文件系统例题与习题

同济大学计算机系操作系统　　2024-1-2

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
```

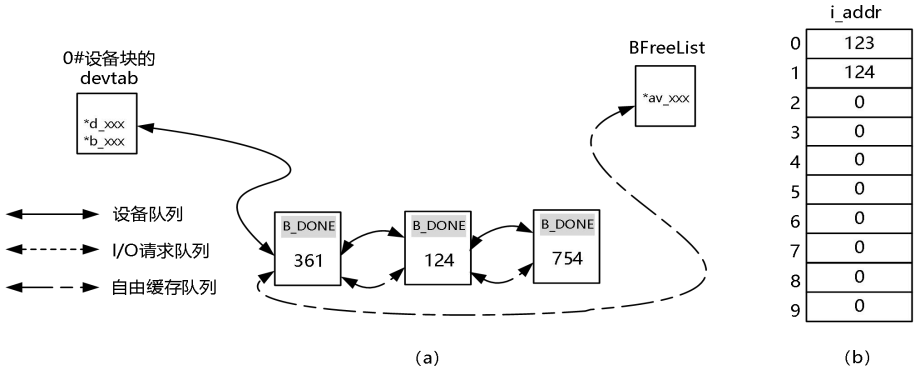


图 1、缓存队列 和 Jerry 文件的索引表

详见文档《文件系统例题与习题 2 完整的文件读写过程》

1. open 系统调用

首先 open 系统调用根据传入的文件名查找 DiskInode 是否存在。若不存在则报错，存在进行下一步

用 diskInode 号搜索内存中的 Inode 表，若存在则对应 inode 表项的 i_count++，否则需要分配空闲的 inode，读入磁盘 diskinode 进行初始化

检查文件访问权限，若 RWX 不支持当前 mode 则报错，否则进行下一步

创建文件的打开结构，在进程打开文件表中分配 File 结构和索引号 fd，构建 Ffd->File->Inode 的连接关系，最后返回索引号。

open 系统调用结束，i_lastr = -1

2. seek 文件指针移动

seek 将 File 结构中的 f_offset 文件指针移动到 500，即 f_offset=500

3. read 系统调用

首先初始化参数:m_offset=f_offset=500，m_base=array, m_count=300

由于 i_last_t + 1 = 0 = b_n = m_offset / 512，查找混合索引表，得到当前块的 bn 物理块号伪 123，下一块的物理块号为 124，则调用 Breada(123， 124)，当前缓存块 123 不命中，所以需要 Getblk 由自由缓存队列分配缓存块，当前进程构造 IO 访存块，放入 IO 请求队列中，当前进程入睡等待 IO 操作完成。IO 完成后唤醒当前进程，执行 iomove 将缓存块中 500#-511#字节复制到用户空间 data[0]-data[11]，缓存使用完毕之后解锁。修改 IO 参数，m_offset=512, m_base = data+12, m_count = 300-12 非零则仍需要继续读取下一块

当前逻辑块号 bn = m_offset / 512 = 1 ，块内偏移量 0，读取 250-12=238 字节。而 bn=i_lastr + 1 breada(124, 0)，缓存命中 124 则将其上锁并复用缓存块内容，执行 iomove 将缓存中 0#-237#字节复制到用户空间 data[12]-data[249]中，解锁缓存内容，修改 IO 参数，读操作结束，返回实际读入的字数 250

4. write 系统调用：

f_offset ==750，文件长度 750。write 系统调用将 data 数组中的 300 个字节追加写在文 件尾部。写操作完成后，文件长度增加至 1050。

初始化 IO 参数：m_offset = f_offset = 750，m_base = data，m_count = 300 写 1#逻辑块当前逻辑块 bn = m_offset/512 = 1，块内偏移量 238，写 274 字节至块结束。查混合索引表，得当前块 bn 的物理块号 124。 先读后写，字节数 274 不足 512 字节。124#物理块缓存命中，先读操作只需锁住该缓存块。iomove 将用户空间中的数据 data[0] ~ data[273]写入这块缓存。 修改 IO 参数，m_offset = 1024，m_base = data+274，m_count = 300-274。写至 1#逻辑块底部，异步写回磁盘。m_count 非 0，还要继续写。

写操作导致文件长度增加，i_size = 1024。2.3 写 2#逻辑块当前逻辑块 bn = m_offset/512 = 2，块内偏移量 0，此次写 26 个字节。混合索引表中 2#逻辑块的物理块号是 0，系统为其分配新数据块 new，登记： i_addr[2] = new。 读后写，这是因为写入的字节数 26 不足 512 字节。系统为新物理块 new 分配缓存 块，启动 IO 操作、同步读入磁盘数据块 new。iomove 将用户空间中的数据 data[274] ~ data[299]写入这块缓存。修改 IO 参数，m_offset = 1050，m_count = 0。未写至 1#逻辑块底部，延迟写。缓存块打脏标记，释放。 写操作导致文件长度增加，i_size = 1050。m_count 是 0，write 系统调用结束。返回实际写入文件的字节数 300。

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 =X+Y+Z，1000 + Z，1000 + Z**?

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 > 6 个块**

2 个一次间接索引块。i0 ， i1，**一个间接索引块管 128 个**

共计 202 个扇区。

混合索引结构如下：

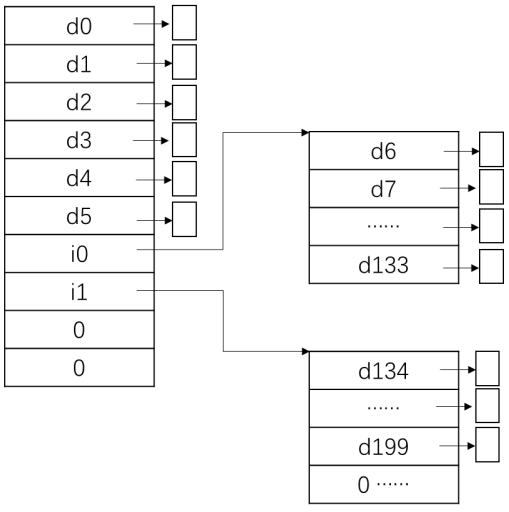
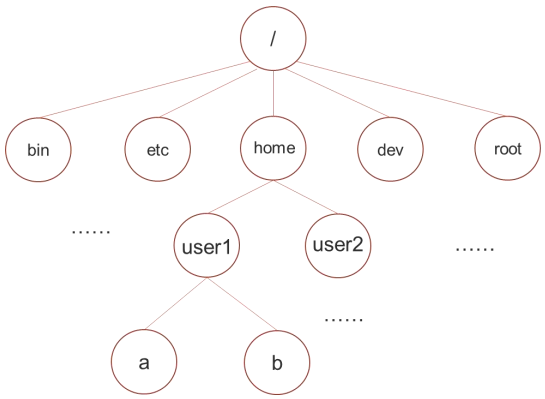


图 1

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 12#扇区，是这个扇区的 4#inode。
- bp=Bread(0,12)，将 12#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

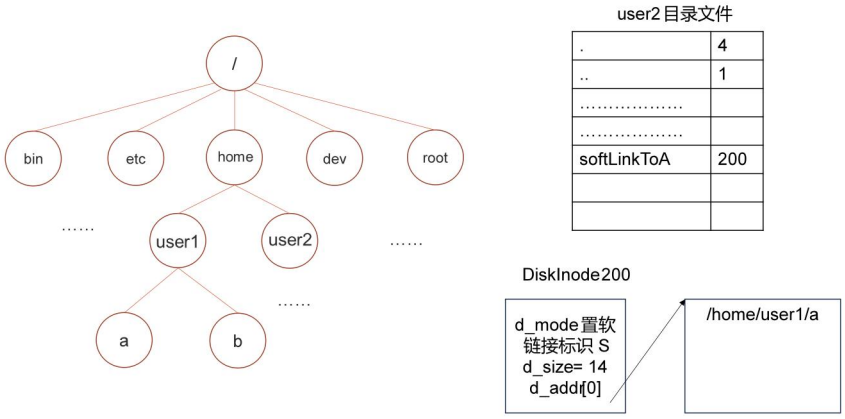
4、某磁盘剩余空间 30%，无法创建新文件。这是什么情况？怎样避免这种情况的发生。

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（**A**）个内存 Inode 指向该文件？（**B**）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）？(**File 结构（f_flag）**)

文件的读写指针保存在（**对应进程文件打开表项对应的 File 结构中**）？

组成文件的每个逻辑盘块（信息块）在磁盘上的地址保存在（**AC**）？

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（**A**）操作；稍后，P1 关闭 example 文件，引发（**ABC**）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

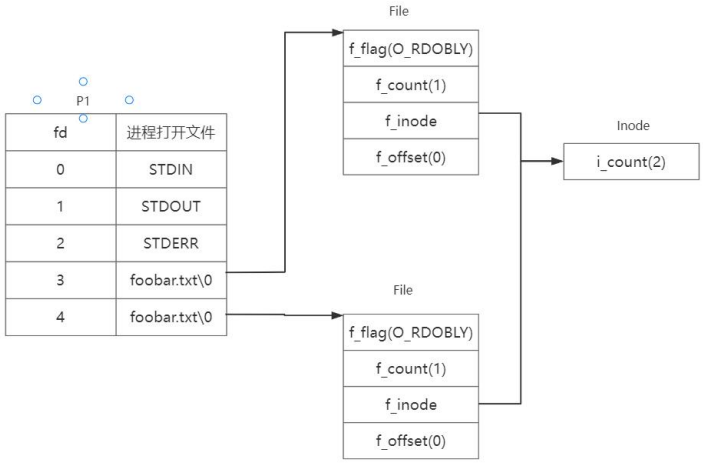
（1）这个程序的输出是什么？（2）画进程的打开文件结构

```
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);
}
```

（1）输出是：1

（2）



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);
}
```

均为 3

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。

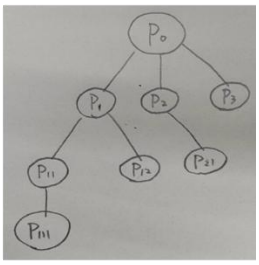
输出重定向怎么实现呢？以下是一种可行的方法：shell 进程创建一个子进程，子进程 exec(cat)之前，连续执行系统调用 close 和 open：

```
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
}
```

。。。父进程 shell 的程序分支。。。

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，输出是 20 行，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，遇到回车输入、输出。普通磁盘文件，块缓冲，写满 4096 字节输出。

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输出，所以，前者会重复输出父进程的字符行（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 write 系统调用，将用户缓存中留存的字符行写入内核缓存块。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、以图 1 中代码为例，线性目录搜索文件“/usr/ast/Jerry”。如果各级目录文件的内容如图 7.32 所示，则整个目录搜索的过程如下：

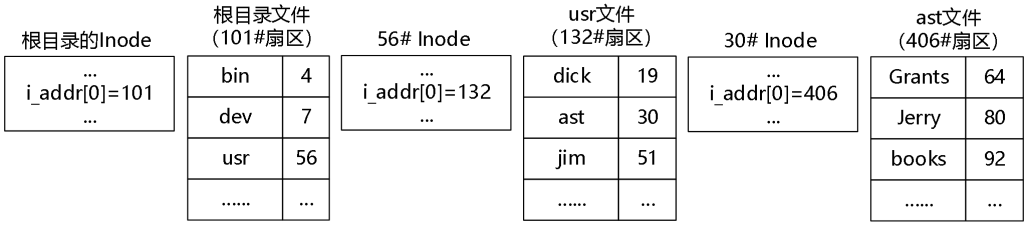


图 7.32：目录搜索示例

（1）根据 1 号 inode（根目录文件的 Inode），打开根目录文件，由 i_mode 中的标志位判断该文件确实为目录文件，由 i_addr 找到根目录文件在磁盘中的位置，如图 7.32 所示，从 101 号盘块开始依次读入根目录文件；在根目录文件中，逐条记录查找到文件名为 usr 的目录项，该项显示 usr 文件的磁盘 Inode 号为 56 号。根目录 DiskInode 常驻内存，使用不需要执行 IO 操作，遍历根目录文件之前需要将 101#盘块读入内存，1 次 IO。

（2）根据 56 号 Inode（user 文件的 Inode），打开 usr 文件，由 i_mode 中的标志位判断该文件确实为目录文件，则由 i_addr 找到 usr 目录文件在磁盘中的位置，如图 7.32 所示，从 132 号盘块开始依次读入 user 目录文件；在 usr 目录文件中，逐条记录查找到文件名为 ast 的目录项，该项显示 ast 文件的磁盘 Inode 号为 30 号。2 次 IO，分别读入目录文件 usr 的 DiskInode（56#）和数据块（132#盘块）。

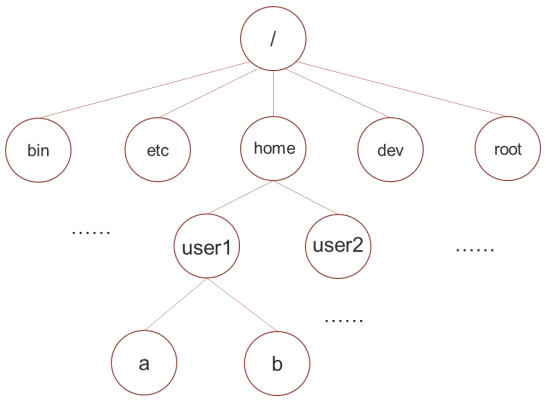
（3）根据 30 号 Inode（ast 文件的 Inode），打开 ast 文件，由 i_mode 中的标志位判断该文件确实为目录文件，则由 i_addr 找到 ast 目录文件在磁盘中的位置，如图 7.32 所示，从 406 号盘块开始依次读入 ast 目录文件；在 ast 目录文件中逐条记录查找到文件名为 Jerry 的目录项，该项显示 Jerry 文件的磁盘 Inode 号为 80 号。2 次 IO，分别读入目录文件 ast 的 DiskInode（30#）和数据块（406#盘块）。

（4）找到 80 号 Inode，即找到“/usr/ast/Jerry”文件。

结论：线性目录搜索好慢好慢呀呀呀。。。一定要想办法优化。

能用，尽量用相对路径名引用文件，会快好多。看下面的例子。

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。

需要读取父目录文件 user1，数据块缓存不命中，IO 一次。

需要读取目标文件 a 的 DiskInode，缓存不命中，IO 一次。

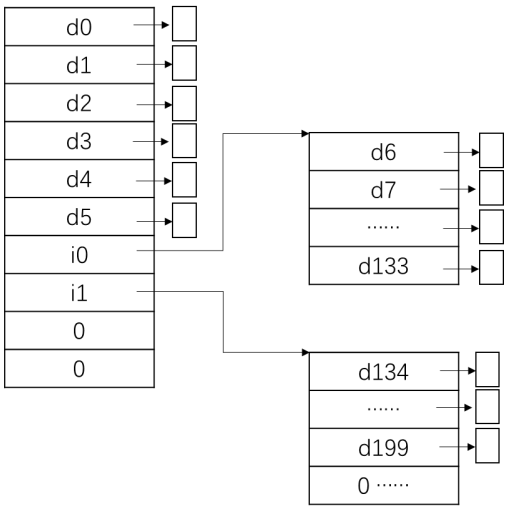
所以，open 系统调用需要执行 2 次 IO 操作。

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]

（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#逻辑块的地址映射关系在第一个索引块中，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。