第五章 文件系统的系统调用

本章主要介绍针对上层用户使用的系统调用。用户通过使用本章介绍的系统调用来申请操作系统中有关文件和文件系统操作的各项功能。

本章介绍的算法是基于第四章所介绍的底层文件系统算法之上的，主要包括七大类操作：

返回文件描述符类操作

路径名转换类操作

分配索引节点类操作

文件属性类操作

文件输入输出类操作

文件系统装卸类操作

文件系统目录树操作

系 统 调 用						
返回文件描述符	使用namei	分配索引节点	文件属性	文件I/O	文件系统结构	目录树操作
open dup	open stat creat	creat	chown	read	mount	chdir
creat pipe	link chdir unlink	mknod	chmod	write	umount	chroot
close	chroot mknod chown	link	stat	stat		
	mount chmod umount	unlink				

底层文件系统算法		
namei	ialloc ifree	alloc free bmap
iget iput		
缓冲区分配算法		
getblk brelse bread breada bwrite		

1、算法 open

输入：文件名

打开文件类型

文件许可权方式（对以创建方式打开而言）

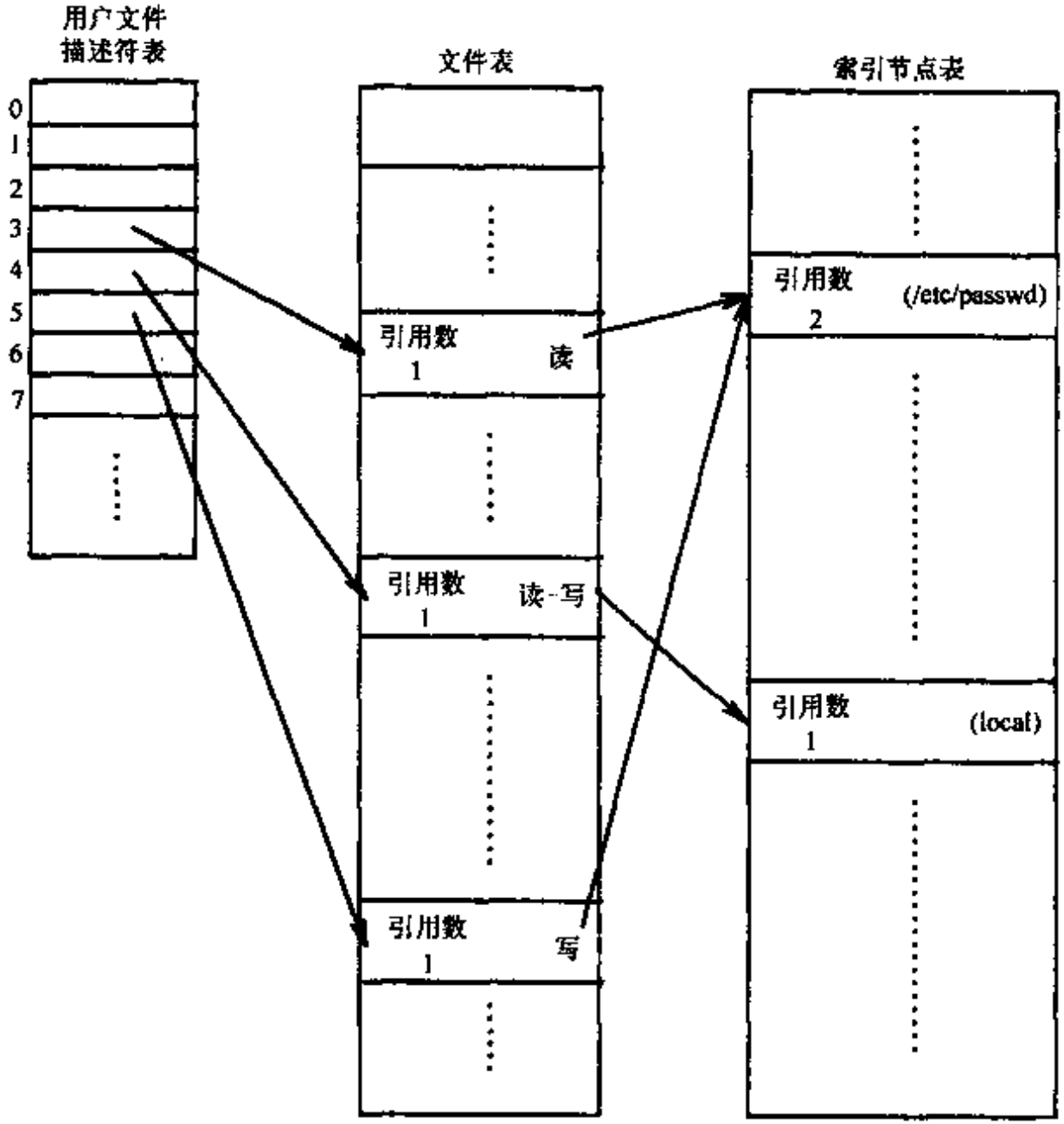
输出：文件描述符

```
{
    将文件名转换为索引节点（算法namei）；
    if (文件不存在或不允许存取)
        return (错)；
    为索引节点分配系统打开文件表项，设置引用计数和偏移量；
    分配用户文件描述符表项，将指针指向系统打开文件表项；
    if （打开的类型规定清除文件）
        释放占用的所有文件系统块（算法free）；
    解锁（索引节点）；    /* 在上面的namei算法中上了锁 */
    return（用户文件描述符）；
}
```

假定一个进程执行下列代码：

```
fd1=open(“/etc/passwd”,O_RDONLY);
fd2=open(“local”,O_WRONLY);
fd3=open(“/etc/passwd”,O_RDWR);
```

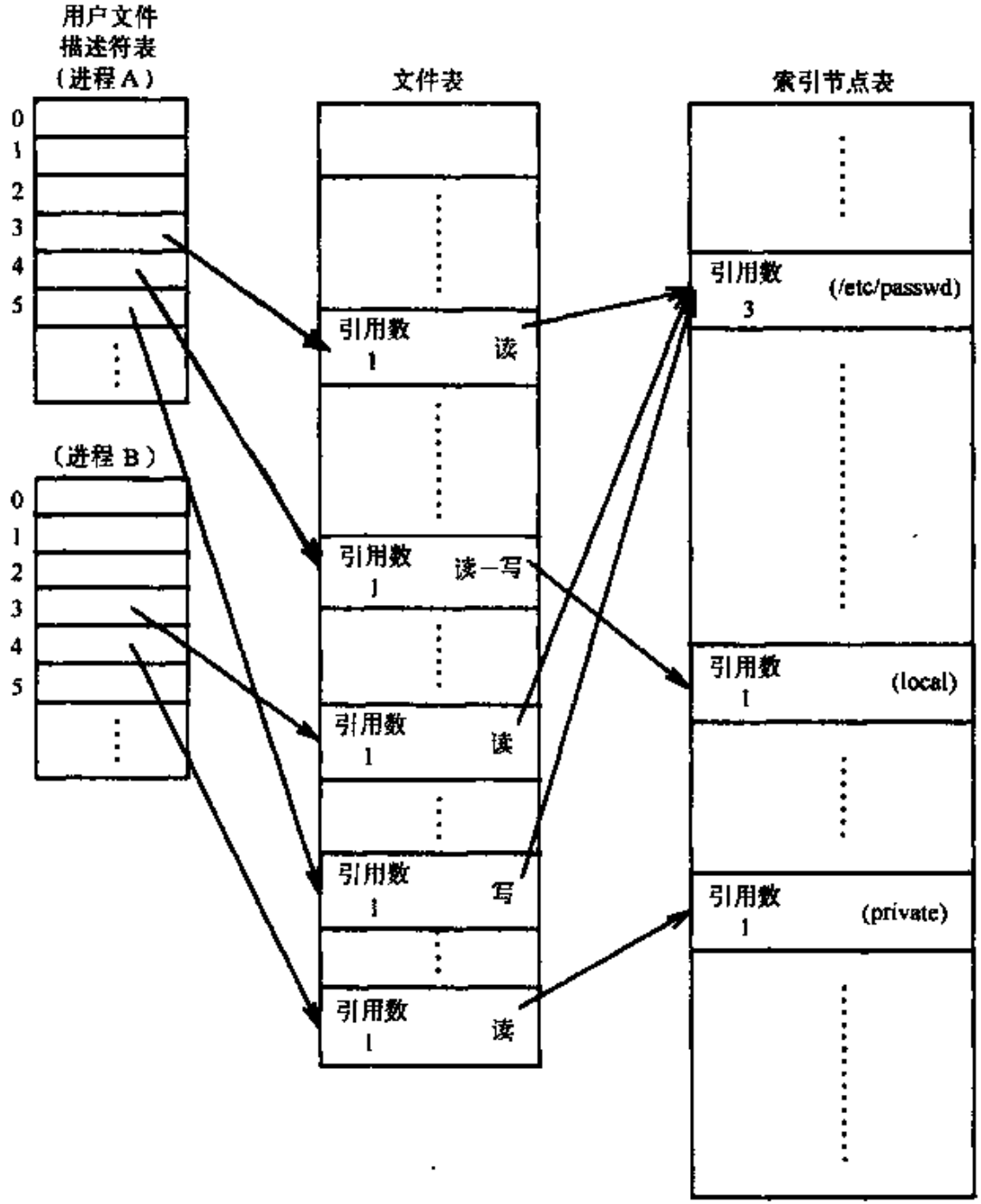
下图为打开文件后的数据结构：


用户文件
描述符表
文件表
索引节点表
0
1
2
3
4
5
6
7
引用数
1
读
引用数
1
读-写
引用数
1
写
引用数
2
(/etc/passwd)
引用数
1
(local)

假定第二个进程执行下列代码：

```
fd1 = open(“/etc/passwd”, O_RDONLY);
fd2 = open(“private”, O_RDONLY);
```

则相关的数据结构如下：


用户文件
描述符表
（进程 A）
0
1
2
3
4
5
（进程 B）
0
1
2
3
4
5
文件表
引用数
1
读
引用数
1
读—写
引用数
1
读
引用数
1
写
引用数
1
读
索引节点表
引用数
3
(/etc/passwd)
引用数
1
(local)
引用数
1
(private)

2、算法 read

系统调用read的语法格式如下：

```
number = read (fd, buffer, count);
```

fd 是由open返回的文件描述符
buffer 是用户进程中的用于保存数据的缓冲区地址
count 是用户要读的字节数
number 是实际读出的字节数

在u区中保存的I/O参数为

方式	指示读或写
计数	读或写的字节数
偏移量	文件中的字节偏移量
地址	拷贝数据的目的地址，在用户或核心存储器中
标志	指出地址是在用户空间还是核心空间

算法 read

输入：用户文件描述符

用户进程中的缓冲区地址

要读的字节数

输出：拷贝到用户区的字节数

{

由用户文件描述符得到系统打开文件表项（file表项）；

检查文件的可存取性；

在u区中设置用户地址、字节计数、输入/输出到用户的参数；

从file表项找到索引节点；

索引节点上锁；

用file表项中的偏移量设置u区中的字节偏移量；

（接下页）

```
while (要读的字节数还不满足)
{
    将文件偏移量转换为文件系统块号（算法bmap）;
    计算块中的偏移量和要读的字节数;
    if (要读的字节数为0)                    /*  企图读文件尾 */
        break;                              /*  退出循环  */
    读文件块(如果要预读，用breada，否则用bread);
    将数据从系统缓冲区拷贝到用户地址;
    修改u区中的字节偏移量、读计数、再写的用户空间地址;
    释放缓冲区;                             /*  在bread中上了锁  */
}
解锁索引节点;
修改file表中的偏移量，用作下次读操作;
return(已读的总字节数);
}
```

读文件程序实例1

```
#include  <fcntl.h>
main( )
{
    int  fd;
    char lilbuf[20], bigbuf[1024];

    fd = open( “/etc/passwd”, O_RDONLY);
    read(fd, lilbuf, 20);
    read(fd, bigbuf, 1024);
    read(fd, lilbuf, 20);
}
```

```
#include  <fcntl.h>
main( )     /*   进程A  */
{
    int  fd;
    char  buf[512];
    fd = open(“/etc/passwd”, O_RDONLY);
    read(fd, buf, sizeof(buf));              /*  read1  */
    read(fd, buf, sizeof(buf));              /*  read2  */
}

main( )      /*   进程B  */
{
    int  fd, I;
    char  buf[512];
    for(i=0;  i<sizeof(buf);  i++)
          buf[i] = ‘a’;
    fd = open(“/etc/passwd”, O_WRONLY);
    write(fd,  buf,  sizeof(buf));          /*  write1  */
    write(fd,  buf,  sizeof(buf));          /*  write2  */
}
```

3、文件I/O位置调整 lseek

position = lseek (fd, offset, reference);

其中：fd —— 文件描述符
offset —— 字节偏移量
reference —— 偏移参照点：0：从文件头开始
1：从当前位置开始
2：从文件尾开始

下图为lseek应用实例：

```
#include  <fcntl.h>
main(int argc, char *argv[])
{
    int  fd, skval;
    char  c;
    if(argc != 2)
        exit();
    fd = open(argv[1], O_RDONLY);
    if(fd == -1)
        exit();
    while((skval = read((fd, &c, 1)) == 1)
    {
        printf( “char %c\n”, c);
        skval = lseek(fd, 1023L, 1);
        printf( “new seek val %d\n”, skval);
    }
}
```

4、读取索引节点状态参数 stat/fstat

系统调用stat和fstat永续进程查询文件的状态，它们返回诸如文件类型、文件所有者、存取权限、文件大小、链接数目、索引节点号、文件的访问时间等信息：

```
stat(pathname, statbuffer);
fstat(fd, statbuffer);
```

其中pathname是文件名；fd是文件描述符；statbuffer是用户进程中的一个类别为stat的数据结构，在系统调用完成时用于存放返回的文件状态信息。

数据结构 **stat** 的定义：

```
struct   stat   {
        dev_t     st_dev;        文件所在的设备号
        ino_t     st_ino;        文件的I节点号
        ushort    st_mode;       读写保护模式
        short     st_nlink;      文件的链接数
        short     st_uid;        用户标识
        short     st_gid;        组标识
        dev_t     st_rdev;       文件的起始位置
        off_t     st_size;       文件大小
        time_t    st_atime;      最近访问时间
        time_t    st_mtime;      最近修改时间
        time_t    st_ctime;      最近（状态）改变时间
        long      st_blksize;    文件块大小
        long      st_blocks;     文件所用块数
}
```

实例：

打印一个文件的i节点号、文件名和文件大小（类似于ls命令的功能）：

```
#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/stat.h>
main(int argc,   char *argv[ ])
{
        struct  stat  mystat;
        stat(argv[1],  &mystat);
        printf("%d   %s   %d\n",  mystat.st_ino,  *argv[1],
                mystat.st_size);
}
```

5、建立无名管道　pipe

pipe(fdptr)

fdptr[0]　　读管道指针

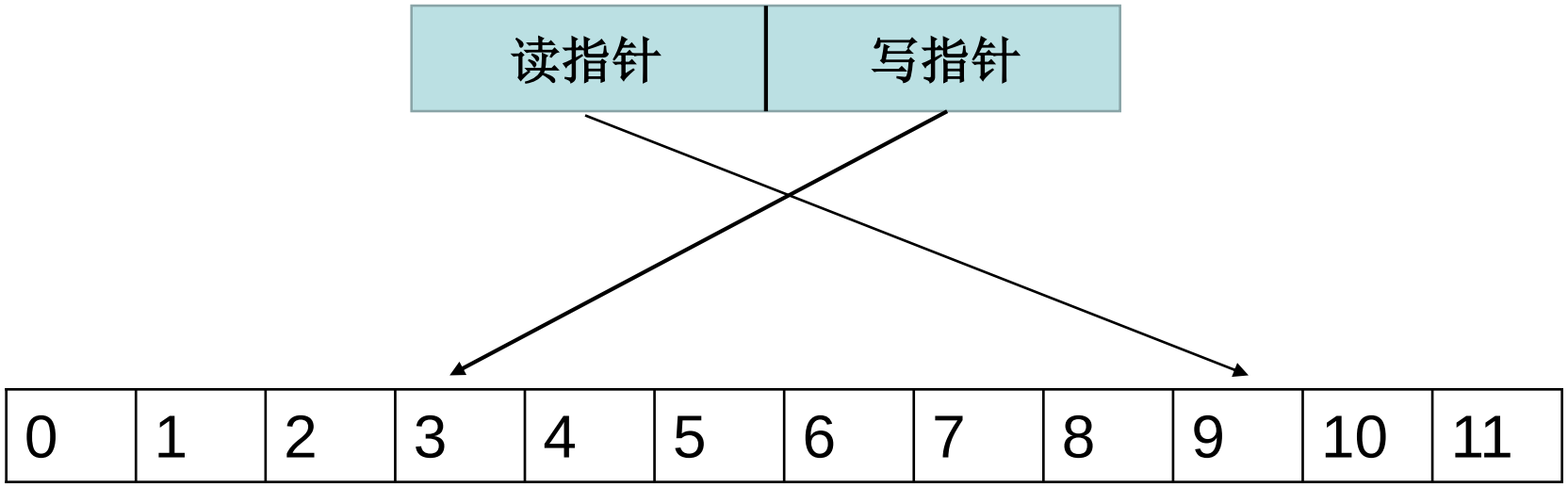
fdptr[1]　　写管道指针

无名管道只能是建立管道的进程的子进程，才能共享无名管道。

mknod("pipe_name", rw_mode)　　建立有名管道

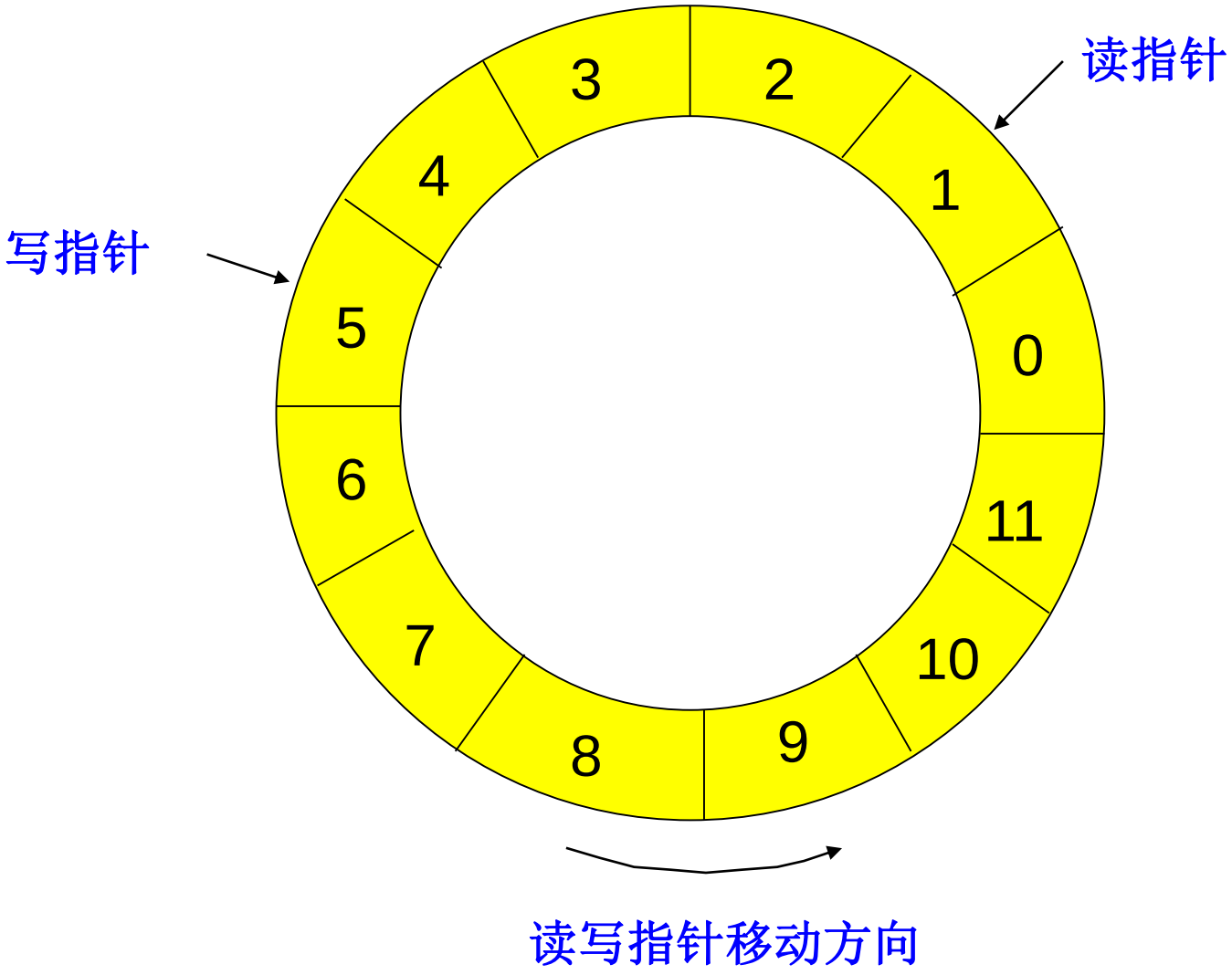
使用有名管道的进程间可以没有任何父子关系。

管道的结构和读写指针（1）



索引节点的直接索引块

管道的结构和读写指针（2）



管道读写的四种典型状况：

（1）、写管道，管道中有空闲的存储空间满足写操作；

（2）、读管道，管道中有足够的数据满足读操作；

（3）、读管道，管道中没有足够的数据满足读操作；

（4）、写管道，管道中没有足够的空闲空间存放数据。

（1）、写管道，管道中有空闲的存储空间满足写操作

进程向管道中写数据。当写指针指向到一个间接块时（即超出数据索引表中的直接块索引表的大小），则写指针被重新置为0。因为核心能够确定，只要写指针没有超过读指针，则数据就不会超出管道的容量。

写操作完成后，核心唤醒所有等待从该管道中读数据的进程。

（2）、读管道，管道中有足够的数据满足读操作

进程从管道中读取数据。读操作完成后，核心要唤醒所有等待着向管道中写数据的进程。同时，把读指针存放到活动inode表中。为什么不像普通进程那样把读写指针放到系统打开文件表file表中？

因为写进程必须要知道读进程读到管道的哪里了

读进程必须要知道写进程写到管道的哪里了

（3）、读管道，管道中没有足够的数据满足读操作

进程读取管道中的全部数据，并成功返回，虽未完全满足读取数量。

如管道为空，通常进程进入睡眠状态，等待“管道中有数据可读”这个时间发生时被唤醒，并与可能存在的其他读进程竞争。

（4）、写管道，管道中没有足够的空间满足写操作

进程写数据到管道中，直到管道被写满为止，并进入睡眠状态，等待其他的读进程读取数据后，使管道中腾出空间。

当另一个读进程从管道中读取数据后，会唤醒包括本进程在内的所有写进程，本进程再次有机会把未写完的数据写入管道。

6、复制用户文件描述符 dup

dup把一个文件描述符复制到本用户的打开文件表中从头的第一个可用的空表项中。新的表项（文件描述符）由dup返回。

由于文件描述符被复制，因此两个文件描述符指向同一个系统打开文件表项，即指向同一个文件读写指针，该读写指针的引用计数加一。

dup的应用举例：标准输出重定向

command > abc

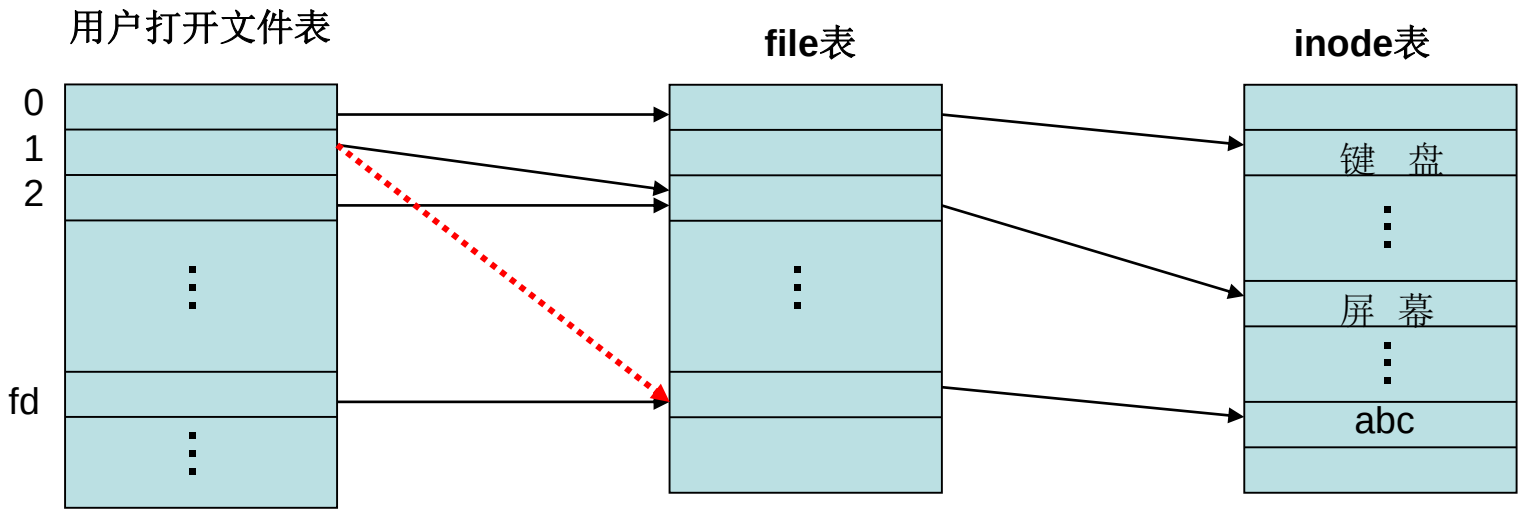
①、fd = open("abc", "w")

②、close(1)

③、dup(fd)

④、close(fd)

⑤、command



7、安装文件系统　mount

只有超级用户可以安装和拆卸文件系统

安装点目录通常为空目录

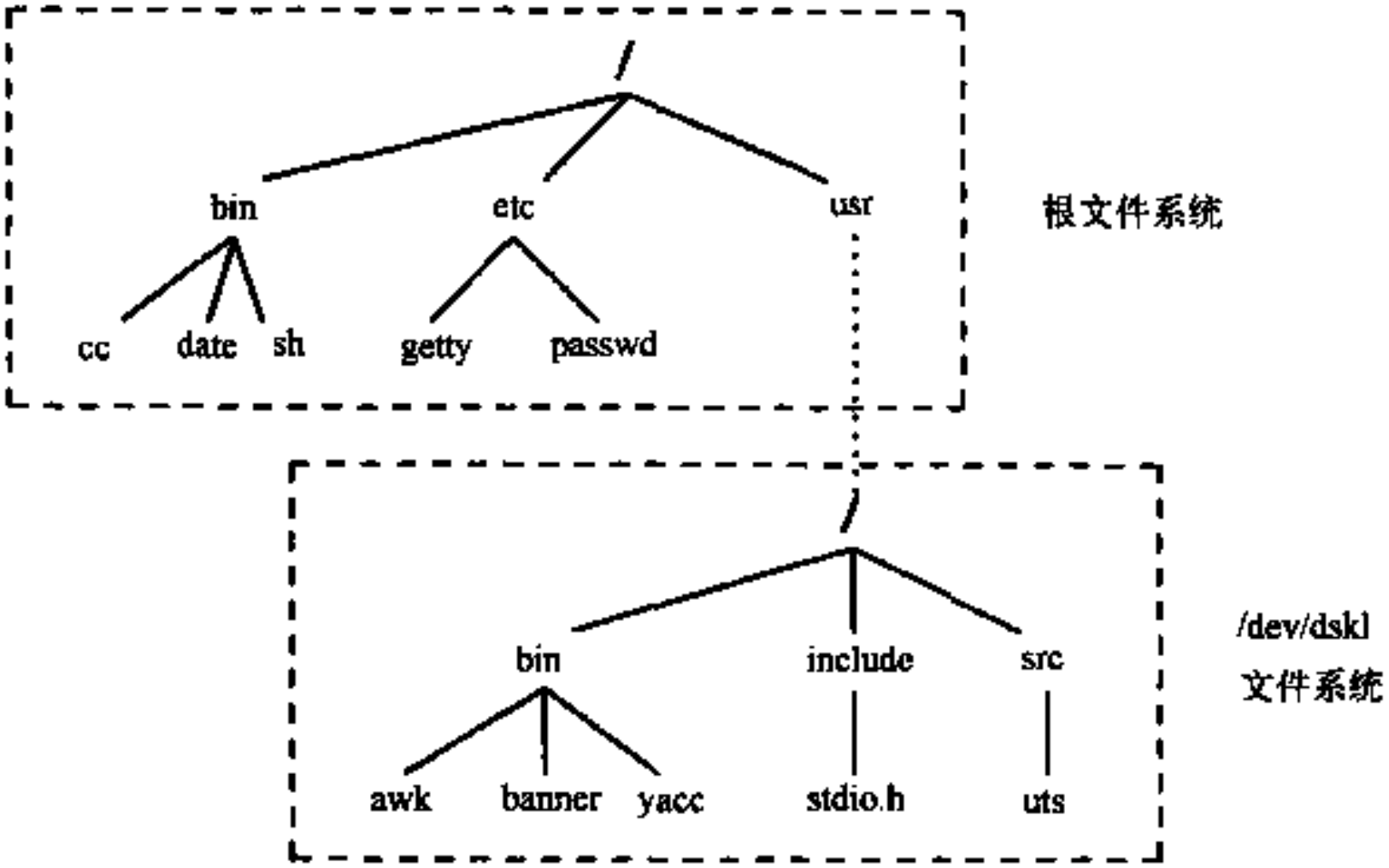
安装点的引用计数只能为一

在**mount**表中占用一个表项，保持了安装点和被安装文件系统根节点之间的对应关系

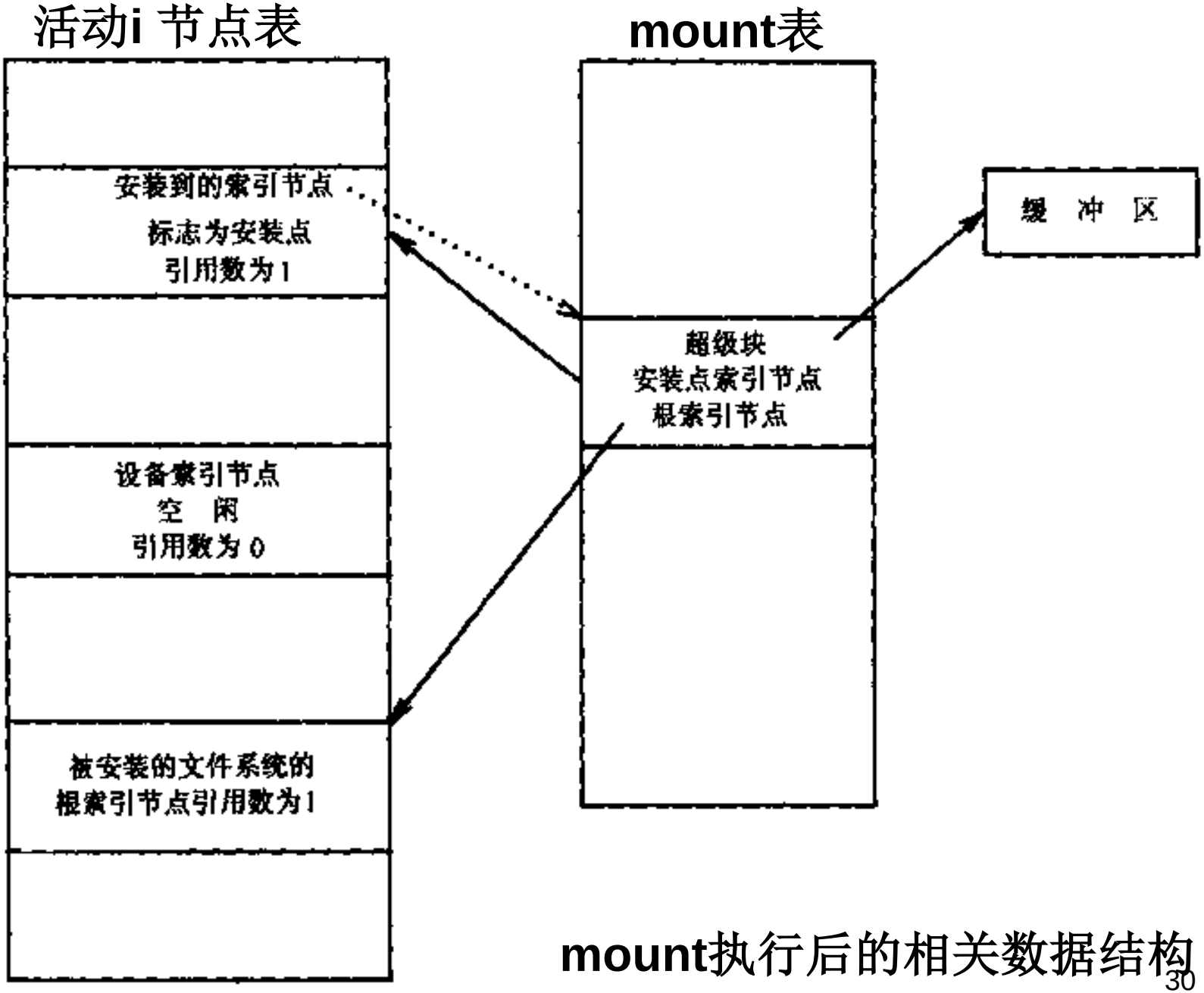
读入被安装文件系统的超级块，并保持指向超级块的指针

例如：　**mount("/dev/dsk1", "/usr", 0)**

把逻辑设备（子文件系统）**/dev/dsk1**以可读可写的方式安装到目录**/usr**下面。



把子文件系统 **/dev/dsk1** 安装到 **/usr**目录下



mount执行后的相关数据结构

```
算法   mount
输入：块特殊文件的文件名
      安装点的目录名
      选择项（只读）
输出：无
{
        if （ 非超级用户 ）
                return（错）；
        取块特殊文件的索引节点（算法namei）；
        合法性检查；
        取安装点目录名的索引节点（算法namei）；
        if（不是目录，或引用数大于1）
        {
                释放索引节点（算法iput）；
                return（错）；
        }
```


（ 接下页 ）

（接上页）

查找安装表中的空项；
调用块设备驱动程序的open子程序；
从高速缓冲中去空闲缓冲区；
将超级块读入空闲缓冲区；
初始化超级块中的各个域；
取被安装设备的根索引节点，并保存在安装表中；
标记安装点的目录索引节点为安装点；
释放特殊文件的索引节点（算法iput）；
解锁安装点目录的索引节点；

}

进程在存取一个目录节点时，可能遇到该目录是一个安装点，这时就要从安装点所在文件系统跨越到被安装的子文件系统中，这种跨越是在检查到该目录节点上有“安装点”标志时进行。

包含这种跨越动作的常用系统调用包含：

分配内存活动i节点算法 **iget**

把路径名转换为索引节点算法 **namei**

8、删除一个目录项 unlink

unlink(pathname);

该系统调用删除一个名为pathname的目录项。如果该目录项是该文件的最后一个链接，则核心删除该文件的索引节点，并释放文件的数据块。如果该文件有多个链接，则其他文件名仍能正常存取该文件。

下面为unlink的算法：

```
算法  unlink
输入：文件名
输出：无
{
    取要删除的文件的父目录的索引节点（算法namei）；
    if （文件名的最后分量是“.”）
        父目录索引节点的引用计数加1；
    else
        去要被删除的文件的索引节点（算法iget）；
    if （要删除的是目录的链接项，但用户不是超级用户）
    {
        释放索引节点（算法iput）;
        return（错）；
    }
    if （要删除的是共享正文文件，且链接数为1）
        从区表中清除；
    写父目录：将被删除文件的索引节点号置为0；
    释放父目录的索引节点（算法iput）；
    文件链接数减1；
    释放文件的索引节点（算法iput）;
        /* iput检查链接数是否为0；如果是，
         * 则释放文件的数据块（算法free）
         * 并释放磁盘索引节点 icommon
         */
}
```

系统调用 unlink 的特殊应用

当一个进程使一个文件处于打开状态时，另一个进程可能在该文件打开期间删除该文件（甚至做这件事的进程本身就是发出系统调用open的进程）。

第一个进程除非关闭这个文件，否则就可一直读写该文件。而其他进程因为该文件已被删除而无法访问该文件。

—— 安全无痕地使用文件的方法！

9、文件系统管理　fsck

fsck 命令通常由具有超级用户权限的系统管理员执行, 用于检测和修复文件系统的错误. 运行时显示如下过程信息:

**** Phase 1 – Check Blocks and Sizes**

检查索引节点表中文件大小和所用块数

**** Phase 2 – Check Pathnames**

检查目录和文件路径的正确性

**** Phase 3 – Check Connectivity**

检查各目录之间的联结关系

**** Phase 4 – Check Reference Counts**

检查各文件的引用计数

**** Phase 5 – Check Free List**

检查文件系统的空闲块表