SHANDONG UNIVERSITY

操作系统第 4 次实验报告

网络空间安全学院 (研究院)
2021 网安三班
谢子洋 202100460116

2023 年 7 月 2 日

目录

1 文件系统设计

1.1 文件系统

1.1.1 文件与 FAT

磁盘管理以块 (block) 为基本单位, 一个硬盘分为很多块 (block),block 大小一般设置在 1-4KB. 本次实验中的模拟硬盘容量较小, 因此设置 block 大小为 16Byte.

一个文件实际上被保存到多个 block 中, 并且多个 block 不是连续分配, 使用链表串联起所有 block. 显式链表实现访问一个文件只能遍历所有的块, 导致随机访问性能低. FAT16 在此基础上将链表保存到内存里, 可任意查询某个块的下一块而不必遍历, 即文件分配表 FAT. 在实现上本文使用数组表示 FAT, 数组元素 2 字节长, 作为 FAT 的一个表项.

FAT 中表项可进行如下取值:

表 1: FAT16 表项取值

取值	含义
0000	未使用簇
0002-FFEF	分配号
FFF0-FFF6	保留
FFF7	坏簇
FFF8-FFFF	文件结束簇

1.1.2 目录及目录项

FAT 文件系统将目录同样视作文件, 并为每个文件和目录都被分配一个目录项.

一个目录项 32 字节长, 由如下部分组成:

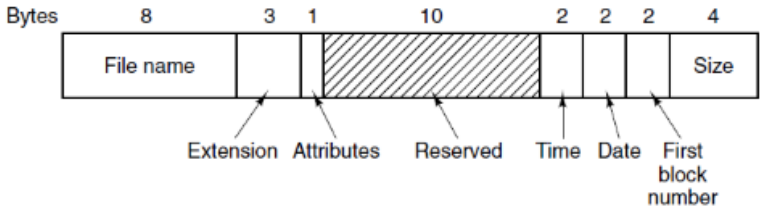


图 1: 目录项

表 2: 目录项组成

长度	组成	作用
8	File name	文件名
3	Extension	拓展名
1	Attributes	属性字节
10	Reserved	系统保留
2	Time	修改时间
2	Data	修改日期
2	First block number	文件第一个块号
4	Size	文件大小, 理论上最大 4GB

表 3: 属性字节

取值	属性
00000000	读/写
00000001	只读
00000010	隐藏
00000100	系统
00001000	卷标
00010000	子目录
00100000	存档

FAT16 文件名限长 8 字节以内, 拓展名最多只有 3 字节长. 属性字节表示目录项指向文件的属性, 如右表所示. 目录项还包括文件建立和最后修改的时间及日期, 时间被划分为时 (5bit)、分 (6bit)、秒 (5bit), 日期被划分为年 (7bit)、月 (4bit)、日 (5bit). 因此秒只能精准到 pm 2s, 年最高能表示到 2107 年 (1980+127). 文件第一个块号表示该目录项指向文件 (链表) 的第一个块号, 据此我们使用 FAT 可以正确找到该文件的所有块. FAT16 中文件大小最高为 4GB.

一个目录文件中包含多个目录项, 不同目录项同属于该目录文件, 目录项之间同样使用链表 (FAT) 连接. 因此目录项与文件 (目录) 之间实际为二维链表的结构.

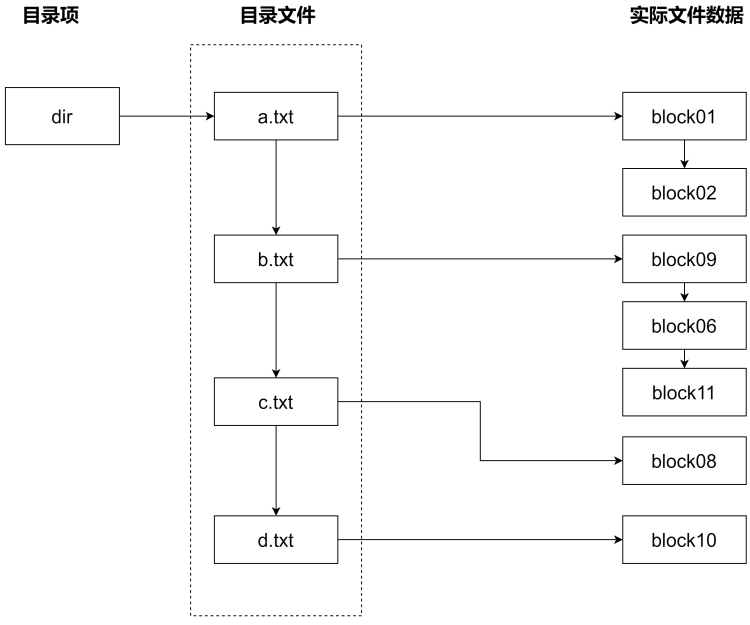


图 2: 二维链表结构

如上图所示, 单个文件内部使用 FAT 保存链表指针, 目录项与文件之间使用目录项的"第一个块号" 作为链表指针.

1.1.3 DBR

DBR 中包括一个引导程序和一个称为 BPB 的本分区参数记录表。其中 BPB 参数块记录着本分区的启始扇区、结束扇区、文件存储格式、硬盘介质描述符、根目录大小、FAT 个数、分配单元的大小等重要参数。

图 3: 完整 DBR

偏移地址	长度（字节）	含义	数值
0h	3	跳转命令	EB 3C 90
3h	8	OEM NAME	4D 53 44 4F 53 35 2E 30 = MSDOS5.0
Bh	2	每扇区字节数	00 02 = 0x200 = 512bytes
Dh	1	每簇扇区数	20 = 0x20 = 32 扇区
Eh	2	保留扇区数	01 00 = 0x1=1 扇区
10h	1	FAT数量	02 = 0x2 FAT
11h	2	根目录项数	00 02 =0x200=512
16h	2	每个FAT所占扇区数	15 00 = 0x15 = 21个扇区
18h	2	每个道所占扇区数	3F 00 = 0x3F = 63
1Ch	4	隐藏扇区数	0
36h	8	FAT NAME	FAT16
3Eh	448	引导程序执行代码	…
1FEh	2	扇区结束标志	AA 55

本次实验中省略了 DBR 中的引导程序, 并将 BPB 简化至 32 字节.

简化后 PBP 组成如下:

表 4: 简化 DBR

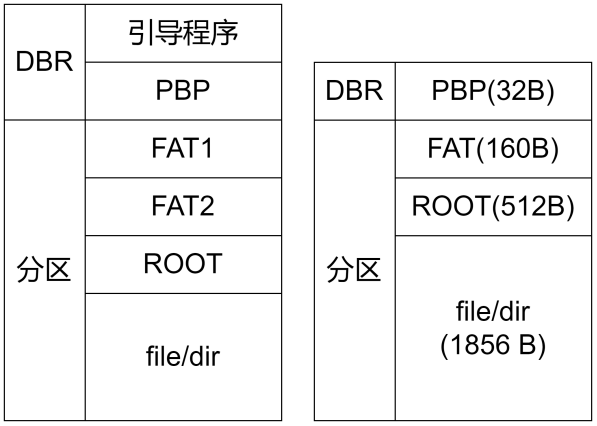
长度 (字节)	组成	数值
8	文件系统名	"FAT16"
4	每簇字节数	32
4	保留字节数	32
4	FAT 数量	1
4	单个 FAT 所占字节数	160
4	根目录项数	16
2	隐藏簇数	0
2	DBR 结束标志	0xAA55

1.1.4 根目录

根目录的作用是保存根下文件或者目录的首簇号, 以及文件的长度 (目录的长度是 0)。

FAT16 中，根目录的所占空间可以根据 BPB 中的参数-根目录项数调整。

1.1.5 FAT16 整体结构



(a) 原 FAT16 文件系统　　(b) 简化后 FAT16 文件系统

图 4: 文件系统对比

1.1.6 文件寻找

首先从 0 地址开始读取 PBP 数据, 直到检测到结束标志 0xAA55. 根据 PBP 中参数可以计算出各部分的长度及偏移量:

表 5: 分区各组成

组成部分	长度	偏移量
PBP	32	0
FAT	160	32
根目录	$512(16 \times 32)$	192
文件/目录	1856	704

因此, 我们可以找到根目录起始地址, 查找根目录下的文件/目录, 确定下一层目录/文件的第一个簇的地址.

又因为我们已知 FAT 的偏移量, 因此根据第一个簇的地址, 很容易利用链表查询到该文件所有簇的地址.

1.2 文件互斥访问

1.2.1 信号量与互斥量

信号量是实现互斥访问资源的一种方式, 可以使同一时刻有要求数量条件下线程/进程可以访问资源，可以实现进程/线程之间的同步.

对信号量有两种操作 P 和 V, 且都为原子操作:

P(s):

如果 s 的值大于零, 就给它减 1;

如果 s 值为零, 就挂起该进程的执行.

V(s):

如果有其他进程因等待 s 而被挂起, 就让它恢复运行;

如果没有进程因等待 s 而挂起, 就给 s 加 1.

信号量可以实现特定数量内进程访问有限资源; 而互斥量是信号量的特殊情况, 允许仅单个进程访问互斥资源.

1.2.2 文件互斥

在 FAT16 文件系统中, 同一文件可同时被多个进程读取, 但仅允许单个进程写入. 因此本文使用互斥量实现文件的互斥访问.

对于任意文件创建其专有的互斥量, 当进程访问 (open) 该文件时先对信号量进行 P 操作, 防止其他进程同时访问该文件. 而当进程停止访问该文件, 需要释放临界区资源, 即对信号量进行 V 操作. 使用信号量能有效避免多任务系统中的资源访问问题.

2 文件系统实现

2.1 FAT16 文件系统

2.1.1 文件系统格式化

硬盘不能直接被操作系统使用，必须先经过格式化，

将虚拟硬盘初始化为 FAT16 文件系统需要如下几步:

1. 初始化 PBP.
2. 初始化 FAT.
3. 创建根目录.
4. 文件部分初始化为 0.

```
disk::disk(char Disk[DISK_MAXLEN])
  {
      this->BLOCK_SIZE = 32;
      this->PBP_SIZE = 32;
      this->FAT_NUM = 0;
      this->FAT_SIZE = 160;
      this->ROOT_NUM = 16;
      this->CATALOG_SIZE = 32;
      this->HIDE_BLOCK_NUM = 0;

      this->DBRpoi = (Disk);
      this->FATpoi = (uint16_t*)(Disk + PBP_SIZE);
      this->ROOTpoi = (char*)(Disk + PBP_SIZE + FAT_SIZE);
      this->ENDpoi = &Disk[DISK_MAXLEN];
      //  1. 写入 DBR 部分的值.
      int* composeSize = (int*)FAT16DBR; char* poi = &Disk[0];
      strcpy(poi, "FAT16");   poi += composeSize[0];
      *((uint32_t*)poi) = 32; poi += composeSize[1];
      *((uint32_t*)poi) = 32; poi += composeSize[2];
      *((uint32_t*)poi) = 1;  poi += composeSize[3];
      *((uint32_t*)poi) = 160; poi += composeSize[4];
      *((uint32_t*)poi) = 16; poi += composeSize[5];
      *((uint16_t*)poi) = 0;  poi += composeSize[6];
      *((uint16_t*)poi) = 0xAA55; poi += composeSize[7];
      //  2. 初始化 FAT.
      int end1 = (PBP_SIZE + FAT_SIZE) / BLOCK_SIZE;
      int end2 = FAT_SIZE / 2;
      for (int i = 0; i < end1; i++) {
          FATpoi[i] = 0xFFF0;        //FFF0-FFF6 保留
      }
      for (int i = end1; i < end2; i++) {
          FATpoi[i] = 0x0000;         //未使用
      }
      //  3. 创建根(root) 目录文件,包含 "." 和 ".." 目录项
      int rootBlock = addr2ord(ROOTpoi);
      int block1 = creatDirEnt(".");
      int block2 = creatDirEnt("..");
      FATpoi[block1] = block2;
      FATpoi[block2] = FAT_END;
```

```
        dir(block1)->firstBlockNum = rootBlock;
        dir(block2)->firstBlockNum = rootBlock;
        this->curPathStr = new char[240];
        curPathStr[0] = 0x0;
        this->curPath = rootBlock;
    }
```

2.1.2 创建目录-mkdir

检查路径上任一目录是否存在, 不存在则创建该目录, 直到路径上所有目录都创建完成.

```
int disk::Command_mkdir(char* path)
{
    printf("sudo mkdir  %s\n", path);
    //初始化
    char** pathSplit;
    int step;
    int dirBlock;
    int fileBlock;
    analysPath(path, pathSplit, dirBlock, step);
    return Command_mkdir(pathSplit, dirBlock, step);
}
int disk::Command_mkdir(char** pathSplit, int dirBlock, int step)
{
    //若已存在则不进行任何操作
    //返回最深层匹配目录项
    //fileBlock指向匹配目录项
    int fileBlock;
    //逐层寻找
    int isFind = 0;
    for (int i = 1; i < step; i++)
    {
        //1. 检查该目录文件 中是否存在匹配目录项
        int _;
        isFind = findFileInDir(dirBlock, pathSplit[i], fileBlock, _);
        if (isFind == INNER_FILE_NOT_FIND)
        {   //不存在匹配项则创建 目录项并与目录文件链接
            int newBlock = creatDirAndFile(pathSplit[i]);
            int p = dir(newBlock)->firstBlockNum;
            int pp = FATpoi[p];
            dir(p)->firstBlockNum = newBlock;
            dir(pp)->firstBlockNum = dir(dirBlock)->firstBlockNum;
            FATpoi[fileBlock] = newBlock;
            FATpoi[newBlock] = FAT_END;
            fileBlock = newBlock;
        }
        //2. 进入目录文件
        dirBlock = dir(fileBlock)->firstBlockNum;
    }
    return fileBlock;
}
```

2.1.3 显示目录-ls

首先跳转到路径中最后一级目录, 遍历该目录文件中所有目录项, 并将目录项中的信息打印在屏幕上.

代码略, 详见附件.

2.1.4 删除文件-delete

跳转到路径中最后一级目录, 递归删除该目录中包含的所有目录/文件. 删除某个 block 只需将 FAT 中该 block 对应的表项改为" 未使用簇", 实际保存数据无需进行操作.

```
1     int disk::deleteDirFile(int block)
2     {
3         //input :删除文件目录项
4         //result: 递归删除该目录项指向 文件
5
6
7             //1. 删除目录项指向内容
8         if (isDir(block)) {
9             //1.1 指向目录文件则递归删除
10            int poi = dir(block)->firstBlockNum;
11            while (true)
12            {
13                int tempStore = poi;
14                poi = FATpoi[poi];
15                if (strcmp(".", dir(tempStore)->name) != 0 && strcmp("..", ...
                      dir(tempStore)->name) != 0)
16                {
17                    deleteDirFile(tempStore);
18                }
19                else
20                {
21                    FATpoi[tempStore] = FAT_NOT_USE;
22                }
23                if (poi == FAT_END)
24                {
25                    break;
26                }
27
28            }
29        }
30        else
31        {
32            //1.2 指向一般文件则直接删除
33            int curB = dir(block)->firstBlockNum;
34            int nextB;
35            while (true)
36            {
37                nextB = FATpoi[curB];
38                FATpoi[curB] = FAT_NOT_USE;
39                if (nextBlockStatus(curB) != STATUS_FIND)
40                {
41                    break;
42                }
```

```
                curB = nextB;
            }
        }
        //2. 删除目录项(不进行链接)
        FATpoi[block] = FAT_NOT_USE;
        return 1;
    }
    int disk::Command_delete(char* path)
    {
        printf("sudo rm -rf %s\n\n",path);
        //递归删除目录项下所有
        //
        //1. 定位要删除文件目录项
        int prevBlock;
        int curBlock;
        int findMark = checkPath(path, curBlock, prevBlock);
        if (findMark == PATH_NOT_EXIST)
        {
            return PATH_NOT_EXIST;
        }

        //2. 解除目录文件中目录项的链接,删除目录项指向文件
        FATpoi[prevBlock] = FATpoi[curBlock];
        deleteDirFile(curBlock);
        FATpoi[curBlock] = FAT_NOT_USE;
        return 1;
    }
```

2.1.5 打开文件-open

定义新的数据结构类型保存文件 I/O 的各种信息, 在 open() 函数中初始化该结构体.

```
struct FileStream
  {
    bool write = 0;
    bool read = 0;
    int DirBlock = -1;  //目录项块
    int byteSize = 0;
    int curBlockOrd = -1; //块号
    int curByteOrd = 0;  //块内字节序号
  };
```

2.1.6 关闭文件-close

传入上述结构体, 根据其中数据更新对应目录项属性, 如文件大小、修改日期等.

2.1.7 写入文件-write

根据新定义的 FileStream 结构体, 我们可以定位到该文件的首块号, 据此可将数据写入组成该文件的块中. 当原文件块数不足以保存新的数据时, 需要请求新的空闲块以保存额外数据.

2.1.8 读取文件-read

根据传入的 FileStream 结构体, 确定文件大小和首块号, 遍历所有的块及其中数据, 保存到缓冲区.

2.2 文件资源互斥访问

通过调用 C++ 中的"mutex" 和"condition_variable" 库, 本文实现了互斥量及其 P、V 操作.

```
class Semaphore
{
private:
    int count;
    mutex m;
    condition_variable cv;
public:
    Semaphore(int count_);
    void P() {
      unique_lock<mutex> loc(m);
      if (--count < 0) {
          cv.wait(loc);
      }
    }
    void V() {
      unique_lock<mutex> loc(m);
      if (++count ≤ 0) {
          cv.notify_one();
      }
    }
};
```

为实现文件的互斥访问, 我们在 open()、close() 函数的合适位置对信号量进行 P、V 操作. 简单来说, 在 open 函数中先对信号量进行 P 操作再访问临界区, 而在 close 函数中访问完临界区后对信号量进行 V 操作.

```
int disk::Command_open(char* path, const char* mode, FileStream& file)
{
  printf("open %s\n", path);
  //模式: r w a r+ w+ a+

  mutexSem.P();
  int dstBlock, _;
  int status = checkPath(path, dstBlock, _);
  if (status == PATH_EXIST)
  {
    file.DirBlock = dstBlock; //指向文件目录项块号
  }
  else if (status == PATH_NOT_EXIST && strcmp(mode, "w+") == 0)
  {
    dstBlock = creatEmptyFile(path);//文件不存在且模式为w+,创建文件
    if (dstBlock == -1)
    {
      printf("error");
```

```
19          return PATH_NOT_EXIST;
20        }
21        file.DirBlock = dstBlock;
22      }
23      else
24      {
25        return PATH_NOT_EXIST;
26      }
27      //设置其他参数
28      if (mode[1] == '+' || mode[0] == 'r') { file.read = 1; };
29      if (mode[1] == '+' || mode[0] == 'w' || mode[0] == 'a') { file.write = 1; };
30      file.byteSize = dir(dstBlock)->size;
31      if (mode[0] != 'a')
32      {
33        file.curBlockOrd = dir(dstBlock)->firstBlockNum;
34        file.curByteOrd = 0;
35      }
36      else
37      {
38        file.curByteOrd = dir(dstBlock)->size % BLOCK_SIZE;
39        int blockNum = (dir(dstBlock)->size - 1) / BLOCK_SIZE + 1;
40      }
41
42      return PATH_EXIST;
43    }
44
45    int disk::Command_close(FileStream& file)
46    {
47      int dirBlock = file.DirBlock;
48      dir(dirBlock)->size = file.byteSize;
49      setTime(dir(dirBlock)->date, dir(dirBlock)->time);
50      file.byteSize = 0;
51      file.curBlockOrd = 0;
52      file.curByteOrd = 0;
53      file.DirBlock = 0;
54      file.read = 0;
55      file.write = 0;
56
57      mutexSem.V();
58      return 1;
59    }
```

3 实验结果

mkdir

本文通过调用 mkdir() 函数创建多个目录, 并使用 ls() 函数展示文件组织结构.

测试结果如下图所示:

```
sudo mkdir  /home
sudo mkdir  /boot
sudo mkdir  /etc
sudo mkdir  /usr
sudo mkdir  /bin
sudo mkdir  /home/a/a1
sudo mkdir  /home/a/a2
sudo mkdir  /home/a/a3
sudo mkdir  /home/b


ls /
00010000 0Byte  2023. 7. 1 19:15: 5    .
00010000 0Byte  2023. 7. 1 19:15: 5    ..
00010000 0Byte  2023. 7. 1 19:15: 5    home
00010000 0Byte  2023. 7. 1 19:15: 5    boot
00010000 0Byte  2023. 7. 1 19:15: 5    etc
00010000 0Byte  2023. 7. 1 19:15: 5    usr
00010000 0Byte  2023. 7. 1 19:15: 5    bin

ls /home
00010000 0Byte  2023. 7. 1 19:15: 5    .
00010000 0Byte  2023. 7. 1 19:15: 5    ..
00010000 0Byte  2023. 7. 1 19:15: 5    a
00010000 0Byte  2023. 7. 1 19:15: 5    b

ls /home/a
00010000 0Byte  2023. 7. 1 19:15: 5    .
00010000 0Byte  2023. 7. 1 19:15: 5    ..
00010000 0Byte  2023. 7. 1 19:15: 5    a1
00010000 0Byte  2023. 7. 1 19:15: 5    a2
00010000 0Byte  2023. 7. 1 19:15: 5    a3

ls /home/a/a1
00010000 0Byte  2023. 7. 1 19:15: 5    .
00010000 0Byte  2023. 7. 1 19:15: 5    ..
```

图 5: mkdir+ls

open,write,read,close 打开文件并写入内容后关闭文件, 其后再次打开文件并读取数据.

测试结果如下:

```


open /home/b/file
Write data:
"When love beckons to you, follow him,Though his ways are hard and steep."

open /home/b/file
Read data:
"When love beckons to you, follow him,Though his ways are hard and steep."
```

图 6: open+r/w+close

delete

在上面的基础上又调用了 delete 函数删除了部分文件/目录, 其后再调用 ls() 函数展示删除后的文件组织结构.

```
sudo rm -rf /home/a/a1

ls /home/a
00010000 0Byte  2023. 7. 1 19:15: 5    .
00010000 0Byte  2023. 7. 1 19:15: 5    ..
00010000 0Byte  2023. 7. 1 19:15: 5    a2
00010000 0Byte  2023. 7. 1 19:15: 5    a3

ls /home/a/a1
ls: cannot access /home/a/a1: No such file or directory

sudo rm -rf /home/a

ls /home/a
ls: cannot access /home/a: No such file or directory

ls /home/a/a1
ls: cannot access /home/a/a1: No such file or directory
```

图 7: rm

4 实验心得

FAT16 文件系统在市面上使用最广泛, 兼容性最好. FAT16 文件系统的最大特点在于其使用了文件分配表 FAT, 避免了一般链表结构导致的随机访问速度慢问题.

但 FAT16 文件系统也存在很多问题: 目录项长度固定, 导致文件名与拓展名存在固定长度限制. 年份表示的最大值为 2107, 与现在十分接近.FAT 需要整个保存到内存中, 当硬盘空间较大时占用较多内存空间. 单个文件受限于目录项最大只能达到 4GB.

现代操作系统基本都是多任务操作系统，即同时有大量可调度实体在运行. 而在多任务操作系统中，同时运行的多个任务可能都需要访问/使用同一种资源. 这种多进程之间的通信及资源共享，可能会导致数据异常. 因此常使用信号量实现资源的互斥访问, 保证数据读写的准确性.

5 实验分工

代码实现: 谢子洋

论文撰写: 谢子洋

参考文献

[1] Andrew S.Tanenbaum,Herbert Bos. 现代操作系统. 北京: 机械工业出版社,2017.

A code

```
#pragma once
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
using namespace std;    // 如果不写这句，后面的mutex、cout等前面都要加std::
class Semaphore
{
private:
    int count;
    mutex m;
    condition_variable cv;
public:
    Semaphore(int count_);
    void P();
    void V();
};
```

```
  #include <iostream>
  #include <thread>
  #include <mutex>
  #include"semaphore.h"
  #include <condition_variable>
  using namespace std;


  Semaphore::Semaphore(int count_) : count(count_){}
  void Semaphore::P() {
       unique_lock<mutex> loc(m);
       if (--count < 0) {
           cv.wait(loc);
       }
   }
  void Semaphore::V() {
       unique_lock<mutex> loc(m);
       if (++count ≤ 0) {
           cv.notify_one();
       }
  }
```

```
  #pragma once
  #include<semaphore>
  #define FAT_NOT_USE  0x0000
  #define FAT_RESERVE 0xFFF0
  #define FAT_ERROE 0xFFF7
```

```
#define FAT_END 0xFFF8

#define STATUS_FIND 1
#define STATUS_NOT_USE 0
#define STATUS_RESERVED -1
#define STATUS_END -2
#define STATUS_BAD -3

#define FREE_BLOCK_NOT_FIND -1
#define FREE_BLOCK_FIND 1

#define PATH_NOT_EXIST 0
#define PATH_EXIST 1

#define INNER_FILE_NOT_FIND 0
#define INNER_FILE_FIND 1
#define DISK_MAXLEN 2560

int setTime(uint16_t& data_, uint16_t& time_);
int printTime(uint16_t data_, uint16_t time_);
void printf_bin(int num);


const int FAT16DBR[8] = { 8,4,4,4,4,4,2,2 };
struct Block
{
  //32Byte
  char* data[32];

};
struct DicEntry
{
  char name[8];
  char extension[3];
  uint8_t attributes;
  uint8_t reserved[10];

  uint16_t time;
  uint16_t date;
  uint16_t firstBlockNum;
  int size;
};
struct FileStream
{
  bool write = 0;
  bool read = 0;


  int DirBlock = -1;  //目录项块
  int byteSize = 0;

  int curBlockOrd = -1; //块号
  int curByteOrd = 0;  //块内字节序号
};
class disk
{
public:
  int BLOCK_SIZE = 0;
  int PBP_SIZE = 0;
```

```
    int FAT_NUM = 0;
    int FAT_SIZE = 0;
    int ROOT_NUM = 0;
    int CATALOG_SIZE = 0;
    int HIDE_BLOCK_NUM = 0;

    int curPath;//指向目录项
    char* curPathStr;
    char* DBRpoi;
    uint16_t* FATpoi;
    char* ROOTpoi;
    char* ENDpoi;

    disk(char Disk[DISK_MAXLEN]);
    DicEntry* dir(int blockOrd);
    char* ord2addr(int blockOrd);
    int addr2ord(char* addr);
    int analysPath(const char* path, char**& pathSplit, int& startBlock, int& stepNum);
    int findFreeBlock();
    int nextBlockStatus(int blockOrdinal);
    int creatDirEnt(const char* dirName);
    int creatDirAndFile(const char* dirName);
    int creatEmptyFile(const char* path);
    int isDir(int DirEnt);
    int findFileInDir(int dirBlock, char* name, int& dstBlock, int& prevBlock);
    int checkPath(const char* path, int& dstBlock, int& prevBlock);
    int Command_mkdir(char* path);
    int Command_mkdir(char** pathSplit, int dirBlock, int step);
    int deleteDirFile(int block);
    int Command_delete(char* path);
    int Command_open(char* path, const char* mode, FileStream& file);
    int Command_write(FileStream& file, char* data, int dataLength);
    int Command_read(FileStream& file, char* buf, int bufLength);
    int Command_close(FileStream& file);
    int Command_ls(const char* path);
};
```

```
#include"iostream"
#include"Disk.h"
#include"semaphore.h"
using namespace std;


int setTime(uint16_t& data_, uint16_t& time_)
{
  time_t now = time(0);
  tm* ltm = localtime(&now);
  ltm->tm_hour;
  //时间：秒分时
  time_ = 0;
  time_ += (ltm->tm_sec / 2) & 0b11111;
  time_ = time_ << 5;
  time_ += ltm->tm_min & 0b111111;
  time_ = time_ << 5;
  time_ += ltm->tm_hour & 0b11111;


```

```
21      //日期：日月年
22      data_ = 0;
23      data_ += (ltm->tm_mday) & 0b11111;
24      data_ = data_ << 4;
25      data_ += ltm->tm_mon & 0b1111;
26      data_ = data_ << 7;
27      data_ += ltm->tm_year & 0b1111111;
28
29
30
31      return 1;
32    }
33    int printTime(uint16_t data_, uint16_t time_)
34    {
35      unsigned int temp;
36      temp = data_ & 0b1111111;
37      printf(" %4d.", temp + 1900);
38      data_ = data_ >> 7;
39      temp = data_ & 0b1111;
40      printf("%2d.", temp + 1);
41      data_ = data_ >> 4;
42      temp = data_ & 0b11111;
43      printf("%2d ", temp);
44
45
46      temp = time_ & 0b11111;
47      printf("%2d:", temp);
48      time_ = time_ >> 5;
49      temp = time_ & 0b111111;
50      printf("%2d:", temp);
51      time_ = time_ >> 6;
52      temp = time_ & 0b11111;
53      printf("%2d    ", temp);
54      return 1;
55    }
56    void printf_bin(uint8_t num)
57    {
58      for (int i = 0; i < 8; i++)
59      {
60        if (num&0b10000000)
61        {
62          printf("1");
63        }
64        else
65        {
66          printf("0");
67        }
68        num =num<<1;
69      }
70    }
71
72
73    disk::disk(char Disk[DISK_MAXLEN])
74      {
75        this->BLOCK_SIZE = 32;
76        this->PBP_SIZE = 32;
77        this->FAT_NUM = 0;
78        this->FAT_SIZE = 160;
79        this->ROOT_NUM = 16;
```

```
80        this->CATALOG_SIZE = 32;
81        this->HIDE_BLOCK_NUM = 0;
82
83        this->DBRpoi = (Disk);
84        this->FATpoi = (uint16_t*)(Disk + PBP_SIZE);
85        this->ROOTpoi = (char*)(Disk + PBP_SIZE + FAT_SIZE);
86        this->ENDpoi = &Disk[DISK_MAXLEN];
87        //  1. 写入 DBR 部分的值.
88        int* composeSize = (int*)FAT16DBR; char* poi = &Disk[0];
89        strcpy(poi, "FAT16");   poi += composeSize[0];
90        *((uint32_t*)poi) = 32; poi += composeSize[1];
91        *((uint32_t*)poi) = 32; poi += composeSize[2];
92        *((uint32_t*)poi) = 1;  poi += composeSize[3];
93        *((uint32_t*)poi) = 160; poi += composeSize[4];
94        *((uint32_t*)poi) = 16; poi += composeSize[5];
95        *((uint16_t*)poi) = 0;  poi += composeSize[6];
96        *((uint16_t*)poi) = 0xAA55; poi += composeSize[7];
97        //  2. 初始化 FAT.
98        int end1 = (PBP_SIZE + FAT_SIZE) / BLOCK_SIZE;
99        int end2 = FAT_SIZE / 2;
100       for (int i = 0; i < end1; i++) {
101         FATpoi[i] = 0xFFF0;         //FFF0-FFF6 保留
102       }
103       for (int i = end1; i < end2; i++) {
104         FATpoi[i] = 0x0000;          //未使用
105       }
106       //    3. 创建根(root) 目录文件,包含 "." 和 ".." 目录项
107       int rootBlock = addr2ord(ROOTpoi);
108       int block1 = creatDirEnt(".");
109       int block2 = creatDirEnt("..");
110       FATpoi[block1] = block2;
111       FATpoi[block2] = FAT_END;
112       dir(block1)->firstBlockNum = rootBlock;
113       dir(block2)->firstBlockNum = rootBlock;
114       this->curPathStr = new char[240];
115       curPathStr[0] = 0x0;
116       this->curPath = rootBlock;
117     }
118     DicEntry* disk::dir(int blockOrd)
119     {
120       DicEntry* temp = (DicEntry*)ord2addr(blockOrd);
121       return (DicEntry*)ord2addr(blockOrd);
122
123     }
124     char* disk::ord2addr(int blockOrd)
125     {
126       return DBRpoi + BLOCK_SIZE * blockOrd;
127     }
128     int disk::addr2ord(char* addr)
129     {
130       return ((char*)addr - DBRpoi) / BLOCK_SIZE;
131     }
132     int disk::analysPath(const char* path, char**& pathSplit, int& startBlock, int& ...
          stepNum)
133     {
134       //设定开始块为 根目录文件首块/当前目录项
135
136       //分析路径字符串,拆分为多个文件名
137       int loopTime = strlen(path);
```

```
138         stepNum = 1;
139         for (int i = 0; i < loopTime; i++)
140         {
141           if (path[i] == '/')
142           {
143             stepNum += 1;
144           }
145         }
146         if (path[loopTime - 1] == '/') { stepNum -= 1; }
147         pathSplit = new char* [stepNum];
148
149
150
151
152         int readOrd = 0;
153         char temp = path[readOrd];
154         int in = 0;
155         for (int i = 0; i < stepNum; i++)
156         {
157           pathSplit[i] = new char[20];
158           while (temp != '/' && temp != 0x0)
159           {
160             pathSplit[i][in] = temp;
161             in++;
162             temp = path[++readOrd];
163           }
164           pathSplit[i][in] = 0x0;
165           temp = path[++readOrd];
166           in = 0;
167         }
168         if (strcmp(pathSplit[0], ""))
169         {//根目录开始
170           startBlock = addr2ord(ROOTpoi);
171           return 0;
172         }
173         startBlock = curPath;
174         return 1;
175       }
176       int disk::findFreeBlock()
177       {
178         int FAT_EntryNum = FAT_SIZE / 2;
179
180         for (int i = 0; i < FAT_EntryNum; i++)
181         {//遍历FAT每一项直到找到0x0000
182           if (FATpoi[i] == FAT_NOT_USE)
183           {
184             return i;
185           }
186         }
187         return FREE_BLOCK_NOT_FIND;
188       }
189       int disk::nextBlockStatus(int blockOrdinal)
190       {
191         //查询FAT表：给定地址确定的块 的下一个块的状态
192         uint16_t nextBlockOrdinal = FATpoi[blockOrdinal];
193         if (nextBlockOrdinal == 0) {
194           return STATUS_NOT_USE;
195         }
196         if (nextBlockOrdinal == 0xFFF7)
```

```
197         {
198           return STATUS_BAD;
199         }
200         else if ((nextBlockOrdinal ≥ 0xFFF0 && nextBlockOrdinal ≤ 0xFFF6))
201         {
202           return STATUS_RESERVED;
203         }
204         else if ((nextBlockOrdinal ≥ 0xFFF8 && nextBlockOrdinal ≤ 0xFFFF))
205         {
206           return STATUS_END;
207         }
208         return STATUS_FIND;
209       }
210
211       int disk::creatDirEnt(const char* dirName)
212       {
213         //返回簇号
214         int newBlock = findFreeBlock();
215         FATpoi[newBlock] = FAT_END;
216         strcpy(dir(newBlock)->name, dirName);
217         strcpy(dir(newBlock)->extension, "");
218         dir(newBlock)->attributes = 0b00010000;
219         dir(newBlock)->firstBlockNum = FAT_END;
220         setTime(dir(newBlock)->date, dir(newBlock)->time);
221         dir(newBlock)->size = 0;
222
223
224         return newBlock;
225       }
226       int disk::creatDirAndFile(const char* dirName)
227       {
228         //创建目录项及其指向目录文件
229         // return 目录项号
230         //目录文件中有"."".." 但并未指向前文件夹.
231
232         //目录项
233         int newBlock = creatDirEnt(dirName);
234
235
236         //创建目录文件 ,包含目录项: "." "..", 并链接
237         int block1 = creatDirEnt(".");
238         int block2 = creatDirEnt("..");
239         FATpoi[block1] = block2;
240         FATpoi[block2] = FAT_END;
241         dir(newBlock)->firstBlockNum = block1;
242
243         return newBlock;
244       }
245       int disk::creatEmptyFile(const char* path)
246       {
247
248         //1. 创建目录文件
249         char** pathSplit;
250         int step;
251         int dirBlock;
252         int fileBlock;
253         analysPath(path, pathSplit, dirBlock, step);
254         dirBlock = Command_mkdir(pathSplit, dirBlock, step - 1);//返回目录项,指向目录文件
255
```

```
256
257         //2. 创建文件
258         fileBlock = dir(dirBlock)->firstBlockNum;//目录文件第一块
259         int _;
260         int reVal = findFileInDir(fileBlock, pathSplit[step - 1], fileBlock, ...
                _);//查看匹配目录文件中是否有对应文件
261         if (reVal == INNER_FILE_FIND)
262         {
263           return -1;
264         }
265         else if (reVal == INNER_FILE_NOT_FIND)
266         {
267           //创建文件目录项块
268           int newBlock = findFreeBlock();
269           FATpoi[newBlock] = FAT_END;
270           strcpy(dir(newBlock)->name, pathSplit[step - 1]);
271           strcpy(dir(newBlock)->extension, "");
272           dir(newBlock)->attributes = 0b00000000;
273           setTime(dir(newBlock)->date, dir(newBlock)->time);
274           dir(newBlock)->size = 0;
275           FATpoi[fileBlock] = newBlock;
276           //创建文件块
277           int fileBlock = findFreeBlock();
278           FATpoi[fileBlock] = FAT_END;
279           //链接
280           dir(newBlock)->firstBlockNum = fileBlock;//匹配文件名的目录项指空白块
281           return newBlock;
282         }
283         return -1;
284       }
285
286
287       int disk::isDir(int DirEnt)
288       {
289         //判断目录项是否为
290         return dir(DirEnt)->attributes & 0b00010000;
291       }
292
293       int disk::findFileInDir(int dirBlock, char* name, int& dstBlock, int& prevBlock)
294       {
295         //查找目录文件下重名目录项
296         //dstBlock返回匹配目录项或最后一块
297
298         //tempBlock负责文件内移动
299
300         int tempBlock = dirBlock;//从一个目录文件的"."目录项开始
301         int findMark = 0;
302         while (true)
303         {
304           if (strcmp(dir(tempBlock)->name, name) == 0) {
305             findMark = 1;
306             break;
307           }
308           if (FATpoi[tempBlock] == FAT_END) { break; }
309           prevBlock = tempBlock;
310           tempBlock = FATpoi[tempBlock];
311         }
312         dstBlock = tempBlock;
313
```

```
314         if (findMark) { return INNER_FILE_FIND; }
315         return INNER_FILE_NOT_FIND;
316       }
317       int disk::checkPath(const char* path, int& dstBlock, int& prevBlock)
318       {
319         //返回匹配目录项块号/不变
320
321         //curBlock负责文件间移动
322         //innerBlock负责文件内移动
323         //tempPrev指向innerBlock前一个目录项
324
325         char** pathSplit;
326         int step;
327         int curBlock;
328         analysPath(path, pathSplit, curBlock, step);
329
330         int innerBlock = curBlock;//(step为1的情况下,目标即为)
331         int tempPrev = curBlock;
332
333         //逐层寻找,并进入下一目录文件 .项
334         int isFind = 0;
335         for (int i = 1; i < step; i++)
336         {
337           //1. 检查该目录目录文件中是否存在匹配目录项
338           isFind = findFileInDir(curBlock, pathSplit[i], innerBlock, tempPrev);
339           if (isFind == INNER_FILE_NOT_FIND)
340           {
341             return PATH_NOT_EXIST;
342           }
343           //2. 进入目录文件
344           if (i == step - 1) { break; }
345           curBlock = dir(innerBlock)->firstBlockNum;
346         }
347         dstBlock = innerBlock;
348         prevBlock = tempPrev;
349         return PATH_EXIST;
350       }
351       int disk::Command_mkdir(char* path)
352       {
353         printf("sudo mkdir  %s\n", path);
354         //初始化
355         char** pathSplit;
356         int step;
357         int dirBlock;
358         int fileBlock;
359         analysPath(path, pathSplit, dirBlock, step);
360         return Command_mkdir(pathSplit, dirBlock, step);
361       }
362       int disk::Command_mkdir(char** pathSplit, int dirBlock, int step)
363       {
364         //若已存在则不进行任何操作
365         //返回最深层匹配目录项
366         //fileBlock指向匹配目录项
367         int fileBlock;
368         //逐层寻找
369         int isFind = 0;
370         for (int i = 1; i < step; i++)
371         {
372           //1. 检查该目录文件 中是否存在匹配目录项
```

```
373         int _;
374         isFind = findFileInDir(dirBlock, pathSplit[i], fileBlock, _);
375         if (isFind == INNER_FILE_NOT_FIND)
376         {   //不存在匹配项则创建 目录项并与目录文件链接
377           int newBlock = creatDirAndFile(pathSplit[i]);
378           int p = dir(newBlock)->firstBlockNum;
379           int pp = FATpoi[p];
380           dir(p)->firstBlockNum = newBlock;
381           dir(pp)->firstBlockNum = dir(dirBlock)->firstBlockNum;
382           FATpoi[fileBlock] = newBlock;
383           FATpoi[newBlock] = FAT_END;
384           fileBlock = newBlock;
385         }
386         //2. 进入目录文件
387         dirBlock = dir(fileBlock)->firstBlockNum;
388       }
389       return fileBlock;
390     }
391
392     int disk::deleteDirFile(int block)
393     {
394       //input :删除文件目录项
395       //result: 递归删除该目录项指向 文件
396
397
398         //1. 删除目录项指向内容
399       if (isDir(block)) {
400         //1.1 指向目录文件则递归删除
401         int poi = dir(block)->firstBlockNum;
402         while (true)
403         {
404           int tempStore = poi;
405           poi = FATpoi[poi];
406           if (strcmp(".", dir(tempStore)->name) != 0 && strcmp("..", ...
                  dir(tempStore)->name) != 0)
407           {
408             deleteDirFile(tempStore);
409           }
410           else
411           {
412             FATpoi[tempStore] = FAT_NOT_USE;
413           }
414           if (poi == FAT_END)
415           {
416             break;
417           }
418
419         }
420       }
421       else
422       {
423         //1.2 指向一般文件则直接删除
424         int curB = dir(block)->firstBlockNum;
425         int nextB;
426         while (true)
427         {
428           nextB = FATpoi[curB];
429           FATpoi[curB] = FAT_NOT_USE;
430           if (nextBlockStatus(curB) != STATUS_FIND)
```

```
            {
              break;
            }
            curB = nextB;
          }
        }
        //2. 删除目录项(不进行链接)
        FATpoi[block] = FAT_NOT_USE;
        return 1;
      }
      int disk::Command_delete(char* path)
      {
        printf("sudo rm -rf %s\n\n",path);
        //递归删除目录项下所有
        //
        //1. 定位要删除文件目录项
        int prevBlock;
        int curBlock;
        int findMark = checkPath(path, curBlock, prevBlock);
        if (findMark == PATH_NOT_EXIST)
        {
          return PATH_NOT_EXIST;
        }

        //2. 解除目录文件中目录项的链接,删除目录项指向文件
        FATpoi[prevBlock] = FATpoi[curBlock];
        deleteDirFile(curBlock);
        FATpoi[curBlock] = FAT_NOT_USE;
        return 1;
      }

      Semaphore mutexSem(1);

      int disk::Command_open(char* path, const char* mode, FileStream& file)
      {
        printf("open %s\n", path);
        //模式: r w a r+ w+ a+

        mutexSem.P();
        int dstBlock, _;
        int status = checkPath(path, dstBlock, _);
        if (status == PATH_EXIST)
        {
          file.DirBlock = dstBlock; //指向文件目录项块号
        }
        else if (status == PATH_NOT_EXIST && strcmp(mode, "w+") == 0)
        {
          dstBlock = creatEmptyFile(path);//文件不存在且模式为w+,创建文件
          if (dstBlock == -1)
          {
            printf("error");
            return PATH_NOT_EXIST;
          }
          file.DirBlock = dstBlock;
        }
        else
        {
          return PATH_NOT_EXIST;
        }
```

```
490
491         //设置其他参数
492         if (mode[1] == '+' || mode[0] == 'r') { file.read = 1; };
493         if (mode[1] == '+' || mode[0] == 'w' || mode[0] == 'a') { file.write = 1; };
494         file.byteSize = dir(dstBlock)->size;
495         if (mode[0] != 'a')
496         {
497           file.curBlockOrd = dir(dstBlock)->firstBlockNum;
498           file.curByteOrd = 0;
499         }
500         else
501         {
502           file.curByteOrd = dir(dstBlock)->size % BLOCK_SIZE;
503           int blockNum = (dir(dstBlock)->size - 1) / BLOCK_SIZE + 1;
504         }
505
506         return PATH_EXIST;
507       }
508
509       int disk::Command_write(FileStream& file, char* data, int dataLength)
510       {
511         file.byteSize += dataLength;
512         int writeNum = 0;
513         while (dataLength != writeNum)
514         {
515           char* addr = ord2addr(file.curBlockOrd);
516           addr[file.curByteOrd] = data[writeNum];
517           file.curByteOrd++;
518           writeNum++;
519           if (file.curByteOrd == BLOCK_SIZE)
520           {
521             file.curByteOrd = 0;
522             if (nextBlockStatus(file.curBlockOrd) == STATUS_FIND)
523             {
524               file.curBlockOrd = FATpoi[file.curBlockOrd];
525             }
526             else
527             {
528               //创建新文件块
529               int newBlock = findFreeBlock();
530               FATpoi[file.curBlockOrd] = newBlock;
531               FATpoi[newBlock] = FAT_END;
532               file.curBlockOrd = newBlock;
533             }
534
535           }
536         }
537         return 1;
538       }
539
540       int disk::Command_read(FileStream& file, char* buf, int bufLength)
541       {
542         //将文件所有数据存放到buf数组,要求bufLength≥数据长度,否则不予读取
543         int totalLength = file.byteSize;
544         int readNum = 0;
545         if (bufLength < totalLength) {
546           return 0;
547         }
548         while (readNum != totalLength)
```

```
549         {
550           char* addr = ord2addr(file.curBlockOrd);
551           buf[readNum] = addr[file.curByteOrd];
552           file.curByteOrd++;
553           readNum++;
554           if (file.curByteOrd == BLOCK_SIZE)
555           {
556             file.curByteOrd = 0;
557             file.curBlockOrd = FATpoi[file.curBlockOrd];
558           }
559         }
560         buf[readNum] = 0;
561       }
562
563       int disk::Command_close(FileStream& file)
564       {
565         int dirBlock = file.DirBlock;
566         dir(dirBlock)->size = file.byteSize;
567         setTime(dir(dirBlock)->date, dir(dirBlock)->time);
568         file.byteSize = 0;
569         file.curBlockOrd = 0;
570         file.curByteOrd = 0;
571         file.DirBlock = 0;
572         file.read = 0;
573         file.write = 0;
574
575         mutexSem.V();
576         return 1;
577       }
578
579       int disk::Command_ls(const char* path)
580       {
581
582         printf("ls %s\n", path);
583         //1. 找到目录文件
584         int dstBlock;
585         int _;
586         int pathExist = checkPath(path, dstBlock, _);
587         if (pathExist == PATH_NOT_EXIST)
588         {
589           printf("ls: cannot access %s: No such file or directory\n\n",path);
590           //printf(" %s NOT EXIST\n", path);
591           return 0;
592         }
593         dstBlock = dir(dstBlock)->firstBlockNum;//目录项移动到目录文件首块
594
595         //2.显示目录文件中所有目录项
596
597         while (true)
598         {
599           //printf("%2hx", dir(dstBlock)->attributes);
600           printf_bin(dir(dstBlock)->attributes);
601           cout << " " << dir(dstBlock)->size << "Byte ";
602           printTime(dir(dstBlock)->date, dir(dstBlock)->time);
603           cout << dir(dstBlock)->name << endl;
604           if (nextBlockStatus(dstBlock) == STATUS_FIND)
605           {
606             dstBlock = FATpoi[dstBlock];
607           }
```

```
608           else
609           {
610             break;
611           }
612         }
613         printf("\n");
614         return 1;
615       }
```