```c
#include <stdio.h>
#include <string.h>
#include "ext2_func.h"
#include "shell.h"
#include <time.h>
#include <stdlib.h>

char current_path[256]; //当前路径名
char path_head[256]; //路径名头

static unsigned long getCurrentTime() {
    // 获取当前时间
    time_t rawtime;
    time(&rawtime);

    // 将 time_t 转换为 unsigned long
    unsigned long currentTimeStamp = (unsigned long)rawtime;

    return currentTimeStamp;
}

char* convertTimeStampToString(unsigned long timestamp) {
    // 将 unsigned long 转换为 time_t
    time_t rawtime = (time_t)timestamp;

    // 将 time_t 转换为 struct tm
    struct tm *timeinfo = localtime(&rawtime);

    // 格式化时间
    static char buffer[20];  // 用于存储格式化后的时间字符串
    strftime(buffer, sizeof(buffer), "%Y.%m.%d %H:%M:%S", timeinfo);

    return buffer;
}

static void update_super_block(void) //写超级块
{
    fp=fopen("./Ext2","r+");
    fseek(fp,DISK_START,SEEK_SET);
    fwrite(sb_block,SB_SIZE,1,fp);
    fflush(fp); //立刻将缓冲区的内容输出，保证磁盘内存数据的一致性
}
```

```c
static void reload_super_block(void) //读超级块
{
    fseek(fp,DISK_START,SEEK_SET);
    fread(sb_block,SB_SIZE,1,fp);//读取内容到超级块缓冲区中
}

static void update_group_desc(void) //写组描述符
{
    fp=fopen("./Ext2","r+");
    fseek(fp,GDT_START,SEEK_SET);
    fwrite(gdt,GD_SIZE,1,fp);
    fflush(fp);
}

static void reload_group_desc(void) // 读组描述符
{

    fseek(fp,GDT_START,SEEK_SET);
    fread(gdt,GD_SIZE,1,fp);
}

static void update_inode_entry(unsigned short i) // 写第i个inode
{
    fp=fopen("./Ext2","r+");
    fseek(fp,INODE_TABLE+(i-1)*INODE_SIZE,SEEK_SET);
    fwrite(inode_area,INODE_SIZE,1,fp);
    fflush(fp);
}

static void reload_inode_entry(unsigned short i) // 读第i个inode
{
    fseek(fp,INODE_TABLE+(i-1)*INODE_SIZE,SEEK_SET);
    fread(inode_area,INODE_SIZE,1,fp);
}

static void update_dir(unsigned short i) //   写第i个 数据块
{
    fp=fopen("./Ext2","r+");
    fseek(fp,DATA_BLOCK+i*BLOCK_SIZE,SEEK_SET);
    fwrite(dir,BLOCK_SIZE,1,fp);
    fflush(fp);
}
```

```c
static void reload_dir(unsigned short i) // 读第i个 数据块
{
    fseek(fp,DATA_BLOCK+i*BLOCK_SIZE,SEEK_SET);
    fread(dir,BLOCK_SIZE,1,fp);
    //fclose(fp);
}

static void update_block_bitmap(void) //写block位图
{
    fp=fopen("./Ext2","r+");
    fseek(fp,BLOCK_BITMAP,SEEK_SET);
    fwrite(bitbuf,BLOCK_SIZE,1,fp);
    fflush(fp);
}

static void reload_block_bitmap(void) //读block位图
{
    fseek(fp,BLOCK_BITMAP,SEEK_SET);
    fread(bitbuf,BLOCK_SIZE,1,fp);
}

static void update_inode_bitmap(void) //写inode位图
{
    fp=fopen("./Ext2","r+");
    fseek(fp,INODE_BITMAP,SEEK_SET);
    fwrite(ibuf,BLOCK_SIZE,1,fp);
    fflush(fp);
}

static void reload_inode_bitmap(void) // 读inode位图
{
    fseek(fp,INODE_BITMAP,SEEK_SET);
    fread(ibuf,BLOCK_SIZE,1,fp);
}

static void update_block(unsigned short i) // 写第i个数据块
{
    fp=fopen("./Ext2","r+");
    fseek(fp,DATA_BLOCK+i*BLOCK_SIZE,SEEK_SET);
    fwrite(Buffer,BLOCK_SIZE,1,fp);
    fflush(fp);
}
```

```
static void reload_block(unsigned short i) // 读第i个数据块
{
    fseek(fp,DATA_BLOCK+i*BLOCK_SIZE,SEEK_SET);
    fread(Buffer,BLOCK_SIZE,1,fp);
}


static int alloc_block(void) // 分配一个数据块,返回数据块号
{

    //bitbuf共有512个字节，表示4096个数据块。根据last_alloc_block/8计算它在bitbuf的哪一个字节

    unsigned short cur=last_alloc_block;
    //printf("cur: %d\n",cur);
    unsigned char con=128; // 1000 0000b
    int flag=0;
    if(gdt[0].bg_free_blocks_count==0)
    {
        printf("There is no block to be alloced!\n");
        return(0);
    }
    reload_block_bitmap();
    cur/=8;
    while(bitbuf[cur]==255)//该字节的8个bit都已有数据
    {
        if(cur==511)cur=0; //最后一个字节也已经满，从头开始寻找
        else cur++;
    }
    while(bitbuf[cur]&con) //在一个字节中找具体的某一个bit
    {
        con=con/2;
        flag++;
    }
    bitbuf[cur]=bitbuf[cur]+con;
    last_alloc_block=cur*8+flag;

    update_block_bitmap();
    gdt[0].bg_free_blocks_count--;
    update_group_desc();
    return last_alloc_block;
}
```

```
static int get_inode(void) // 分配一个inode
{
    unsigned short cur=last_alloc_inode;
    unsigned char con=128;
    int flag=0;
    if(gdt[0].bg_free_inodes_count==0)
    {
        printf("There is no Inode to be alloced!\n");
        return 0;
    }
    reload_inode_bitmap();

    cur=(cur-1)/8;   //第一个标号是1，但是存储是从0开始的
    //printf("%s",)
    while(ibuf[cur]==255) //先看该字节的8个位是否已经填满
    {
        if(cur==511)cur=0;
        else cur++;
    }
    while(ibuf[cur]&con)  //再看某个字节的具体哪一位没有被占用
    {
        con=con/2;
        flag++;
    }
    ibuf[cur]=ibuf[cur]+con;
    last_alloc_inode=cur*8+flag+1;
    update_inode_bitmap();
    gdt[0].bg_free_inodes_count--;
    update_group_desc();
    return last_alloc_inode;
}

//当前目录中查找文件或目录为tmp，并得到该文件的 inode 号，它在上级目录中的数据块号以及数据块中目录的项号

static unsigned short reserch_file(char tmp[9],int file_type,unsigned short *inode_num,

{
    unsigned short j,k;
    reload_inode_entry(current_dir); //进入当前目录
    j=0;
    while(j<inode_area[0].i_blocks)
    {
        reload_dir(inode_area[0].i_block[j]);
        k=0;
        while(k<32)
        {
            if(!dir[k].inode||dir[k].file_type!=file_type||strcmp(dir[k].name,tmp))
            {
                k++;
            }
            else
            {
                *inode_num=dir[k].inode;
                *block_num=j;
                *dir_num=k;
                return 1;
            }
        }
        j++;
    }
    return 0;
}

/*为新增目录或文件分配 dir_entry
对于新增文件，只需分配一个inode号
对于新增目录，除了inode号外，还需要分配数据区存储.和..两个目录项*/

static void dir_prepare(unsigned short tmp,unsigned short len,int type)
{
    reload_inode_entry(tmp);

    if(type==2) // 目录
    {
        inode_area[0].i_size=32;
        inode_area[0].i_blocks=1;
        inode_area[0].i_block[0]=alloc_block();
        unsigned long temp_time=getCurrentTime();
        inode_area[0].i_atime=temp_time;
        inode_area[0].i_ctime=temp_time;
        inode_area[0].i_mtime=temp_time;
        inode_area[0].i_dtime=0;
        dir[0].inode=tmp;
        dir[1].inode=current_dir;
        dir[0].name_len=len;
        dir[1].name_len=current_dirlen;
        dir[0].file_type=dir[1].file_type=2;
        for(type=2;type<32;type++)
            dir[type].inode=0;
        strcpy(dir[0].name,".");
        strcpy(dir[1].name,"..");
        update_dir(inode_area[0].i_block[0]);

        inode_area[0].i_mode=0b0000001000000110;
    }
    else
    {
        inode_area[0].i_size=0;
        inode_area[0].i_blocks=0;
        inode_area[0].i_mode=0b0000000100000110;
        unsigned long temp_time=getCurrentTime();
        inode_area[0].i_atime=temp_time;
        inode_area[0].i_ctime=temp_time;
        inode_area[0].i_mtime=temp_time;
        inode_area[0].i_dtime=0;

    }
    update_inode_entry(tmp);
}


//删除一个块号

static void remove_block(unsigned short del_num)
{
    unsigned short tmp;
    tmp=del_num/8;
    reload_block_bitmap();
    switch(del_num%8) // 更新block位图 将具体的位置为0
    {
        case 0:bitbuf[tmp]=bitbuf[tmp]&127;break; // bitbuf[tmp] & 0111 1111b
        case 1:bitbuf[tmp]=bitbuf[tmp]&191;break; //bitbuf[tmp]  & 1011 1111b
        case 2:bitbuf[tmp]=bitbuf[tmp]&223;break; //bitbuf[tmp]  & 1101 1111b
        case 3:bitbuf[tmp]=bitbuf[tmp]&239;break; //bitbbuf[tmp] & 1110 1111b
        case 4:bitbuf[tmp]=bitbuf[tmp]&247;break; //bitbuf[tmp]  & 1111 0111b
        case 5:bitbuf[tmp]=bitbuf[tmp]&251;break; //bitbuf[tmp]  & 1111 1011b
        case 6:bitbuf[tmp]=bitbuf[tmp]&253;break; //bitbuf[tmp]  & 1111 1101b
        case 7:bitbuf[tmp]=bitbuf[tmp]&254;break; // bitbuf[tmp] & 1111 1110b
    }
    update_block_bitmap();
    gdt[0].bg_free_blocks_count++;
    update_group_desc();
}


//删除一个inode 号

static void remove_inode(unsigned short del_num)
{
    unsigned short tmp;
    tmp=(del_num-1)/8;
    reload_inode_bitmap();
    switch((del_num-1)%8)//更改block位图
    {
        case 0:bitbuf[tmp]=bitbuf[tmp]&127;break;
        case 1:bitbuf[tmp]=bitbuf[tmp]&191;break;
        case 2:bitbuf[tmp]=bitbuf[tmp]&223;break;
        case 3:bitbuf[tmp]=bitbuf[tmp]&239;break;
        case 4:bitbuf[tmp]=bitbuf[tmp]&247;break;
        case 5:bitbuf[tmp]=bitbuf[tmp]&251;break;
        case 6:bitbuf[tmp]=bitbuf[tmp]&253;break;
        case 7:bitbuf[tmp]=bitbuf[tmp]&254;break;
    }
    update_inode_bitmap();
    gdt[0].bg_free_inodes_count++;
    update_group_desc();
}

//在打开文件表中查找是否已打开文件
static unsigned short search_file(unsigned short Inode)
{
    unsigned short fopen_table_point=0;
    while(fopen_table_point<16&&fopen_table[fopen_table_point++]!=Inode);
    if(fopen_table_point==16)
    {
        return 0;
    }
    return 1;
}

void initialize_disk(void)  //初始化磁盘
{
```

```
int i=0;
printf("Creating the ext2 file system\n");
printf("Please wait ");
while(i<1)
{
    printf("... ");
    ++i;
}
printf("\n");
last_alloc_inode=1;
last_alloc_block=0;
for(i=0;i<16;i++)
{
    fopen_table[i]=0; //清空缓冲表
}
for(i=0;i<BLOCK_SIZE;i++)
{
    Buffer[i]=0; // 清空缓冲区
}
if(fp!=NULL)
{
    fclose(fp);
}
fp=fopen("./Ext2","w+"); //此文件大小是4612*512=2361344B，用此文件来模拟文件系统
fseek(fp,DISK_START,SEEK_SET);//将文件指针从0开始
for(i=0;i<DISK_SIZE;i++)
{
    fwrite(Buffer,BLOCK_SIZE,1,fp); // 清空文件，即清空磁盘全部用0填充 Buffer为缓冲区起始地址，B
}
// 初始化超级块内容
char username[10]; //用户名
    char password[10]; //密码
printf("Please set your username and password (less than 10 characters)\n");
printf("username: ");
scanf("%s",username);
printf("password: ");
scanf("%s",password);
reload_super_block();
strcpy(sb_block[0].sb_volume_name,VOLUME_NAME);
strcpy(sb_block[0].username,username);
strcpy(sb_block[0].password,password);
sb_block[0].sb_disk_size=DISK_SIZE;
sb_block[0].sb_blocks_per_group=BLOCKS_PER_GROUP;
```

```
sb_block[0].sb_size_per_block=BLOCK_SIZE;
update_super_block();
// 根目录的inode号为1
reload_inode_entry(1);

reload_dir(0);
char temp1[256] = "";
strcpy(temp1,sb_block[0].username);
strcat(temp1,"@Ubuntu:~");
strcpy(path_head, temp1);
strcpy(current_path, path_head); // 修改路径名为根目录
// 初始化组描述符内容
reload_group_desc();

gdt[0].bg_block_bitmap=BLOCK_BITMAP; //第一个块位图的起始地址
gdt[0].bg_inode_bitmap=INODE_BITMAP; //第一个inode位图的起始地址
gdt[0].bg_inode_table=INODE_TABLE;   //inode表的起始地址
gdt[0].bg_free_blocks_count=DATA_BLOCK_COUNTS; //空闲数据块数
gdt[0].bg_free_inodes_count=INODE_TABLE_COUNTS; //空闲inode数
gdt[0].bg_used_dirs_count=0; // 初始分配给目录的节点数是0
update_group_desc(); // 更新组描述符内容

reload_block_bitmap();
reload_inode_bitmap();

unsigned long temp_time=getCurrentTime();
inode_area[0].i_mode=518;
inode_area[0].i_blocks=0;
inode_area[0].i_size=32;
inode_area[0].i_atime=temp_time;
inode_area[0].i_ctime=temp_time;
inode_area[0].i_mtime=temp_time;
inode_area[0].i_dtime=0;
inode_area[0].i_block[0]=alloc_block(); //分配数据块
//printf("%d f\n",inode_area[0].i_block[0]);
inode_area[0].i_blocks++;
current_dir=get_inode();
update_inode_entry(current_dir);

//初始化根目录的目录项
dir[0].inode=dir[1].inode=current_dir;
dir[0].name_len=0;
dir[1].name_len=0;
```

```
    dir[0].file_type=dir[1].file_type=2;
    strcpy(dir[0].name,".");
    strcpy(dir[1].name,"..");
    update_dir(inode_area[0].i_block[0]);
    printf("The ext2 file system has been installed!\n");
    check_disk();
    fclose(fp);
}

//初始化内存
void initialize_memory(void)
{
    int i=0;
    last_alloc_inode=1;
    last_alloc_block=0;
    for(i=0;i<16;i++)
    {
        fopen_table[i]=0;
    }
    current_dir=1;
    fp=fopen("./Ext2","r+");
    if(fp==NULL)
    {
        printf("The File system does not exist!\n");
        initialize_disk();
        exit(0);
        return ;
    }
    reload_super_block();
    char temp1[256] = "";
    strcpy(temp1,sb_block[0].username);
    strcat(temp1,"@Ubuntu:~");
    strcpy(path_head, temp1);
    strcpy(current_path, path_head); // 修改路径名为根目录
    if(strcmp(sb_block[0].sb_volume_name,VOLUME_NAME))
    {
        printf("The File system [%s] is not suppoted yet!\n", sb_block[0].sb_volume_name);
        printf("The File system loaded error!\n");
        initialize_disk();
        return ;
    }
    reload_group_desc();
}
```

```
//格式化
void format(void)
{
    initialize_disk();
    initialize_memory();
}

//进入某个目录，实际上是改变当前路径
void cd(char tmp[9])
{
    //返回根目录
    if(!strcmp(tmp,"~"))
    {
        strcpy(current_path,path_head);
        current_dir=1;
        current_dirlen=0;
        return;
    }
    unsigned short i,j,k,flag;
    flag=reserch_file(tmp,2,&i,&j,&k);
    if(flag)
    {
        reload_inode_entry(i);
        if (inode_area[0].i_mode&1 != 1)
        {
            printf("You don't have permission to enter this directory!\n");
            return;
        }
        inode_area[0].i_atime=getCurrentTime();
        update_inode_entry(i);

        current_dir=i;
        if(!strcmp(tmp,"..")&&dir[k-1].name_len) /* 到上一级目录且不是..目录 */
        {
            current_path[strlen(current_path)-dir[k-1].name_len-1]='\0';
            current_dirlen=dir[k].name_len;
        }
        else if(!strcmp(tmp,"."))
        {
                return ;
        }
        else if(strcmp(tmp,"..")) // cd 到子目录
        {
            strcat(current_path,"/");
            current_dirlen=strlen(tmp);
            strcat(current_path,tmp);
        }
    }
    else
    {
        printf("The directory %s not exists!\n",tmp);
    }
}

// 创建目录
void mkdir(char tmp[9],int type)
{
    //printf("%s %d\n",tmp,type);
    unsigned short tmpno,i,j,k,flag;

    // 当前目录下新增目录或文件
    reload_inode_entry(current_dir);
    if(!reserch_file(tmp,type,&i,&j,&k)) // 未找到同名文件
    {
        if(inode_area[0].i_size==4096) // 目录项已满
        {
            printf("Directory has no room to be alloced!\n");
            return;
        }
        flag=1;
        if(inode_area[0].i_size!=inode_area[0].i_blocks*512) // 目录中有某些块中32个 dir_entry 未
        {
            i=0;
            while(flag&&i<inode_area[0].i_blocks)
            {
                reload_dir(inode_area[0].i_block[i]);
                j=0;
                while(j<32)
                {
                    if(dir[j].inode==0)
                    {
                        flag=0; //找到某个未装满目录项的块
                        break;
                    }
                    j++;
                }
                i++;
            }
            tmpno=dir[j].inode=get_inode();

            dir[j].name_len=strlen(tmp);
            dir[j].file_type=type;
            strcpy(dir[j].name,tmp);
            update_dir(inode_area[0].i_block[i-1]);
        }
        else // 全满 新增加块
        {
            inode_area[0].i_block[inode_area[0].i_blocks]=alloc_block();
            inode_area[0].i_blocks++;
            reload_dir(inode_area[0].i_block[inode_area[0].i_blocks-1]);
            tmpno=dir[0].inode=get_inode();
            dir[0].name_len=strlen(tmp);
            dir[0].file_type=type;
            strcpy(dir[0].name,tmp);
            // 初始化新块的其余目录项
            for(flag=1;flag<32;flag++)
            {
                dir[flag].inode=0;
            }
            update_dir(inode_area[0].i_block[inode_area[0].i_blocks-1]);
        }
        inode_area[0].i_size+=16;

        update_inode_entry(current_dir);

        // 为新增目录分配 dir_entry
        dir_prepare(tmpno,strlen(tmp),type);
    }
    else  // 已经存在同名文件或目录
    {
        printf("Directory has already existed!\n");
    }

}
//创建文件
void cat(char tmp[9],int type)
{
    unsigned short tmpno,i,j,k,flag;
```

```
reload_inode_entry(current_dir);
if(!reserch_file(tmp,type,&i,&j,&k))
{
    if(inode_area[0].i_size==4096)
    {
        printf("Directory has no room to be alloced!\n");
        return;
    }
    flag=1;
    if(inode_area[0].i_size!=inode_area[0].i_blocks*512)
    {
        i=0;
        while(flag&&i<inode_area[0].i_blocks)
        {
            reload_dir(inode_area[0].i_block[i]);
            j=0;
            while(j<32)
            {
                if(dir[j].inode==0)//找到了未分配的目录项
                {
                    flag=0;
                    break;
                }
                j++;
            }
            i++;
        }
        tmpno=dir[j].inode=get_inode();//分配一个新的inode项
        dir[j].name_len=strlen(tmp);
        dir[j].file_type=type;
        strcpy(dir[j].name,tmp);
        update_dir(inode_area[0].i_block[i-1]);
    }
    else //分配一个新的数据块
    {
        inode_area[0].i_block[inode_area[0].i_blocks]=alloc_block();
        inode_area[0].i_blocks++;
        reload_dir(inode_area[0].i_block[inode_area[0].i_blocks-1]);
        tmpno=dir[0].inode=get_inode();
        dir[0].name_len=strlen(tmp);
        dir[0].file_type=type;
        strcpy(dir[0].name,tmp);
        //初始化新快其他项目为0
```

```
            for(flag=1;flag<32;flag++)
            {
                dir[flag].inode=0;
            }
            update_dir(inode_area[0].i_block[inode_area[0].i_blocks-1]);
        }
        inode_area[0].i_size+=16;
        update_inode_entry(current_dir);
        //将新增文件的inode节点初始化
        dir_prepare(tmpno,strlen(tmp),type);

        //当文件无后缀或者后缀为.exe时，将文件的权限设置为可执行
        //判断是否为.exe文件
        int condition1 = strcmp(tmp + strlen(tmp) - 4, ".exe");
        //判断是否为无后缀文件，查找文件名中是否有.
        int condition2 = 1;
        for (int i = 0; i < strlen(tmp); i++) {
            if (tmp[i] == '.') {
                condition2 = 0;
                break;
            }
        }
        if (condition1 == 0 || condition2 == 1) {
            reload_inode_entry(tmpno);
            inode_area[0].i_mode = 0b0000001000000111;
            update_inode_entry(tmpno);
        }


    }
    else
    {
        printf("File has already existed!\n");
    }
}

//删除一个空目录
void rmdir(char tmp[9])
{
    unsigned short i,j,k,flag;
    unsigned short m,n;
    unsigned short temp = current_dir;
    if (!strcmp(tmp, "..") || !strcmp(tmp, "."))
```

```
{
    printf("The directory can not be deleted!\n");
    return;
}
flag=reserch_file(tmp,2,&i,&j,&k);
if (flag)
{
    reload_inode_entry(dir[k].inode); // 加载要删除的节点
    if(inode_area[0].i_size==32)  // 只有.and ..
    {
        inode_area[0].i_size=0;
        inode_area[0].i_blocks=0;

        remove_block(inode_area[0].i_block[0]);
        // 更新 tmp 所在父目录
        reload_inode_entry(current_dir);
        reload_dir(inode_area[0].i_block[j]);
        remove_inode(dir[k].inode);
        dir[k].inode=0;
        update_dir(inode_area[0].i_block[j]);
        inode_area[0].i_size-=16;
        flag=0;

        m=1;
        while(flag<32&&m<inode_area[0].i_blocks)
        {
            flag=n=0;
            reload_dir(inode_area[0].i_block[m]);
            while(n<32)
            {
                if(!dir[n].inode)
                {
                    flag++;
                }
                n++;
            }
            //如果删除过后，整个数据块的目录项全都为空。类似于在数组中删除某一个位置
            if(flag==32)
            {
                remove_block(inode_area[0].i_block[m]);
                inode_area[0].i_blocks--;
                while(m<inode_area[0].i_blocks)
                {
                    inode_area[0].i_block[m]=inode_area[0].i_block[m+1];
                    ++m;
                }
            }
        }
        update_inode_entry(current_dir);
        return;
    }
    else
    {
        for(int l=0;l<inode_area[0].i_blocks;l++)
        {
            reload_dir(inode_area[0].i_block[l]);
            for(int m=0;m<32;m++)
            {
                if(!strcmp(dir[m].name,".")||!strcmp(dir[m].name,"..")||dir[m].inode==0)
                    continue;
                if(dir[m].file_type==2)
                {
                    strcpy(current_path,tmp);
                    current_dir = i;
                    rmdir(dir[m].name);
                    current_dir = temp;
                }
                else if(dir[m].file_type==1)
                {
                    current_dir = i;
                    del(dir[m].name);
                    current_dir = temp;
                }
            }
            if(inode_area[0].i_size==32)
            {
                strcpy(current_path,path_head);
                current_dir=temp;
                rmdir(tmp);
            }
        }
        return;
            printf("Directory is not null!\n");
    }
}
else
```

```
    {
        printf("Directory to be deleted not exists!\n");
    }
}

//删除文件
void del(char tmp[9])
{
    unsigned short i,j,k,m,n,flag;
    m=0;
    flag=reserch_file(tmp,1,&i,&j,&k);
    if(flag)
    {
        //删除文件需要父目录的写权限
        reload_inode_entry(current_dir);
        if (inode_area[0].i_mode&2 != 2)
        {
            printf("You don't have permission to delete this file!\n");
            return;
        }
        flag = 0;
        // 若文件 tmp 已打开，则将对应的 fopen_table 项清0
        while(fopen_table[flag]!=dir[k].inode&&flag<16)
        {
                flag++;
        }
        if(flag<16)
        {
                fopen_table[flag]=0;
        }
        reload_inode_entry(i); // 加载删除文件 inode
        //删除文件对应的数据块
        while(m<inode_area[0].i_blocks)
        {
                remove_block(inode_area[0].i_block[m++]);
        }
        inode_area[0].i_blocks=0;
        inode_area[0].i_size=0;
        remove_inode(i);
        // 更新父目录
        reload_inode_entry(current_dir);
        reload_dir(inode_area[0].i_block[j]);
        dir[k].inode=0; //删除inode节点
        update_dir(inode_area[0].i_block[j]);
        inode_area[0].i_size-=16;
        m=1;
        //删除一项后整个数据块为空，则将该数据块删除
        while(m<inode_area[i].i_blocks)
        {
            flag=n=0;
            reload_dir(inode_area[0].i_block[m]);
            while(n<32)
            {
                if(!dir[n].inode)
                {
                        flag++;
                }
                n++;
            }
            if(flag==32)
            {
                remove_block(inode_area[i].i_block[m]);
                inode_area[i].i_blocks--;
                while(m<inode_area[i].i_blocks)
                {
                        inode_area[i].i_block[m]=inode_area[i].i_block[m+1];
                        ++m;
                }
            }
        }
        update_inode_entry(current_dir);
    }
    else
    {
        printf("The file %s not exists!\n",tmp);
    }
}

//打开文件
void open_file(char tmp[9])
{
    unsigned short flag,i,j,k;
    flag=reserch_file(tmp,1,&i,&j,&k);
    reload_inode_entry(i);
    inode_area[0].i_atime=getCurrentTime();
```

```c
    update_inode_entry(i);
    if(flag)
    {
        if(search_file(dir[k].inode))
        {
                printf("The file %s has opened!\n",tmp);
        }
        else
        {
            flag=0;
            while(fopen_table[flag])
            {
                flag++;
            }
            fopen_table[flag]=dir[k].inode;
            printf("File %s opened!\n",tmp);
        }
    }
    else printf("The file %s does not exist!\n",tmp);
}

//关闭文件
void close_file(char tmp[9])
{
    unsigned short flag,i,j,k;
    flag=reserch_file(tmp,1,&i,&j,&k);

    if(flag)
    {
        if(search_file(dir[k].inode))
        {
            flag=0;
            while(fopen_table[flag]!=dir[k].inode)
            {
                flag++;
            }
            fopen_table[flag]=0;
            printf("File %s closed!\n",tmp);
        }
        else
        {
                printf("The file %s has not been opened!\n",tmp);
        }
    }
    else
    {
        printf("The file %s does not exist!\n",tmp);
    }
}

// 读文件
void read_file(char tmp[9])
{
    unsigned short flag,i,j,k,t;
    unsigned short b1,b2,b3;
    b1=b2=b3=0;
    flag=reserch_file(tmp,1,&i,&j,&k);
    if(flag)
    {
        if(search_file(dir[k].inode)) //读文件的前提是该文件已经打开
        {
            reload_inode_entry(dir[k].inode);
            //判断是否有读的权限
            if(!(inode_area[0].i_mode&4)) // i_mode:111b:读,写,执行
            {
                printf("The file %s can not be read!\n",tmp);
                return;
            }
            //输出直接索引的内容
            if (inode_area[0].i_blocks<=6){
                b1=inode_area[0].i_blocks;
            }else if (inode_area[0].i_blocks>6){
                b1=6;
            }
            if (b1>0){
                for(flag=0;flag<b1;flag++)
                {
                    reload_block(inode_area[0].i_block[flag]);
                    for(t=0;t<inode_area[0].i_size-flag*512;++t)
                    {
                        printf("%c",Buffer[t]);
                    }
                }
            }

            //输出一级索引的内容
```

```
if(inode_area[0].i_blocks>6&&inode_area[0].i_blocks<=262)
{
    b2=inode_area[0].i_blocks-6;
}else if (inode_area[0].i_blocks>262){
    b2=256;
}
if (b2>0){
    reload_block(inode_area[0].i_block[6]);
    char index_1[512];
    memcpy(index_1,Buffer,512);
    for (flag = 0; flag < b2; flag++)
    {
        unsigned short block_num_1;
        memcpy(&block_num_1, &index_1[flag * 2], sizeof(unsigned short));
        reload_block(block_num_1);
        for (t = 0; t < inode_area[0].i_size - (flag + 6) * 512; ++t)
        {
            printf("%c", Buffer[t]);
        }
    }
}

//输出二级索引的内容
if(inode_area[0].i_blocks>262&&inode_area[0].i_blocks<=65818)
{
    b3=inode_area[0].i_blocks-262;
}

if (b3>0){
    reload_block(inode_area[0].i_block[7]);
    char index_2[512];
    memcpy(index_2,Buffer,512);
    for (flag = 0; flag < b3; flag++)
    {
        unsigned short block_num_2;
        memcpy(&block_num_2, &index_2[flag * 2], sizeof(unsigned short));
        reload_block(block_num_2);
        char index_3[512];
        memcpy(index_3,Buffer,512);
        for (t = 0; t < 256; t++)
        {
            unsigned short block_num_3;
            memcpy(&block_num_3, &index_3[t * 2], sizeof(unsigned short));
```

```
                        reload_block(block_num_3);
                        for (int m = 0; m < inode_area[0].i_size - (flag + 262) * 512; ++m)
                        {
                            printf("%c", Buffer[m]);
                        }
                    }
                }
            }


            if(inode_area[0].i_blocks==0)
            {
                printf("The file %s is empty!\n",tmp);
            }
            else
            {
                printf("\n");
            }
        }
        else
        {
                printf("The file %s has not been opened!\n",tmp);
        }
    }
    else printf("The file %s not exists!\n",tmp);
}

void write_file(char tmp[9]) // 写文件
{
    unsigned short flag,i,j,k,size=0,need_blocks,length;
    flag=reserch_file(tmp,1,&i,&j,&k);
    if(flag){
        reload_inode_entry(i);
        if ((inode_area[0].i_mode & 2) == 0)
        {
            printf("You don't have permission to write this file!\n");
            return;
        }else{
            write_file_111(tmp);
        }
    }
}
```

```
//文件以覆盖方式写入
void write_file_(char tmp[9]) // 写文件
{
    unsigned short flag,i,j,k,size=0,need_blocks,length;
    flag=reserch_file(tmp,1,&i,&j,&k);
    if (flag)
    {
        reload_inode_entry(i);
        inode_area[0].i_mtime=getCurrentTime();
        update_inode_entry(i);
        if(search_file(dir[k].inode))
        {
            reload_inode_entry(dir[k].inode);
            while(1)
            {
                tempbuf[size]=getchar();
                if(tempbuf[size]=='#')
                {
                    tempbuf[size]='\0';
                    break;
                }
                if(size>=4095)
                {
                    printf("Sorry,the max size of a file is 4KB!\n");
                    break;
                }
                size++;
            }
            if(size>=4095)
            {
                length=4096;
            }
            else
            {
                length=strlen(tempbuf);
            }
            //计算需要的数据块数目
            need_blocks=length/512;
            if(length%512)
            {
                need_blocks++;
            }
            if(need_blocks<9) // 文件最大 8 个 blocks(512 bytes)
```

```
{
    // 分配文件所需块数目
    //因为以覆盖写的方式写，要先判断原有的数据块数目
    if(inode_area[0].i_blocks<=need_blocks)
    {
        while(inode_area[0].i_blocks<need_blocks)
        {
            inode_area[0].i_block[inode_area[0].i_blocks]=alloc_block();
            inode_area[0].i_blocks++;
        }
    }
    else
    {
        while(inode_area[0].i_blocks>need_blocks)
        {
            remove_block(inode_area[0].i_block[inode_area[0].i_blocks - 1]);
            inode_area[0].i_blocks--;
        }
    }
    j=0;
    while(j<need_blocks)
    {
        if(j!=need_blocks-1)
        {
            reload_block(inode_area[0].i_block[j]);
            memcpy(Buffer,tempbuf+j*BLOCK_SIZE,BLOCK_SIZE);
            update_block(inode_area[0].i_block[j]);
        }
        else
        {
            reload_block(inode_area[0].i_block[j]);
            memcpy(Buffer,tempbuf+j*BLOCK_SIZE,length-j*BLOCK_SIZE);
            inode_area[0].i_size=length;
            update_block(inode_area[0].i_block[j]);
        }
        j++;
    }
    update_inode_entry(dir[k].inode);
}
else
{
    printf("Sorry,the max size of a file is 4KB!\n");
}
```

```
        }
        else
        {
                printf("The file %s has not opened!\n",tmp);
        }
    }
    else
    {
        printf("The file %s does not exist!\n",tmp);
    }
}

void write_file_111(char tem[9])
{
    fflush(stdin);
    unsigned short flag,i,j,k,size=0,need_blocks;
    int length1,length2,length3;//1级写入，2级写入，3级写入
    length1=length2=length3=0;
    flag=reserch_file(tem,1,&i,&j,&k);
    if (flag){
        reload_inode_entry(i);
        inode_area[0].i_mtime=getCurrentTime();
        update_inode_entry(i);
        reload_group_desc();
        if(search_file(dir[k].inode)){
            reload_inode_entry(dir[k].inode);
            while(1)
            {
                tempbuf[size]=getchar();
                if(tempbuf[size]=='#')
                {
                    tempbuf[size]='\0';
                    break;
                }
                if(size>=gdt->bg_free_blocks_count*512)//判断文件大小是否超过最大值，超过全部不写入
                {
                    printf("Sorry,the max size of a file is %dKB!\n",gdt->bg_free_blocks_count/
                    printf("Write failed!\n");
                    return;
                }
                size++;
            }
            if (size<=6*512){
```

```
        length1 = strlen(tempbuf);
    }else if (size<=12*512){
        length1 = 6*512;
    }
    if(length1>0){
        need_blocks=length1/512;
        if(length1%512)
        {
            need_blocks++;
        }
        int x = 0;
        while (x <= need_blocks)
        {
            inode_area[0].i_block[x]=alloc_block();
            reload_block(inode_area[0].i_block[x]);
            memcpy(Buffer,tempbuf+x*BLOCK_SIZE,BLOCK_SIZE);
            update_block(inode_area[0].i_block[x]);
            x++;
        }
        if(inode_area[0].i_blocks>need_blocks)//清空剩下的块
        {
            while (x <= 6)
            {
                x++;
                remove_block(inode_area[0].i_block[x]);
            }
        }
    }
    if (size > 6*512 && size <= 262*512){
        length2 = size - 6*512;
    }else if (size > 262*512){
        length2 = 256*512;
    }
    if (length2){//采用一级索引
        need_blocks=length2/512;
        if(length2%512)
        {
            need_blocks++;
        }
        inode_area[0].i_block[6]=alloc_block();
        reload_block(inode_area[0].i_block[6]);
        char index_1[512];          //一级索引
        memcpy(index_1,Buffer,512);
        int x;
        for (x = 0; x < need_blocks; x++) // x代表写入的块数
        {
            unsigned short block_index_temp = alloc_block();
            unsigned short* target = (unsigned short*)(index_1 + sizeof(unsigned short)
            memcpy(target, &block_index_temp, sizeof(unsigned short));
            reload_block(block_index_temp);
            memcpy(Buffer,tempbuf+(x+6)*BLOCK_SIZE,BLOCK_SIZE);
            update_block(block_index_temp);
        }
        if(inode_area[0].i_blocks-6>need_blocks)//清空剩下的块
        {
            while(x<=255){
                unsigned short temp;
                memcpy(&temp, &index_1[x * 2], sizeof(unsigned short));
                remove_block(temp);
                x++;
            }
        }
        memcpy(Buffer,index_1,512);
        update_block(inode_area[0].i_block[6]);//更新一级索引
    }
    if (size > 262*512 ){
        length3 = size - 262*512;
    }
    if (length3 >0){
        need_blocks=length3/512;
        if(length3%512)
        {
            need_blocks++;
        }
        inode_area[0].i_block[7]=alloc_block();
        unsigned short num_1 = need_blocks/256;
        unsigned short num_2 = need_blocks%256;
        reload_block(inode_area[0].i_block[7]);
        char index_2[512];          //二级索引
        memcpy(index_2,Buffer,512);
        int x;
        for (x = 0; x < num_1; x++) // x代表写入的块数
        {
            unsigned short block_index_temp = alloc_block();
            unsigned short* target = (unsigned short*)(index_2 + sizeof(unsigned short)
            memcpy(target,&block_index_temp,sizeof(unsigned short));
```

```
                    reload_block(block_index_temp);
                    char index_1[512];          //一级索引
                    memcpy(index_1,Buffer,512);
                    int y;
                    for (y = 0; y < 256; y++) // y代表写入的块数
                    {
                        unsigned short block_index_temp = alloc_block();
                        unsigned short* target = (unsigned short*)(index_1 + sizeof(unsigned sho
                        memcpy(target,&block_index_temp,sizeof(unsigned short));
                        reload_block(block_index_temp);
                        memcpy(Buffer,tempbuf+(x*256+y+262)*BLOCK_SIZE,BLOCK_SIZE);
                        update_block(block_index_temp);
                    }
                    memcpy(Buffer,index_1,512);
                    update_block(block_index_temp);
                }
                memcpy(Buffer,index_2,512);
                update_block(inode_area[0].i_block[7]);//更新二级索引
            }
            inode_area[0].i_size=size;
            inode_area[0].i_blocks= size%512==0?size/512:size/512+1;
            update_inode_entry(dir[k].inode);
        }else {
            printf("The file %s has not opened!\n",tem);
        }
    }else{
        printf("The file %s does not exist!\n",tem);
    }
}




//查看目录下的内容
void ls(void)
{
    printf("items          type          mode          size                 Access "
        "time                 Creation ctime          Modification mtime\n");
    unsigned short i,j,k,flag;
    i=0;
    reload_inode_entry(current_dir);
    while(i<inode_area[0].i_blocks)
    {
```

```
k=0;
reload_dir(inode_area[0].i_block[i]);
while(k<32)
{
    if(dir[k].inode)
    {
        printf("%s",dir[k].name);
        if(dir[k].file_type==2)
        {
            j=0;
            reload_inode_entry(dir[k].inode);
            if(!strcmp(dir[k].name,".."))
            {
                while(j++<13)
                {
                        printf(" ");
                }
                flag=1;
            }
            else if(!strcmp(dir[k].name,"."))
            {
                while(j++<14)
                {
                        printf(" ");
                }
                flag=0;
            }
            else
            {
                while(j++<15-dir[k].name_len)
                {
                        printf(" ");
                }
                flag=2;
            }
            printf("<DIR>          ");
            switch(inode_area[0].i_mode&7)
            {
                case 1:printf("____x");break;
                case 2:printf("__w__");break;
                case 3:printf("__w_x");break;
                case 4:printf("r____");break;
                case 5:printf("r___x");break;
                case 6:printf("r_w__");break;
                case 7:printf("r_w_x");break;
            }
            if(flag!=2)
            {
                printf("          ----");
            }
            else
            {
                printf("          ");
                printf("%4d",inode_area[0].i_size);
            }
            printf("%29s",convertTimeStampToString(inode_area[0].i_atime));
            printf("%29s",convertTimeStampToString(inode_area[0].i_ctime));
            printf("%29s",convertTimeStampToString(inode_area[0].i_mtime));

        }
        else if(dir[k].file_type==1)
        {
            j=0;
            reload_inode_entry(dir[k].inode);
            while(j++<15-dir[k].name_len)printf(" ");
            printf("<FILE>         ");
            switch(inode_area[0].i_mode&7)
            {
                case 1:printf("____x");break;
                case 2:printf("__w__");break;
                case 3:printf("__w_x");break;
                case 4:printf("r____");break;
                case 5:printf("r___x");break;
                case 6:printf("r_w__");break;
                case 7:printf("r_w_x");break;
            }
            printf("          ");
            printf("%4d",inode_area[0].i_size);
            printf("%29s",convertTimeStampToString(inode_area[0].i_atime));
            printf("%29s",convertTimeStampToString(inode_area[0].i_ctime));
            printf("%29s",convertTimeStampToString(inode_area[0].i_mtime));
        }
        printf("\n");
    }
    k++;
    reload_inode_entry(current_dir);
```

```c
        }
        i++;
    }
}

//检查磁盘状态
void check_disk(void)
{
        reload_super_block();
        printf("volume name       : %s\n", sb_block[0].sb_volume_name);
        printf("disk size         : %d(blocks)\n", sb_block[0].sb_disk_size);
        printf("blocks per group  : %d(blocks)\n", sb_block[0].sb_blocks_per_group);
        printf("ext2 file size    : %d(kb)\n", sb_block[0].sb_disk_size*sb_block[0].sb_size_per_
        printf("block size        : %d(kb)\n", sb_block[0].sb_size_per_block);
}


int ext2_load()
{
    int times=5;
    printf("Welcome to ext2 file system!\n");
    while (1)
    {
        char temp_username[10],temp_password[10];
        printf("Username:");
        fflush(stdout);
        scanf("%s",temp_username);
        printf("Password:");
        fflush(stdout);
        scanf("%s",temp_password);
        if(strcmp(temp_username,sb_block[0].username)==0&&strcmp(temp_password,sb_block[0].passw
            printf("Login successfully!\n");
            if (strcmp(temp_username,USER_NAME)==0&&strcmp(temp_password,PASSWORD)==0){
                printf("You are using the default username and password, please reset your usern
                reset_password();
            }
            break;
        }
        else{
            printf("Username or password is wrong!\n");
            printf("You have %d times to try!\n",times);
            times--;
            if(times==0){
                printf("You have tried too many times!\n");
                return 0;
            }
        }
    }
    return 1;
}

int reset_password(void)
{
    printf("Please input the new username:");
    fflush(stdout);
    scanf("%s",sb_block[0].username);
    printf("Please input the new password:");
    fflush(stdout);
    scanf("%s",sb_block[0].password);
    printf("Reset password successfully!\n");
    update_super_block();
    exit(0);
    return 1;
}

void help(){
    printf("==============================================\n");
    printf("%-8s: %s\n", "format", "format the disk");
    printf("%-8s: %s\n", "mkdir", "create a directory");
    printf("%-8s: %s\n", "rmdir", "remove a directory");
    printf("%-8s: %s\n", "cd", "change the current directory");
    printf("%-8s: %s\n", "ls", "list the files in the current directory");
    printf("%-8s: %s\n", "touch", "create a file");
    printf("%-8s: %s\n", "rm", "delete a file");
    printf("%-8s: %s\n", "open", "open a file");
    printf("%-8s: %s\n", "close", "close a file");
    printf("%-8s: %s\n", "read", "read a file");
    printf("%-8s: %s\n", "write", "write a file");
    printf("%-8s: %s\n", "chmod", "change the mode of a file");
    printf("%-8s: %s\n", "help", "show the help information");
    printf("%-8s: %s\n", "quit", "exit the file system");
    printf("==============================================\n");
}


//修改文件的权限
```

```c
void chmod(char tmp[9],char mod[4])
{
    unsigned short flag,i,j,k,t;
    flag=reserch_file(tmp,1,&i,&j,&k);
    reload_inode_entry(i);
    if(flag){
        unsigned short tt = 0b1111111100000000;
        if (!strcmp(mod, "r")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 4;
        } else if (!strcmp(mod, "w")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 2;
        } else if (!strcmp(mod, "x")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 1;
        } else if (!strcmp(mod, "rw")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 6;
        } else if (!strcmp(mod, "rx")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 5;
        } else if (!strcmp(mod, "wx")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 3;
        } else if (!strcmp(mod, "rwx")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 7;
        } else {
            printf("The mode is wrong!\n");
            return;
        }
        update_inode_entry(i);
    }
}
```