文件系统例题与习题 3、综合

同济大学计算机系操作系统　　初稿 2024-1-2，修订 2024/12/5　　邓蓉

姓名　　学号

一、完整的文件读写过程

1、

```
int fd = open("/usr/ast/Jerry",3);//以可读可写方式打开文件
char data[300];
seek(fd, 500, 0);//将文件读写指针定位到第500字节
int count = read (fd, data, 300);//从文件读300字节到data
count = write(fd, data, 300);//从 data 写 300 字节到文件
```

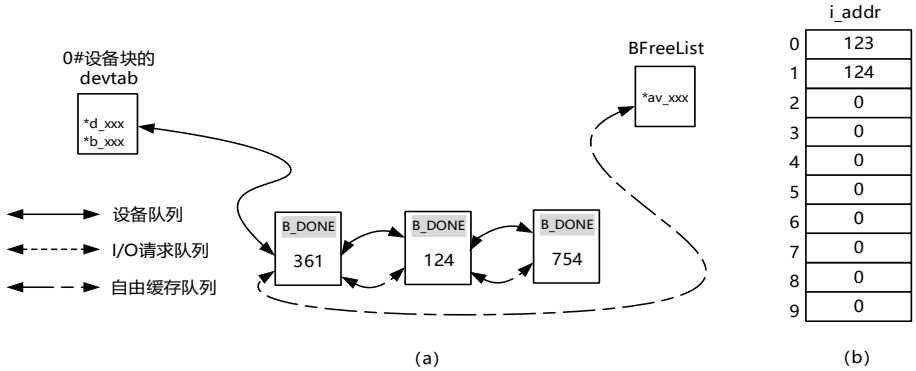


图 1、缓存队列 和 Jerry 文件的索引表

详见文档《文件系统例题与习题 2 完整的文件读写过程》

2、识别文件的顺序读写操作 和 随机读写操作。每个系统调用完成后，f_offset =_____?

二、文件系统的静态结构

1、Unix V6++系统，存放一个长 102400 字节的文件 file1，需要使用多少磁盘存储资源 ？

答：一个目录项，存放在父目录文件中。一个 DiskInode，存放在 Inode 区。

数据区：102400/512 = 200 扇区用来存放文件数据。d0 ~ d199

2 个一次间接索引块。i0 ， i1

共计 202 个扇区。

混合索引结构如下：

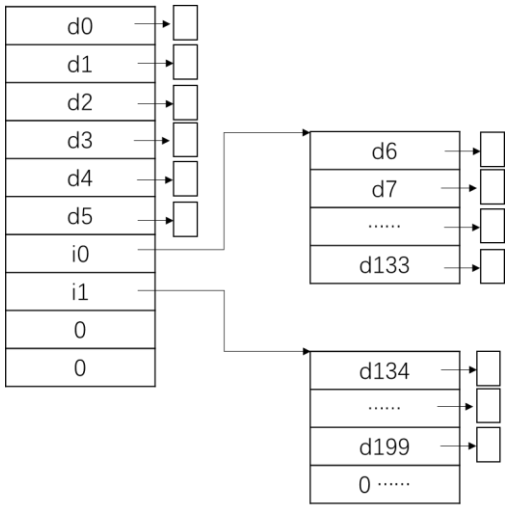
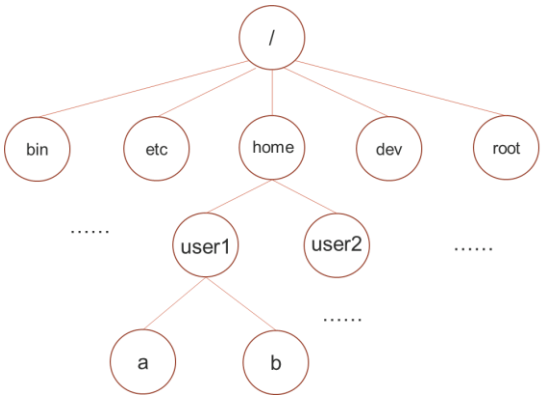


图 1

2、Unix V6++系统，如下目录树。已知：目录 /，bin，etc，home，dev，root，user1 和 user2 分别是 1#，2#，3#，4#，5#，6#，10#和 12#文件。请填空补全 / 和 home 目录文件，多余的目录项，所有字段填 0。



/目录文件

.	1
..	1
bin	2
etc	3
home	4
dev	5
root	6

home目录文件

.	4
..	1
user1	10
user2	12
0	0
0	0
0	0

图 2

3、Unix V6++系统，超级块 1024 字节，DiskInode 64 字节，每个扇区 512 字节。简述系统加载 100#DiskInode 的过程。

图 3

答：系统使用 100#文件前，需要将 100#DiskInode 加载进分配给它的内存 Inode。具体过程如下：

- 100 除以 8，商 12，余 4。100#inode 在 14#扇区，是这个扇区的 4#inode。
- bp=Bread(0,14)，将 14#扇区读入缓存块 bp。
- IOmove(bp->b_addr+256, &Inode[i], 64）// Inode[i]是分配给 100#DiskInode 的内存 Inode

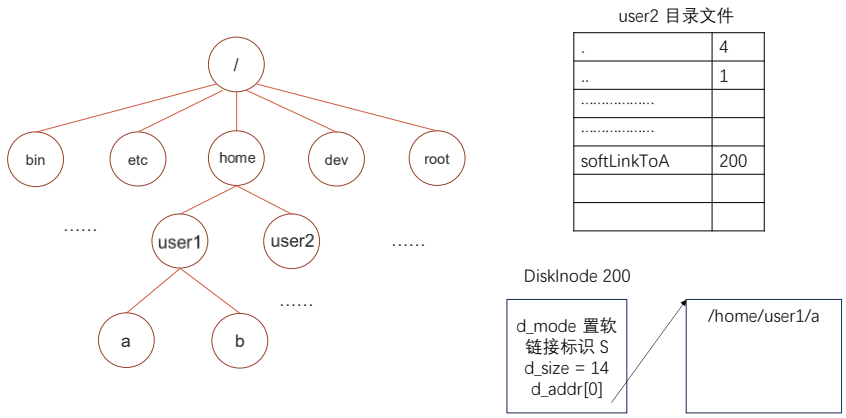
4、某磁盘剩余空间 30%，无法创建新文件。这是什么情况？怎样避免这种情况的发生。

答：inode 区用完了，磁盘存放了大量的小文件。

避免这种情况发生，最重要的是要明确文件卷的应用场合。如果存的都是小文件，格式化时，inode 区要給大点儿。全是大文件，inode 区小点儿。磁盘空间足够用，忽略。

5*、软链接

软链接是一个存有目标文件名的小文件。有自己的 DiskInode。分配有数据块。



user2 目录下创建一个引用 a 文件的软链接 softLinkToA。200#DiskInode 分配给这个软链接文件。

三、Unix 文件系统的使用

（一）打开文件结构

1、T0 时刻，系统中有两个进程 P1 和 P2，分别独立打开并同时访问小文件 example。则在内存打开文件结构中有（ ）个内存 Inode 指向该文件？（ ）个 File 结构记录着进程对文件的访问情况？

A. 1　　B. 2

在哪个数据结构中登记有进程对文件的访问方式（读或读写）？（ ）

文件的读写指针保存在（ ）？

组成文件的每个逻辑盘块（信息块）在磁盘上的地址保存在（ ）？

A. 内存 Inode　B. File 结构　C. i_addr 数组

若 P2 进程向文件追加写入 10000 个字符后关闭该文件，引发（ ）操作；稍后，P1 关闭 example 文件，引发（ ）操作。

A. 释放 file 结构　　B. 释放内存 i 节点

C. 将内存 i 节点写回磁盘　　D. 不执行任何操作

2、假设 foobar.txt 文件的内容是字符串“1234567890”。请问（1）这个程序的输出是什么？（2）画进程的打开文件结构

```
int main()
{
    int fd1, fd2;
    char c;

    fd1 = Open("foobar.txt", O_RDONLY, 0);
    fd2 = Open("foobar.txt", O_RDONLY, 0);
    Read(fd1, &c, 1);
    Read(fd2, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

3、fd1，fd2 的值是几？

```
int main()
{
    int fd1, fd2;

    fd1 = Open("foo.txt", O_RDONLY, 0);
    Close(fd1);
    fd2 = Open("baz.txt", O_RDONLY, 0);
    printf("fd2 = %d\n", fd2);
    exit(0);
}
```

4*、输入、输出重定向的实现

已知：库函数 printf(“格式化串”，输出的内容)的执行分 2 步（1）使用格式化串处理输出的内容，生成待显示的字符串 formatted_string（2）执行系统调用

write(1，formatted_string，sizeof(formatted_string)。

1，是进程的标准输出文件，默认是终端的输出缓存；即进程打开文件表 1#表项引用的 File 结构指向的是 tty 的内存 inode。

输出重定向是指修改进程的 1#文件描述符，让 printf 将生成的字符串写入一个指定的磁盘文件。例如：以下命令将 cat 程序的输出重定向至磁盘文件 outFile。

$ cat existFile > outFile

cat 命令向屏幕输出 existFile 文件内容。输出重定向后，cat 命令不再向屏幕输出，原本输出的内容写入 outFile 文件。

输出重定向怎么实现呢？以下是一种可行的方法：shell 进程创建一个子进程，子进程 exec(cat)之前，连续执行系统调用 close 和 open：

```
if ( fork( )==0 )
{
    close(1);
    fd = open(outFile，写);      // fd 是 1
    exec(cat, ****); // cat 程序执行时标准输出（1#文件描述符）是 outFile 文件
}
```

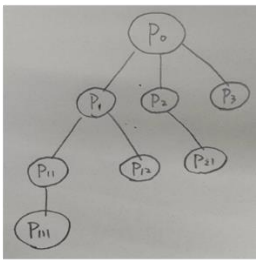
。。。shell 程序中父进程的分支。。。

5*、fork 套在 for 循环中，输出重定向之前，输出是 8 行，每个进程输出一行。输出重定向之后，

$ forkDemo > outFile

outFile 文件中有 20 行输出，为什么会是这样呢?

```
L1:     #include <stdio.h>
L2:     void main(void)
L3:     {   int i;
L4:         printf ("%d  %d \n",  getpid ( ),  getppid ( ) );
L5:         for (i = 0; i < 3; ++i)
L6:             if ( fork( ) == 0 )
L7:                 printf ("%d  %d \n",  getpid ( ),  getppid ( ) );
L8:     }
```



[参考答案]　库函数 printf（其实是标准输入输出库 stdio）在用户空间为每个文件描述符维护一个用户缓存。缓冲的工作模式与文件描述符引用的文件类型(d_mode)相关。字符设备，也就是终端 tty，行缓冲，遇到回车输出、执行一次 write 系统调用写屏幕。普通磁盘文件，块缓冲，写满 4096 字节才会输出，执行 write 系统调用写磁盘文件。

输出重定向之前，printf 向终端屏幕输出，所以 1#文件描述符是行缓冲的。每个进程的输出有回车，故，创建子进程之前，父进程会将输出的字符行写入 tty 输出缓存并且清空库函数使用的用户缓存。也就是，子进程继承来的父进程图像中，没有父进程输出的字符行。。。每个进程输出一行，整个程序一共输出 8 行。

磁盘文件是块缓冲的，用户空间的这块缓存 4096 字节。本例（1）执行 fork 系统调用时，所有父进程的输出不足 4096 字节，留在用户缓存里（2）子进程继承父进程的用户缓存，所以会继承父进程没有写到屏幕的所有字符行（3）所有进程最终输出的数据量远远小于 4096 字节，所以任何进程终止前不会执行 write 系统调用（4）所有进程终止时执行一次 write 系统调用，将用户缓存中留存的字符行全部写入磁盘文件 output。综上，进程树上高度为 4 的节点输出 4 行，高度为 3、2、1 的节点分别输出 3 行、2 行和 1 行。合起来，程序一共输出 20 行。

（二）系统调用的语义、执行过程和例题。详见文档《Unix V6++的目录和与之有关的系统调用》

1、fd = open(name,mode);

2、fd = creat(name,mode);　　// creat 还是 create，无所谓

3、close(fd);

4、unlink(name);

5、link(name1,name2);

（三）系统调用执行时的 IO 次数

1、线性目录搜索文件 “/usr/ast/Jerry”。假设各级目录文件只有一个数据块，内容如图 7.32 所示。简述目录搜索过程，计算目录搜索需要执行的 IO 次数。

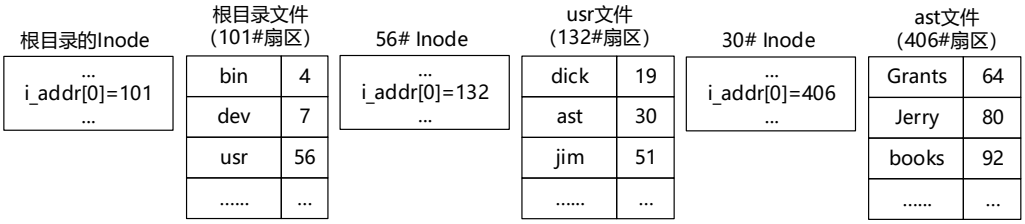


图 7.32：目录搜索示例

这是绝对路径名。搜索起点是根目录，1#文件。

目录搜索需要遍历目录文件，直至找到相关目录项。遍历过程分 2 步：（1）读入目录文件的 inode（2）逐个读入目录文件的数据块，遍历之。

本例，有 3 个路径名分量，目录搜索要遍历 3 个目录文件

（1）遍历根目录（1#文件），搜索文件名”usr”。成功，usr 文件的 inode 号是 56。

细节：根目录文件 Inode 常驻主存，无需读入，没有 IO。

遍历，需要读入 101#数据块，1 次 IO

（2）遍历 usr 目录（56#文件），搜索文件名”ast”。成功，ast 文件的 inode 号是 30。

细节：56# Inode 不在主存，需要读入，1 次 IO。

遍历，需要读入 132#数据块，1 次 IO

（3）遍历 ast 目录（30#文件），搜索文件名”Jerry”。成功，Jerry 文件的 inode 号是 80。

细节：30# Inode 不在主存，需要读入，1 次 IO。

遍历，需要读入 406#数据块，1 次 IO

目录搜索成功，返回 Jerry 文件的 inode 号 80。此次目录搜索，IO 5 次，将 5 个磁盘数据块读入磁盘高速缓存。

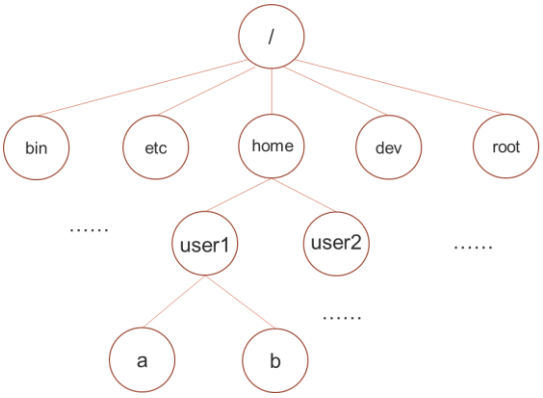
评论：线性目录搜索开销巨大。（1）好慢（2）淘汰掉磁盘高速缓存中的 5 个 LRU 数据块。一定要想办法优化。

优化的方法：（1）尽量用相对路径名引用文件，路径短，会快好多。看下面的例子。

（2）文件系统设置目录项缓存，保留最近用过的目录项。

（3）目录文件中，目录项排序。

2、用户 user1 登录后，打开文件 a。请问，执行系统调用 open(“a”, RDWR)需要执行几次 IO 操作?



[参考答案]　2 次。

用户登录时，系统自动打开家目录。所以，user1 上机的整个过程，家目录/home/user1 的 DiskInode 常驻内存，为其提供相对路径名目录搜索服务。

这个 open 系统调用使用相对路径名打开文件 a，无需加载父目录 DiskInode。

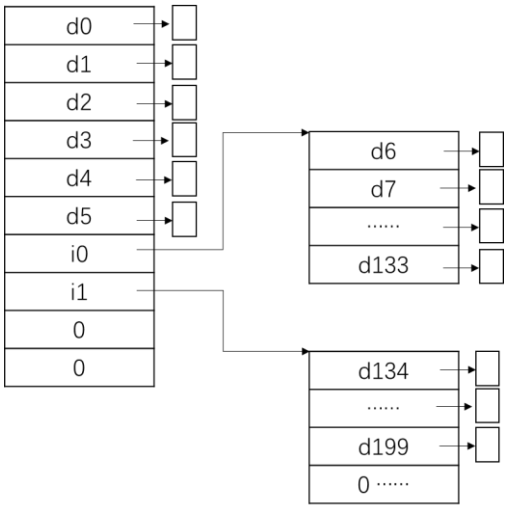
无需加载父目录 user1 的 DiskInode。父目录文件 user1 的数据块缓存不命中，遍历这个文件需要 IO 一次。

需要读取目标文件 a 的 DiskInode，缓存不命中，IO 一次。

所以，open 系统调用需要执行 2 次 IO 操作。

3、open 系统调用结束后，读下图中的文件 file1，假设所有数据块和缓存块缓存不命中，不考虑预读。（1）读 3#逻辑块，几次 IO？（2）读 6#逻辑块，几次 IO？（3）读入 6#逻辑块之后，接着读 100#逻辑块，几次 IO？（4）随后，写 198#逻辑块中的第 200#字节，几次 IO?

[参考答案]



open 系统调用结束后，file1 的 Inode 在内存里。

（1）读 3#逻辑块，1 次 IO。0#~5#逻辑块的物理块号在 Inode 中，文件打开后在内存。所以，读 3#逻辑块 IO 一次，读入物理块 d3。

（2）读 6#逻辑块，2 次 IO。第一次读入物理块 i0、得到文件的第一个索引块，get 物理块号 d6。第二次读入物理块 d6，get 文件数据。

（3）读 100#逻辑块，1 次 IO。100#逻辑块的物理块号登记在第一个索引块，已经在内存里了。所以，只需一次 IO 读入物理块 d100。

（4）写 198#逻辑块中的第 200#字节，2 次 IO。一次读入物理块 i1，获得第 2 个索引块，查询得知 198#逻辑块存放在 d198#物理块中。第二次 IO，执行先读操作，将物理块 198 读入磁盘高速缓存。没有写 IO，因为这个逻辑块没有写满，就让它呆在缓存池里。