代码注释

src

File.h

```
#ifndef FILE_H
#define FILE_H

#include "INode.h"

/*
 * 打开文件控制块File类。
 * 该结构记录了进程打开文件
 * 的读、写请求类型，文件读写位置等等。
 */
class File
{
public:
    /* Enumerate */
    enum FileFlags
    {
        FREAD = 0x1,  /* 读请求类型 */
        FWRITE = 0x2, /* 写请求类型 */
        FPIPE = 0x4   /* 管道类型 */
    };

    /* Functions */
public:
    /* Constructors */
    File();
    /* Destructors */
    ~File();

    /* Member */
    unsigned int f_flag; /* 对打开文件的读、写操作要求 */
    int f_count;         /* 当前引用该文件控制块的进程数量 */
    Inode *f_inode;      /* 指向打开文件的内存Inode指针 */
    int f_offset;        /* 文件读写位置指针 */
};

/*
 * 进程打开文件描述符表(OpenFiles)的定义
 * 进程的u结构中包含OpenFiles类的一个对象，
 * 维护了当前进程的所有打开文件。
 */
class OpenFiles
{
    /* static members */
public:
    static const int NOFILES = 15; /* 进程允许打开的最大文件数 */
```

```
    /* Functions */
public:
    /* Constructors */
    OpenFiles();
    /* Destructors */
    ~OpenFiles();

    /*
     * @comment 进程请求打开文件时，在打开文件描述符表中分配一个空闲表项
     */
    int AllocFreeSlot();

    /*
     * @comment Dup系统调用时复制打开文件描述符表中的描述符
     */
    int Clone(int fd);

    /*
     * @comment 根据用户系统调用提供的文件描述符参数fd，
     * 找到对应的打开文件控制块File结构
     */
    File *GetF(int fd);
    /*
     * @comment 为已分配到的空闲描述符fd和已分配的打开文件表中
     * 空闲File对象建立勾连关系
     */
    void SetF(int fd, File *pFile);

    /* Members */
private:
    File *ProcessOpenFileTable[NOFILES]; /* File对象的指针数组，指向系统打开文件表中的
File对象 */
};

/*
 * 文件I/O的参数类
 * 对文件读、写时需用到的读、写偏移量、
 * 字节数以及目标区域首地址参数。
 */
class IOParameter
{
    /* Functions */
public:
    /* Constructors */
    IOParameter();
    /* Destructors */
    ~IOParameter();

    /* Members */
public:
    unsigned char *m_Base; /* 当前读、写用户目标区域的首地址 */
    int m_Offset;          /* 当前读、写文件的字节偏移量 */
    int m_Count;           /* 当前还剩余的读、写字节数量 */
};

#endif
```

File.cpp

```
#include "File.h"
#include "Utility.h" //for NULL
#include "Kernel.h"

/*==============================class File==================================*/
File::File()
{
    this->f_count = 0;
    this->f_flag = 0;
    this->f_offset = 0;
    this->f_inode = NULL;
}

File::~File()
{
    // nothing to do here
}

/*==============================class
OpenFiles================================*/
OpenFiles::OpenFiles()
{
}

OpenFiles::~OpenFiles()
{
}

int OpenFiles::AllocFreeSlot()
{
    int i;
    User &u = Kernel::Instance().GetUser();

    for (i = 0; i < OpenFiles::NOFILES; i++)
    {
        /* 进程打开文件描述符表中找到空闲项，则返回之 */
        if (this->ProcessOpenFileTable[i] == NULL)
        {
            /* 设置核心栈现场保护区中的EAX寄存器的值，即系统调用返回值 */
            u.u_ar0[User::EAX] = i;
            return i;
        }
    }

    u.u_ar0[User::EAX] = -1; /* Open1，需要一个标志。当打开文件结构创建失败时，可以回收系
统资源*/
    u.u_error = User::EMFILE;
    return -1;
}

int OpenFiles::Clone(int fd)
{
```

```
    return 0;
}

File *OpenFiles::GetF(int fd)
{
    File *pFile;
    User &u = Kernel::Instance().GetUser();

    /* 如果打开文件描述符的值超出了范围 */
    if (fd < 0 || fd >= OpenFiles::NOFILES)
    {
        u.u_error = User::EBADF;
        return NULL;
    }

    pFile = this->ProcessOpenFileTable[fd];
    if (pFile == NULL)
    {
        u.u_error = User::EBADF;
    }

    return pFile; /* 即使pFile==NULL也返回它，由调用GetF的函数来判断返回值 */
}

void OpenFiles::SetF(int fd, File *pFile)
{
    if (fd < 0 || fd >= OpenFiles::NOFILES)
    {
        return;
    }
    /* 进程打开文件描述符指向系统打开文件表中相应的File结构 */
    this->ProcessOpenFileTable[fd] = pFile;
}

/*==============================class
IOParameter====================================*/
IOParameter::IOParameter()
{
    this->m_Base = 0;
    this->m_Count = 0;
    this->m_Offset = 0;
}

IOParameter::~IOParameter()
{
    // nothing to do here
}
```

FileManager.h

```
#ifndef FILE_MANAGER_H
#define FILE_MANAGER_H

#include "FileSystem.h"
```

```
#include "OpenFileManager.h"
#include "File.h"

/*
 * 文件管理类(FileManager)
 * 封装了文件系统的各种系统调用在核心态下处理过程，
 * 如对文件的Open()、Close()、Read()、Write()等等
 * 封装了对文件系统访问的具体细节。
 */
class FileManager
{
public:
    /* 目录搜索模式，用于NameI()函数 */
    enum DirectorySearchMode
    {
        OPEN = 0,   /* 以打开文件方式搜索目录 */
        CREATE = 1, /* 以新建文件方式搜索目录 */
        DELETE = 2  /* 以删除文件方式搜索目录 */
    };

    /* Functions */
public:
    /* Constructors */
    FileManager();
    /* Destructors */
    ~FileManager();

    /*
     * @comment 初始化对全局对象的引用
     */
    void Initialize();

    /*
     * @comment Open()系统调用处理过程
     */
    void Open();

    /*
     * @comment Creat()系统调用处理过程
     */
    void Creat();

    /*
     * @comment Open()、Creat()系统调用的公共部分
     */
    void Open1(Inode *pInode, int mode, int trf);

    /*
     * @comment Close()系统调用处理过程
     */
    void Close();

    /*
     * @comment Seek()系统调用处理过程
     */
    void Seek();

    /*
```

```
     * @comment Dup()复制进程打开文件描述符
     */
    void Dup();

    /*
     * @comment FStat()获取文件信息
     */
    void FStat();

    /*
     * @comment FStat()获取文件信息
     */
    void Stat();

    /* FStat()和Stat()系统调用的共享例程 */
    void Stat1(Inode *pInode, unsigned long statBuf);

    /*
     * @comment Read()系统调用处理过程
     */
    void Read();

    /*
     * @comment Write()系统调用处理过程
     */
    void Write();

    /*
     * @comment 读写系统调用公共部分代码
     */
    void Rdwr(enum File::FileFlags mode);

    /*
     * @comment Pipe()管道建立系统调用处理过程
     */
    void Pipe();

    /*
     * @comment 管道读操作
     */
    void ReadP(File *pFile);

    /*
     * @comment 管道写操作
     */
    void WriteP(File *pFile);

    /*
     * @comment 目录搜索，将路径转化为相应的Inode，
     * 返回上锁后的Inode
     */
    Inode *NameI(char (*func)(), enum DirectorySearchMode mode);

    /*
     * @comment 获取路径中的下一个字符
     */
    static char NextChar();
```

```
    /*
     * @comment 被Creat()系统调用使用，用于为创建新文件分配内核资源
     */
    Inode *MakNode(unsigned int mode);

    /*
     * @comment 向父目录的目录文件写入一个目录项
     */
    void WriteDir(Inode *pInode);

    /*
     * @comment 设置当前工作路径
     */
    void SetCurDir(char *pathname);

    /*
     * @comment 检查对文件或目录的搜索、访问权限，作为系统调用的辅助函数
     */
    int Access(Inode *pInode, unsigned int mode);

    /*
     * @comment 检查文件是否属于当前用户
     */
    Inode *Owner();

    /* 改变文件访问模式 */
    void ChMod();

    /* 改变文件文件所有者user ID及其group ID */
    void ChOwn();

    /* 改变当前工作目录 */
    void ChDir();

    /* 创建文件的异名引用 */
    void Link();

    /* 取消文件 */
    void UnLink();

    /* 用于建立特殊设备文件的系统调用 */
    void MkNod();

public:
    /* 根目录内存Inode */
    Inode *rootDirInode;

    /* 对全局对象g_FileSystem的引用，该对象负责管理文件系统存储资源 */
    FileSystem *m_FileSystem;

    /* 对全局对象g_InodeTable的引用，该对象负责内存Inode表的管理 */
    InodeTable *m_InodeTable;

    /* 对全局对象g_OpenFileTable的引用，该对象负责打开文件表项的管理 */
    OpenFileTable *m_OpenFileTable;
};

class DirectoryEntry
```

```
{
    /* static members */
public:
    static const int DIRSIZ = 28; /* 目录项中路径部分的最大字符串长度 */

    /* Functions */
public:
    /* Constructors */
    DirectoryEntry();
    /* Destructors */
    ~DirectoryEntry();

    /* Members */
public:
    int m_ino;           /* 目录项中Inode编号部分 */
    char m_name[DIRSIZ]; /* 目录项中路径名部分 */
};

#endif
```

FileManager.cpp

```
#include "FileManager.h"
#include "Kernel.h"
#include "Utility.h"
#include "TimeInterrupt.h"

/*==========================class FileManager===============================*/
FileManager::FileManager()
{
    // nothing to do here
}

FileManager::~FileManager()
{
    // nothing to do here
}

void FileManager::Initialize()
{
    this->m_FileSystem = &Kernel::Instance().GetFileSystem();

    this->m_InodeTable = &g_InodeTable;
    this->m_OpenFileTable = &g_OpenFileTable;

    this->m_InodeTable->Initialize();
}

/*
 * 功能：打开文件
 * 效果：建立打开文件结构，内存i节点开锁 、i_count 为正数（i_count ++）
 * */
void FileManager::Open()
{
    Inode *pInode;
```

```
    User &u = Kernel::Instance().GetUser();

    pInode = this->NameI(NextChar, FileManager::OPEN); /* 0 = Open, not create
*/
    /* 没有找到相应的Inode */
    if (NULL == pInode)
    {
        return;
    }
    this->Open1(pInode, u.u_arg[1], 0);
}

/*
 * 功能：创建一个新的文件
 * 效果：建立打开文件结构，内存i节点开锁 、i_count 为正数（应该是 1）
 * */
void FileManager::Creat()
{
    Inode *pInode;
    User &u = Kernel::Instance().GetUser();
    unsigned int newACCMode = u.u_arg[1] & (Inode::IRWXU | Inode::IRWXG |
Inode::IRWXO);

    /* 搜索目录的模式为1，表示创建；若父目录不可写，出错返回 */
    pInode = this->NameI(NextChar, FileManager::CREATE);
    /* 没有找到相应的Inode，或NameI出错 */
    if (NULL == pInode)
    {
        if (u.u_error)
            return;
        /* 创建Inode */
        pInode = this->MakNode(newACCMode & (~Inode::ISVTX));
        /* 创建失败 */
        if (NULL == pInode)
        {
            return;
        }

        /*
         * 如果所希望的名字不存在，使用参数trf = 2来调用open1()。
         * 不需要进行权限检查，因为刚刚建立的文件的权限和传入参数mode
         * 所表示的权限内容是一样的。
         */
        this->Open1(pInode, File::FWRITE, 2);
    }
    else
    {
        /* 如果NameI()搜索到已经存在要创建的文件，则清空该文件（用算法ITrunc()）。UID没有改
变
         * 原来UNIX的设计是这样：文件看上去就像新建的文件一样。然而，新文件所有者和许可权方式
没变。
         * 也就是说creat指定的RWX比特无效。
         * 邓蓉认为这是不合理的，应该改变。
         * 现在的实现：creat指定的RWX比特有效 */
        this->Open1(pInode, File::FWRITE, 1);
        pInode->i_mode |= newACCMode;
    }
}
```

```
/*
 * trf == 0由open调用
 * trf == 1由creat调用，creat文件的时候搜索到同文件名的文件
 * trf == 2由creat调用，creat文件的时候未搜索到同文件名的文件，这是文件创建时更一般的情况
 * mode参数：打开文件模式，表示文件操作是 读、写还是读写
 */
void FileManager::Open1(Inode *pInode, int mode, int trf)
{
    User &u = Kernel::Instance().GetUser();

    /*
     * 对所希望的文件已存在的情况下，即trf == 0或trf == 1进行权限检查
     * 如果所希望的名字不存在，即trf == 2，不需要进行权限检查，因为刚建立
     * 的文件的权限和传入的参数mode的所表示的权限内容是一样的。
     */
    if (trf != 2)
    {
        if (mode & File::FREAD)
        {
            /* 检查读权限 */
            this->Access(pInode, Inode::IREAD);
        }
        if (mode & File::FWRITE)
        {
            /* 检查写权限 */
            this->Access(pInode, Inode::IWRITE);
            /* 系统调用去写目录文件是不允许的 */
            if ((pInode->i_mode & Inode::IFMT) == Inode::IFDIR)
            {
                u.u_error = User::EISDIR;
            }
        }
    }

    if (u.u_error)
    {
        this->m_InodeTable->IPut(pInode);
        return;
    }

    /* 在creat文件的时候搜索到同文件名的文件，释放该文件所占据的所有盘块 */
    if (1 == trf)
    {
        pInode->ITrunc();
    }

    /* 解锁inode!
     * 线性目录搜索涉及大量的磁盘读写操作，期间进程会入睡。
     * 因此，进程必须上锁操作涉及的i节点。这就是NameI中执行的IGet上锁操作。
     * 行至此，后续不再有可能会引起进程切换的操作，可以解锁i节点。
     */
    pInode->Prele();

    /* 分配打开文件控制块File结构 */
    File *pFile = this->m_OpenFileTable->FAlloc();
    if (NULL == pFile)
    {
```

```
        this->m_InodeTable->IPut(pInode);
        return;
    }
    /* 设置打开文件方式，建立File结构和内存Inode的勾连关系 */
    pFile->f_flag = mode & (File::FREAD | File::FWRITE);
    pFile->f_inode = pInode;

    /* 特殊设备打开函数 */
    pInode->OpenI(mode & File::FWRITE);

    /* 为打开或者创建文件的各种资源都已成功分配，函数返回 */
    if (u.u_error == 0)
    {
        return;
    }
    else /* 如果出错则释放资源 */
    {
        /* 释放打开文件描述符 */
        int fd = u.u_ar0[User::EAX];
        if (fd != -1)
        {
            u.u_ofiles.SetF(fd, NULL);
            /* 递减File结构和Inode的引用计数 ,File结构没有锁 f_count为0就是释放File结构
了*/
            pFile->f_count--;
        }
        this->m_InodeTable->IPut(pInode);
    }
}

void FileManager::Close()
{
    User &u = Kernel::Instance().GetUser();
    int fd = u.u_arg[0];

    /* 获取打开文件控制块File结构 */
    File *pFile = u.u_ofiles.GetF(fd);
    if (NULL == pFile)
    {
        return;
    }

    /* 释放打开文件描述符fd，递减File结构引用计数 */
    u.u_ofiles.SetF(fd, NULL);
    this->m_OpenFileTable->CloseF(pFile);
}

void FileManager::Seek()
{
    File *pFile;
    User &u = Kernel::Instance().GetUser();
    int fd = u.u_arg[0];

    pFile = u.u_ofiles.GetF(fd);
    if (NULL == pFile)
    {
        return; /* 若FILE不存在，GetF有设出错码 */
    }
```

```
    /* 管道文件不允许seek */
    if (pFile->f_flag & File::FPIPE)
    {
        u.u_error = User::ESPIPE;
        return;
    }

    int offset = u.u_arg[1];

    /* 如果u.u_arg[2]在3 ~ 5之间，那么长度单位由字节变为512字节 */
    if (u.u_arg[2] > 2)
    {
        offset = offset << 9;
        u.u_arg[2] -= 3;
    }

    switch (u.u_arg[2])
    {
    /* 读写位置设置为offset */
    case 0:
        pFile->f_offset = offset;
        break;
    /* 读写位置加offset(可正可负) */
    case 1:
        pFile->f_offset += offset;
        break;
    /* 读写位置调整为文件长度加offset */
    case 2:
        pFile->f_offset = pFile->f_inode->i_size + offset;
        break;
    }
}

void FileManager::Dup()
{
    File *pFile;
    User &u = Kernel::Instance().GetUser();
    int fd = u.u_arg[0];

    pFile = u.u_ofiles.GetF(fd);
    if (NULL == pFile)
    {
        return;
    }

    int newFd = u.u_ofiles.AllocFreeSlot();
    if (newFd < 0)
    {
        return;
    }
    /* 至此分配新描述符newFd成功 */
    u.u_ofiles.SetF(newFd, pFile);
    pFile->f_count++;
}

void FileManager::FStat()
{
```

```
    File *pFile;
    User &u = Kernel::Instance().GetUser();
    int fd = u.u_arg[0];

    pFile = u.u_ofiles.GetF(fd);
    if (NULL == pFile)
    {
        return;
    }

    /* u.u_arg[1] = pStatBuf */
    this->Stat1(pFile->f_inode, u.u_arg[1]);
}

void FileManager::Stat()
{
    Inode *pInode;
    User &u = Kernel::Instance().GetUser();

    pInode = this->NameI(FileManager::NextChar, FileManager::OPEN);
    if (NULL == pInode)
    {
        return;
    }
    this->Stat1(pInode, u.u_arg[1]);
    this->m_InodeTable->IPut(pInode);
}

void FileManager::Stat1(Inode *pInode, unsigned long statBuf)
{
    Buf *pBuf;
    BufferManager &bufMgr = Kernel::Instance().GetBufferManager();

    pInode->IUpdate(Time::time);
    pBuf = bufMgr.Bread(pInode->i_dev, FileSystem::INODE_ZONE_START_SECTOR +
pInode->i_number / FileSystem::INODE_NUMBER_PER_SECTOR);

    /* 将p指向缓存区中编号为inumber外存Inode的偏移位置 */
    unsigned char *p = pBuf->b_addr + (pInode->i_number %
FileSystem::INODE_NUMBER_PER_SECTOR) * sizeof(DiskInode);
    Utility::DWordCopy((int *)p, (int *)statBuf, sizeof(DiskInode) /
sizeof(int));

    bufMgr.Brelse(pBuf);
}

void FileManager::Read()
{
    /* 直接调用Rdwr()函数即可 */
    this->Rdwr(File::FREAD);
}

void FileManager::Write()
{
    /* 直接调用Rdwr()函数即可 */
    this->Rdwr(File::FWRITE);
}
```

```
void FileManager::Rdwr(enum File::FileFlags mode)
{
    File *pFile;
    User &u = Kernel::Instance().GetUser();

    /* 根据Read()/Write()的系统调用参数fd获取打开文件控制块结构 */
    pFile = u.u_ofiles.GetF(u.u_arg[0]); /* fd */
    if (NULL == pFile)
    {
        /* 不存在该打开文件，GetF已经设置过出错码，所以这里不需要再设置了 */
        /*  u.u_error = User::EBADF;    */
        return;
    }

    /* 读写的模式不正确 */
    if ((pFile->f_flag & mode) == 0)
    {
        u.u_error = User::EACCES;
        return;
    }

    u.u_IOParam.m_Base = (unsigned char *)u.u_arg[1]; /* 目标缓冲区首址 */
    u.u_IOParam.m_Count = u.u_arg[2];                 /* 要求读/写的字节数 */
    u.u_segflg = 0;                                   /* User Space I/O，读入的内容
要送数据段或用户栈段 */

    /* 管道读写 */
    if (pFile->f_flag & File::FPIPE)
    {
        if (File::FREAD == mode)
        {
            this->ReadP(pFile);
        }
        else
        {
            this->WriteP(pFile);
        }
    }
    else
    /* 普通文件读写 ，或读写特殊文件。对文件实施互斥访问，互斥的粒度：每次系统调用。
    为此Inode类需要增加两个方法：NFlock()、NFrele()。
    这不是V6的设计。read、write系统调用对内存i节点上锁是为了给实施IO的进程提供一致的文件视
图。*/
    {
        pFile->f_inode->NFlock();
        /* 设置文件起始读位置 */
        u.u_IOParam.m_Offset = pFile->f_offset;
        if (File::FREAD == mode)
        {
            pFile->f_inode->ReadI();
        }
        else
        {
            pFile->f_inode->WriteI();
        }

        /* 根据读写字数，移动文件读写偏移指针 */
        pFile->f_offset += (u.u_arg[2] - u.u_IOParam.m_Count);
```

```
        pFile->f_inode->NFrele();
    }

    /* 返回实际读写的字节数，修改存放系统调用返回值的核心栈单元 */
    u.u_ar0[User::EAX] = u.u_arg[2] - u.u_IOParam.m_Count;
}

void FileManager::Pipe()
{
    Inode *pInode;
    File *pFileRead;
    File *pFileWrite;
    int fd[2];
    User &u = Kernel::Instance().GetUser();

    /* 分配一个Inode用于创建管道文件 */
    pInode = this->m_FileSystem->IAlloc(DeviceManager::ROOTDEV);
    if (NULL == pInode)
    {
        return;
    }

    /* 分配读管道的File结构 */
    pFileRead = this->m_OpenFileTable->FAlloc();
    if (NULL == pFileRead)
    {
        this->m_InodeTable->IPut(pInode);
        return;
    }
    /* 读管道的打开文件描述符 */
    fd[0] = u.u_ar0[User::EAX];

    /* 分配写管道的File结构 */
    pFileWrite = this->m_OpenFileTable->FAlloc();
    if (NULL == pFileWrite) /*若分配失败，擦除管道读端相关的所有打开文件结构*/
    {
        pFileRead->f_count = 0;
        u.u_ofiles.SetF(fd[0], NULL);
        this->m_InodeTable->IPut(pInode);
        return;
    }

    /* 写管道的打开文件描述符 */
    fd[1] = u.u_ar0[User::EAX];

    /* Pipe(int* fd)的参数在u.u_arg[0]中，将分配成功的2个fd返回给用户态程序 */
    int *pFdarr = (int *)u.u_arg[0];
    pFdarr[0] = fd[0];
    pFdarr[1] = fd[1];

    /* 设置读、写管道File结构的属性 ，以后read、write系统调用需要这个标识*/
    pFileRead->f_flag = File::FREAD | File::FPIPE;
    pFileRead->f_inode = pInode;
    pFileWrite->f_flag = File::FWRITE | File::FPIPE;
    pFileWrite->f_inode = pInode;

    pInode->i_count = 2;
    pInode->i_flag = Inode::IACC | Inode::IUPD;
```

```
    pInode->i_mode = Inode::IALLOC;
}

void FileManager::ReadP(File *pFile)
{
    Inode *pInode = pFile->f_inode;
    User &u = Kernel::Instance().GetUser();

loop:
    /* 对管道文件上锁保证互斥 ，在现在的V6版本普通文件的读写也采取这种非常保守的上锁方式*/
    pInode->Plock();

    /* 管道中没有数据可读取 。管道文件从尾部开始写，故i_size是写指针。*/
    if (pFile->f_offset == pInode->i_size)
    {
        if (pFile->f_offset != 0)
        {
            /* 读管道文件偏移量和管道文件大小重置为0 */
            pFile->f_offset = 0;
            pInode->i_size = 0;

            /* 如果置上IWRITE标志，则表示有进程正在等待写此管道，所以必须唤醒相应的进程。*/
            if (pInode->i_mode & Inode::IWRITE)
            {
                pInode->i_mode &= (~Inode::IWRITE);
                Kernel::Instance().GetProcessManager().WakeUpAll((unsigned long)
(pInode + 1));
            }
        }

        pInode->Prele(); /* 不解锁的话，写管道进程无法对管道实施操作。系统死锁 */

        /* 如果管道的读者、写者中已经有一方关闭，则返回 */
        if (pInode->i_count < 2)
        {
            return;
        }

        /* IREAD标志表示有进程等待读Pipe */
        pInode->i_mode |= Inode::IREAD;
        u.u_procp->Sleep((unsigned long)(pInode + 2), ProcessManager::PPIPE);
        goto loop;
    }

    /* 管道中有有可读取的数据 */
    u.u_IOParam.m_Offset = pFile->f_offset;
    pInode->ReadI();
    pFile->f_offset = u.u_IOParam.m_Offset;
    pInode->Prele();
}

void FileManager::WriteP(File *pFile)
{
    Inode *pInode = pFile->f_inode;
    User &u = Kernel::Instance().GetUser();

    int count = u.u_IOParam.m_Count;
```

```
loop:
    pInode->Plock();

    /* 已完成所有数据写入管道，对管道unlock并返回 */
    if (0 == count)
    {
        pInode->Prele();
        u.u_IOParam.m_Count = 0;
        return;
    }

    /* 管道读者进程已关闭读端、用信号SIGPIPE通知应用程序 */
    if (pInode->i_count < 2)
    {
        pInode->Prele();
        u.u_error = User::EPIPE;
        u.u_procp->PSignal(User::SIGPIPE);
        return;
    }

    /* 如果已经到达管道的底，则置上同步标志，睡眠等待 */
    if (Inode::PIPSIZ == pInode->i_size)
    {
        pInode->i_mode |= Inode::IWRITE;
        pInode->Prele();
        u.u_procp->Sleep((unsigned long)(pInode + 1), ProcessManager::PPIPE);
        goto loop;
    }

    /* 将待写入数据尽可能多地写入管道 */
    u.u_IOParam.m_Offset = pInode->i_size;
    u.u_IOParam.m_Count = Utility::Min(count, Inode::PIPSIZ -
u.u_IOParam.m_Offset);
    count -= u.u_IOParam.m_Count;
    pInode->WriteI();
    pInode->Prele();

    /* 唤醒读管道进程 */
    if (pInode->i_mode & Inode::IREAD)
    {
        pInode->i_mode &= (~Inode::IREAD);
        Kernel::Instance().GetProcessManager().WakeUpAll((unsigned long)(pInode
+ 2));
    }
    goto loop;
}

/* 返回NULL表示目录搜索失败，否则是根指针，指向文件的内存打开i节点 ，上锁的内存i节点  */
Inode *FileManager::NameI(char (*func)(), enum DirectorySearchMode mode)
{
    Inode *pInode;
    Buf *pBuf;
    char curchar;
    char *pChar;
    int freeEntryOffset; /* 以创建文件模式搜索目录时，记录空闲目录项的偏移量 */
    User &u = Kernel::Instance().GetUser();
    BufferManager &bufMgr = Kernel::Instance().GetBufferManager();
```

```
    /*
     * 如果该路径是'/'开头的，从根目录开始搜索，
     * 否则从进程当前工作目录开始搜索。
     */
    pInode = u.u_cdir;
    if ('/' == (curchar = (*func)()))
    {
        pInode = this->rootDirInode;
    }

    /* 检查该Inode是否正在被使用，以及保证在整个目录搜索过程中该Inode不被释放 */
    this->m_InodeTable->IGet(pInode->i_dev, pInode->i_number);

    /* 允许出现////a//b 这种路径 这种路径等价于/a/b */
    while ('/' == curchar)
    {
        curchar = (*func)();
    }
    /* 如果试图更改和删除当前目录文件则出错 */
    if ('\0' == curchar && mode != FileManager::OPEN)
    {
        u.u_error = User::ENOENT;
        goto out;
    }

    /* 外层循环每次处理pathname中一段路径分量 */
    while (true)
    {
        /* 如果出错则释放当前搜索到的目录文件Inode，并退出 */
        if (u.u_error != User::NOERROR)
        {
            break; /* goto out; */
        }

        /* 整个路径搜索完毕，返回相应Inode指针。目录搜索成功返回。 */
        if ('\0' == curchar)
        {
            return pInode;
        }

        /* 如果要进行搜索的不是目录文件，释放相关Inode资源则退出 */
        if ((pInode->i_mode & Inode::IFMT) != Inode::IFDIR)
        {
            u.u_error = User::ENOTDIR;
            break; /* goto out; */
        }

        /* 进行目录搜索权限检查,IEXEC在目录文件中表示搜索权限 */
        if (this->Access(pInode, Inode::IEXEC))
        {
            u.u_error = User::EACCES;
            break; /* 不具备目录搜索权限，goto out; */
        }

        /*
         * 将Pathname中当前准备进行匹配的路径分量拷贝到u.u_dbuf[]中，
         * 便于和目录项进行比较。
         */
```

```
        pChar = &(u.u_dbuf[0]);
        while ('/' != curchar && '\0' != curchar && u.u_error == User::NOERROR)
        {
            if (pChar < &(u.u_dbuf[DirectoryEntry::DIRSIZ]))
            {
                *pChar = curchar;
                pChar++;
            }
            curchar = (*func)();
        }
        /* 将u_dbuf剩余的部分填充为'\0' */
        while (pChar < &(u.u_dbuf[DirectoryEntry::DIRSIZ]))
        {
            *pChar = '\0';
            pChar++;
        }

        /* 允许出现////a//b 这种路径 这种路径等价于/a/b */
        while ('/' == curchar)
        {
            curchar = (*func)();
        }

        if (u.u_error != User::NOERROR)
        {
            break; /* goto out; */
        }

        /* 内层循环部分对于u.u_dbuf[]中的路径名分量，逐个搜寻匹配的目录项 */
        u.u_IOParam.m_Offset = 0;
        /* 设置为目录项个数 ，含空白的目录项*/
        u.u_IOParam.m_Count = pInode->i_size / (DirectoryEntry::DIRSIZ + 4);
        freeEntryOffset = 0;
        pBuf = NULL;

        while (true)
        {
            /* 对目录项已经搜索完毕 */
            if (0 == u.u_IOParam.m_Count)
            {
                if (NULL != pBuf)
                {
                    bufMgr.Brelse(pBuf);
                }
                /* 如果是创建新文件 */
                if (FileManager::CREATE == mode && curchar == '\0')
                {
                    /* 判断该目录是否可写 */
                    if (this->Access(pInode, Inode::IWRITE))
                    {
                        u.u_error = User::EACCES;
                        goto out; /* Failed */
                    }

                    /* 将父目录Inode指针保存起来，以后写目录项WriteDir()函数会用到 */
                    u.u_pdir = pInode;
```

```
                    if (freeEntryOffset) /* 此变量存放了空闲目录项位于目录文件中的偏移量
*/
                    {
                        /* 将空闲目录项偏移量存入u区中，写目录项WriteDir()会用到 */
                        u.u_IOParam.m_Offset = freeEntryOffset -
(DirectoryEntry::DIRSIZ + 4);
                    }
                    else /*问题：为何if分支没有置IUPD标志？  这是因为文件的长度没有变呀*/
                    {
                        pInode->i_flag |= Inode::IUPD;
                    }
                    /* 找到可以写入的空闲目录项位置，NameI()函数返回 */
                    return NULL;
                }

                /* 目录项搜索完毕而没有找到匹配项，释放相关Inode资源，并推出 */
                u.u_error = User::ENOENT;
                goto out;
            }

            /* 已读完目录文件的当前盘块，需要读入下一目录项数据盘块 */
            if (0 == u.u_IOParam.m_Offset % Inode::BLOCK_SIZE)
            {
                if (NULL != pBuf)
                {
                    bufMgr.Brelse(pBuf);
                }
                /* 计算要读的物理盘块号 */
                int phyBlkno = pInode->Bmap(u.u_IOParam.m_Offset /
Inode::BLOCK_SIZE);
                pBuf = bufMgr.Bread(pInode->i_dev, phyBlkno);
            }

            /* 没有读完当前目录项盘块，则读取下一目录项至u.u_dent */
            int *src = (int *)(pBuf->b_addr + (u.u_IOParam.m_Offset %
Inode::BLOCK_SIZE));
            Utility::DWordCopy(src, (int *)&u.u_dent, sizeof(DirectoryEntry) /
sizeof(int));

            u.u_IOParam.m_Offset += (DirectoryEntry::DIRSIZ + 4);
            u.u_IOParam.m_Count--;

            /* 如果是空闲目录项，记录该项位于目录文件中偏移量 */
            if (0 == u.u_dent.m_ino)
            {
                if (0 == freeEntryOffset)
                {
                    freeEntryOffset = u.u_IOParam.m_Offset;
                }
                /* 跳过空闲目录项，继续比较下一目录项 */
                continue;
            }

            int i;
            for (i = 0; i < DirectoryEntry::DIRSIZ; i++)
            {
                if (u.u_dbuf[i] != u.u_dent.m_name[i])
                {
```

```
                    break; /* 匹配至某一字符不符，跳出for循环 */
                }
            }

            if (i < DirectoryEntry::DIRSIZ)
            {
                /* 不是要搜索的目录项，继续匹配下一目录项 */
                continue;
            }
            else
            {
                /* 目录项匹配成功，回到外层While(true)循环 */
                break;
            }
        }

        /*
         * 从内层目录项匹配循环跳至此处，说明pathname中
         * 当前路径分量匹配成功了，还需匹配pathname中下一路径
         * 分量，直至遇到'\0'结束。
         */
        if (NULL != pBuf)
        {
            bufMgr.Brelse(pBuf);
        }

        /* 如果是删除操作，则返回父目录Inode，而要删除文件的Inode号在u.u_dent.m_ino中 */
        if (FileManager::DELETE == mode && '\0' == curchar)
        {
            /* 如果对父目录没有写的权限 */
            if (this->Access(pInode, Inode::IWRITE))
            {
                u.u_error = User::EACCES;
                break; /* goto out; */
            }
            return pInode;
        }

        /*
         * 匹配目录项成功，则释放当前目录Inode，根据匹配成功的
         * 目录项m_ino字段获取相应下一级目录或文件的Inode。
         */
        short dev = pInode->i_dev;
        this->m_InodeTable->IPut(pInode);
        pInode = this->m_InodeTable->IGet(dev, u.u_dent.m_ino);
        /* 回到外层While(true)循环，继续匹配Pathname中下一路径分量 */

        if (NULL == pInode) /* 获取失败 */
        {
            return NULL;
        }
    }
out:
    this->m_InodeTable->IPut(pInode);
    return NULL;
}

char FileManager::NextChar()
```

```
{
    User &u = Kernel::Instance().GetUser();

    /* u.u_dirp指向pathname中的字符 */
    return *u.u_dirp++;
}

/* 由creat调用。
 * 为新创建的文件写新的i节点和新的目录项
 * 返回的pInode是上了锁的内存i节点，其中的i_count是 1。
 *
 * 在程序的最后会调用 WriteDir，在这里把属于自己的目录项写进父目录，修改父目录文件的i节点 、将其写回磁盘。
 *
 */
Inode *FileManager::MakNode(unsigned int mode)
{
    Inode *pInode;
    User &u = Kernel::Instance().GetUser();

    /* 分配一个空闲DiskInode，里面内容已全部清空 */
    pInode = this->m_FileSystem->IAlloc(u.u_pdir->i_dev);
    if (NULL == pInode)
    {
        return NULL;
    }

    pInode->i_flag |= (Inode::IACC | Inode::IUPD);
    pInode->i_mode = mode | Inode::IALLOC;
    pInode->i_nlink = 1;
    pInode->i_uid = u.u_uid;
    pInode->i_gid = u.u_gid;
    /* 将目录项写入u.u_dent，随后写入目录文件 */
    this->WriteDir(pInode);
    return pInode;
}

void FileManager::WriteDir(Inode *pInode)
{
    User &u = Kernel::Instance().GetUser();

    /* 设置目录项中Inode编号部分 */
    u.u_dent.m_ino = pInode->i_number;

    /* 设置目录项中pathname分量部分 */
    for (int i = 0; i < DirectoryEntry::DIRSIZ; i++)
    {
        u.u_dent.m_name[i] = u.u_dbuf[i];
    }

    u.u_IOParam.m_Count = DirectoryEntry::DIRSIZ + 4;
    u.u_IOParam.m_Base = (unsigned char *)&u.u_dent;
    u.u_segflg = 1;

    /* 将目录项写入父目录文件 */
    u.u_pdir->WriteI();
    this->m_InodeTable->IPut(u.u_pdir);
}
```

```
void FileManager::SetCurDir(char *pathname)
{
    User &u = Kernel::Instance().GetUser();

    /* 路径不是从根目录'/'开始，则在现有u.u_curdir后面加上当前路径分量 */
    if (pathname[0] != '/')
    {
        int length = Utility::StringLength(u.u_curdir);
        if (u.u_curdir[length - 1] != '/')
        {
            u.u_curdir[length] = '/';
            length++;
        }
        Utility::StringCopy(pathname, u.u_curdir + length);
    }
    else /* 如果是从根目录'/'开始，则取代原有工作目录 */
    {
        Utility::StringCopy(pathname, u.u_curdir);
    }
}

/*
 * 返回值是0，表示拥有打开文件的权限；1表示没有所需的访问权限。文件未能打开的原因记录在
u.u_error变量中。
 */
int FileManager::Access(Inode *pInode, unsigned int mode)
{
    User &u = Kernel::Instance().GetUser();

    /* 对于写的权限，必须检查该文件系统是否是只读的 */
    if (Inode::IWRITE == mode)
    {
        if (this->m_FileSystem->GetFS(pInode->i_dev)->s_ronly != 0)
        {
            u.u_error = User::EROFS;
            return 1;
        }
    }
    /*
     * 对于超级用户，读写任何文件都是允许的
     * 而要执行某文件时，必须在i_mode有可执行标志
     */
    if (u.u_uid == 0)
    {
        if (Inode::IEXEC == mode && (pInode->i_mode & (Inode::IEXEC |
(Inode::IEXEC >> 3) | (Inode::IEXEC >> 6))) == 0)
        {
            u.u_error = User::EACCES;
            return 1;
        }
        return 0; /* Permission Check Succeed! */
    }
    if (u.u_uid != pInode->i_uid)
    {
        mode = mode >> 3;
        if (u.u_gid != pInode->i_gid)
        {
```

```
            mode = mode >> 3;
        }
    }
    if ((pInode->i_mode & mode) != 0)
    {
        return 0;
    }

    u.u_error = User::EACCES;
    return 1;
}

Inode *FileManager::Owner()
{
    Inode *pInode;
    User &u = Kernel::Instance().GetUser();

    if ((pInode = this->NameI(NextChar, FileManager::OPEN)) == NULL)
    {
        return NULL;
    }

    if (u.u_uid == pInode->i_uid || u.SUser())
    {
        return pInode;
    }

    this->m_InodeTable->IPut(pInode);
    return NULL;
}

void FileManager::ChMod()
{
    Inode *pInode;
    User &u = Kernel::Instance().GetUser();
    unsigned int mode = u.u_arg[1];

    if ((pInode = this->Owner()) == NULL)
    {
        return;
    }
    /* clear i_mode字段中的ISGID, ISUID, ISTVX以及rwxrwxrwx这12比特 */
    pInode->i_mode &= (~0xFFF);
    /* 根据系统调用的参数重新设置i_mode字段 */
    pInode->i_mode |= (mode & 0xFFF);
    pInode->i_flag |= Inode::IUPD;

    this->m_InodeTable->IPut(pInode);
    return;
}

void FileManager::ChOwn()
{
    Inode *pInode;
    User &u = Kernel::Instance().GetUser();
    short uid = u.u_arg[1];
    short gid = u.u_arg[2];
```

```
    /* 不是超级用户或者不是文件主则返回 */
    if (!u.SUser() || (pInode = this->Owner()) == NULL)
    {
        return;
    }
    pInode->i_uid = uid;
    pInode->i_gid = gid;
    pInode->i_flag |= Inode::IUPD;

    this->m_InodeTable->IPut(pInode);
}

void FileManager::ChDir()
{
    Inode *pInode;
    User &u = Kernel::Instance().GetUser();

    pInode = this->NameI(FileManager::NextChar, FileManager::OPEN);
    if (NULL == pInode)
    {
        return;
    }
    /* 搜索到的文件不是目录文件 */
    if ((pInode->i_mode & Inode::IFMT) != Inode::IFDIR)
    {
        u.u_error = User::ENOTDIR;
        this->m_InodeTable->IPut(pInode);
        return;
    }
    if (this->Access(pInode, Inode::IEXEC))
    {
        this->m_InodeTable->IPut(pInode);
        return;
    }
    this->m_InodeTable->IPut(u.u_cdir);
    u.u_cdir = pInode;
    pInode->Prele();

    this->SetCurDir((char *)u.u_arg[0] /* pathname */);
}

void FileManager::Link()
{
    Inode *pInode;
    Inode *pNewInode;
    User &u = Kernel::Instance().GetUser();

    pInode = this->NameI(FileManager::NextChar, FileManager::OPEN);
    /* 打卡文件失败 */
    if (NULL == pInode)
    {
        return;
    }
    /* 链接的数量已经最大 */
    if (pInode->i_nlink >= 255)
    {
        u.u_error = User::EMLINK;
        /* 出错，释放资源并退出 */
```

```
        this->m_InodeTable->IPut(pInode);
        return;
    }
    /* 对目录文件的链接只能由超级用户进行 */
    if ((pInode->i_mode & Inode::IFMT) == Inode::IFDIR && !u.SUser())
    {
        /* 出错，释放资源并退出 */
        this->m_InodeTable->IPut(pInode);
        return;
    }

    /* 解锁现存文件Inode,以避免在以下搜索新文件时产生死锁 */
    pInode->i_flag &= (~Inode::ILOCK);
    /* 指向要创建的新路径newPathname */
    u.u_dirp = (char *)u.u_arg[1];
    pNewInode = this->NameI(FileManager::NextChar, FileManager::CREATE);
    /* 如果文件已存在 */
    if (NULL != pNewInode)
    {
        u.u_error = User::EEXIST;
        this->m_InodeTable->IPut(pNewInode);
    }
    if (User::NOERROR != u.u_error)
    {
        /* 出错，释放资源并退出 */
        this->m_InodeTable->IPut(pInode);
        return;
    }
    /* 检查目录与该文件是否在同一个设备上 */
    if (u.u_pdir->i_dev != pInode->i_dev)
    {
        this->m_InodeTable->IPut(u.u_pdir);
        u.u_error = User::EXDEV;
        /* 出错，释放资源并退出 */
        this->m_InodeTable->IPut(pInode);
        return;
    }

    this->WriteDir(pInode);
    pInode->i_nlink++;
    pInode->i_flag |= Inode::IUPD;
    this->m_InodeTable->IPut(pInode);
}

void FileManager::UnLink()
{
    Inode *pInode;
    Inode *pDeleteInode;
    User &u = Kernel::Instance().GetUser();

    pDeleteInode = this->NameI(FileManager::NextChar, FileManager::DELETE);
    if (NULL == pDeleteInode)
    {
        return;
    }
    pDeleteInode->Prele();

    pInode = this->m_InodeTable->IGet(pDeleteInode->i_dev, u.u_dent.m_ino);
```

```
    if (NULL == pInode)
    {
        Utility::Panic("unlink -- iget");
    }
    /* 只有root可以unlink目录文件 */
    if ((pInode->i_mode & Inode::IFMT) == Inode::IFDIR && !u.SUser())
    {
        this->m_InodeTable->IPut(pDeleteInode);
        this->m_InodeTable->IPut(pInode);
        return;
    }
    /* 写入清零后的目录项 */
    u.u_IOParam.m_Offset -= (DirectoryEntry::DIRSIZ + 4);
    u.u_IOParam.m_Base = (unsigned char *)&u.u_dent;
    u.u_IOParam.m_Count = DirectoryEntry::DIRSIZ + 4;

    u.u_dent.m_ino = 0;
    pDeleteInode->WriteI();

    /* 修改inode项 */
    pInode->i_nlink--;
    pInode->i_flag |= Inode::IUPD;

    this->m_InodeTable->IPut(pDeleteInode);
    this->m_InodeTable->IPut(pInode);
}

void FileManager::MkNod()
{
    Inode *pInode;
    User &u = Kernel::Instance().GetUser();

    /* 检查uid是否是root，该系统调用只有uid==root时才可被调用 */
    if (u.SUser())
    {
        pInode = this->NameI(FileManager::NextChar, FileManager::CREATE);
        /* 要创建的文件已经存在,这里并不能去覆盖此文件 */
        if (pInode != NULL)
        {
            u.u_error = User::EEXIST;
            this->m_InodeTable->IPut(pInode);
            return;
        }
    }
    else
    {
        /* 非root用户执行mknod()系统调用返回User::EPERM */
        u.u_error = User::EPERM;
        return;
    }
    /* 没有通过SUser()的检查 */
    if (User::NOERROR != u.u_error)
    {
        return; /* 没有需要释放的资源，直接退出 */
    }
    pInode = this->MakNode(u.u_arg[1]);
    if (NULL == pInode)
    {
```

```
        return;
    }
    /* 所建立是设备文件 */
    if ((pInode->i_mode & (Inode::IFBLK | Inode::IFCHR)) != 0)
    {
        pInode->i_addr[0] = u.u_arg[2];
    }
    this->m_InodeTable->IPut(pInode);
}
/*============================class
DirectoryEntry================================*/
DirectoryEntry::DirectoryEntry()
{
    this->m_ino = 0;
    this->m_name[0] = '\0';
}

DirectoryEntry::~DirectoryEntry()
{
    // nothing to do here
}
```

FileSystem.h

```
#ifndef FILE_SYSTEM_H
#define FILE_SYSTEM_H

#include "INode.h"
#include "Buf.h"
#include "BufferManager.h"

/*
 * 文件系统存储资源管理块(Super Block)的定义。
 */
class SuperBlock
{
    /* Functions */
public:
    /* Constructors */
    SuperBlock();
    /* Destructors */
    ~SuperBlock();

    /* Members */
public:
    int s_isize; /* 外存Inode区占用的盘块数 */
    int s_fsize; /* 盘块总数 */

    int s_nfree;     /* 直接管理的空闲盘块数量 */
    int s_free[100]; /* 直接管理的空闲盘块索引表 */

    int s_ninode;     /* 直接管理的空闲外存Inode数量 */
    int s_inode[100]; /* 直接管理的空闲外存Inode索引表 */

    int s_flock; /* 封锁空闲盘块索引表标志 */
    int s_ilock; /* 封锁空闲Inode表标志 */
```

```
    int s_fmod;      /* 内存中super block副本被修改标志，意味着需要更新外存对应的Super
Block */
    int s_ronly;     /* 本文件系统只能读出 */
    int s_time;      /* 最近一次更新时间 */
    int padding[47]; /* 填充使SuperBlock块大小等于1024字节，占据2个扇区 */
};

/*
 * 文件系统装配块(Mount)的定义。
 * 装配块用于实现子文件系统与
 * 根文件系统的连接。
 */
class Mount
{
    /* Functions */
public:
    /* Constructors */
    Mount();
    /* Destructors */
    ~Mount();

    /* Members */
public:
    short m_dev;       /* 文件系统设备号 */
    SuperBlock *m_spb; /* 指向文件系统的Super Block对象在内存中的副本 */
    Inode *m_inodep;   /* 指向挂载子文件系统的内存INode */
};

/*
 * 文件系统类(FileSystem)管理文件存储设备中
 * 的各类存储资源，磁盘块、外存INode的分配、
 * 释放。
 */
class FileSystem
{
public:
    /* static consts */
    static const int NMOUNT = 5; /* 系统中用于挂载子文件系统的装配块数量 */

    static const int SUPER_BLOCK_SECTOR_NUMBER = 200; /* 定义SuperBlock位于磁盘上的
扇区号，占据200，201两个扇区。 */

    static const int ROOTINO = 1; /* 文件系统根目录外存Inode编号 */

    static const int INODE_NUMBER_PER_SECTOR = 8;   /* 外存INode对象长度为64字节，每
个磁盘块可以存放512/64 = 8个外存Inode */
    static const int INODE_ZONE_START_SECTOR = 202; /* 外存Inode区位于磁盘上的起始扇
区号 */
    static const int INODE_ZONE_SIZE = 1024 - 202;  /* 磁盘上外存Inode区占据的扇区数
*/

    static const int DATA_ZONE_START_SECTOR = 1024;                    /* 数据区的
起始扇区号 */
    static const int DATA_ZONE_END_SECTOR = 18000 - 1;                 /* 数据区的
结束扇区号 */
    static const int DATA_ZONE_SIZE = 18000 - DATA_ZONE_START_SECTOR; /* 数据区占
据的扇区数量 */
```

```
    /* Functions */
public:
    /* Constructors */
    FileSystem();
    /* Destructors */
    ~FileSystem();

    /*
     * @comment 初始化成员变量
     */
    void Initialize();

    /*
     * @comment 系统初始化时读入SuperBlock
     */
    void LoadSuperBlock();

    /*
     * @comment 根据文件存储设备的设备号dev获取
     * 该文件系统的SuperBlock
     */
    SuperBlock *GetFS(short dev);
    /*
     * @comment 将SuperBlock对象的内存副本更新到
     * 存储设备的SuperBlock中去
     */
    void Update();

    /*
     * @comment  在存储设备dev上分配一个空闲
     * 外存INode，一般用于创建新的文件。
     */
    Inode *IAlloc(short dev);
    /*
     * @comment  释放存储设备dev上编号为number
     * 的外存INode，一般用于删除文件。
     */
    void IFree(short dev, int number);

    /*
     * @comment 在存储设备dev上分配空闲磁盘块
     */
    Buf *Alloc(short dev);
    /*
     * @comment 释放存储设备dev上编号为blkno的磁盘块
     */
    void Free(short dev, int blkno);

    /*
     * @comment 查找文件系统装配表，搜索指定Inode对应的Mount装配块
     */
    Mount *GetMount(Inode *pInode);

private:
    /*
     * @comment 检查设备dev上编号blkno的磁盘块是否属于
     * 数据盘块区
```

```
     */
    bool BadBlock(SuperBlock *spb, short dev, int blkno);

    /* Members */
public:
    Mount m_Mount[NMOUNT]; /* 文件系统装配块表，Mount[0]用于根文件系统 */

private:
    BufferManager *m_BufferManager; /* FileSystem类需要缓存管理模块(BufferManager)提
供的接口 */
    int updlock;                    /* Update()函数的锁，该函数用于同步内存各个
SuperBlock副本以及，
                                       被修改过的内存Inode。任一时刻只允许一个进程调用该函数
*/
};

#endif
```

FileSystem.cpp

```
#include "FileSystem.h"
#include "Utility.h"
#include "New.h"
#include "Kernel.h"
#include "OpenFileManager.h"
#include "TimeInterrupt.h"
#include "Video.h"

/*==============================class
SuperBlock===================================*/
/* 系统全局超级块SuperBlock对象 */
SuperBlock g_spb;

SuperBlock::SuperBlock()
{
    // nothing to do here
}

SuperBlock::~SuperBlock()
{
    // nothing to do here
}

/*==============================class Mount=================================*/
Mount::Mount()
{
    this->m_dev = -1;
    this->m_spb = NULL;
    this->m_inodep = NULL;
}

Mount::~Mount()
{
    this->m_dev = -1;
    this->m_inodep = NULL;
```

```
    // 释放内存SuperBlock副本
    if (this->m_spb != NULL)
    {
        delete this->m_spb;
        this->m_spb = NULL;
    }
}

/*==============================class
FileSystem====================================*/
FileSystem::FileSystem()
{
    // nothing to do here
}

FileSystem::~FileSystem()
{
    // nothing to do here
}

void FileSystem::Initialize()
{
    this->m_BufferManager = &Kernel::Instance().GetBufferManager();
    this->updlock = 0;
}

void FileSystem::LoadSuperBlock()
{
    User &u = Kernel::Instance().GetUser();
    BufferManager &bufMgr = Kernel::Instance().GetBufferManager();
    Buf *pBuf;

    for (int i = 0; i < 2; i++)
    {
        int *p = (int *)&g_spb + i * 128;

        pBuf = bufMgr.Bread(DeviceManager::ROOTDEV,
FileSystem::SUPER_BLOCK_SECTOR_NUMBER + i);

        Utility::DWordCopy((int *)pBuf->b_addr, p, 128);

        bufMgr.Brelse(pBuf);
    }
    if (User::NOERROR != u.u_error)
    {
        Utility::Panic("Load SuperBlock Error....!\n");
    }

    this->m_Mount[0].m_dev = DeviceManager::ROOTDEV;
    this->m_Mount[0].m_spb = &g_spb;

    g_spb.s_flock = 0;
    g_spb.s_ilock = 0;
    g_spb.s_ronly = 0;
    g_spb.s_time = Time::time;
}

SuperBlock *FileSystem::GetFS(short dev)
```

```
{
    SuperBlock *sb;

    /* 遍历系统装配块表，寻找设备号为dev的设备中文件系统的SuperBlock */
    for (int i = 0; i < FileSystem::NMOUNT; i++)
    {
        if (this->m_Mount[i].m_spb != NULL && this->m_Mount[i].m_dev == dev)
        {
            /* 获取SuperBlock的地址 */
            sb = this->m_Mount[i].m_spb;
            if (sb->s_nfree > 100 || sb->s_ninode > 100)
            {
                sb->s_nfree = 0;
                sb->s_ninode = 0;
            }
            return sb;
        }
    }

    Utility::Panic("No File System!");
    return NULL;
}

void FileSystem::Update()
{
    int i;
    SuperBlock *sb;
    Buf *pBuf;

    /* 另一进程正在进行同步，则直接返回 */
    if (this->updlock)
    {
        return;
    }

    /* 设置Update()函数的互斥锁，防止其它进程重入 */
    this->updlock++;

    /* 同步SuperBlock到磁盘 */
    for (i = 0; i < FileSystem::NMOUNT; i++)
    {
        if (this->m_Mount[i].m_spb != NULL) /* 该Mount装配块对应某个文件系统 */
        {
            sb = this->m_Mount[i].m_spb;

            /* 如果该SuperBlock内存副本没有被修改，直接管理inode和空闲盘块被上锁或该文件系
统是只读文件系统 */
            if (sb->s_fmod == 0 || sb->s_ilock != 0 || sb->s_flock != 0 || sb-
>s_ronly != 0)
            {
                continue;
            }

            /* 清SuperBlock修改标志 */
            sb->s_fmod = 0;
            /* 写入SuperBlock最后存访时间 */
            sb->s_time = Time::time;
```

```
            /*
             * 为将要写回到磁盘上去的SuperBlock申请一块缓存，由于缓存块大小为512字节，
             * SuperBlock大小为1024字节，占据2个连续的扇区，所以需要2次写入操作。
             */
            for (int j = 0; j < 2; j++)
            {
                /* 第一次p指向SuperBlock的第0字节，第二次p指向第512字节 */
                int *p = (int *)sb + j * 128;

                /* 将要写入到设备dev上的SUPER_BLOCK_SECTOR_NUMBER + j扇区中去 */
                pBuf = this->m_BufferManager->GetBlk(this->m_Mount[i].m_dev,
FileSystem::SUPER_BLOCK_SECTOR_NUMBER + j);

                /* 将SuperBlock中第0 - 511字节写入缓存区 */
                Utility::DWordCopy(p, (int *)pBuf->b_addr, 128);

                /* 将缓冲区中的数据写到磁盘上 */
                this->m_BufferManager->Bwrite(pBuf);
            }
        }
    }

    /* 同步修改过的内存Inode到对应外存Inode */
    g_InodeTable.UpdateInodeTable();

    /* 清除Update()函数锁 */
    this->updlock = 0;

    /* 将延迟写的缓存块写到磁盘上 */
    this->m_BufferManager->Bflush(DeviceManager::NODEV);
}

Inode *FileSystem::IAlloc(short dev)
{
    SuperBlock *sb;
    Buf *pBuf;
    Inode *pNode;
    User &u = Kernel::Instance().GetUser();
    int ino; /* 分配到的空闲外存Inode编号 */

    /* 获取相应设备的SuperBlock内存副本 */
    sb = this->GetFS(dev);

    /* 如果SuperBlock空闲Inode表被上锁，则睡眠等待至解锁 */
    while (sb->s_ilock)
    {
        u.u_procp->Sleep((unsigned long)&sb->s_ilock, ProcessManager::PINOD);
    }

    /*
     * SuperBlock直接管理的空闲Inode索引表已空，
     * 必须到磁盘上搜索空闲Inode。先对inode列表上锁，
     * 因为在以下程序中会进行读盘操作可能会导致进程切换，
     * 其他进程有可能访问该索引表，将会导致不一致性。
     */
    if (sb->s_ninode <= 0)
    {
        /* 空闲Inode索引表上锁 */
```

```
        sb->s_ilock++;

        /* 外存Inode编号从0开始，这不同于Unix V6中外存Inode从1开始编号 */
        ino = -1;

        /* 依次读入磁盘Inode区中的磁盘块，搜索其中空闲外存Inode，记入空闲Inode索引表 */
        for (int i = 0; i < sb->s_isize; i++)
        {
            pBuf = this->m_BufferManager->Bread(dev,
FileSystem::INODE_ZONE_START_SECTOR + i);

            /* 获取缓冲区首址 */
            int *p = (int *)pBuf->b_addr;

            /* 检查该缓冲区中每个外存Inode的i_mode != 0，表示已经被占用 */
            for (int j = 0; j < FileSystem::INODE_NUMBER_PER_SECTOR; j++)
            {
                ino++;

                int mode = *(p + j * sizeof(DiskInode) / sizeof(int));

                /* 该外存Inode已被占用，不能记入空闲Inode索引表 */
                if (mode != 0)
                {
                    continue;
                }

                /*
                 * 如果外存inode的i_mode==0，此时并不能确定
                 * 该inode是空闲的，因为有可能是内存inode没有写到
                 * 磁盘上,所以要继续搜索内存inode中是否有相应的项
                 */
                if (g_InodeTable.IsLoaded(dev, ino) == -1)
                {
                    /* 该外存Inode没有对应的内存拷贝，将其记入空闲Inode索引表 */
                    sb->s_inode[sb->s_ninode++] = ino;

                    /* 如果空闲索引表已经装满，则不继续搜索 */
                    if (sb->s_ninode >= 100)
                    {
                        break;
                    }
                }
            }

            /* 至此已读完当前磁盘块，释放相应的缓存 */
            this->m_BufferManager->Brelse(pBuf);

            /* 如果空闲索引表已经装满，则不继续搜索 */
            if (sb->s_ninode >= 100)
            {
                break;
            }
        }
        /* 解除对空闲外存Inode索引表的锁，唤醒因为等待锁而睡眠的进程 */
        sb->s_ilock = 0;
        Kernel::Instance().GetProcessManager().WakeUpAll((unsigned long)&sb-
>s_ilock);
```

```
        /* 如果在磁盘上没有搜索到任何可用外存Inode，返回NULL */
        if (sb->s_ninode <= 0)
        {
            Diagnose::Write("No Space On %d !\n", dev);
            u.u_error = User::ENOSPC;
            return NULL;
        }
    }

    /*
     * 上面部分已经保证，除非系统中没有可用外存Inode，
     * 否则空闲Inode索引表中必定会记录可用外存Inode的编号。
     */
    while (true)
    {
        /* 从索引表“栈顶”获取空闲外存Inode编号 */
        ino = sb->s_inode[--sb->s_ninode];

        /* 将空闲Inode读入内存 */
        pNode = g_InodeTable.IGet(dev, ino);
        /* 未能分配到内存inode */
        if (NULL == pNode)
        {
            return NULL;
        }

        /* 如果该Inode空闲,清空Inode中的数据 */
        if (0 == pNode->i_mode)
        {
            pNode->Clean();
            /* 设置SuperBlock被修改标志 */
            sb->s_fmod = 1;
            return pNode;
        }
        else /* 如果该Inode已被占用 */
        {
            g_InodeTable.IPut(pNode);
            continue; /* while循环 */
        }
    }
    return NULL; /* GCC likes it! */
}

void FileSystem::IFree(short dev, int number)
{
    SuperBlock *sb;

    sb = this->GetFS(dev); /* 获取相应设备的SuperBlock内存副本 */

    /*
     * 如果超级块直接管理的空闲Inode表上锁，
     * 则释放的外存Inode散落在磁盘Inode区中。
     */
    if (sb->s_ilock)
    {
        return;
    }
```

```
    /*
     * 如果超级块直接管理的空闲外存Inode超过100个，
     * 同样让释放的外存Inode散落在磁盘Inode区中。
     */
    if (sb->s_ninode >= 100)
    {
        return;
    }

    sb->s_inode[sb->s_ninode++] = number;

    /* 设置SuperBlock被修改标志 */
    sb->s_fmod = 1;
}

Buf *FileSystem::Alloc(short dev)
{
    int blkno; /* 分配到的空闲磁盘块编号 */
    SuperBlock *sb;
    Buf *pBuf;
    User &u = Kernel::Instance().GetUser();

    /* 获取SuperBlock对象的内存副本 */
    sb = this->GetFS(dev);

    /*
     * 如果空闲磁盘块索引表正在被上锁，表明有其它进程
     * 正在操作空闲磁盘块索引表，因而对其上锁。这通常
     * 是由于其余进程调用Free()或Alloc()造成的。
     */
    while (sb->s_flock)
    {
        /* 进入睡眠直到获得该锁才继续 */
        u.u_procp->Sleep((unsigned long)&sb->s_flock, ProcessManager::PINOD);
    }

    /* 从索引表“栈顶”获取空闲磁盘块编号 */
    blkno = sb->s_free[--sb->s_nfree];

    /*
     * 若获取磁盘块编号为零，则表示已分配尽所有的空闲磁盘块。
     * 或者分配到的空闲磁盘块编号不属于数据盘块区域中(由BadBlock()检查)，
     * 都意味着分配空闲磁盘块操作失败。
     */
    if (0 == blkno)
    {
        sb->s_nfree = 0;
        Diagnose::Write("No Space On %d !\n", dev);
        u.u_error = User::ENOSPC;
        return NULL;
    }
    if (this->BadBlock(sb, dev, blkno))
    {
        return NULL;
    }

    /*
```

```
     * 栈已空，新分配到空闲磁盘块中记录了下一组空闲磁盘块的编号，
     * 将下一组空闲磁盘块的编号读入SuperBlock的空闲磁盘块索引表s_free[100]中。
     */
    if (sb->s_nfree <= 0)
    {
        /*
         * 此处加锁，因为以下要进行读盘操作，有可能发生进程切换，
         * 新上台的进程可能对SuperBlock的空闲盘块索引表访问，会导致不一致性。
         */
        sb->s_flock++;

        /* 读入该空闲磁盘块 */
        pBuf = this->m_BufferManager->Bread(dev, blkno);

        /* 从该磁盘块的0字节开始记录，共占据4(s_nfree)+400(s_free[100])个字节 */
        int *p = (int *)pBuf->b_addr;

        /* 首先读出空闲盘块数s_nfree */
        sb->s_nfree = *p++;

        /* 读取缓存中后续位置的数据，写入到SuperBlock空闲盘块索引表s_free[100]中 */
        Utility::DWordCopy(p, sb->s_free, 100);

        /* 缓存使用完毕，释放以便被其它进程使用 */
        this->m_BufferManager->Brelse(pBuf);

        /* 解除对空闲磁盘块索引表的锁，唤醒因为等待锁而睡眠的进程 */
        sb->s_flock = 0;
        Kernel::Instance().GetProcessManager().WakeUpAll((unsigned long)&sb-
>s_flock);
    }

    /* 普通情况下成功分配到一空闲磁盘块 */
    pBuf = this->m_BufferManager->GetBlk(dev, blkno); /* 为该磁盘块申请缓存 */
    this->m_BufferManager->ClrBuf(pBuf);              /* 清空缓存中的数据 */
    sb->s_fmod = 1;                                   /* 设置SuperBlock被修改标志
*/

    return pBuf;
}

void FileSystem::Free(short dev, int blkno)
{
    SuperBlock *sb;
    Buf *pBuf;
    User &u = Kernel::Instance().GetUser();

    sb = this->GetFS(dev);

    /*
     * 尽早设置SuperBlock被修改标志，以防止在释放
     * 磁盘块Free()执行过程中，对SuperBlock内存副本
     * 的修改仅进行了一半，就更新到磁盘SuperBlock去
     */
    sb->s_fmod = 1;

    /* 如果空闲磁盘块索引表被上锁，则睡眠等待解锁 */
    while (sb->s_flock)
```

```
    {
        u.u_procp->Sleep((unsigned long)&sb->s_flock, ProcessManager::PINOD);
    }

    /* 检查释放磁盘块的合法性 */
    if (this->BadBlock(sb, dev, blkno))
    {
        return;
    }

    /*
     * 如果先前系统中已经没有空闲盘块，
     * 现在释放的是系统中第1块空闲盘块
     */
    if (sb->s_nfree <= 0)
    {
        sb->s_nfree = 1;
        sb->s_free[0] = 0; /* 使用0标记空闲盘块链结束标志 */
    }

    /* SuperBlock中直接管理空闲磁盘块号的栈已满 */
    if (sb->s_nfree >= 100)
    {
        sb->s_flock++;

        /*
         * 使用当前Free()函数正要释放的磁盘块，存放前一组100个空闲
         * 磁盘块的索引表
         */
        pBuf = this->m_BufferManager->GetBlk(dev, blkno); /* 为当前正要释放的磁盘块分
配缓存 */

        /* 从该磁盘块的0字节开始记录，共占据4(s_nfree)+400(s_free[100])个字节 */
        int *p = (int *)pBuf->b_addr;

        /* 首先写入空闲盘块数，除了第一组为99块，后续每组都是100块 */
        *p++ = sb->s_nfree;
        /* 将SuperBlock的空闲盘块索引表s_free[100]写入缓存中后续位置 */
        Utility::DWordCopy(sb->s_free, p, 100);

        sb->s_nfree = 0;
        /* 将存放空闲盘块索引表的“当前释放盘块”写入磁盘，即实现了空闲盘块记录空闲盘块号的目标
*/
        this->m_BufferManager->Bwrite(pBuf);

        sb->s_flock = 0;
        Kernel::Instance().GetProcessManager().WakeUpAll((unsigned long)&sb-
>s_flock);
    }
    sb->s_free[sb->s_nfree++] = blkno; /* SuperBlock中记录下当前释放盘块号 */
    sb->s_fmod = 1;
}

Mount *FileSystem::GetMount(Inode *pInode)
{
    /* 遍历系统的装配块表 */
    for (int i = 0; i <= FileSystem::NMOUNT; i++)
    {
```

```
        Mount *pMount = &(this->m_Mount[i]);

        /* 找到内存Inode对应的Mount装配块 */
        if (pMount->m_inodep == pInode)
        {
            return pMount;
        }
    }
    return NULL; /* 查找失败 */
}

bool FileSystem::BadBlock(SuperBlock *spb, short dev, int blkno)
{
    return 0;
}
```

Inode.h

```
#ifndef INODE_H
#define INODE_H

#include "Buf.h"

/*
 * 内存索引节点(INode)的定义
 * 系统中每一个打开的文件、当前访问目录、
 * 挂载的子文件系统都对应唯一的内存inode。
 * 每个内存inode通过外存inode所在存储设备的设备号(i_dev)
 * 以及该设备外存inode区中的编号(i_number)来确定
 * 其对应的外存inode。
 */
class Inode
{
public:
    /* i_flag中标志位 */
    enum INodeFlag
    {
        ILOCK = 0x1,  /* 索引节点上锁 */
        IUPD = 0x2,   /* 内存inode被修改过，需要更新相应外存inode */
        IACC = 0x4,   /* 内存inode被访问过，需要修改最近一次访问时间 */
        IMOUNT = 0x8, /* 内存inode用于挂载子文件系统 */
        IWANT = 0x10, /* 有进程正在等待该内存inode被解锁，清ILOCK标志时，要唤醒这种进程
*/
        ITEXT = 0x20  /* 内存inode对应进程图像的正文段 */
    };

    /* static const member */
    static const unsigned int IALLOC = 0x8000;                  /* 文件被使用 */
    static const unsigned int IFMT = 0x6000;                    /* 文件类型掩码 */
    static const unsigned int IFDIR = 0x4000;                   /* 文件类型：目录文
件 */
    static const unsigned int IFCHR = 0x2000;                   /* 字符设备特殊类型
文件 */
```

```
    static const unsigned int IFBLK = 0x6000;                   /* 块设备特殊类型文
件，为0表示常规数据文件 */
    static const unsigned int ILARG = 0x1000;                   /* 文件长度类型：大
型或巨型文件 */
    static const unsigned int ISUID = 0x800;                    /* 执行时文件时将用
户的有效用户ID修改为文件所有者的User ID */
    static const unsigned int ISGID = 0x400;                    /* 执行时文件时将用
户的有效组ID修改为文件所有者的Group ID */
    static const unsigned int ISVTX = 0x200;                    /* 使用后仍然位于交
换区上的正文段 */
    static const unsigned int IREAD = 0x100;                    /* 对文件的读权限
*/
    static const unsigned int IWRITE = 0x80;                    /* 对文件的写权限
*/
    static const unsigned int IEXEC = 0x40;                     /* 对文件的执行权限
*/
    static const unsigned int IRWXU = (IREAD | IWRITE | IEXEC); /* 文件主对文件的
读、写、执行权限 */
    static const unsigned int IRWXG = ((IRWXU) >> 3);           /* 文件主同组用户对
文件的读、写、执行权限 */
    static const unsigned int IRWXO = ((IRWXU) >> 6);           /* 其他用户对文件的
读、写、执行权限 */

    static const int BLOCK_SIZE = 512;                                  /* 文件
逻辑块大小: 512字节 */
    static const int ADDRESS_PER_INDEX_BLOCK = BLOCK_SIZE / sizeof(int); /* 每个
间接索引表(或索引块)包含的物理盘块号 */

    static const int SMALL_FILE_BLOCK = 6;                          /* 小型文件：直
接索引表最多可寻址的逻辑块号 */
    static const int LARGE_FILE_BLOCK = 128 * 2 + 6;                /* 大型文件：经
一次间接索引表最多可寻址的逻辑块号 */
    static const int HUGE_FILE_BLOCK = 128 * 128 * 2 + 128 * 2 + 6; /* 巨型文件:
经二次间接索引最大可寻址文件逻辑块号 */

    static const int PIPSIZ = SMALL_FILE_BLOCK * BLOCK_SIZE;

    /* static member */
    static int rablock; /* 顺序读时，使用预读技术读入文件的下一字符块，rablock记录了下一逻
辑块号
                           经过bmap转换得到的物理盘块号。将rablock作为静态变量的原因：调用一
次bmap的开销
                           对当前块和预读块的逻辑块号进行转换，bmap返回当前块的物理盘块号，并
且将预读块
                           的物理盘块号保存在rablock中。 */

    /* Functions */
public:
    /* Constructors */
    Inode();
    /* Destructors */
    ~Inode();

    /*
     * @comment 根据Inode对象中的物理磁盘块索引表，读取相应
     * 的文件数据
     */
    void ReadI();
```

```
    /*
     * @comment 根据Inode对象中的物理磁盘块索引表，将数据写入文件
     */
    void WriteI();
    /*
     * @comment 将文件的逻辑块号转换成对应的物理盘块号
     */
    int Bmap(int lbn);

    /*
     * @comment 对特殊字符设备、块设备文件，调用该设备注册在块设备开关表
     * 中的设备初始化程序
     */
    void OpenI(int mode);
    /*
     * @comment 对特殊字符设备、块设备文件。如果对该设备的引用计数为0，
     * 则调用该设备的关闭程序
     */
    void CloseI(int mode);

    /*
     * @comment 更新外存Inode的最后的访问时间、修改时间
     */
    void IUpdate(int time);
    /*
     * @comment 释放Inode对应文件占用的磁盘块
     */
    void ITrunc();

    /*
     * @comment 对Pipe或者Inode解锁，并且唤醒因等待锁而睡眠的进程
     */
    void Prele();

    /*
     * @comment 对Pipe上锁，如果Pipe已经被上锁，则增设IWANT标志并睡眠等待直至解锁
     */
    void Plock();

    /*
     * @comment 对Pipe或者Inode解锁，并且唤醒因等待锁而睡眠的进程
     */
    void NFrele();

    /*
     * @comment 对Pipe上锁，如果Pipe已经被上锁，则增设IWANT标志并睡眠等待直至解锁
     */
    void NFlock();

    /*
     * @comment 清空Inode对象中的数据
     */
    void Clean();
    /*
     * @comment 将包含外存Inode字符块中信息拷贝到内存Inode中
     */
    void ICopy(Buf *bp, int inumber);
```

```
    /* Members */
public:
    unsigned int i_flag; /* 状态的标志位，定义见enum INodeFlag */
    unsigned int i_mode; /* 文件工作方式信息 */

    int i_count; /* 引用计数 */
    int i_nlink; /* 文件联结计数，即该文件在目录树中不同路径名的数量 */

    short i_dev;  /* 外存inode所在存储设备的设备号 */
    int i_number; /* 外存inode区中的编号 */

    short i_uid; /* 文件所有者的用户标识数 */
    short i_gid; /* 文件所有者的组标识数 */

    int i_size;     /* 文件大小，字节为单位 */
    int i_addr[10]; /* 用于文件逻辑块好和物理块好转换的基本索引表 */

    int i_lastr; /* 存放最近一次读取文件的逻辑块号，用于判断是否需要预读 */
};

/*
 * 外存索引节点(DiskINode)的定义
 * 外存Inode位于文件存储设备上的
 * 外存Inode区中。每个文件有唯一对应
 * 的外存Inode，其作用是记录了该文件
 * 对应的控制信息。
 * 外存Inode中许多字段和内存Inode中字段
 * 相对应。外存INode对象长度为64字节，
 * 每个磁盘块可以存放512/64 = 8个外存Inode
 */
class DiskInode
{
    /* Functions */
public:
    /* Constructors */
    DiskInode();
    /* Destructors */
    ~DiskInode();

    /* Members */
public:
    unsigned int d_mode; /* 状态的标志位，定义见enum INodeFlag */
    int d_nlink;         /* 文件联结计数，即该文件在目录树中不同路径名的数量 */

    short d_uid; /* 文件所有者的用户标识数 */
    short d_gid; /* 文件所有者的组标识数 */

    int d_size;     /* 文件大小，字节为单位 */
    int d_addr[10]; /* 用于文件逻辑块好和物理块好转换的基本索引表 */

    int d_atime; /* 最后访问时间 */
    int d_mtime; /* 最后修改时间 */
};

#endif
```

Inode.cpp

```
#include "INode.h"
#include "Utility.h"
#include "DeviceManager.h"
#include "Kernel.h"

/*==============================class Inode===================================*/
/*  预读块的块号，对普通文件这是预读块所在的物理块号。对硬盘而言，这是当前物理块（扇区）的下一个物理块（扇区）*/
int Inode::rablock = 0;

/* 内存打开 i节点*/
Inode::Inode()
{
    /* 清空Inode对象中的数据 */
    // this->Clean();
    /* 去除this->Clean();的理由：
     * Inode::Clean()特定用于IAlloc()中清空新分配DiskInode的原有数据，
     * 即旧文件信息。Clean()函数中不应当清除i_dev, i_number, i_flag, i_count,
     * 这是属于内存Inode而非DiskInode包含的旧文件信息，而Inode类构造函数需要
     * 将其初始化为无效值。
     */

    /* 将Inode对象的成员变量初始化为无效值 */
    this->i_flag = 0;
    this->i_mode = 0;
    this->i_count = 0;
    this->i_nlink = 0;
    this->i_dev = -1;
    this->i_number = -1;
    this->i_uid = -1;
    this->i_gid = -1;
    this->i_size = 0;
    this->i_lastr = -1;
    for (int i = 0; i < 10; i++)
    {
        this->i_addr[i] = 0;
    }
}

Inode::~Inode()
{
    // nothing to do here
}

void Inode::ReadI()
{
    int lbn;    /* 文件逻辑块号 */
    int bn;     /* lbn对应的物理盘块号 */
    int offset; /* 当前字符块内起始传送位置 */
    int nbytes; /* 传送至用户目标区字节数量 */
    short dev;
    Buf *pBuf;
    User &u = Kernel::Instance().GetUser();
```

```
    BufferManager &bufMgr = Kernel::Instance().GetBufferManager();
    DeviceManager &devMgr = Kernel::Instance().GetDeviceManager();

    if (0 == u.u_IOParam.m_Count)
    {
        /* 需要读字节数为零，则返回 */
        return;
    }

    this->i_flag |= Inode::IACC;

    /* 如果是字符设备文件 ，调用外设读函数*/
    if ((this->i_mode & Inode::IFMT) == Inode::IFCHR)
    {
        short major = Utility::GetMajor(this->i_addr[0]);

        devMgr.GetCharDevice(major).Read(this->i_addr[0]);
        return;
    }

    /* 一次一个字符块地读入所需全部数据，直至遇到文件尾 */
    while (User::NOERROR == u.u_error && u.u_IOParam.m_Count != 0)
    {
        lbn = bn = u.u_IOParam.m_Offset / Inode::BLOCK_SIZE;
        offset = u.u_IOParam.m_Offset % Inode::BLOCK_SIZE;
        /* 传送到用户区的字节数量，取读请求的剩余字节数与当前字符块内有效字节数较小值 */
        nbytes = Utility::Min(Inode::BLOCK_SIZE - offset /* 块内有效字节数 */,
u.u_IOParam.m_Count);

        if ((this->i_mode & Inode::IFMT) != Inode::IFBLK)
        { /* 如果不是特殊块设备文件 */

            int remain = this->i_size - u.u_IOParam.m_Offset;
            /* 如果已读到超过文件结尾 */
            if (remain <= 0)
            {
                return;
            }
            /* 传送的字节数量还取决于剩余文件的长度 */
            nbytes = Utility::Min(nbytes, remain);

            /* 将逻辑块号lbn转换成物理盘块号bn ，Bmap有设置Inode::rablock。当UNIX认为获
取预读块的开销太大时，
             * 会放弃预读，此时 Inode::rablock 值为 0。
             * */
            if ((bn = this->Bmap(lbn)) == 0)
            {
                return;
            }
            dev = this->i_dev;
        }
        else /* 如果是特殊块设备文件 */
        {
            dev = this->i_addr[0]; /* 特殊块设备文件i_addr[0]中存放的是设备号 */
            Inode::rablock = bn + 1;
        }

        if (this->i_lastr + 1 == lbn) /* 如果是顺序读，则进行预读 */
```

```
        {
            /* 读当前块，并预读下一块 */
            pBuf = bufMgr.Breada(dev, bn, Inode::rablock);
        }
        else
        {
            pBuf = bufMgr.Bread(dev, bn);
        }
        /* 记录最近读取字符块的逻辑块号 */
        this->i_lastr = lbn;

        /* 缓存中数据起始读位置 */
        unsigned char *start = pBuf->b_addr + offset;

        /* 读操作: 从缓冲区拷贝到用户目标区
         * i386芯片用同一张页表映射用户空间和内核空间，这一点硬件上的差异 使得i386上实现
iomove操作
         * 比PDP-11要容易许多*/
        Utility::IOMove(start, u.u_IOParam.m_Base, nbytes);

        /* 用传送字节数nbytes更新读写位置 */
        u.u_IOParam.m_Base += nbytes;
        u.u_IOParam.m_Offset += nbytes;
        u.u_IOParam.m_Count -= nbytes;

        bufMgr.Brelse(pBuf); /* 使用完缓存，释放该资源 */
    }
}

void Inode::WriteI()
{
    int lbn;    /* 文件逻辑块号 */
    int bn;     /* lbn对应的物理盘块号 */
    int offset; /* 当前字符块内起始传送位置 */
    int nbytes; /* 传送字节数量 */
    short dev;
    Buf *pBuf;
    User &u = Kernel::Instance().GetUser();
    BufferManager &bufMgr = Kernel::Instance().GetBufferManager();
    DeviceManager &devMgr = Kernel::Instance().GetDeviceManager();

    /* 设置Inode被访问标志位 */
    this->i_flag |= (Inode::IACC | Inode::IUPD);

    /* 对字符设备的访问 */
    if ((this->i_mode & Inode::IFMT) == Inode::IFCHR)
    {
        short major = Utility::GetMajor(this->i_addr[0]);

        devMgr.GetCharDevice(major).Write(this->i_addr[0]);
        return;
    }

    if (0 == u.u_IOParam.m_Count)
    {
        /* 需要读字节数为零，则返回 */
        return;
    }
```

```
    while (User::NOERROR == u.u_error && u.u_IOParam.m_Count != 0)
    {
        lbn = u.u_IOParam.m_Offset / Inode::BLOCK_SIZE;
        offset = u.u_IOParam.m_Offset % Inode::BLOCK_SIZE;
        nbytes = Utility::Min(Inode::BLOCK_SIZE - offset, u.u_IOParam.m_Count);

        if ((this->i_mode & Inode::IFMT) != Inode::IFBLK)
        { /* 普通文件 */

            /* 将逻辑块号lbn转换成物理盘块号bn */
            if ((bn = this->Bmap(lbn)) == 0)
            {
                return;
            }
            dev = this->i_dev;
        }
        else
        { /* 块设备文件，也就是硬盘 */
            dev = this->i_addr[0];
        }

        if (Inode::BLOCK_SIZE == nbytes)
        {
            /* 如果写入数据正好满一个字符块，则为其分配缓存 */
            pBuf = bufMgr.GetBlk(dev, bn);
        }
        else
        {
            /* 写入数据不满一个字符块，先读后写（读出该字符块以保护不需要重写的数据） */
            pBuf = bufMgr.Bread(dev, bn);
        }

        /* 缓存中数据的起始写位置 */
        unsigned char *start = pBuf->b_addr + offset;

        /* 写操作：从用户目标区拷贝数据到缓冲区 */
        Utility::IOMove(u.u_IOParam.m_Base, start, nbytes);

        /* 用传送字节数nbytes更新读写位置 */
        u.u_IOParam.m_Base += nbytes;
        u.u_IOParam.m_Offset += nbytes;
        u.u_IOParam.m_Count -= nbytes;

        if (u.u_error != User::NOERROR) /* 写过程中出错 */
        {
            bufMgr.Brelse(pBuf);
        }
        else if ((u.u_IOParam.m_Offset % Inode::BLOCK_SIZE) == 0) /* 如果写满一个字
符块 */
        {
            /* 以异步方式将字符块写入磁盘，进程不需等待I/O操作结束，可以继续往下执行 */
            bufMgr.Bawrite(pBuf);
        }
        else /* 如果缓冲区未写满 */
        {
            /* 将缓存标记为延迟写，不急于进行I/O操作将字符块输出到磁盘上 */
            bufMgr.Bdwrite(pBuf);
```

```
        }

        /* 普通文件长度增加 */
        if ((this->i_size < u.u_IOParam.m_Offset) && (this->i_mode &
(Inode::IFBLK & Inode::IFCHR)) == 0)
        {
            this->i_size = u.u_IOParam.m_Offset;
        }

        /*
         * 之前过程中读盘可能导致进程切换，在进程睡眠期间当前内存Inode可能
         * 被同步到外存Inode，在此需要重新设置更新标志位。
         * 好像没有必要呀！即使write系统调用没有上锁，iput看到i_count减到0之后才会将内存i节
点同步回磁盘。而这在
         * 文件没有close之前是不会发生的。
         * 我们的系统对write系统调用上锁就更不可能出现这种情况了。
         * 真的想把它去掉。
         */
        this->i_flag |= Inode::IUPD;
    }
}

int Inode::Bmap(int lbn)
{
    Buf *pFirstBuf;
    Buf *pSecondBuf;
    int phyBlkno; /* 转换后的物理盘块号 */
    int *iTable;  /* 用于访问索引盘块中一次间接、两次间接索引表 */
    int index;
    User &u = Kernel::Instance().GetUser();
    BufferManager &bufMgr = Kernel::Instance().GetBufferManager();
    FileSystem &fileSys = Kernel::Instance().GetFileSystem();

    /*
     * Unix V6++的文件索引结构：(小型、大型和巨型文件)
     * (1) i_addr[0] - i_addr[5]为直接索引表，文件长度范围是0 - 6个盘块；
     *
     * (2) i_addr[6] - i_addr[7]存放一次间接索引表所在磁盘块号，每磁盘块
     * 上存放128个文件数据盘块号，此类文件长度范围是7 - (128 * 2 + 6)个盘块；
     *
     * (3) i_addr[8] - i_addr[9]存放二次间接索引表所在磁盘块号，每个二次间接
     * 索引表记录128个一次间接索引表所在磁盘块号，此类文件长度范围是
     * (128 * 2 + 6 ) < size <= (128 * 128 * 2 + 128 * 2 + 6)
     */

    if (lbn >= Inode::HUGE_FILE_BLOCK)
    {
        u.u_error = User::EFBIG;
        return 0;
    }

    if (lbn < 6) /* 如果是小型文件，从基本索引表i_addr[0-5]中获得物理盘块号即可 */
    {
        phyBlkno = this->i_addr[lbn];

        /*
         * 如果该逻辑块号还没有相应的物理盘块号与之对应，则分配一个物理块。
         * 这通常发生在对文件的写入，当写入位置超出文件大小，即对当前
```

```
         * 文件进行扩充写入，就需要分配额外的磁盘块，并为之建立逻辑块号
         * 与物理盘块号之间的映射。
         */
        if (phyBlkno == 0 && (pFirstBuf = fileSys.Alloc(this->i_dev)) != NULL)
        {
            /*
             * 因为后面很可能马上还要用到此处新分配的数据块，所以不急于立刻输出到
             * 磁盘上；而是将缓存标记为延迟写方式，这样可以减少系统的I/O操作。
             */
            bufMgr.Bdwrite(pFirstBuf);
            phyBlkno = pFirstBuf->b_blkno;
            /* 将逻辑块号lbn映射到物理盘块号phyBlkno */
            this->i_addr[lbn] = phyBlkno;
            this->i_flag |= Inode::IUPD;
        }
        /* 找到预读块对应的物理盘块号 */
        Inode::rablock = 0;
        if (lbn <= 4)
        {
            /*
             * i_addr[0] - i_addr[5]为直接索引表。如果预读块对应物理块号可以从
             * 直接索引表中获得，则记录在Inode::rablock中。如果需要额外的I/O开销
             * 读入间接索引块，就显得不太值得了。漂亮!
             */
            Inode::rablock = this->i_addr[lbn + 1];
        }

        return phyBlkno;
    }
    else /* lbn >= 6 大型、巨型文件 */
    {
        /* 计算逻辑块号lbn对应i_addr[]中的索引 */

        if (lbn < Inode::LARGE_FILE_BLOCK) /* 大型文件: 长度介于7 - (128 * 2 + 6)个
盘块之间 */
        {
            index = (lbn - Inode::SMALL_FILE_BLOCK) /
Inode::ADDRESS_PER_INDEX_BLOCK + 6;
        }
        else /* 巨型文件: 长度介于263 - (128 * 128 * 2 + 128 * 2 + 6)个盘块之间 */
        {
            index = (lbn - Inode::LARGE_FILE_BLOCK) /
(Inode::ADDRESS_PER_INDEX_BLOCK * Inode::ADDRESS_PER_INDEX_BLOCK) + 8;
        }

        phyBlkno = this->i_addr[index];
        /* 若该项为零，则表示不存在相应的间接索引表块 */
        if (0 == phyBlkno)
        {
            this->i_flag |= Inode::IUPD;
            /* 分配一空闲盘块存放间接索引表 */
            if ((pFirstBuf = fileSys.Alloc(this->i_dev)) == NULL)
            {
                return 0; /* 分配失败 */
            }
            /* i_addr[index]中记录间接索引表的物理盘块号 */
            this->i_addr[index] = pFirstBuf->b_blkno;
        }
```

```
        else
        {
            /* 读出存储间接索引表的字符块 */
            pFirstBuf = bufMgr.Bread(this->i_dev, phyBlkno);
        }
        /* 获取缓冲区首址 */
        iTable = (int *)pFirstBuf->b_addr;

        if (index >= 8) /* ASSERT: 8 <= index <= 9 */
        {
            /*
             * 对于巨型文件的情况，pFirstBuf中是二次间接索引表，
             * 还需根据逻辑块号，经由二次间接索引表找到一次间接索引表
             */
            index = ((lbn - Inode::LARGE_FILE_BLOCK) /
Inode::ADDRESS_PER_INDEX_BLOCK) % Inode::ADDRESS_PER_INDEX_BLOCK;

            /* iTable指向缓存中的二次间接索引表。该项为零，不存在一次间接索引表 */
            phyBlkno = iTable[index];
            if (0 == phyBlkno)
            {
                if ((pSecondBuf = fileSys.Alloc(this->i_dev)) == NULL)
                {
                    /* 分配一次间接索引表磁盘块失败，释放缓存中的二次间接索引表，然后返回 */
                    bufMgr.Brelse(pFirstBuf);
                    return 0;
                }
                /* 将新分配的一次间接索引表磁盘块号，记入二次间接索引表相应项 */
                iTable[index] = pSecondBuf->b_blkno;
                /* 将更改后的二次间接索引表延迟写方式输出到磁盘 */
                bufMgr.Bdwrite(pFirstBuf);
            }
            else
            {
                /* 释放二次间接索引表占用的缓存，并读入一次间接索引表 */
                bufMgr.Brelse(pFirstBuf);
                pSecondBuf = bufMgr.Bread(this->i_dev, phyBlkno);
            }

            pFirstBuf = pSecondBuf;
            /* 令iTable指向一次间接索引表 */
            iTable = (int *)pSecondBuf->b_addr;
        }

        /* 计算逻辑块号lbn最终位于一次间接索引表中的表项序号index */

        if (lbn < Inode::LARGE_FILE_BLOCK)
        {
            index = (lbn - Inode::SMALL_FILE_BLOCK) %
Inode::ADDRESS_PER_INDEX_BLOCK;
        }
        else
        {
            index = (lbn - Inode::LARGE_FILE_BLOCK) %
Inode::ADDRESS_PER_INDEX_BLOCK;
        }
```

```
        if ((phyBlkno = iTable[index]) == 0 && (pSecondBuf = fileSys.Alloc(this->i_dev)) != NULL)
        {
            /* 将分配到的文件数据盘块号登记在一次间接索引表中 */
            phyBlkno = pSecondBuf->b_blkno;
            iTable[index] = phyBlkno;
            /* 将数据盘块、更改后的一次间接索引表用延迟写方式输出到磁盘 */
            bufMgr.Bdwrite(pSecondBuf);
            bufMgr.Bdwrite(pFirstBuf);
        }
        else
        {
            /* 释放一次间接索引表占用缓存 */
            bufMgr.Brelse(pFirstBuf);
        }
        /* 找到预读块对应的物理盘块号，如果获取预读块号需要额外的一次for间接索引块的IO，不合算，放弃 */
        Inode::rablock = 0;
        if (index + 1 < Inode::ADDRESS_PER_INDEX_BLOCK)
        {
            Inode::rablock = iTable[index + 1];
        }
        return phyBlkno;
    }
}

void Inode::OpenI(int mode)
{
    short dev;
    DeviceManager &devMgr = Kernel::Instance().GetDeviceManager();
    User &u = Kernel::Instance().GetUser();

    /*
     * 对于特殊块设备、字符设备文件，i_addr[]不再是
     * 磁盘块号索引表，addr[0]中存放了设备号dev
     */
    dev = this->i_addr[0];

    /* 提取主设备号 */
    short major = Utility::GetMajor(dev);

    switch (this->i_mode & Inode::IFMT)
    {
    case Inode::IFCHR: /* 字符设备特殊类型文件 */
        if (major >= devMgr.GetNChrDev())
        {
            u.u_error = User::ENXIO; /* no such device */
            return;
        }
        devMgr.GetCharDevice(major).Open(dev, mode);
        break;

    case Inode::IFBLK: /* 块设备特殊类型文件 */
        /* 检查设备号是否超出系统中块设备数量 */
        if (major >= devMgr.GetNBlkDev())
        {
            u.u_error = User::ENXIO; /* no such device */
            return;
```

```
        }
        /* 根据主设备号获取对应的块设备BlockDevice对象引用 */
        BlockDevice &bdev = devMgr.GetBlockDevice(major);
        /* 调用该设备的特定初始化逻辑 */
        bdev.Open(dev, mode);
        break;
    }

    return;
}

void Inode::CloseI(int mode)
{
    short dev;
    DeviceManager &devMgr = Kernel::Instance().GetDeviceManager();

    /* addr[0]中存放了设备号dev */
    dev = this->i_addr[0];

    short major = Utility::GetMajor(dev);

    /* 不再使用该文件,关闭特殊文件 */
    if (this->i_count <= 1)
    {
        switch (this->i_mode & Inode::IFMT)
        {
        case Inode::IFCHR:
            devMgr.GetCharDevice(major).Close(dev, mode);
            break;

        case Inode::IFBLK:
            /* 根据主设备号获取对应的块设备BlockDevice对象引用 */
            BlockDevice &bdev = devMgr.GetBlockDevice(major);
            /* 调用该设备的特定关闭逻辑 */
            bdev.Close(dev, mode);
            break;
        }
    }
}

void Inode::IUpdate(int time)
{
    Buf *pBuf;
    DiskInode dInode;
    FileSystem &filesys = Kernel::Instance().GetFileSystem();
    BufferManager &bufMgr = Kernel::Instance().GetBufferManager();

    /* 当IUPD和IACC标志之一被设置，才需要更新相应DiskInode
     * 目录搜索，不会设置所途径的目录文件的IACC和IUPD标志 */
    if ((this->i_flag & (Inode::IUPD | Inode::IACC)) != 0)
    {
        if (filesys.GetFS(this->i_dev)->s_ronly != 0)
        {
            /* 如果该文件系统只读 */
            return;
        }

        /* 邓蓉的注释：在缓存池中找到包含本i节点（this->i_number）的缓存块
```

```
         * 这是一个上锁的缓存块，本段代码中的Bwrite()在将缓存块写回磁盘后会释放该缓存块。
         * 将该存放该DiskInode的字符块读入缓冲区 */
        pBuf = bufMgr.Bread(this->i_dev, FileSystem::INODE_ZONE_START_SECTOR +
this->i_number / FileSystem::INODE_NUMBER_PER_SECTOR);

        /* 将内存Inode副本中的信息复制到dInode中，然后将dInode覆盖缓存中旧的外存Inode */
        dInode.d_mode = this->i_mode;
        dInode.d_nlink = this->i_nlink;
        dInode.d_uid = this->i_uid;
        dInode.d_gid = this->i_gid;
        dInode.d_size = this->i_size;
        for (int i = 0; i < 10; i++)
        {
            dInode.d_addr[i] = this->i_addr[i];
        }
        if (this->i_flag & Inode::IACC)
        {
            /* 更新最后访问时间 */
            dInode.d_atime = time;
        }
        if (this->i_flag & Inode::IUPD)
        {
            /* 更新最后访问时间 */
            dInode.d_mtime = time;
        }

        /* 将p指向缓存区中旧外存Inode的偏移位置 */
        unsigned char *p = pBuf->b_addr + (this->i_number %
FileSystem::INODE_NUMBER_PER_SECTOR) * sizeof(DiskInode);
        DiskInode *pNode = &dInode;

        /* 用dInode中的新数据覆盖缓存中的旧外存Inode */
        Utility::DWordCopy((int *)pNode, (int *)p, sizeof(DiskInode) /
sizeof(int));

        /* 将缓存写回至磁盘，达到更新旧外存Inode的目的 */
        bufMgr.Bwrite(pBuf);
    }
}

void Inode::ITrunc()
{
    /* 经由磁盘高速缓存读取存放一次间接、两次间接索引表的磁盘块 */
    BufferManager &bm = Kernel::Instance().GetBufferManager();
    /* 获取g_FileSystem对象的引用，执行释放磁盘块的操作 */
    FileSystem &filesys = Kernel::Instance().GetFileSystem();

    /* 如果是字符设备或者块设备则退出 */
    if (this->i_mode & (Inode::IFCHR & Inode::IFBLK))
    {
        return;
    }

    /* 采用FILO方式释放，以尽量使得SuperBlock中记录的空闲盘块号连续。
     *
     * Unix V6++的文件索引结构：(小型、大型和巨型文件)
     * (1) i_addr[0] - i_addr[5]为直接索引表，文件长度范围是0 - 6个盘块；
     *
```

```
     * (2) i_addr[6] - i_addr[7]存放一次间接索引表所在磁盘块号，每磁盘块
     * 上存放128个文件数据盘块号，此类文件长度范围是7 - (128 * 2 + 6)个盘块；
     *
     * (3) i_addr[8] - i_addr[9]存放二次间接索引表所在磁盘块号，每个二次间接
     * 索引表记录128个一次间接索引表所在磁盘块号，此类文件长度范围是
     * (128 * 2 + 6 ) < size <= (128 * 128 * 2 + 128 * 2 + 6)
     */
    for (int i = 9; i >= 0; i--) /* 从i_addr[9]到i_addr[0] */
    {
        /* 如果i_addr[]中第i项存在索引 */
        if (this->i_addr[i] != 0)
        {
            /* 如果是i_addr[]中的一次间接、两次间接索引项 */
            if (i >= 6 && i <= 9)
            {
                /* 将间接索引表读入缓存 */
                Buf *pFirstBuf = bm.Bread(this->i_dev, this->i_addr[i]);
                /* 获取缓冲区首址 */
                int *pFirst = (int *)pFirstBuf->b_addr;

                /* 每张间接索引表记录 512/sizeof(int) = 128个磁盘块号，遍历这全部128个磁
盘块 */
                for (int j = 128 - 1; j >= 0; j--)
                {
                    if (pFirst[j] != 0) /* 如果该项存在索引 */
                    {
                        /*
                         * 如果是两次间接索引表，i_addr[8]或i_addr[9]项，
                         * 那么该字符块记录的是128个一次间接索引表存放的磁盘块号
                         */
                        if (i >= 8 && i <= 9)
                        {
                            Buf *pSecondBuf = bm.Bread(this->i_dev, pFirst[j]);
                            int *pSecond = (int *)pSecondBuf->b_addr;

                            for (int k = 128 - 1; k >= 0; k--)
                            {
                                if (pSecond[k] != 0)
                                {
                                    /* 释放指定的磁盘块 */
                                    filesys.Free(this->i_dev, pSecond[k]);
                                }
                            }
                            /* 缓存使用完毕，释放以便被其它进程使用 */
                            bm.Brelse(pSecondBuf);
                        }
                        filesys.Free(this->i_dev, pFirst[j]);
                    }
                }
                bm.Brelse(pFirstBuf);
            }
            /* 释放索引表本身占用的磁盘块 */
            filesys.Free(this->i_dev, this->i_addr[i]);
            /* 0表示该项不包含索引 */
            this->i_addr[i] = 0;
        }
    }
```

```
    /* 盘块释放完毕，文件大小清零 */
    this->i_size = 0;
    /* 增设IUPD标志位，表示此内存Inode需要同步到相应外存Inode */
    this->i_flag |= Inode::IUPD;
    /* 清大文件标志 和原来的RWXRWXRWX比特*/
    this->i_mode &= ~(Inode::ILARG & Inode::IRWXU & Inode::IRWXG &
Inode::IRWXO);
    this->i_nlink = 1;
}

void Inode::NFrele()
{
    /* 解锁pipe或Inode,并且唤醒相应进程 */
    this->i_flag &= ~Inode::ILOCK;

    if (this->i_flag & Inode::IWANT)
    {
        this->i_flag &= ~Inode::IWANT;
        Kernel::Instance().GetProcessManager().WakeUpAll((unsigned long)this);
    }
}

void Inode::NFlock()
{
    User &u = Kernel::Instance().GetUser();

    while (this->i_flag & Inode::ILOCK)
    {
        this->i_flag |= Inode::IWANT;
        u.u_procp->Sleep((unsigned long)this, ProcessManager::PRIBIO);
    }
    this->i_flag |= Inode::ILOCK;
}

void Inode::Prele()
{
    /* 解锁pipe或Inode,并且唤醒相应进程 */
    this->i_flag &= ~Inode::ILOCK;

    if (this->i_flag & Inode::IWANT)
    {
        this->i_flag &= ~Inode::IWANT;
        Kernel::Instance().GetProcessManager().WakeUpAll((unsigned long)this);
    }
}

void Inode::Plock()
{
    User &u = Kernel::Instance().GetUser();

    while (this->i_flag & Inode::ILOCK)
    {
        this->i_flag |= Inode::IWANT;
        u.u_procp->Sleep((unsigned long)this, ProcessManager::PPIPE);
    }
    this->i_flag |= Inode::ILOCK;
}
```

```
void Inode::Clean()
{
    /*
     * Inode::Clean()特定用于IAlloc()中清空新分配DiskInode的原有数据，
     * 即旧文件信息。Clean()函数中不应当清除i_dev, i_number, i_flag, i_count,
     * 这是属于内存Inode而非DiskInode包含的旧文件信息，而Inode类构造函数需要
     * 将其初始化为无效值。
     */

    // this->i_flag = 0;
    this->i_mode = 0;
    // this->i_count = 0;
    this->i_nlink = 0;
    // this->i_dev = -1;
    // this->i_number = -1;
    this->i_uid = -1;
    this->i_gid = -1;
    this->i_size = 0;
    this->i_lastr = -1;
    for (int i = 0; i < 10; i++)
    {
        this->i_addr[i] = 0;
    }
}

void Inode::ICopy(Buf *bp, int inumber)
{
    DiskInode dInode;
    DiskInode *pNode = &dInode;

    /* 将p指向缓存区中编号为inumber外存Inode的偏移位置 */
    unsigned char *p = bp->b_addr + (inumber %
FileSystem::INODE_NUMBER_PER_SECTOR) * sizeof(DiskInode);
    /* 将缓存中外存Inode数据拷贝到临时变量dInode中，按4字节拷贝 */
    Utility::DWordCopy((int *)p, (int *)pNode, sizeof(DiskInode) / sizeof(int));

    /* 将外存Inode变量dInode中信息复制到内存Inode中 */
    this->i_mode = dInode.d_mode;
    this->i_nlink = dInode.d_nlink;
    this->i_uid = dInode.d_uid;
    this->i_gid = dInode.d_gid;
    this->i_size = dInode.d_size;
    for (int i = 0; i < 10; i++)
    {
        this->i_addr[i] = dInode.d_addr[i];
    }
}

/*============================class DiskInode=================================*/

DiskInode::DiskInode()
{
    /*
     * 如果DiskInode没有构造函数，会发生如下较难察觉的错误：
     * DiskInode作为局部变量占据函数Stack Frame中的内存空间，但是
     * 这段空间没有被正确初始化，仍旧保留着先前栈内容，由于并不是
     * DiskInode所有字段都会被更新，将DiskInode写回到磁盘上时，可能
     * 将先前栈内容一同写回，导致写回结果出现莫名其妙的数据。
```

```
     */
    this->d_mode = 0;
    this->d_nlink = 0;
    this->d_uid = -1;
    this->d_gid = -1;
    this->d_size = 0;
    for (int i = 0; i < 10; i++)
    {
        this->d_addr[i] = 0;
    }
    this->d_atime = 0;
    this->d_mtime = 0;
}

DiskInode::~DiskInode()
{
    // nothing to do here
}
```

OpenFileManager.h

```
#ifndef OPEN_FILE_MANAGER_H
#define OPEN_FILE_MANAGER_H

#include "INode.h"
#include "File.h"
#include "FileSystem.h"

/* Forward Declaration */
class OpenFileTable;
class InodeTable;

/* 以下2个对象实例定义在OpenFileManager.cpp文件中 */
extern InodeTable g_InodeTable;
extern OpenFileTable g_OpenFileTable;

/*
 * 打开文件管理类(OpenFileManager)负责
 * 内核中对打开文件机构的管理，为进程
 * 打开文件建立内核数据结构之间的勾连
 * 关系。
 * 勾连关系指进程u区中打开文件描述符指向
 * 打开文件表中的File打开文件控制结构，
 * 以及从File结构指向文件对应的内存Inode。
 */
class OpenFileTable
{
public:
    /* static consts */
    // static const int NINODE  = 100;  /* 内存Inode的数量 */
    static const int NFILE = 100; /* 打开文件控制块File结构的数量 */

    /* Functions */
public:
    /* Constructors */
    OpenFileTable();
```

```
    /* Destructors */
    ~OpenFileTable();

    // /*
    // * @comment 根据用户系统调用提供的文件描述符参数fd，
    // * 找到对应的打开文件控制块File结构
    // */
    // File* GetF(int fd);
    /*
     * @comment 在系统打开文件表中分配一个空闲的File结构
     */
    File *FAlloc();
    /*
     * @comment 对打开文件控制块File结构的引用计数f_count减1，
     * 若引用计数f_count为0，则释放File结构。
     */
    void CloseF(File *pFile);

    /* Members */
public:
    File m_File[NFILE]; /* 系统打开文件表，为所有进程共享，进程打开文件描述符表
                           中包含指向打开文件表中对应File结构的指针。*/
};

/*
 * 内存Inode表(class InodeTable)
 * 负责内存Inode的分配和释放。
 */
class InodeTable
{
    /* static consts */
public:
    static const int NINODE = 100; /* 内存Inode的数量 */

    /* Functions */
public:
    /* Constructors */
    InodeTable();
    /* Destructors */
    ~InodeTable();

    /*
     * @comment 初始化对g_FileSystem对象的引用
     */
    void Initialize();
    /*
     * @comment 根据指定设备号dev，外存Inode编号获取对应
     * Inode。如果该Inode已经在内存中，对其上锁并返回该内存Inode，
     * 如果不在内存中，则将其读入内存后上锁并返回该内存Inode
     */
    Inode *IGet(short dev, int inumber);
    /*
     * @comment 减少该内存Inode的引用计数，如果此Inode已经没有目录项指向它，
     * 且无进程引用该Inode，则释放此文件占用的磁盘块。
     */
    void IPut(Inode *pNode);

    /*
```

```
     * @comment 将所有被修改过的内存Inode更新到对应外存Inode中
     */
    void UpdateInodeTable();

    /*
     * @comment 检查设备dev上编号为inumber的外存inode是否有内存拷贝，
     * 如果有则返回该内存Inode在内存Inode表中的索引
     */
    int IsLoaded(short dev, int inumber);
    /*
     * @comment 在内存Inode表中寻找一个空闲的内存Inode
     */
    Inode *GetFreeInode();

    /* Members */
public:
    Inode m_Inode[NINODE]; /* 内存Inode数组，每个打开文件都会占用一个内存Inode */

    FileSystem *m_FileSystem; /* 对全局对象g_FileSystem的引用 */
};

#endif
```

OpenFileManager.cpp

```
#include "OpenFileManager.h"
#include "Kernel.h"
#include "TimeInterrupt.h"
#include "Video.h"

/*==============================class
OpenFileTable====================================*/
/* 系统全局打开文件表对象实例的定义 */
OpenFileTable g_OpenFileTable;

OpenFileTable::OpenFileTable()
{
    // nothing to do here
}

OpenFileTable::~OpenFileTable()
{
    // nothing to do here
}

/*作用：进程打开文件描述符表中找的空闲项  之 下标  写入 u_ar0[EAX]*/
File *OpenFileTable::FAlloc()
{
    int fd;
    User &u = Kernel::Instance().GetUser();

    /* 在进程打开文件描述符表中获取一个空闲项 */
    fd = u.u_ofiles.AllocFreeSlot();

    if (fd < 0) /* 如果寻找空闲项失败 */
    {
```

```
        return NULL;
    }

    for (int i = 0; i < OpenFileTable::NFILE; i++)
    {
        /* f_count==0表示该项空闲 */
        if (this->m_File[i].f_count == 0)
        {
            /* 建立描述符和File结构的勾连关系 */
            u.u_ofiles.SetF(fd, &this->m_File[i]);
            /* 增加对file结构的引用计数 */
            this->m_File[i].f_count++;
            /* 清空文件读、写位置 */
            this->m_File[i].f_offset = 0;
            return (&this->m_File[i]);
        }
    }

    Diagnose::Write("No Free File Struct\n");
    u.u_error = User::ENFILE;
    return NULL;
}

void OpenFileTable::CloseF(File *pFile)
{
    Inode *pNode;
    ProcessManager &procMgr = Kernel::Instance().GetProcessManager();

    /* 管道类型 */
    if (pFile->f_flag & File::FPIPE)
    {
        pNode = pFile->f_inode;
        pNode->i_mode &= ~(Inode::IREAD | Inode::IWRITE);
        procMgr.WakeUpAll((unsigned long)(pNode + 1));
        procMgr.WakeUpAll((unsigned long)(pNode + 2));
    }

    if (pFile->f_count <= 1)
    {
        /*
         * 如果当前进程是最后一个引用该文件的进程，
         * 对特殊块设备、字符设备文件调用相应的关闭函数
         */
        pFile->f_inode->CloseI(pFile->f_flag & File::FWRITE);
        g_InodeTable.IPut(pFile->f_inode);
    }

    /* 引用当前File的进程数减1 */
    pFile->f_count--;
}

/*==============================class
InodeTable====================================*/
/*  定义内存Inode表的实例 */
InodeTable g_InodeTable;

InodeTable::InodeTable()
{
```

```
    // nothing to do here
}

InodeTable::~InodeTable()
{
    // nothing to do here
}

void InodeTable::Initialize()
{
    /* 获取对g_FileSystem的引用 */
    this->m_FileSystem = &Kernel::Instance().GetFileSystem();
}

Inode *InodeTable::IGet(short dev, int inumber)
{
    Inode *pInode;
    User &u = Kernel::Instance().GetUser();

    while (true)
    {
        /* 检查指定设备dev中编号为inumber的外存Inode是否有内存拷贝 */
        int index = this->IsLoaded(dev, inumber);
        if (index >= 0) /* 找到内存拷贝 */
        {
            pInode = &(this->m_Inode[index]);
            /* 如果该内存Inode被上锁 */
            if (pInode->i_flag & Inode::ILOCK)
            {
                /* 增设IWANT标志，然后睡眠 */
                pInode->i_flag |= Inode::IWANT;

                u.u_procp->Sleep((unsigned long)&pInode, ProcessManager::PINOD);

                /* 回到while循环，需要重新搜索，因为该内存Inode可能已经失效 */
                continue;
            }

            /* 如果该内存Inode用于连接子文件系统，查找该Inode对应的Mount装配块 */
            if (pInode->i_flag & Inode::IMOUNT)
            {
                Mount *pMount = this->m_FileSystem->GetMount(pInode);
                if (NULL == pMount)
                {
                    /* 没有找到 */
                    Utility::Panic("No Mount Tab...");
                }
                else
                {
                    /* 将参数设为子文件系统设备号、根目录Inode编号 */
                    dev = pMount->m_dev;
                    inumber = FileSystem::ROOTINO;
                    /* 回到while循环，以新dev，inumber值重新搜索 */
                    continue;
                }
            }

            /*
```

```
                * 程序执行到这里表示：内存Inode高速缓存中找到相应内存Inode，
                * 增加其引用计数，增设ILOCK标志并返回之
                */
            pInode->i_count++;
            pInode->i_flag |= Inode::ILOCK;
            return pInode;
        }
        else /* 没有Inode的内存拷贝，则分配一个空闲内存Inode */
        {
            pInode = this->GetFreeInode();
            /* 若内存Inode表已满，分配空闲Inode失败 */
            if (NULL == pInode)
            {
                Diagnose::Write("Inode Table Overflow !\n");
                u.u_error = User::ENFILE;
                return NULL;
            }
            else /* 分配空闲Inode成功，将外存Inode读入新分配的内存Inode */
            {
                /* 设置新的设备号、外存Inode编号，增加引用计数，对索引节点上锁 */
                pInode->i_dev = dev;
                pInode->i_number = inumber;
                pInode->i_flag = Inode::ILOCK;
                pInode->i_count++;
                pInode->i_lastr = -1;

                BufferManager &bm = Kernel::Instance().GetBufferManager();
                /* 将该外存Inode读入缓冲区 */
                Buf *pBuf = bm.Bread(dev, FileSystem::INODE_ZONE_START_SECTOR +
inumber / FileSystem::INODE_NUMBER_PER_SECTOR);

                /* 如果发生I/O错误 */
                if (pBuf->b_flags & Buf::B_ERROR)
                {
                    /* 释放缓存 */
                    bm.Brelse(pBuf);
                    /* 释放占据的内存Inode */
                    this->IPut(pInode);
                    return NULL;
                }

                /* 将缓冲区中的外存Inode信息拷贝到新分配的内存Inode中 */
                pInode->ICopy(pBuf, inumber);
                /* 释放缓存 */
                bm.Brelse(pBuf);
                return pInode;
            }
        }
    }
    return NULL; /* GCC likes it! */
}

/* close文件时会调用Iput
 *      主要做的操作：内存i节点计数 i_count--；若为0，释放内存 i节点、若有改动写回磁盘
 * 搜索文件途径的所有目录文件，搜索经过后都会Iput其内存i节点。路径名的倒数第2个路径分量一定是个
 *   目录文件，如果是在其中创建新文件、或是删除一个已有文件；再如果是在其中创建删除子目录。那么
 *      必须将这个目录文件所对应的内存 i节点写回磁盘。
 *      这个目录文件无论是否经历过更改，我们必须将它的i节点写回磁盘。
```

```
 * */
void InodeTable::IPut(Inode *pNode)
{
    /* 当前进程为引用该内存Inode的唯一进程，且准备释放该内存Inode */
    if (pNode->i_count == 1)
    {
        /*
         * 上锁，因为在整个释放过程中可能因为磁盘操作而使得该进程睡眠，
         * 此时有可能另一个进程会对该内存Inode进行操作，这将有可能导致错误。
         */
        pNode->i_flag |= Inode::ILOCK;

        /* 该文件已经没有目录路径指向它 */
        if (pNode->i_nlink <= 0)
        {
            /* 释放该文件占据的数据盘块 */
            pNode->ITrunc();
            pNode->i_mode = 0;
            /* 释放对应的外存Inode */
            this->m_FileSystem->IFree(pNode->i_dev, pNode->i_number);
        }

        /* 更新外存Inode信息 */
        pNode->IUpdate(Time::time);

        /* 解锁内存Inode，并且唤醒等待进程 */
        pNode->Prele();
        /* 清除内存Inode的所有标志位 */
        pNode->i_flag = 0;
        /* 这是内存inode空闲的标志之一，另一个是i_count == 0 */
        pNode->i_number = -1;
    }

    /* 减少内存Inode的引用计数，唤醒等待进程 */
    pNode->i_count--;
    pNode->Prele();
}

void InodeTable::UpdateInodeTable()
{
    for (int i = 0; i < InodeTable::NINODE; i++)
    {
        /*
         * 如果Inode对象没有被上锁，即当前未被其它进程使用，可以同步到外存Inode；
         * 并且count不等于0，count == 0意味着该内存Inode未被任何打开文件引用，无需同步。
         */
        if ((this->m_Inode[i].i_flag & Inode::ILOCK) == 0 && this-
>m_Inode[i].i_count != 0)
        {
            /* 将内存Inode上锁后同步到外存Inode */
            this->m_Inode[i].i_flag |= Inode::ILOCK;
            this->m_Inode[i].IUpdate(Time::time);

            /* 对内存Inode解锁 */
            this->m_Inode[i].Prele();
        }
    }
}
```

```
int InodeTable::IsLoaded(short dev, int inumber)
{
    /* 寻找指定外存Inode的内存拷贝 */
    for (int i = 0; i < InodeTable::NINODE; i++)
    {
        if (this->m_Inode[i].i_dev == dev && this->m_Inode[i].i_number ==
inumber && this->m_Inode[i].i_count != 0)
        {
            return i;
        }
    }
    return -1;
}

Inode *InodeTable::GetFreeInode()
{
    for (int i = 0; i < InodeTable::NINODE; i++)
    {
        /* 如果该内存Inode引用计数为零，则该Inode表示空闲 */
        if (this->m_Inode[i].i_count == 0)
        {
            return &(this->m_Inode[i]);
        }
    }
    return NULL; /* 寻找失败 */
}
```

dev

BufferManager.h

```
#ifndef BUFFER_MANAGER_H
#define BUFFER_MANAGER_H

#include "Buf.h"
#include "DeviceManager.h"

class BufferManager
{
public:
    /* static const member */
    static const int NBUF = 15;          /* 缓存控制块、缓冲区的数量 */
    static const int BUFFER_SIZE = 512; /* 缓冲区大小。 以字节为单位 */

public:
    BufferManager();
    ~BufferManager();

    void Initialize(); /* 缓存控制块队列的初始化。将缓存控制块中b_addr指向相应缓冲区首地
址。*/
```

```
    Buf *GetBlk(short dev, int blkno); /* 申请一块缓存，用于读写设备dev上的字符块
blkno。*/
    void Brelse(Buf *bp);              /* 释放缓存控制块buf */
    void IOWait(Buf *bp);              /* 同步方式I/O，等待I/O操作结束 */
    void IODone(Buf *bp);              /* I/O操作结束善后处理 */

    Buf *Bread(short dev, int blkno);                /* 读一个磁盘块。dev为主、次设备
号，blkno为目标磁盘块逻辑块号。 */
    Buf *Breada(short adev, int blkno, int rablkno); /* 读一个磁盘块，带有预读方式。
                                                      * adev为主、次设备号。blkno为
目标磁盘块逻辑块号，同步方式读blkno。
                                                      * rablkno为预读磁盘块逻辑块
号，异步方式读rablkno。 */
    void Bwrite(Buf *bp);                            /* 写一个磁盘块 */
    void Bdwrite(Buf *bp);                           /* 延迟写磁盘块 */
    void Bawrite(Buf *bp);                           /* 异步写磁盘块 */

    void ClrBuf(Buf *bp);   /* 清空缓冲区内容 */
    void Bflush(short dev); /* 将dev指定设备队列中延迟写的缓存全部输出到磁盘 */
    bool Swap(int blkno, unsigned long addr, int count, enum Buf::BufFlag flag);
    /* Swap I/O 用于进程图像在内存和盘交换区之间传输
     * blkno: 交换区中盘块号；addr:  进程图像(传送部分)内存起始地址；
     * count: 进行传输字节数，byte为单位；传输方向flag: 内存->交换区 or 交换区->内存。 */
    Buf &GetSwapBuf();   /* 获取进程图像传送请求块Buf对象引用 */
    Buf &GetBFreeList(); /* 获取自由缓存队列控制块Buf对象引用 */

private:
    void GetError(Buf *bp);             /* 获取I/O操作中发生的错误信息 */
    void NotAvail(Buf *bp);             /* 从自由队列中摘下指定的缓存控制块buf */
    Buf *InCore(short adev, int blkno); /* 检查指定字符块是否已在缓存中 */

private:
    Buf bFreeList;                              /* 自由缓存队列控制块 */
    Buf SwBuf;                                  /* 进程图像传送请求块 */
    Buf m_Buf[NBUF];                            /* 缓存控制块数组 */
    unsigned char Buffer[NBUF][BUFFER_SIZE]; /* 缓冲区数组 */

    DeviceManager *m_DeviceManager; /* 指向设备管理模块全局对象 */
};

#endif
```

BufferManager.cpp

```
#include "BufferManager.h"
#include "Kernel.h"

BufferManager::BufferManager()
{
    // nothing to do here
}

BufferManager::~BufferManager()
{
```

```
    // nothing to do here
}

void BufferManager::Initialize()
{
    int i;
    Buf *bp;

    this->bFreeList.b_forw = this->bFreeList.b_back = &(this->bFreeList);
    this->bFreeList.av_forw = this->bFreeList.av_back = &(this->bFreeList);

    for (i = 0; i < NBUF; i++)
    {
        bp = &(this->m_Buf[i]);
        bp->b_dev = -1;
        bp->b_addr = this->Buffer[i];
        /* 初始化NODEV队列 */
        bp->b_back = &(this->bFreeList);
        bp->b_forw = this->bFreeList.b_forw;
        this->bFreeList.b_forw->b_back = bp;
        this->bFreeList.b_forw = bp;
        /* 初始化自由队列 */
        bp->b_flags = Buf::B_BUSY;
        Brelse(bp);
    }
    this->m_DeviceManager = &Kernel::Instance().GetDeviceManager();
    return;
}

Buf *BufferManager::GetBlk(short dev, int blkno)
{
    Buf *bp;
    Devtab *dp;
    User &u = Kernel::Instance().GetUser();

    /* 如果主设备号超出了系统中块设备数量 */
    if (Utility::GetMajor(dev) >= this->m_DeviceManager->GetNBlkDev())
    {
        Utility::Panic("nblkdev: There doesn't exist the device");
    }

    /*
     * 如果设备队列中已经存在相应缓存，则返回该缓存；
     * 否则从自由队列中分配新的缓存用于字符块读写。
     */
loop:
    /* 表示请求NODEV设备中字符块 */
    if (dev < 0)
    {
        dp = (Devtab *)(&this->bFreeList);
    }
    else
    {
        short major = Utility::GetMajor(dev);
        /* 根据主设备号获得块设备表 */
        dp = this->m_DeviceManager->GetBlockDevice(major).d_tab;

        if (dp == NULL)
```

```
        {
            Utility::Panic("Null devtab!");
        }
        /* 首先在该设备队列中搜索是否有相应的缓存 */
        for (bp = dp->b_forw; bp != (Buf *)dp; bp = bp->b_forw)
        {
            /* 不是要找的缓存，则继续 */
            if (bp->b_blkno != blkno || bp->b_dev != dev)
                continue;

            /*
             * 临界区之所以要从这里开始，而不是从上面的for循环开始。
             * 主要是因为，中断服务程序并不会去修改块设备表中的
             * 设备buf队列(b_forw)，所以不会引起冲突。
             */
            X86Assembly::CLI();
            if (bp->b_flags & Buf::B_BUSY)
            {
                bp->b_flags |= Buf::B_WANTED;
                u.u_procp->Sleep((unsigned long)bp, ProcessManager::PRIBIO);
                X86Assembly::STI();
                goto loop;
            }
            X86Assembly::STI();
            /* 从自由队列中抽取出来 */
            this->NotAvail(bp);
            return bp;
        }
    } // end of else

    X86Assembly::CLI();
    /* 如果自由队列为空 */
    if (this->bFreeList.av_forw == &this->bFreeList)
    {
        this->bFreeList.b_flags |= Buf::B_WANTED;
        u.u_procp->Sleep((unsigned long)&this->bFreeList,
ProcessManager::PRIBIO);
        X86Assembly::STI();
        goto loop;
    }
    X86Assembly::STI();

    /* 取自由队列第一个空闲块 */
    bp = this->bFreeList.av_forw;
    this->NotAvail(bp);

    /* 如果该字符块是延迟写，将其异步写到磁盘上 */
    if (bp->b_flags & Buf::B_DELWRI)
    {
        bp->b_flags |= Buf::B_ASYNC;
        this->Bwrite(bp);
        goto loop;
    }
    /* 注意: 这里清除了所有其他位，只设了B_BUSY */
    bp->b_flags = Buf::B_BUSY;

    /* 从原设备队列中抽出 */
    bp->b_back->b_forw = bp->b_forw;
```

```
    bp->b_forw->b_back = bp->b_back;
    /* 加入新的设备队列 */
    bp->b_forw = dp->b_forw;
    bp->b_back = (Buf *)dp;
    dp->b_forw->b_back = bp;
    dp->b_forw = bp;

    bp->b_dev = dev;
    bp->b_blkno = blkno;
    return bp;
}

void BufferManager::Brelse(Buf *bp)
{
    ProcessManager &procMgr = Kernel::Instance().GetProcessManager();

    if (bp->b_flags & Buf::B_WANTED)
    {
        procMgr.WakeUpAll((unsigned long)bp);
    }

    /* 如果有进程正在等待分配自由队列中的缓存，则唤醒相应进程 */
    if (this->bFreeList.b_flags & Buf::B_WANTED)
    {
        /* 清楚B_WANTED标志，表示已有空闲缓存 */
        this->bFreeList.b_flags &= (~Buf::B_WANTED);
        procMgr.WakeUpAll((unsigned long)&this->bFreeList);
    }

    /* 如果有错误发生，则将次设备号改掉，
     * 避免后续进程误用缓冲区中的错误数据。
     */
    if (bp->b_flags & Buf::B_ERROR)
    {
        Utility::SetMinor(bp->b_dev, -1);
    }

    /* 临界资源，比如：在同步读末期会调用这个函数，
     * 此时很有可能会产生磁盘中断，同样会调用这个函数。
     */
    X86Assembly::CLI(); /* spl6();  UNIX V6的做法 */

    /* 注意以下操作并没有清除B_DELWRI、B_WRITE、B_READ、B_DONE标志
     * B_DELWRI表示虽然将该控制块释放到自由队列里面，但是有可能还没有些到磁盘上。
     * B_DONE则是指该缓存的内容正确地反映了存储在或应存储在磁盘上的信息
     */
    bp->b_flags &= ~(Buf::B_WANTED | Buf::B_BUSY | Buf::B_ASYNC);
    (this->bFreeList.av_back)->av_forw = bp;
    bp->av_back = this->bFreeList.av_back;
    bp->av_forw = &(this->bFreeList);
    this->bFreeList.av_back = bp;

    X86Assembly::STI();
    return;
}

void BufferManager::IOWait(Buf *bp)
{
```

```
    User &u = Kernel::Instance().GetUser();

    /* 这里涉及到临界区
     * 因为在执行这段程序的时候，很有可能出现硬盘中断，
     * 在硬盘中断中，将会修改B_DONE如果此时已经进入循环
     * 则将使得改进程永远睡眠
     */
    X86Assembly::CLI();
    while ((bp->b_flags & Buf::B_DONE) == 0)
    {
        u.u_procp->Sleep((unsigned long)bp, ProcessManager::PRIBIO);
    }
    X86Assembly::STI();

    this->GetError(bp);
    return;
}

void BufferManager::IODone(Buf *bp)
{
    /* 置上I/O完成标志 */
    bp->b_flags |= Buf::B_DONE;
    if (bp->b_flags & Buf::B_ASYNC)
    {
        /* 如果是异步操作,立即释放缓存块 */
        this->Brelse(bp);
    }
    else
    {
        /* 清除B_WANTED标志位 */
        bp->b_flags &= (~Buf::B_WANTED);
        Kernel::Instance().GetProcessManager().WakeUpAll((unsigned long)bp);
    }
    return;
}

Buf *BufferManager::Bread(short dev, int blkno)
{
    Buf *bp;
    /* 根据设备号，字符块号申请缓存 */
    bp = this->GetBlk(dev, blkno);
    /* 如果在设备队列中找到所需缓存，即B_DONE已设置，就不需进行I/O操作 */
    if (bp->b_flags & Buf::B_DONE)
    {
        return bp;
    }
    /* 没有找到相应缓存，构成I/O读请求块 */
    bp->b_flags |= Buf::B_READ;
    bp->b_wcount = BufferManager::BUFFER_SIZE;

    /*
     * 将I/O请求块送入相应设备I/O请求队列，如无其它I/O请求，则将立即执行本次I/O请求；
     * 否则等待当前I/O请求执行完毕后，由中断处理程序启动执行此请求。
     * 注：Strategy()函数将I/O请求块送入设备请求队列后，不等I/O操作执行完毕，就直接返回。
     */
    this->m_DeviceManager->GetBlockDevice(Utility::GetMajor(dev)).Strategy(bp);
    /* 同步读，等待I/O操作结束 */
    this->IOWait(bp);
```

```
    return bp;
}

Buf *BufferManager::Breada(short adev, int blkno, int rablkno)
{
    Buf *bp = NULL;                          /* 非预读字符块的缓存Buf */
    Buf *abp;                                /* 预读字符块的缓存Buf */
    short major = Utility::GetMajor(adev); /* 主设备号 */

    /* 当前字符块是否已在设备Buf队列中 */
    if (!this->InCore(adev, blkno))
    {
        bp = this->GetBlk(adev, blkno); /* 若没找到，GetBlk()分配缓存 */

        /* 如果分配到缓存的B_DONE标志已设置，意味着在InCore()检查之后，
         * 其它进程碰巧读取同一字符块，因而在GetBlk()中再次搜索的时候
         * 发现该字符块已在设备Buf队列缓冲区中，本进程重用该缓存。*/
        if ((bp->b_flags & Buf::B_DONE) == 0)
        {
            /* 构成读请求块 */
            bp->b_flags |= Buf::B_READ;
            bp->b_wcount = BufferManager::BUFFER_SIZE;
            /* 驱动块设备进行I/O操作 */
            this->m_DeviceManager->GetBlockDevice(major).Strategy(bp);
        }
    }
    else
        /*UNIX V6没这条语句。加入的原因：如果当前块在缓存池中，放弃预读
         * 这是因为，预读的价值在于利用当前块和预读块磁盘位置大概率邻近的事实，
         * 用预读操作减少磁臂移动次数提高有效磁盘读带宽。若当前块在缓存池，磁头不一定在当前块所
在的位置，
         * 此时预读，收益有限*/
        rablkno = 0;

    /* 预读操作有2点值得注意：
     * 1、rablkno为0，说明UNIX打算放弃预读。
     *       这是开销与收益的权衡
     * 2、若预读字符块在设备Buf队列中，针对预读块的操作已经成功
     *       这是因为：
     *       作为预读块，并非是进程此次读盘的目的。
     *       所以如果不及时释放，将使得预读块一直得不到释放。
     *       而将其释放它依然存在在设备队列中，如果在短时间内
     *       使用这一块，那么依然可以找到。
     * */
    if (rablkno && !this->InCore(adev, rablkno))
    {
        abp = this->GetBlk(adev, rablkno); /* 若没找到，GetBlk()分配缓存 */

        /* 检查B_DONE标志位，理由同上。 */
        if (abp->b_flags & Buf::B_DONE)
        {
            /* 预读字符块已在缓存中，释放占用的缓存。
             * 因为进程未必后面一定会使用预读的字符块，
             * 也就不会去释放该缓存，有可能导致缓存资源
             * 的长时间占用。
             */
            this->Brelse(abp);
        }
```

```
        else
        {
            /* 异步读预读字符块 */
            abp->b_flags |= (Buf::B_READ | Buf::B_ASYNC);
            abp->b_wcount = BufferManager::BUFFER_SIZE;
            /* 驱动块设备进行I/O操作 */
            this->m_DeviceManager->GetBlockDevice(major).Strategy(abp);
        }
    }

    /* bp == NULL意味着InCore()函数检查时刻，非预读块在设备队列中，
     * 但是InCore()只是“检查”，并不“摘取”。经过一段时间执行到此处，
     * 有可能该字符块已经重新分配它用。
     * 因而重新调用Bread()重读字符块，Bread()中调用GetBlk()将字符块“摘取”
     * 过来。短时间内该字符块仍在设备队列中，所以此处Bread()一般也就是将
     * 缓存重用，而不必重新执行一次I/O读取操作。
     */
    if (NULL == bp)
    {
        return (this->Bread(adev, blkno));
    }

    /* InCore()函数检查时刻未找到非预读字符块，等待I/O操作完成 */
    this->IOWait(bp);
    return bp;
}

void BufferManager::Bwrite(Buf *bp)
{
    unsigned int flags;

    flags = bp->b_flags;
    bp->b_flags &= ~(Buf::B_READ | Buf::B_DONE | Buf::B_ERROR | Buf::B_DELWRI);
    bp->b_wcount = BufferManager::BUFFER_SIZE; /* 512字节 */

    this->m_DeviceManager->GetBlockDevice(Utility::GetMajor(bp-
>b_dev)).Strategy(bp);

    if ((flags & Buf::B_ASYNC) == 0)
    {
        /* 同步写，需要等待I/O操作结束 */
        this->IOWait(bp);
        this->Brelse(bp);
    }
    else if ((flags & Buf::B_DELWRI) == 0)
    {
        /*
         * 如果不是延迟写，则检查错误；否则不检查。
         * 这是因为如果延迟写，则很有可能当前进程不是
         * 操作这一缓存块的进程，而在GetError()主要是
         * 给当前进程附上错误标志。
         */
        this->GetError(bp);
    }
    return;
}

void BufferManager::Bdwrite(Buf *bp)
```

```
{
    /* 置上B_DONE允许其它进程使用该磁盘块内容 */
    bp->b_flags |= (Buf::B_DELWRI | Buf::B_DONE);
    this->Brelse(bp);
    return;
}

void BufferManager::Bawrite(Buf *bp)
{
    /* 标记为异步写 */
    bp->b_flags |= Buf::B_ASYNC;
    this->Bwrite(bp);
    return;
}

void BufferManager::ClrBuf(Buf *bp)
{
    int *pInt = (int *)bp->b_addr;

    /* 将缓冲区中数据清零 */
    for (unsigned int i = 0; i < BufferManager::BUFFER_SIZE / sizeof(int); i++)
    {
        pInt[i] = 0;
    }
    return;
}

void BufferManager::Bflush(short dev)
{
    Buf *bp;
    /* 注意：这里之所以要在搜索到一个块之后重新开始搜索，
     * 因为在bwite()进入到驱动程序中时有开中断的操作，所以
     * 等到bwrite执行完成后，CPU已处于开中断状态，所以很
     * 有可能在这期间产生磁盘中断，使得bfreelist队列出现变化，
     * 如果这里继续往下搜索，而不是重新开始搜索那么很可能在
     * 操作bfreelist队列的时候出现错误。
     */
loop:
    X86Assembly::CLI();
    for (bp = this->bFreeList.av_forw; bp != &(this->bFreeList); bp = bp->av_forw)
    {
        /* 找出自由队列中所有延迟写的块 */
        if ((bp->b_flags & Buf::B_DELWRI) && (dev == DeviceManager::NODEV || dev == bp->b_dev))
        {
            bp->b_flags |= Buf::B_ASYNC;
            this->NotAvail(bp);
            this->Bwrite(bp);
            goto loop;
        }
    }
    X86Assembly::STI();
    return;
}

bool BufferManager::Swap(int blkno, unsigned long addr, int count, enum Buf::BufFlag flag)
```

```
{
    User &u = Kernel::Instance().GetUser();

    X86Assembly::CLI();

    /* swbuf正在被其它进程使用，则睡眠等待 */
    while (this->SwBuf.b_flags & Buf::B_BUSY)
    {
        this->SwBuf.b_flags |= Buf::B_WANTED;
        u.u_procp->Sleep((unsigned long)&SwBuf, ProcessManager::PSWP);
    }

    this->SwBuf.b_flags = Buf::B_BUSY | flag;
    this->SwBuf.b_dev = DeviceManager::ROOTDEV;
    this->SwBuf.b_wcount = count;
    this->SwBuf.b_blkno = blkno;
    /* b_addr指向要传输部分的内存首地址 */
    this->SwBuf.b_addr = (unsigned char *)addr;
    this->m_DeviceManager->GetBlockDevice(Utility::GetMajor(this->SwBuf.b_dev)).Strategy(&this->SwBuf);

    /* 关中断进行B_DONE标志的检查 */
    X86Assembly::CLI();
    /* 这里Sleep()等同于同步I/O中IOWait()的效果 */
    while ((this->SwBuf.b_flags & Buf::B_DONE) == 0)
    {
        u.u_procp->Sleep((unsigned long)&SwBuf, ProcessManager::PSWP);
    }

    /* 这里Wakeup()等同于Brelse()的效果 */
    if (this->SwBuf.b_flags & Buf::B_WANTED)
    {
        Kernel::Instance().GetProcessManager().WakeUpAll((unsigned long)&SwBuf);
    }
    X86Assembly::STI();
    this->SwBuf.b_flags &= ~(Buf::B_BUSY | Buf::B_WANTED);

    if (this->SwBuf.b_flags & Buf::B_ERROR)
    {
        return false;
    }
    return true;
}

void BufferManager::GetError(Buf *bp)
{
    User &u = Kernel::Instance().GetUser();

    if (bp->b_flags & Buf::B_ERROR)
    {
        u.u_error = User::EIO;
    }
    return;
}

void BufferManager::NotAvail(Buf *bp)
{
    X86Assembly::CLI(); /* spl6();  UNIX V6的做法 */
```

```
    /* 从自由队列中取出 */
    bp->av_back->av_forw = bp->av_forw;
    bp->av_forw->av_back = bp->av_back;
    /* 设置B_BUSY标志 */
    bp->b_flags |= Buf::B_BUSY;
    X86Assembly::STI();
    return;
}

Buf *BufferManager::InCore(short adev, int blkno)
{
    Buf *bp;
    Devtab *dp;
    short major = Utility::GetMajor(adev);

    dp = this->m_DeviceManager->GetBlockDevice(major).d_tab;
    for (bp = dp->b_forw; bp != (Buf *)dp; bp = bp->b_forw)
    {
        if (bp->b_blkno == blkno && bp->b_dev == adev)
            return bp;
    }
    return NULL;
}

Buf &BufferManager::GetSwapBuf()
{
    return this->SwBuf;
}

Buf &BufferManager::GetBFreeList()
{
    return this->bFreeList;
}
```