# 文件系统模块

文件系统模块向用户程序提供了统一的文件访问接口，用户程序可以通过文件系统的系统调用接口访问文件，设备等资源。文件对于用户和对于操作系统内核应该是完全不同的，从用户程序角度来看，文件应该是无结构的字符流，用户通过给定文件名，要存取的文件逻辑偏移量对文件数据进行访问；而从内核的角度来说，系统中每一个文件都有一个唯一对应的索引节点，索引节点包含着进程访问该文件所需要的一切信息，内核负责对文件的管理其实是处理该文件的索引节点，文件数据占据的盘块和内核打开文件数据结构等。

文件系统模块即是建立用户角度的文件与文件物理存储之间的对应关系，将用户程序对于文件的访问操作转换到对文件实际物理存储结构上。

一般来说，文件系统主要完成以下几方面的任务：(1)文件在内核中的表示，即文件对应的索引节点，用于建立文件的逻辑结构与磁盘上的物理结构之间的映射关系。(2)文件系统提供了对打开文件的管理，维护内核中打开文件机构相关的数据结构之间的勾连关系，用户可以由打开文件号寻找到被打开文件的信息并对该文件进行相应的操作。(3)文件系统存储资源的管理，包括空闲盘块，空闲索引节点的管理。

文件系统模块尤其是文件系统调用这一部分将是Unix V6++修改最少的部分，因为文件系统操作大多通过高速缓存接口与底层硬件交互，因此只要保持高速缓存结构不变，文件系统的改动将会很有限。同时，Unix V6文件系统的设计采用了很多面向对象的设计思想，因此只需可以较好的保持原有系统的设计。文件系统中主要将会涉及FilesManager，Filesystem， InodeTable，Inode，File等类，具体将会在后面章节详细介绍。

## 文件系统模块类结构

Unix V6++中，FileManager类是文件管理模块的中心管理类，在操作系统中有且仅有一个实体，由Kernel类中m\_FileManager类引用。下图给出了文件系统模块中几个管理类之间的关系(图11.1)。

FileManager类封装了文件系统中系统调用部分，包括创建、打开、关闭和读写文件等系统调用。在文件系统相关系统调用执行期间，需要对内核中表示文件的数据结构如索引节点，内存打开文件结构以及文件系统的存储资源进行操作，因此，FileManager类还与内存索引节点表InodeTable、系统打开文件表OpenFileTable以及文件系统存储资源管理FileSystem类存在关联，FileManager类中系统调用通过调用以上三个类提供的接口进行文件操作相关的数据结构的管理。



#### 图11.1 文件系统模块主要类结构

其中，FileSystem类用于管理文件系统的存储资源，包括空闲磁盘块，空闲外存索引节点(DiskInode)这类外存资源，对这类资源的分配和释放算法，以及文件系统挂载点的信息，其中m\_Mount[0]项记录根文件系统的超级块(SuperBlock)信息。

InodeTable和OpenFileTable类用于管理文件系统模块中内存相关数据结构，主要是内存索引节点(Inode)表和系统打开文件控制块表这两个用于管理当前打开文件在内存中的管理机构。这些类的详细结构将在后面具体介绍。

## 存储资源管理—FileSystem类

FileSystem类管理外存存储资源，如空闲磁盘块，空闲外存索引节点(DiskInode)，文件系统的超级块(SuperBlock)，以及少部分和外存存储资源相关的内存数据结构，譬如文件系统SuperBlock相关的挂载点。

### 磁盘存储空间划分

文件系统的存储设备为磁盘，磁盘上的存储空间以扇区为基本单位，每个扇区512字节大小。在Unix V6++中，对磁盘设备上的存储空间划分做了如下规定，如图11.2所示。

引导

超级块

DiskInode块

**。。。。**

DiskInode块

数据块

**。。。。**

数据块

0

(200,201)

202

s\_isize+201

s\_fsize-1

DiskInode区

数据盘块

内核映像

1024

(1,199)

内核映像区

#### 图11.2 磁盘存储空间分布

磁盘上的0#扇是引导扇区，存放引导程序；Unix V6++中将紧随引导扇区其后的是内核映像区，分配用于存放内核映像的扇区区间为1# ~ 199#扇区，由于内核代码编译后生成的文件较小，所以分配这段区间已经足够。磁盘上前200个扇区存放引导程序和内核映像，这部分不归属于Unix V6++的文件系统管理。

从200#扇区开始直到磁盘末尾扇区这部分内容属于Unix V6++文件系统管理的存储资源。首先是文件系统超级块(SuperBlock)，一个SuperBlock大小为1024字节，因而它需要占据(200，201)两个扇区。

外存索引节点(DiskInode)区紧随在SuperBlock之后，占据202# ~ 1023#(或s\_isize+1)扇区区间。这部分扇区中存放文件索引节点，文件系统中每一个文件都在DiskInode区中存在唯一对应的索引节点，包含着该文件的控制信息；当然DiskInode区中还有空闲索引节点，即该索引节点目前尚不与任何文件绑定，可以在将来创建新文件时分配使用。在Unix V6++中，每个DiskInode的大小为64字节，所以每个扇区中存放8个外存索引节点。

最后从1024#盘块开始直至末尾扇区的区间作为数据盘块，其中部分已经被分配存放文件数据内容，而另一部分为空闲盘块等待分配。任何文件的数据都占用整数个盘块，即使该文件的最后一块扇区未被填满，也不会让两个文件共享同一数据盘块。

### 文件系统超级块—SuperBlock类

文件系统超级块(SuperBlock)记录了该存储设备上文件系统的管理信息，通过它可以知道该设备的文件系统中还有多少空闲磁盘块、多少空闲外存索引节点，以及这些空闲磁盘块、索引节点的位置等信息，并且可以从超级块顺藤摸瓜找到这些空闲存储资源。每一个存储设备上的文件系统都有其对应的SuperBlock，负责记录该设备上文件系统的信息。当系统中存在多个文件系统存储设备时，其中一个作为根文件系统，其余作为子文件系统挂载于根文件系统的目录树中。

在操作系统初始化时，会将磁盘上的SuperBlock复制一份留作内存副本，方便内核随时访问内存副本来获取该文件系统的管理信息。内核中使用SuperBlock类来记录存储设备上对应的外存资源的信息，其定义如下：(代码11.1)

class SuperBlock

{

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 \*/

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 \*/

};

#### 代码11.1 SuperBlock类

Unix V6++的SuperBlock类沿用原Unix V6中的存储资源管理信息块filsys数据结构而未做改动。SuperBlock类包含以下两类重要成员：

* 外存索引节点(DiskInode)管理的相关成员
  1. s\_isize：表示存储设备上DiskInode区占据的扇区数。Unix V6++中DiskInode区起始扇区号为202#，所以DiskInode区的最后盘块号为(s\_isize + 202) – 1。
  2. s\_ninode、s\_inode[100]：SuperBlock直接管理的空闲DiskInode数量和空闲DiskInode索引表，索引表中记录的是这些DiskInode的编号。

在DiskInode区中的所有节点按顺序从0开始编号，依次递增，由于每个扇区存放8个节点，DiskInode区中第一个扇区存放编号0 ~ 7的外存索引节点，第二个扇区存放编号8 ~ 15的节点，依次类推。根据空闲DiskInode索引表中的节点编号可以计算出该节点所在的扇区号和扇区内偏移量，从而将该节点读入内存使用。

* 1. s\_ilock：空闲DiskInode索引表的封锁标志。对空闲DiskInode索引表进行检查和修改的程序段是临界区，为实施临界区互斥，设置本标志。
* 空闲盘块管理的相关成员
  1. s\_fsize：表示文件系统盘块总数，即SuperBlock、DiskInode区和数据盘块区三部分的扇区数之和。
  2. s\_nfree、s\_free[100]：SuperBlock直接管理的空闲盘块数和空闲盘块索引表。
  3. s\_flock：空闲盘块索引表封锁标志，对空闲盘块进行索引表进行操作时需要设置此标志，实施临界区互斥。

### 对空闲DiskInode的管理

在DiskInode区中，空闲外存索引节点的数量可能相当大，如果全部由SuperBlock直接管理将会占用较多内存。因此，有SuperBlock直接管理最多100个空闲外存索引节点，所有这些节点的编号都列在空闲DiskInode索引表s\_inode[100]中，该表实际管理的空闲DiskInode数由s\_ninode表示。

###### 空闲外存索引节点的分配—IAlloc()

空闲外存索引节点表使用栈方式管理DiskInode。当需要分配DiskInode时，如果s\_ninode不为0，则将spb->s\_inode[--spb->s\_ninode]所指示的DiskInode分配出去。如果s\_ninode已为0，说明空闲外存索引节点表已不包含任何空闲节点。于是，就要搜索DiskInode区，将找到的空闲好顺次等入s\_inode表中，直至该表已满或已搜索完整个DiskInode区。内核中用于分配空闲外存索引节点的函数是FileSystem::IAlloc()，其流程如下：(图11.3)



#### 图11.3 FileSystem::IAlloc()

首先检查超级块索引节点表是否被上锁，如果被上锁表示当前正有其它进程也在访问索引节点表，则需要分配DiskInode的进程进入睡眠状态等待其解锁；如果索引节点表没有被上锁，则当前进程可以进行DiskInode的分配。

进而检查超级块索引节点表是否为空，如果为空的话，则先要通过搜索DiskInode区寻找空闲索引节点，填入超级块的索引节点表中，直至填满超级块索引节点表或找不到空闲索引节点，然后才可进行分配DiskInode的操作。由于此搜索过程需要等待磁盘I/O操作完成，会让当前进程睡眠并调度其它进程上台，因而其前后需要对超级块索引节点表进行上锁和解锁，构成临界区，防止其它进程在此期间操作索引节点表，譬如，其它进程也发现索引节点表为空，需要搜索DiskInode区寻找空闲DiskInode，两个进程同时扫描DiskInode区将会导致同一空闲DiskInode被重复记入超级块索引节点表中。

在从索引节点表“栈顶”获取空闲DiskInode编号之后，需要将其读入内存并验证该索引节点是否真的空闲，如果是则初始化并返回该索引节点；如果不是真的空闲，则需要放弃这个已分配的索引节点，重新从索引节点表“栈顶”获取新的索引节点编号。

###### 空闲外存索引节点的释放—IFree()

相对于DiskInode的分配过程，其释放的流程比较简单。内核中是否一个索引节点的函数是FileSystem::IFree()，其流程如下：(图11.4)



#### 图11.4 FileSystem::IFree()

当释放一个DiskInode时，如果超级块索引节点表中空闲DiskInode的个数小于100，则将该索引节点编号记入“栈顶”；只有在以下两种情况下不将释放的索引节点编号记入表中，而是任其散落在磁盘DiskInode区中：

1. 超级块索引节点表被上锁；修改超级块索引节点表需要首先获取该锁，而释放一个DiskInode时让进程为获取该锁而睡眠则代价过高。
2. 当超级块索引节点表中记录的DiskInode已满情况下，让释放的DiskInode散落在DiskInode区中。

### 对空闲盘块的管理

超级块中的空闲盘块索引表s\_free[100]用栈方式管理空闲盘块，但是最多只能直接管理100个空闲块，它实际管理的块数由s\_nfree表示。若空闲块数超过100，采用分组链式索引法对它们进行管理。

每100个空闲块构成一组(第一组99块)，余下部分也构成一组。最后一组直接由超级块中的空闲块索引表s\_free[100]管理，其余各组的索引表则分别存放在它们下一组第一个盘块的开头404个字节中。例如：若空闲块数349，则第一组包含99块，第二、三组皆为100块，第四组50块。所以s\_nfree为50，索引表s\_free也只用0~49各项。第一组的索引表存放在第二组第一块的404个字节中，其位置和格式与s\_nfree、s\_free相同； 其中，第二个字的值为0，它是空闲盘块链的结束标志，表示下面不再有空闲块可供使用，其余各组的情况与此类似(图11.5)。

**s\_nfree:50**

**s\_free[0]**

.

.

s\_free[49]

**s\_nfree:100**

**s\_free[0]**

**.**

**..**

**s\_free[99]**

**s\_nfree:100**

**s\_free[0]**

**..**

**.**

**s\_free[99]**

**s\_nfree:100**

**s\_free[0]**

**..**

**.**

**s\_free[99]**

SuperBlock

第一组空闲块

(共99块)

**。。。**

**。。。**

**。。。**

**。。。**

**。。。**

**。。。**

第四组空闲块

(共50块)

第三组空闲块

(共100块)

第二组空闲块

(共100块)

**。。。**

#### 图11.5空闲盘块分组链式索引

###### 空闲盘块的分配—Alloc()

分配空闲盘块总是从索引表中取其最后一项的值，即spb->s\_free[--spb->s\_nfree];，相当于出栈。当发现这是直接管理的最后一个空闲块被分配后(s\_nfree减1后为0)，就将该盘块的头404字读入超级块的s\_nfree和索引表s\_free[100]中，使得用间接方式管理的下一组变为直接管理。如此类推直至最后一组。当系统中最后一个空闲块被分配使用后，s\_nfree的值为1。当再次企图分配盘块时，发现spb->s\_free[--spb->s\_nfree]的值为0，说明已到空闲盘块链尾，没有盘块可供分配。由此可见，虽然通常s\_nfree解释为该时刻直接管理的空闲盘块数，但其更一般的意义是s\_free表中有意义的剩余项数。例如：s\_nfree为50时，说明s\_free[0]~ s\_free[49]的内容是有意义的。所谓有意义既包括指示一个空闲块，也可以是空闲盘块链结尾标志。

内核中用于分配空闲盘块的函数是FileSystem::Alloc()，其流程如下：(图11.6)



#### 图11.6 FileSystem::Alloc()

在开始正式分配盘块编号之前，先要获取空闲块索引表的锁，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块索引表，当前进程需要进入睡眠等待其解锁。

检查通过之后，从索引表“栈顶”得到空闲盘块编号，并且检查取出编号之后索引表是否变为空，如果为空的话表明当前空闲块是超级块目前直接管理的最后一个空闲块，那也就意味着该盘块上记录着下一组空闲盘块编号，因此在使用该盘块之前，先要将它上面记录的下一组空闲盘块编号读入到超级块直接管理的空闲块索引表中，然后才可将盘块分配作它用；如果该盘块不是栈中的最后一块，也就是说盘块上没有包含任何重要信息，因此可以直接清空后移作它用，设置超级修改标志标明SuperBlock内存副本已改动，需要在以后更新到磁盘上，随后返回新分配空闲盘块对应的缓存。

###### 空闲盘块的释放—Free()

释放空闲盘块时，将盘块号记入空闲盘块索引表中第一个未被占用的项spb->s\_free[spb->s\_nfree++]，相当于压栈。例如若s\_nfree原先只为50，则将释放块号登入s\_free[50]，然后s\_nfree加1为51。如果在记入索引表之前发现索引表已经占满，则应将当前超级块直接管理的100个空闲盘块编号写到当前正在释放的盘块上。这样，原先由超级块直接管理的100个空闲块就变成为由释放块间接管理。然后将此释放块块号压入栈底，s\_nfree置为1。内核中用于分配空闲盘块的函数是FileSystem::Free()，其流程如下：(图11.7)



#### 图11.7 FileSystem::Free()

由此可见，对空闲盘块的分配和释放类似于栈，是使用后进先出算法。但其管理机构分成两级，一级常驻内存，另一级则驻在各组的第一个盘块上，充分利用空闲盘块来存储空闲盘块编号，节约了文件系统管理数据结构占据的磁盘开销。

## 文件索引节点

文件系统中每个文件都有一个唯一对应的索引节点，其中包含了关于该文件的控制信息。索引节点是文件在操作系统内核中的内部表示，它是以静态形式存储于磁盘上，通常是在存储设备上开辟一个专门存放外存索引节点的DiskInode区，其中的索引节点按顺序从0开始编号；当需要访问某一文件时，则将该文件对应的DiskInode读入到内存索引节点表中对其进行操作。

内存索引节点表是由一组连续的内存索引节点构成的，每当需要读入一个DiskInode时，都先要在内存索引节点表中分配一个内存Inode。内存Inode基本完全拷贝DiskInode中的信息，仅作少量修改以满足管理的需要。

由此可以看到，索引节点需要区分外存索引节点DiskInode和内存索引节点Inode两种，DiskInode规定了外存索引节点在磁盘上存放的格式，而Inode则用于和文件系统模块中其它内存打开文件机构共同协作，将对文件的操作映射到对该文件实际存储盘块数据的读写操作。

### 外存索引节点—DiskInode类

外存索引节点(DiskInode)可以说是一种纯粹的数据结构定义，它只是规定了一个外存索引节点(64字节大小)在磁盘上的布局，也就是DiskInode的各个成员的排布的先后顺序，当内核将存储DiskInode的磁盘块读入之后，必须按照什么样的格式去解析该盘块中的数据，才可以正确获得DiskInode中的文件控制信息。

Unix V6++中，DiskInode类基本沿用原Unix V6中外存索引节点的定义，仅仅在用于文件逻辑块到物理盘块号转换的索引表大小上有所变化，其定义如下：

class DiskInode

{

public:

unsigned int d\_mode; /\* 状态的标志位 \*/

int d\_nlink; /\* 文件联结计数，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 \*/

};

#### 代码11.2 DiskInode类

其中各个成员变量名都以d\_为前缀，表示属于DiskInode的成员，方便在代码中和Inode类中成员以i\_作为前缀进行区分，而DiskInode类与Inode类中绝大部分成员完全吻合的，内存Inode仅仅是从磁盘上读入的DiskInode对象的副本。

DiskInode的成员大致可以分成以下几类：

* 文件长度及数据存储的物理位置

**d\_size**表示文件长度，单位是字节；

**d\_addr[10]**数组是逻辑文件和物理文件的基本地址对照表，或称为基本索引表。索引表的完整结构和用法将在后面具体解释。

* 文件主的用户标识数及组标识数

**d\_uid**和**d\_gid**分别是文件主用户标识数和文件主组标识数；当一个进程企图对某一文件进行操作时，将其p\_uid和文件主用户标识以及u\_gid和文件主组标识数进行比较就可以判断出该进程是否属于文件主或其同组用户。

* 文件工作方式信息

**d\_mode**包含了文件工作方式信息，主要有以下四方面：

1. 文件类型

文件系统中的文件分为四类，它们是一般数据文件 、目录文件、块设备文件和字符设备文件，各种文件类型在d\_mode中比特位定义如下：(代码11.3)

const unsigned int IFMT = 0x6000; /\* 文件类型掩码 \*/

const unsigned int IFDIR = 0x4000; /\* 文件类型：目录文件 \*/

const unsigned int IFCHR = 0x2000; /\* 字符设备特殊类型文件 \*/

const unsigned int IFBLK = 0x6000; /\* 块设备特殊类型文件，0表示常规数据文件 \*/

#### 代码11.3 d\_mode中文件类型常量定义

块设备文件和字符设备文件也称为特殊文件，以表示他们和一般数据文件、目录文件有所区别。对于特殊文件，数组d\_addr[]不再是基本索引表。仅用d\_addr[0]表示块设备或字符设备的设备号，高8位为主设备号，低8位为次设备号。

1. 用户访问权限

文件系统将用户分为三类：文件主、同组用户和其他用户。存取权分为读、写、执行三种操作的组合。文件主对文件的读、写、执行三种权限在d\_mode中的比特位定义如下，而同组用户和其他用户的读、写、执行权限占据d\_mode中的最低6比特(代码11.4)。

const unsigned int IREAD = 0x100; /\* 文件主对文件的读权限 \*/

const unsigned int IWRITE = 0x80; /\* 文件主对文件的写权限 \*/

const unsigned int IEXEC = 0x40; /\* 文件主对文件的执行权限 \*/

#### 代码11.4 d\_mode中文件访问权限常量定义

1. 设置文件主标识数

在执行该文件时，是否需将进程p\_uid和u\_gid设置为文件主标识数d\_uid和文件主组标识数d\_gid。

1. 文件长度类型信息

文件系统中的文件根据其大小可以分为小型、大型和巨型文件。在原Unix V6中通过检查d\_mode字段中的ILARG位来区分是小型文件还是大型或巨型文件，这是因为在原来Unix V6中Inode的基本索引表中的项存在二义性，对小型文件直接是文件数据的物理块号，而对大型或巨型文件则是存储一次或二次间接索引表的盘块号。

在Unix V6++中，由于对基本索引表d\_addr[10]数组的各项进行了重新规划，所以不存在以上所述的二义性，因此也就不需要判断d\_mode字段中的ILARG位。

* 文件联结计数

**d\_nlink**表示该文件在目录树中不同路径名的数量。

* 时间信息

**d\_atime**表示对该文件的最近一次访问的时间；**d\_mtime**表示最近一次对该索引节点进行修改的时间。

### 内存索引节点—Inode类

每当需要读入一个外存索引节点时，都先要在内存索引节点表中分配一个内存Inode。内存Inode基本完全拷贝DiskInode中的信息，因此内存索引节点的构成与外存索引节点基本一样，这里只对两者差异部分进行解释。Inode类定义如下：(代码11.5)

class Inode /\* 仅列出与DiskInode差异部分 \*/

{

public:

unsigned int i\_flag; /\* 状态的标志位 \*/

int i\_count; /\* 引用计数 \*/

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 \*/

int i\_lastr; /\* 存放上一次读取文件的逻辑块号，判断是否需要预读 \*/

};

#### 代码11.5 Inode类

内存Inode中特有的成员有以下几个：

* Inode状态标志位

i\_flag用于指示该内存Inode当前的状态，如Inode是否被上锁，是否有进程正在等待该内存Inode解锁，内存Inode副本是否已被修改需要更新到磁盘上，Inode是否为子文件系统挂载点等等，其标志位使用枚举变量INodeFlag定义如下：(代码11.6)

enum INodeFlag

{

ILOCK = 0x1, /\* 索引节点上锁 \*/

IUPD = 0x2, /\* 内存Inode被修改过，需要更新相应外存索引节点\*/

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解锁 \*/

ITEXT = 0x20 /\* 内存Inode对应进程图像的正文段 \*/

};

#### 代码11.6 Inode状态标志位定义

* 对应外存索引节点的位置信息

内存Inode是磁盘上DiskInode的副本，并且两者之间存在一一绑定的关系，即任何DiskInode在内存中只可能有一个副本，无论有多少进程需要访问该外存索引节点，它最多被载入内存一次，分配一个空闲内存Inode对象，该内存Inode由所有进程共享；同时被拷贝到内存Inode中的索引节点数据需要知道它来自于哪个DiskInode，以便将来内存副本被修改后更新到对应DiskInode中去，因此Inode类中包含了用于记录DiskInode位置信息的设备号i\_dev和外存索引节点编号i\_number。

* 引用计数

i\_count指出该索引节点当前活跃的实例数目。譬如有进程通过系统调用Open()打开文件，则该内存Inode的引用计数会+1。引用计数是判断一个Inode是否空闲的的依据，如果i\_count为0，则表示该Inode空闲，可以被分配作它用，通常需要新分配一个Inode时，就会从内存索引节点表中寻找i\_count等于0的空闲Inode；反之则表示当前有进程正在引用该Inode。当内核中最后一个引用进程释放该Inode时，内核会将其引用计数值清零，使之成为空闲Inode。

* 预读判断

i\_lastr记录上一次读取文件的逻辑块号，在文件读操作过程中，会判断进程是否在按顺序方式读取文件，如果是的话，那么根据程序的局部性原理，该进程很有可能会在短时间之后读取当前逻辑块的下一块，因此在读取当前块时将下一块一并读入内存，如果预读命中的话将会提高进程前进的速度。

### 文件索引结构

有关文件数据在磁盘上的位置信息包含在索引节点中的基本索引表i\_addr[10]数组中，磁盘上的每一盘块都有唯一的编号，所以基本索引表i\_addr[10]是一系列物理盘块号的集合。文件的数据可以散落在磁盘数据盘块区域的任意角落，所以基本索引表实际上是一张记录文件数据占据的物理盘块编号的清单。

由于基本索引表做多只能存放10个物理盘块号，以此作为文件大小限制显然无法满足用户要求，需要使文件的索引结构可以组织大型和巨型文件，同时又要保持基本索引表体积较小。

在Unix V6++中，由于对基本索引表i\_addr[10]数组的各项进行了重新规划，避免了让基本索引表中的相同项在应用于小型文件与大型或巨型文件时表示不同的含义，存在二义性，而需要借助ILARG标志位来区分是小型文件还是大型或巨型文件。

根据文件长度的不同，文件系统中的文件根据其大小可以分为小型、大型和巨型文件。下面分别对这三种类型文件如何使用基本索引表进行介绍。

* 小型文件

Unix V6++中规定小型文件使用基本索引表中i\_addr[0] - i\_addr[5]作为直接索引表，也就相当于限制了小型文件长度范围是0 ~ 6个数据块，每个数据盘块大小等于一个扇区512字节。其中，文件的逻辑块号n对应的物理块号记录在i\_addr[n]中。譬如，i\_addr[2]的值为2058表示该文件的第3块数据(偏移量1024 ~ 1535字节)占据的物理盘块编号为2058。

* 大型文件

当文件的长度超出小型文件限制时，就需要采用大型文件的索引结构了。大型文件除了使用i\_addr[0] - i\_addr[5]记录该文件前6个物理块号之外，还用到i\_addr[6]、i\_addr[7]两项，每一项指向一个含有物理块号明细表的一次间接索引盘块，每个索引表块可以容纳128个文件数据物理块号，因此大型文件的长度范围是7 ~ (128 \* 2 + 6)个数据块。如果内核需要通过一次间接索引表访问数据，那么先要将一次间接索引表块读入内存，从中找到逻辑块号对应的物理块号，然后再读取该物理块上的文件数据。

* 巨型文件

如果文件长度比大型文件还要大，则除了用到基本索引表中的i\_addr[0] - i\_addr[5]以及i\_addr[6]、i\_addr[7]两项记录该文件前128 \* 2 + 6个物理块号之外，还要使用i\_addr[8]、i\_addr[9]记录二次间接索引盘块表的物理块号。

二次间接索引表作用类似于一次间接索引表，表中的每一项都记录着它所指向的一次间接索引表的物理块号，然后再由该一次间接索引表中的项记录文件数据的物理块号。每张二次间接索引表最多可记录128张一次间接索引表的物理块号，由此可以算出巨型文件的长度范围是(128 \* 2 + 6 ) < FileSize <= (128 \* 128 \* 2 + 128 \* 2 + 6)。

综上所述，Unix V6++中小型、大型和巨型文件的索引结构由下图给出。

一次间接块号

二次间接块号

其他字段

i\_addr[0]

i\_addr[1]

i\_addr[2]

i\_addr[3]

i\_addr[4]

i\_addr[5]

i\_addr[6]

i\_addr[7]

i\_addr[8]

i\_addr[9]

索引节点

直接块号

一次间接索引块

二次间接索引块

一次间接索引块

#### 图 11.8 Unix V6++中文件索引结构

### 逻辑块号到物理块号的转换—Bmap()

进程存取文件中的数据时使用的是字节偏移量，将整个文件看作是从字节0开始直至文件大小的字节流。内核将用户的字节流转换成逻辑块的看法，把文件分割成大小相等的逻辑块，文件从第0个逻辑块开始直至文件大小对应的逻辑块号结束。

对文件进行读、写时，总需要先将文件逻辑号变换成文件在存储设备上的物理块号，从上面关于Unix V6++中文件索引结构的内容可以看出，文件逻辑块与物理块之间的对应关系包含在索引节点中的i\_addr[10]数组中，于是从逻辑块号到物理块号的转换就是对索引节点中基本索引表的操作了。

内核中执行这项转换工作的是内存索引节点Inode类的成员Bmap()函数，其流程大致如下：



#### 图11.9 Inode::Bmap()

Inode::Bmap()函数中进行逻辑块号转换的执行路径大致分为两条，分别是用于对小型文件以及大型或巨型文件。

首先判断文件长度属于小型文件还是大型或巨型文件，如果为小型文件，从基本索引表i\_addr[0 ~ 5]中获得物理盘块号即可。如果该逻辑块号还没有相应的物理盘块号与之对应，即相应索引表项为0，则分配一个物理块。这通常发生在对文件的写入，若写入位置超没有对应的数据内容，即对当前文件进行扩充写入，就需要分配额外的物理盘块存放文件数据，并为之建立逻辑块号与物理盘块号之间的映射。

如果是大型或巨型文件，则通过计算出逻辑块号在基本索引表的对应项，该项记录着间接索引表块的物理块号，对于大型文件为一次间接索引表块，而对于巨型文件则为二次间接索引表块，根据物理块号读入此一次(或二次)间接索引表块，并根据要转换的逻辑块号计算出位于一次间接索引表中的表项下标，从而得到实际文件数据的物理块号。须注意，巨型文件要比大型文件增加一环节，即依据逻辑块号计算得到二次间接索引表中表项的下标，找到并读入一次间接索引表块，然后按照大型文件操作一次间接索引表块方法找到数据盘块。

具体来说，如果需要转换的逻辑块号是0 <= lbn < 6，那么直接返回基本索引表i\_addr[lbn]中相应值；如果逻辑块号在6 <= lbn < 128 \* 2 + 6之间，那么为一次间接索引，通过i\_addr[6]或i\_addr[7]找到一次间接索引表块，然后在这一索引表块中找到对应的物理块号；如果逻辑块号在(128 \* 2 + 6 ) <= lbn < (128 \* 128 \* 2 + 128 \* 2 + 6)之间，那么通过i\_addr[8]或i\_addr[9]找到二次间接索引表块，根据二次间接索引表块找到一次间接索引表块，进而最后找到对应的物理块号。

## 内存打开文件管理机构

如前所述，外存上文件管理机构记录了整个文件系统各种资源的完整信息，包括对SuperBlock、空闲数据盘块、外存索引节点的统筹管理。同时，为了便于对文件索引节点的查询和修改，管理和记录进程的打开文件以及对这些打开文件的读写位置等信息，内核还设置了一个内存管理机构。该机构用于管理当前被进程打开的文件而非文件系统中的所有文件，也就是说这一机构不应当是外存上文件管理机构的全部拷贝，而应适宜于管理最近正在使用的一些文件。

并且，为了系统管理和用户使用的方便，对这些已经打开的文件进行存访、处理时也不再使用符号路径名，而只要求使用整型编号数。但系统很难判断哪些文件是用户最近阶段需要使用的，因此只能由用户以一定方式通知系统，这就是打开文件系统调用open。文件打开后，它由在内存中的一套机构管理；文件被关闭时再从这种管理机构中退出。

内存中的文件管理机构包括内存索引节点Inode、打开文件控制块File以及进程打开文件描述符表OpenFiles三部分数据结构以及为进程打开文件所建立的内核数据结构之间勾连关系。

关于内存Inode及其内含的文件索引结构相关内容前文已经介绍，因此以下只解释打开文件控制块File以及进程打开文件描述符表OpenFiles。

### 打开文件控制块—File类

一个文件可以被同一进程或不同进程，用同一或不同路径名，相同或互异的操作要求(读、写)同时打开。而Inode基本上包含的是文件的物理结构、在目录结构中的勾连情况、对各类用户规定的存取权等静态信息，不能容纳这些个别的在打开文件结构中应该加以反映的动态要求。为此内核设置打开文件控制块File，以便记录进程打开文件的读、写请求类型，文件读写位置等动态信息，其定义如下：

class File

{

public:

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 \*/

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 \*/

};

#### 代码11.7 打开文件控制块File类

其中，f\_flag包含了对打开文件请求类型，包括读、写以及管道类型，使用枚举类型定义(代码11.8)。

enum FileFlags

{

FREAD = 0x1, /\* 读请求类型 \*/

FWRITE = 0x2, /\* 写请求类型 \*/

FPIPE = 0x4 /\* 管道类型 \*/

};

#### 代码11.8 打开文件请求类型

f\_inode指向一个打开文件的内存Inode，不同进程(或同一进程以不同读、写请求类型)打开同一文件会分配各自独立的File控制块对象，这些File控制块对象的f\_inode都会指向同一个内存Inode，从而实现在多个打开文件控制块File对象之间共享同一文件的索引节点。

f\_count是该File控制块的引用计数，表示当前引用该文件控制块的进程数。多个进程可以共享同一打开文件File控制块对象，即共享对同一文件的读写请求方式和文件读、写偏移量位置，这种共享行为通常发生在父子进程之间，父进程在创建子进程时，会将其打开文件的File控制块引用计数f\_count值+1。f\_count的值为0则表示该File控制块空闲，可以分配作它用。

f\_offset是相对应打开文件进行读、写的位置指针。文件刚打开时，读、写位置指针初始值为0。每次读、写后，都将其移到已读、写部分的下一个字节。

### 进程打开文件描述符表—OpenFiles类

每个进程都有一张进程打开文件描述符表，每一个被进程打开的文件都在描述符表中占据一项。不同于内存索引节点表以及系统打开文件这类属于内核全局的数据结构，进程打开文件描述符表是属于每个进程各自所有的，记录该进程的打开文件情况。

在Unix V6++中，我们使用OpenFiles类管理每个进程的打开文件，并且因为打开文件描述符表是进程独有的数据，因而将OpenFiles类的实例放在User类中作为成员变量。进程打开文件描述符表OpenFiles本质上是一个File对象的指针数组，其中每一项都是一个指针，指向一个打开文件控制块File对象(代码11.9)。

class OpenFiles

{

public:

static const int NOFILES = 15; /\* 进程允许打开的最大文件数 \*/

int AllocFreeSlot(); /\* 在打开文件描述符表中分配一个空闲表项 \*/

private:

File \*ProcessOpenFileTable[NOFILES]; /\* File对象的指针数组 \*/

};

#### 代码11.9 打开文件描述符表OpenFiles类

进程打开一个文件时，不但要找到或分配一个内存Inode，分配一个File控制块，建立二者的勾连关系，还要在其打开文件描述符表中分配一项，并指向相应的File控制块。

NOFILE一般是15所以每个进程最多可同时打开15个文件，从进程的角度看，打开文件被抽象为文件描述符fd，该描述符为相应被打开文件在OpenFiles.ProcessOpenFileTable[]数组中的索引。一旦通过Open()系统调用打开文件后，对该打开文件的读、写直接使用其打开文件描述符fd，不再使用文件路径名。

OpenFiles类中提供AllocFreeSlot()成员函数用于在打开文件描述符表中分配一个空闲表项。该函数线性扫描进程打开文件描述符表，寻找File指针为NULL的空闲项分配，并将该空闲项在OpenFiles.ProcessOpenFileTable[]数组中的索引作为打开文件描述符fd，返回给执行Open()系统调用的进程。

创建一个子进程时，它复制了父进程User中的OpenFiles对象，因此也就继承了父进程的全部打开文件。此时，各相应File控制块中的f\_count的值+1，所以通常f\_count是共享该File控制块的进程数。但是在同一打开文件描述符表中也可以同时有几项指向同一个File控制块，所以从更一般意义说，f\_count是各个进程打开文件描述符表中包含的指向该File控制块的指针数。

进程关闭一个先前打开过的或由父进程那里继承过来的打开文件时，要清空OpenFiles.ProcessOpenFileTable[]数组中相应的项，并将相应File控制块中的f\_count减1。若该项减为0，说明它已成为空闲项，可再分配它用。

### 系统打开文件表—OpenFileTable类

在Unix V6++中，我们使用OpenFileTable类来封装系统打开文件表，其本质上是一个打开文件控制块File[NFILE]数组，数组的每一项都是一个File控制块，当其被分配出去记录进程打开文件的动态信息后，由该进程打开文件描述符表中的对应File指针指向它。NFILE定义为100，表示系统中总共有100个打开文件控制块File对象可供使用。OpenFileTable类的具体定义如下：(代码11.10)

class OpenFileTable

{

public:

static const int NFILE = 100; /\* 打开文件控制块File控制块的数量 \*/

File\* FAlloc(); /\* 在系统打开文件表中分配一个空闲的File控制块 \*/

Void CloseF(File\* pFile); /\* 释放对File控制块的引用 \*/

private:

File m\_File[NFILE]; /\* 系统打开文件表 \*/

};

#### 代码11.10系统打开文件表OpenFileTable类

系统打开文件表中的File控制块数组供系统中所有进程使用，每当一个进程需要打开文件时，都要在File[NFILE]数组分配一个空闲项，用于记录进程本次打开文件的读写请求类型，以及记录进程读写该文件的偏移量指针等动态信息。可以说，OpenFileTable类记录了当前系统中所有进程对文件进行读写操作的状态信息，以及一些尚未被分配出去的空闲File控制块。

OpenFileTable类还提供了两个成员函数FAlloc()和CloseF()分别用于在系统打开文件表中分配一个空闲的File控制块以及进程关闭文件时释放对File控制块的引用。

FAlloc()函数进行File控制块的分配，分配算法比较简单，线性搜索OpenFileTable中的File[NFILE]数组，寻找其中引用计数f\_count为0的空闲File控制块对象，如果分配成功则与进程打开文件描述符表中的对应File指针建立勾连关系；否则的话设置出错标志指示Open系统调用执行失败。

CloseF()函数用于进程关闭文件时释放对File控制块的引用。由于File控制块可能被多个进程共享，所以CloseF()函数一般只是递减该File控制块的引用计数f\_count值，并不释放该File控制块f\_inode指向的内存索引节点。只有在当前进程为引用该File控制块的唯一进程情况下，当前进程释放才将File控制块及其指向的内存Inode一起释放，即将f\_count值设置为0，表示该File控制块为空闲，并调用IPut()函数将对应内存Inode也释放成为空闲内存Inode。

### 内存索引节点表—InodeTable类

之前已经介绍了内存索引节点Inode类，该数据结构是用于当进程需要访问某一文件时，则将该文件对应的外存索引节点读入到内存之后才能对其进行操作。因此内核在内存中设置了一张内存索引节点表，该表是由一组连续的内存Inode构成的，每当需要读入一个DiskInode时，都先要在该表中分配一个空闲内存Inode，用于存放读入的DiskInode信息。内存Inode基本完全拷贝DiskInode中的信息，进增、减少部分成员用于管理的目的。

在Unix V6++中，使用InodeTable类来封装内存索引节点表数据结构，本质上来说是一个内存索引节点m\_Inode[NINODE]数组，NINODE定义为100。该类主要用于管理内存索引节点的获取和释放，其定义如下(代码11.11) ：

class InodeTable

{

public:

static const int NINODE = 100; /\* 内存Inode的数量 \*/

Inode\* IGet(short dev, int inumber); /\* 获取指定的索引节点 \*/

void IPut(Inode\* pNode); /\* 释放索引节点 \*/

private:

Inode m\_Inode[NINODE]; /\* 内存Inode数组 \*/

};

#### 代码11.11内存索引节点表InodeTable类

###### 索引节点的获取—IGet()

每当需要访问一个文件时，都要找到该文件对应的DiskInode，并预先在内存索引节点表中分配一个空闲内存Inode项，然后将DiskInode的信息拷贝至内存Inode中。然而也有可能要访问文件的索引节点已经在内存中，之前强调过无论有多少进程需要访问一个DiskInode，它都不应该被读入内存两次，也就是说最多只能有一个内存Inode副本，因此在索引节点的获取过程中需要将这种情况考虑进去。

Unix V6++中使用InodeTable::IGet(dev, inumber)函数来获得指定DiskInode的内存拷贝，其接受设备号dev和外存索引节点编号inumber来定位DiskInode。其大致流程如下：(图11.10)



#### 图11.10 InodeTable::IGet(dev, inumber)

InodeTable::IGet()函数首先检查要读入DiskInode是否已有内存拷贝，如果有但是该内存Inode被上锁，表示该Inode正在被另一进程使用，于是当前进程对该Inode增设IWANT标志指明有进程正在等待个Inode被解锁，然后当前进程睡眠，直至另一进程使用完这个Inode将其解锁时会唤醒等待使用该Inode的睡眠进程；如果没有上锁或Inode已经解锁，则进一步检查该Inode是否用于连接子文件系统，如果是则查找子文件系统根目录Inode并将其返回；经过以上判断后当前进程已经获得了对该Inode的使用权，于是增加其引用计数，并增设ILOCK标志对其上锁，以避免其它进程同时访问该Inode导致的数据不一致性。

如果内存中还没有相应DiskInode的内存拷贝，则尝试在InodeTable中分配一空闲Inode项，分配成功则对其进行适当初始化，使其能够反映该Inode来自于哪个外存索引节点，增加内存Inode的引用计数并上锁，然后驱动磁盘读入相应DiskInode的内容，并返回该内存Inode指针。如果InodeTable已满，分配空闲Inode失败的话，则直接返回NULL。

需要注意，进程并没有在这里睡眠等待直至内核中有空闲的Inode项可供分配，因为系统中内存Inode的分配和释放是由用户程序调用Open和Close系统调用间接控制的，内核无法保证在较短的时间内系统中会有空闲Inode可用，一个因为等待空闲Inode项而睡眠的进程可能一直无法醒来，相当于进入“假死”状态，所以直接返回NULL表示系统调用在申请空闲Inode项环节失败，由用户程序来决定下一步该怎么走。

###### 索引节点的释放—IPut()

当进程关闭一个文件时，需要释放该文件对应的索引节点。由于同一文件可能同时被多个进程打开，这些进程共享同一个内存Inode，即这个内存Inode可能被多个File控制块指向，因此IPut()函数通常只是递减该Inode的引用计数值，并不真正释放内存索引节点。

Unix V6++中使用InodeTable::IPut()函数来释放当前进程对该内存Inode的引用，其大致流程如下：(图11.11)



#### 图11.11 InodeTable::IPut()

InodeTable::IPut()函数首先检查当前进程是否为引用该内存Inode的唯一进程，如果是则准备释放该内存Inode，将其上锁，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。

在释放内存Inode过程中考虑是否要将对应的DiskInode一并释放掉，通过判断该文件有没有目录路径指向它，如果发现该文件已被用户删除，仅仅因为当前进程正在使用而没有释放其占据的DiskInode和数据盘块，那么此时当前进程释放该文件占据的数据盘块，释放其DiskInode收归SuperBlock直接管理，通常情况下文件没有被用户删除则只需要将内存拷贝更新至对应DiskInode即可。

然后解锁内存Inode，唤醒因等待该Inode解锁而进入睡眠的进程，之所以前面已经判断过当前进程是引用该内存Inode的唯一进程，而这里又要唤醒潜在可能等待该Inode的睡眠进程，这是因为在将内存Inode更新到对应DiskInode过程中，当前进程可能因为磁盘操作而睡眠，在此期间系统中对此Inode的需求可能发生变化，譬如有新的进程打算使用该Inode发现已被上锁，因此需要在此处唤醒潜在可能的睡眠等待进程。随后则可以清空该Inode的标志位将其转为空闲Inode项。

而如果当前释放进程不是引用该Inode的唯一进程，释放过程就相当简单了，只需要递减该Inode的引用计数，然后解锁Inode，唤醒等待使用该Inode的睡眠进程。

### 内存打开文件结构勾连关系

内核中用于管理打开文件的数据结构包括：进程打开文件描述符表OpenFiles、File控制块构成的系统打开文件表OpenFileTable以及内存索引节点表InodeTable三者之间的勾连关系如下图所示：(图11.12)

系统打开文件表

**。。。**

**。。。**

**。。。**

f\_count : 2

f\_inode

f\_count : 2

f\_inode

f\_count :1

f\_inode

f\_count :1

f\_inode

f\_count :1

f\_inode

f\_count :1

f\_inode

f\_count :1

f\_inode

i\_count : 1

i\_count : 1

i\_count : 1

i\_count : 1

i\_count : 2

i\_count : 1

进程pb

进程pc

内存索引节点表

进程pa

#### 图11.12 内存打开文件结构勾连

其中进程pb是进程pa的子进程，它仍保留了两个从进程pa继承过来的打开文件，又自行独立打开了另外2个文件。进程pc独自打开的一个文件正好也已经由进程pb打开。进程pb、pc在打开这一文件时，使用的可能是同一个路径名，也可能是不同路径名。这一点在打开文件结构中不能也没有必要反映。

一个进程的0#打开文件称为标准输入文件，1#打开文件称为标准输出文件。他们一般就是与该进程相关连的终端的输入、输出部分。这两个文件最初由系统自动打开，然后在生成子进程时，由子进程继承共享。子进程继承共享父进程的所有打开文件有利于统一进程族共享使用同一终端，也有利于构成进程之间的通讯机构pipe。

## 目录

目录是使文件系统具有树形结构的那些文件，目录在路径名到索引节点的转换中承担着重要作用。目录是文件，它是由一系列被称为目录项的数据构成的一张表，每个目录项由一个索引节点号和一个当前目录下的文件名(或子目录名)构成。

### 文件目录结构

Unix V6++中采用树形带交叉勾连的目录结构。每张目录表也是一个文件，整个目录结构系统包含有若干个目录文件，每个目录文件由一系列目录项组成。目录项是目录文件的基本构成单位，每个目录项由32个字节组成，一个数据盘块可容纳16个目录项。在Unix V6++中使用DirectoryEntry数据结构定义目录项：(代码11.12)

class DirectoryEntry

{

public:

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

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 \*/

};

#### 代码11.12 目录项DirectoryEntry类

在目录项中前4个字节为相应文件的外存索引节点DiskInode号，它是该文件的内部标识，后28个字符为文件节点名，它是该文件的外部标识，所以文件目录项为其内、外部标识建立了对照关系。依据文件名如果能够找到相应目录项，那么也就非常方便的找到该文件的索引节点信息。

每个文件系统都有一个根目录文件，其索引节点是相应文件存储设备上DiskInode区中的0# DiskInode，位置固定很容易找到，下图给出了一个从文件系统根目录Inode开始的目录结构示例(图11.13)。

/a0/b0

或/a0/bx0

a0

。。。

a31

ax0

。。。

ax3

b0

。。。

b31

bx0

。。。

bx3

Inode

Inode

根目录文件

一般数据文件/a31或/a0/bx31

/ax0

/ax31

/a0/b0

图例

Inode

目录文件

存储块

数据文件

存储块

根目录Inode

#### 图11.13 树形目录结构示例

### 目录结构中的勾连

文件系统的目录结构中可以带有交叉勾连，图11.13中用虚线画出的就是这种勾连。在下图中对目录结构中的勾连情况进一步说明(图11.14)。

**。。。**

name1

**。。。**

**。。。**

name2

**。。。**

**。。。**

**d\_nlink:2**

目录文件/a/b

目录文件/c/d

DiskInode-n

文件数据块

#### 图11.14 目录结构中的勾连

设一个数据文件的原路径名为/a/b/name1，在图11.14的上半部分所示。如果为其再起一个路径民/c/d/name2，则在目录文件/c/d中就要构成一新目录项，其节点名部分填入name2，其索引节点号部分填入n，DiskInode-n中的i\_nlink值加1。这样就有两个目录项同时指向DiskInode-n，这就是勾连。在逻辑结构上，/a/b/name1、/c/d/name2对于DiskInode-n的地位是相同的。

单独取消/a/b/name1或/c/d/name2都不能取消DiskInode-n及其代表的文件实体。只有二者都取消才意味着DiskInode-n已游离在目录结构之外。此时，如果没有一个进程正在使用相应文件，那么就可以释放DiskInode-n和相应文件实体占用的所有存储资源。取消文件路径名/a/b/name1或/c/d/name2都称为解勾。解勾时，除清除相应目录项外，对DiskInode中的i\_nlink要作减1处理。

### 目录的搜索—NameI()

目录的搜索过程实际上是将路径名到索引节点转换的过程，用户程序对文件的存取是通过路径名的，而内核中使用的是索引节点号而不是路径名字符串，所以就需要路径名转换为索引节点号。

Unix V6++中执行目录搜索的是FileManager::NameI()函数，它负责将用户给出的路径名转换为对应文件的索引节点，其大致流程如下：(图11.15)



#### 图11.15 FileManager::NameI()

每个进程都与一个当前工作目录相联系，进程的User中包含一个指向当前目录内存Inode 的指针。在NameI()函数进行目录搜索的过程一般都是从该进程的当前工作目录开始的，除非路径名是以'/'开始，则表示要从根目录开始搜索。无论是从当前工作目录或是从根目录开始搜索，NameI()函数都可以直接从User中或是FileManager.rootDirInode中找到相应的内存Inode。

从当前工作目录或根目录的内存Inode起始，逐级搜索路径分量对应的下一级目录(或文件)索引节点号。在NameI()函数每次循环期间，如果当前路径分量不是路径中的最后一个分量，都要验证该路径分量确实是目录，否则就违反了非目录文件只能作为目录树中的叶子节点的规定。

在每一级搜索当前工作目录过程中，根据用户给定的路径名中每一个路径分量，在相应的目录文件中目录项逐个匹配。由于目录文件的目录项数据在磁盘上，所以首先需要通过目录文件的索引节点，从目录文件的第0块逻辑块开始文件末尾，依次将每一块的逻辑块号转换为对应的物理块号并读入缓存，然后将当前路径分量与缓存中的目录项数据比较，如果匹配成功，则意味着完成了当前一级上目录的搜索，记录下一级路径分量的DiskInode号，然后释放读入目录项数据的缓存以及当前工作目录的索引节点，获取下一级目录(或文件)DiskInode的内存副本，为后面下一级目录搜索做好准备。

如果已经比较完该目录文件的所有目录项仍然没有找到匹配项，则记录下目录文件中的首个空闲目录项的偏移量位置，用于为新创建文件分配目录项做准备。因为匹配失败的情况对于打开、删除文件时的路径搜索意味着查找失败，而对于创建文件则恰恰需要这种匹配失败的发生。

NameI()函数接受一个enum DirectorySearchMode枚举类型的传入参数，该参数用于指定目录搜索模式，其定义如下：(代码11.13)

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 \*/

};

#### 代码11.13 目录搜索模式DirectorySearchMode

DirectorySearchMode.OPEN表示要根据给出的文件路径名，在目录结构中找到相应的目录项。打开文件时进行的目录搜索属于这一类。DirectorySearchMode.CREATE表示在目录结构中要形成一新目录项。创建文件时的目录搜索属于这一类。DirectorySearchMode.DELETE表示应删除给定路径名的相应目录项。用于删除文件时进行的目录搜索。

不论对于哪种请求，NameI()函数都有执行成功或失败两种可能。如果失败，则将出错码送入User.u\_error中，函数返回NULL。如果执行成功，则对于不同的目录搜索模式具有以下不同含义：DirectorySearchMode.OPEN返回搜索到的目录或文件的Inode指针；DirectorySearchMode.CREATE返回NULL的同时将当前工作目录Inode指针保存在User.u\_pdir中，并记下空闲目录项的偏移量，用于随后写入目录项内容。DirectorySearchMode.DELETE返回要删除目录项所在目录文件的内存Inode指针，同时记下要删除目录项的偏移量，为随后清空该目录项做好准备。

## 文件的读、写操作过程

在Unix V6++中，文件系统的系统调用在内核中的处理函数都包含在了FileManager类中，由该类向上对用户程序提供文件的打开、关闭以及文件读写的功能，同时向下管理着内存中打开文件管理机构和文件系统存储资源等，可以说FileManager类起着衔接用户对文件的操作以及对内核数据结构相应处理的作用，这里要解释的关于文件的读、写操作过程相关函数基本都是来自于FileManager类中。

### 读、写文件公共过程—Rdwr()

系统调用read和write的格式一般如下：(代码11.14)

nbytes = read(fd, buf, count);

nbytes = write(fd, buf, count);

#### 代码11.14 read、write系统调用格式

其中，fd是打开文件描述符编号，buf是用户进程中目标数据区内存首地址，count是要读、写的字节数，返回值nbytes表示实际读、写的字节数。

read和write系统调用都会执行内核中的文件读、写公共函数FileManager::Rdwr()，只是使用不同参数表明要进行的读、写操作类型，read系统调用传递给Rdwr()的参数为File::FREAD，而write系统调用传递的参数为File::FWRITE。Rdwr()的大致流程如下：(图11.16)

Rdwr()函数首先根据read、write系统调用的打开文件描述符fd参数获取打开文件控制块结构，然后检查本次读、写请求类型是否符合对该文件的打开方式，如果不符合则拒绝执行读、写操作，譬如要对一个以只读方式打开的文件进行写操作，显然不会通过此处的检查；检查通过则继续。

随后将read、write系统调用的参数如目标数据区首地址、要求读/写的字节数送入进程User中的文件I/O参数成员User.u\_IOParam。判断本次读、写是针对普通文件还是管道文件，然后分别调用相应的读、写函数，就普通文件而言，最后调用到的实际文件读、写函数分别是Inode::ReadI()和Inode::WriteI()，由它们具体负责经由高速缓存读取或写入文件数据。最后Rdwr()函数根据读、写字节数，修改File控制块中文件读写字节偏移量，并设置系统调用返回值为实际读、写的字节数。



#### 图11.16 FileManager::Rdwr()

### 文件I/O参数--IOParameter类

IOParameter类是对原Unix V6中User结构中记录文件I/O参数的字段提取出来并封装，其定义如下：(代码11.15)

class IOParameter

{

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 \*/

};

#### 代码11.15 文件I/O参数 IOParameter类

m\_Base为要读、写的用户目标区域的首地址，在读、写操作执行的过程中，它不断从内存低地值向高地址方向移动来指向当前用户目标区中正要读、写的字节位置。

m\_Offset记录当前读写文件中的字节偏移量，需要注意的是，打开文件控制卡File中的f\_offset与这里m\_Offset都是表示文件的字节偏移量，而两者的区别在于，譬如进程要从文件读取600字节的内容，假设对该文件当前访问字节偏移量初始值为0，由于读取的文件数据需要经由高速缓存送入用户目标内存区，因此一次能传送的字节数受到缓存512字节大小的限制，m\_Offset值的变化过程为读取前512字节后，m\_Offset从0增加到512，读完剩余的88字节后，m\_Offset字节的值增加到600；而File控制块中的f\_offset的值则是在本次I/O请求执行结束之后直接从0增加至600字节。换言之，m\_Offset记录的是一次I/O请求过程中间各个阶段的字节偏移量，而f\_offset则只记录I/O请求执行前后字节偏移量的变化。

m\_Count表示还剩余要读、写的字节数，初始为read、write系统调用中用户指定要读、写的字节数，每当读、写完一块数据时，m\_Count减去该数据块的大小。通常，一次I/O读写请求执行到剩余要读、写的字节数m\_Count为0时才结束，但如果已经达到文件的末尾，m\_Count的值仍然不为0，本次I/O操作同样执行结束。

### 文件读、写函数—ReadI() & WriteI()

内核中实际进行文件读、写操作的函数分别是Inode::ReadI()和Inode::WriteI()，由它们具体负责经由高速缓存读取或写入文件数据。它们的流程图如下：(图11.17 & 图11.18)

 

#### 图11.17 ReadI() 图11.18 WriteI()

ReadI()函数的作用是根据User.u\_IOParam中的文件I/O参数，包括用户目标区首地址、要读取的字节数以及当前文件读写字节偏移量，通过访问该文件的Inode找到对应的数据块，经由高速缓存将进程需要的数据送入用户目标区。其中采取了预读策略，如果发现进程正在顺序读文件，则在读取文件当前的逻辑块时将下一块一并读入，如果预读命中则可以有效减少下次I/O操作的开销，提高进程的前进速度。

对于WriteI()函数，其功能是将用户指定内存区中的数据经由高速缓存写入文件，数据写入文件的位置同样取决于User.u\_IOParam中的文件I/O参数，整个工作流程与ReadI()函数相似，但仍有以下几方面需注意：(1)文件中的某些字符块可能只需要部分重写，为了保护不需要重写的部分，应该先将该字符块读至缓存，部分改写后再写回；(2) 全部已经改写或新扩充已写满的字符块立即写回相应盘块，仅部分改写的字符块则暂缓，用延迟写方式处理，这样做是考虑到进程可能在稍后会再次读写该字符块，使用延迟写方式可以让该字符块在高速缓存队列中停留较长时间不被分配做它用，从而增加了在高速缓存中直接找到该字符块的概率。