第六章 文件管理

刘晓建

2017年11月21日

文件系统是操作系统的重要子系统，也是普通用户接触最多的子系统。用户的各种数据通常以文件的形式保存在计算机中。文件通常存储在非易失性存储介质中，因此其生命周期可以跨越进程边界，当进程终止之后，甚至当计算机掉电或重启之后，文件并不会消失，而是持久的保存在计算机中。

在学习文件系统时，尤其需要注意文件的两个基本特性：共享性和安全性。进程的重要特性是“隔离”，而文件的一个重要特性是“共享”。

6.1.1 文件系统的概念

文件系统是管理用户和系统信息的存储、检索、更新、共享和保护，并为用户提供一整套方便有效的文件使用和操作方法的子系统。

文件系统包括：

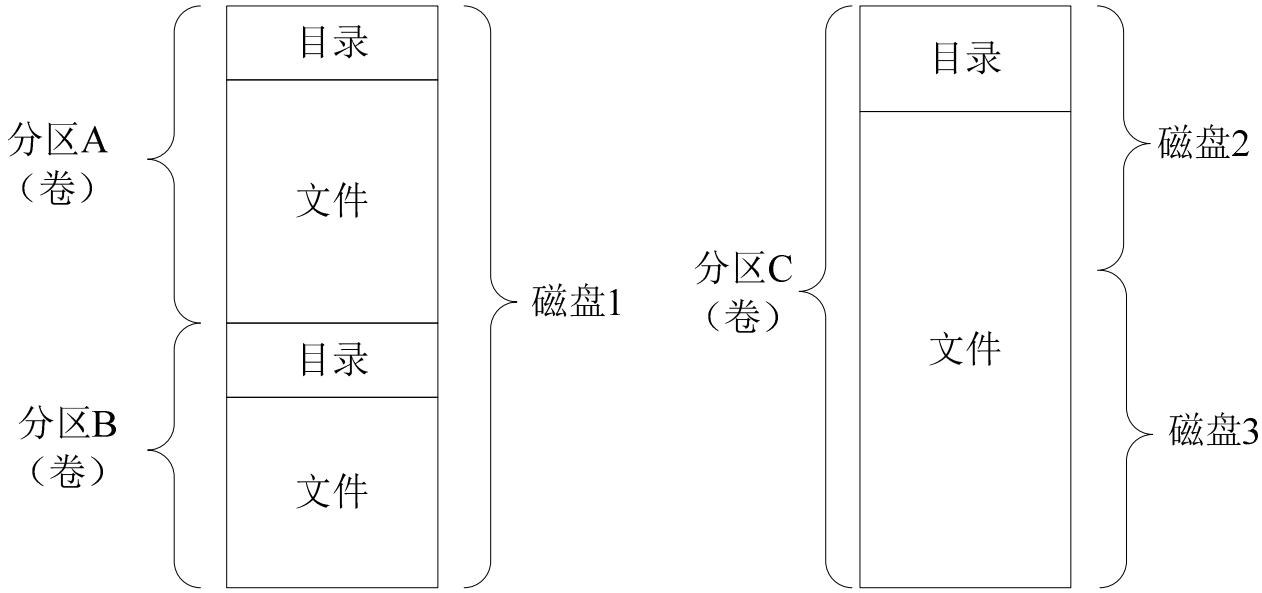
- 文件集合和
- 目录结构两部分，录结构提供系统中所有文件的信息。
- 文件系统驻留在非易失性存储设备上。

一个计算机系统通常支持多个文件系统，文件系统的类别也多种多样。例如，一个典型的Solaris系统可能包括若干个UFS文件系统，一个VFS虚拟文件系统和一些NFS网络文件系统。再如，Windows操作系统支持NTFS和FAT两种文件系统。

6.1.2 文件系统的存储结构

分区：一个磁盘可以分为一个或多个分区，若干个磁盘也可以构成一个分区。分区是存储空间的概念，一个分区中可以不安装任何文件系统。

卷：安装了文件系统的分区就称为“卷”（Volume）。卷中的文件信息存放在卷表中。

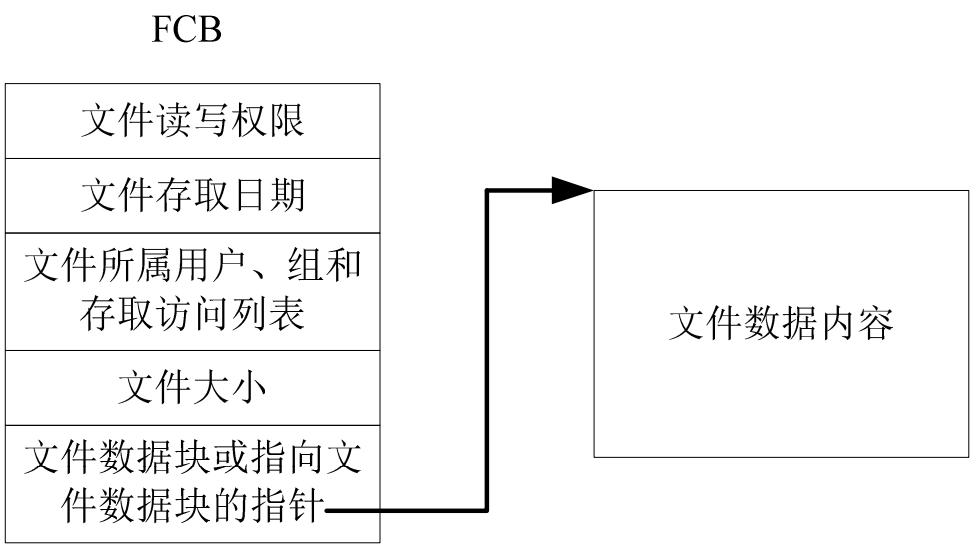


基本信息	
文件名	一个文件通常具有一个容易被人们识记的符号名。一旦文件被命名，它就与进程、用户甚至创建它的系统独立开来。
标识符	文件的唯一标识，通常是一个数字，在文件系统内部标识一个文件。脱离了文件系统，标识符就失去了标识的作用。标识符通常不容易为人们所识记。
文件类型	对支持多种文件类型的操作系统有用
文件组织	供那些支持不同组织的系统使用
地址信息	
位置信息	是一个指向设备的指针和该设备上指向文件存储位置的指针。通常用卷和起始地址来表示。
使用大小	文件的当前大小，单位为字节、字或块。
分配大小	文件所允许的最大大小
访问控制信息	
所有者	控制该文件的用户。所有者可以授权或拒绝其他用户的访问，并可以改变给予他们的权限
访问信息	最简单的形式包括每个授权用户的用户名和口令
允许的操作	读、写、执行以及在网上传送
使用信息	
创建者身份	通常是当前所有者，但并不一定必须是当前所有者
当前使用信息	有关当前文件活动的信息，如打开文件的进程、是否被一个进程加锁、文件是否在内存中被修改但没有在磁盘中修改等
时间、日期和用户标识	文件创建时间、最后一次修改和最后一次使用时间。这些属性对于文件保护、保密和监视文件的使用是非常有用的。

一个文件的所有属性信息构成了该文件的**元信息或元数据**（Metadata）。一个文件的元数据通常保存在**文件控制块（FCB，**File Control Block）中。这样，为了完整的描述一个文件，需要把文件分为两部分：

File = FCB + Content，

其中FCB并不是文件本身，而是描述文件属性信息的数据结构；而Content才是文件所包含的实际数据内容。FCB和Content并不是完全无关的，通过FCB中的“位置信息”属性可以找到文件实际内容所在的存储空间。



操作	说明
创建文件	创建文件需要两步：首先，在文件系统中为该文件找到存储空间。其次，在目录中为该文件建立一个目录项。
写入文件	写入一个文件需要提供文件名和将被写入的信息。文件系统使用文件名在目录中找到文件的位置。系统必须维护一个指向文件写入位置的“写指针”，下一次写操作就从写指针指向的位置开始写入。当写操作发生时，写指针必须更新。
读取文件	读取一个文件需要提供文件名和读取的文件数据应放入的内存缓冲区。同样，需要搜索目录查找对应的目录项，系统需要维护一个指向文件读取位置的“读指针”，下一次读操作就从该位置开始读取。当读操作发生时，读指针必须更新。
文件重定位	把当前文件位置指针设置为给定值。文件重定位操作不会引起任何实际I/O操作。
删除文件	首先根据文件名找到对应的目录项，然后释放文件的所有存储空间，然后删除目录项。
截断文件(Truncating file)	有时用户希望删除一个文件的内容同时保持文件的属性不变。这时，可以对文件施加截断操作。该操作的后效是：除了文件的长度被设为0之外，文件的其它属性均保持不变，而且文件所占的存储空间被释放。

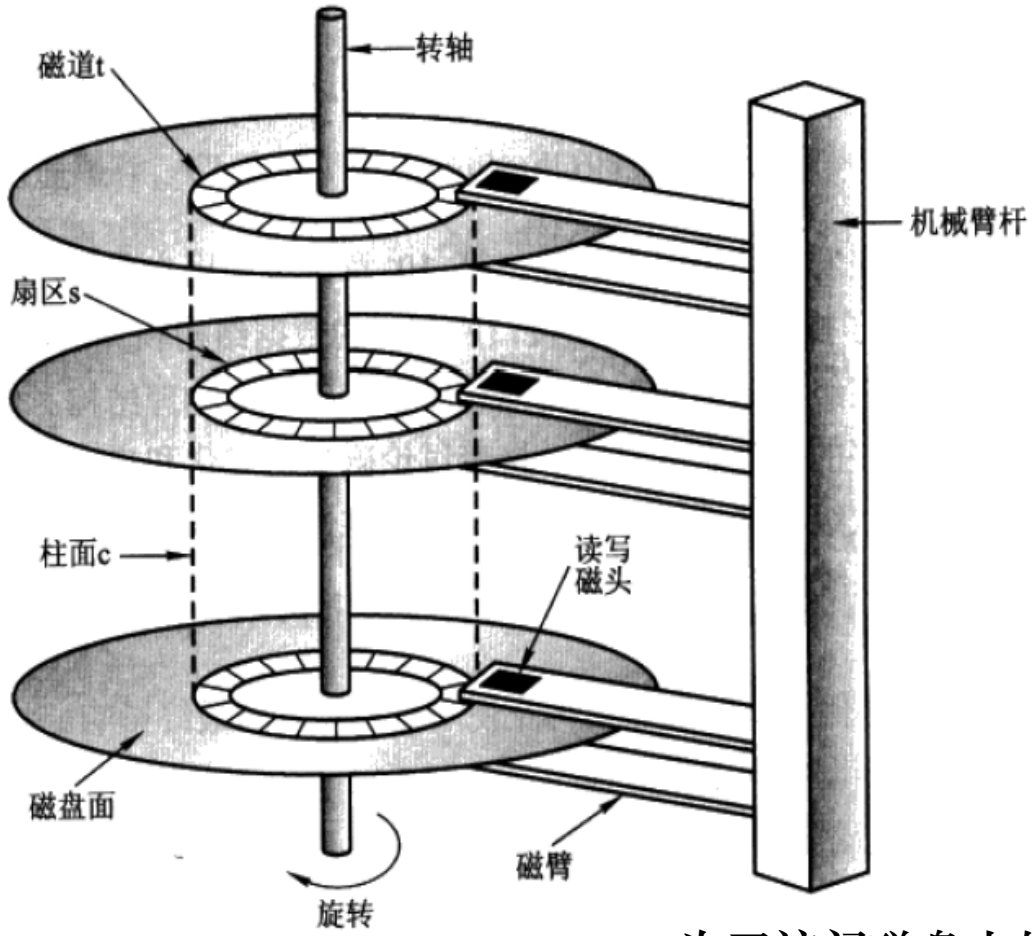
6.2.4 文件的存储设备

常见的存储介质有磁盘、磁带、闪存等。

块是存储介质上连续信息所组成的一个区域，也叫**物理记录**。块是内存和辅助存储设备进行信息交换的物理单位，每次总是交换一块或整数块信息。决定块大小的因素有用户使用方式、数据传输效率和存储设备类型等。不同类型的存储介质，块的大小常常有所不相同；对同一类型的存储介质，块的大小也可以不同。

文件的存储结构依赖于存储设备的物理特性，下面介绍两种不同类型的文件存储设备，它们决定了文件的两种存取方式。

- 顺序存取存储设备：磁带（机）。磁头在磁带的始端，为了读出第100块上的记录信息，必须正向引带走过前面99块。对于磁带机，除了读/写一块物理记录的通道命令外，通常还有辅助命令，如反读、前跳、后退和反绕等，以实现多种灵活的控制。与老式卡带式录音机磁带类似。
- 直接存取存储设备：磁盘。



为了访问磁盘上的一个物理记录，必须给出三个参数：<柱面号，磁头号，扇区号>

特性	Seagate Barracuda 180	Seagate Barracuda 36ES	Toshiba HDD1242	Hitachi Microdrive
应用	高容量服务器	入门台式机	便携	手持设备
容量	181.6GB	18.4GB	5GB	4GB
最小寻道时间	0.8ms	1.0ms	—	1.0ms
平均寻道时间	7.4ms	9.5ms	15ms	12ms
轴心速度	7200r/m	7200r/m	4200r/m	3600r/m
平均旋转延迟	4.17ms	4.17ms	7.14ms	8.33ms
最大传送率	160MB/s	25MB/s	66MB/s	7.2MB/s
每扇区的字节数	512	512	512	512
每磁道的扇区数	793	600	63	—
每柱面的磁道数（盘面数）	24	2	2	2
柱面数（盘片一面的磁道数）	24247	29851	10350	—

6.3.1 文件的逻辑结构

文件的逻辑结构是用户看待和使用文件的方式。用户不必考虑文件信息在物理介质上的存储问题，只需了解文件的逻辑结构，利用文件名和有关操作就能存储、检索和处理文件信息。

- 流式文件——文件的数据是一串字节流。这种文件常常按长度来读取所需信息，可以用插入的特殊字符作为分界。如Unix文件，文本文件。
- 记录式文件——由若干逻辑记录组成，逻辑记录是相关数据项的集合。如数据库表文件。

	数据项1	数据项2	数据项3	数据项4	数据项5
	学号	姓名	出生年月	性别	籍贯
记录1	0001	张三	1994.1	男	陕西
记录2	0002	李四	1994.10	女	湖北
…	…	…	…	…	…
记录100	000100	王五	1993.7	男	北京

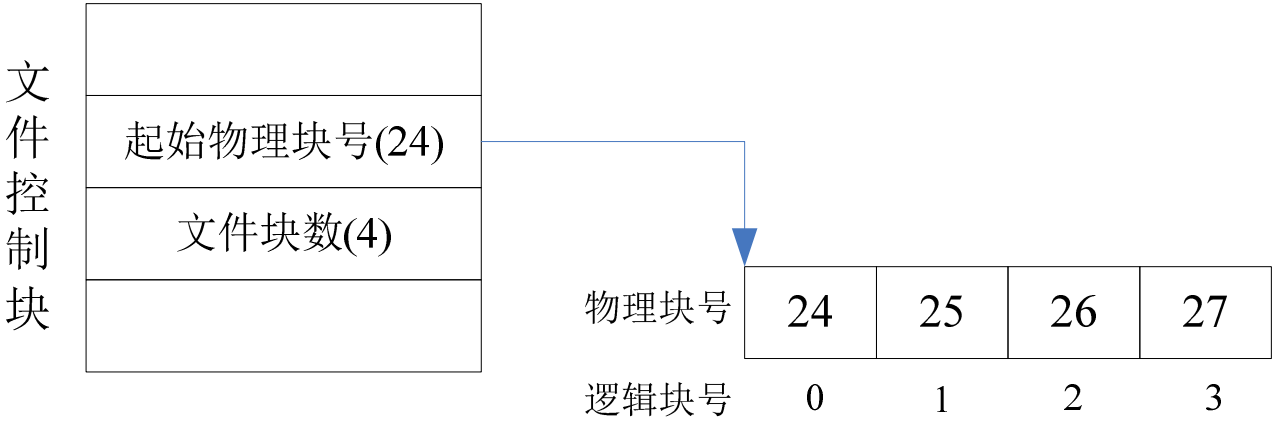
6.3.2 文件的物理结构

文件的**物理结构和组织**是指逻辑文件在物理存储空间中的存放方式和组织关系。这时，文件被看作为物理文件，即相关物理块的集合。构造文件物理结构的方法有两类。

- **计算法**——设计一个映射算法，例如线性计算法、杂凑法等，把记录键转换成对应的物理块地址，从而把一条逻辑记录映射到一个物理块。存取效率较高，又不必增加存储空间存放附加控制信息，能把分布范围较广的键均匀的映射到一个存储区域中。
- 指针法——这类方法设置专门指针，指明相应记录的物理地址或表达各记录之间的关联。优点是可将文件信息的逻辑次序与存储介质上的物理排列次序完全分开，便于随机存取，便于更新，能加快存取速度。但使用指针要耗用额外的存储空间，大型文件的索引查找要耗用较多处理机时间。

下面介绍几种常用的文件物理结构和组织形式。

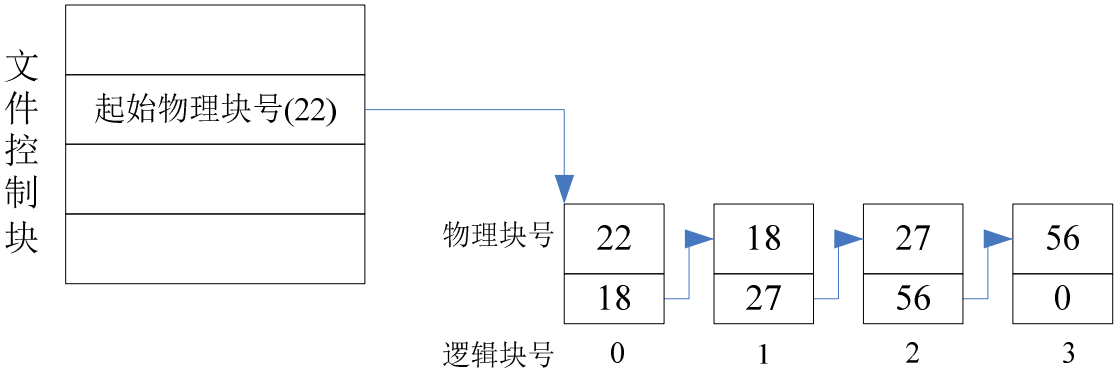
顺序文件——将一个文件的信息存放到依次相邻的存储块上，这类文件也叫**连续文件**。显然，顺序文件的逻辑记录与物理记录顺序完全一致。



优点是：顺序文件的记录按物理邻接次序排列，存取记录时速度较快，所以批处理文件，系统文件用得最多。然而，由于多道程序的访问，在同一时刻另外的用户进程可能驱动磁头移向其它文件，打破了原来文件的顺序存取，因而降低了顺序文件的这一优势。

缺点是：建立文件前需要能预先确定文件的长度，以便分配存储空间；修改、插入和增加文件时有困难；

连接文件——连接文件又称串联文件，其特点是使用链接字（或指针）来表示各个物理块之间的关联。第一块文件信息的物理地址由文件控制块给出，而每一块的链接字指出了下一个物理块。连接字为0时，表示文件至本块结束。

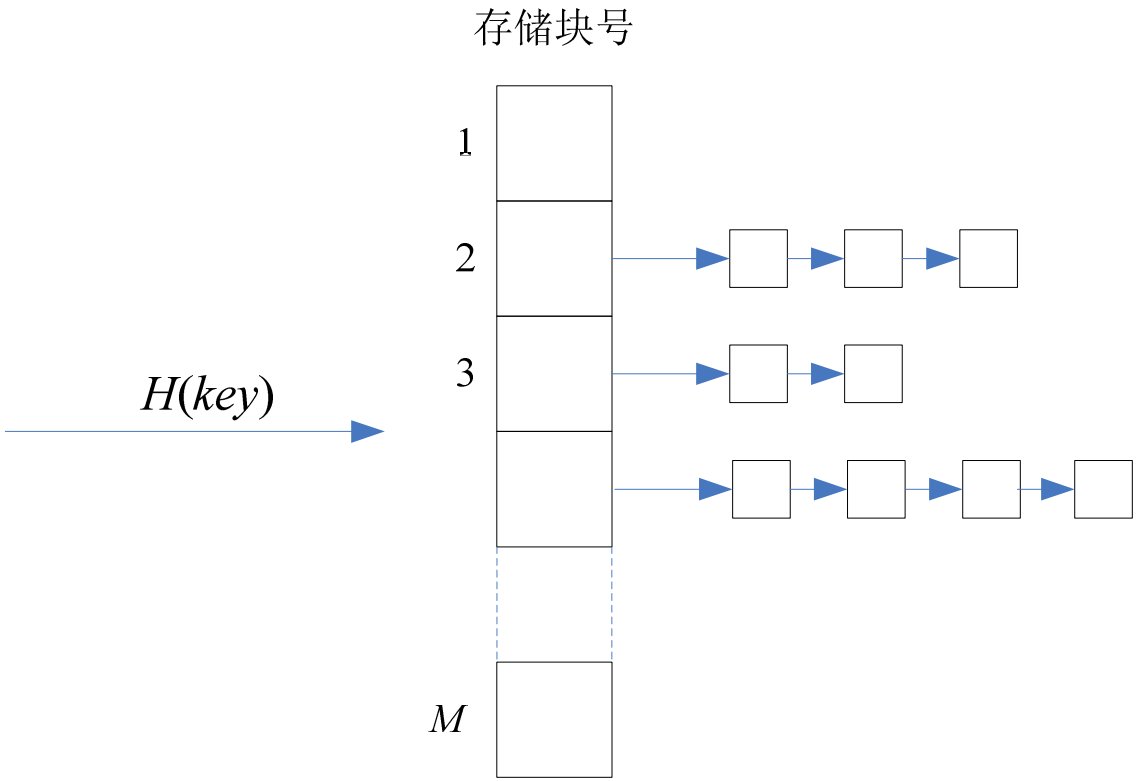


优点是：易于对文件记录作增、删、改，易于动态增长记录；不必先确定文件长度；不必连续分配，从而存储空间利用率高。

缺点是：存放指针需额外的存储空间；必须将连接字和数据信息存放在一起，破坏了物理块的完整性；

由于存取需通过获得链接字后，才能找到下一物理块的地址，因而仅适合于顺序存取。

直接文件（散列文件）——通过某种函数变换*H*，把逻辑记录的关键字*key*直接映射到该记录存储地址的一种文件物理结构形式。



理想情况下，如果函数H满足如下两个条件，我们就可以通过计算直接得到记录$R[i]$的物理块号：

- 对于所有的关键字key，$H(key)$的值在1和M之间；
- 对任意的两个关键字key_i和key_j（$i \neq j$），都有$H(key_i) \neq H(key_j)$。

但大部分情况下，一个存储块可以存放多条记录，记录键值范围也大大超过所用地址的范围，这时就会出现多个键值被映射到同一个物理块的情况，我们把这种情况称为“冲突”。

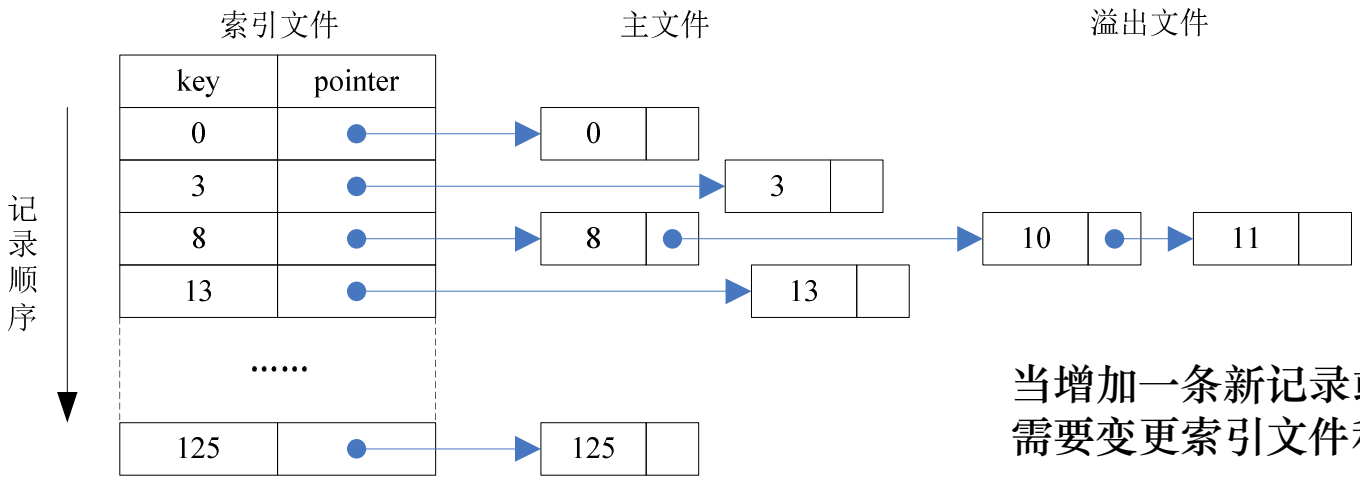
这种存储结构用在不能采用顺序组织法、次序较乱又需在极短时间内存取的场合，象实时处理文件、操作系统目录文件、编译程序变量名表等。

索引顺序文件

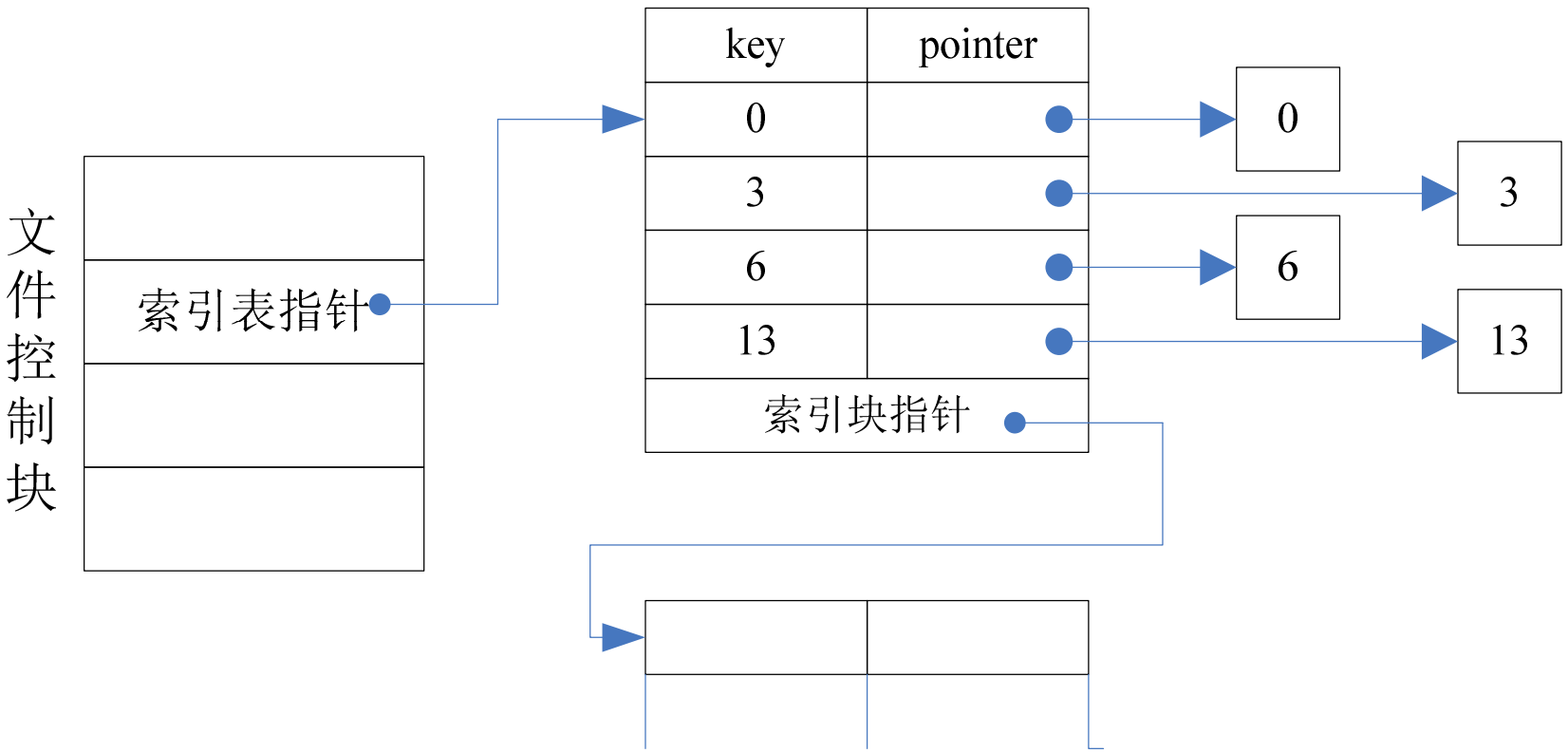
索引结构通过在文件中增加一个冗余数据结构，即索引表，来实现文件的非连续存储以及提高记录的访问速度，它适用于数据记录保存在随机存取存储设备上的文件。索引表由若干表目按照一定顺序组成，其中每个表目是一个偶对：

<key, pointer>

其中，key是一个记录键，pointer是该记录的存储地址，存储地址可以是记录的物理地址。索引表实际上是一个简单的顺序文件，称为**索引文件**，简称**索引**。



当增加一条新记录或删除一条记录时，需要变更索引文件和主文件。


文件控制块
索引表指针
key
pointer
0
3
6
13
索引块指针
0
3
6
13

索引文件结构具有如下优点：

• 文件逻辑记录的顺序与记录的存储顺序相分离，因此不需要预先确定文件的长度，对于文件内容的修改、插入、增加和删除没有限制；

• 索引文件可以方便的进行随机存取。通过记录键值，在索引表中找到对应的存储块号，然后通过随机存取存储设备直接存取存储块。

缺点：

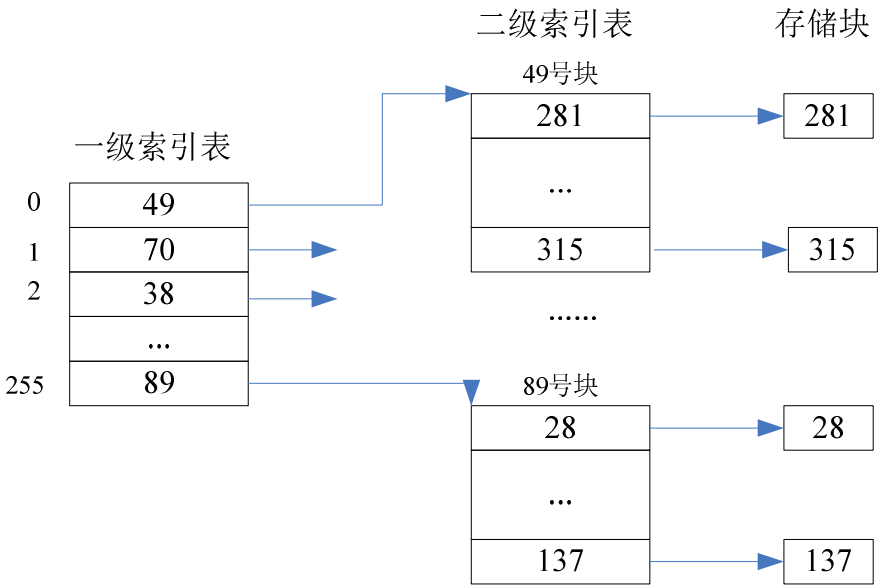
• 索引文件的缺点是保存索引表本身需要一定的存储空间开销，有时索引表的信息量甚至可能远远超过文件记录本身的信息量；

• 增加了文件存取时间。在存取文件时，需要两步操作：第一步查找文件索引表，获得记录的物理地址；第二步以物理地址获得记录数据。

索引表的存储

当索引表的表目不多时，仅用一个存储块就可以容纳，但当表目增多时，一个存储块不够存放索引表时，就需要多个存储块来存放。存放索引表的存储块称为索引表块。多个索引表块的组织方式有两种：

- 连接文件方式：将多个索引表块按连接文件的方式链接起来
- 多重索引方式：建立多级索引表

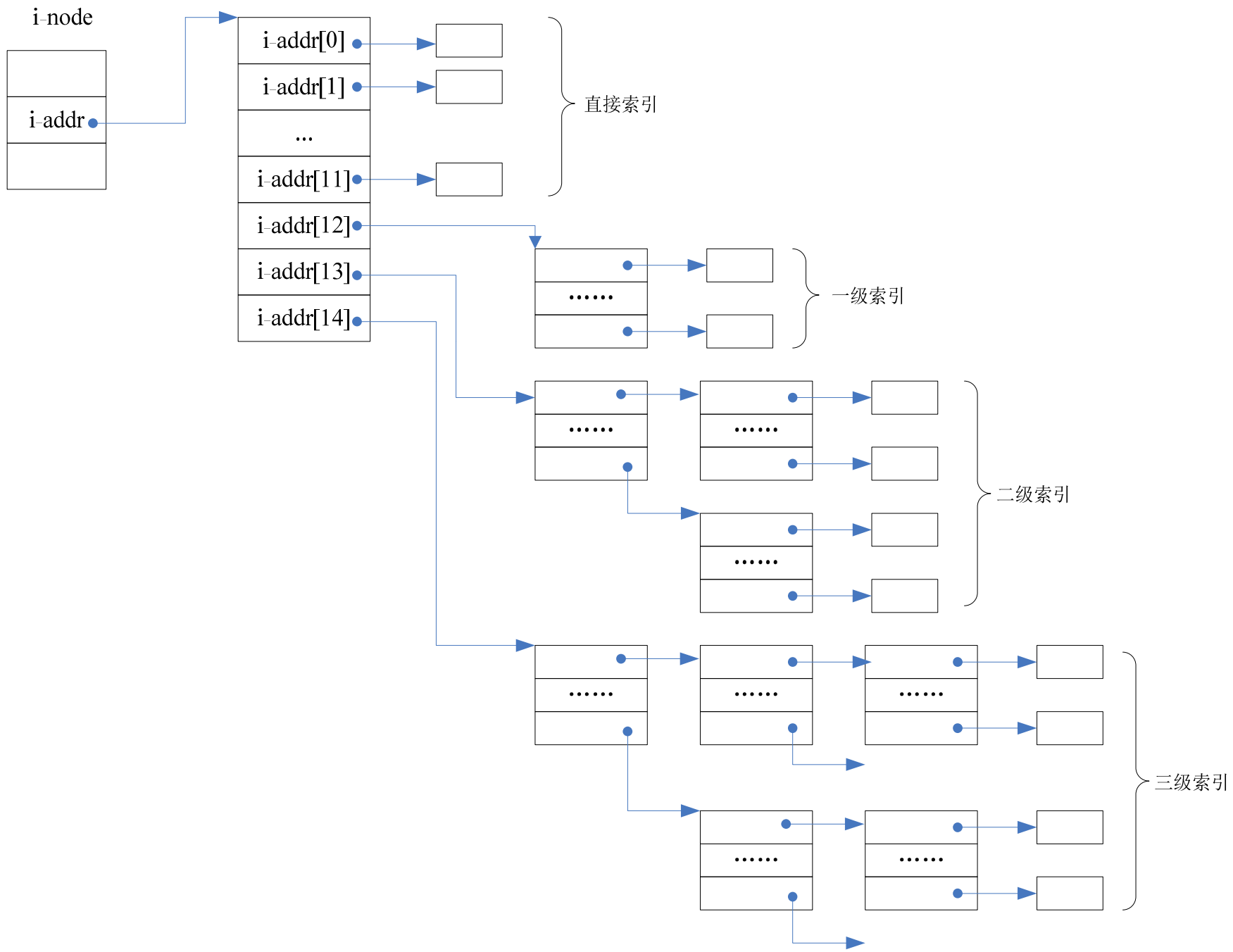


【例6-1】设一个存储块大小为512字节，每个索引表项占2个字节。如果采用两级索引表结构来描述一个文件的物理结构，问：

- 能表示的文件最大大小是多少字节?
- 两级索引表最多占多少个存储块?
- 两级索引表占据的存储空间相比文件存储空间的比例是多少?

UNIX文件的物理结构——混合索引表

在UNIX系统中，采用**流式文件**作为文件的逻辑结构，采用**混合索引表**来描述文件的物理组织结构。一个文件的混合索引表位于该文件的i-node数据结构中。在i-node中定义了一个120字节的数组i-addr[15]，每个元素占8个字节，用于存放磁盘地址或指针。系统把常规文件分成小型、中型、大型和巨型四种文件。


i-node
i-addr
i-addr[0]
i-addr[1]
...
i-addr[11]
i-addr[12]
i-addr[13]
i-addr[14]
直接索引
一级索引
二级索引
三级索引

【例6-2】对于如图6-11所示的UNIX文件物理结构。假设逻辑块和磁盘块大小都是8KB，磁盘块指针是32位，其中8位用于标识物理磁盘，24位用于标识物理块，那么

- 该系统支持的文件大小最大是多少?
- 该系统支持的分区数最多有多少？最大文件系统分区是多少?
- 假设只有文件索引节点保存在内存中，访问第13423956字节需要多少次磁盘访问?

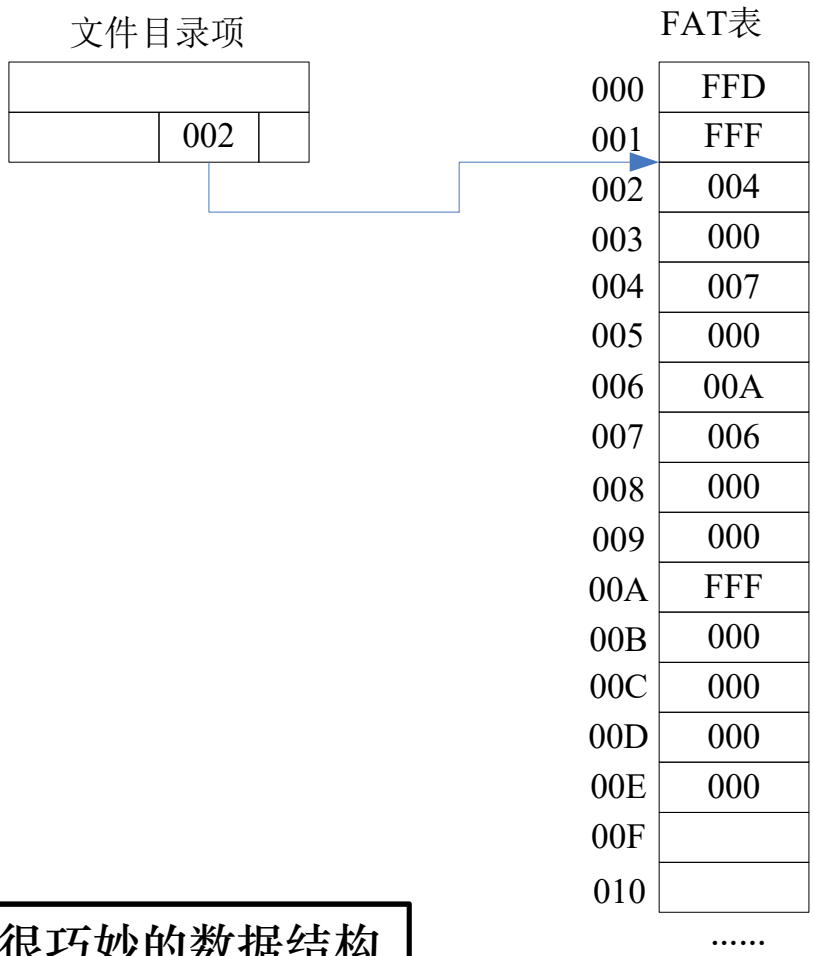
§6.4 文件存储空间管理

磁盘等大容量存储空间被操作系统及多个用户共享，用户进程运行期间常常要建立和删除文件，操作系统应能自动管理和控制存储空间。存储空间的有效分配和释放是文件系统应解决的一个主要问题。常用的存储空间管理方法有以下几种。

- 空闲区表
- 空白块链
- 位示图

位	0	1	2	3	4	5	6	7
字节1	0	0	0	1	1	0	0	1
字节2	0	0	1	1	1	0	0	0
……	……							
字节n	0	0	1	1	1	1	1	1

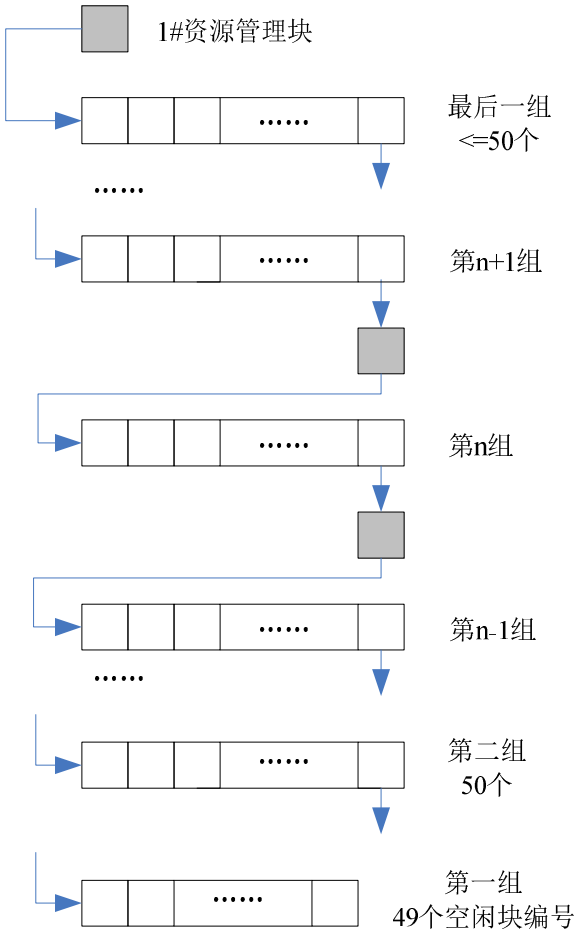
6.4.4 MS-DOS的盘空间管理



FAT相当于一个单向链表，很巧妙的数据结构

6.4.5 UNIX文件存储空间的管理

成组空闲块编号



成组空闲块编号构成栈结构

【例6-5】假定磁盘中空闲块编号的范围是150~250，351~380，401~449。请用成组链接法组织这些空闲块。

解：首先按照空闲块编号从大到小的次序把空闲磁盘块划分成四组：

第一组：401~449，恰好49个块；

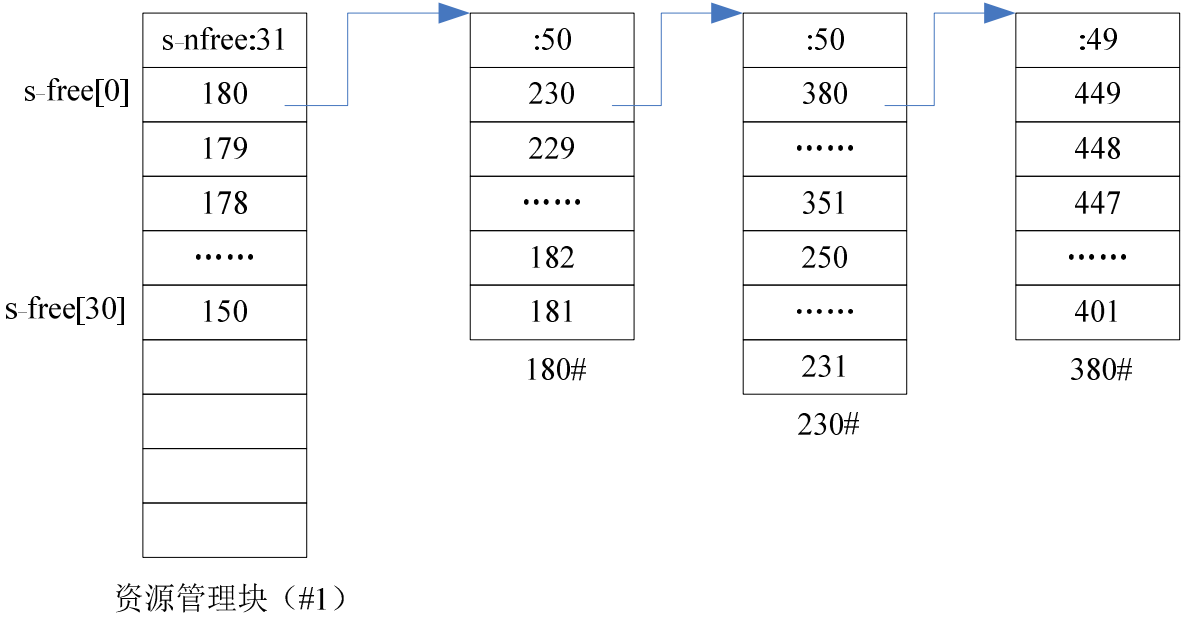
第二组：231~250，351~380， 50个块；

第三组：181~230，共50个块；

最后一组：150~180，共31个块。

思考：

实际用来存放空闲块编号的磁盘块有几块?



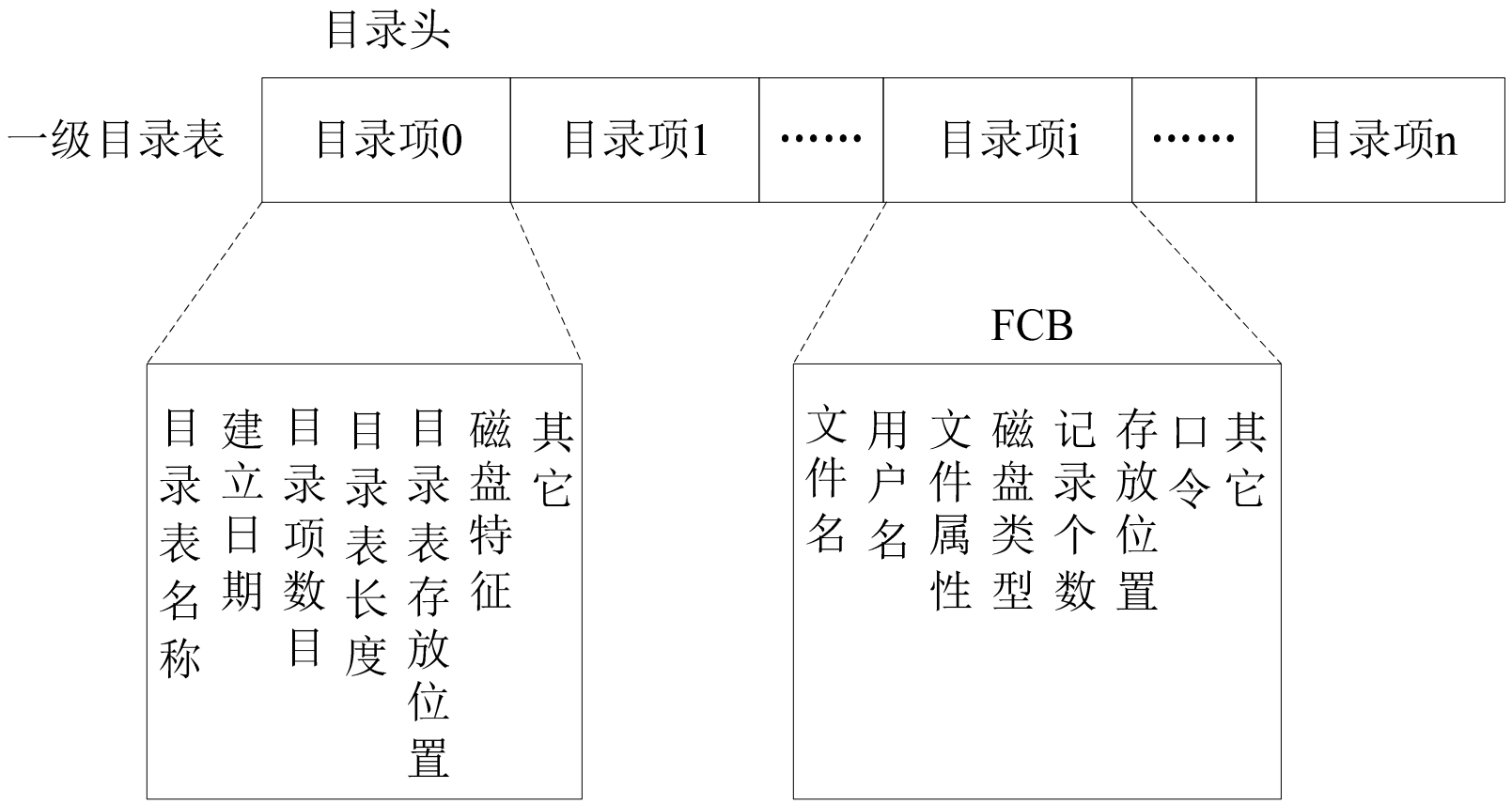
§6.5 文件目录结构

文件系统包括文件的集合和目录结构。目录结构的功能有两个:

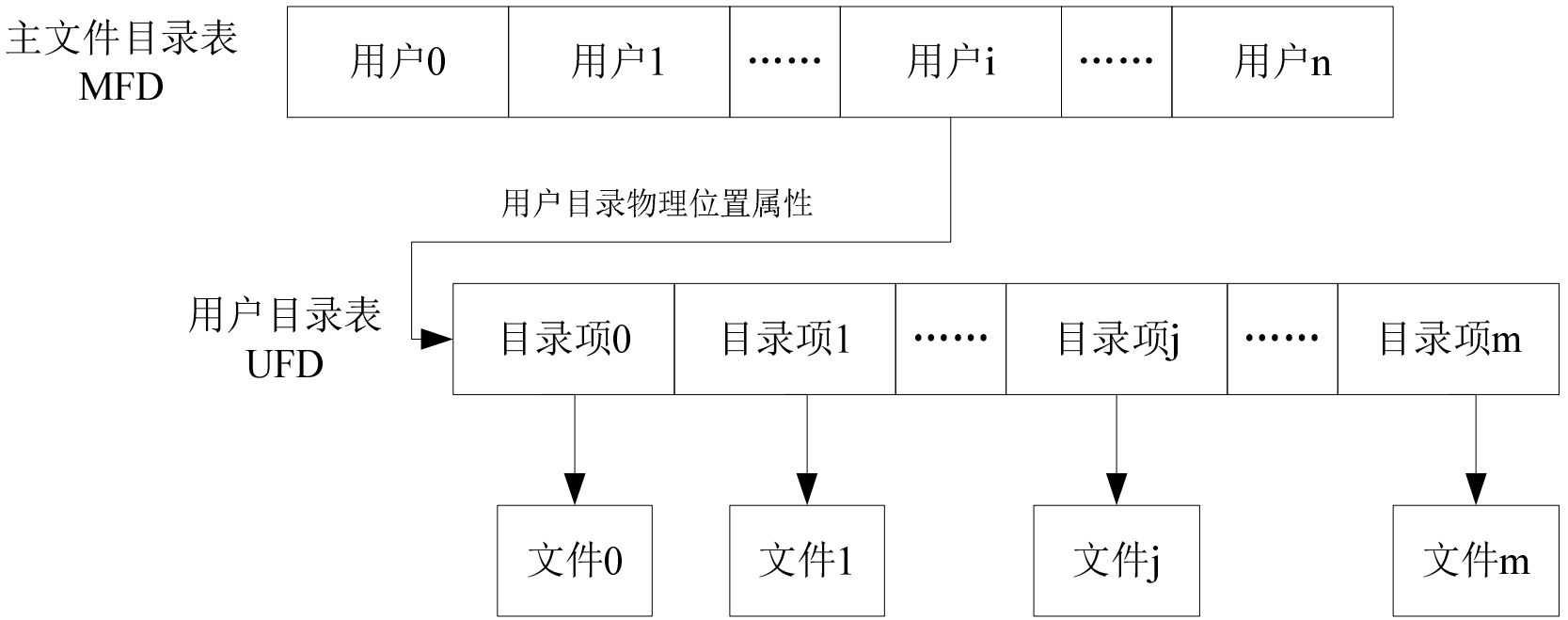
- 一是存放目录下的文件的相关信息（即文件控制块FCB），——既然目录存放文件信息，那么目录本身也就是一个文件。
- 二是提供通过文件名搜索文件属性信息和文件数据内容的机制。

目录也必须保存在非易失性存储介质中。

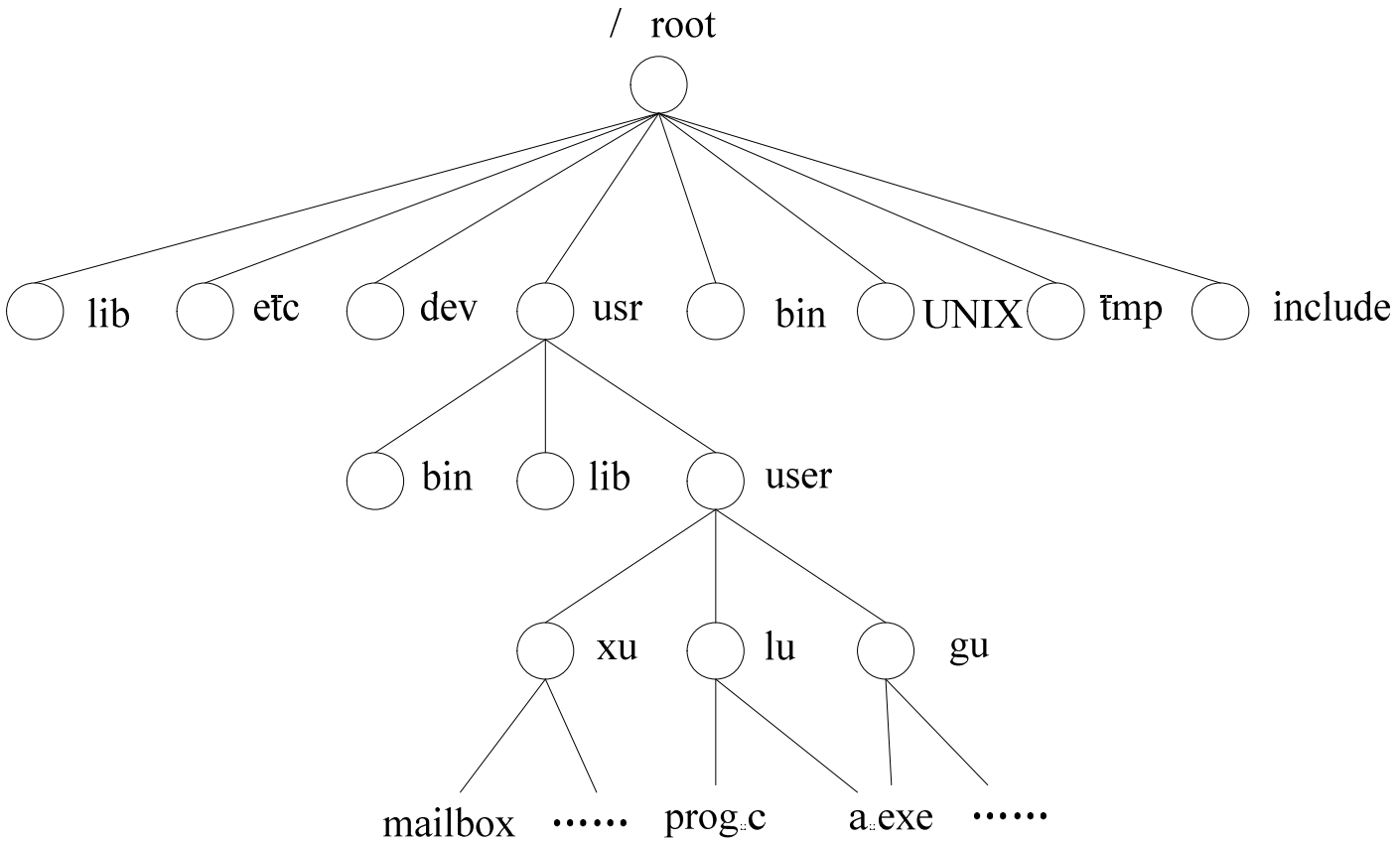
单级目录（Single-Level Directory）



二级目录


主文件目录表
MFD
用户0
用户1
……
用户i
……
用户n
用户目录物理位置属性
用户目录表
UFD
目录项0
目录项1
……
目录项j
……
目录项m
文件0
文件1
文件j
文件m

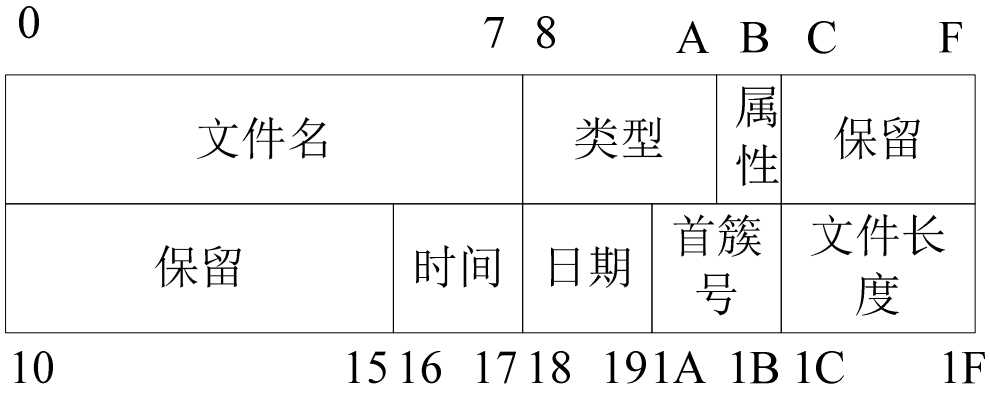
多级目录


/ root
lib
etc
dev
usr
bin
UNIX
tmp
include
bin
lib
user
xu
lu
gu
mailbox
……
prog.c
a.exe
……

6.5.2 目录和目录项的实现

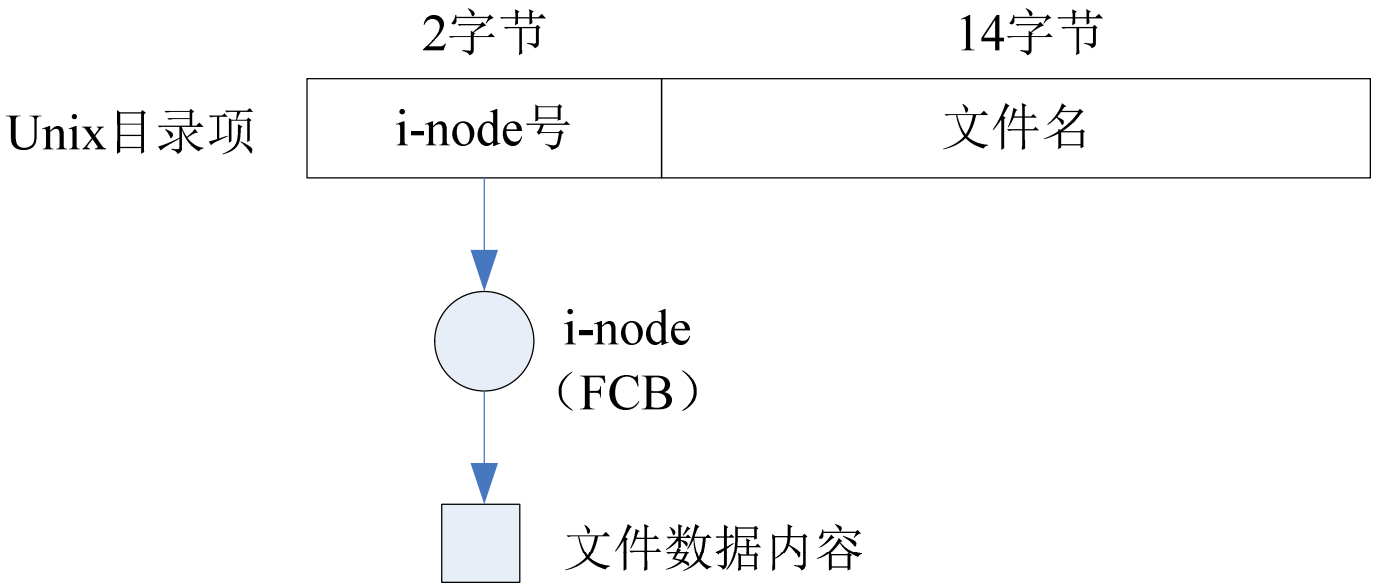
目录是目录项的一个线性列表。实现目录结构的关键在于定义一个目录项。不同操作系统定义目录项的方式不同。

MS-DOS中的目录项

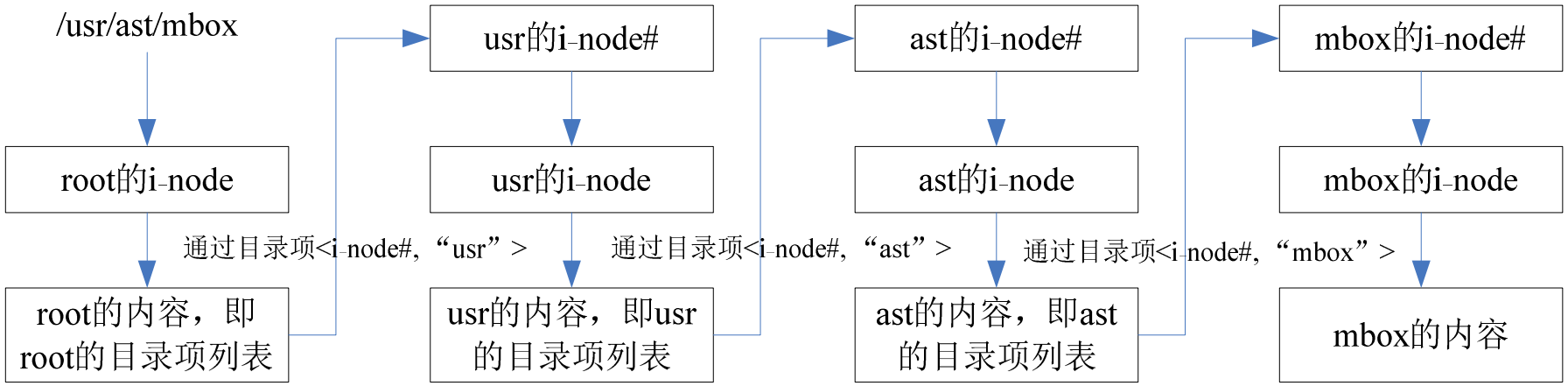


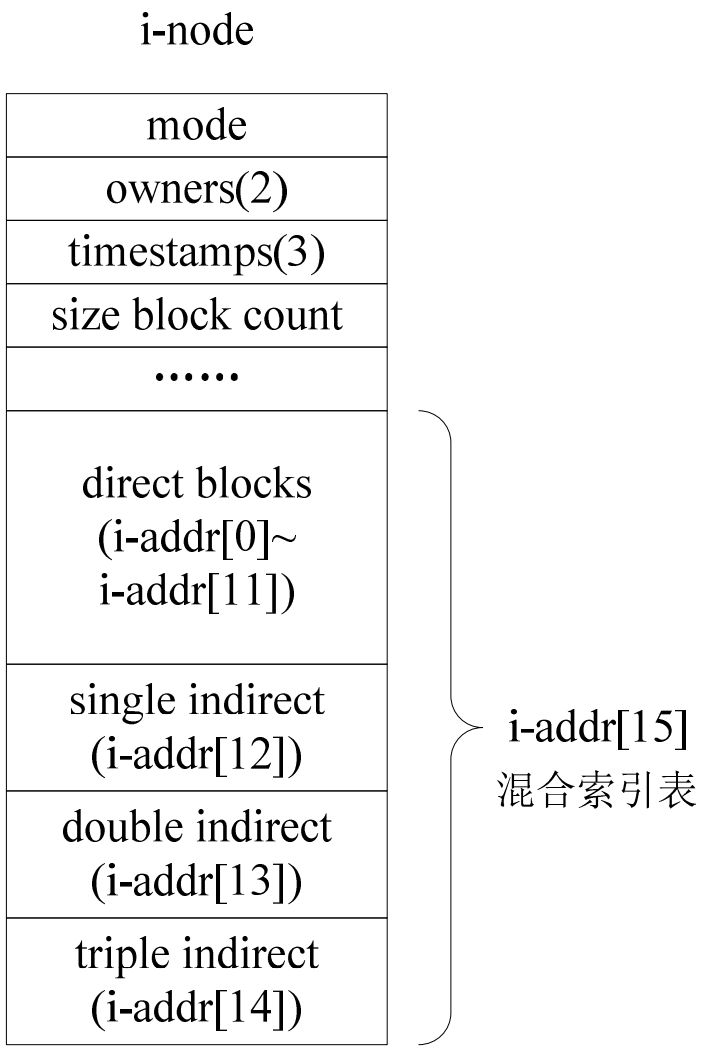
UNIX中的目录项

UNIX中每个目录项仅包含一个文件名及其i-node号。文件的其它属性均放在i-node中。根据i-node号，在专门存放i-node的磁盘区域中，可以找到对应的i-node，进而获得文件的所有属性、控制管理信息以及文件数据内容的存储分布。

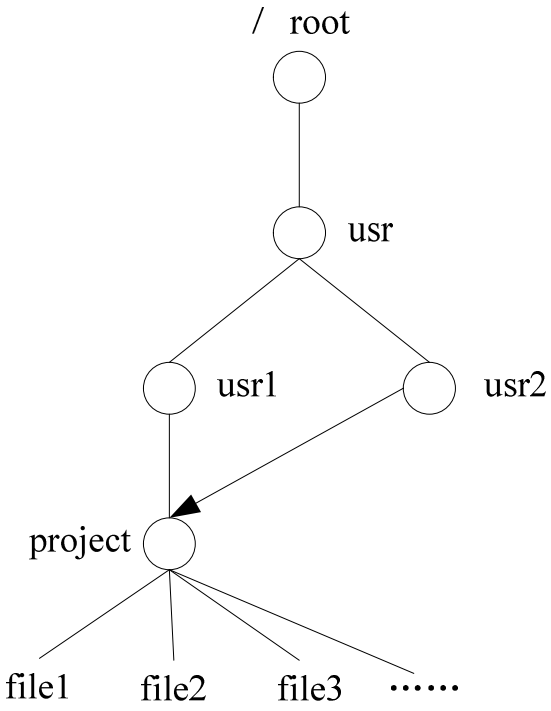


例如，要找到文件/usr/ast/mbox。根目录“/”、子目录usr和ast都有相应的i-node，因此首先从根目录的i-node找起。由于文件系统总是能够知道根目录i-node的存储位置，因此能够访问到根目录i-node。整个查找过程如下：

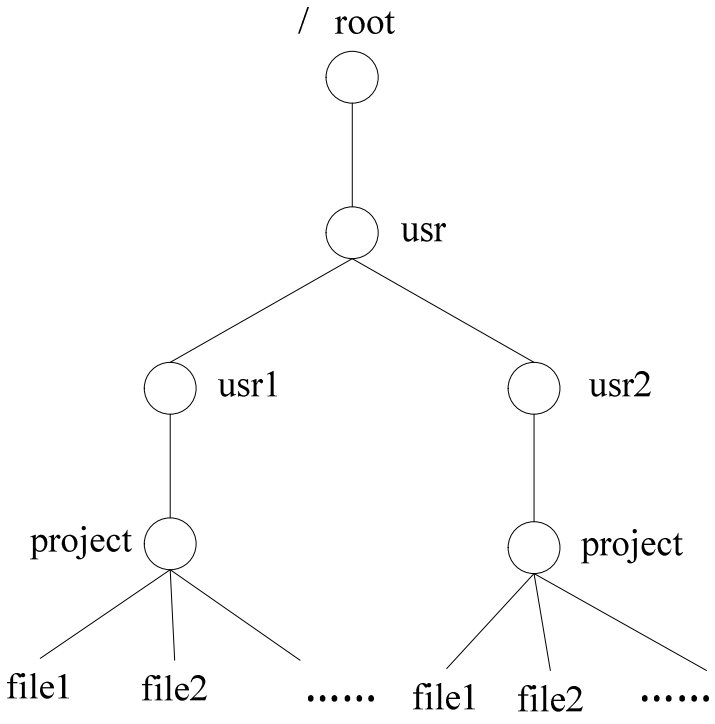



i-node
mode
owners(2)
timestamps(3)
size block count
……
direct blocks
(i-addr[0]~
i-addr[11])
single indirect
(i-addr[12])
double indirect
(i-addr[13])
triple indirect
(i-addr[14])
i-addr[15]
混合索引表

6.5.3 文件链接——文件共享的需求



建立链接的方式

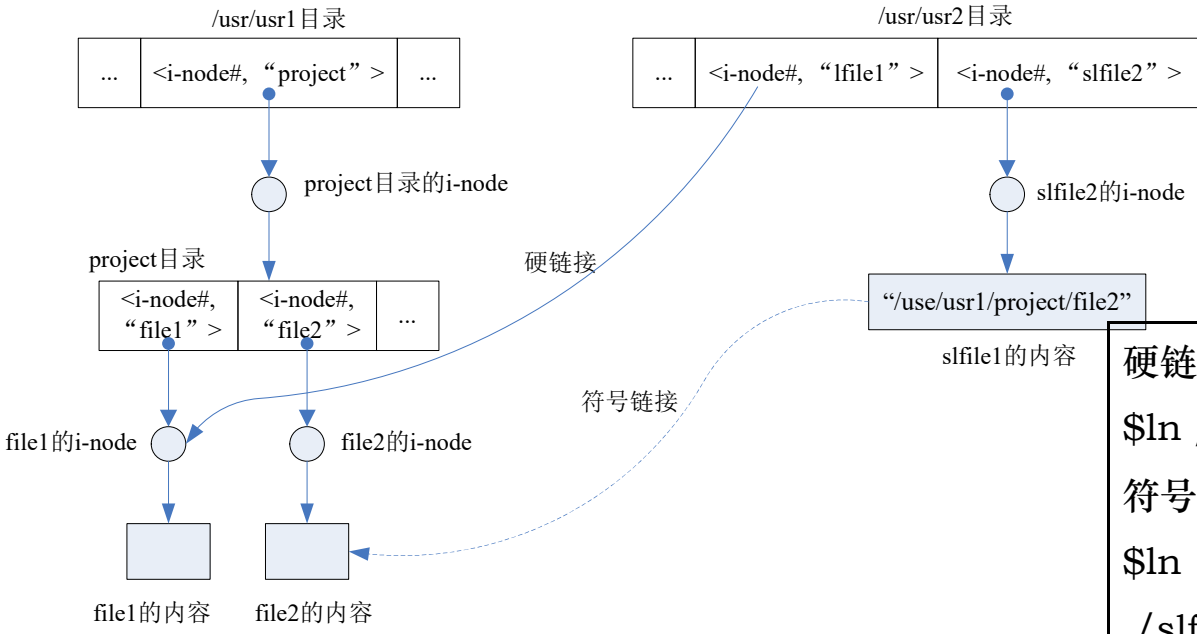


copy的方式

硬链接和符号链接

在UNIX中文件和目录的共享可以通过两种方式来实现：

- **硬链接**（Hard link）（或简称**链接**）：实际上并没有拷贝该文件，而是建立了一个指向该文件的i-node的目录项；
- **符号链接**（Symbolic link）。实际上是建立了一个新文件，这个新文件的内容记录了被链接文件的路径。



硬链接：

$ln /usr/usr1/project/file1 ./lfile1

符号链接：

$ln -s /use/usr1/project/file2 ./slfile2

链接计数

无论是硬链接还是符号链接，成功的建立链接之后，就可以通过多条路径访问到同一个文件或目录。这样做的好处是实现了文件的共享，但是也给文件管理带来了额外的复杂性。在文件的i-node中引入一个“链接计数”（Link count）属性，用来记录i-node上硬链接的个数。

- 当一个文件创建时，count=1，当在其上建立一个硬链接之后，count:=count+1。
- 当删除一个文件时，分两种情况：
  - 当通过硬链接删除一个文件时，首先把i-node的计数递减，即count:=count-1，这时如果count！=0，说明还有其它链接引用该文件，那么就不能删除该文件；如果count=0，说明没有任何链接引用该文件，这时就可以将其安全的删除掉，并释放其存储空间。
  - 当通过符号链接删除文件时，只会删除符号链接文件本身，而对于被链接文件及其i-node不会造成任何影响；当通过其它非符号链接路径删除被链接文件时，不会对符号链接造成任何影响；当一个文件被删除并释放后，仍然可以通过符号链接访问该文件，但是这时文件系统会抛出错误信息。

	硬链接	符号链接
是否能在目录上建立	一般不允许在目录上建立	可以在文件或目录上建立
对i-node的影响	硬链接实际上是一个目录项，它指向被链接对象的i-node。建立硬链接时，i-node的链接计数递增。	符号链接实际上是一个新文件，它拥有自己的i-node，与被链接文件的i-node无关，也不会引起被链接文件i-node的任何变化。
名字解析快慢	因为硬链接包含对象的直接引用，因此解析速度较快。	符号链接包含对象的路径名，该路径名必须被再次解析以找到相应的对象，因此总体解析速度较慢。
对象是否存在	建立硬链接时，被链接对象必须首先存在。	建立符号链接时，被链接对象不要求必须存在。
对象删除	为了删除一个对象，该对象上的所有硬链接必须被完全解除。	一个对象可以被删除，即使还有指向它的符号链接存在。
范围	硬链接只限定在同一文件系统内部。	符号链接可以跨文件系统，甚至跨计算机、跨网络。
作用	提供文件共享功能；为重要文件建立硬链接，可以防止误删。	提供快捷对象引用方式。

§6.6 文件共享

在一个进程打开并使用一个文件期间，另一个进程也可以打开并使用该文件，于是该文件就在这两个进程间得到共享。文件共享为操作系统带来了额外的复杂性：

- 一是并发控制的复杂性。
- 二是文件管理的复杂性，共享语义变得更复杂。

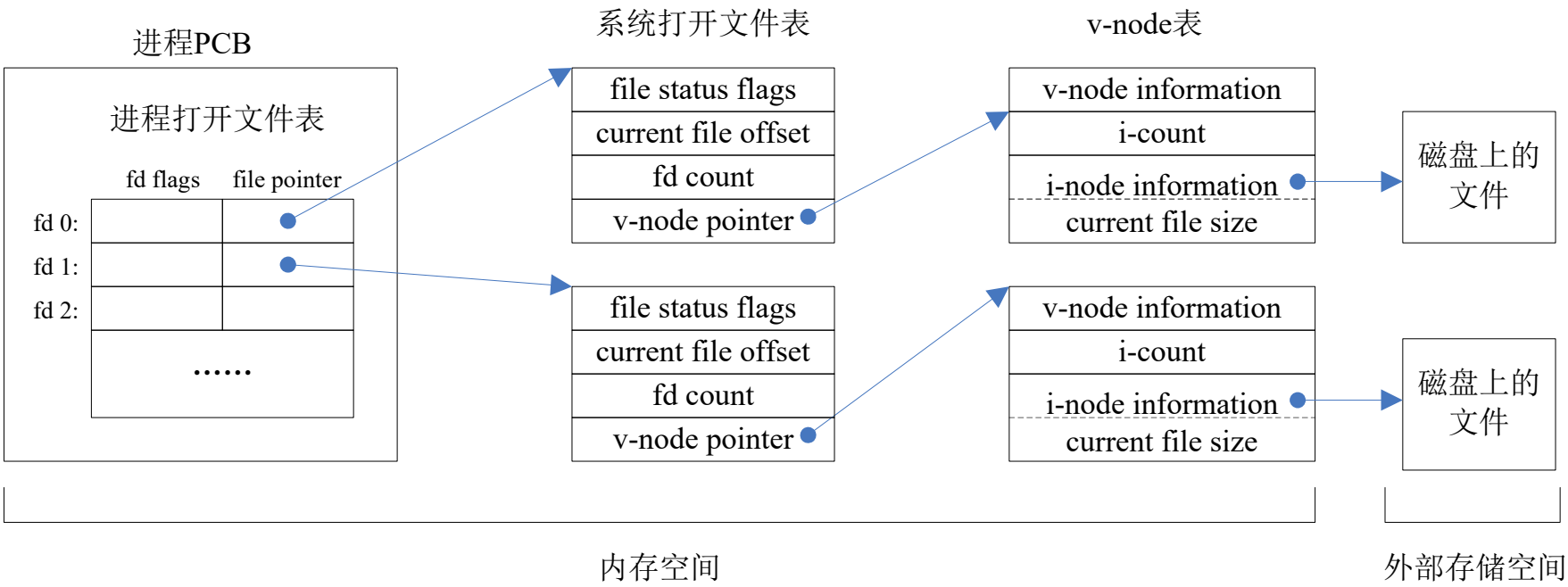
文件共享在带来复杂性的同时，也为我们带来了一些便利。正是由于文件可以在多个进程间共享，我们才能通过文件在进程间方便的进行通信。

内核中用三个数据结构来表示被打开的文件（以下简称“打开文件”）：

•每个进程的PCB中维护一个**进程打开文件表**（per-process open-file table），每一个打开文件表项代表一个打开文件的描述符fd（file descriptor）（Windows系统中称为文件句柄file handler）。

• 内核为所有打开文件维护一个**系统打开文件表**（system-wide open-file table），每个打开文件占据一个表项。表项中包含：文件打开状态标志；当前文件偏移量；引用计数fd count和一个指向v-node表项的指针。

• v-node表。每个v-node表项中包含文件i-node信息，这些信息是在文件被打开时，从磁盘中的i-node读入的，因此关于文件的所有持久性信息都可以从v-node得到，例如文件的拥有者、文件大小、存储文件的数据块的指针等。


进程PCB
进程打开文件表
fd flags
file pointer
fd 0:
fd 1:
fd 2:
.....
系统打开文件表
file status flags
current file offset
fd count
v-node pointer
file status flags
current file offset
fd count
v-node pointer
v-node表
v-node information
i-count
i-node information
current file size
v-node information
i-count
i-node information
current file size
磁盘上的文件
磁盘上的文件
内存空间
外部存储空间

6.6.2进程间的文件共享

当一个进程首次打开一个文件时，操作系统将为该文件分配一个进程打开文件表项、系统打开文件表项和v-node表项，并将磁盘中i-node的信息拷贝到v-node表项中，建立起打开文件的内核数据结构。另一个进程可以通过两种方式共享一个已经打开的文件：

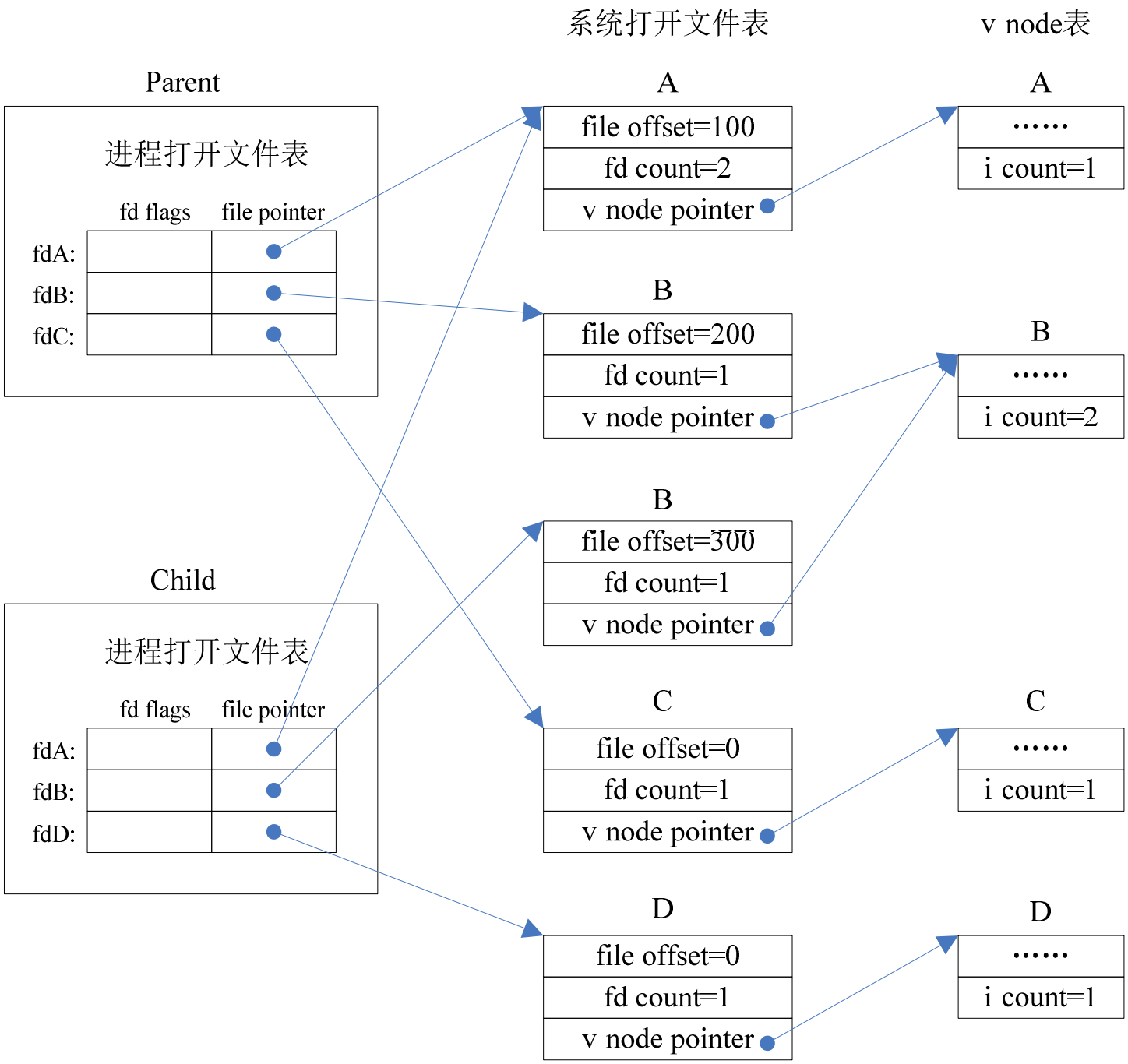
（1）父进程通过fork创建子进程让其共享已经打开的文件；——共享的是进程打开文件表项

（2）一个进程通过同名或异名打开一个文件来共享。——共享的是系统打开文件表项

【例6-8】假定父进程Parent通过fork调用创建了一个子进程Child。它们的文件操作如下伪码所示。

- 进程Parent的打开文件A能否被进程Child共享？如果能够共享，这时Child观察到的文件A的读指针（即文件偏移量）是多少？
- 文件B能否被两个进程共享？两个进程观察到的文件B的读指针是否会相互干扰？

Parent:	Child:
fdA=open(“A”);	fdB=open(“B”);
read(fdA, bufA, 100);	read(fdB, buf, 300);
fork();	fdD=open(“D”);
fdB=open(“B”);	
read(fdB, bufB, 200);	
fdC=open(“C”);	


系统打开文件表
v node表
Parent
进程打开文件表
fd flags
file pointer
fdA:
fdB:
fdC:
Child
进程打开文件表
fd flags
file pointer
fdA:
fdB:
fdD:
A
file offset=100
fd count=2
v node pointer
B
file offset=200
fd count=1
v node pointer
B
file offset=300
fd count=1
v node pointer
C
file offset=0
fd count=1
v node pointer
D
file offset=0
fd count=1
v node pointer
A
……
i count=1
B
……
i count=2
C
……
i count=1
D
……
i count=1

6.6.3 打开文件的一致性语义和文件锁

不同文件系统所采用的文件一致性语义有所不同。UNIX文件系统采用如下一致性语义：

- 一个进程对一个打开文件的写入，能够立即被共享这一打开文件的其它进程观察到；
- 当多个进程共享同一系统打开文件表项时，它们共享文件当前偏移量，因此一个进程对文件偏移量的移动能够影响所有其它共享的进程。

这种情况下，文件只有一个镜像，所有共享的进程对它互斥交叠的存取。

Andrew文件系统采用下面的一致性语义：

- 一个进程对一个打开文件的写入，不能立即被共享这一打开文件的其它进程观察到；
- 仅当一个文件上的会话结束时，该文件上的变更对之后开启的会话是可见的。在一个会话结束前打开的文件实例观察不到这些 变更。

根据该语义，一个文件可以同时与多个文件镜像（这些镜像可能相同，也可能不同）相关联。因此，允许多个进程（用户）在各自的文件镜像上进行并发读写操作，几乎不需要施加任何并发控制。

文件锁是一种并发控制机制，使用它可以保证多个进程对共享文件的互斥访问。为了提高并发度，文件锁分为“共享锁”（Shared lock）和“排他锁”（Exclusive lock）。

封锁矩阵

	Share	Exlusive
Share	√	X
Exlusive	X	X

操作系统还提供**强制锁**（Mandatory locking）和**咨询锁**（Advisory locking）文件封锁机制。

- 如果采用强制封锁机制，那么当一个进程获得一个排他锁，操作系统将阻止任何其它进程企图对封锁文件的读、写访问请求，无论这些进程有没有使用锁机制。
- 咨询锁机制要比强制锁机制放松一些，当一个进程获得一个排他锁，操作系统只能阻止那些使用了封锁请求的进程，而对那些没有使用锁机制而且企图对封锁文件进行读、写访问请求的进程不会起到排斥作用。

6.6.4管道

管道、消息队列、共享内存和套接字是进程/计算机间的四种通信方式。管道实际上是一个能够被两个进程共享的打开文件，一个进程向其中写入，另一个进程从其中读出，这样就完成了一次通信过程。——但需要注意并发控制。

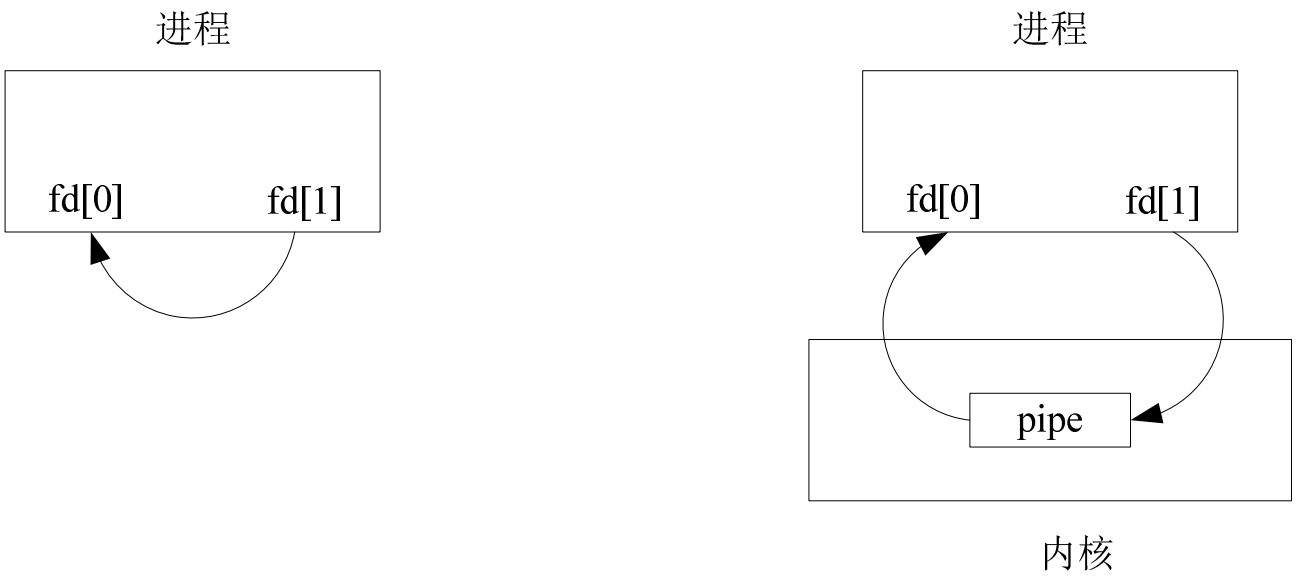
管道是UNIX系统中最早采用的一种进程间通信机制，在所有版本中得到了支持。但是管道的使用具有如下限制：

- 管道是半双工的（Half duplex），即管道的通信方向是单向的。要进行进程间的双向通信，必须建立两个管道；
- 管道只能用于父、子进程间通信，或者具有同一祖先的进程间通信。一般的，父进程先创建一个管道，然后通过fork( )创建一个子进程，这样父子进程就共享该管道。

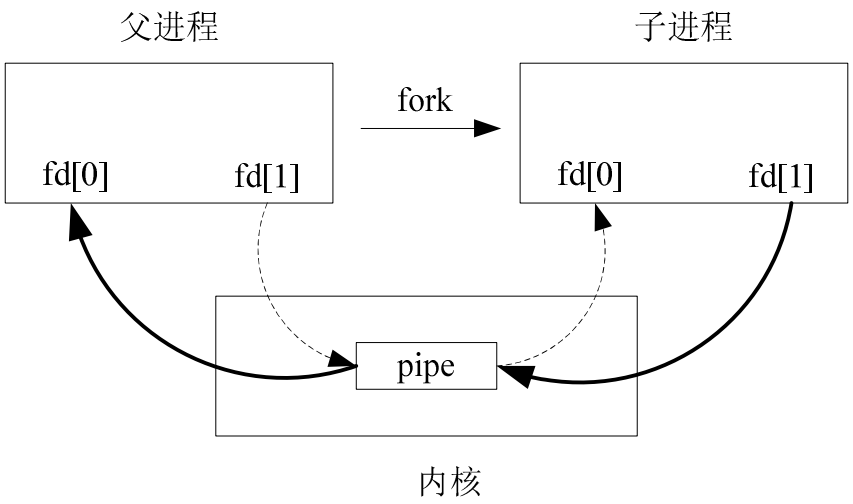
通过pipe系统调用创建一个管道：

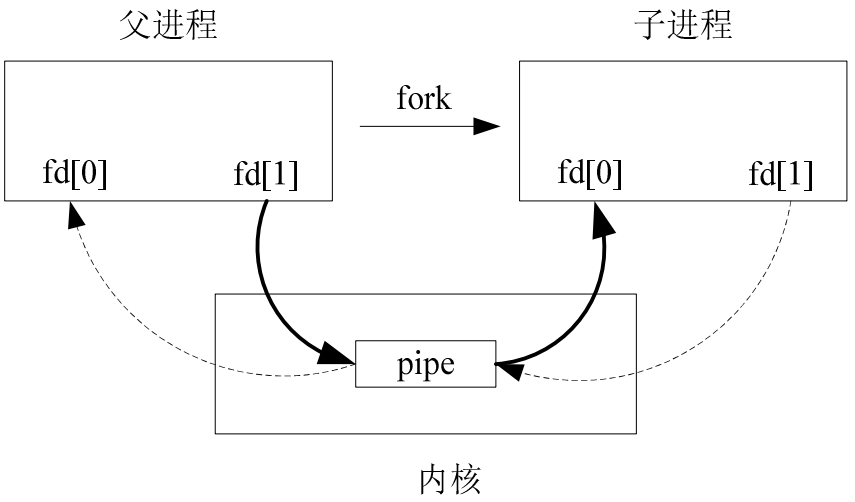
该系统调用创建两个文件描述字：fd[0]是读端口，fd[1]是写端口，fd[1]的输出是fd[0]的输入，

```
#include <unistd.h>
int pipe (int fd[2]);
                                    Returns: 0 if OK, -1 on error
```



通常的做法是，一个进程创建一个管道后，接着调用fork( )，这样管道就被子进程继承下来，





管道读写的并发问题

（1）与普通文件的读写操作不同，管道的读、写操作需要同步，也就是可能存在阻塞调用进程的问题。具体的：

- 当管道中没有数据可读时，read调用将会阻塞，即调用进程进入阻塞状态，直到有数据到来为止；
- 当管道满的时候，write调用将会阻塞，直到有进程读出数据为止。

（2）另外，当对管道进行读、写操作时，还可能出现对端端口已经关闭的情况。如果对这种情况不加处理，有可能导致进程永远阻塞下去。对于这种情况，UNIX是这样约定的：

- 如果管道的读端口被关闭，则write调用会抛出一个异常信号，并立即返回-1值，不等待read操作；
- 如果管道的写端口被关闭，那么管道中剩余的数据都被读出后，再次执行read时，会返回0值，并不等待write操作。

（3）一般情况下，可能有多个进程打开管道文件，同时向管道写入，那么会不会出现多个写操作交叠（即一个进程正在执行写操作时，被另一个进程的写操作所中断）的情况呢？对此，UNIX可以保证：当要写入的数据量不大于管道的容量时，系统将保证写入操作的原子性；当要写入的数据量大于管道容量时，系统将不再保证写入操作的原子性。

```
1        int main(void){
2          int n;
3          int fd[2];
4          pid_t pid;
5           char line[MAXLINE];
6           if (pipe(fd)<0)          //创建一个管道
7             err_sys("pipe error");
8           if ((pid=fork()) <0){     //创建子进程
9             err_sys("fork error");
10          }
11          else if(pid>0){          //父进程
12             close(fd[0]);        //关闭读端口
13             write(fd[1], "hello world\n", 12);
14          }
15          else{                    //子进程
16             close(fd[1]);       //关闭写端口
17             n=read(fd[0], line, MAXLINE);
18             write(STDOUT_FILENO, line, n);
19          }
20          exit(0);                 //两个进程都退出，释放管道资源
21       }
22
```

§6.7 文件系统的保护

首先，我们需要区分两个重要概念：

- **保护**（Protection）：对进程（或用户）访问计算机资源加以控制的一种方式。保护必须提供两方面内容：一是能够对施加的访问控制加以说明，二是能够对访问控制加以强制执行。
- **安全**（Security）。安全概念的外延要比保护广泛得多，它是指对计算机系统及其数据保持完整性的信任程度的一种度量。

文件是系统资源之一，其访问应当受到控制和保护。我们把一个受保护的对象（如文件）称为一个“客体”（Object）。一个客体包括数据或状态以及一个良定义的操作集。我们把能够访问客体的实体称为“主体”（Subject）（如进程或登录用户等）。主体只能通过操作集中的操作来访问客体的数据或状态。

进行安全保护的措施之一是定义主体对客体的访问权。所谓访问权（Access right）是指一个主体能够执行一个客体上的某些操作的能力。访问权通常用一个三元组来描述：

<Subject, Object, rights-set>

其中Subject是主体，Object是客体，rights-set是主体能够访问客体的操作集合。

一个主体可能对多个客体具有访问权，这些访问权的集合就构成了主体的一个**保护域**（Domain）。保护域是与一个主体相关联的所有访问权的集合。主体只能拥有保护域中所约定的权限，除此之外再无其它任何权限。

例如，名为Alice的用户，对名为F1的文件具有读、写和执行的权限，对文件F2具有只读权，那么Alice的访问权可以写作<F1, {r,w,e}>和<F2, {r}>，与Alice相关联的保护域为

D={<F1, {r,w,e}>, <F2,{r}>}

访问控制矩阵

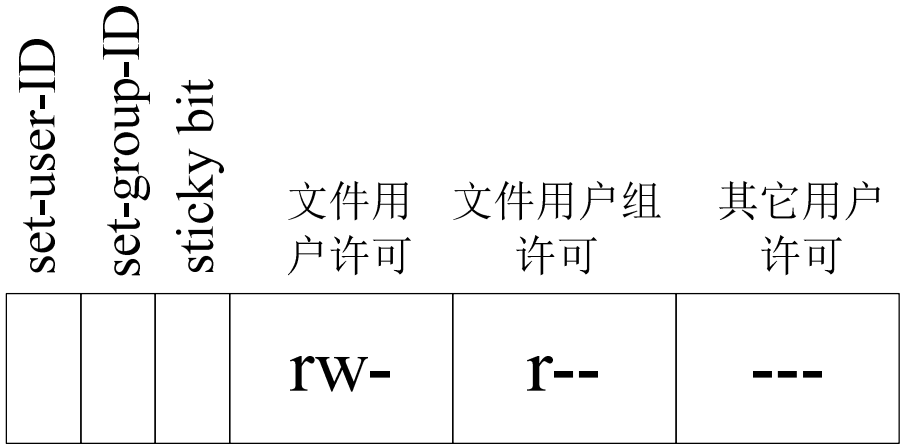
	SQRT	TEST	AAA	BAS
XU	RE	RWE	RW	R
LU	RW	E	None	E
GU	E	None	R	None

6.7.2 UNIX文件系统的访问控制机制

UID——用户ID

GID——用户组ID。一个用户只能有一个基本组

sGID——辅助用户组ID。一个用户可以有多达16个辅助用户组



	许可位	普通文件	目录
文件用户	user-read	允许文件用户读取文件	允许文件用户读目录项，即读取目录下的所有文件和子目录的元信息。注意：这不等于能够读取目录下的文件内容。
	user-write	允许文件用户写入文件	允许文件用户在目录中删除和创建文件或子目录
	user-execute	允许文件用户执行文件	允许文件用户在目录中搜索给定的路径名
文件用户组	group-read	允许组内其它用户读取文件	允许组内其它用户读取目录项
	group-write	允许组内其它用户写入文件	允许组内其它用户在目录中删除和创建文件或子目录
	group-execute	允许组内其它用户执行文件	允许组内其它用户在目录中搜索给定的路径名
其它用户	other-read	允许其它用户读取文件	允许其它用户读取目录项
	other-write	允许其它用户写入文件	允许其它用户在目录中删除和创建文件或子目录
	other-execute	允许其它用户执行文件	允许其它用户在目录中搜索给定的

real user ID real group ID	标识进程的实际用户（组）ID
effective user ID effective group ID supplementary group IDs	有效用户（组）ID，用于文件访问许可检查的用户（组）ID

文件访问控制测试步骤如下：

•如果*p*.effective user ID=0（即*p*是超级用户），那么对文件*f* 的访问被允许，即超级用户对于整个文件系统拥有所有访问权。

•如果*p*.effective user ID=*f*.user ID（即*p*是*f* 的拥有者），那么*p*要想拥有特定的许可，当且仅当文件的对应许可位被设置。

•如果*p*.effective group ID=*f*.group ID或存在*p*的一个附加组=*f*.group ID，那么*p*要拥有特定的许可，当且仅当文件的对应许可位被设置。

•如果*p*的有效用户ID不是文件*f* 的拥有者而且有效用户组也不等于文件的用户组，那么它要想拥有特定的许可，就必须考察*f*的other用户许可位。

•在进行访问控制测试时，以上四个步骤按顺序依次测试，前一个测试成功之后，不用再进行后面的测试了。

6.8.1 文件读、写的系统调用

（1）打开或创建一个文件

```
#include <fcntl.h>
int open(const char* pathname, int oflag, … /*mode_t  mode*/);
                                    Returns: file descriptor if OK, -1 on error
```

后效

1. 如果成功的打开一个已存在的文件，则为该文件创建进程打开文件表项和系统打开文件表项。如果是第一次打开这个文件，还需要为它创建一个v-node表项；如果其它进程先前已经打开了该文件，则共享该文件的v-node表项，并将v-node表项中的i-count递增。建立进程打开文件表项、系统打开文件表项和v-node之间的关联。
2. 如果成功的打开了一个文件，则根据oflag选项，设置系统打开文件表项的file status flag，如read、write、append，sync、nonblocking等。
3. 如果用open成功的创建一个文件，则在目录中创建该文件的对应目录项、i-node，设置文件允许位为mode，设置文件的user id和group id，并为文件分配存储空间。

（3）关闭一个文件

```
#include <unistd.h>
int close(int filedes);
                                    Returns: 0 if OK, -1 on error
```

关闭一个文件也随之释放调用进程施加在该文件上的所有记录锁。当一个进程终止时，它的所有打开文件被内核自动关闭。许多程序就利用这一点，并不显式的关闭打开文件。

（4）移动当前文件偏移量

```
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int whence);
                                    Returns: new file offset if OK, -1 on error
```

参数offset的解释依赖于参数whence。

- 当whence=SEEK_SET时，文件的偏移量被设置为offset；
- 当whence=SEEK_CUR时，文件的偏移量被设置为“当前值+offset”，offset可以是正值或负值；
- 当whence=SEEK_END时，文件的偏移量被设置为“文件大小+offset”，offset可以是正值或负值。

(5) 读写文件内容

```
#include <unistd.h>
ssize_t read(int filedes, void *buf, size_t nbytes);
                        Returns: number of bytes read, 0 if end of file, -1 on error
```

```
#include  <unistd.h>
ssize_t write(int filedes, const void *buf, size_t nbytes);
                        Returns: number of bytes written if OK, -1 on error
```

（2）改变文件的访问许可位

```
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
                                        Returns: 0 if OK, -1 on error
```

前置条件：调用进程的effective user ID必须等于文件用户的user ID或者调用进程拥有超级用户权限。

比如，文件bar当前访问允许位为“rw-------”，即只有文件拥有者才能读写。现在要将其访问允许位改为“rw-r--r--”，可以这样调用：

chmod(“bar”, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

（3）改变文件的所属

```
#include <sys/stat.h>

int chown(const char *pathname, uid_t owner, gid_t group);

                                        Returns: 0 if OK, -1 on error
```

（1）创建文件硬链接

```
#include <unistd.h>
int link (const char *existingpath, const char *newpath);
                                        Returns: 0 if OK, -1 on error
```

```
#include <unistd.h>

int unlink (const char *pathname);

                                        Returns: 0 if OK, -1 on error
```

（4）创建符号链接

```
#include <unistd.h>
int symlink (const char *actualpath, const char *sympath);
                                    Returns: 0 if OK, -1 if error
```

由于符号链接本身就是一个文件，而它又可以用来引用原文件，因此当使用符号链接名进行系统调用时，必须清楚的知道该系统调用是作用在符号链接文件上，还是作用在它所引用的原文件上。

chmod(“bar”, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

lchown (“bar”, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

- 作业

- P240　2、8、11、13、14