实验五-文件系统 实验报告

孙汉武 16281047 安全1601

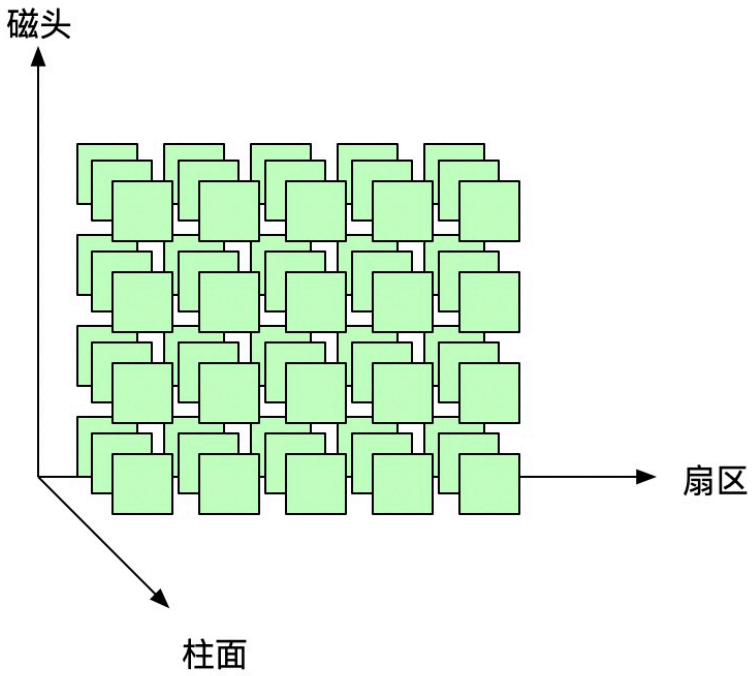
[实验源码链接:https://github.com/sunhanwu/16281047_OperatingSystemExperiment/tree/master/lab5](https://github.com/sunhanwu/16281047_OperatingSystemExperiment/tree/master/lab5)

一 概要设计

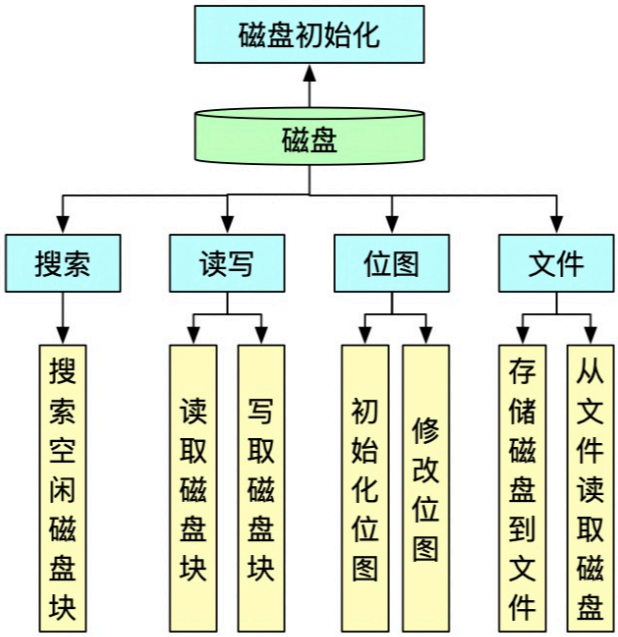
本次的实验的实验目的是在模拟的I/O系统中开发一个简单的文件系统，并且提供一些借口给用户用于交互，从实验目的可以看出，本实验重点在于构建模拟的I/O系统和基于I/O系统的文件系统。所以，在概要设计中，将详细介绍模拟I/O系统的设计、文件系统的设计和测试模块的设计这三个部分。

1.1 I/O系统设计

IO系设计首先要解决的是需要有一个物理磁盘，为此，我们通过定义一个三维的磁盘块结构体数组表示物理磁盘，该结构体数组的每一个维度分别表示物理磁盘中的一个层次。



我们模拟的磁盘如上所示，第一维表柱面，第二维表示磁头，最后一维表示扇区。磁盘定义好之后需要定义一系列的函数用于操作磁盘。IO系统工作的流程和结构如下所示：



IO系统提供的操作磁盘的API如上图所示，主要分为五个大类，分别是初始化磁盘、磁盘搜索、磁盘读写、磁盘位图处理、磁盘与文件转化等，下面的表格分别介绍了各个类别提供的API的详细信息。

- 磁盘初始化

函数名	参数	返回值	功能
InitDisk()	无	无	初始化磁盘数组

- 磁盘搜索

函数名	参数	返回值	功能
SearchBitMap	无	空闲磁盘块号	搜索并返回最小的空闲磁盘块号

- 磁盘读写

函数名	参数	返回值	功能
ReadBlock	int i :指定读取的磁盘块号 char *p： 返回读取内容	无	读取指定磁盘块内容
WriteBlock	int i：指定写入的磁盘块号 char *p:写入的内容	无	写入内容到指定磁盘块

- 磁盘位图处理

函数名	参数	返回值	功能
InitBitMap	无	无	初始化位图
ChangeBitMap	int i:要修改的磁盘块号 char p:修改的内容(Y/N)	无	修改位图每一位的值

- 磁盘与文件的转化

由于是在内从中创建数组模拟物理磁盘，所以这种方式无法模拟物理磁盘断电不丢失信息的特性。为了满足这个要求，设计一下的函数用于将数组中的信息存储到文件中和文件中读取相关信息。

函数名	参数	返回值	功能
DiskToFile	char filename[]：文件名	无	将磁盘数组信息存储为文件
FileToDsik	char filename[]:文件名	无	将文件读取到磁盘数组中

1.2 文件系统设计

在上一节设计的IO系统的基础上，进行文件系统的设计，文件系统设计时有两个很重要的概念，分别是文件描述符和目录项这两个数据结构的定以及在这个基础上进行的一系列操作。

1. 用户接口

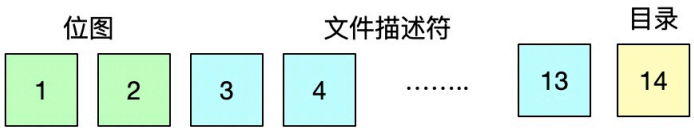
文件系统提供了一系列便于用户操作的接口，用于对文件系统中的文件进行增删改查，具体接口信息如下：

函数名	参数	返回值	功能
create	char filename[]	无	创建文件
destroy	char filename[]	无	删除文件
open	char filename[]	无	打开文件
close	char filename[]	无	关闭文件
read	index:文件描述符号 mem_area:读取的位置 count:读取的字节数	无	读取文件内容
write	index:文件描述符 mem_area:x写入的位置 count:写入字节数	无	向文件中写入信息
lseek	index:文件描述符 号 pos:位置	无	移动文件读写指针

文件系统提供的上述接口已经可以满足对文件系统的常规操作。上面的接口中提到的文件描述符在文件系统中是一个很重要的概念，每个文件都必须 通过一个文件描述符来表示 ， 其文件长度信息 ， 文件存储位置等常规 信息都存储在文件描述符中。

为此我设计了一个结构体`FileDescriptor`用于表示文件描述符，并且将该结构体进行4字节对齐，方便后续以二进制形式存储在文件中。

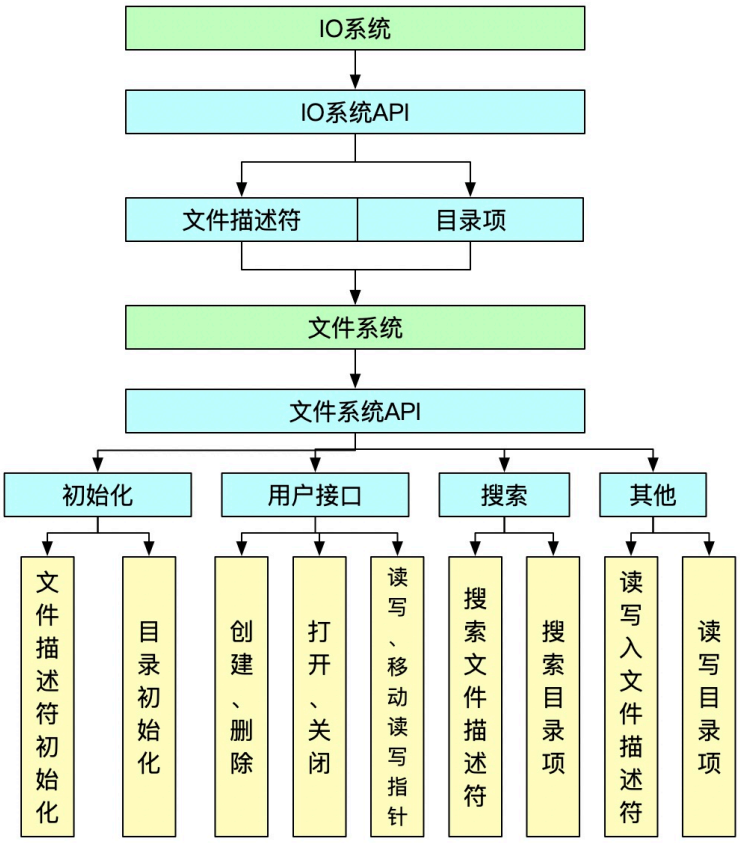
在文件系统中另外一个重要概念就是目录，因此定义一个结构体表示目录项，用于存储文件名和文件描述符号。



文件描述符和目录项在磁盘中存储的位置如上图所示，其中磁盘块的第一和第二块用于存储磁盘块的位图，第3-13块用于存储文描述符，14块用于存储目录项信息。

经过4字节对齐之后的文件描述符结构体大小为24字节，而一个磁盘块有512字节存储空间，所以一个磁盘块最多存储21个文件描述符，而本实验中设置的用于存储文件描述符的磁盘块为10块，最多可以存储210个文件描述符。

目录也可以看做一个文件，所以也会占据一个文件文件描述符，本实验中目录占用的文件描述符是第一个文件描述符。并且存储目录信息的磁盘是14号磁盘，经过4字节对齐的目录项结构体大小为16B，所以最多有32个文件。



文件系统的API如上图所示，我将文件系统的API分为初始化、用户接口、搜索和其他这四类，每一个大类具体的函数如下表所示：

- 文件系统初始化

函数名	参数	返回值	功能
InitFileDescriptor	无	无	初始化文件描述符数组
InitMenu	无	无	初始化目录项数组

- 文件系统用户接口(上面已经介绍过，不在赘述)
- 文件系统搜索

函数名	参数	返回值	功能
SearchFileDescriptor	无	无	搜索空闲的文件描述符
SearchMenuItem	无	无	搜索空闲的目录项

- 其他

这部分主要定义的是一些方便操作的函数，例如将文件描述符数组写入磁盘块中去，将目录系统写入磁盘块等等。

函数名	参数	返回值	功能
DiskToFileDescriptor	无	无	从磁盘块中恢复文件描述符数组
FileDescriptorToDisk	无	无	将文件描述符数组信息写入磁盘块
MenuToFileDescriptor	无	无	将目录项数组写入第一个文件描述符
FileDescriptorToMenu	无	无	将第一个文件描述符内容恢复到目录项数组

到此，文件系统部分所有数据结构和函数均介绍完毕

1.3 菜单系统设计

完成IO系统的设计和文件系统的设计，需要对上述的功能设计一个外壳程序，即一个用户界面便于使用，总结文件系统的所有功能，设计的菜单驱动程序包含如下两层菜单。

1. 一级菜单
    - 创建新磁盘系统
    - 从文件中恢复历史磁盘系统
2. 二级菜单
    - 查看目录
    - 创建文件
    - 删除文件
    - 打开文件
    - 修改文件
    - 查看位图
    - 保存磁盘
    - 退出

二 I/O系统

2.1 磁盘块结构体

> I/O系统部分全部代码都在IO.h文件中

1. 磁盘块结构体BLOCK

设计整个I/O系统的基础就是设计模拟磁盘块的结构体，并用结构体数组代表磁盘，通过定义三维的结构体数组来模拟出整个磁盘的物理结构，这三维分别代表柱面、磁头和扇区。下面是磁盘块(逻辑块)结构体的定义：

```
typedef struct BLOCK
{
    char Content[512]; //逻辑块存储的内容
    int BlockNnum; //逻辑块号
    int c; // 柱面号
    int h; //磁头号
    int b; //扇区号
}BLOCK;
```

磁盘块结构体最重要的成员就是用于存储信息Content，这是一个字符型数组，大小为512个字节。而另外的逻辑块号，柱面号，磁头号和扇区号这四个成员是为了便于后续程序设计的。下面的表格详细表示每个成员的信息：

成员名	类型	大小	作用
Content	字符数组	512字节	存储磁盘块内容
BlockNum	整型	4字节	磁盘块的逻辑块号
c	整型	4字节	柱面号
h	整型	4字节	磁头号
b	整型	4字节	扇区号

2. 物理磁盘(BLOCK数组)

磁盘块结构体定义好之后，可以模拟出一个磁盘块，但是完整的磁盘是一个三维的磁盘块结构体数组构成的。

```
#define C 10 // 柱面号
#define H 10 //磁头号
#define B 10 //扇区号
BLOCK ldisk[C][H][B];//磁盘模型
```

在`IO.h`直接定义ldisk数组，模拟物理磁盘。并且通过宏定义C、H和B三个量调整物理磁盘的大小。

3. 磁盘初始化函数

在模拟物理磁盘的三维结构体数组定义好之后，需要对该数组进行初始化，对数组中的每个元素，即每个磁盘块进行初始化。

```
void InitDisk(void)
{
    for(int i=0;i<C;i++)
        for(int j=0;j<H;j++)
            for(int k=0;k<B;k++)
            {
                ldisk[i][j][k].c=i;
                ldisk[i][j][k].h=j;
                ldisk[i][j][k].b=k;
                ldisk[i][j][k].BlockNnum=DiskNumToBlock(i,j,k);//计算对应的逻辑块号
            }
}
```

可以看到对磁盘块数组进行初始化的方式非常简单，主要包含两个工作，一个就是顺序编号其柱面号、磁头号和扇区号；还有一个功能就是计算逻辑块号

2.2 磁盘读写

说道磁盘系统的API函数，最重要的两个函数就是对磁盘块进行读写操作的两个函数。下面分别介绍这个两个函数的详细内容。

1. 读磁盘块函数：`ReadBlock(int i,char *p):`

```
void ReadBlock(int i,char *p)
{
    int c,h,b;//磁盘的柱面 磁道 扇区
    c = i % (H*B);//
    h = (i -c*H*B) % B;//
    b = i-c*H*B - h*B;//
    memcpy(p,ldisk[c][h][b].Content,512);
}
```

3-6行计算逻辑号为i的逻辑块在磁盘系统中的柱面号、磁头号和扇区号；

第7行可以看作此函数的核心操作，完成的工作就是将制定磁盘块中的内容通过`memcpy`函数复制到字符型指针p中去。

> 注意：这里不能使用strcpy函数复制字符串，因为strcpy函数在复制的时候会遇到第一个\0就会停止复制，但是磁盘块中存储的信息可能不是连续的字符串，可能是一些其他信息，例如文件描述符等，这个时候就会碰到一些空位置用\0补充，但是后面还有有用的信息，所以使用memcp函数，按照指定字节数复制，而不考虑\0的问题

2. 写磁盘块函数：`WriteBlock(int i,char *p):`

```
1  void WriteBlock(int i,char *p)
2  {
3      int c,h,b;
4      b = i % B;
5      h = ((i - b) / B) % H;
6      c = (i -b -h*B) / (H*B);
7      b = i -c*H*B -h*B ;
8      memcpy(ldisk[c][h][b].Content,p,512);
9  }
```

3-6行操作与上面相似，就是计算柱面号、磁头号和扇区号这三个参数

8第8行主要用于将参数p指针中的内容复制到磁盘块中

2.3 磁盘位图

为了方便查询磁盘中的空闲磁盘块，直接遍历查询的效率非常低，所以本实验中采用了位图的方式来表示磁盘块的占用与否，一个字符表示一个磁盘块的占用与否，其中Y表示占用，N表示空闲。而位图编号就是磁盘块的逻辑块号数。

1. 初始化磁盘位图函数：`InitBitMap(void)：`

```
1  void InitBitMap(void)
2  {
3      //第0，1号磁盘已经被占用
4      ChangeBitMap(0,'Y');
5      ChangeBitMap(1,'Y');
6      for (int i=2;i<C*H*B;i++)
7      {
8          ChangeBitMap(i,'N');
9      }
10 }
```

可以看到初始化函数中将磁盘块数量(C*B*H)个单位的字符修改为N，表示为占用。4-5两行表示第一块和第二块物理磁盘初始化就被占用，因为这两块物理磁盘用于存储位图。

2. 修改位图函数：`ChangeBitMap(int i,char p):`

```
1  void ChangeBitMap(int i,char p)
2  {
3      if(i <512)
4          ldisk[0][0][0].Content[i] = p;
5      else
6          ldisk[0][0][1].Content[i-512] = p;
7  }
```

这个函数使用一个if结构判断逻辑块号，如果大于512的话存入 第二个磁盘块中，否则存入第一个磁盘块中。

2.4 磁盘文件的存取

在内存中定义的磁盘块结构体数组无法满足断电后信息还能保存的特性，因此需要内存中的磁盘块数组中的信息保存到文件中去，需要的时候再加载出来。

1. 将磁盘数组保存为文件：`DiskToFile(char filename[])`：

```
1  void DiskToFile(char filename[])
2  {
3  //    FileDescriptorToDisk();
4      FILE *fp;
5      fp = fopen(filename,"wb");
6      //判断fp打开成功
7      if (fp ==NULL)
8      {
9          printf("File Open Fail");
10         exit(1);
11     }
12     // 循环遍历，将磁盘块内容写入二进制文件中去
13     for(int i=0;i<C;i++)
14         for(int j=0;j<H;j++)
15             for(int k=0;k<B;k++)
16                 //以二进制的形式写入二进制文件中
17                 fwrite(ldisk[i][j][k].Content,512,1,fp);
18     fclose(fp);
19 }
```

3-5行以二进制写入形式打开指定文件名的文件

7-11行判断文件是否打开成功，没打开成功的话输出错误信息并退出程序

13-17行，遍历磁盘数组，以二进制的形式将磁盘块中存储的内容写入到文件中去。最后关闭文件

2. 加载文件到磁盘块数组：`FileToDisk(char filename[])`：

```
1  //从文件中读取数据，恢复磁盘系统
2  void FileToDisk(char filename[])
3  {
4      FILE *fp;
5      fp = fopen(filename,"rb");
6      if(fp == NULL)
7      {
8          printf("File Open Fail");
9          exit(1);
10     }
11     int index = 0;
12     while(!feof(fp))
13     {
14         int c,h,b;
15         b = index % B;
```

```
            h = ((index - b) / B) % H;
            c = (index -b -h*B) / (H*B);
            b = index -c*H*B -h*B ;
            fread(ldisk[c][h][b].Content,512,1,fp);
            ldisk[c][h][b].c = c;
            ldisk[c][h][b].h = h;
            ldisk[c][h][b].b = b;
            index++;
        }
        fclose(fp);
}
```

这段程序除了打开文件等常规操作之外，核心的代码是while循环中的，首先计算逻辑号为index的逻辑块的柱面号、磁头号和扇区号；之后每次读取512字节数据到对应磁盘块数组中的磁盘块中去。

2.5 空闲磁盘块搜索

2.3节中定义了位图，用于存储磁块的空闲状态，所以当需要使用磁盘块的时候 ，需要查询位图找到一个空闲磁盘块号返回。

1. 空闲磁盘块搜索函数：`SearchBitMap(void)：

```
//搜索位图，找到空闲磁盘块号
int SearchBitMap(void)
{
    for(int i=14;i<C*H*B;i++)
    {
        if(i<512)
        {
            if(ldisk[0][0][0].Content[i]=='N')
                return i;
        }
        else
        {
            if(ldisk[0][0][1].Content[i-512]=='N')
                return i;
        }
    }
}
```

本函数的实现方式就是通过遍位图知道找到一个空闲的磁盘块。不过由于所有的位图 信息并不是全部存储在一个磁盘块中，而是两个磁盘块，所以在遍历的时候需要判断在哪个磁盘块。

三 文件系统

> 文件系统全部代码存储在FS.h文件中

3.1 文件描述符结构体 & 目录项结构体

1. 文件描述符结构体定义

文件系统采用文件描述符来记录每一个文件的信息，下面是文件描述符的结构体定义：

```
#pragma pack(4)
typedef struct FileDescriptor //此文件描述符总共占据磁盘24字节
{
    int Length;//文件长度
    int DiskNum[DiskNumLen]; //第二个3只是表示每个磁盘块好最大长度是3位
    int Num; //文件描述符号
    char IsFree; //表示此文件描述符是否空闲
}FileDescriptor;
#pragma pack(pop)
```

> 在这个结构体定义有一个特别的地方需要注意，就是使用了4字节对齐机制，因为后来这些结构体需要存储到字符型数组中，如果不采用对齐的话可能会导致不同结构体的长度不同，读取的时候就没办法读取.

下面是该结构体各个成员的详细解释：

成员名	类型	大小	作用
Length	整型	4字节	存储文件大小
DiskNum	整型数组	12字节	存储文件内容的磁盘块好数组
Num	整型	4字节	文件描述符号
IsFree	字符型	4字节(对齐后)	表示当前描述符时候空闲

2. 目录项结构体定义

目录是文件系统必不可缺的组成部分，本实验中通过目录项数组组成一个目录，而每个目录项由文件名和文件描述符号组成。下面是目录项结构体定义：

```
#pragma pack(4)
typedef struct MenuItem //目录对应0号文件描述符,一个目录项占据16字节，所以一个文件描述符可以存储96个文件
{
    char FileName[12]; //目录项中文件名的最大长度为16字节
    int FileDescriptorNum;//文件描述符号
}MenuItem;
#pragma pack(pop)
```

> 同样目录项结构体也是经过4字节对齐的，作用与上面的文件描述符结构体相似

成员名	类型	大小	作用
FileName	字符型数组	12字节	存储文件名
FileDescriptorNum	整型	4字节	文件描述符号

3.2 文件系统初始化

文件系统的初始化包括对文件描述符数组的初始化和目录项数组的初始化，和IO系统中的磁盘数组初始化一样，文件系统的初始化也就是对这两个数组进行一些编号操作等基本操作。

1. 文件描述符初始化：`InitFileDescriptor(void):`

```
void InitFileDescriptor(void)
{
    for(int i=0;i<256;i++)
    {
      int  DiskNum[3]; //磁盘号数组
      DiskNum[0] = i;
      DiskNum[1] = -1;
      DiskNum[2] = -1;
      int FileDescriptorNum = i; //文件描述符号

 ChangeFileDescriptor(&filedescriptor[i],0,DiskNum,FileDescriptorNum,'Y');
    }
    filedescriptor[0].IsFree = 'N';
    FileDescriptorToDisk();
}
```

首先遍历整个文件描述符数组，进行编号并且初始化的时候文件描述符对应的三个磁盘块只分配一个。

2. 目录项初始化

```
void InitMenu(void)
{
    for(int i=0;i<32;i++)
        menuitem[i].FileDescriptorNum = i;
}
```

3.3 文件系统用户接口

1. 创建文件：`create(char filename[]):`

```
//下面是文件系统与用户直接的接口
void create(char filename[])
{
    int FileDescriptorNum,MenuItemNum,DiskNum;
```

```
    //寻找空闲目录项
    MenuItemNum = SearchMenuItem();
    strcpy(menuitem[MenuItemNum].FileName,filename);
    //寻找空闲文件描述符
    FileDescriptorNum = SearchFileDescriptor();
    menuitem[MenuItemNum].FileDescriptorNum = FileDescriptorNum;
    //寻找空闲磁盘块
    DiskNum = SearchBitMap();
    filedescriptor[FileDescriptorNum].DiskNum[0] = DiskNum;
    filedescriptor[FileDescriptorNum].IsFree = 'N';
    //修改磁盘位图
    ChangeBitMap(DiskNum,'Y');
}
```

创建一个新的文件的时候，首先需要搜索一个空闲的目录项，将文件名存储在目录项中，然后在搜索一个空闲的描述符，分配该文件描述符给他文件，在搜索 一个空闲的磁盘块，将该磁盘块存储在文件描述符中。最后修改文件描述符状态和位图对应磁盘块的状态为占用。

2. 删除文件：`destroy(char filename[])`：

```
void destroy(char filename[])
{
    int MenuItemNum=-1;
    for(int i=0;i<32;i++)
        if(strcmp(menuitem[i].FileName,filename)==0)
            MenuItemNum = i;
    if(MenuItemNum==-1)
    {
        printf("目录中没有此文件！\n");
        return;
    }
    int FileDesCriptorNum = menuitem[MenuItemNum].FileDescriptorNum;
    //将目录项重置,重置时只需要将文件名删除，而不需要重置文件描述符，因为前面判断文件是否存在的条件是文件名是否存在
    memset(menuitem[MenuItemNum].FileName,0,sizeof(menuitem[MenuItemNum].FileName));
    //修改文件描述符为空闲状态
    filedescriptor[FileDesCriptorNum].IsFree = 'Y';
    for(int i=0;i<3;i++)
    {
        if(filedescriptor[FileDesCriptorNum].DiskNum[i]!=-1)
        ChangeBitMap(filedescriptor[FileDesCriptorNum].DiskNum[i],'N');
    }
}
```

根据文件名在目录项数组中搜索对应的目录项，删除文件名，再找到目录项后读取该文件对应的文件描述符号，修改文件描述符状态为空闲；再讲文件描述符中记录的所有磁盘块状态全部改为空闲。

3. 打开文件：`open(char filename[])`

```
int open(char filename[])
{
    int MenuItemNum=-1;
    for(int i=0;i<32;i++)
        if(strcmp(menuitem[i].FileName,filename)==0)
            MenuItemNum = i;
    if(MenuItemNum==-1)
    {
        printf("目录中没有此文件！\n");
        return -1;
    }
    else
        //返回文件描述符号
        return menuitem[MenuItemNum].FileDescriptorNum;
}
```

这个函数通过遍历目录项数组，找到文件名符合的目录项，读取其文件描述符号返回，没有找到的话打印错误信息并返回-1

4. 读取文件：`read(int index,int mem_area,int count)：`

```
char* read(int index,int mem_area,int count)
{
   char *temp;
   char block[512];
   temp = (char *)malloc(count* sizeof(char));
   ReadBlock(filedescriptor[index].DiskNum[0],block);
   memcpy(temp,&block[mem_area],count);
   return temp;
}
```

读取文件内容函数首先找到文件描述符中的磁盘号，然后调用IO系统提供的读取磁盘块的接口读取该磁盘块，读取后按照要求取对应位置指定长度的数据返回。

5. 写文件：`write(int index,int mem_area,int count,char content[])：`

```
void write(int index,int mem_area,int count,char content[])
{
    char temp[512];
    char *s1=(char*)malloc(mem_area* sizeof(char));
    char *s2=(char*)malloc(mem_area* sizeof(char));
    char *s;
    int DiskNum = filedescriptor[index].DiskNum[0];
    ReadBlock(DiskNum,temp);
    memcpy(s1,temp,mem_area);
    memcpy(s2,&temp[mem_area],512-mem_area);
    s = strcat(s1,content);
    s = strcat(s,s2);
```

```
    filedescriptor[index].Length = strlen(s);
    WriteBlock(DiskNum,s);
}
```

写文件的时候需要考虑可能是在原来文件的基础上，在某段插入一些内容，所以用s1字符指针保存mem_area之前的信息，s2保存mem_area之后的信息，加入要加入的内容后在连接成为一个完整的字符数组，最后调用IO系统提供的写入磁盘块接口写入 对应磁盘块。

3.4 搜索文件系统

上面文件系统的用户接口中 很多地方用到了搜索文件描述符、搜索目录项等操作，所以需要单独写几个函数用于搜索文件描述符和目录项等结构。

1. 搜索空闲文件描述符:`SearchFileDescriptor():`

```
int SearchFileDescriptor()
{
    for(int i=0;i<256;i++)
    {
        if(filedescriptor[i].IsFree == 'Y')
            return i;
    }
    return -1;
}
```

遍历文件描述符数组，找到空闲的文件描述符号返回

2. 搜索空闲目录项：`SearchMenuItem()：`

```
int SearchMenuItem()
{
    for(int i=0;i<32;i++)
    {
        if(strlen(menuitem[i].FileName)==0)
            return i;
    }
    return -1;
}
```

遍历所有的目录项数组，知道找到空闲的目录项返回目录项号

3.5 其它文件系统函数

除了上面介绍的文件操作之外，还需一些函数，例如将文件描述符写入到磁盘中去，将目录项数组写入到第一个文件描述符对应的磁盘中；从磁盘中恢复文件描述符数组，恢复目录项数组。

1. 将文件描述符数组写入磁盘：`FileDescriptorToDisk(void)：`

```
//将文件描述符写入磁盘中
```

```
void FileDescriptorToDisk(void)
{
    char temp_block[512];
    int index = 0;
    int DiskNumIndex = 2;
    for(int i=0;i<256;i++)
    {
        char temp_descriptor[24];
        memcpy(temp_descriptor,&filedescriptor[i],
sizeof(FileDescriptor));
        memcpy(&temp_block[index*24],temp_descriptor,24);
        index++;
        int t = index % 21;
        if(t == 0)
        {
            index = 0;
            if(DiskNumIndex<10)
            {
                memcpy(ldisk[0][0][DiskNumIndex].Content,temp_block,512);
                ChangeBitMap(DiskNumIndex,'Y');
            }
            else
            {
                memcpy(ldisk[0][1][DiskNumIndex-
B].Content,temp_block,512);
                ChangeBitMap(DiskNumIndex,'Y');//修改位图
            }
            DiskNumIndex++;
        }
    }
}
```

遍历整个文件描述符数组，每个文件描述符占据24字节信息，21个文件描述符一组，一共504字节，将每组504字节信息存入到一个磁盘块中，存入后修改磁盘的状态为占用。

2. 从磁盘恢复文件描述符数组：`DiskToFileDescriptor(void)`：

```
//将磁盘读取的信息恢复
void DiskToFileDescriptor(void)
{
    for(int i=2;i<15;i++)
    {
        char temp[512];
        if (i<B)
            memcpy(temp,ldisk[0][0][i].Content,512);
        else
            memcpy(temp,ldisk[0][1][i-B].Content,512);
        for (int j=0;j<21;j++)
        {
```

```
            if(((i-2)*21+j)>256)
                break;
            char temp_FileDescriptor[24];
            memcpy(temp_FileDescriptor,&temp[j*24],24);
            FileDescriptor *f;
            f = (FileDescriptor*)temp_FileDescriptor;
            int num = (i-2)*21+j;
            filedescriptor[num].IsFree = f->IsFree;
            filedescriptor[num].DiskNum[0] = f->DiskNum[0];
            filedescriptor[num].DiskNum[1] = f->DiskNum[1];
            filedescriptor[num].DiskNum[2] = f->DiskNum[2];
            filedescriptor[num].Length = f->Length;
            filedescriptor[num].Num = f->Num;
        }
    }
}
```

由于每个磁盘块的空间只能存储21个文件描述符，所以每隔21就需要将index归零一次，用于从新读取一个新的磁盘的文件描述符信息，每次读取的是一整个磁盘的信息，长度是512字节，而每个文件描述符的大小为24，所以首先全部服务磁盘块信息到`temp_block`，然后每次读取24字节信息到`temp_descriptor`,之后通过强制类型转换，将字符换数组转化为文件描述符结构体指针，这样就将磁盘块中的信息读入。

3. 将目录项数组写入文件描述符：`MenuToFileDescriptor(void)`：

```
//将目录内容写入文件描述中
void MenuToFileDescriptor(void)
{
    char temp_FileDescriptor[512];
    for(int i=0;i<32;i++)
    {
        char temp_menuitem[16];
        memcpy(temp_menuitem,&menuitem[i],16);
        memcpy(&temp_FileDescriptor[i*16],temp_menuitem,16);
    }
    filedescriptor[0].IsFree = 'N';
    filedescriptor[0].DiskNum[0] = SearchBitMap();
    filedescriptor[0].Length = 512;
    WriteBlock(filedescriptor[0].DiskNum[0],temp_FileDescriptor);
}
```

将所有的目录项合并为一个512字节的字符数组(注意使用的是memcpy而不是strcpy)然后将第一个文件描述符的状态修改位占用，并将字符数组写入第一个文件描述符对应的磁盘块。

4. 从第一个文件描述符恢复目录项数组：`FileDescriptorToMenu(void)`：

```
void FileDescriptorToMenu(void)
{
    char MenuContent[512];
```

```
    ReadBlock(filedescriptor[0].DiskNum[0],MenuContent);
    for(int i=0;i<32;i++)
    {
        char temp_menuitem[16];
        memcpy(temp_menuitem,&MenuContent[i*16],16);
        MenuItem *t;
        t = (MenuItem *)temp_menuitem;
        strcpy(menuitem[i].FileName,t->FileName);
        menuitem[i].FileDescriptorNum = t->FileDescriptorNum;
    }
}
```

首先读取第一个文件描述符对应的磁盘块信息到`MenuContent`字符数中去，然后每次读取16字节信息，将读取的16字节信息强制转换为目录项指针。这样磁盘上存储的所欲目录项信息就会被全部读取。

四 菜单系统

> 菜单系统代码在main.cpp文件中

IO系统和文件系统准备好之后就可以更具需要的功能设计出具体的功能，并对应写出一个菜单系统。

对应的菜单系统有如下函数：

1. 查看目录函数：`ShowDir()`

```
void ShowDir()
{
    int index =1;
    int exist = 0;
    printf("****************目录**********************\n");
    printf("当前目录下文件有:\n");
    for(int i=0;i<32;i++)
    {
        if(strlen(menuitem[i].FileName)!=0)
        {
            printf("%d %s
%dB\n",index,menuitem[i].FileName,filedescriptor[menuitem[i].FileDescripto
rNum].Length);
            exist++;
        }
        index++;
    }
    printf("一共存在%d个文件\n",exist);
    printf("****************************************\n");
}
```

可以看到这个查看目录函数遍历目录项数组，打印所有非空目录项 内容，包括文件名和文件大小，最后统计出一共存在多少个文件。

2. 打印位图：`ShowBitMap()`

```
void ShowBitMap(void)
{
    printf("\n***************位图*********************\n");
    int used = 0;
    printf("当前的磁盘使用情况如下(Y表示使用，N表示未使用)\n");
    for(int i=0;i<C;i++)
    {
        printf("%d号柱面磁盘信息如下:\n",i);
        printf("    区:0 1 2 3 4 5 6 7 8 9\n");
        printf("头\n");
        for(int j=0;j<H;j++)
        {
            printf("%d\t:",j);
            for(int k=0;k<B;k++)
            {
                int t=i*H*B+j*B+k;
                if(t<512)
                {
                    printf("%c ",ldisk[0][0][0].Content[t]);
                    if(ldisk[0][0][0].Content[t] == 'Y')
                        used++;
                }
                else
                {
                    printf("%c ",ldisk[0][0][1].Content[t-512]);
                    if(ldisk[0][0][0].Content[t] == 'Y')
                        used++;
                }
            }
            printf("\n");
        }
    }
    printf("总共使用%d个磁盘块，剩余%d个磁盘块空闲\n",used,(C*B*H-used));
    printf("****************************************\n");
}
```

这个函数的大部分代码在进行位图打印信息的排版，每个柱面为一页，每一页中每一行表示一个磁头，每一列表示一个扇区。最后统计出所有磁盘的使用占比。

3. 主菜单程序

> 由于整个函数代码太长，所以只展示核心代码，完整代码请查看github

```
switch (choice2)
{
    case 1:ShowDir();break;
    case 2:
```

```
        printf("请输入要创建的文件名：");
        scanf("%s",filename);
        create(filename);
        break;
    case 3:
        printf("请输入要删除的文件名：");
        scanf("%s",filename);
        destroy(filename);
        break;
    case 4:
        printf("请输入要打开的文件名：");
        scanf("%s",filename);
        ReadFile(filename);
        break;
    case 5:
        int choice3;
        printf("1. 增加内容\t2. 删除内容");
        printf("\n请选择：");
        scanf("%d",&choice3);
        printf("请输入要修改的文件名:");
        scanf("%s",filename);
        if(choice3==1)
            ChangeFileAdd(filename);
        else if(choice3==2)
            ChangeFileDel(filename);

        break;
    case 6:
        ShowBitMap();
        break;
    case 7:
        printf("请输入要保存的文件名：");
        scanf("%s",filename);
        save(filename);
        break;
    case 8:
        flag=1;
        break;
    default:
        break;
}
```

这个switch结构提供个8个选择，对应8个功能。

五 文件系统测试

IO系统、文件系统和菜单系统完成之后需要对文件系统进行测试，下面是测试的详细过程。

5.1 测试概述

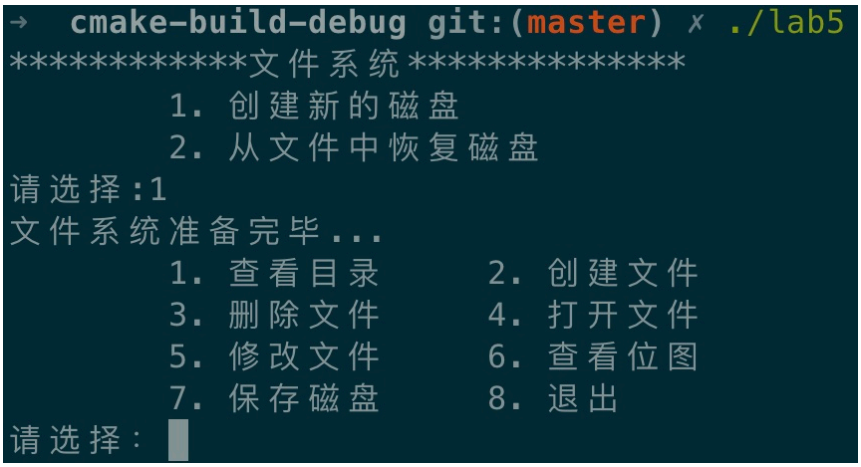
测试部分分别测试IO系统、文件系统对应的功能，测试计划如下所示：

测试名称	测试描述	被测试模块
保存磁盘文件测试	将当前磁盘信息存入二进制文件	IO系统磁盘写入文件功能、菜单系统等
读取磁盘文件测试	从文件系统中读取磁盘文件，装载到磁盘系统中	IO系统磁盘写入文件功能、菜单系统等
目录查看测试	查看当前目录中存在的所有文件	文件系统目录模块、文件描述符模块； 菜单系统
文件创建测试	创建新的文件	文件系统目录模块、文件系统用户接口
文件删除测试	删除已有(不存在)文件	文件系统目录模块、文件系统用户接口
打开文件测试	打开并查看文件内容	文件系统用户接口
修改文件测试	修改文件内容	文件系统目录模块、文件系统用户接口，IO系统
查看位图测试	查看当前磁盘位图信息	IO系统位图模块，菜单系统

5.2 系统测试

1. 保存磁盘文件测试 & 文件创建测试

- 保存磁盘系统之前首先需要创建一个新的磁盘系统



- 在新的文件系统中创建文件

```
请选择： 2
请输入要创建的文件名： test.txt
        1. 查看目录      2. 创建文件
        3. 删除文件      4. 打开文件
        5. 修改文件      6. 查看位图
        7. 保存磁盘      8. 退出
请选择： 1
*****************目录***********************
当前目录下文件有：
1 test.txt 0B
一共存在1个文件
****************************************
```

> 可以看到查看文件目录的时候看到刚刚创建的目录

- 给刚刚创建的文件添加一点内容

```
****************************************
        1. 查看目录      2. 创建文件
        3. 删除文件      4. 打开文件
        5. 修改文件      6. 查看位图
        7. 保存磁盘      8. 退出
请选择： 5
1. 增加内容      2. 删除内容
请选择： 1
请输入要修改的文件名:test.txt

请输入要增加的位置(向后增加)： 0
请输入增加的字节数： 20
请输入增加的内容:HelloWorldHelloWorld
HelloWorldHelloWorldorld
        1. 查看目录      2. 创建文件
        3. 删除文件      4. 打开文件
        5. 修改文件      6. 查看位图
        7. 保存磁盘      8. 退出
请选择： 4
请输入要打开的文件名： test.txt
HelloWorldHelloWorldorld
```

> 可以看到添加了内容之后内容保存到文中去

- 再次查看目录

```
*****************目录***********************
当前目录下文件有：
1 test.txt 24B
一共存在1个文件
```

可以看到文件的长度确实发生了变化

- 保存磁盘文件

```
**************************************
        1. 查看目录       2. 创建文件
        3. 删除文件       4. 打开文件
        5. 修改文件       6. 查看位图
        7. 保存磁盘       8. 退出
请选择： 7
请输入要保存的文件名： test.dat
```

> 将磁盘系统保存到test.dat中去

- 为了验证刚刚的磁盘信息确实保存了下来，使用xxd工具查看test.dat文件的内容

```
→  cmake-build-debug git:(master) ✗ xxd test.dat | more
00000000: 5959 5959 5959 5959 5959 5959 5959 594e  YYYYYYYYYYYYYYN
00000010: 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e  NNNNNNNNNNNNNNN
00000020: 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e  NNNNNNNNNNNNNNN
00000030: 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e  NNNNNNNNNNNNNNN
00000040: 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e  NNNNNNNNNNNNNNN
00000050: 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e  NNNNNNNNNNNNNNN
00000060: 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e  NNNNNNNNNNNNNNN
00000070: 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e 4e4e  NNNNNNNNNNNNNNN
```

> 可以看到test.dat中前面是位图信息，共占用15个磁盘块，标志位'Y',其他所有磁盘块状态为空闲，标志位'N'

- 再检索刚刚创建的文件是否存在

```
→  cmake-build-debug git:(master) ✗ xxd test.dat | grep Hello
00001c00: 4865 6c6c 6f57 6f72 6c64 4865 6c6c 6f57  HelloWorldHelloW
→  cmake-build-debug git:(master) ✗ xxd test.dat | grep test
00001e00: 7465 7374 2e74 7874 0000 0000 0100 0000  test.txt........
```

> 可以看到检索'Hello'和文件名'test'的时候都有对应内容，说明磁盘信息缺失保存了下来

2. 读取磁盘文件测试 & 目录查看测试 & 打开文件测试

- 打开上面测试中保存的`test.dat`文件

```
➜  cmake-build-debug git:(master) ✗ ./lab5
************文件系统**************
        1. 创建新的磁盘
        2. 从文件中恢复磁盘
请选择:2
请输入加载文件名:test.dat
文件系统准备完毕...
        1. 查看目录      2. 创建文件
        3. 删除文件      4. 打开文件
        5. 修改文件      6. 查看位图
        7. 保存磁盘      8. 退出
请选择： 1
*****************目录************************
当前目录下文件有：
1 test.txt 24B
一共存在1个文件
*****************************************
        1. 查看目录      2. 创建文件
        3. 删除文件      4. 打开文件
        5. 修改文件      6. 查看位图
        7. 保存磁盘      8. 退出
请选择：
```

> 可以看到磁盘信息全部恢复了

- 打开文件内容具体查看一下，内容是否存在变化

```
*****************************************
        1. 查看目录      2. 创建文件
        3. 删除文件      4. 打开文件
        5. 修改文件      6. 查看位图
        7. 保存磁盘      8. 退出
请选择： 4
请输入要打开的文件名： test.txt
HelloWorldHelloWorldorld
```

> 可以看到文件内容没有发生改变

3. 文件删除测试

- 为了便于进行文件删除测试，首先先创建一个文件`test2.txt`

```
        1. 查看目录      2. 创建文件
        3. 删除文件      4. 打开文件
        5. 修改文件      6. 查看位图
        7. 保存磁盘      8. 退出
请选择: 2
请输入要创建的文件名: test2.txt
        1. 查看目录      2. 创建文件
        3. 删除文件      4. 打开文件
        5. 修改文件      6. 查看位图
        7. 保存磁盘      8. 退出
请选择: 1
**********目录**********
当前目录下文件有:
1 test.txt 24B
2 test2.txt 0B
一共存在2个文件
```

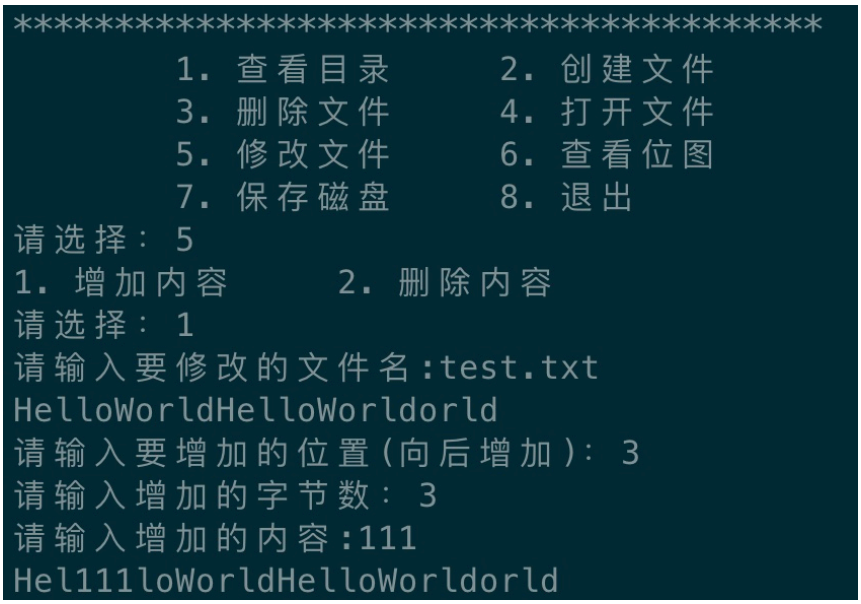
- 删除文件`test2.txt`

```
****************************************
        1. 查看目录      2. 创建文件
        3. 删除文件      4. 打开文件
        5. 修改文件      6. 查看位图
        7. 保存磁盘      8. 退出
请选择: 3
请输入要删除的文件名: test2.txt
        1. 查看目录      2. 创建文件
        3. 删除文件      4. 打开文件
        5. 修改文件      6. 查看位图
        7. 保存磁盘      8. 退出
请选择: 1
**********目录**********
当前目录下文件有:
1 test.txt 24B
一共存在1个文件
```

4. 修改文件测试

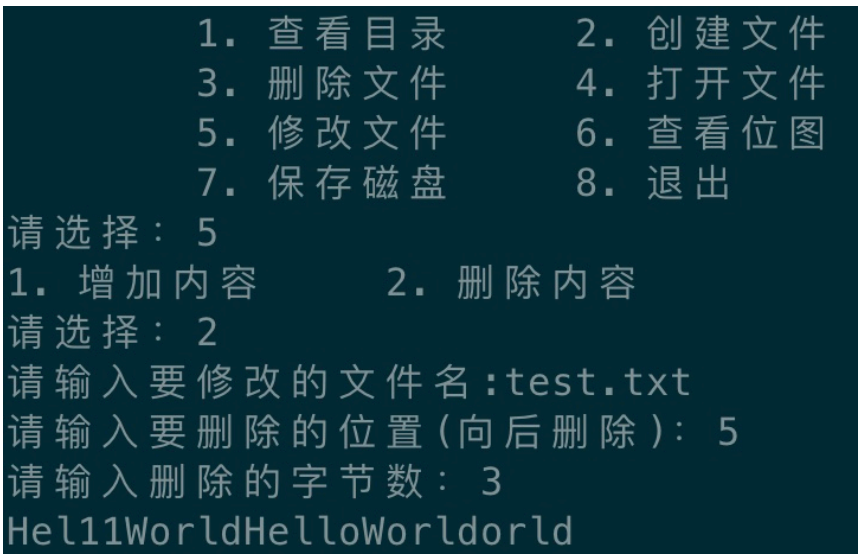
修改文件测时候分为在原有文件的基础上增加内容和删除内容，我们这在`test.txt`的基础上进行增加和删除操作

- 在`test.txt`上增加内容



> 可以看到在指定位置增加了指定的内容

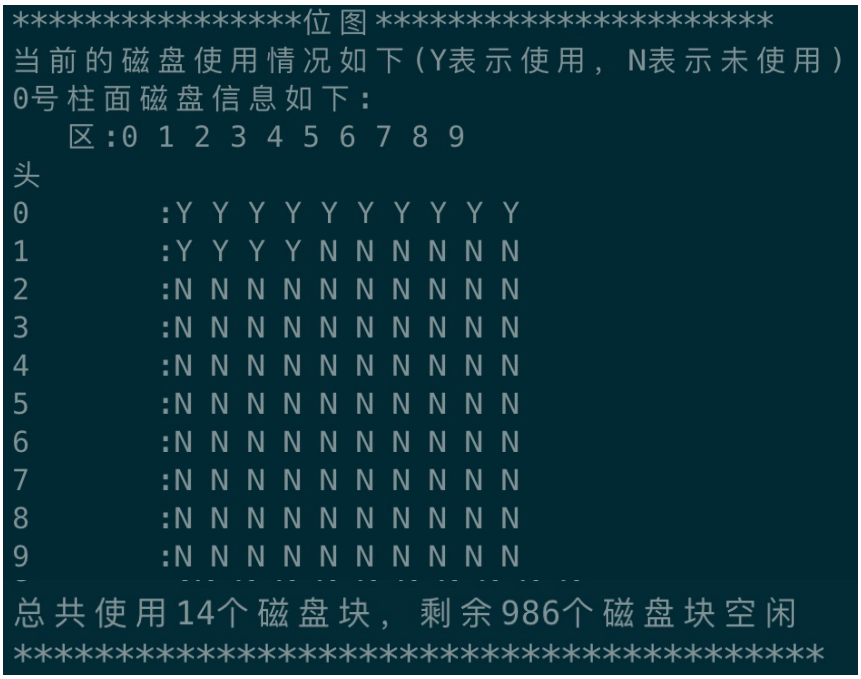
- 在`test.txt`删除内容



> 可以看到指定位置的指定内容被删除了

5. 查看位图测试

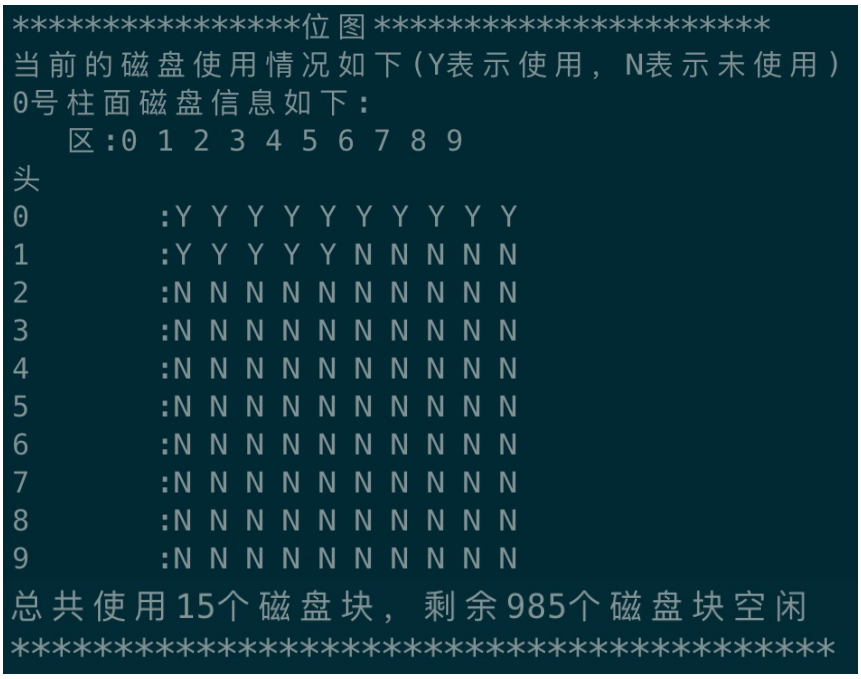
- 文件系统在初始化之后应该有15个磁盘块被占用，其中2个用于存储位图，12个用于存储文件描述符，1个用于存储目录



> 可以看到0号柱面的磁盘位图信息如上所示,总共使用14个磁盘块，其他全部空闲

- 增加一个文件之后，查看磁盘位图

增加文件之后，当文件内容小于一个磁盘块大小时，暂时只分配一个磁盘块，所以应该只占用前个磁盘块



六 实验总结

通过本次实验收获到了许多的东西，也许到了很多知识。实验之前，看完整个实验要求之后没有一个整体的思路就开始编写程序，导致后期的时候很多地方考虑不够全面，各个系统中函数组织混乱，整个构架不够完整，本次实验收获到的第一点就是在进行实验之前一定要提前设计好实验的思路，最好做好整个概要设计；第二点收获就是对文件系统有了更加深刻的认知，实验从最底层的磁盘块开始模拟，一点一点到IO系统，再到文件系、目录等等，通过自己的实践更加深刻的了解了文件系统；最后一点 收获就是在编程能力上的收获，这次通过编写这个文件系统，再次巩固了自己对于C语言的掌握能力，并

且了解到了以前所用的处理字符串的一系列函数的缺点，例如strcpy只能复制\0之前的内容，strcat只能连接两个字符换的可见内容等等，学会了新的函数memcpy，通过这个函数实现将结构体以二进制的形式存储到字符串中和将字符串再恢复到结构体中。