0.作者

冯亚林 191850036

1.项目结构

```
a.img 软盘镜像；因为硬编码，改名就没法正确读取了；

源码文件：
1.main.cpp主程序
2.structure.h 头文件，声明了4种数据结构，Sector、FAT、Entry、Directory
3.structure.cpp  实现数据结构的方法
4.constants.h 头文件，一些常量
5.my_print.asm nasm写的输出函数

可执行文件：
test linux环境下生成的可执行文件

中间代码：
my_print.o nasm汇编后的中间代码
```

2.实验目标

知识目标：熟悉掌握FAT12文件系统、gcc+nasm联合编译，了解实模式与保护模式的基本内容。

代码目标：用C/C++和nasm编写一个FAT12镜像查看工具，读取一个.img格式的文件并响应用户输入。

具体功能：读取FAT12文件、提示用户输入、实现ls功能、实现ls -l功能、实现cat功能；

3.制作FAT12镜像

在虚拟机里制作FAT12镜像

> 创建镜像：

制作FAT12镜像

- Linux
- 1.在当前目录(.)下创建一个新的软盘镜像a.img

  mkfs.fat -C a.img 1440
- 2. 在当前目录下创建一个新目录(./mount)作为挂载点

  mkdir mount
- 3. 将镜像./a.img挂载到./mount下

  sudo mount a.img mount

注意事项：

注意事项

- 挂载后，就可以通过操作./mount文件夹，来向a.img加入和查看文件。可使用系统自带的资源管理器类似 GUI工具，或者使用命令行操作。
- 在操作挂载后的img镜像时，若使用命令行进行操作，需要使用root权限运行所有操作（例如mkdir, touch等）

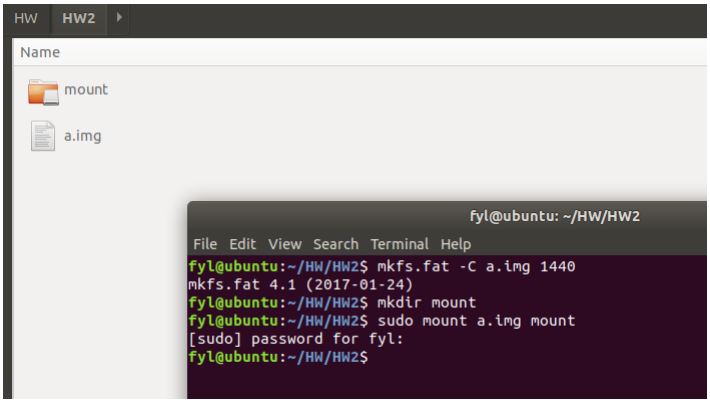
其实还有一个注意事项：

linux中的挂载硬盘，除了系统盘，每次开机为什么都要重新挂载一次?

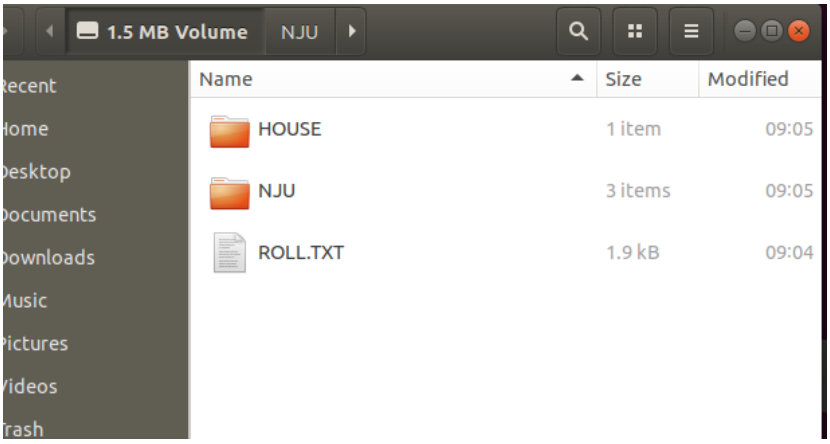
答：除了系统盘，其他硬盘可以在/etc/fstab中定义是否开机时自动挂载，没有在/etc/fstab定义的挂载硬盘都要开机时重新挂载、

实验结果：

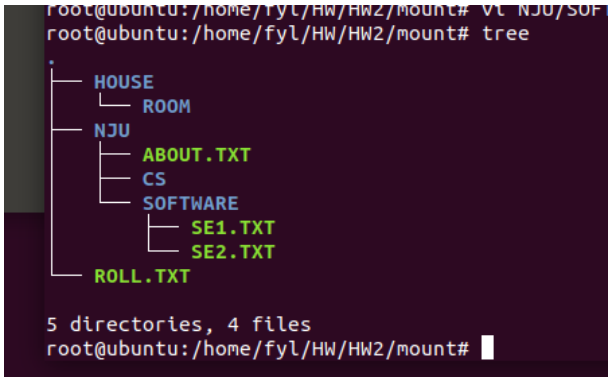
在ubuntu18.04虚拟机里：可以看到目录mount有不同的图标说明挂载成功了；



可以在里面添加一些目录和文件，以便后面测试；注意要使用带root的权限执行操作



顺便用sudo apt install tree安装了个tree，用tree看看现在的挂载目录：



自问：img文件是什么？将img文件挂载到目录上什么意思？挂载后有什么用？

答：首先，img格式是一种文件压缩格式，主要用于创建软盘的镜像；换句话说，img文件就是软盘镜像。

那什么是镜像？镜像就是将一个软件及其运行环境打包后形成的文件包，可以还原为对应的软件。

比如许多操作系统的镜像 iso 文件，创建虚拟机时需要使用。

挂载又是什么意思？

最容易记忆的说法是，挂载是将**存储设备与文件系统联系起来。**这样我们就能通过文件系统访问存储设备了。

这里创建了软盘镜像，相当于是有了一个“虚拟软盘”的存储设备，又有了目录mount，将两个挂载起来，就是将二者联系起来，可以通过将文件移入目录来相当于将文件写入软盘。

这些都是**早就学过**的知识，要时时复习。

4.阅读要求，创建项目

仔细阅读要求——确定大概流程——创建项目

> 要求1.

用C/C++和nasm编写一个FAT12镜像查看工具，读取一个.img格式的文件并响应用户输入。

功能列表

1. 运行程序后，读取FAT12镜像文件，并提示用户输入指令
2. 用户输入`ls 路径`，输出根目录及其子目录的文件和目录列表。
    1. 首先输出路径名，加一个冒号:，换行，再输出文件和目录列表；
    2. 使用红色(`\033[31m`)颜色输出目录的文件名，不添加特殊颜色输出文件的文件名。
    3. 当用户不添加任何选项执行`ls`命令时，每个文件/目录项之前用**两个空格**隔开
    4. 当用户添加`-l`为参数时，
        1. 在路径名后，冒号前，另输出此目录下**直接子目录**和**直接子文件**的数目，两个数字之间用**空格**连接。此两个数字**不添加特殊颜色**
        2. 每个文件/目录占据**一行**，在输出文件/目录名后，**空一格**，之后：
            1. 若项为目录，输出此目录下**直接子目录**和**直接子文件**的数目，两个数字之间用**空格**连接。此两个数字**不添加特殊颜色**
                1. 不输出`.`和`..`目录的子目录、子文件数目
            2. 若项为文件，输出文件的大小
        3. 对于`-l`参数用户可以在命令**任何位置**、**设置任意多次`-l`参数**，但**只能设置一次文件名**
        4. 直接子目录不计算`.`和`..`
    5. 当用户给出不支持的命令参数时，**报错**
    6. 当用户不设定路径时，默认路径为**镜像文件根目录**
3. 用户输入`cat 文件名`，输出路径对应文件的内容， 若路径不存在或不是一个普通文件则给出提示，提示内容不严格限定，但必须体现出错误所在。
4. 用户输入`exit`， 退出程序。

要求2.

常见问题

- 以下命令均等价于`ls /NJU -l`:
    - `ls -l /NJU`
    - `ls -ll /NJU`
    - `ls -l /NJU -ll`
- 以下命令为出错:
    - `ls -L /NJU`
    - `ls -al /NJU`
    - `ls /NJU -lL`
    - `ls /NJU /NJU`

- **直接子文件/目录**即不计算自己的子目录的子文件/目录的数量
- `ls`中列出多个目录时，其顺序不做规定，每个目录的子文件/目录列表中的各个项的顺序不做规定
- 程序由两个源文件构成，`main.c(cpp)` 和 `my_print.asm`， 其中`main.c(cpp)` 是主程序，可以使用C/C++库，但是输出不能使用库函数，要求在`my_print.asm`中使用汇编编写函数用于输出。
- 要求使用`Makefile`编译链接项目。
    - Windows 平台可使用CMake或其它构建工具。
- .img文件的名称可以直接在代码中指定（俗称硬编码）。
- 要求根据FAT12文件系统格式**直接读取.img中的二进制内容**，不允许挂载镜像。
- FAT12中名称均为大写，**只需要输入为英文大写/数字的指定路径/文件名**（即小写的文件/目录名为不存在），不考虑中文字符、不需要支持长文件名。
- 输入指令以回车符号结束，要求可以多次不断输入。
- 程序应该对用户错误的输入做出恰当的提示，**指出错误所在**，不能崩溃

请提交运行截图、 源文件和`Makefile`文件。

评分标准

基本得分：实现基本功能 (`ls`, `ls -l`, `cat`)

附加得分：`cat`命令支持输出超过 512 字节的文件

注意这一段：

> 程序由两个源文件构成，main.c(cpp) 和 my_print.asm， **其中main.c(cpp) 是主程序，可以使用C/C++库**，但是**输出不能使用库函数，要求在my_print.asm中使用汇编编写函数用于输出。**

项目分析

项目源码文件：main.cpp, my_print.asm

main程序任务：

·读取FAT12镜像文件（可以硬编码），并提示用户输入指令

·接收用户输入命令，根据命令具体类型做switch-case； 命令类型： ls 、ls -l、cat、错误命令

·用户输入exit，退出程序

·输出时要调用my_print.asm里定义的函数，这就涉及到gcc与nasm联合编译

my_print.asm:

·职责是定义一个输出函数；可以复用第一次写的IO汇编代码。

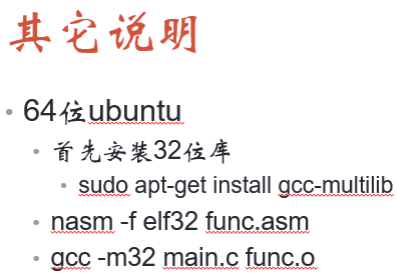
5.复用nasm输出函数

> 直接复用第一次写的nasm输出函数，略加修改（删除了64位系统下没有的push a和pop a，修改了函数名）

my_print.asm

```
;---------------------------封装函数区-----------------------------------;
;第一次实验中已经写好了 read 和 print 方法，可以直接复用
;这次考察的要点不是 my_print.asm的编写，而是怎样在 C/Cpp文件里调用 asm里的函数，怎样将二者正确地编译和链接
global my_print;
        section .text
;职责:   向屏幕输出 print
;arg:    (rax,rdi), rsi, rdx;前两个参数在函数内设置
;               rsi:要输出的字符串起始地址
;               rdx: 字符串长度(单位:字节)
;有call就有ret
;ret:    rax中存储输出的字节数s
my_print:
        mov rax,1                ; sys_write的系统调s用编号为1
        mov rdi,1                ; 文件句柄1对应输出流stdout
        syscall                  ; 系统调用(64bit下可用,对比32位下是int 80h)
        ret                      ; 返回
```

> 如果要用64位模拟32位，需要注意：~~（但我选择不模拟）~~



6.测试g++和nasm编译

C++读取文件可以用文件输入流fstream; 提示输入要用到nasm的函数；

main.cpp和my_print.asm编译链接很容易出错，先写一个简单的main.cpp测试一下；

```cpp
//三个参数分别是寄存器rsi=1 rdi(输出字符串的首地址) rdx(输出字节数)
size_t my_print(int fildes, const void* buf, size_t nbytes);
int main(void) {

    //打开软盘 a.img
    fstream fs;
    fs.open("a.img", ios::in);

    //提示
    string tips = "please input some commands: \n ls/ls -l/cat [filename]/exit\n";
    my_print(1, tips.c_str(), tips.size());

    fs.close();
    return 0;
}
```

然后在终端输入：

```
nasm -f elf64 my_print.asm -o my_print.o //生成my_print的中间文件（二进制文件）

gcc -m64 main.cpp my_print.o -o test -no-pie  //用gcc编译main.cpp，并和my_print.o链接，生成可执行程序test，-m64是64位系统下使用
```

结果错误

发现了可能解决问题的博客：

https://blog.csdn.net/HermitSun/article/details/102905085

https://blog.csdn.net/wtzhu_13/article/details/105654408

重新做：

1.用sudo apt install g++-multilb安装g++的库

2.用g++编译

3.修改了main.cpp里如何引用nasm的函数:

> **一定要注意extern “C” { }的写法，里面是nasm写的my_print**

```cpp
extern "C" {
    size_t my_print(int fildes, const void* buf, size_t nbytes);
}
int main(void) {

    //打开软盘 a.img
    fstream fs;
    fs.open("a.img", ios::in);

    //提示
    string tips = "please input some commands: \n ls/ls -l/cat [filename]/exit\n";
    my_print(1, tips.c_str(), tips.size());

    fs.close();
    return 0;
```

重新操作，编译和链接成功，并且运行程序：

```
fyl@ubuntu: ~/HW/HW2
File  Edit  View  Search  Terminal  Help
fyl@ubuntu:~/HW/HW2$ nasm -f elf64 my_print.asm -o my_print.o
fyl@ubuntu:~/HW/HW2$ g++ -m64 main.cpp my_print.o -o test -no-pie
fyl@ubuntu:~/HW/HW2$ ./test
please input some commands:
 ls/ls -l/cat [filename]/exit
fyl@ubuntu:~/HW/HW2$
```

7.自动化编译：学习配置makefile

配置makefile：

```
Open   makefile  ~/HW/HW2   Save
test:main.cpp my_print.o structures.h
        @echo "===== 开始生成可执行文件 test ====="
        g++ -m64 main.cpp my_print.o -o test -no-pie
        @echo "===== test 文件已生成 ====="
my_print.o:my_print.asm
        @echo "===== 开始编译 my_print.asm ====="
        nasm -f elf64 my_print.asm -o my_print.o
        @echo "===== 编译结束 ====="
```

配置好以后，在目录下执行make命令：

```
fyl@ubuntu: ~/HW/HW2
File  Edit  View  Search  Terminal  Help
fyl@ubuntu:~/HW/HW2$ make
===== 开始生成可执行文件 test =====
g++ -m64 main.cpp my_print.o -o test -no-pie
===== test 文件已生成 =====
fyl@ubuntu:~/HW/HW2$
```

以下是没用的个人笔记：

什么是 make？(复习)

第一次实验报告里 我写的：

GNU make是一个项目包的高效管理和构建工具，可以**方便地编译、链接多个源代码文件**，**自动**决定哪些源文件需要重新编译（即所谓的高效构建项目）；它也可以用来卸载项目包;

什么是 makefile?

Makefile 文件描述了 Linux 系统下 C/C++ 工程的编译规则，它用来自动化编译 C/C++ 项目。一旦写编写好 Makefile 文件，只需要一个 make 命令，整个工程就开始自动编译，不再需要手动执行 GCC 命令。

用我自己的话来说：makefile就是**make指令的操作手册**，它写明了需要编译哪些文件、怎么样链接、最后生成什么样的可执行文件，写好makefile以后，make指令就可以“自动化编译”。

Makefile写法

> **1. 语法规则**
>
> ```
> 目标...: 依赖...
>     命令1
>     命令2
>     ...
> ```
>
> **2. 目标**
>
> 目标即要生成的文件。如果目标文件的更新时间晚于依赖文件的更新时间，则说明依赖文件没有改动，目标文件不需要重新编译。否则重新编译并更新目标。
>
> **3. 依赖**
>
> 即目标文件由哪些文件生成。如果依赖条件中存在不存在的依赖条件，则会寻找其它规则是否可以产生依赖条件。
>
> 例如：规则一是生成目标 hello.out 需要使用到依赖条件 hello.o，但是 hello.o 不存在。则 Makefile 会寻找到一个生成 hello.o 的规则二并执行。
>
> **4. 命令**
>
> 即通过执行该命令，由依赖文件生成目标文件。
>
> 注意每条命令前必须有且仅有一个 **tab** 保持缩进，这是语法要求。
>
> **5. ALL**
>
> Makefile 文件默认只生成第一个目标文件即完成编译，但是我们可以通过 “ALL” 指定需要生成的目标文件。

一些参考：

https://zhuanlan.zhihu.com/p/47390641

http://c.biancheng.net/makefile/

https://blog.csdn.net/afei__/article/details/82696682

https://blog.csdn.net/weixin_33769125/article/details/93958436

*8.完善程序

经过以上的步骤，完成了：待读取镜像、nasm的输出函数、自动编译环境；

最后的任务非常简单了，那就是完善

8.1 处理的框架：输入—while(true)

> 首先：读取和处理输入

```
/*
* 遍历镜像文件，设置数据区的起始扇区、FAT12表、根目录区;
* 注意!
* linux上这里是fs.read(buff,512) 没有问题
* 在windows的VS2019上尝试读取起始扇区时存在对齐问题，调试后修改成fs.read(buff,510)
*   在windows上读入510个字节才是正确的
*
* fst_data_sec:数据区的起始扇区号,在设置过程中，遍历引导扇区;
* set_FAT:遍历FAT1，并将数据保存;
* fs.ignore(FAT_SECTORS * SECTOR_SIZE):丢弃FAT2;
* set_directory_area:遍历根目录区并设置好目录树;
* buff += … 在遍历以后，通过修改指针丢弃根目录区,最后剩下数据区
*/
    fs.read(buff, SECTOR_SIZE);//TODO! linux上这里fs.read(buff,512) 没有问题
    boot.set(buff,SECTOR_SIZE);

    fst_data_sec = 0 + 1 + FAT_SECTORS * 2 + (boot.get_BPB_RootEntCnt() / 16);

    fat.set_FAT(fs);
    fs.ignore(FAT_SECTORS * SECTOR_SIZE);

    fs.read(buff, SECTOR_SIZE * (2880 - 1 - FAT_SECTORS * 2));
    tree.set_root_dir(buff, fst_data_sec,fat);
    buff += boot.get_BPB_RootEntCnt() * ENTRY_SIZE;
```

运行程序以后可以用 while(true){ 嵌套 if + 终止条件} 搭起一个程序框架；

其中iss是字符串输入流，便于后续利用操作符>>，获取命令项(ls/cat/exit等)

```
    //二、循环处理输入,程序核心部分

    while (true){

        /*
        * 提示输入命令,获取每行输入input,设置字符流iss
        */
        my_print(1, tips.c_str(), tips.size());
        getline(cin, input);
        iss.str(input);

        //判断是否只输入了空字符(\t \n space)
        if (!(iss >> cmd)) {
            //"No valid input\n"
            my_print(1, RES_STRINGS[4].c_str(), RES_STRINGS[4].size());
        }

        //exit 提示+出循环
        else if (cmd == "exit"){
            //"Program ends\n",
            my_print(1,RES_STRINGS[0].c_str(), RES_STRINGS[0].size());
            break;
        }

        //ls
        else if (cmd == "ls") {
            /*
```

8.2 获取命令参数+测试

ls可能有0个——任意个参数，其中最多一个路径，其他可以是-l -ll -lll，也就是可以是任意个-l+

cat后面必须有一个filename

1.代码1

```
//根据命令处理:
//exit: 没有任何额外参数，最简单，提示+出循环即可
else if (cmd.compare("exit")==0) {
    my_print(1,res_strs[0].c_str(), res_strs[0].size());
    break;
}
//ls
else if (cmd.compare("ls")==0) {
    /*
    * ————————————1.输出提示信息、初始化————————————
    * ls命令最多有2个有用的后续参数：1个是-l，1个是目录
    * has_dir:是否带有地址参数
    * has_l:是否带有-l 参数
    * dir:地址，默认为映像文件的根目录
    * wrong_param: 后续参数是否正确
    */
    my_print(1, res_strs[1].c_str(), res_strs[1].size());

    string tmp;
    bool has_dir = false;
    bool has_l = false;
    bool right_param = true;
    string dir = IMGPATH + "/";
```

2.代码2

```
    /*
    * ————————————2.处理 ls 附带参数————————————
    * 职责: 设置好变量 has_dir、has_l、dir
    * option:正则匹配 -l 的pattern，只能是 '-' 开头，后面跟至少1个l，也就是正则里的'+'
    * dir_path:正则匹配 目录 的pattern，以 "./"开头,或者以 "/"开头
    */
    regex option("-l+");
    regex dir_path("(/.*)|(\\./.*)");
    while (iss >> tmp) {
        //-l+ :这里有个 !has_l 意思是如果多次输入-l,后面的-l 可以直接无视，加快处理速度
        if (!has_l && regex_match(tmp, option)) {
            has_l = true;
            my_print(1, "ls receive -l option\n", 22);
        }

        //如果已经设置了目录参数，但又一次检查到了目录参数，直接报错
        else if (has_dir && regex_match(tmp, dir_path)) {
            my_print(1, "Error!ls receives more than 1 directory\n", 41);
            break;
        }

        //没有设置目录，检查到了目录参数
        else if (!has_dir&&regex_match(tmp, dir_path)) {
            has_dir = true;
            dir = "./" + tmp;
            //TODO: 测试dir
            my_print(1, "test dir", 9);
            my_print(1, "\n", 1);
            my_print(1, dir.c_str(), dir.size());
            my_print(1, "\n", 1);
        }

        //检查到了不支持的参数
        else {
            right_param = false;
            break;
        }
    }
```

3.代码2

```
            *----------3.输出信息----------
            *
            * ! right_param:不支持的后续参数
            */
            if (!right_param)
                my_print(1, "Unsuppoted parameter!\n", 23);


        }
        //cat
        else if (cmd.compare("cat")==0) {
            my_print(1, res_strs[2].c_str(), res_strs[2].size());
            string file;
            if (iss >> file) {
                my_print(1, "Successfully get filename:\n", 28);
                my_print(1, file.c_str(), file.size());
                my_print(1,"\n", 1);

            }
            else {
                my_print(1, "missing file path or file name!\n", 33);
            }
        }
        //不支持的命令
        else {
            my_print(1, res_strs[3].c_str(), res_strs[3].size());
        }

        /*
        * 处理完一次命令
        * 清空字符流iss;清空一行输入input;清空保存命令的变量cmd
        */

        iss.clear();
        input.clear();
        cmd.clear();
    }
```

4.成功接收到了参数：

```
                                                    fyl@ubuntu: ~/HW/HW2
File  Edit  View  Search  Terminal  Help
fyl@ubuntu:~/HW/HW2$ ./test
Please input any command below:
ls [-l] [path] OR cat filename OR exit
ls
test ls
Please input any command below:
ls [-l] [path] OR cat filename OR exit
ls -l
test ls
ls receive -l option
Please input any command below:
ls [-l] [path] OR cat filename OR exit
ls -l /nju
test ls
ls receive -l option
test dir
.//nju
Please input any command below:
ls [-l] [path] OR cat filename OR exit
cat roll.txt
test cat
Successfully get filename:
roll.txt
Please input any command below:
ls [-l] [path] OR cat filename OR exit
```

8.3 完善数据结构

在参考PPT和博客：https://zhuanlan.zhihu.com/p/121807427等后，对FAT12有了基本的认识。

设计了几个数据结构：扇区Sector、FAT12表、目录文件的条目Entry、目录树Directory

其中核心数据结构还是Entry

FAT：

```
//————————————————————————————————FAT 分割线————————————=——————————————————
/**
* FAT12
*       成员变量: vector<int> clusters;
*       提供方法: void set_FAT(fstream& fs); 根据输入流确定FAT表内容
*                 int& operator[](int); 操作符重载,获取索引处内容
*                 int get(int);获取索引处内容
*                 vector<int>& get_all_clus(const int&); 根据首簇号,获得某个 Entry 的所有簇号
*/
class FAT12 { ... };
```

Entry目录的条目：

```
/**
* 树结点——文件/目录 Entry
*       成员变量:
*                 string content; 32位的目录内容
*                 string name; 文件或目录名
*                 char attr; 类型:0x10目录 0x20文件
*                 int fst_clus ;首簇号
*                 vector<Entry*> childs;目录的子女结点
*                 Entry* parent; 父结点
*                 Entry* prev; 前一个兄弟结点
*                 Entry* succ; 后一个兄弟结点
*       提供方法:
*                 void set_Entry(const char*,int); 设置content、name、attr
*                 char get(int);    char& operator[](int);获取content内容
*                 set_child; 递归地设置目录文件的四类指针;childs、parent、prev、succ;
*                           如果该目录的子文件有目录，将对子目录递归调用set_child
*/
class Entry { ... };
```

Directory目录树（设计的不好，entries比较多余）：

```
/*
* 目录树
* 成员变量:
*          head 根结点
*          entries 根目录下的文件;其实就是head->childs;
* 方法:
*          Entry* set_root_dir:根据输入的字符串char* 、第一个数据扇区号、 FAT表,设置好目录树的各种链接关系;
*          Directory();构造函数,主要是初始化head
*/
class Directory { ... };
```

8.4 输入的路径处理 + 根据路径名找文件

大概流程：

1、首先要定义一个C++下没有的字符串分割函数 split

2、用split对输入的路径做划分，划分字符是 '/'

3、根据划分结果和根目录来查找：

路径是否匹配到了某个文件；

怎么查找？首先要有一个指向目录树根节点的指针p

遇到 "." 指针p不动

遇到 ".." 指针p 指向 parent; p=p->parent

遇到 文件名 ， 指针在自己的childs里找有没有同名的文件 x，有就让 p=x; 没有就报错，并且设置 bool matched=false;

4、返回：没匹配到，返回nullptr；匹配到，返回对应文件的条目Entry

5、输入是千奇百怪的，需要对以上步骤做一些细微的修正；比如如果输入路径是"//////"，那么划分后，全是空串，需要让它正确指向根结点。或者遇到“../../../”这种，要做一些规范化处理；

8.5 ls

ls命令最多有2个有用的后续参数：1个是-l，1个是目录

在匹配ls 后面是否有 -l 和 路径名时，要注意正则表达式的运用。

regex(“-l+”)可以匹配 -l 参数；

regex dir_path("(/.)|(\./.)|(\./.)|(\.\./.)"); 这个正则用于匹配ls后面是否有路径，但其实不是很准确，勉强能用

ls的具体执行：

> 1.初始化

```
//ls
else if (cmd == "ls") {
    /*
    * ————————————1.输出提示信息、初始化————————————
    * ls命令最多有2个有用的后续参数：1个是-l，1个是目录
    * has_dir:是否带有地址参数
    * has_l:是否带有-l 参数
    * dir:地址，默认为映像文件的根目录
    * wrong_param: 后续参数是否正确
    */
    string tmp;
    bool has_dir = false;
    bool has_l = false;
    bool right_param = true;
    string path = "/";
```

> 2.处理附带参数

```
/*
* ————————————2.处理 ls 附带参数————————————————
* 职责：设置好变量 has_dir、has_l、dir
* option:正则匹配 -l 的pattern，只能是 '-'开头，后面跟至少1个l，也就是正则里的'+'
* dir_path:正则匹配 目录 的pattern，以 "./"开头,或者以 "/"开头
*/
regex option( "-l+");
regex dir_path("(/.*)|(\\./.*)|(\\../.*)|(\\.\\./.*)");
while (iss >> tmp) {
//-l+ ;这里有个 !has_l 意思是如果多次输入-l,后面的-l 可以直接无视，加快处理速度
    if (!has_l && regex_match(tmp, option)) {
        has_l = true;
        //my_print(1, "ls receive -l option\n", 22);
    }

    //如果已经设置了目录参数，但又一次检查到了目录参数，直接报错
    else if (has_dir && regex_match(tmp, dir_path)) {
        //"Error!ls receives more than 1 directory\n"
        my_print(1, RES_STRINGS[5].c_str(), RES_STRINGS[5].size());
        break;
    }


    //没有设置目录，检查到了目录参数
    else if (!has_dir&&regex_match(tmp, dir_path)) {
        has_dir = true;
        path = tmp;
    }

    //检查到了不支持的参数 如 -t -x
    else {
        right_param = false;
        break;
    }
}
```

3.根据以上两步，确定是哪一种ls，然后确定从哪一个结点开始

```
    /*
    * ————————————3.输出信息————————————————————-
    * ! right_param:不支持的后续参数
    */
    if (!right_param)
        //"Unsuppoted parameter!\n"
        my_print(1, RES_STRINGS[6].c_str(), RES_STRINGS[6].size());

    else if (!has_l) {
        const Entry* e= parse_path(path,tree.head);
        if (e != nullptr)
            no_l_print(path, e);
        else
            //"Can't find target\n";
            my_print(1, RES_STRINGS[7].c_str(), RES_STRINGS[7].size());
    }

    else if (has_l) {
        const Entry* e = parse_path(path, tree.head);
        if (e != nullptr)
            l_print(path, e);
        else
            //"Can't find target\n";
            my_print(1, RES_STRINGS[7].c_str(), RES_STRINGS[7].size());
    }
}
```

4.找到 父结点后，递归的进行输出

5.

没有-l

第一行输出是当前路径

第二行输出是当前路径的所有子节点

有-l

就需要多设计2个函数，1个计算目录下直接子目录和直接文件的多少，1个计算文件的字节数；

4.一些细枝末节的修修补补

> 没有-l

```
/* 不带 -l 的ls */
void no_l_print(string path, const Entry* const p) {
    vector<Entry*> entries = p->childs;

    //在路径名最后加'/'
    if (path[path.size() - 1] != '/')
        path.push_back('/');

    //打印第一行(路径)
    string tmp = "\033[37m"+ path + ":\n";
    my_print(1, tmp.c_str(), tmp.size());
    tmp.clear();

    //打印目录内文件
    for (int i = 0; i < entries.size(); i++) {
        if (entries[i]->attr == 0x10) {
            tmp += "\033[31m" + entries[i]->name;
        }
        else
            tmp += "\033[37m"+ entries[i]->name;
        if (i == entries.size() - 1)
            tmp += "\n";
        else
            tmp += " ";
    }
    my_print(1, tmp.c_str(), tmp.size());

    //递归输出子目录
    tmp = path;
    for (auto i = 0; i < entries.size(); i++) {
        Entry* parent = entries[i];
        bool cond = parent->name != "." && parent->name != "..";
        if (cond && parent->attr == 0x10) {
            tmp += parent->name + "/";
            no_l_print(tmp, parent);
        }
    }
```

8.6 cat filename

流程： 读入文件名—— 查目录树，找对应条目——找不到，提示——找到条目——>调用函数 cat_file

函数内：寻找条目首簇号——查FAT表，找到所有该文件的簇号——根据簇号查数据区的扇区号——获取数据内容——输出

```
/* 输出文件 p 的内容
* arg:  p:要输出的文件条目
*       fat:FAT表，要查文件的所有簇号
*       data:映像文件a.img的数据区
*/
void cat_file(const Entry* const p,FAT12 fat,const char* const data) {
    //获得所有相关的簇号——获得所有扇区内容,保存在string中——输出
    vector<int> clusters = fat.get_all_clus(p->fst_clus);
    string str;
    for (size_t i = 0; i < clusters.size(); i++){
        int sec_no = clusters[i] - 2;
        if (str.empty())
            str = string(data + sec_no * SECTOR_SIZE, SECTOR_SIZE);
        else {
            string tmp(data + sec_no * SECTOR_SIZE, SECTOR_SIZE);
            str.insert(str.end(), tmp.begin(), tmp.end());
        }
    }
    str += "\n";
    cout << str;
    //my_print(1, str.c_str(), str.size());
}
```

9.颜色

http://zh.wikipedia.org/wiki/Echo_%28%E5%91%BD%E4%BB%A4%29 (wiki)

https://www.cnblogs.com/memset/p/linux_printf_with_color.html

https://blog.csdn.net/qq_29831163/article/details/105027939

https://blog.csdn.net/huntinux/article/details/13998071

直接将表示颜色的串紧贴在要输出的之前就可以了

红色 \033[31m

白色 \033[37m

比如

```
//打印第一行(路径)
string tmp = "\033[37m"+ path + ":\n";
//my_print(1, tmp.c_str(), tmp.size());
cout << tmp;
tmp.clear();
```

10.踩的大坑

1.在linux和windows平台上要注意平台差别；

在虚拟机ubuntu上，我读取起始扇区是没有问题的；

即用 fstream.read(buffer,512);读取512个字节没有问题；

但在windows上，我用 fstream.read(buffer,512)读取；

发现

buffer[508]=0x55;

buffer[509]=0xaa;

buffer[510]=0xF0;

buffer[511]=0xFF;

也就是说不知道为什么中间漏了2个字节，让起始扇区抢了FAT区的2个字节；

但什么原因我并没有搜到……

2.在windows上调试时

如果想要用nasm写的print还需要配置一些环境，可以先在windows上用cout/printf先输出，移植到虚拟机上时，再把输出换成用nasm写的 print；

在VS2019上配置nasm还是有些麻烦；

11.碎碎念

给的4个PPT都看了一遍，没有任何收获。

准备阅读《orange‘s》……读了一下第3、4、5章，收获还是很少；

至少对保护模式、FAT12等有了粗浅认识；

感觉啥都不懂；

继续看PPT；感觉得先制作一个镜像；

—11.13号

我真是大傻逼……我直接在windows上全用cout，到linux上再用nasm输出不就好了，在虚拟机上debug真痛苦

-—11.15号

写了多少天了，终于写出来了，我的代码能力真是一坨shit；

——11.17号