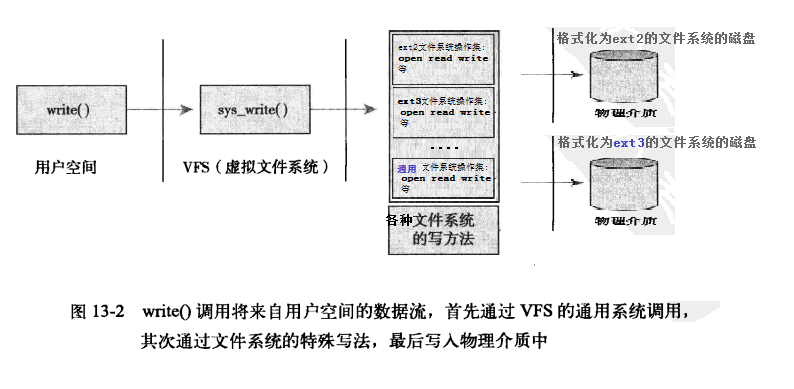
**虚拟文件系统VFS**

虚拟文件系统是内核的一个子系统,这个子系统主要是为系统兼容的**多种文件系统**提供**统一**的**用户空间访问文件系统的接口,**用户空间不用考虑访问的具体文件系统是哪一种,就能对这个文件系统所在的磁盘进行读写等操作

虚拟文件系统的实现原理主要是其提供了一个**通用的文件系统模型**,这个模型囊括了任何文件系统的常用功能集和行为,定义了所有文件系统都支持的,基本的,概念上的接口和数据结构,同时实际的文件系统在实现时的一些如”如何打开文件””目录是什么”等概念与VFS中的这个**通用文件系统模型**保持一致,当一个新的文件系统想让linux支持时,只需要提供VFS中的这个通用文件系统模型所期望的接口和数据结构

当系统调用操作文件时,如write(fd,buf,len);会调用到VFS中的sys\_write()接口,sys\_write()会最终找到fd所在的文件系统实际给出的写操作接口,然后执行实际的文件系统写操作,写入磁盘中.



**如下介绍VFS中的这个通用文件系统模型的结构**

这个通用系统模型是从unix文件系统上提取出来的,所以这里介绍**unix文件系统**

**所谓文件系统**其实就是**将磁盘空间**用这种用户可以**直观的看到的文件目录结构表示**出来,用户**直接操作文件或目录**,数据就会在**磁盘上更新**.

unix文件系统由:**文件**,**目录项**,**索引节点**,**安装点**概念组成,具体的在”linux应用之\_文件IO前世今生”中已经做了介绍,这里粘贴出来

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**文件系统**

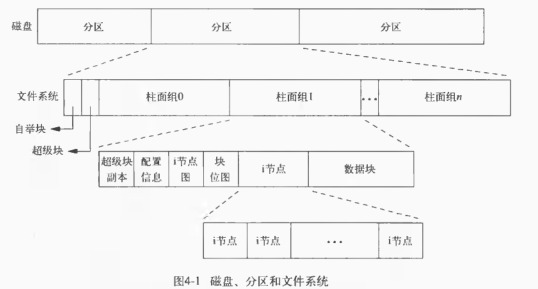
对于一块磁盘，想使用的话首先得分区，然后对分区进行格式化成系统能识别出的文件系统，**文件系统就相当于将一段给定的磁盘存储空间用抽象的文件概念来管理**，操作文件就相当于操作了磁盘，**逻辑块**是将分区格式化为文件系统是指定的最小存储单位

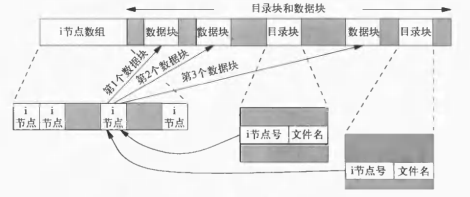
linux中最标准的文件系统ext2，这个文件系统中将文件内容分为两部分存储，一个是文件的属性，一个是文件的内容，属性用inode存储，内容用块存储（所有的逻辑块大致分为两大区域，一个用于放所有的indeo结点，一个用于放文件内容）

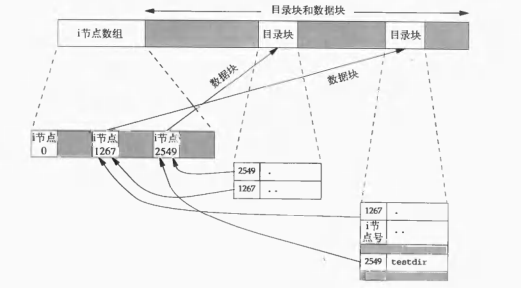
linux中创建一个普通文件时，会分配到至少一个inode（存放文件的相关属性及文件内容的块指针）和能放下文件内容的N个块（记录文件内容）

linux中创建一个目录文件时，会分配到一个inode（存放目录的相关属性及目录内容的块指针）和至少一个块（用于记录这个目录下的相关文件或目录的关联性，也就是此目录下文件或目录的inode结点位置在哪，**每一项成为一个目录项**）

inode记录文件或者目录的属性（**文件所有者，用户组，访问权限，文件设置用户及设置用户组位，文件类型，文件三个时间，文件大小，文件**），同时具有指针的作用，指向文件或目录内容放置的块







由上图得出两个结论：

1：文件名存放在目录的内容块中

2：看到的**文件夹名A**是上一级目录块中文件夹名，上一级目录块中还有指向这个文件夹A的inode指针

看到的**文件名**是上一级目录块中的文件名,上一级目录块中还有指向这个文件的inode指针

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

VFS中有四个主要的对象类型

超级块对象( 内核用struct **super\_block**结构表示):表示一个具体的已经安装的文件系统

索引节点对象(内核用struct **inode**结构表示),代表一个具体的文件的属性及数据块指针

目录项对象( 内核用struct **dentry**结构表示):目录块中的一项

文件对象( 内核用struct **file**结构表示):真正文件数据存放的地方

针对这四个对象,内核提供对应的操作接口集:

**super\_operation**

**inode\_operation**

**dentry\_operation**

**file\_operation**

对于许多方法,可以继承VFS提供的通用函数,如果通用函数无法满足要求,则必须使用实际文件系提供的这些方法,也就是当应用层执行了某个文件IO,比如修改文件的属性,则最终调到这个文件所在的文件系统的操作文件属性的方法(**inode\_operation**中的某个方法),如果通用的方法可以,则也可以用通用的方法

**超级块对象**

内核用struct **super\_block**结构表示,存储文件系统的信息,其中比较重要的几个域

struct super\_block{

struct **file\_system\_type** s\_type;//**文件系统类型**

struct **super\_operation** s\_op;//**超级块操作方法**

struct **list\_head** s\_inodes;//**inodes链表**

};

文件系统安装时,**从磁盘读取文件系统超级块**,并填充到这个结构体中

其中的超级块操作方法

struct **super\_operation** s\_op

**索引节点对象**

内核用 struct **inode**结构表示,存放文件的属性及数据块的指针,

索引节点对象包含内核操作文件或者目录所需要的全部信息，**这些信息是从磁盘中直接读入的**。

其中比较重要的域为

struct inode{

loff\_t i\_size;//**文件大小**,以字节为单位

struct timespec i\_atime;//文件数据最后的**访问时间**

struct timespec i\_mtime;//文件数据最后的**修改时间**

struct timespec i\_stime;//**inode节点最后的修改时间**

umode\_t i\_mode;//文件的**访问权限**

struct **inode\_operation** \*i\_op;//**inode的操作集**

union {

struct pipe\_inode\_info \*i\_pipi;//管道信息结构体

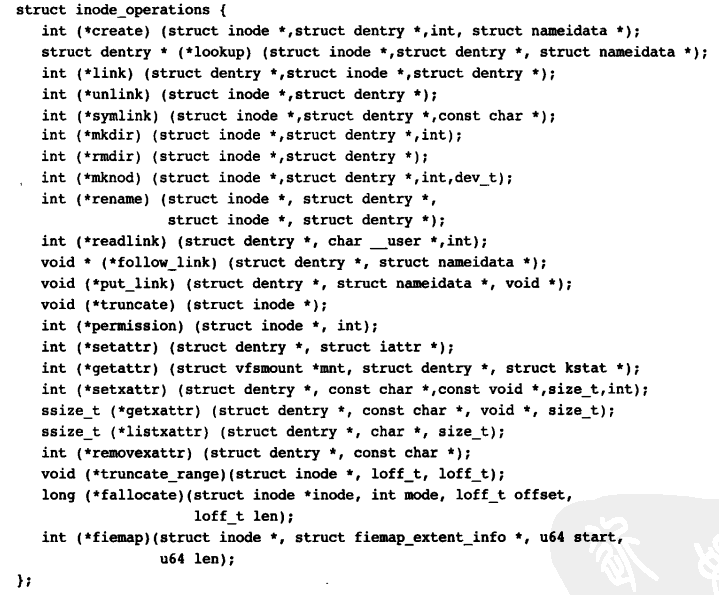
struct block\_device \*i\_bdev;//块设备结构体

struct cdev \*i\_cdev;//字符设备结构体

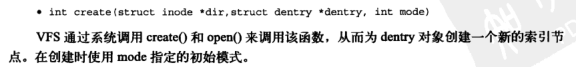
};

};

**inode的操作集**



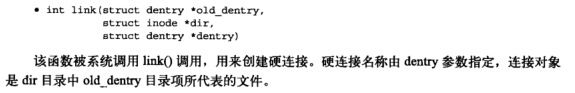
------------------------------------------------------------------------------------



------------------------------------------------------------------------------------



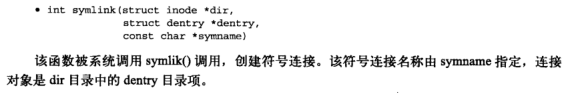
------------------------------------------------------------------------------------



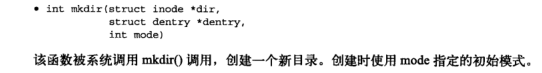
------------------------------------------------------------------------------------



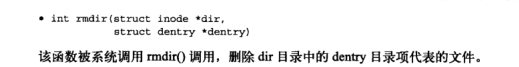
------------------------------------------------------------------------------------



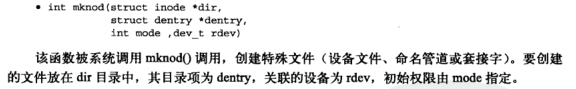
------------------------------------------------------------------------------------



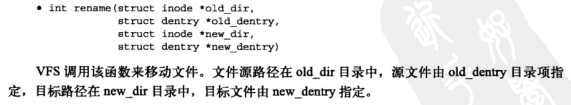
------------------------------------------------------------------------------------



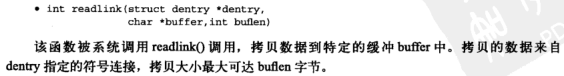
------------------------------------------------------------------------------------



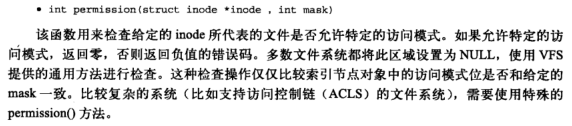
------------------------------------------------------------------------------------



------------------------------------------------------------------------------------



------------------------------------------------------------------------------------



**目录项对象**

目录项对象就是**目录的**inode指向的**数据块**中的**一行内容,这个数据块中可能有多个目录项**

目录项是**文件系统启动后创建的，不是从磁盘读出的**

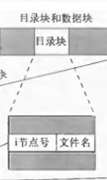
内核用struct **dentry**结构表示,主要的域如下

struct dentry{

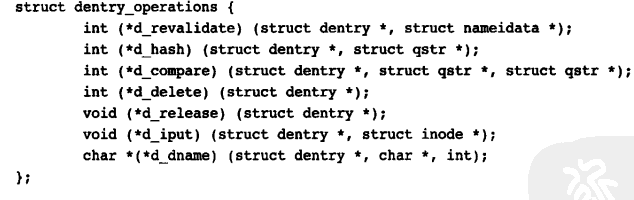
struc inode \*d\_inode;//相关联的索引节点指针

struct qstr d\_name;//目录项内容

};



目录项struct dentry的操作集



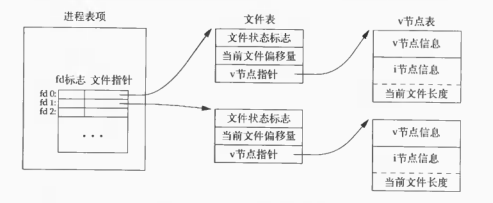
**文件对象**

**文件对象**是内核对某一个进程打开的文件的一种表示

**个人理解 文件对象**其就是在“linux应用之\_文件IO前世今生中的文件共享”介绍到的**文件表,或者是二者之间信息很多都一样**：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

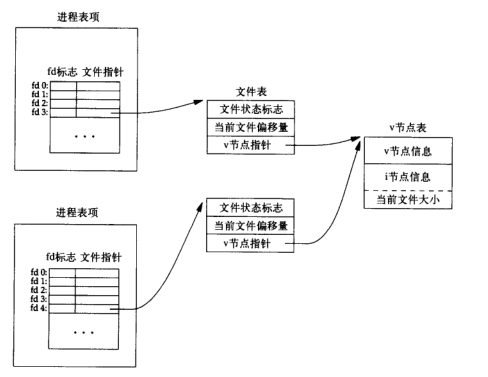
每个进程的描述符中有一个记录项,其中包含一个打开的文件描述符表,并包含每个描述符指定文件的文件表,文件表示内核为打开的文件维护的一个表,其中的v节点(v-node)比较重要(**每个文件只有一个v节点表,v节点有文件对象操作集**),v节点的中的**i节点(i-node) 有文件的所有者,文件长度,文件实际数据块在的磁盘等信息**.



当两个进程打开同一个文件时,内核中有如下实现:

同一文件被不同进程打开都会有一个文件表,同一个文件的文件偏移量对不同进程是不一样的.当然在不同进程打开同一个文件时,可以让他们都指向同一个文件表,如fork创建父子进程时,父子进程对于每一个打开的文件描述符共享同一个文件表

同一个文件不管被多少进程打开只有一个v节点



/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

如进程A打开文件txt，会生成一个文件对象，同时进程B也去打开文件txt，会生出另一个文件对象，这两个文件对象内容有部分不同，但指向的文件是一个，不同的地方比如文件的偏移量等，所以**文件对象**类似于目录项一样**没有对应的磁盘数据**

**文件对象由open函数创**建，close函数撤销

struct file{

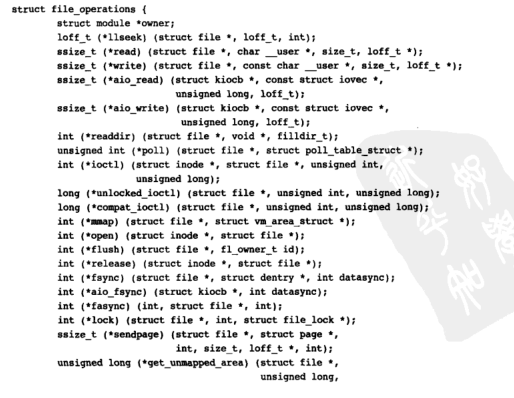
struct file\_operations \*f\_op;//**文件对象操作表**，操作文件对象的接口集合，如修改偏移量，如**write函数**用于**写**文件对象中的v节点指针中的**inode节点指向的数据块**

mode\_t f\_mode;//**文件的访问模式**（**疑问：通常所看到的文件权限九段字与open等输入的权限参数有何关系**）

loff\_t f\_pos;//**文件当前偏移量**

};

文件对象操作函数，**注意不只操作文件内容，还会操作文件对象其他成员操作，所以不能叫文件操作函数**：



**与文件系统相关的数据结构**

超级块机构体中的文件系统类型结构体如下：

struct file\_system\_type {

const char \*name ;//文件系统名字

...

};

每种文件系统，不管有多少个实例安装到系统中，都只有一个file\_system\_type结构

当文件系统安装到系统中时，有一个**vfsmount**结构体会创建出来，这个结构体代表**文件系统实例**，代表**安装点**

struct vfsmount {

struct dentry \*mnt\_mountpoint;//安装点得目录项

int mnt\_flags //安装标志，如禁止访问这个文件系统上的设备文件等

};

**进程打开文件时返回的文件描述符怎么连接到具体文件**

进程描述符中的files域指向如下结构体

struct files\_struct {

struct embedded\_fd\_set open\_fds\_init; //打开的文件描述符链表

struct file \*fd\_array[NR\_OPEN\_DEFAULT]; //文件对象数组

。。。

};

进程描述符中的fs域指向如下结构体，其中包含当前工作目录（pwd）和根目录

struct fs\_struct {

struct path root;//根目录路径

struct path pwd;//当前工作路径的目录

。。。

};

进程描述符中的mmt\_namespace域指向如下结构体

struct mmt\_namespace {

struct vfsmount \* root;//根目录的安装点对象

。。。

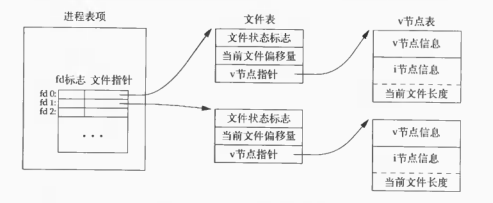
};

其中的struct files\_struct struct fs\_struct这两个结构体可以被父子进程共享

下面讨论**操作一个描述符**怎么就**完成对文件的操作**：

个人理解：

1：当进程掉用open等系统调用后，会创建一个文件对象struct file，所谓**文件对象**其就是在“linux应用之\_文件IO前世今生中的文件共享”介绍到的**文件表,或者是二者之间信息很多都一样**



同时内核会选择一个未启用文件描述符，其实就是一个整形数，然后将这个**数字**与这个**文件对象**关联，

这种关联关系放在进程描述符**struct task\_struct中的files域**

2:当open成功打开一个文件后，调用write等接口传入文件描述符，则从进程描述符中的files域找到其文件表指针，这个指针可以找到文件的属性，偏移，数据块的位置，进而操作它们。

3：关闭文件后，相当于将进程描述符中的files中的**数字**与这个**文件对象**关联撤销，并释放这个文件对象所在空间（就是上面的文件表，v节点等所占空间）

20161111 王超群