IndexBlock

{

u16 index[BlockSize / 2] = { 0 };

};

//索引节点inode, 保存文件信息及位置

//sizeof(Inode) = 64 Bytes

struct Inode

{

u16 i\_mode = 0;//高8位是文件类型, 低八位是访问权限

u16 i\_blocks = 0;//文件占用的数据块个数

u32 i\_size = 0;//文件大小, 单位Byte

u64 i\_atime = 0, i\_ctime = 0, i\_mtime = 0, i\_dtime = 0;//a访问, c创建, m修改, d删除时间

u16 i\_block[8] = { 0 };//文件所占用数据块的索引表(存储数据块号)

char i\_pad[8];//填充至64字节

//perm=permission,没词用了

Inode(char type = 1, char perm = 7)

{

i\_mode = (type << 8) + perm;

i\_atime = i\_ctime = i\_mtime = time(NULL);

for (int i = 0; i < 8; i++)

{

i\_pad[i] = (char)0xff;

}

}

//修改访问权限

void set\_access(char p) {

i\_mode = (i\_mode & (u16)0xff00) + p;

}

char get\_access() {

return i\_mode & (u16)0x00ff;

}

void set\_type(char type) {

i\_mode = (i\_mode & (u16)0x00ff) + (type << 8);

}

char get\_type() {

return (i\_mode & (u16)0xff00) >> 8;

}

//读取时

void access() {

i\_atime = time(NULL);

}

//修改时,size为文件新大小

void modify(u32 size) {

i\_mtime = time(NULL);

i\_size = size;

}

//删除时

void del() {

i\_dtime = time(NULL);

i\_mode = 0;

i\_blocks = 0;

i\_size = 0;

i\_atime = i\_ctime = i\_mtime = 0;

for (int i = 0; i < 8; i++) {

i\_block[i] = 0;

}

}

void print() {

u16 type = (i\_mode >> 8) & 0x00ff;

char rwx[4] = "---", d = '-', buff[26];

if (type == 2)

d = 'd';

rwx[2] = (i\_mode & (0x0001 << 0)) > 0 ? 'x' : '-';

rwx[1] = (i\_mode & (0x0001 << 1)) > 0 ? 'w' : '-';

rwx[0] = (i\_mode & (0x0001 << 2)) > 0 ? 'r' : '-';

ctime\_s(buff, sizeof buff, (time\_t\*)&(i\_mtime));

buff[24] = 0;

std::cout << d << rwx << rwx << rwx << ' ' << i\_size << '\t' << buff;

}

};

//目录文件内容中的一项

struct DirEntry

{

u16 inode = 0;//此目录项对应文件的inode号, 非0则为正确的目录项

u16 rec\_len = 0;//此目录项的长度(不定长,至少7,可能大于实际长度,是为了在有空隙的情况下找到下一个目录项)

char name\_len = 0;//文件名长度(不包括\0)

char file\_type = 0;//文件类型

char name[256] = { 0 };//文件名

DirEntry(u16 nodei, std::string nm, char type)

:inode(nodei), file\_type(type) {

u32 strl = (nm.size() > sizeof(name) - 1) ? sizeof(name) - 1 : nm.size();

nm.copy(name, strl);

rec\_len = 7 + strl;

name\_len = strl;

}

DirEntry() {}

//判断一段二进制字节流是否为正确存在的目录项

//若此目录项inode为0, 返回false, rec\_len不变

//若此目录项inode为1, 返回true, rec\_len置为此目录项的rec\_len

std::pair<bool, u16> is\_alive(char\* head) {

u16\* test = (u16\*)head;

if (\*test == 0)

return { false, \*(++test) };

return { true, \*(++test) };

}

//从二进制数据中建立结构, 返回可能存在的下一项的首地址

//使用时需保证数据的正确性

char\* init(char\* head) {

u16\* pt16 = (u16\*)head;

inode = \*(pt16++);

rec\_len = \*pt16;

char\* pt8 = head + 4;

name\_len = \*(pt8++);

file\_type = \*(pt8++);

u16 i = 0;

for (; i < name\_len; i++) {

name[i] = \*(pt8++);

}

name[i] = 0;

return head + rec\_len;

}

//获取从指针位置, 到下一个已用目录项之间的空隙的长度

//要求指针指向一个已被删除的目录项首地址, 且其后仍有正确存在的目录项

//同样提供最大长度,

//u64 gap\_len(char\* bits) {

// u64 sum\_len = 0;

// u16 this\_len = 0;

// while (!is\_alive(bits)) {

// this\_len = \*(((u16\*)bits) + 1);

// sum\_len += this\_len;

// bits += this\_len;

// }

// return sum\_len;

//}

//从指针head开始, 找到下一个目录项的指针

//从head指向的字节算起, 若在max\_len内无目录项, 则返回nullptr

//max\_len=buf\_tail-head=可用长度

//下一项至少要有inode和rec\_len存在,便于指针移去之后的判断

char\* next\_head(char\* head, u64 max\_len) {

u16\* p16 = (u16\*)head;

char\* p8 = head;

u64 len1 = \*(p16 + 1);

if (len1 > max\_len - 4)

return nullptr;

p8 += len1;//p8->next\_head

return p8;

}

};

### 3.8.2附件2 Readme

**README.md**

# MyExt2类文件系统

## 环境

由于是使用一个现有操作系统上的文件来代替硬盘进行硬件模拟, 所以选择在Visual Studio下, 使用C++编写. 但在编写过程中尽量注意代码的跨平台性, 做到能够在Linux下运行.

## 理解需求

"硬盘"中包括两部分:

1. "控制部分"包括:

> 兼并了超级块的组描述符, 用于记录整个文件系统的一些信息

数据块位图, 用二进制位表示哪些数据块用过了

索引节点位图, 用二进制位表示在索引节点表中, 哪些位置被索引节点占用了

索引节点表, 用于存放一定数量的索引节点(inode)

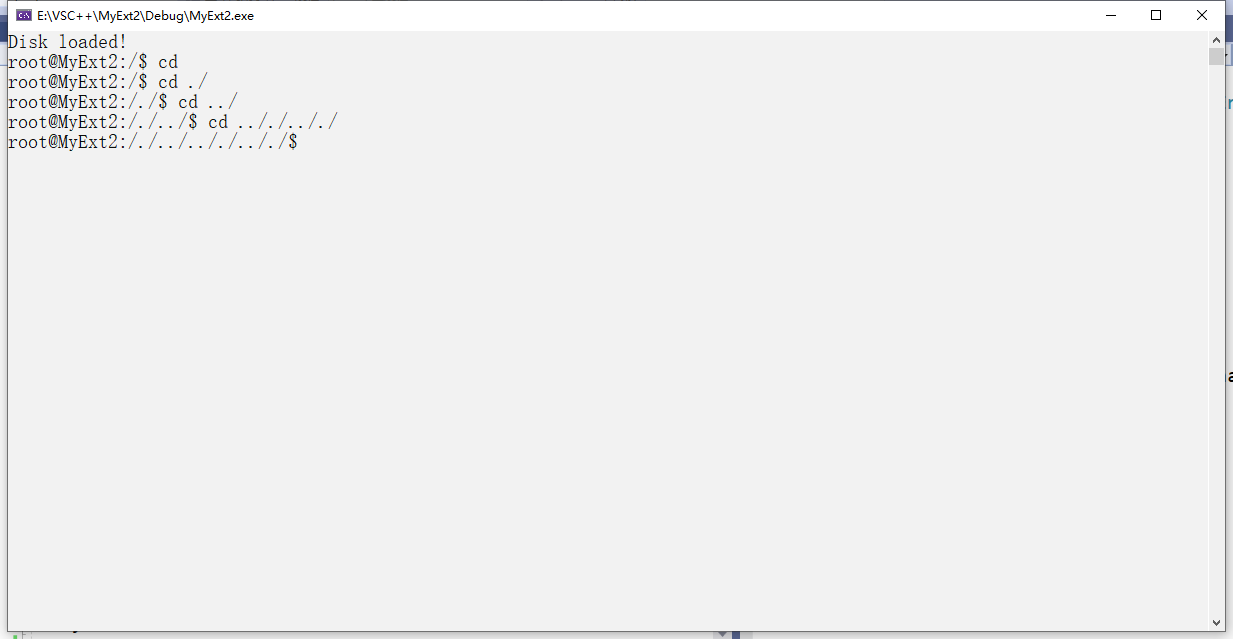
索引节点(inode), 一个inode包含了一个文件的属性信息和在磁盘上的位置. 其中采用直接索引, 一级索引, 二级索引想结合的方式来记录位置.

2. "数据部分"包括一定数量的数据块. 数据块是存储文件内容的地方, 而文件又分为数据文件和目录文件:

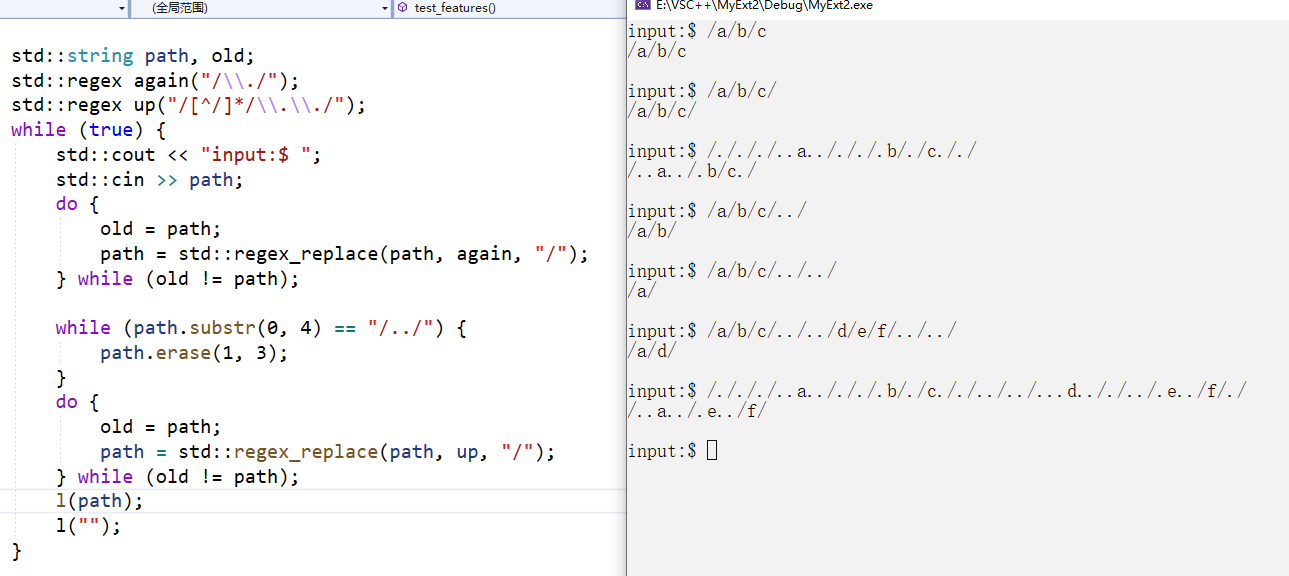
1. 数据文件所占据的数据块全部用来存储文件内容

2. 目录文件的内容由若干个目录项构成. 目录项的数量等于此目录下的文件/文件夹数量+2(还有./和../两个默认文件夹). 每个目录项的内容由dir\_entry结构给出.

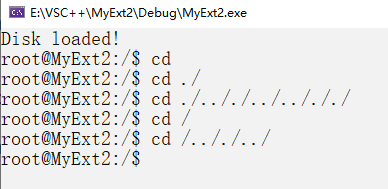
**README.docx**



问题:路径显示不正确



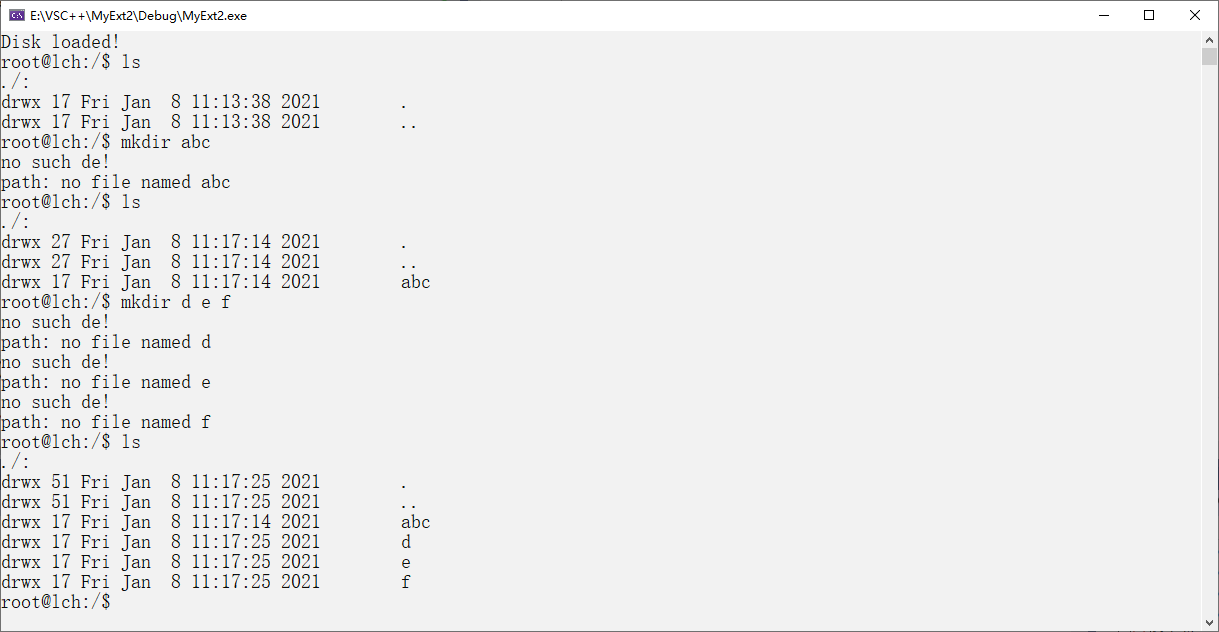
关于路径显示的bug修复: 使用正则表达式

效果: 

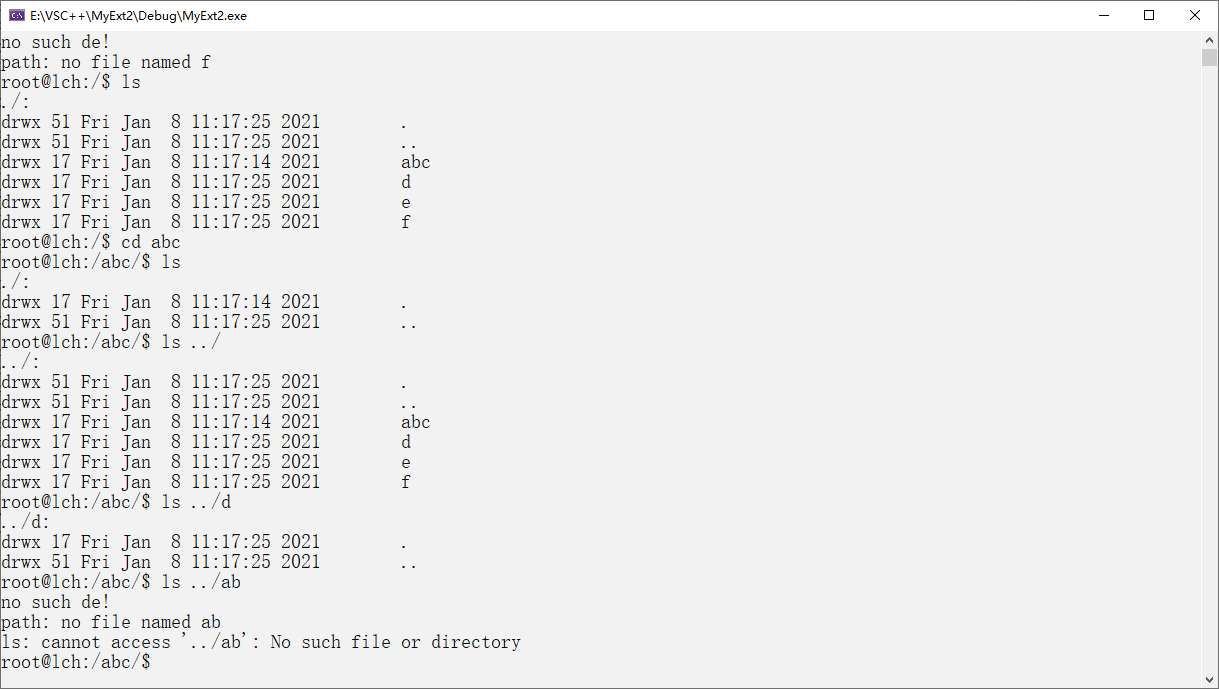
问题: mkdir无效, ls不显示新建的文件夹

解决:路径解析函数 目录项查找函数 均出现问题,已debug; 还有一个原因是添加新的目录项成功, 但是目录文件的长度没有更改, 导致下次读取时还是只能读取到之前的. ..两个目录项(共17字节, 后来添加了文件长度修改的代码后, 能够正常ls显示出来)

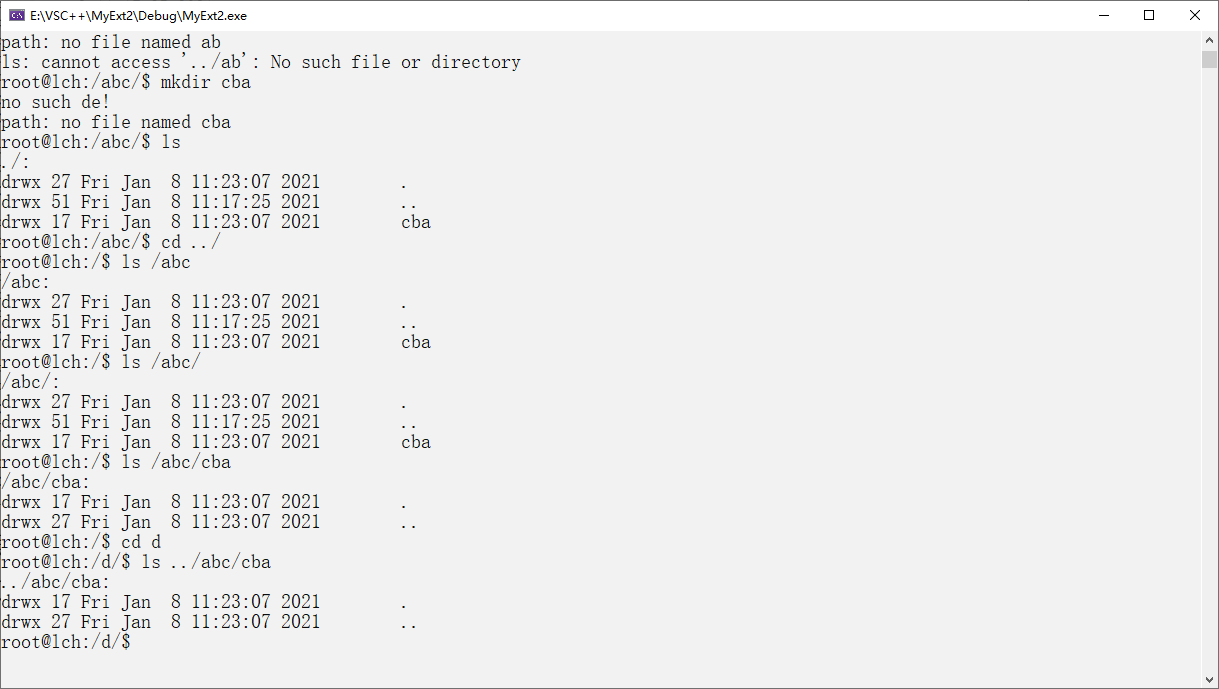
效果:



ls与mkdir准确无误, 其中mkdir支持批量创建(在使用mkdir新建文件夹时, 需要检测是否已存在此文件夹, 所以复用了路径解析函数, 在目前debug的条件下, 它会在找不到文件时输出两句警示, 这是正常现象, 可以无视)

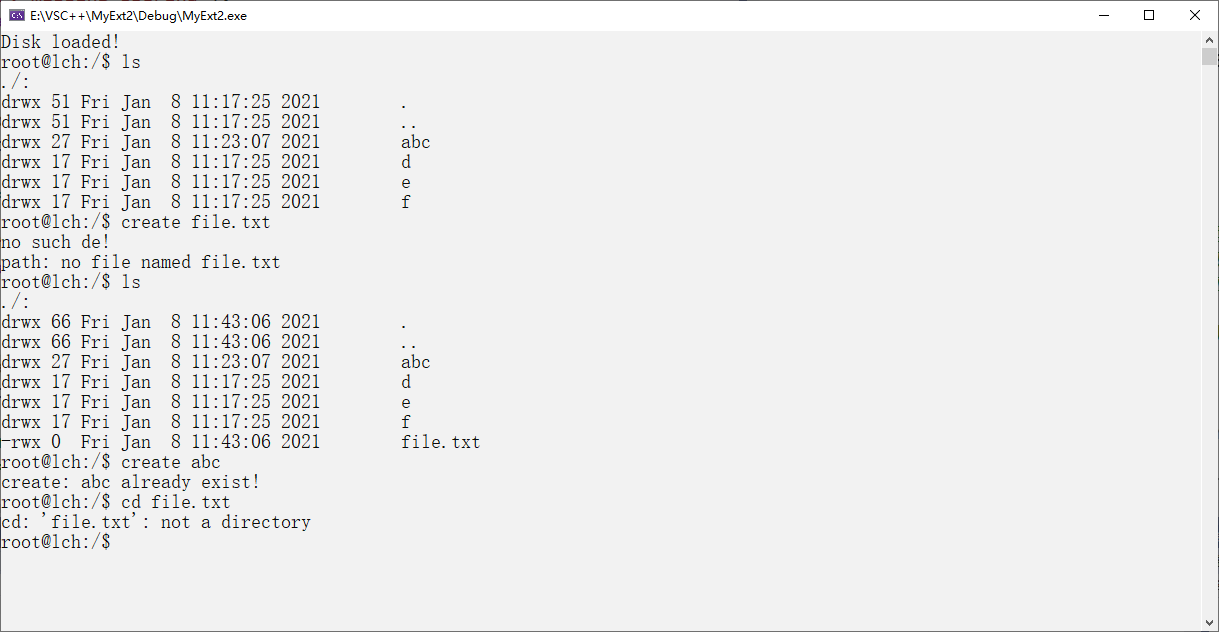


cd无误, ls在非根目录下同样无误; ls允许输入路径, 列出其它目录下的内容, 这有赖于一个统一的路径处理函数; 通过ls一个不存在的路径, 可以看到路径处理函数也具有错误检测能力.



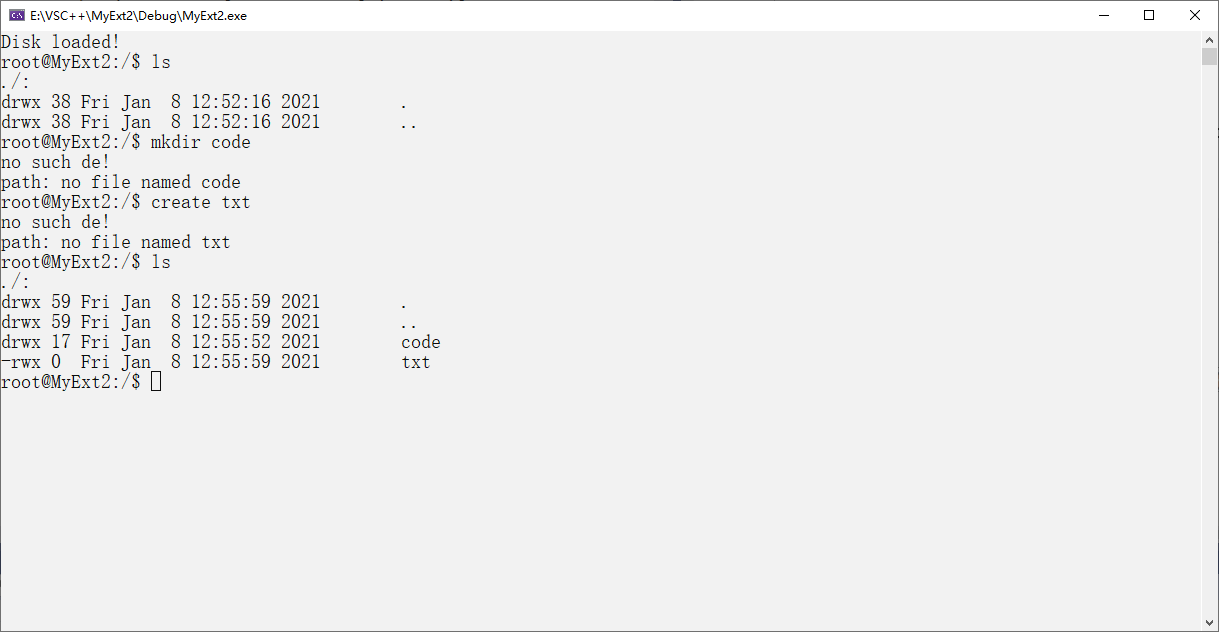
在上图前,根目录/ 下有4个创建的文件夹:/abc /d /e /f . 现在在/abc下创建新文件夹/abc cba, 由ls命令可知创建成功. 此时用cd 分别回退到根目录下以及/d目录中, 通过相对路径和绝对路径来ls二级目录/abc/cba, 结果均成功, 证明此文件系统是树形文件系统, 且具备相应的处理能力.

create展示:

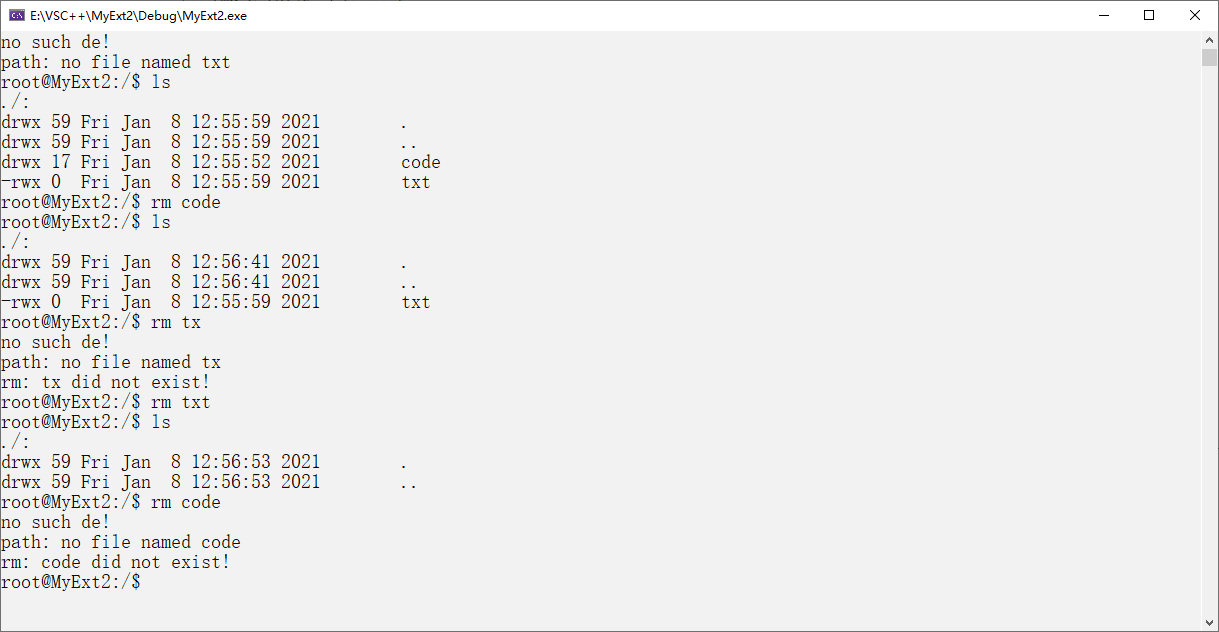


新建文件file.txt, 前后的ls可证明成功, 且新建时不会新建同名文件, 且新建的是文件, cd能对跳转到非目录文件有正确的反馈(跳转失败且正确报错)

rm展示:



新建一个文件夹/code/ , 以及一个文件txt.

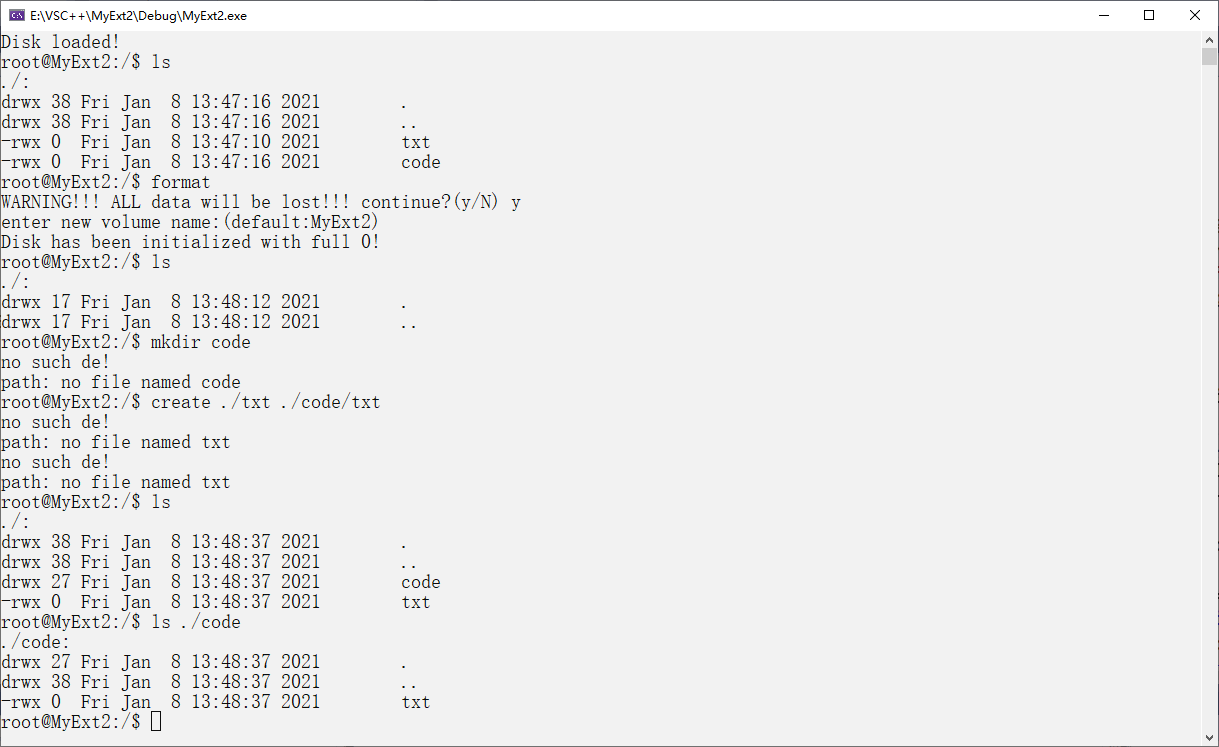


对code/和txt分别进行删除, ls可得删除成功.

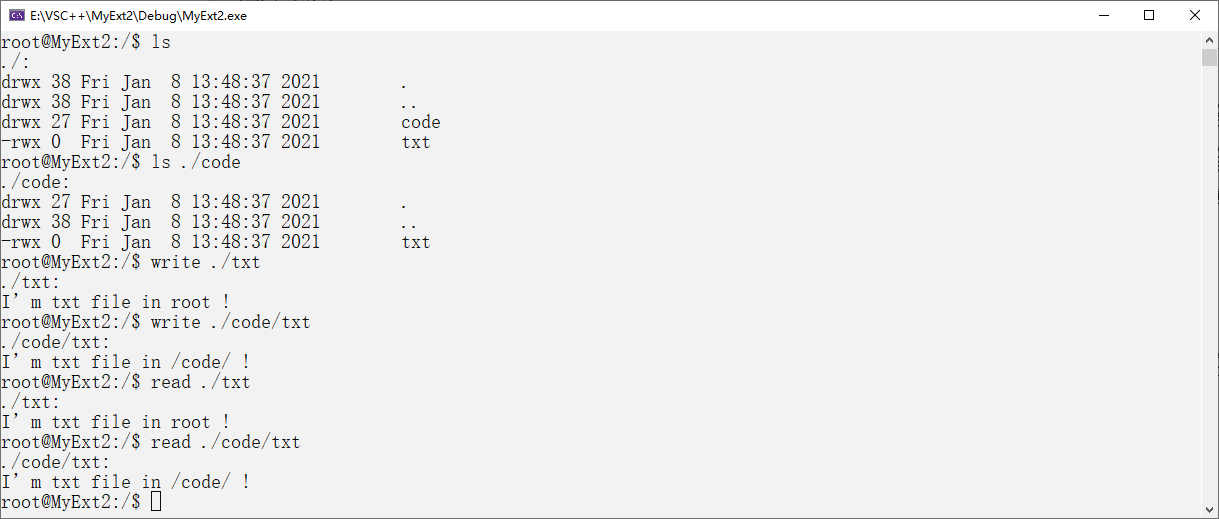
编写测试rm时也遇到很多问题:目录文件的next函数不能正确判断是否到达文件终点, 在死循环里出不来; 删除目录项的函数没有将缓冲区的脏标志dirty置为true, 导致修改完的缓冲区没有被写入磁盘, 进而删除不生效, 等等。以上问题均借助vs的断点、监视等功能完成debug。

read和write展示：

在本次展示中，将在根目录/下新建文件txt，并写入“I’m txt file in root !”，然后再在文件夹/code/下新建文件txt，并写入“I’m txt file in /code/ !”，随后变幻不同位置进行读取，验证效果。



在上图中，由于之前文件系统已经拥有文件，故进行格式化；格式化后新建code以及位于两个不同文件夹下的txt文件。

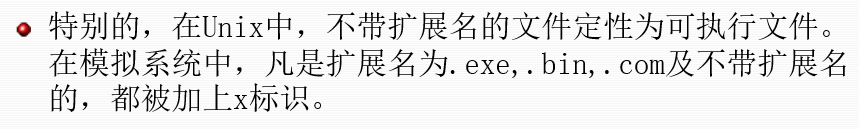


在上图中，按计划向两个文件写入不同字符，并读取。其结果与预期一致。

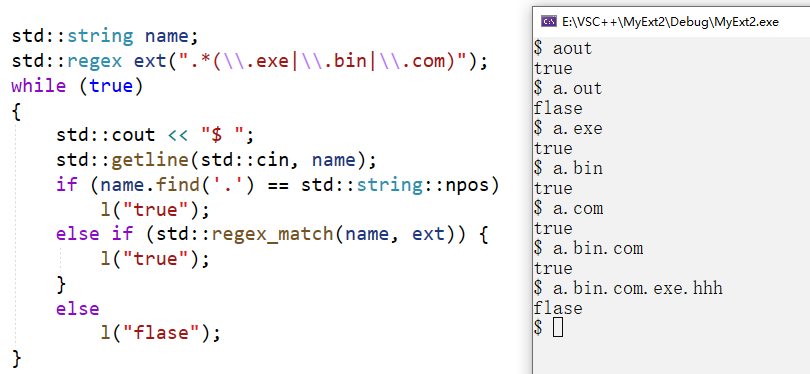
与此同时，它也证明了这是一个树形文件系统：它允许两个不同文件夹下的文件同名，且内容也可不同（是两个独立的文件。）

rwx权限管理:

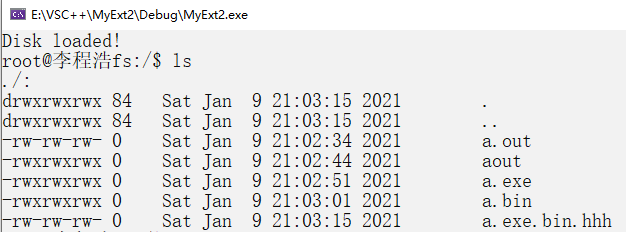
首先, ppt上有这样一条要求:



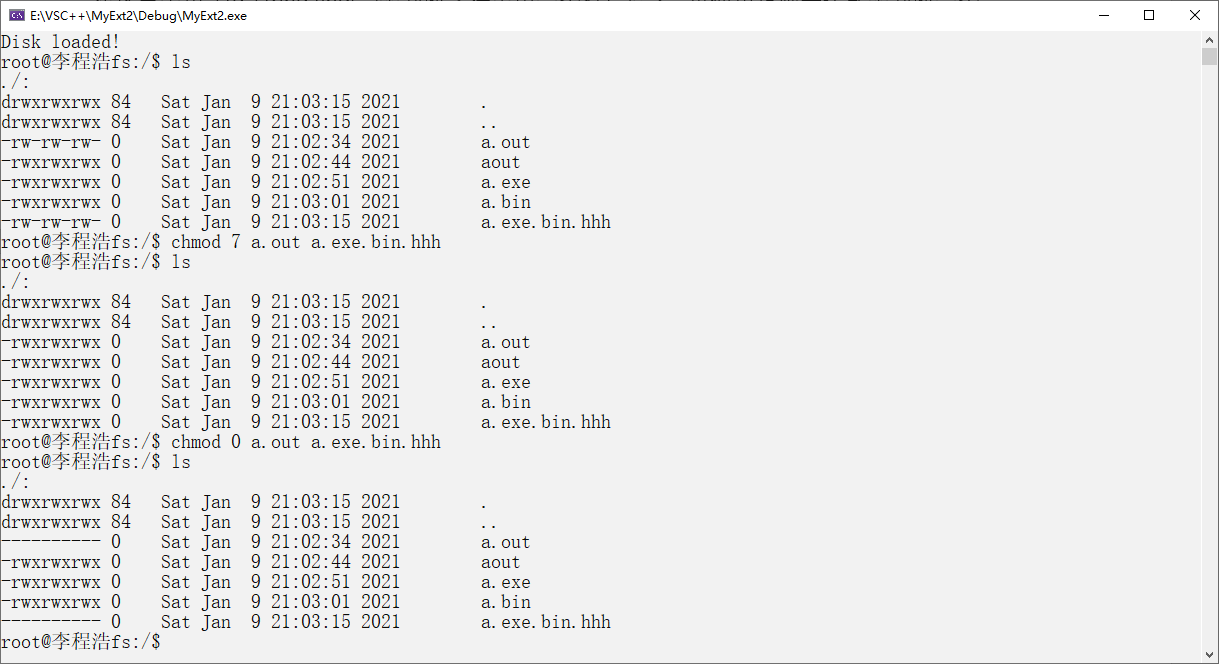
在这里采用std::string.find(‘.’)来判断字符串中是否有后缀名, 再使用正则表达式来判断是否是由以上三种后缀名结尾, 其测试效果如下图:



测试成功, 将其添加进系统内, 此时创建文件时自动生成的文件权限如下图:

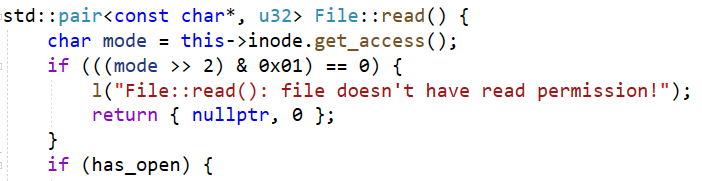


随后加入chmod命令, 它先接受一个数字, 其3位二进制位分别代表将要修改的文件的r(读取)w(写入)x(执行)权限, 如7就代表rwx=111,即此文件可读可写可执行;而6则代表rwx=110,即此文件可读可写, 但不可执行.随后的若干个参数, 是要修改权限的若干个文件的路径, 以此达到批量修改的目的, 方便使用.使用情况如下图:

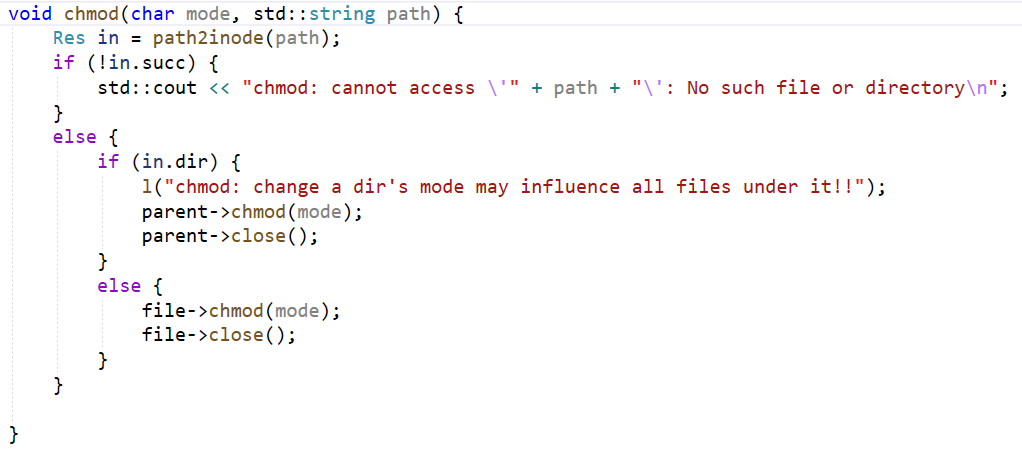


在上图中, 先用chmod 7，将没有执行权限a.out、a.exe.bin.hhh两个文件赋予所有权限，然后用ls证实；再用chmod 0，将其所有权限全部剥夺。测试结果是准确无误的。

之前在进行文件读写时一直没有加入权限对其的控制，现在加入。由于设计架构时采用了多层次的面向对象结构，此时修改起来非常轻松：只需要在File类中的read write等几个方法中加入几行判断语句即可，Dir类由于继承自File类，无需进行任何修改。如下图：



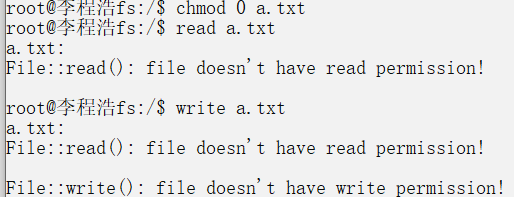
需要注意的是目录文件如果没有写权限，则无法向其添加子文件；如果没有读权限，则不能访问到它之下的全部子文件和子文件夹，因此目录文件的默认权限是rwx=111，并且在修改其权限时也会发出警告，如下图：



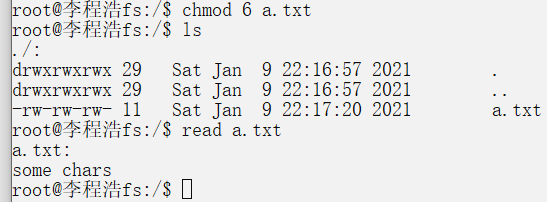
对权限管理进行综合展示：



如上图，创建一个文件a.txt，根据其后缀名.txt判断其不是可执行文件，故创建时默认为rwx=110权限。随后正常读写其内容。



如上图，将其所有权限置0，则read 与write命令都失效。



上图设置其读写权限，则a.txt又可以正常读写。综上，权限管理已完成既定目标。