华中农业大学
HUAZHONG AGRICULTURAL UNIVERSITY

综合实训指导书

题目	**Ext2 文件系统的仿真**
版本	**0.1 版本**
院系	**信息学院计算机科学系**
编写时间	2015.4
编写教师	任继平

实训目标

系统级软件的理解和开发是计算机专业学生应具备的重要能力之一，操作系统在系统软件中具有核心地位。文件系统是操作系统的重要组成部分，是操作系统组织和管理资源必不可少的内容。学生对操作系统的直观认识就是从使用文件系统开始的，在现代操作系统中，文件系统已是操作系统内核中进程管理，内存管理，设备管理等有关实现赖以管理的基础，进而网络，数据库，各种应用等等也以文件系统为前提。因此，对文件系统的理解关乎对整个计算机体系的理解。本综合实训在内存中仿真 Ext2 文件系统，包括文件系统的外存格式，内存存在的实现，并以此为基础对文件进行操作和管理。以此让学生理解文件系统的实现原理和文件系统的开发实践，为系统级软件的理解和开发奠定坚实的基础。

实训要求

（1）要求在内存中仿真 Ext2 文件系统格式，实现其各个部分的数据结构，理解每个数据结构的内容。

（2）要求实现 Ext2 文件系统的内存实现和操作。

（3）要求对文件进行打开，读，写等有关操作。

（4）独立完成不得抄袭

考核点

（1）熟练掌握 Ext2 文件系统格式。

（2）熟练掌握文件系统在操作系统中的实现。

详细内容

一、内存中仿真 Ext2 文件系统格式

1、创建 Ext2 文件系统的步骤

创建 Ext2 文件系统，完成的步骤如下：

1） 初始化超级块和组描述符。

2）对于每个块组，保留存放超级块、组描述符、索引节点表及两

个位图所需要的所有磁盘块。

3）把索引节点位图和每个块组的数据映射位图都初始化为 0。

4）初始化每个块组的索引节点表。

5）创建/root 目录。

6）在前两个已经创建的目录所在的块组中，更新块组中的索引节点位图和数据块位图。

2、ext2 文件系统整体结构

ext2 文件系统整体结构如下：

硬盘首先分区,然后格式化,才能使用。格式化的过程会在硬盘上建立很多块,为了管理这些块,文件系统将其分组,每个组称为块组(block group).每个块组又由六部分组成(见图 1):超级块(super block)、块组描述表(group descriptor table)、块位图(blockbitmap)、索引位图(inode bitmap)、索引表(inodetable)和数据块(data block)。

图 1　硬盘、分区、块组的组成

3、ext2 文件系统组成部分的数据结构

每个部分的组成的数据结构如下：

1）引导块 BootBlock

每个硬件分区的开头 1024 字节，即 0byte 至 1023byte 是分区的启动扇区。存放由 ROM BLOS 自动读入的引导程序和数据，但这只对引导设备有效，而对于非引导设备，该引导块不含代码。这个

块与 ext2 没有任何关系，仿真时可以不实现，但是在仿真整个文件系统中要有引导块概念。

2）超级块 SuperBlock

分区剩余部分被分为若干个组。每个组里均由一个 Super Block 块和一个 Group Descriptor（组描述符）块组成。Group 0 中的 Super Bock 被内核所用。定义了诸如文件系统的静态结构，包括块的大小，总块数，每组内的 inode 数，空闲块，索引节点数等全局信息。其他 Group 中的 Super Block 则仅是 Group 0 中的 Super Block 的一个拷贝。

超级块的数据结构定义为 struct ext2_super_block。

系统启动时 super block 0 的内容读入内存，某个组损坏可以来恢复。超级块数据结构内容如下：

```
struct ext2_super_block
{  __u32    s_inodes_count;  /* 文件系统中索引节点总数 */
   __u32    s_blocks_count;  /*文件系统中总块数 */
   __u32    s_r_blocks_count;        /* 为超级用户保留的块数 */
   __u32    s_free_blocks_count; /*文件系统中空闲块总数 */
   __u32    s_free_inodes_count; /*文件系统中空闲索引节点总数*/
   __u32    s_first_data_block;    /* 文件系统中第一个数据块 */
   __u32    s_log_block_size;      /* 用于计算逻辑块大小 */
   __s32    s_log_frag_size;   /* 用于计算片大小 */
   __u32    s_blocks_per_group;  /* 每组中块数 */
   __u32    s_frags_per_group;   /* 每组中片数 */
   __u32    s_inodes_per_group;  /* 每组中索引节点数 */
```

```
__u32   s_mtime;    /*最后一次安装操作的时间 */
__u32   s_wtime;   /*最后一次对该超级块进行写操作的时间 */
__u16   s_mnt_count;    /* 安装计数 */
__s16   s_max_mnt_count;    /* 最大可安装计数 */
__u16   s_magic;   /* 用于确定文件系统版本的标志 */
__u16   s_state;   /* 文件系统的状态*/
__u16   s_errors;   /* 当检测到有错误时如何处理 */
__u16   s_minor_rev_level;   /* 次版本号 */
__u32   s_lastcheck; /* 最后一次检测文件系统状态的时间 */
 __u32  s_checkinterval; /* 两次对文件系统状态进行检测的间隔时间 */
__u32   s_rev_level; /* 版本号 */
__u16   s_def_resuid;    /* 保留块的默认用户标识号 */
__u16   s_def_resgid;   /* 保留块的默认用户组标识号*/
 __u32    s_first_ino;   /* 第一个非保留的索引节点 */
 __u16    s_inode_size;   /* 索引节点的大小 */
__u16    s_block_group_nr;   /* 该超级块的块组号 */
__u32    s_feature_compat;   /* 兼容特点的位图*/
 __u32    s_feature_incompat;   /* 非兼容特点的位图 */
__u32    s_feature_ro_compat;   /* 只读兼容特点的位图*/
__u8     s_uuid[16];   /* 128位的文件系统标识号*/
```

```
    char        s_volume_name[16];   /* 卷名 */
    char        s_last_mounted[64];   /* 最后一个安装点的路径名 */
    __u32       s_algorithm_usage_bitmap;   /* 用于压缩*/
    __u8        s_prealloc_blocks;   /* 预分配的块数*/
    __u8        s_prealloc_dir_blocks;   /* 给目录预分配的块数 */
    __u16       s_padding1;   /* 填充 */
    __u32       s_reserved[204];   /* 用null填充块的末尾 */
};
```

下面对其中一些域的解释。

（1）文件系统中并非所有的块普通用户都可以使用，有一些块是保留给超级用户专用的，这些块的数目就是在s_r_blocks_count中定义的。一旦空闲块总数等于保留块数，普通用户无法再申请到块了。如果保留块也被使用，则系统就可能无法启动了。有了保留块，我们就可以确保一个最小的空间用于引导系统。

（2）逻辑块是从0开始编号的，对块大小为1K的文件系统，s_first_data_block为1，对其它文件系统，则为0。

（3）s_log_block_size是一个整数，以2的幂次方表示块的大小， 用1024字节作为单位。因此，0表示1024字节的块，1表示2048字节的块，如此等等。

同样 ,片的大小计算方法也是类似的 ,因为Ext2中还没有实现片，因此，s_log_frag_size与s_log_block_size相等。

（4）Ext2要定期检查自己的状态，它的状态取下面两个值之一。

```
#define EXT2_VALID_FS       0x0001
```

文件系统没有出错。

#define EXT2_ERROR_FS   0x0002

内核检测到错误。

s_lastcheck就是用来记录最近一次检查状态的时间，而s_checkinterval则规定了两次检查状态的最大允许间隔时间。

3) 组块描述表 group descriptor table

块组中，紧跟在超级块后面的是组描述符表，其每一项称为组描述符，定义为ext2_group_desc的数据结构，共32字节。它是用来描述某个块组的整体信息的。

```
struct ext2_group_desc
{
    __u32   bg_block_bitmap;      /* 组中块位图所在的块号 */
    __u32   bg_inode_bitmap;   /* 组中索引节点位图所在块的块号 */
    __u32   bg_inode_table;   /*组中索引节点表的首块号 */
    __u16   bg_free_blocks_count;   /* 组中空闲块数 */
    __u16   bg_free_inodes_count;   /* 组中空闲索引节点数 */
    __u16   bg_used_dirs_count;/*组中分配给目录的节点数 */
    __u16   bg_pad;   /*填充，对齐到字*/
    __u32 [ 3 ]  bg_reserved;    /*用null填充12个字节*/
}
```

每个块组都有一个相应的组描述符来描述它,所有的组描述符形成一个组描述符表,组描述符表可能占多个数据块。组描述符就相当于每个块组的超级块,一旦某个组描述符遭到破坏,整个块组将无法使用,所以组描述符表也像超级块那样,在每个块组中进行备份,以防遭到破坏。组描述符表所占的块和普通的数据块一样,在使用时被调入块高速缓存。

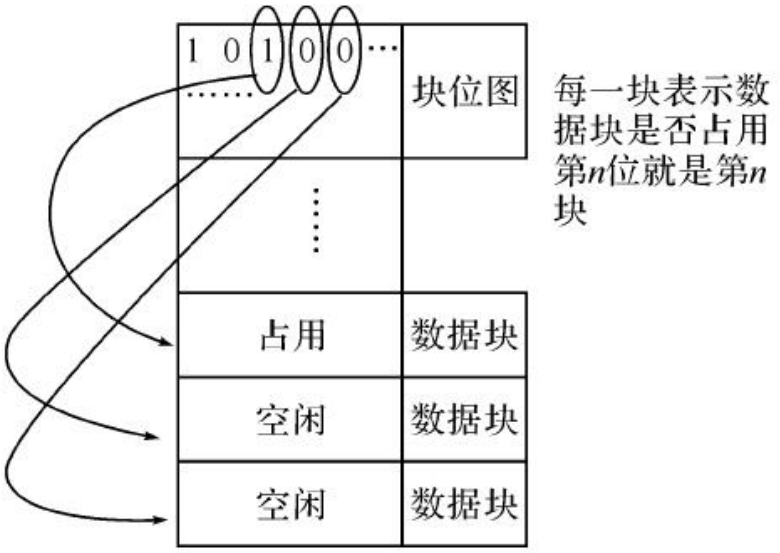

图 2　位图描述块

4）位图 bitmap

文件系统用位图来管理磁盘块和 I 节点,位图分为块位图和 I 节点位图。块位图占用一个磁盘块,当某位为“1”时,表示磁盘块空闲,为“0”时表示磁盘块被占用。I 节点位图也占用一个磁盘块,当它为“0”时,表示组内某个对应的 I 节点空闲,为“1”时表示已被占用。位图使系统能够快速地分配 I 节点和数据块,保证同一文件的数据块能在磁盘上连续存放,从而大大地提高了系统的实时性能。

在创建文件时,文件系统必须在块位图中查找第一个空闲 I 节点,把它分配给这个新创建的文件。在该空闲 I 节点分配使用后,就需要修改指针,使它指向下一个空闲 I 节点。同样地,I 节点被释放后,则需要修改指向第一个空闲 I 节点的指针。

5）外存索引节点 Inode

Ext2使用索引节点来记录文件信息。每一个普通文件和目录都有唯一的索引节点与之对应，索引节点中含有文件或目录的重要信息。当你要访问一个文件或目录时，通过文件或目录名首先找到与之对应的索引节点，然后通过索引节点得到文件或目录的信息及磁盘上的具体的存储位置。Ext2的索引节点的数据结构d定义为ext2_inode，下面是其结构及各个域的含义。

```
struct ext2_inode {
    __u16  i_mode; /*  文件类型和访问权限  */
    __u16    i_uid;    /*  文件拥有者标识号*/
    __u32    i_size; /*  以字节计的文件大小  */
    __u32    i_atime; /*  文件的最后一次访问时间  */
    __u32    i_ctime; /*  该节点最后被修改时间  */
    __u32    i_mtime; /*  文件内容的最后修改时间  */
    __u32    i_dtime; /*  文件删除时间  */
    __u16    i_gid;    /*  文件的用户组标志符  */
    __u16    i_links_count; /*  文件的硬链接计数  */
    __u32    i_blocks; /*  文件所占块数（每块以512字节计）*/
    __u32    i_flags; /*  打开文件的方式  */
    union     /*特定操作系统的信息*/
    __u32    i_block[Ext2_N_BLOCKS];    /*  指向数据块的指针数组  */
    __u32    i_version;        /*  文件的版本号（用于  NFS）  */
    __u32    i_file_acl; /*文件访问控制表（已不再使用）  */
```

```
    __u32    i_dir_acl;      /*目录  访问控制表（已不再使用）*/
     __u8    l_i_frag;  /*  每块中的片数  */
     __u32     i_faddr;    /*  片的地址  */
}
```

从中可以看出，索引节点是用来描述文件或目录信息的。

文件类型（12-15位）：

普通文件

目录文件

块设备

字符设备

套接口

符号链接

管道/FIFO

访问许可（0-8位）：

IRUSR (-000400)　（文件主可读）

IWUSR(-000200)　（文件主可写）

IXUSR (-000100)　（文件主可执行）

IRGRP (-000040)　（同组用户可读）

IWGRP(-000020)　（同组用户可写）

IXGRP (-000010)　　（同组用户可执行）

IROTH (-000004)　　（其它用户可读）

IWOTH(-000002)　　（其它用户可写）

IXOTH (-000001)　　（其它用户可执行）

i_block[Ext2_N_BLOCKS]是指向数据块的指针数组如图3

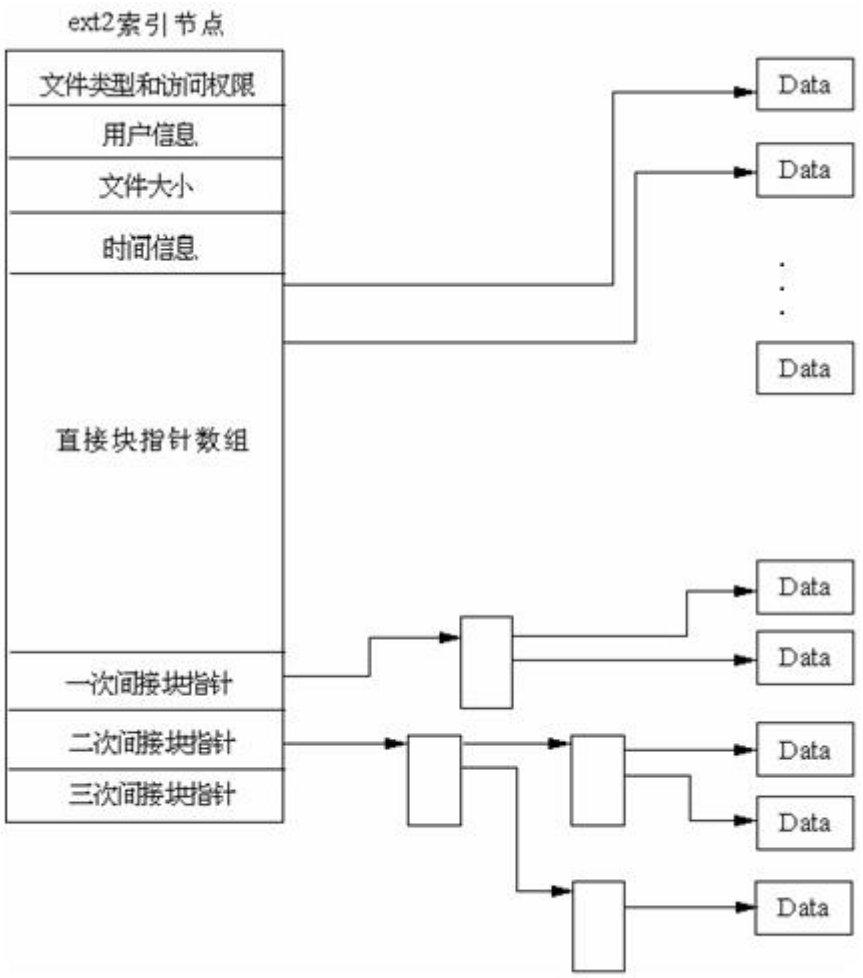

图 3 指向数据块的指针数组

6）数据块 DataBlock

根据不同的文件类型有以下几种情况：

1.对于常规文件，文件的数据存储在数据块中。

2.对于目录，该目录下的所有文件名和目录名存储在数据块中，注意文件名保存在它所在目录的数据块中。注意这个概念：目录也是一种文件，是一种特殊类型的文件。

3.对于符号链接，如果目标路径名较短则直接保存在 inode 中以便更快地查找，如果目标路径名较长则分配一个数据块来保存。

4.设备文件、FIFO 和 socket 等特殊文件没有数据块，设备文件的主设备号和次设备号保存在 inode 中。

在 ext2 文件系统中，目录是作为文件存储的。根目录总是在 inode 表的第二项，而其子目录则在根目录文件的内容中定义。其结构如下：

```
struct ext2_dir_entry_2 {
__le32 inode; //  文件入口的 inode 号，0 表示该项未使用
__le16 rec_len; //  目录项长度
__u8 name_len; //  文件名包含的字符数
__u8 file_type; //  文件类型
char name[255]; //  文件名
};
```

二、文件系统内存映象实现及相关函数

以下为借鉴借鉴 Linux 内核文件系统的实现，仅作为指导参考。

1、文件系统内存映象结构

Linux 内核文件系统的实现，其相关数据结构关系如下：

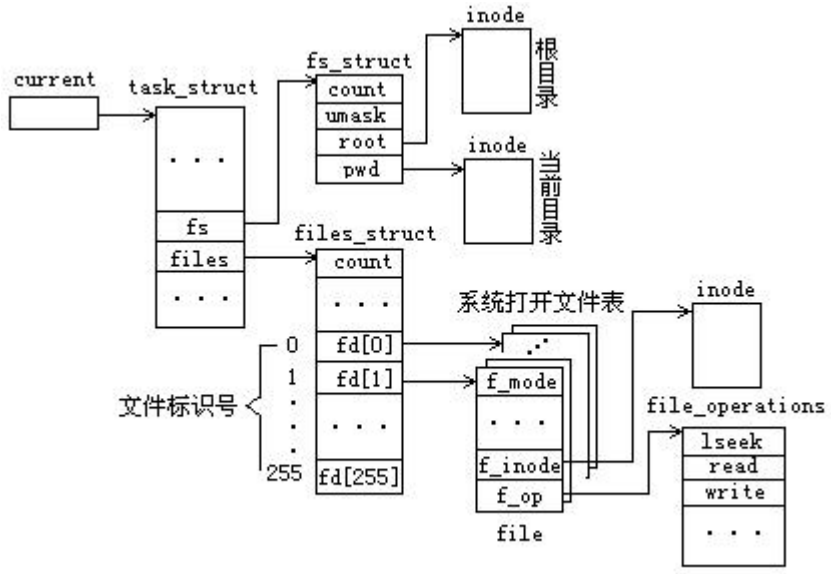

图 4 进程的文件系统实现

2、文件系统内存映象及相关函数

对于一个进程打开的所有文件，由进程的两个私有结构进行管理，

1）fs_struct结构

fs_struct结构记录着文件系统根目录和当前目录，

```
struct fs_struct {
int count;        /* 共享此结构的计数值 */
unsigned short umask;    /* 文件掩码 */
struct inode * root, * pwd;    /* 根目录和当前目录inode指针 */
};
```

root是指向当前目录所在的文件系统的根目录inode，在按照绝对路径访问文件时就从这个指针开始。

pwd是指向当前目录inode的指针，相对路径则从这个指针开始。

2）files_struct结构

files_struct结构包含着进程的打开文件表。

```
struct files_struct {
int count;       /* 共享该结构的计数值 */
fd_set close_on_exec;
fd_set open_fds;
struct file * fd[NR_OPEN];
};
```

fd[]每个元素是一个指向file结构体的指针，该数组称为进程打开文件表。进程每打开一个文件时，建立一个file结构体，并加入到系统打开文件表中，然后把该file结构体　的首地址写入fd[]数组的第一个空闲元素中一个进程所有打开的文件都记载在fd[]数组中。fd[]数组的下标称为文件标识号。

进程使用文件名打开一个文件。 在此之后对文件的识别就不再使用文件名，而直接使用文件标识号。在系统启动时文件标识号0、1、2由系统分配:

0标准输入设备，1标准输出设备，2标准错误输出设备。

当一个进程通过fork()创建一个子进程后，子进程共享父进程的打开文件表,子进程两者的打开文件表中下标相同的两个元素指向同一个file结构。 这时file的f_count计数值增1。

一个文件可以被某个进程多次打开，每次都分配一个file，并占用该进程打开文件表fd[]的一项,得到一个文件标识号。但它们的file中的f_inode都指向同一个inode。

3）系统打开文件表

把所有进程打开的文件集中管理， 把它们组成“系统打开文件表”。

系统打开文件表是一个双向链表，它的每个表项（节点）是一个file结构，称为文件描述符，其中存放着一个已打开文件的管理控制信息

进程每次打开一个文件就建立一个file结构体，并把它加入到系统打开文件链表中。

全局变量first_file指向系统打开文件表的表头。

```
struct file {
    mode_t f_mode;    /* 文件的打开模式 */
    loff_t f_pos;    /* 文件的当前读写位置 */
    unsigned short f_flags; /* 文件操作标志 */
    unsigned short f_count; /* 共享该结构体的计数值 */
    unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
    struct file *f_next, *f_prev; /* 链接前后节点的指针 */
    struct fown_struct f_owner;    /* SIGIO用PID */
    struct inode * f_inode;    /* 指向文件对应的inode */
    struct file_operations * f_op; /* 指向文件操作结构体的指针 */
    unsigned long f_version; /* 文件版本 */
   void *private_data; /* 指向与文件管理模块有关的私有数据的指针 */
};
```

f_mode是文件创建或打开时指定的文件属性，包括文件操作模式和访问权限。符号常量FMODE_READ（读）和FMODE_WRITE（写）

f_flags指定了文件打开后的处理方式，

O_RDONLY仅为读操作打开文件，

O_WRONLY仅为写操作打开文件，

O_RDWR为读和写操作打开文件等。

f_pos记载文件中当前读写处理所在的字节位置，相当是文件内部的一个位置指针。

f_inode指向文件对应的VFSinode。

f_count记载的是共享该file结构体的进程的数目。

i_count记载共享此文件的独立进程数目。

f_op指向对文件进行操作的函数指针集合

file_operations结构。通过f_op对不同文件系统的文件调用不同的操作函数。

```
struct file_operations {
    int (*lseek) (struct inode *, struct file *, off_t, int);
    int (*read) (struct inode *, struct file *, char *, int);
    int (*write) (struct inode *, struct file *, const char *, int);
    int (*readdir) (struct inode *, struct file *, void *, filldir_t);
    int (*select) (struct inode *, struct file *, int, select_table *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    void (*release) (struct inode *, struct file *);
```

```
    int (*fsync) (struct inode *, struct file *);
    int (*fasync) (struct inode *, struct file *, int);
    int (*check_media_change) (kdev_t dev);
    int (*revalidate) (kdev_t dev);
};
```

lseek(inode, file, offset,origin)

文件定位函数，用于改变文件内部位置指针的值。

read(inode,file,buffer,count)

读文件函数 ,读取inode对应的文件 ,count指定了读取的字节数，读取的数据置入以buffer为首址的内存区域。

write(inode,file,buffer,count)

写文件函数，把内存buffer缓冲区的数据写入inode对应的文件中，count为写入数据的字节数。

readdir(inode,file,dirent,count)

读目录函数，从inode对应的目录项结构体dirent中读取数据。dirent类似于EXT2文件的目录项结构ext2_dir_entry，

select(inode, file,type,wait)

文件读写检测函数，检测能否对设备进行读或写操作。inode和file指定了操作对象的设备文件，type指定操作类型:

type =SEL_IN为从设备读取，

type =SEL_OUT为向设备写入。

当wait不为NULL时，在设备可以利用之前进程等待。

ioctl(inode, file,cmd,arg)

参数变更函数，用于对设备文件的某些参数的变更。

mmap(inode, file, vm_area)

文件映射函数，把文件的一部分映射到用户的虚拟内存区域。

vm_area是映射文件对应的vm_area_struct结构体。

open(inode, file)

文件打开函数。在进程打开一个文件调用。该操作函数应该属于inode_operations结构 ,把它置于file_operations之中是因为通过file结构更便于对文件的操作。

release(inode, file)

file结构释放函数 ,当file结构的f_count为0时调用此函数释放该结构体。

fsync(inode, file)

文件同步函数，当文件在缓冲区中的内容被修改时，调用该函数把其内容写回外存的该文件中。

fasync(inode,file,on)

文件异步函数，用于终端设备和网络套接口的异步I/O操作。

check_media_change(dev)

媒体检测函数，检测非固定连接媒体的设备是否已发生变更，若已变更返值为1，无变更返值为0。

revalidate(dev)

媒体重置函数，当非固定连接媒体设备发生变更时，调用该函数重新设置该媒体对应的各个数据结构中的有关数据。

4）内存索引节点

文件被打开后，系统为它在内存索引结点表区中建一内存索引结点，以方便用户和系统对文件的访问。其中，一部分信息是直接从磁盘索引结点拷贝过来的，如 i_mode、i_uid、i_gid、i_size、i_addr、i_nlink 等，并又增加了如下各信息：　　索引结点编号 i_number。作为内存索引结点的标识符；

状态 i_flag。指示内存索引结点是否已上锁、是否有进程等待此 i 结点解锁、i 结点是否被修改、是否有最近被访问等标志；

引用计数 i_count。记录当前有几个进程正在访问此 i 结点，每当有进程访问此 i 结点时，对 i_count+1，退出-1；

设备号 i_dev。文件所属文 S 件系统的逻辑设备号；

前向指针 i_forw。Hash 队列的前向指针；

后向指针 i_back。Hash 队列的后向指针；

有关内存索引节点的操作是 inode_operations 结构。

```
struct inode_operations {
    struct file_operations * default_file_ops;
    int (*create) (struct inode *,const char *,int,int,struct inode
    **);
    int (*lookup) (struct inode *,const char *,int,struct inode **);
    int (*link) (struct inode *,struct inode *,const char *,int);
    int (*unlink) (struct inode *,const char *,int);
    int (*symlink) (struct inode *,const char *,int,const char *);
    int (*mkdir) (struct inode *,const char *,int,int);
```

```
    int (*rmdir) (struct inode *,const char *,int);
    int (*mknod) (struct inode *,const char *,int,int,int);
    int (*rename) (struct inode *,const char *,int,struct inode
    *,const char *,int, int);
    int (*readlink) (struct inode *,char *,int);
    int (*follow_link) (struct inode *,struct inode *,int,int,struct
    inode **);
    int (*readpage) (struct inode *, struct page *);
    int (*writepage) (struct inode *, struct page *);
    int (*bmap) (struct inode *,int);
    void (*truncate) (struct inode *);
    int (*permission) (struct inode *, int);
    int (*smap) (struct inode *,int);
};
```

default_file_ops指向结构体file_operations，其中集合了对打开的文件进行各种操作的函数指针。

create (dir,name,len,mode,res_inode)文件创建函数，在指定的目录中建立一个文件的目录项。dir指定文件建立的目录位置，name为文件名，len为文件名长度，mode指定文件的类型和访问权限。参数res_inode返回新建inode的地址。

lookup (dir,name,len,res_inode)文件搜索函数，在dir目录中搜索名字为name、长度len的文件。参数res_inode返回其inode地址。若不存在该文件，返值为ENOTDIR。

link(oldinode,dir,name,len)

文件链接函数 ,用于文件的硬链接。把oldinode对应的文件与dir中名字为name、长度为len的文件进行硬链接。

unlink(dir,name,len)

文件链接撤消函数，从dir中删除名字为name、长度为len的链接文件。

symlink(dir,name,len,symname)

符号链接函数，在dir中建立符号名字为name、长度为len的符号链接。Symname指定了符号链接目标的路径。

mkdir(dir,name,len,mode)

目录创建函数，在dir中建立名字为name、名字长度为len，访问权限属性为mode的子目录。

rmdir(dir,name,len)

目录删除函数，从dir中删除名字为name、名字长度为len的子目录。

mknod(dir,name,len,mode,rdev)

inode创建函数，用于在dir目录中创建设备文件。name为创建的文件名，len为文件名长度，mode为文件的访问权限属性，rdev为特殊文件对应的设备号。

rename(odir,oname,olen,ndir,nname,nlen)

文件重命名函数，odir、oname、olen指定了文件的原目录、名字和名字长度，ndir、nname、nlen指定了文件的新目录新名字和名字长度。

readlink(inode,buf,bufsize)

读符号链接函数 ,在inode中读取符号链接的文件路径 ,写入buf指向的缓冲区内，缓冲区长度为bufsiaz。

follow_link(dir,inode,flag,mode,res_inode)

符号链接搜索函数，在dir中搜索属性为mode、标志为flag的符号链接的inode文件，找到后由参数res_inode返回inode号。

readpage(inode, page)

读取文件页面函数，读取inode对应的文件在内存page页面中的内容。

writepage(inode,page)

写文件页面函数， 向inode对应的文件在内存的page页面中写入数据。

bmap(inode,block)

数据块映射函数 ,得到inode中 ,与逻辑块号block对应的物理块号。

truncate(inode)

文件长度变更函数，改变inode文件的长度。调用前，在inode结构体的i_size中置入文件长度。

permission(inode,perm)

访问权限检验函数。 检验进程对文件inode是否具有prem指定的访问权限。

smap(inode,sector)

扇区映射函数，与数据块映射函数bmap（ ）类似，但该函数是面对磁盘扇区的操作，主要用于UMDOS、MSDOS文件系统。

5）内存超级块

在外存超级块的基础上增加如下数据域：

设备号（所在设备设备号super_dev）

访问计数 （访问此内存超级块的进程数）

使用状态：互斥锁

struct super_operations: 超级块的操作函数

```
struct super_operations {
void (*read_inode) (struct inode *);
int (*notify_change) (struct inode *, struct iattr *);
void (*write_inode) (struct inode *);
void (*put_inode) (struct inode *);
void (*put_super) (struct super_block *);
void (*write_super) (struct super_block *);
void (*statfs) (struct super_block *, struct statfs *, int);
int (*remount_fs) (struct super_block *, int *, char *);
};
```

read_inode（inode）

当在系统中建立一个新的时，则调用该函数从外存中读取一个文件或目录的inode的相关值来填充它。

notify_change（inode,iattr）

该函数主要是对NFS（网络文件系统）执行的操作。当inode的属性改变时，调用该函数通知外部的计算机系统。

iattr指向iattr结构体，其中记载着变更的数据。

write_inode(inode)

当VFSinode的内容发生变动时，调用此函数把它写回到外存中对应的inode中。

put_inode(inode)

调用该函数撤消某个VFSinode。

put_super(sb)

当某个文件系统卸载时，调用该函数撤消其超级块，同时释放与超级块有关的高速缓存空间。然后，把该超级块的成员项s_dev置0，表明该超级块已撤消。以后建立新超级块时可以再次使用它。

write_super(sb)

当超级块的内容发生变化时，调用该函数把超级块的内容写回外存中保存。

statfs(sb, statsbuf, bufsize)

调用该函数可以得到文件系统的某些统计信息时。参数statsbuf指向一个statfs结构体，函数执行中把文件系统的统计信息填入该结构体中。当函数返回时，从statfs结构体中可以有关统计信息。

statfs结构与硬件有关.

三、文件的 open,read,write,flctl 等函数实现

文件的open,read,write,flctl等函数实现

1）打开文件open

int open (const char *pathname, int oflag,.../* , mode_t mode */);

按oflag指定的方式打开pathname 指定的文件，返回文件描述字；

oflag: O_RDONLY /* 只读 */

O_WRONLY /* 只写 */

O_RDWR /* 读写 */

O_APPEND /* 附加写 */

O_CREAT /* 如不存在则创建，配合mode 参数 */

O_EXCL /* 用于原子操作，配合O_CREAT如不存在则创建*/

O_TRUNC /* 如只读或只写则长度截为0 */

O_NOCTTY /* 不作为控制终端 */

O_NONBLOCK /* 非阻塞 */

O_SYNC　/* 等待物理IO完成 */

mode：　当有O_CREAT 时，指出访问权限即其它属性。

实现算法：

（1）按路径名搜索分级目录，找到节点读入内存节点表中；

（2）检查访问权限（i_mode）；

（3）分配填写系统打开文件表项：读写标志(oflag)，文件指针，引用数(+1)，i节点指针；

（4）分配填写用户文件表项：系统文件表项指针；

（5）返回文件描述字（用户打开文件表项索引号）

2）关闭文件close

int close (int fd)；

关闭由文件描述符 fd 指向的文件。

实现算法：

（1）由 fd 找到对应的用户文件表项，系统文件表项和内存 i 节点；

（2）系统文件表项中的引用数减 1 ，如结果为 0则释放此项并将内存i节点中的引用数减 1 ，如结果也为 0 则将此i节点写回磁盘 i 节点，释放内存 i 节点；（如连接数为0则释放所有文件块，释放内存 i 节点，置磁盘 i 节点为空闲。）

（3）如果是设备文件则调用对应的设备关闭函数；

（4）释放对应的用户文件表项。

3）建立文件create

int creat (const char *pathname，mode_t mode);

creat 建立名字为 pathname 属性为mode 的文件，然后按只写方式打开pathname 指定的文件，返回文件描 述字。

实现算法：

（1）按路径名搜索分级目录；

（2）如文件已存在则检查访问权限（i_mode），如不允许则出错返回;

（3）如文不存在则分配磁盘空闲 i 节点，填初值(mode等)，在父目录中分配新目录项，填入文件名和节点号；

（4）磁盘 i 节点 读入内存 i 节点（增减部分内容）；填写系统打开文件表项（读写标志=只写，文件指针=0，引用数+1，i节点指针）；

（5）分配填写用户打开文件表项（文件表项指针）；

（6）如文件已存在则释放存储块；

（7）返回文件描述字（用户文件表项索引号）

4）读文件read

ssize_t read (int fd, void *buff, size_t nbytes);

从文件中文件指针开始读nbytes字节到buff指定的内存区 ,文件指针后移实际读的字节数 , 返回实际读的字节数。 fd　文件描述字；buff 内存地址（通常为数组）；nbytes 要读的字节数。

算法实现：

（1）由fd 找到文件表项，作权限检查，取文件指针；

（2）由文件指针计算文件逻辑块号和块内位移；

逻辑块号 = [文件指针/块大小]取整数；

块内位移 = [文件指针/块大小]取余数；

（3）查i节点表项的索引表，将逻辑块号转换成物理块号；

（4）读出对应物理块到系统缓冲区(bread, breada) , 由块内位移开始取出适当的字节数到指定用户内存(buff)；

（5）如果未达到指定字节数（nbytes），则继续读下一块（转4。），直至读完；

（6）返回实际读出的字节数。

5）写文件write

ssize_t write (int fd, void *buff, size_t nbytes);

从buff指定的内存区写nbytes到从文件指针开始的文件中 ,文件指针后移实际写的字节数，返回实际写的字节数。fd 文件描述字；buff　内存地址（通常为数组）；nbytes 要写的字节数。

算法实现：

(1)由fd 找到文件表项，作权限检查，取文件指针；

(2)由文件指针计算文件逻辑块号和块内位移；

逻辑块号 ＝[文件指针/块大小]取整数；

块内位移 ＝[文件指针/块大小]取余数；

(3)查i节点表项的索引表，将逻辑块号转换成物理块号；

(4)由指定用户内存(buff)复制适当的字节数到系统缓冲区中(上述块内位移)，可能第一块要先读出到缓冲区；然后同步(bwrite)或延迟写(bdwrite)入对应物理块；

(5)如果未达到指定字节数(nbytes),则继续写下一块(转4。),直至写完；

(6)返回实际写入的字节数。

6) 移动文件指针lseek

off_t lseek(int fd, off_t offset, int whence);

文件读写指针设置为(whence,offset)指定的位置(可以超过文件长度，基延长文件长度，空洞部分值为0)，返回新指针值。

whence: 起点

SEEK_SET(0) /* 文件头 */

SEEK_CUR(1) /* 当前文件读写指针所在位移 */

SEEK_END(2) /* 文件尾 */

offset： 位移量

文件读写指针的缺省值：读写打开时为0，附加写打开时为文件尾。

7）文件控制fIctl

int fcntl (int fd, int cmd, .../* int arg */);

改变文件的性质。cmd表示五种功能：

F_DUPFD: 复制文件描述字；

F_GETFD / F_SETFD: 获得/设置文件描述字标志FD_CLOSEXEC（默认为0，表示在进程exec后不关闭该文件）；

F_GETFL ：读文件状态：O_RDONLY（只读），O_WRONLY（只写），O_RDWR（读写），O_APPEND（附加），O_NONBLOCK（非阻塞IO），O_SYNC（同步），O_ASYNC（异步）；

F_SETFL: 设置文件状态: O_APPEND, O_NONBLOCK, O_SYNC, O_ASYNC；

F_GETOWN / F_SETOWN: 获得/设置接受SIGIO信号（异步IO）或SIGURG信号（网络紧急数据）的进程ID或进程组ID；

F_GETLK / F_SETLK / F_SETLKW: 获得/设置文件记录锁。

参考资料

参考Linux 2.4.32版本内核文件系统实现