第四章 文件和文件系统的内部结构

现代**UNIX**的文件系统通常可由三大模块组成：

①本地文件系统（**UFS**）——**User File System**

②网络文件系统（**NFS**）——**Network File System**

③虚拟文件系统（**VFS**）——**Virtual File System**

本地文件系统（UFS）

是UNIX系统中的基本文件系统，它通常固定存放在本地机器的存贮设备上，任何一种结构形式的文件系统都必然会直接或间接地与某个本地文件系统相联系。

本地文件系统的构成

一个根文件系统 + 若干子文件系统所组成

根文件系统

存放本操作系统的最主要和最基本的部分

可独立启动运行

系统起动后，根文件系统就不能卸下来

子文件系统

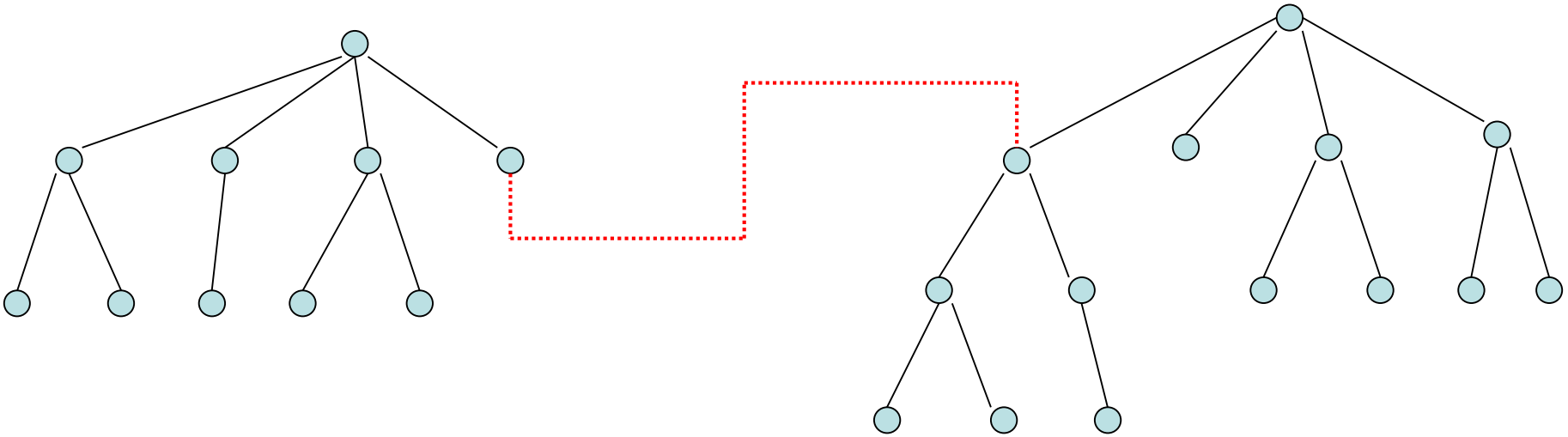
主要存放应用程序和用户文件

一般不能独立启动

系统运行过程中可随时安装和卸下

网络文件系统（NFS）

是本地机器上的文件系统和远地机器上的文件系统之间的介质，它管理和控制所有有关对远地文件的各种操作，给本地用户提供一个访问远地文件的使用方便的高层接口，避免用户直接涉及网络通讯方面的具体细节。

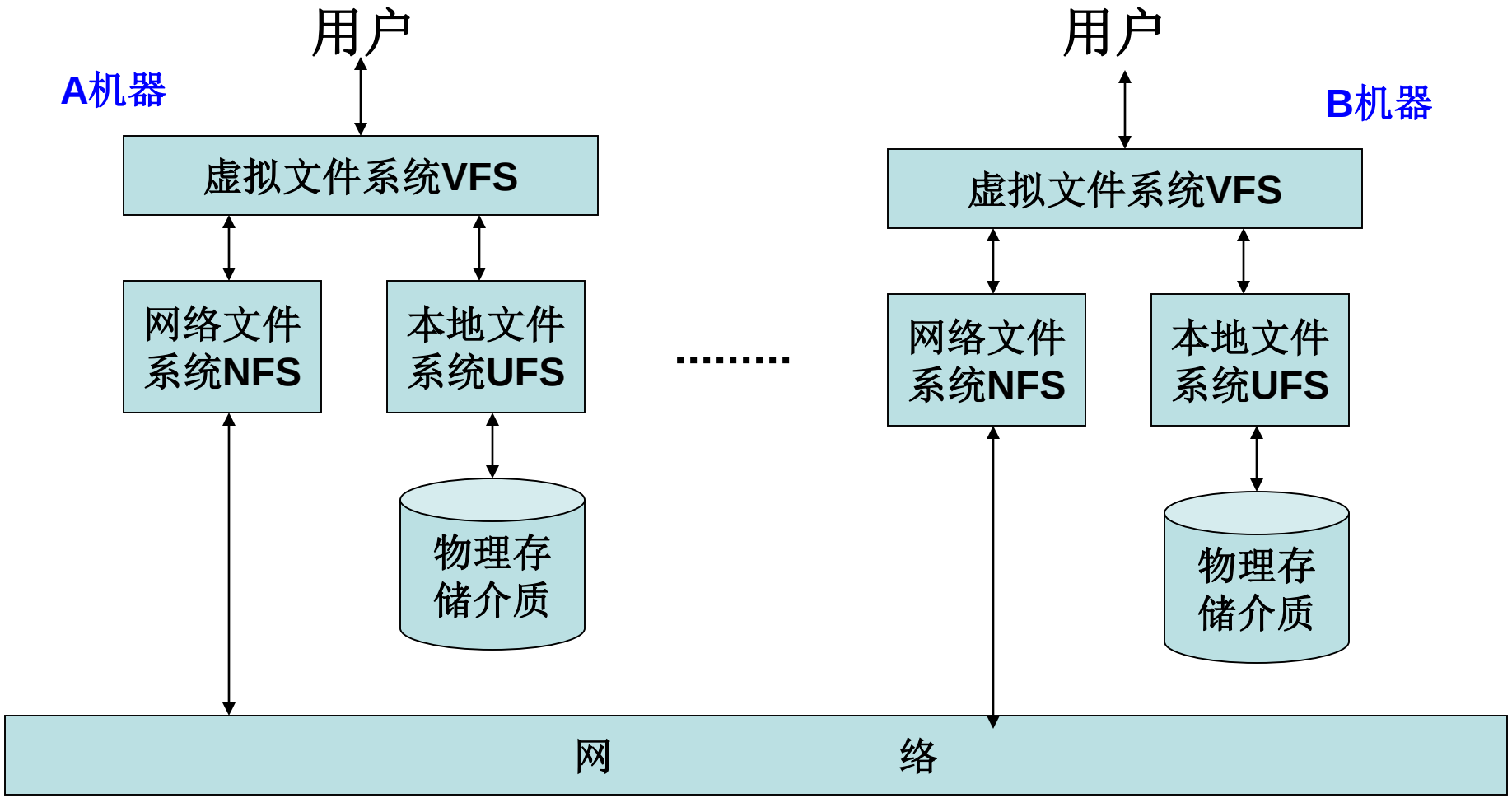


虚拟文件系统（VFS）

VFS是整个操作系统的用户界面，它给用户提供一个统一的文件系统使用接口，避免用户涉及各个子文件系统的特征部分。

用户感觉使用的是一个整体的，比本地机器上实际硬盘空间大得多的文件系统。

虚构文件系统接受来自用户的操作请求，根据该操作所访问的文件是存放在本地机器上，还是存放在远地机器上而分别把操作交给本地文件系统或网络文件系统；本地文件系统或网络文件系统（实际上再传给远地机器上的本地文件系统）进行相应的操作后，将结果返回到虚拟文件系统中再传回给用户。



基于虚拟文件系统的体系结构

4.1 文件系统结构

4.1.1 本地文件系统

1. 文件系统的存储结构

在UNIX系统中，一个物理磁盘通常被划分成一个或多个逻辑文件系统（简称文件系统或子文件系统），每个逻辑文件系统都被当作一个由逻辑设备号标识的逻辑设备。

UNIX的普通文件和目录文件就保存在这样的文件系统中。逻辑文件系统的存储结构可分为两类型：

一级存储结构型：常用于运行环境较小的文件系统中

二级存储结构型：常用于运行环境较大（特别是硬盘空间较大）的文件系统中

①、一级存储结构型

这种类型的逻辑文件系统由超级块、索引节点表块和数据区组成，（如果是根文件系统，就还包括引导块）。整个存储结构是一维的。

引导块	超级块	i节点表块	数据区

引导块： boot程序

超级块： fs结构，存放文件系统的静态参数

i 节点表块：磁盘icommon表

数据区： 各数据块

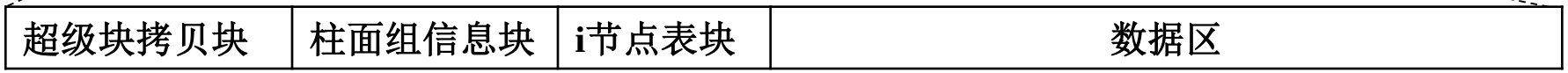
②、两级存储结构型

这种存储结构的文件系统由两级组成：第一级由超级块和若干个柱面组块（cylinder group block）所组成（如果是根文件系统则还包括引导块）。第二级（即柱面组块）又是由超级块拷贝块、柱面组信息块，i节点表块和数据区所组成。文件系统的存储结构是二维的。

目前大多数在大存储环境下运行的UNIX版本都采用这种存贮结构，其优点是能快速定位数据块。

第一级存储结构

第二级存储结构



超级块

是由fs定义的数据结构，用于存放文件系统的静态参数：

```
struct  fs {
```

内存超级块链接指针
超级块的磁盘地址
柱面组块的位移量
最近修改时间
文件系统大小
文件系统块大小
柱面组数
柱面组大小
片大小
文件系统标识数
文件系统标志区
最近访问的柱面组号
确定分配算法的参数

```
}
```

超级块拷贝块：

在每个柱面组块中存放有一个超级块拷贝块，其目的是使系统在超级块被意外破坏时，能从任何一个柱面组中进行恢复而不致使整个文件系统陷入瘫痪。

每个柱面组中的超级块拷贝块的存放位置为安全起见不一定都装在柱面组中的最前面，而是可浮动地装在该柱面组中的任何位置。

一般性的方法是：如果第n号柱面组中的超级块拷贝块开始于该柱面组中的第i磁道，则第n+1柱面组中的超级块拷贝块开始于该柱面组中的第i+1磁道。文件系统一旦建立后，它们的位置就是固定不变的。

柱面组信息块（cg块）

柱面组信息块中存放的是有关该柱面组的静态参数，它由数据结构cg来定义：

```
struct  cg  {
        内存中柱面组块的链接指针
        本柱面组块中i节点表大小
        本柱面组块中数据区大小
        最近一次所用块的位置
        最近一次所用片的位置
        最近一次所用i节点的位置
        本柱面组空闲数据块总数
        i节点位示图
        空闲块位示图
}
```

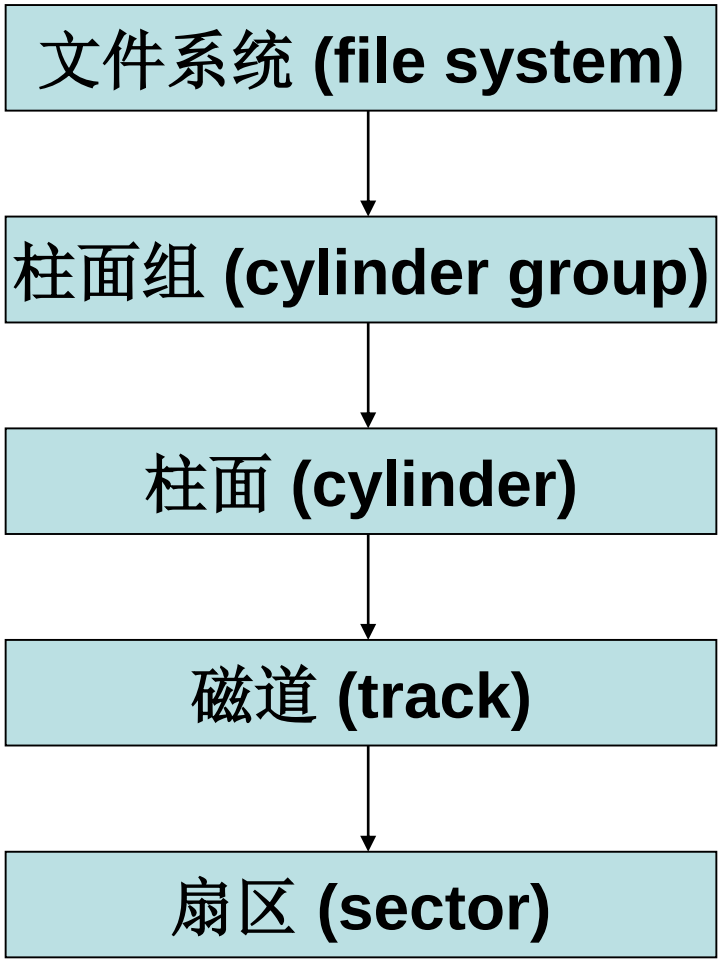
位示图：

位示图为一张表，其中的每一个二进制位（bit）的值来表示某一个资源（例如数据块或i节点）的状态，这样每检测一个字节的值就可以知道八个资源的状态；每检测一个四字节的整数的值就可以知道32个资源的状态。

系统只需要维护一张较小的表（位示图）就可以快速地检测指定资源的忙闲状态，或快速查找可用的空闲资源。

2、文件系统的数据块

在文件系统中，按存储单位来划分，由大到小可有下列层次：



DEV_BSIZE　512字节

文件系统的逻辑块大小：

DEV_BSIZE * 2^n　　即1k、2k、4k、8k、16k …

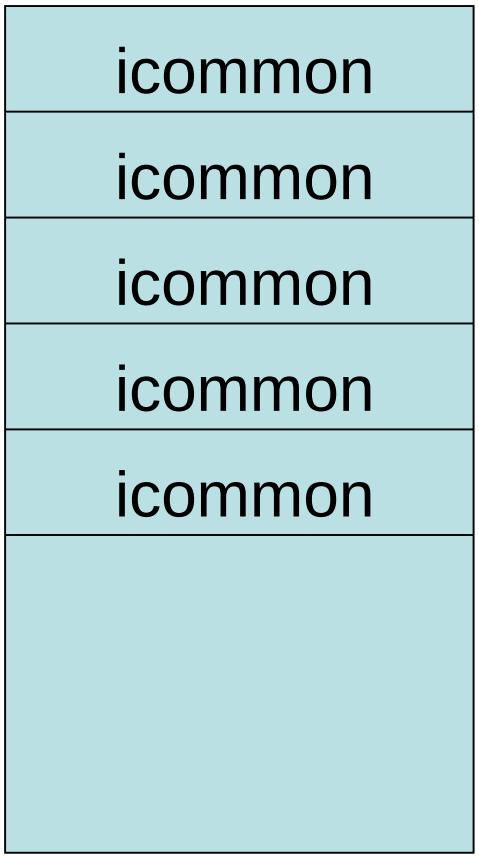
目的：提高传输速度，减少overhead

文件系统的逻辑片大小：

DEV_BSIZE * 2^n　　即1k、2k、4k、8k、16k …

目的：减少文件尾的碎片浪费。

3、i节点与磁盘i节点表



磁盘icommon表

- 文件所有者标识（UID）
- 用户组标识（GID）
- 文件类型（FIFO、DIR、CHR、BLK、REG、LNK等）
- 文件保护模式（存取许可权）mode
- 文件存取时间（atime, mtime, ctime）
- 链接数目 link
- 文件大小 size
- 文件数据块索引表 index table

4. 文件的存贮结构

UNIX的普通文件的逻辑结构是无格式的有序字节流，而它们的物理存贮结构是以索引方式来组织的。

每个文件都是由一个索引节点i节点来表示的，每个i节点由其i节点号来标识。

i节点通常以静态的形式存放在磁盘的i节点表中。每个磁盘i节点表项是由数据结构icommon定义的，描述对应文件的静态参数。

icommon 与 inode 的关系

进程要读写一个文件时，先在内存的活动i节点表（即inode表）中申请一个空闲的活动i节点，并把磁盘上i节点（icommon）中的各项参数读入其中，当核心操作完成后，如果必要，就把在内存中的活动i节点写回到磁盘上去。

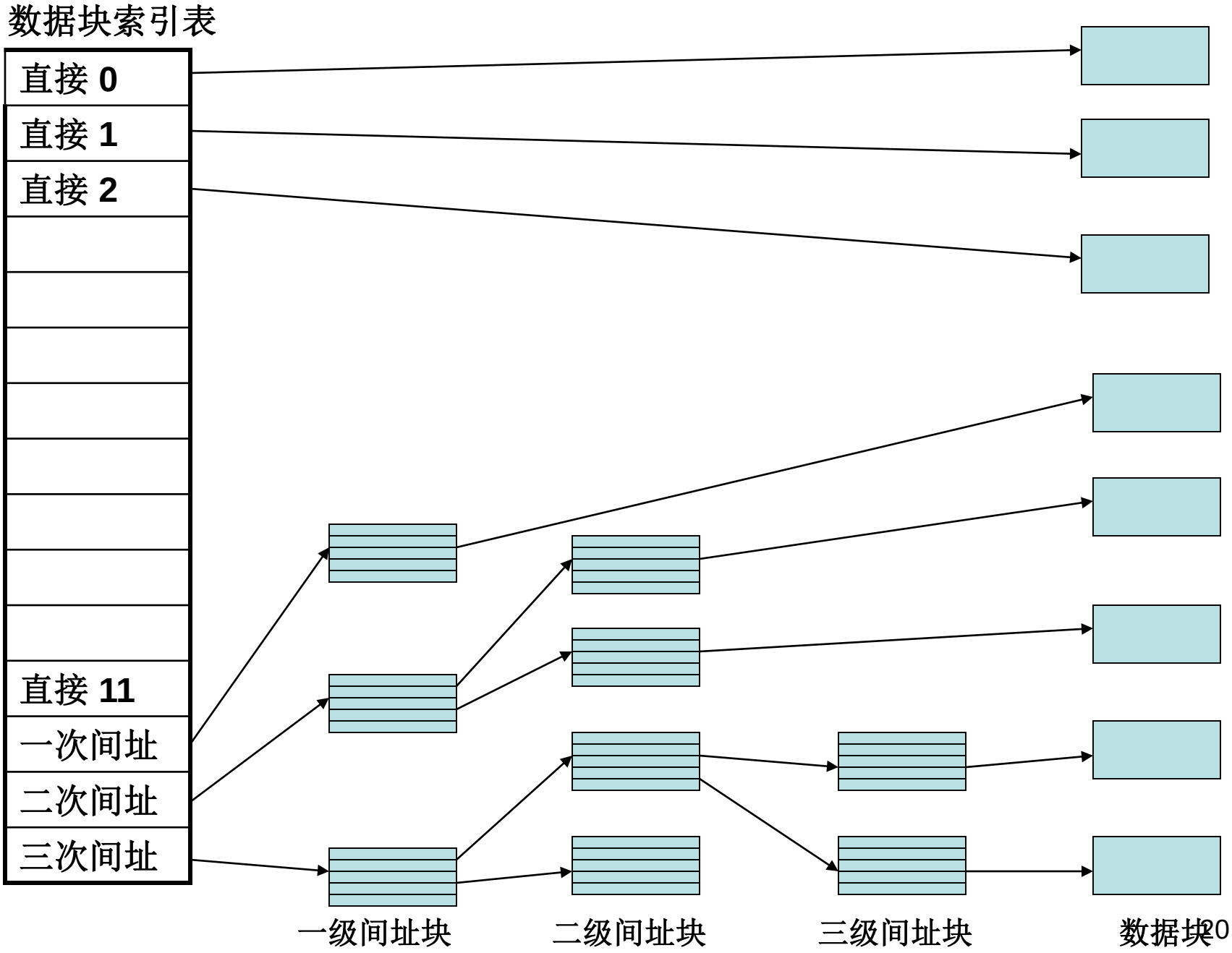
内存活动i节点由数据结构inode来定义，它除了包含磁盘上对应的icommon中的各项参数外，还包含有其它的参数，如该活动i节点的状态、文件所在的逻辑设备号、i节点号、活动i节点链接指针，最近使用的i节点在目录中的位置等动态信息。

```
struct  inode  {
        活动i节点链接指针
        状态标志
        设备号
        i节点号
        最近访问的i节点在目录中的位置
        空闲i节点链接指针
        struct  incommon {
                文件模式和类型（FIFO、DIR、CHR、BLK、REG、LNK等）
                文件链接数
                文件所有者标识数（UID）
                文件所属用户组标识数（GID）
                文件大小
                文件最近存取时间（atime, mtime, ctime）
                数据块索引表
                其它信息
        }
}
```

5、数据块索引表

数据块索引表用于检索本文件占用的数据块。它包含12项直接索引表目和3项间接索引表目。根据要读写的数据在文件中的位置可计算出该数据所在的逻辑块号，查索引表就可找到逻辑块所在的文件系统块号。

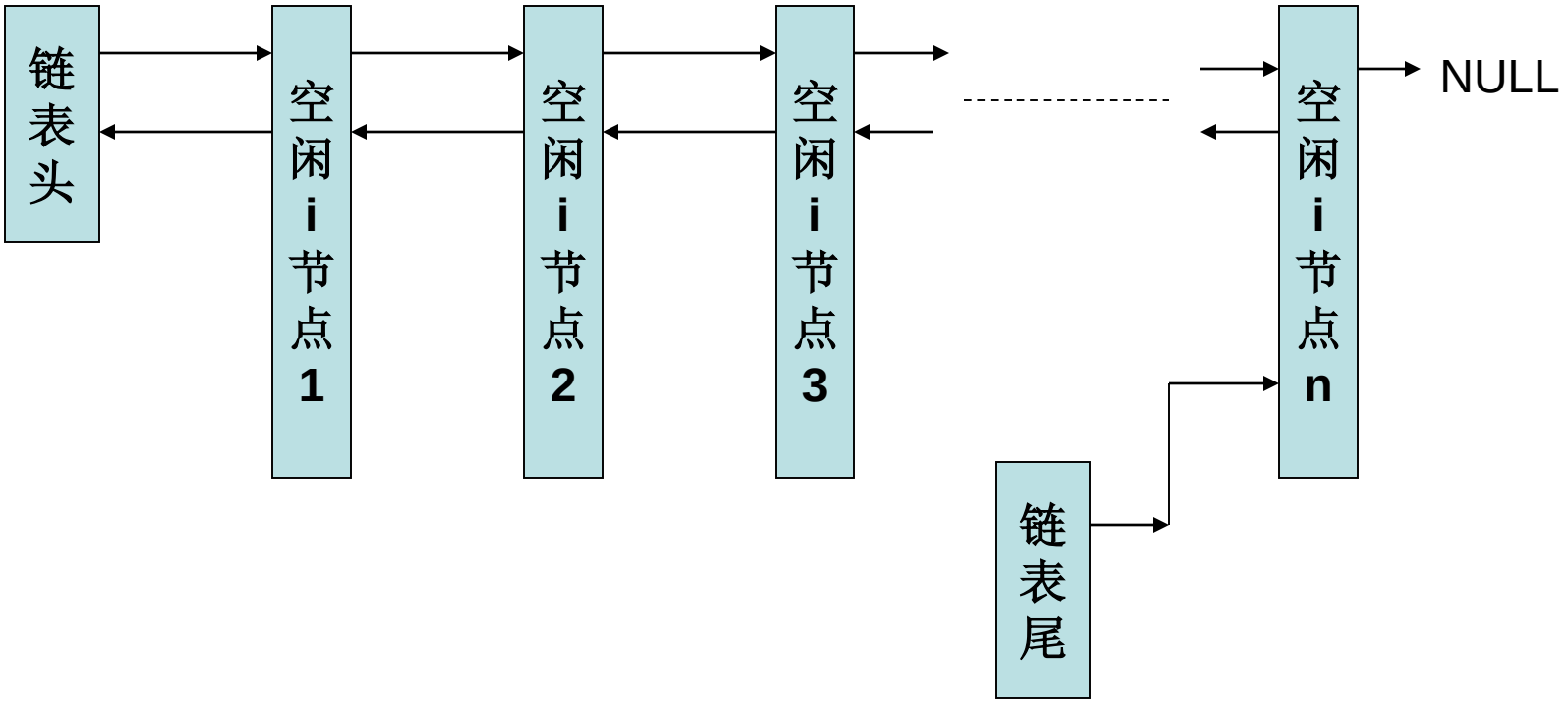
系统根据计算出来的逻辑块号判断是否包含在直接索引表中，如果是，则取出直接索引表中的文件系统块号；如不是，则看是否包含在一次间接索引块中，否则再寻找二次和三次间接索引块。最长要存取三次间址索引块才能找到相应数据的文件系统块号（要取出数据则要读4次磁盘）。


数据块索引表
直接 0
直接 1
直接 2
直接 11
一次间址
二次间址
三次间址
一级间址块
二级间址块
三级间址块
数据块

6、inode表的结构

在内存中，活动i节点表类似于数据缓冲区高速缓冲中的缓冲池结构。活动i节点表中的每一项就是一个活动i节点缓冲区，用来存放一个被打开文件的inode。（以下把活动i节点缓冲区简称为“活动i节点”）。

空闲的活动i节点相互链接在一起构成“空闲活动i节点链表”，这是一个双向（非循环）链表，分别由链头指针和链尾指针指向空闲活动i节点链表的开始和结束。如下图所示：



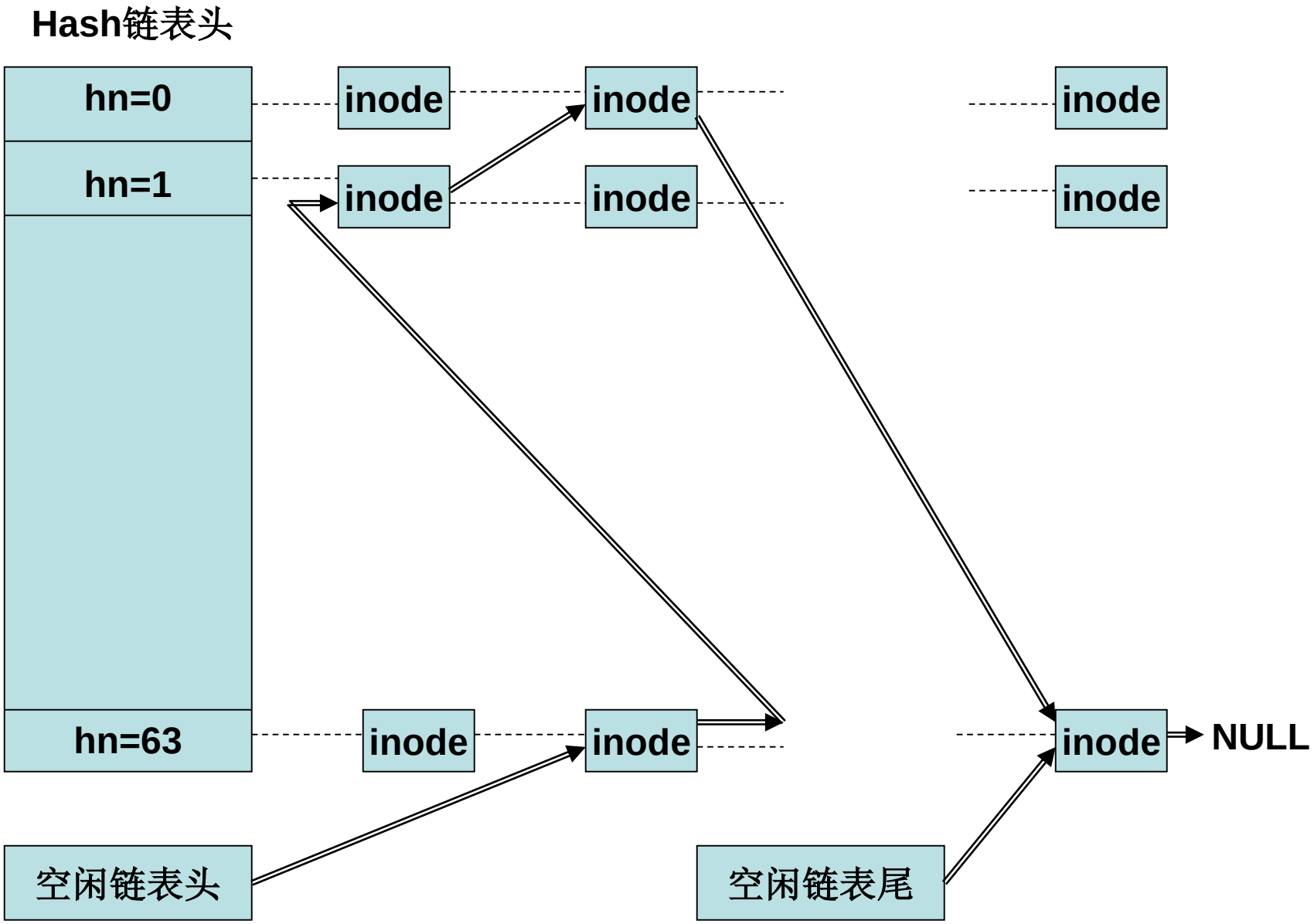
空闲活动i节点链表为双向（非循环）链表，分别由链表头指针和链表尾指针指向空闲链表的首尾。

活动i节点hash链表

当某个文件（即某个磁盘i节点）被打开时，根据该i节点所对应的设备号和i节点号计算其hash值：

hn=(devno+inumber)%64

可得到0~63共64个hash值。具有相同hash值的活动i节点链接在同一个hash链表中，这样内存中就有64个hash链表，每个hash链表都是由hash链头开始的双向链表（与数据缓冲区链表不同的是此处的空闲和非空闲链表都是非循环的）。内存活动i节点表就是由这64个hash链表组成。如下图所示：



活动 inode 表的结构

7. 目录和目录项

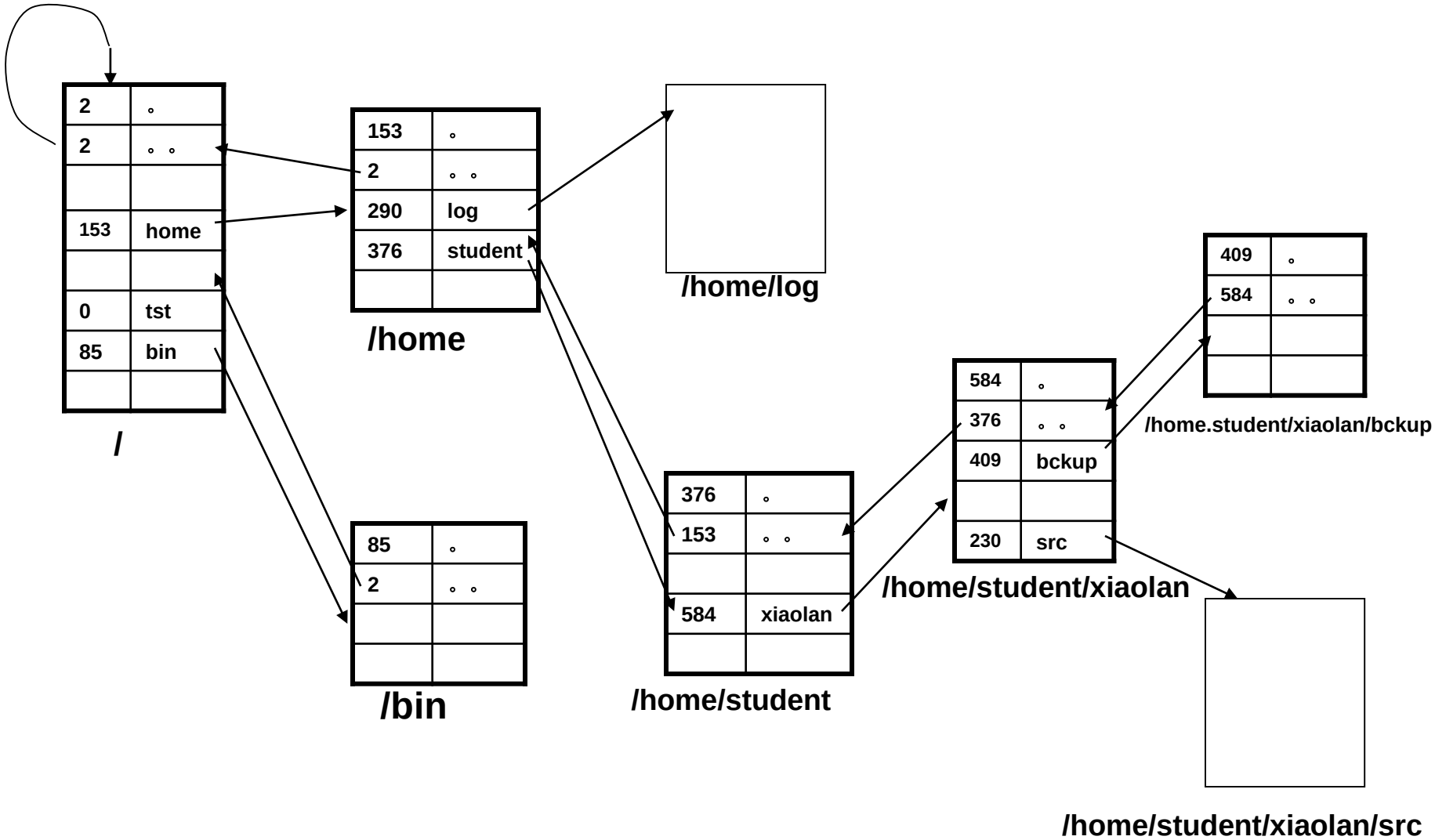
在 UNIX文件系统中，目录的组织形式采用的是树形结构，一个逻辑文件系统就是一棵目录树。

目录也被当作文件进行处理，一个目录文件的结构为表状结构，其中通常包含有若干表项，称为目录项，这些目录项既可以是普通文件的入口，也可以是子目录的入口。

每一个目录项中通常包含两部分内容：

文件的 i 节点号

文件名



目录 **/home/student/xiaolan** 的路径和目录结构

每个目录项由数据结构direct来定义：

```
#define  MAXNAMLEN  14
struct   direct   {
short d_ino;                          /*目录项i节点号*/
char d_name [MAXNAMELEN];    /*目录项名字字符串*/
}
```

每个目录项的长度通常是确定的，为16个字节，其中前两个字节存放文件的i节点号d_ino，后面14个字节存放文件名d_name。

这种定长目录项在算法实现方面比较简单，在使用灵活方面都有所不便，并且可能因许多目录项名字长度不足14字符面有空间浪费。

在UNIX的每个文件系统中，有三个 i 节点号是有固定用途的：

0号i节点：

表示空目录项，当某个目录项被删除时，该目录项的 i 节点号被置为0。

1号i节点：

表示坏块文件，所有的磁盘坏块都划归到该节点上；

2号i节点：

固定表示该逻辑文件系统的根（root）目录；

3号i节点：

表示该文件系统中的 lost+found 目录。

8、变长目录项的目录结构

```
#define   MAXNAMLEN   255
struct   direct   {
   long d_ino;                          /*  目录项i节点号  */
   short d_reclen;                      /*  目录项入口长度（占用长度）  */
   short d_namelen;                     /*  目录项名字长度  */
   char d_name [MAXNAMLEN+1]    /* 名字字符串，+1为串结束符 \0  */
}
```

由于目录中各目录项的长度是变化的，因此必须在目录项中标明本目录项的长度。前一个目录项释放时，把该目录项的空间全部合并到前一个目录项中，形成前面一个目录项占用空间大于实际使用的空间。

在增加新目录时，先查看目录中各目录项是否占用多余空间，如有，则进行压缩，把释放的空间分配给新目录项。

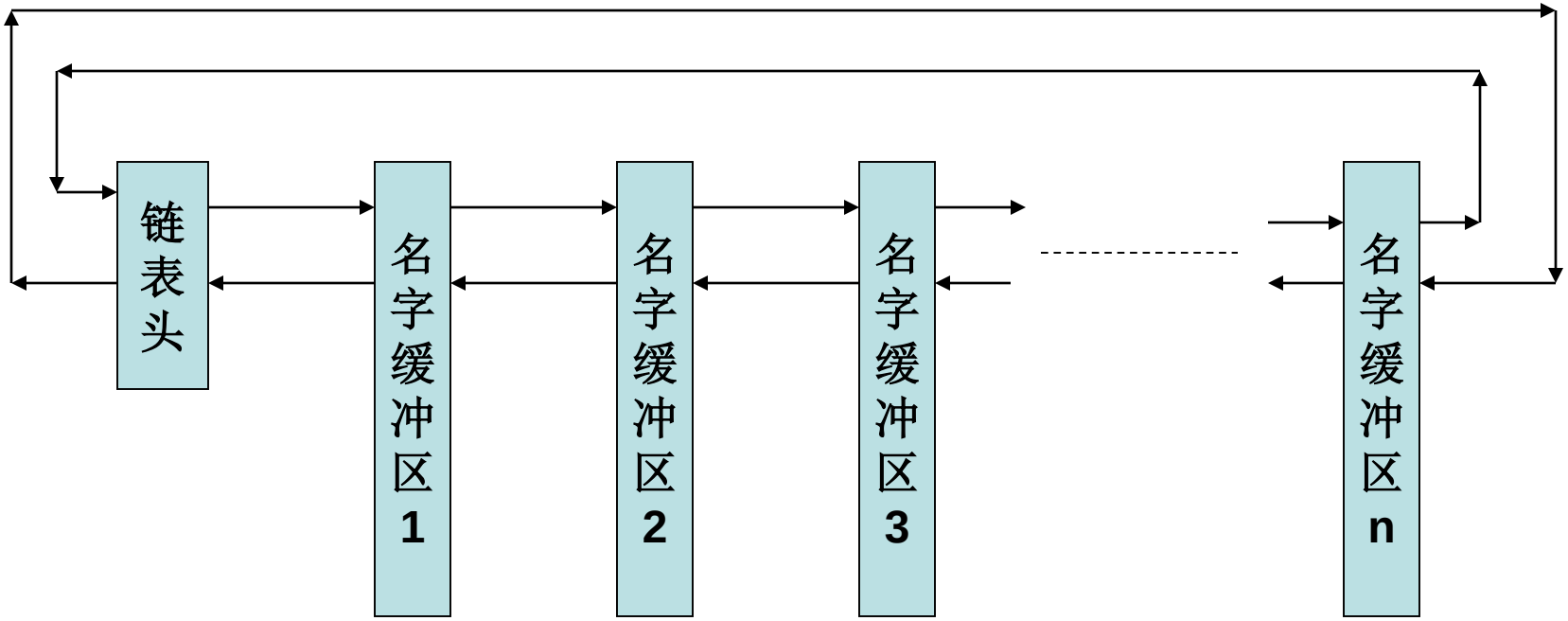
变长目录结构增加了算法复杂性和工作量，通常用在硬件性能较高的大型系统中。

9、文件名和文件名缓冲区

核心在内存中建立了一个高速的名字缓冲区，用来存放最近使用过的文件名，核心认为“最近使用过的文件名马上还要使用的可能性最大”。

名字缓冲区是由**ncache**定义的数据结构，只包含文件名和索引节点指针等重要信息：

```
struct    ncache   {
      hash链表指针
      LRU链表指针
      文件i节点指针              /* 这里只需要节点指针，因内存
                                    中已有活动i节点表 */
      文件父目录节点指针
      文件名
      确认信息
}
```



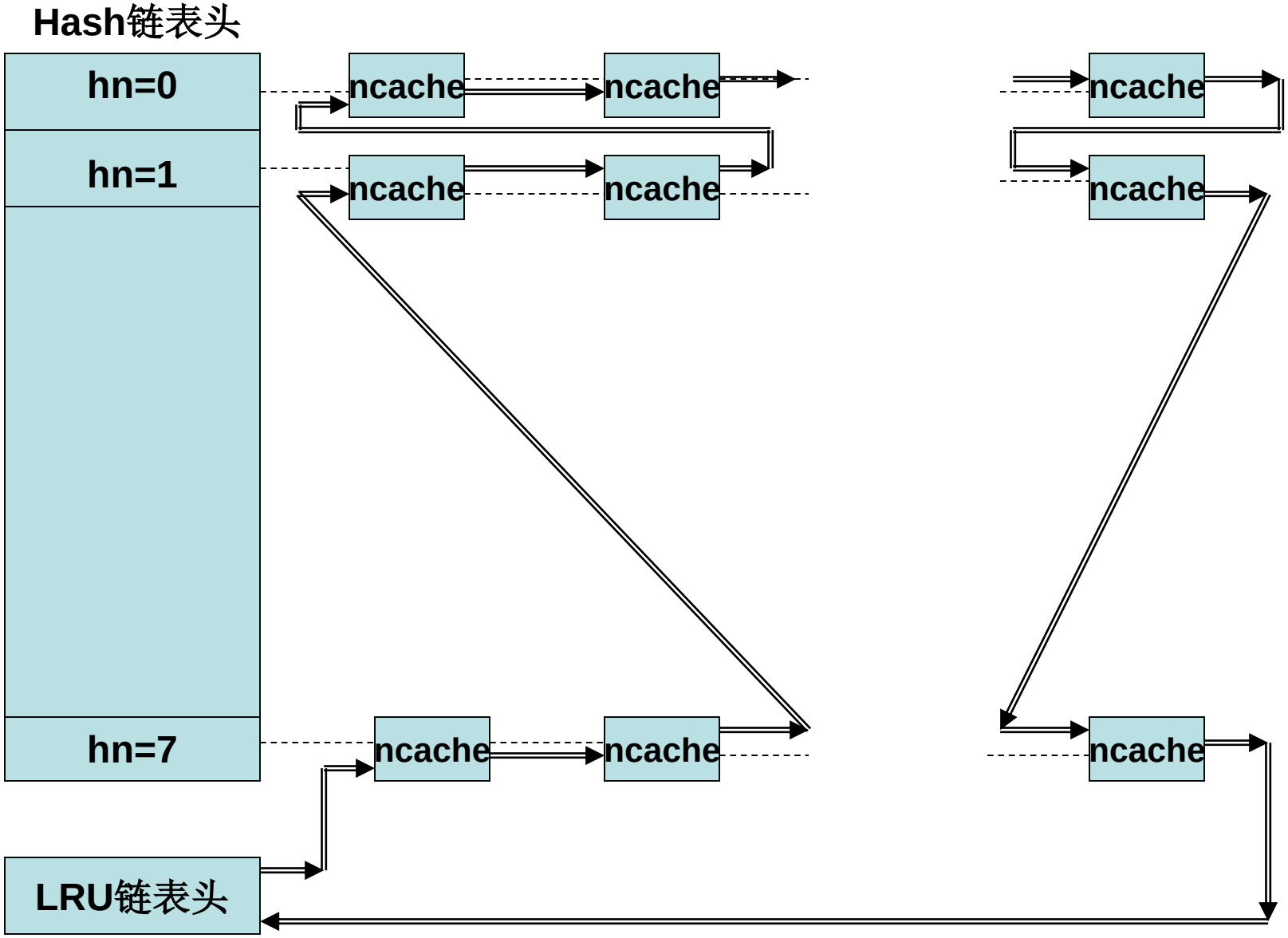
名字缓冲区（**ncache**）链表为定长双向循环链表（**LRU**链）

为提高查找文件名的速度，还根据每个**ncache**的**hash**值，将其挂接到一个**hash**链表中。**ncache**的**hash**值用下列公式计算：

hn=7&(*namep+*(namep-1+slen)+slen+(int)VP)

namep：为指向名字字符串的指针
slen：　为名字长度
VP：　为父目录节点指针

计算出相应的**hash**值（**0~7**）。共建立**8**个**hash**链，每个**hash**链是一个以链表头开始的双向循环链表，每个**ncache**任何时候都同时既在**hash**链上，又在**LRU**链上。



名字缓冲区(ncache)池结构

10、资源保护

UNIX文件系统中的资源保护系统主要是对系统中的两类资源进行保护：

① 静态硬资源

包括存贮空间和索引节点，对这类资源的保护主要是通过quota系统提供的限量机制实现的。

② 动态资源

主要是指临界区资源或独享资源，对这类资源的保护主要是通过上锁机制来实现的。

1）quota系统

quota系统的主要功能就是检测和限制用户对文件系统资源的使用。

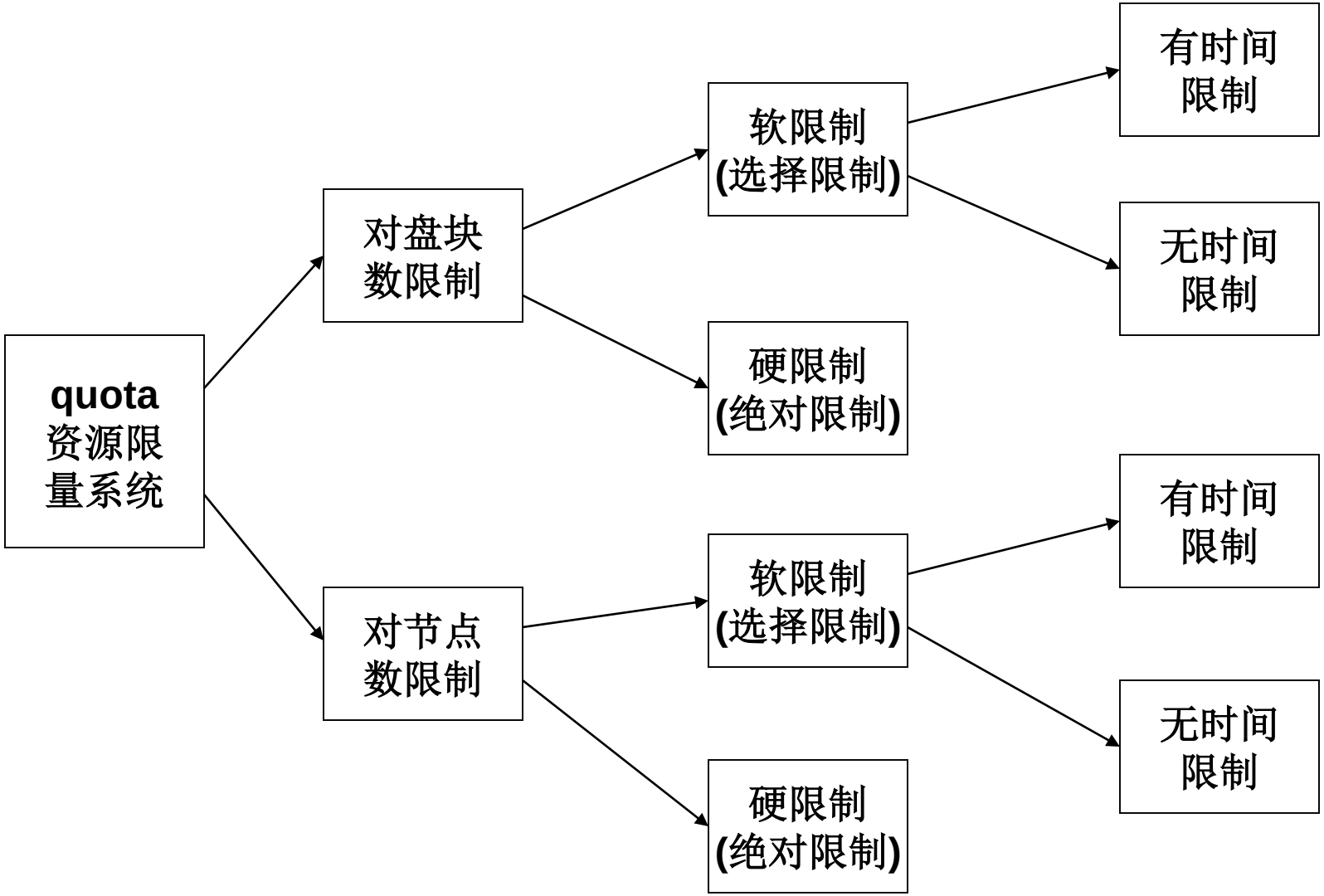
在每个逻辑文件系统（包括根文件系统和各子文件系统）中都可以建立一个磁盘quota文件来限制用户对本文件系统资源的使用。

每个quota文件是由多个dqblk数据结构所组成的表格，每个dqblk都针对一个用户，用于存放该文件系统对该用户的限制参数。

如果一个用户的使用空间包含多个文件系统，则需在每个文件系统的硬盘quota文件中给该用户分配一个dqblk项。

```
struct   dqblk   {
    盘块数硬限制
    盘块数软限制
    已用盘块数
    节点数硬限制
    节点数软限制
    已用节点数
    盘块数时间限制
    节点数时间限制
}
```

系统在运行时，在内存中建立了一个活动**dqblk**链表（类似于活动**i**节点表），每个活动**dqblk**由其**hash**值而分别处在**64**链中的某个链表上，如果某个活动**dqblk**是空闲的，它同时又处在空闲活动**dqblk**链表中（与活动**i**节点表的结构完全一样）。



quota限量系统构成

2）上锁机构

为保持数据的一致性，能正常使用临界区资源，**UNIX**文件系统提供了一种对数据（文件或记录）的上锁机制，即劝告锁（**advisory lock**），用户既可给整个文件上锁，也可给部分记录上锁。

劝告锁包含两种类型：

共享锁（读操作锁）　　互斥锁（写操作锁）

在同一个文件或同一个记录上同时只能有一个互斥锁（或写操作锁），但可以同时有多个共享锁（读操作锁）。共享锁的级别低于互斥锁，互斥锁可以覆盖共享锁。

对一个文件上锁只是一种“自觉性”的保护措施——“劝告锁”，进程在读写该文件之前都必须先检查该文件是否已上锁，然后再进行相应的锁操作后才能对该文件进行读写。否则将使该文件上原有的所有劝告锁失效。

4.1.2 虚拟文件系统

1. 文件结点

在本地文件系统中，每个文件都由一个索引节点inode来代表，对一个文件名的操作都被转换成对该文件的inode进行相应的操作。

类似地，在虚拟文件系统中，每个文件都是由节点vnode来代表，对文件名的操作都被转换成对该文件的vnode进行相应的操作。v节点中包含了它所代表的活动文件的有关信息，如文件状态、上锁情况、引用计数、文件类型等，它由数据结构vnode来定义：

```
struct   vnode   {
    节点状态标志        /*是否为所在OS的根，是否上锁，
                        有否等待上锁进程，是否已被修改*/
    访问计数值          /*多少进程在共享该节点*/
    上锁情况            /*上了多少共享锁，互斥锁**/
    该v节点操作指针     /*指向操作函数组*/
    节点类型            /*VREG， VDIR， VBLK，VCHR，
                          VLNK， VFIFO*/
    设备号              /*所在设备*/
}
```

VFS接受用户的操作请求，把用户对文件名的操作转换成对相应的v节点操作，如果该文件实体在本地机器上，则虚拟文件系统把v节点转换成本地文件系统中对应的i节点，并把用户操作转给本地文件系统去完成；

如果文件实体在远地机器上，虚拟文件系统把操作转给网络文件系统，由网络文件系统起动网络驱动程序，执行对网络上其它机器上某个文件的操作。事实上对远地文件的操作最终落实到远地机器的本地文件系统（对它的i节点的）操作上。

v节点与i节点有许多相似之处，都是对文件的属性进行描述，但又有较大的差异。

两者的关系是：

① i节点是v节点的基础。任何v节点都直接或间接地与某个i节点相联系，而不论该i节点代表的文件实体存放在本地机器上还是存放在远地机器上。

② v节点是i节点的高层节点。它是活动i节点在虚拟文件系统中的代表。

两者的区别：

① v节点是一个动态的概念。在系统中只有当前活动的文件（即正被使用的文件）才有对应的v节点；而磁盘上某个当前未被使用的文件仍然有磁盘i节点与之对应。

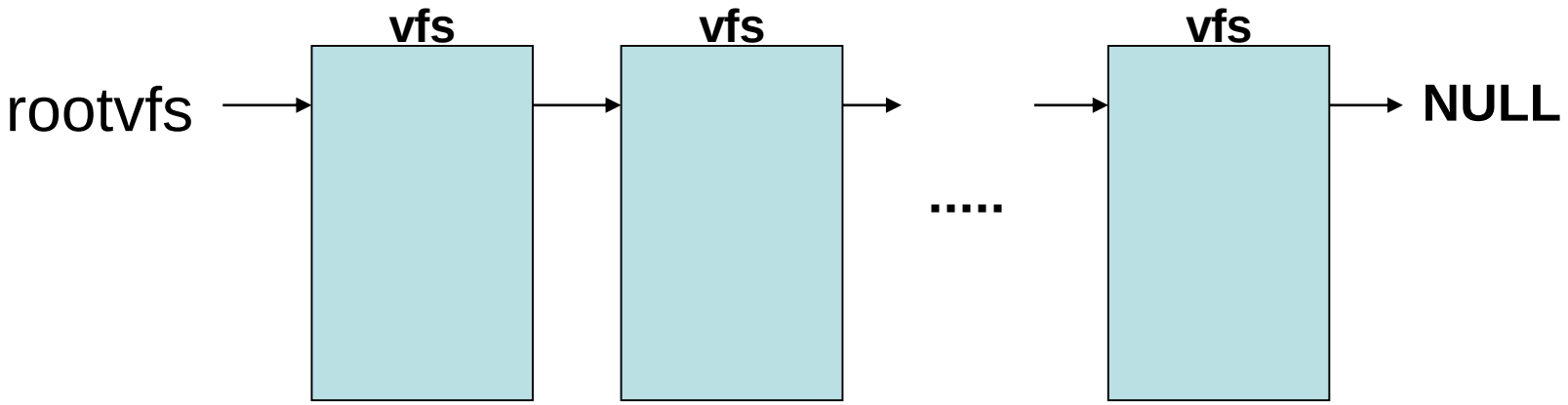
② v节点在磁盘上没有对应的数据区，当某个v节点所代表的文件使用完毕被关闭时，该v节点也随之完全消失；而不像i节点被释放时，要把被修改过的信息拷贝到磁盘上。

2．子文件系统

一个虚拟文件系统通常可由多个子文件系统组成，这些子文件系统既可能是本地的，也可能是远地的。当这些子文件系统被安装到虚拟文件系统中时，为了方便地识别和找到它们，给它们各建立了一个控制块 **vfs** 作为代表，（类似于文件在虚拟文件系统中用**vnode**作代表）。**vfs**中包含的是该子文件系统的一些基本信息，它由数据结构**vfs**定义。

```
struct    vfs    {
      该子文件系统的操作指针
      安装点的v节点
      状态标志                    /* 只读、上锁、有等待上锁进程等 */
      基本文件系统块大小
      文件系统标识数              /* 指明是本地还是远地的文件系统 */
      安装表项指针
}
```

各个**vfs**链接成一个单向链表：**VFS链**



子文件系统被安装时，在系统安装表中给该子文件系统建立了一个安装表项，使得每个子文件系统的 vfs 都与一个安装表项一一对应，每个安装表项由数据结构mount定义：

```
struct  mount  {
    该文件系统的vfs指针
    设备号
    超级块缓冲区指针
    各项quota值
    该文件系统根节点指针
    安装点节点指针
}
```

要对某个子文件系统进行操作，能够方便地由vfs链表找到该子文件系统的超级块:

由vfs→mount→buffer→超级块（fs）

3、虚拟文件系统的实现

通常用户与虚拟文件系统之间的交互界面是32个基本的系统调用，其中26个系统调用是实现对虚拟文件系统中的各个文件进行建立，删除、打开、关闭、查询、链接、修改、换名等操作。

用户通过文件名对文件的操作在虚拟文件系统中都被转换为相应的vnode的操作。在虚拟文件系统中的任何一个文件，无论它是属于本地文件系统还是属于远地文件系统，其vnode都具有完全相同的形式。

但是如果两个文件分别存放在不同的机器上，则在某一台机器上即使是对它们进行同一类型的操作（如读文件），也显然会是执行两个完全不同的程序（一个是起动本地磁盘，而另一个则是设置网络准备进行通讯）。

在虚拟文件系统中，给每个**vnode**都设置了一个操作指针，指向由**26**个功能函数（完成某一特定任务的函数）指针所组成的操作函数组，这**26**个指针分别指向对**vnode**进行各种基本操作的函数，如建立、删除、打开、关闭、查询、修改、链接、读写等。操作函数组由**vnodeops**定义：

```
struct    vnodeops    {

int  (*vn_open)();          打开文件
int  (*vn_close)();         关闭文件
int  (*vn_rdwr)();          读写文件
int  (*vn_ioctl)();         输入输出控制
int  (*vn_select)();        同步操作
int  (*vn_getattr)();       读取属性参数
int  (*vn_setattr)();       设置属性参数
int  (*vn_access)();        检查访问权限
int  (*vn_lookup)();        根据文件名寻找V节点
int  (*vn_create)();        建立文件
int  (*vn_remove)();        删除文件
int  (*vn_link)();          链接文件
int  (*vn_rename)();        文件更名
```

```
int  (*vn_mkdir)();          创建目录
int  (*vn_rmdir)();          删除目录
int  (*vn_readdir)();        读取目录
int  (*vn_symlink)();        符号链接
int  (*vn_readlink)();       读取符号链接文件名
int  (*vn_fsync)();          内存信息复制到磁盘
int  (*vn_inactive)();       释放V节点对应的i节点
int  (*vn_bmap)();           逻辑块到物理块的映射
int  (*vn_strategy)();       启动设备
int  (*vn_bread)();          读入数据块
int  (*vn_brelse)();         释放数据缓冲区
int  (*vn_lockctl)();        上锁控制
int  (*vn_fid)();            建立文件标识结构
}
```

以上在vnodeops定义的操作，是UNIX中对文件的最基本操作，对文件的其它任何操作都是通过对这些基本操作的组合来实现的。

如果根据虚拟文件系统中文件种类的不同，而分别设置这26个功能函数指针，使这些指针指向完成特定功能的函数，从而形成不同类别的操作函数组，就可适合对不同种类文件的打开、读写、链接、更名等基本操作。

这就是虚拟文件系统的基本操作模式。

在虚拟文件系统中定义了两个类型为**vnodeops**的操作函数组**ufs_vnodeops**和**nfs_vnodeops**。

ufs_vnodeops：

其中的**26**个功能函数指针按**vnodeops**中的功能次序被初始化为指向本地文件系统中，执行对文件进行建立、删除、打开、关闭、修改、链接和读写等操作的函数名；

nfs-vnodeops：

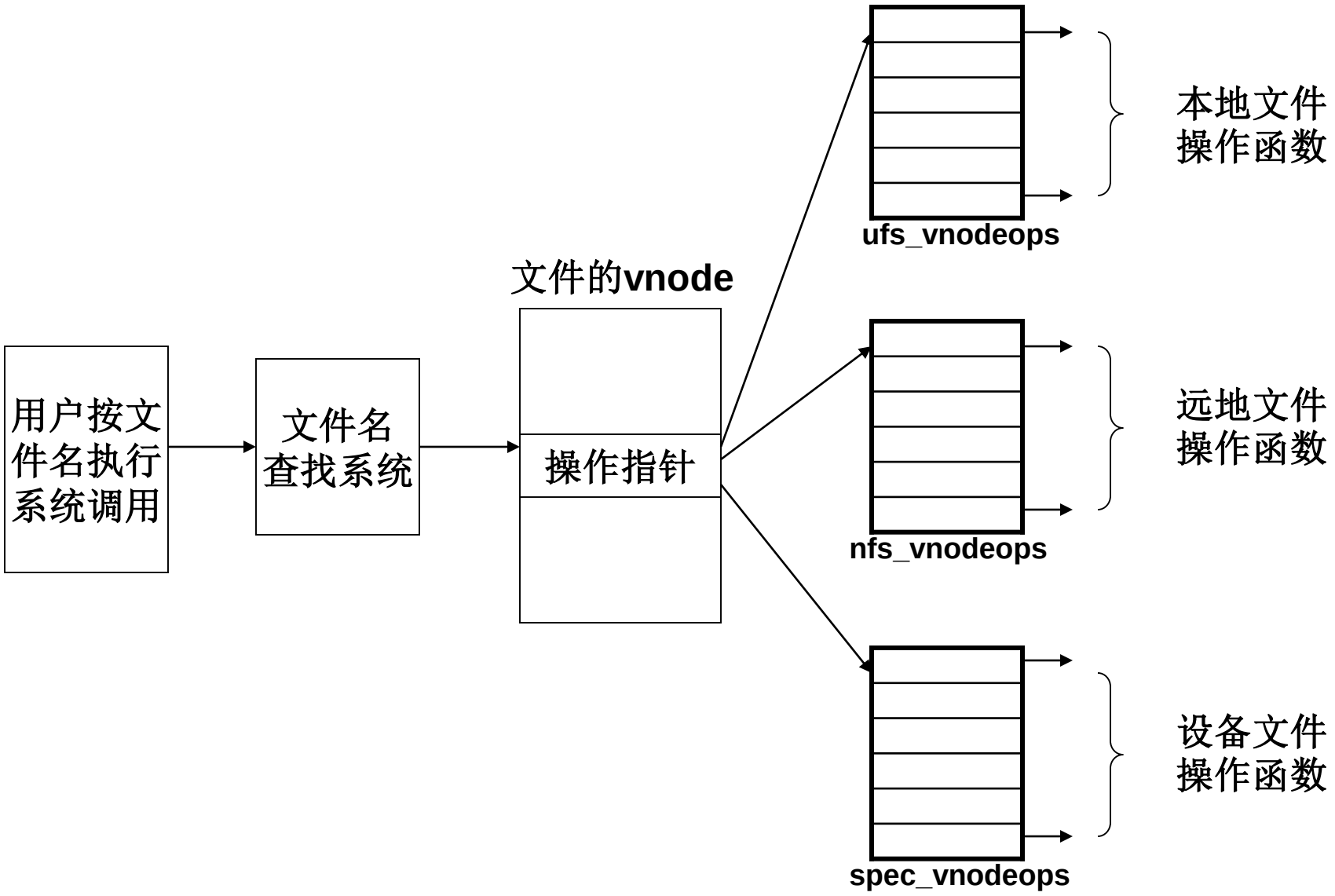
其中的**26**个功能函数指针也按**vnodeops**中的功能次序被初始化为指向网络文件系统中，执行对远地文件进行建立、删除、打开、关闭、修改、链接和读写等操作的函数名。

在vnode被建立时，根据它所代表的文件是本地的还是远地的，而将vnode中的操作指针分别指向ufs_vnodeops或nfs_vnodeops。

当用户对文件名的操作被转换为对相应vnode的操作后，再将该操作转换为调用vnode中操作指针所指向的操作函数指针组中相应功能的函数，即可完成用户指定的适合于该vnode的操作。

由于有这种执行流程的自动转换功能，因此无论对代表本地文件还是远地文件的vnode，都采用同一操作过程，从而实现整个文件系统对用户的透明性，这使用户可以完全不必知道要操作文件的具体存放地址和对不同类型文件的操作差异。

这就是虚拟文件系统基本原理。

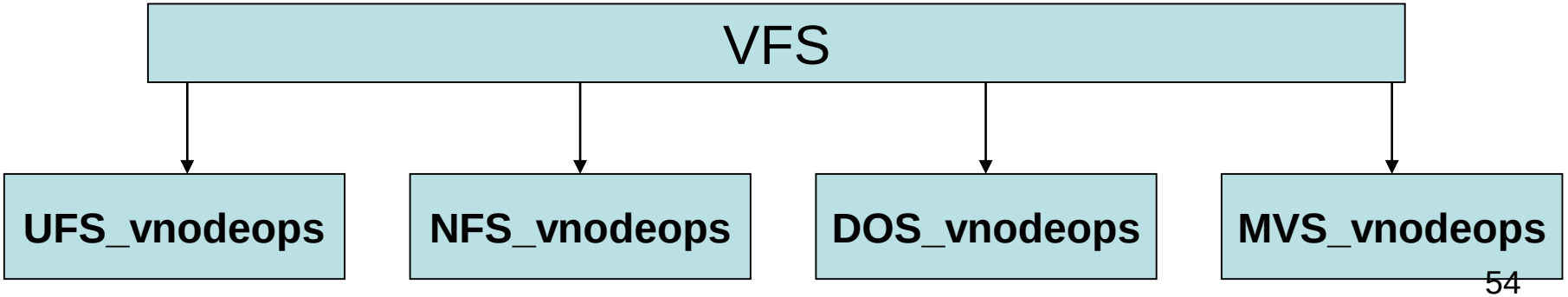


虚拟文件系统中对文件的操作流程

在虚拟文件系统中，除了**ufs_vnodeops**和**nfs_vnodeops**以外，还初始化了另外三个**vnode**的操作函数组，即**bdev_vnodeops**、**fifo_vnodeops**和**spec_vnodeops**，这三个操作函数组是分别为块设备类型、**fifo**类型和特殊设备类型文件的**vnode**准备的基本操作函数。

核心对这些特殊文件的**vnode**进行操作时，也采用与普通文件的**vnode**相同的操作过程，从而增加了整个文件系统的模块化和规范化。

UNIX的开放性也由此突出地体现出来。要增加什么样的功能或文件类型（如**DOS**）只需增加一个相应的操作函数组（如**DOS-vnodeops**）和相应的操作函数即可。



与虚拟文件系统中文件的表示方式类似，每一个子文件系统被安装时，在内存中都由一个**vfs结构**来代表，核心对各子文件系统的操作都是首先通过对它们所对应的**vfs**结构进行操作来完成的。

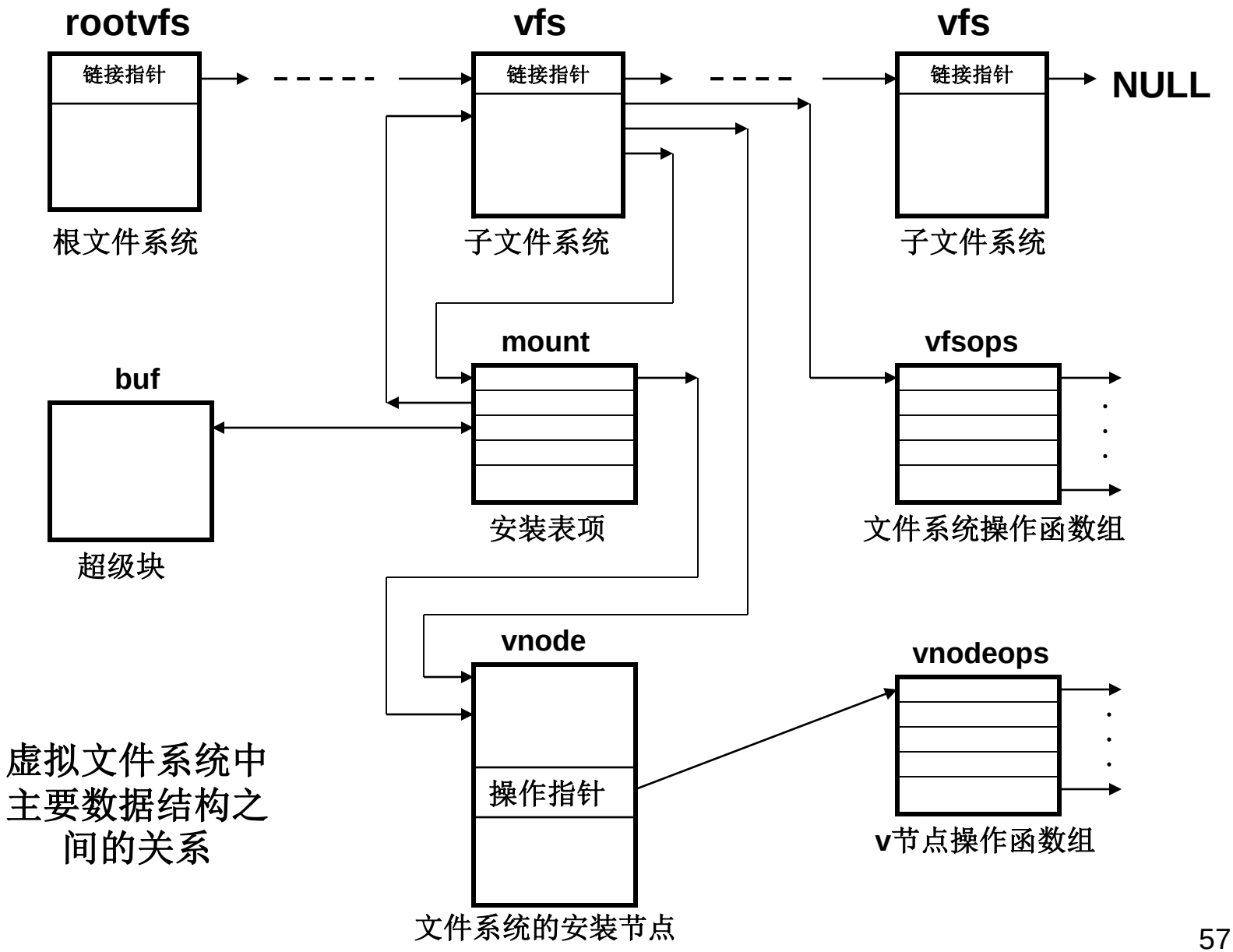
由于这些子文件系统既有本地的，也有远地的，对它们的具体操作必然不相同。

采用与**vnode**类似方法，在每个**vfs**结构中也设置一个操作指针，指向一个由**vfsops**定义的对文件系统进行操作的函数指针组，**vfsops**中共有六个功能函数指针，指向对文件子系统的安装、卸下、读取文件系统设计信息和刷新超级块缓冲区等操作的函数名。

vfsops定义如下：

```
struct    vfsops    {
    int  (*vfs_mount)();        安装文件系统
    int  (*vfs_umount)();       卸下文件系统
    int  (*vfs_root)();         找到根目录的vnode
    int  (*vfs_statfs)();       读取文件系统统计信息
    int  (*vfs_sync)();         将内存缓冲区内容复制到外存
    int  (*vfs_vget)();         由文件标识数找到相应的vnode
}
```

在系统中初始化了两个这样的操作函数组，即**ufs_vfsops**和**nfs_vfsops**，分别适合对本地子文件系统和远地子文件系统的操作。在某个子文件系统被安装时，根据它是本地的还是远地的，而将它的**vfs**结构中的操作指针指向**ufs_vfsops**或**nfs_vfsops**。这样在对某个子文件系统进行操作时，不论它是本地的还是远地的都执行同一操作过程，即调用操作函数组中相应指针所指的函数即可完成希望的操作。



虚拟文件系统中主要数据结构之间的关系

虚拟文件系统小结

在虚拟文件系统中，无论是代表各种类型文件的**vnode**之间，还是代表不同地点的各子文件系统的**vfs**之间，操作过程都是统一的，只是在进行虚拟文件系统之下的功能操作时，才根据各自的操作指针自动转到不同的函数中去。

这种方式使得整个文件系统结构统一，模块性强，增加功能非常方便；对用户来说，整个文件系统的透明性好，使用简便，避免了用户在对不同类型的文件或不同地点的文件系统进行操作时，分别来设置参数和安排操作过程。

4.2 文件系统算法

文件系统的低层算法是建立在缓冲区算法的基础之上的，主要包括以下几项：

分配活动索引节点
释放活动索引节点
逻辑位移量到文件系统块的映射
把路径名转换为索引节点
给新文件分配磁盘i节点
释放磁盘i节点

1、分配活动索引节点 iget

核心根据文件系统号和i 节点号，在内存活动i 节点表（inode表）中申请一个空闲的活动i节点，再把磁盘上对应的i节点（icommon）拷贝到这个空闲的活动i节点中。

如果要找的i节点已经在内存的活动i节点表中，则增加该i节点的引用计数，并将其上锁返回；

如果要找的i节点不在内存的活动i节点表中，则在空闲活动i节点表中申请一个活动i 节点，将磁盘icommon读入其中，重新计算其hash值后放到新的hash链表中。

```
算法 iget
输入：文件系统索引节点号
输出：上锁状态的索引节点
{
        while（未完成）
        {
                if（是索引节点高速缓冲中的索引节点）
                {
                        if（索引节点为上锁状态）
                        {
                                sleep（索引节点变成开锁状态事件）；
                                continue；
                        }
                        /* 对安装节点进行特殊处理 */
                        if （是空闲链表上的索引节点）
                                从空闲链表上移去该节点；
                        索引节点引用计数值加一；
                        return（索引节点）；
                }
                /* 接下页 */
```

```
            /* 不是索引节点高速缓冲中的索引节点 */
            if（空闲链表上没有索引节点）
                    return（错误）；
            从空闲链表上移出一个新的索引节点；
            重置文件系统号和索引节点号；
            从老的hash队列中撤掉索引节点，把索引节点放到新的hash队列中；
            从磁盘上读索引节点（bread）；
            索引节点初始化（如访问计数值置为1）；
            return（索引节点）；
        }
}
```

2、释放活动索引节点 iput

当核心使用完一个文件要释放其i节点时，首先将该索引节点的引用计数（reference count）减一。

如果引用计数不为零，则表明还有其它进程在使用该文件（节点），本进程直接返回。

如果引用计数为零，则表明没有其它进程使用该索引节点了。此时如果该节点的链接数为零，则表明要删除该文件，释放磁盘icommon和数据块；如果链接数不为零，则把修改过的icommon从inode中写回到磁盘上。

最后再把该inode释放到空闲活动inode表中。

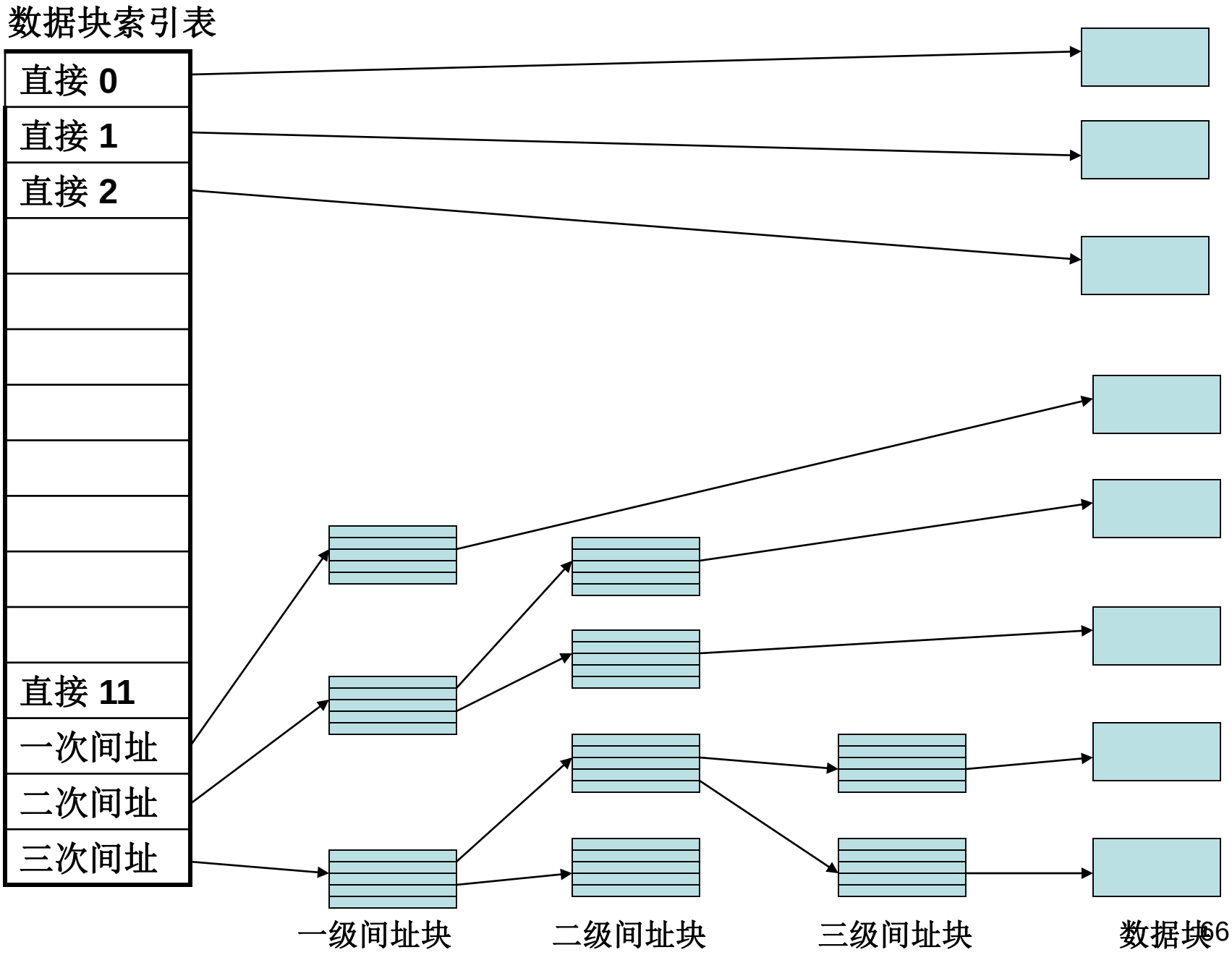
```
算法  iput
输入：指向内存索引节点的指针
输出：无
{
        如果索引节点未上锁，则将其上锁；
        将索引节点的引用计数值减一；
        if（引用计数值为0）
        {
                if（索引节点的链接数为0）
                {
                        释放文件所占的磁盘块（free）；
                        将文件类型置为0；
                        释放索引节点（ifree）；
                }
                if（文件被存取或索引节点被改变或文件内容被改变）
                        修改磁盘索引节点；
                把该索引节点放到空闲链表中；
        }
        为索引节点解锁；
}
```

3、逻辑位移量到文件系统块的映射 bmap

进程用读写指针读写文件时，是以字节为单位来标识的；文件系统下层算法是以数据块来读写数据的，因此必须要把字符偏移量转换成块位移量。

文件占用的数据块在icommon的数据块索引表中标识。

由字节偏移量除以块大小得到块偏移量，判断块偏移量在直接索引表中还是间接表中，如果在直接索引表中，则直接得到块号；如果在间接索引表中，则需先读入间接块，并计算在间接块中的下标，再得到数据块的块号。


数据块索引表
直接 0
直接 1
直接 2
直接 11
一次间址
二次间址
三次间址
一级间址块
二级间址块
三级间址块
数据块

算法 **bmap**

输入：（1）索引节点号

（2）字节偏移量

输出：（1）文件系统中的块号

（2）块中的字节偏移量

（3）块中的I/O字节数

（4）提前读块号

{

由字节偏移量计算出在文件系统中的逻辑块号；

为I/O计算出块中的起始字节； /* 输出2 */

计算出拷贝给用户的字节数； /* 输出3 */

检查是否可用提前读并标记索引节点； /* 输出4 */

决定间接级；

/* 接下页 */

```
        while（没有在必须的间接级上）
        {
                从文件中的逻辑块号计算索引节点中或间接块中的下标；
                从索引节点或间接块上得到磁盘块号；
                如果需要，应释放先前读的磁盘块缓冲区（brelse）；
                if（再也没有间接级了）
                        return（块号）；
                读间接磁盘块（bread）；
                按照间接级调节文件中的逻辑块号；
        }
}
```

4、把路径名转换为索引节点 namei

用户对文件的操作通常是采用带路径的文件名的形式来进行。需要把带路径的文件名转换为文件的索引节点。

在**namei**中，从第一个路径名分量开始，逐个路径名分量进行查找，在指定的目录中从头至尾在各目录项中顺序匹配要查找的路径名分量。

找到相应的目录项后，取出其中的**i**节点号，进入下一级目录查找再下一个分量，直到最后一个路径名分量。

算法 namei

输入：路径名

输出：上了锁的索引节点

{

if（路径名从根开始）

工作索引节点=根索引节点（iget）；

else

工作索引节点=当前目录索引节点（iget）；

while（还有路径名）

{

从输入读下一个路径名分量；

验证工作索引节点确是目录，存取权限OK；

if（工作索引节点为根且分量为“..”）

continue；/* 循环回到while */

通过重复使用算法bmap、bread和brelse来读目录（工作索引节点）；

/* 接下页 */

```
            if（分量与工作索引节点中的一个登记项匹配）
            {
                    得到匹配分量的索引节点号；
                    释放工作索引节点（iput）；
                    工作索引节点=匹配分量的索引节点（iget）；
            }
            else    /*   目录中无此分量   */
                    return（无此索引节点）；
    }
    return（工作索引节点）；
}
```

5、给新文件分配磁盘i节点 ialloc

①、根据预分配的i节点号，实际分配一个i节点(icommon)。预分配的i节点是上一次分配的i节点后面的一个i节点(与预读数据块具有相似的原理)。

②、如果分配不成功，则采用下面的算法快速扫描一遍部分柱面组，寻找空闲i节点。具体算法是：

从预分配i节点所在柱面组的下一个柱面组开始寻找，如果没有找到，就加上一个偏移量后到另外一个柱面组中去寻找；如果还不成功，又再加上一个偏移量到另外一个柱面组中去寻找，如此进行下去。

第n次位移的位移量为2^n，单位为柱面组，终止条件是2^n大于等于文件系统的最大柱面组数，即：

$$E_{max} = \log_2(NCG)$$

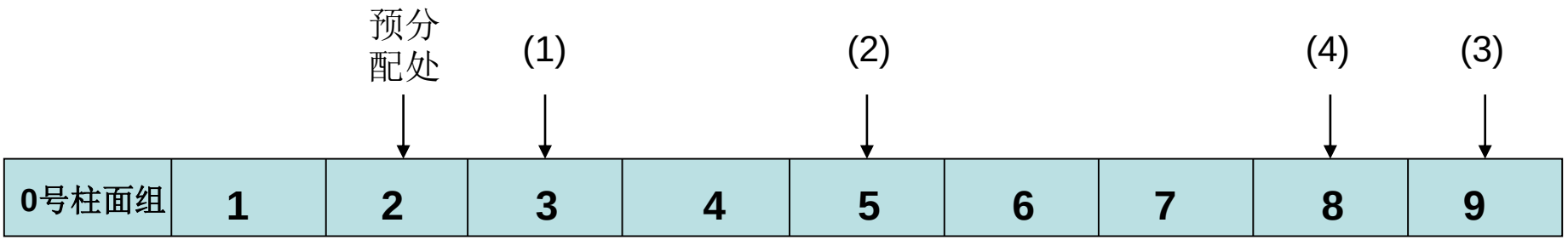
E_{max}取最大整数。

如果第i次位移时当前柱面组号加上2^n后大于最大柱面组号，则把第i次位移点取为当前柱面组号加上2^n后与最大柱面组号的差值。

③、如果上述快速扫描完成后仍然没有找到空闲的i节点，则从预分配点所在柱面组后面的第二个柱面组开始，依次逐个地在各个柱面组中寻找空闲i节点。

④、如果到达最大柱面组处时还未找到，就再从0号柱面组开始到预分配点所在柱面组为止寻找空闲i节点。

⑤、如果还未找到，表明该文件系统中已没有空闲的i节点可供分配了。



预分配处： 2号柱面组

（1）开始（第0次位移）：$2+2^0=3$

（2）$3+2^1=5$

（3）$5+2^2=9$

（4）$9+2^3=17$， $17-9=8$

（5）$8+2^4=24$， $24-9=13$， $13-9=4$，但是 $2^4>9$，故终止；

再从4号柱面组开始向后顺序查找。如果到9号柱面组还未找到，则查找0号和1号柱面组。

说明：

①、这种分配也适合分配磁盘块，实际上ialloc和alloc合用同一个程序；

②、本算法在一级存贮结构的文件系统中也适用，只是将上述的柱面组换成存放硬盘i节点表的各项盘块即可；

③、具体在某一个柱面组中寻找空闲i节点时，是通过查看位示图来进行的；

④、这种算法使得各柱面组中的空闲i节点数分布比较均匀，因为起始点（即预分配的i节点所在的柱面组）是随机任意的（或相邻磁道）。

6、释放磁盘i节点 ifree

当删除一个文件时，需要把该文件对应的icommon释放掉。

核心首先释放该icommon中数据块索引表中标明的数据块，修改超级块中的空闲数据块数，以及数据块位示图；

再修改超级块中的空闲i节点数和i节点位示图。