

Unix V6++的目录和与之有关的系统调用

同济大学计算机系操作系统讲稿 邓蓉 2022-12-22 第 1 版 2023-12-31 第 2 版

目录文件与普通文件不同。普通文件无结构，存放的数据以字节为单位，是线性字符流。目录文件是定长记录文件，每个记录是一个目录项。下面首先介绍目录文件的结构和使用方法。

一、目录文件的结构和使用方法



图 1

1、Unix V6++系统中，每个目录项 32 字节。前 4 字节 m_ino 是 DiskNode 号，后 28 字节 m_name 是文件名（不是完整的路径名，是最后一个路径名分量），不足 28 字节的文件名尾部用\0 补齐。

上图，右子图，目录 ast 有 3 个文件：Grants，Books 和 temp，分别是系统中的第 64#、92#和 80#文件。m_ino==0 的目录项是空闲目录项。图中目录 ast 中原来有 mbox 文件，现已被删除。

目录 ast，当前长度 d_size==6*32 字节。

1.1 新建文件，需要在父目录文件中添加一个目录项。

- 如果父目录有空闲目录项，为新文件分配第一个空闲目录项。
- 否则，新目录项追加写在父目录文件的尾部。

例 1：接上图。在 ast 目录中添加 DiskNode 号是 100 的 newfile1 文件。红色的是分配给新文件的目录项。添加新文件后，父目录文件 ast 长度不变。

ast目录	
15	.
10	..
64	Grants
92	Books
100	newFile1
80	temp

图 2

例 2：接例 1。在 ast 目录中再添加一个 DiskNode 号是 102 的 newfile2 文件。红色的是分配给新文件的目录项。添加新文件后，父目录文件 ast 长度增加 32 字节、1 个目录项。

ast目录	
15	.
10	..
64	Grants
92	Books
100	newFile1
80	temp
102	newFile2

图 3

1.2 删除文件

删除文件比较简单，父目录文件中搜索找到目标文件的目录项，将对应的 m_ino 赋 0。

1.3 在目录文件中搜索一个已存在的目标文件

举例说明。

细节：目录文件中搜索一个已存在的文件

例：父目录文件 “father”（98#DiskNode）中查找已存在文件 “child”。

- 1、为98#DiskNode分配内存Inode，读入磁盘上的DiskNode；
- 2、i_addr数组，确定存放父目录文件的所有数据块；
- 3、外层循环，遍历i_addr数组，Bread父目录文件的每个数据块。
 - 数据块512字节，目录项32字节。每个数据块有16个目录项。
 - 内层循环16次，每次取32字节（1个目录项），匹配字符串 “child\0”。匹配成功，跳出内存循环，跳出外层循环，返回文件child的DiskNode号m_ino。
 - 外层循环结束没有匹配成功，文件不存在，目录搜索失败。




图 4

目录文件，需要先打开，才能使用，搜索其中的目录项。打开操作，需要读取 DiskNode，启动 1 次 IO 操作，同步读磁盘。线性目录搜索技术需要遍历目录文件，这需要启动 IO，读入目录文件的所有数据块。

1.4 目录搜索

按文件路径名搜索文件树。下图是一棵 Linux 文件树。

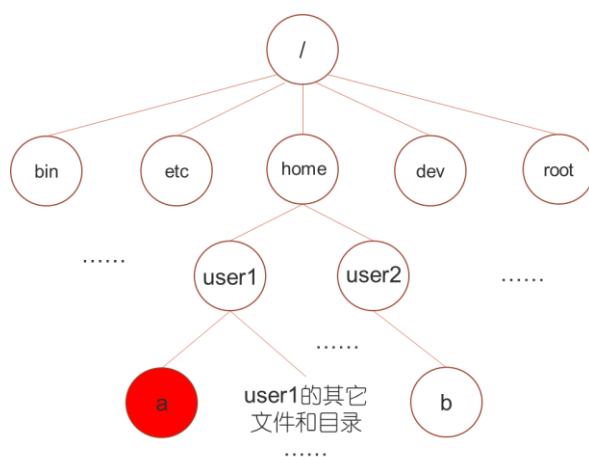


图 5

进程引用图中红色节点可以使用绝对路径名，也可以使用相对路径名。

- 绝对路径名是/home/user1/a，是根目录到这个节点的唯一路径。
- 相对路径名是 a，是进程当前工作目录/home/user1 到 红色节点的唯一路径。

[旁白] 使用相对路径名，目录树依然每个节点可达。比如，user1 用户访问 b 文件，可以用相对路径名：../user2/b。Linux 系统中，普通用户的家目录在 home 目录下。比如，用户 user1 的家目录是 /home/user1。这是进程默认的当前工作目录，除非用 cd 命令改变当前工作目录。/root 是超级用户 root 的家目录。root 是 0#用户，系统管理员。

1.4.1 目录搜索基本算法 —— 线性目录搜索

一、目录搜索的起点

- 系统里有一个全局变量 rootDir，是根目录的 Inode。所有进程共享。这是绝对路径目录搜索的起点。绝对路径名，以字符 '/' 起始。
- 每个进程的 user 结构里，有 u_cdir，是进程当前工作目录的 Inode。这是相对路径目录搜索的起点。相对路径名，起始字符不是 '/'。

二、路径名分量

路径名分量是文件路径名中，用路径名分隔符隔开的文件名字符串。比如：

- 1、/home/user1/a，3 个路径名分量分别是：“home”，“user1”，“a”
 - 2、../user2/b 的 3 个路径名分量，分别是：“..”，“user2”，“b”
- 其中，'/' 是路径名分隔符。

三、线性目录搜索

以例示之。线性目录搜索已有文件 /home/user1/a。

- 1、以 '/' 起始，绝对路径名。目录搜索的起点是根目录，内核全局变量 rootDir。
- 2、路径名共有 3 个路径名分量，分别是：“home”，“user1”，“a”。
- 3、目录搜索很简单，就是依次遍历每个目录文件，搜索当前路径名分量。对应我们的例子：


- 遍历根目录文件，找文件名是“home”的目录项，得 home 目录文件的 inode 号
- 打开、遍历 home 目录文件，找文件名是“user1”的目录项，得 user1 目录文件的 inode 号
- 打开、遍历 user1 目录文件，找文件名是“a”的目录项，返回这个目录项的 inode 号

搜索时，只要有一个路径名分量没找到，就失败。 文件不存在。

[细节] 系统初始化时，打开根目录文件，rootDir 指向该文件的内存 Inode。系统运行时，这个内存 Inode 常驻内存，为所有进程、任何文件提供绝对路径名搜索服务。

u_cdir 指向进程当前工作目录的内存 Inode。进程运行期间，这个内存 Inode 常驻内存，为本进程提供相对路径名搜索服务。进程终止时，关闭当前工作目录。子进程继承父进程的当前工作目录，shell 进程可以使用 cd 命令改变当前工作目录。

具体实现，以例示之： open 系统调用打开文件/usr/ast/Jerry



细节：目录搜索全过程（open系统调用）

fd = open(name, mode);

- 1、完整的路径名在u_arg[0];
- 2、确定目录搜索的起点。当前目录 plnode =
 - 绝对路径，根目录的内存Inode
 - 相对路径，u_cdir（当前工作目录的内存Inode）
- 3、while（true） // 在当前目录中，搜索当前路径名分量
 - u_arg[0]中摘取当前路径名分量
 - 在当前目录中搜索当前路径名分量，得到一个DiskInode号（m_ino）。用上页PPT所述方法。
 - 释放当前目录的内存Inode。plnode=IGet(m_ino) //分配内存Inode，将m_ino#DiskInode读入内存
 - 路径名分量搜索完毕？
 - Y，返回目标文件的内存Inode，plnode。
 - N，当前目录 = plnode

例：目录搜索 /usr/ast/Jerry 文件：

while的第1轮，
当前目录是 /，当前路径名分量是 usr/0。

while的第2轮，
当前目录是usr，当前路径名分量是 ast/0。

while的第3轮，
当前目录是ast，当前路径名分量是 Jerry/0。

。。。目录项里的文件名，是路径名分量

操作系统
电信学院计算机系 邓睿
37

图 6

图 4、图 6 是 open 系统调用调用 NameI()函数实施目录搜索的全过程。NameI() 好重要，搜索文件树。

Inode* FileManager::NameI(char (*func)(), enum DirectorySearchMode mode)

- func 是一个函数，用来从路径名中提取下一个字符。
- mode 是 NameI()函数的工作方式。open 系统调用以打开方式搜索文件树（目录树），mode==OPEN。

```
enum DirectorySearchMode    // NameI的3种工作模式
{
    OPEN = 0,                /* 以打开文件方式搜索目录 */
    CREATE = 1,              /* 以新建文件方式搜索目录 */
    DELETE = 2,              /* 以删除文件方式搜索目录 */
};
```

除 OPEN，目录搜索（NameI 函数）还有 2 种工作模式：

- 1、Creat 系统调用，在父目录中创建一个新文件。CREATE 模式。
- 2、Unlink 系统调用，从父目录中删除一个目录项。DELETE 模式。

OPEN 模式，需要查询返回目标文件的内存 Inode。

CREATE 模式，需要遍历目标文件的父目录文件，比对所有已存在的目录项，确定新文件的确不存在。之后 NameI 函数会返回父目录文件的内存 Inode 和 父目录文件中第一个空闲目录项的文件偏移量。

DELETE 模式，返回父目录文件的内存 Inode 和 被删除目录项在文件中的偏移量。

总结和评论：

NameI 实施的线性目录搜索是最基本的技术，没有经过任何优化。

主要的缺点在于：慢。搜索父目录文件确定目标文件不存在，NameI 需要遍历父目录。如果父目录文件很小，长度小于 16 个目录项，需要 IO 磁盘 2 次，第一次读入父目录文件的 DiskInode，第二次读入父目录文件的数据块。一般而言，父目录有 n 个数据块，NameI 搜索它就需要 $IO\ n+1$ 次。

带给我们的启迪：一个目录装很多文件是不明智的，这会降低目录搜索的速度。

改进的思路。第一个思路（Linux 用的）：内核在核心态内存空间设置目录项缓存，用来保存最近用过的目录项。目录搜索时，只有在目录项缓存不命中的情况下，才会去磁盘搜索目录文件。第二个思路（Windows 用的）：磁盘上的每个目录文件，目录项按文件名、依字典序排序。这样，目录搜索可以用折半查找，IO 次数可以降至 $\log(n)$ 。对于非常大的目录文件，甚至有必要为它配一棵 B+树。

二、Open 系统调用

1、工作流程（要求熟练掌握）

第一步：线性目录搜索，找到目标文件的 DiskInode 号。（NameI 函数）。

第二步：在内存 Inode 池中搜索目标 DiskInode，锁上它。如果不命中，分配内存 Inode，启动磁盘读入 DiskInode。（NameI 函数→IGet 函数）。

第三步：检测进程对目标文件的访问权限。（Access 函数）

第四步：在系统打开文件表中分配一个 File 结构，在进程打开文件表中分配一个空闲表项（下标是 fd），建立打开文件结构。（FAlloc 函数）

第五步：返回文件描述符 fd。

2、源码分析（Access 函数工作过程要求掌握）



Open系统调用的完整过程

```
void FileManager::Open()
{
    Inode* plnode;
    User& u = Kernel::Instance().GetUser();

    // 1、目录搜索，确定目标文件的DiskInode号。调用 IGet函数 分配内存Inode: plnode，启动磁盘同步读入DiskInode & 锁住它。
    plnode = this->NameI(NextChar, FileManager::OPEN); // 若内存命中，不必读。和别的进程互斥使用便是

    // 报错，系统调用返回 1、文件不存在，User::ENOENT； 2、没有内存Inode，u.u_error = User::ENFILE；
    if ( NULL == plnode )
    {
        return;
    }

    this->Open1(plnode, u.u_arg[1], 0); // 2、权限检测，建立打开文件结构，返回文件描述符
}

mode: FREAD, FWRITE 或 FREAD | FWRITE, 供 Open1 检测
```

操作系统 电信学院计算机系 邓蓉 39

Open1() 是 Open 系统调用和 Creat 系统调用需要执行的公共函数，负责检测进程对文件的访问权限 & 建立打开文件结构。它有 3 种工作模式 (trf)，参见代码注释。

```
fd = open(name, mode); // 完成后，执行 fd = creat(name, mode);

/* Open 和 Create 系统调用需要执行的公共函数
 * trf == 0由Open调用： NameI成功搜索到目标文件： Open1(plnode, u.u_arg[1], 0)
 * trf == 1由Creat调用， NameI搜索到同名文件： Open1(plnode, File::FWRITE, 1)
 * trf == 2由Creat调用， NameI未搜索到同名文件： Open1(plnode, File::FWRITE, 2)， 创建一个新文件
 */
void FileManager::Open1(Inode* plnode, int mode, int trf) // 检测进程对文件的访问权限 & 建立打开文件结构
{
    User& u = Kernel::Instance().GetUser();

    /*
     * 打开已有文件（目标DiskInode存在）， 即trf == 0或trf == 1， 需要进行权限检测
     * Creat新建文件， 即trf == 2， DiskInode还没呢。跳过这一步。
     */
    if (trf != 2)
    {
        // 2、权限检测逻辑
    }
}
```

操作系统 电信学院计算机系 邓蓉 40

Open1() 根据 NameI 返回的内存 Inode: plnode，检测进程对文件的访问权限，如下图。

完成后, 执行

`fd = open(name, mode);` // mode, 读打开、写打开 还是 读写打开



2、权限检测逻辑细节

```
if ( mode & File::FREAD )
{
    /* 检查读权限 */
    this->Access(plnode, Inode::IREAD);
}
if ( mode & File::FWRITE )
{
    /* 检查写权限 */
    this->Access(plnode, Inode::IWRITE);
    /* 不可以写目录文件 */
    if ( (plnode->i_mode & Inode::IFMT) == Inode::IFDIR )
    {
        u.u_error = User::EISDIR;
    }
}
// 错误代码：无读写权限, u_error == User::EACCES; 写目录文件, u_error == User::EISDIR
```

Access() 逻辑: 用进程的uid识别访问文件的用户, 确定其对文件的访问权限。具体过程如下:

- 1、从plnode指向的内存Inode中读出文件的 uid(i_uid), gid(i_gid) 和 **RWX**RWX (i_mode)
- 2、从现运行进程的Process结构中读出 p_uid 和 p_gid
- 3、i_uid == p_uid ? Y, 用i_mode中**RWX**进行权限检测。否则, gid相等吗? Y, 用**RWX**。否则, 用RWX。
- 4、确定读写权限。匹配open系统调用的mode 和 Inode中的RWX。对应bit 该是1。失败, u_error = User::EACCES

注: p_uid==0的超级用户, 不受此规则限制。所以, 超级用户可以任何方式访问任何文件。

Write 系统调用不能写目录文件

回到 PPT 40, 看通过权限检测之后, Open1() 下一步要干嘛。



```
if ( u.u_error )
{
    this->m_InodeTable->IPut(plnode);
    return; // 解锁Inode, 系统调用出错返回
}

/* 3、在creat文件的时候搜索到同文件名的文件, 释放该文件所占据的所有盘块, i_size 归0 */
if ( 1 == trf )
{
    plnode->ITrunc();
}

/* 解锁plnode指向的内存Inode, 但计数器 i_count 的值不能动。所以不能用 IPut( )。 */
plnode->Prele();
```

权限检测没通过, open 失败, 返回前, 解锁内存 Inode。

最后, 建立打开文件结构, 返回文件描述符。

$fd = open(name, mode);$

/* 4、分配 File 结构和文件描述符。
分配失败，open/creat 系统调用失败返回 */
 File* pFile = this->m_OpenFileTable->FAlloc();
 if (NULL == pFile)
 {
 this->m_InodeTable->IPut(pInode);
 return;
 }
/* 5、File 结构中登记文件的打开方式
 pFile->f_flag = mode & (File::FREAD | File::FWRITE);
 pFile->f_inode = pInode; **// File 结构指向内存 Inode** ,

/* 6.1 打开文件结构建立完成，成功返回
/* 6.2 打开文件结构未成功建立，释放资源，拆除打开文件结构

操作系统 电信学院计算机系 邓蓉

```

/*作用：进程打开文件描述符表中找的空闲项 之下标 写入 u_ar0[EAX]*/
File* OpenFileTable::FAlloc()
{
    int fd;
    User& u = Kernel::Instance().GetUser();

    /* 在进程打开文件描述符表中获取一个空闲项 */
    fd = u.u_ofiles.AllocFreeSlot(); // 分配文件描述符
    在进程的打开文件表中找空闲项，下标fd → u.u_ar0[User::EAX]
    if (fd < 0) /* 如果寻找空闲项失败 */
    {
        return NULL;
    }

    for (int i = 0; i < OpenFileTable::NFILE; i++)
    {
        /* f_count==0表示该项空闲 */
        if (this->m_File[i].f_count == 0) // 分配空闲 File
        {
            令打开文件表，下标为fd的元素 指向File结构
            u.u_ofiles.SetF(fd, &this->m_File[i]);
            /* 增加对File结构的引用计数 */
            this->m_File[i].f_count++;
            // 初始化读写指针，清0
            this->m_File[i].f_offset = 0;
            return (&this->m_File[i]);
        }
    }

    Diagnose::Write("No Free File Struct\n");
    u.u_error = User::ENFILE;
    return NULL;
}
  
```

open 系统调用的主要开销 = 目录搜索的开销 + 将目标 DiskInode 读入内存 Inode (IO 一次)。目录搜索开销包含从磁盘读取目录文件及其 DiskInode 的开销。使用相对路径名有利于缩小目录搜索开销，这是内核一定要支持相对路径名的一个主要原因。

三、系统调用 close(), creat(), link() 和 unlink()

1、系统调用 close(), 释放资源，拆除打开文件结构。

具体而言，进程打开文件表中，fd 对应的表项置空。拆除打开文件结构，释放 File 结构和内存 Inode。**若内存 Inode 已无进程使用，将其同步写回磁盘（对脏 Inode）。**细节见下图源码分析。

系统调用 Close(fd)

void FileManager::Close()
 {
 User& u = Kernel::Instance().GetUser();
 int fd = u.u_arg[0];

 File* pFile = u.u_ofiles.GetF(fd); **// 1 用打开文件描述符获取File结构**

 if (NULL == pFile)
 {
 return;
 }

 u.u_ofiles.SetF(fd, NULL); **// 2 进程打开文件表，fd对应的元素置空**
 this->m_OpenFileTable->CloseF(pFile); **// 3 递减File结构引用计数。**
 }
 }

void OpenFiles::SetF(int fd, File* pFile)
 {
 if (fd < 0 || fd >= OpenFiles::NOFILES)
 return;
 this->ProcessOpenFileTable[fd] = pFile;
 }
 }

置 null

操作系统 电信学院计算机系 邓蓉



```

void OpenFileTable::CloseF(File *pFile)
{
    Inode* pNode;
    ProcessManager& procMgr = Kernel::Instance().GetProcessManager();

    /* 管道类型 */
    .....

    if(pFile->f_count <= 1) // 如果这是引用File结构的最后一个进程
    {
        pFile->f_inode->CloseI(pFile->f_flag & File::FWRITE); // 关闭特殊文件
        g_InodeTable.IPut(pFile->f_inode); // 内存Inode引用计数减1。如果减至0，
                                           打上时间戳，写回磁盘DiskInode。
    }

    pFile->f_count--; // File结构的引用计数减1。f_count==0, File结构空闲。不需要释放的。
}

```

2、系统调用 Creat() 创建新文件



系统调用 Creat("file" , mode)

- 线性目录搜索，打开新建文件的父目录文件
- 父目录文件，进程有写权限嘛？
 - 没有，出错返回
- 父目录文件中有同名文件 "file" ?
 - 有，清空已有文件：打开目标文件Inode，从0#逻辑块开始，释放所有数据块。i_addr数组清0，i_size=0。
 - 没有，
 - 父目录文件中为新文件分配一个目录项
 - 分配一个DiskInode，d_link=1，d_mode=mode(Creat系统调用的第2个参数)，d_uid=p_uid，d_size=0；
 - 将DiskInode号和文件名写入新目录项
- IPut()父目录文件的内存Inode
- 为新建文件建立打开文件结构。File结构的 f_mode=FWRITE, f_offset=0
- 返回 fd

3、系统调用 Unlink()，删除文件

具体而言，父目录文件中删除该文件对应的目录项。DiskInode 硬链接数 d_link--，减至0，删除文件。删除文件后，分配给文件的数据块号全部送入空闲盘块号栈。DiskInode 号送入空闲 inode 号栈。



系统调用 Unlink()

`unlink("name");` // 在name所在的父目录文件中删除目录项。如果引用的DiskInode, d_link变成0, 删除磁盘文件 (回收数据块和 DiskInode)。把它们送进SuperBlock的空闲盘块号栈和空闲Inode栈。

删除 ast / mbox 文件

- 线性目录搜索 mbox
 - 父目录ast有写权限嘛?
 - 没有, 退出
 - 在ast目录文件中找到mbox对应的目录项 myDirEntry
 - `n = myDirEntry.m_ino`
 - `myDirEntry.m_ino = 0`
 - 修改 n# inode
 - `d_link -1`, 置脏标记
 - `d_link` 是 0 嘛?
- 是: 删除文件: 从0#逻辑块开始, 释放所有数据块 & 释放 n# inode。

ast目录 406#扇区

15	.
10	..
64	Grants
92	Books
9	mbox
80	temp

(a)

ast目录 406#扇区

15	.
10	..
64	Grants
92	Books
0	mbox
80	temp

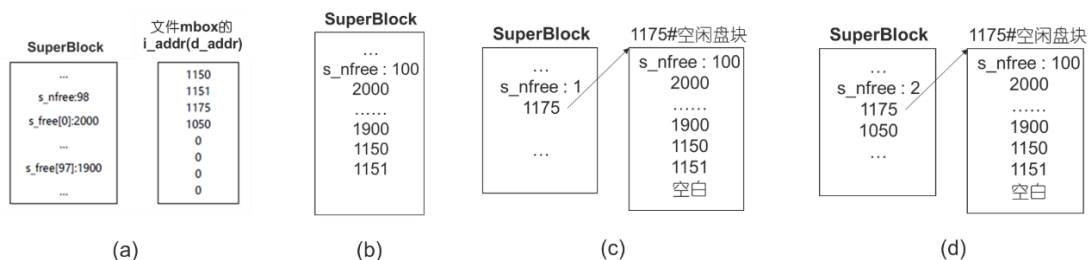
(b)

以例示之。例 3: T 时刻删除 9#磁盘文件 mbox。

(1) 删除目录项。如图 PPT48, 父目录 ast 文件中, mbox 对应目录项 m_ino 清 0。

(2) 遍历 9#DiskNode 的混合索引表, 依次回收所有磁盘数据块。彼时空闲盘块号栈中有 98 个盘块。如下, 子图 a。

- 回收 1150#、1151#数据块后空闲盘块号栈满, 子图 b。(回收的数据块 1150, 1151 依次压栈。。。)
- 继续回收 1175#数据块, 系统将空闲盘块号栈复制进这个新的空闲盘块, 子图 c。现在栈中只有 1175#, 这一个空闲盘块。
- 最后回收 1050#数据块, 子图 d。



(3) 回收 DiskInode。

- 9#DiskInode, d_mode IALLOC 位清 0, 标记空闲。如下, 子图 a。
- 若空闲 inode 栈不满, 9#DiskInode 压栈, 成为新栈顶, 子图 b。
- 否则, 这个空闲 DiskInode 号不回收, 子图 c。



续：Unlink删除文件后的空闲inode号栈 和 磁盘Inode区

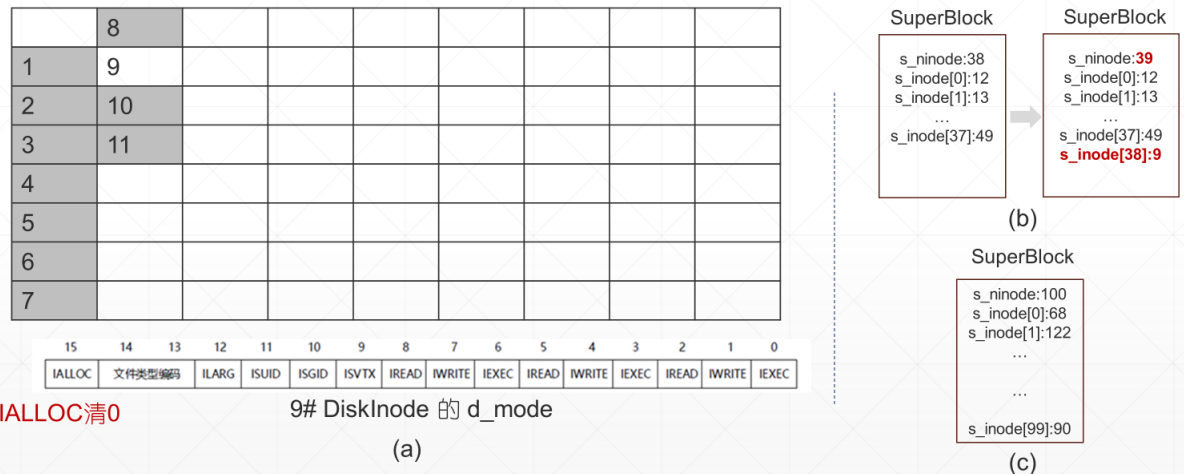


图 3。灰色 DiskNode 已分配，白色的空闲。

例 4：接例 3，10#用户 ken 执行如下程序。在 ast 目录中创建新文件 newfile，写入字符串 "holiday!!!\n"。请问，程序执行完毕后 ast 目录文件，newfile 文件的 DiskNode，空闲盘块号栈 和 空闲 Inode 栈 长什么样？

代码：

```
char * newData = "holiday!!!\n";
int fd = create("ast/newfile", 0666); // 前缀 0 表示 8 进制数
int count = write(fd, newData, 12); // 要包含尾 0
```

答：新建文件做 2 件事，新建一个目录项写入父目录，分配一个空闲 DiskNode，初始化。写文件做 3 件事，为文件数据分配磁盘数据块，为数据块分配缓存块，数据写缓存块。

具体而言，create 系统调用

第 1 步：为文件 newfile 分配一个 DiskNode，9。

第 2 步：在父目录文件 ast 中分配一个目录项，填入文件名 newfile 和 DiskNode# 9。

第 3 步：将 9#DiskNode 读入内存 Inode，初始化。

d_mode = 100***110110110，IALLOC 是 1 表示 DiskNode 被占用，红色的是系统调用 create 提供的文件访问权限 RWXRXRWX。

d_uid = 现运行进程的 p_uid。d_size = 0。d_addr[]全 0。

write 系统调用

第 1 步：为文件 newfile 分配第 1 个数据块，1050。i_addr[0]=1050。

第 2 步：写入字符串 "holiday!!!\n"

第 3 步：写入后文件长度增加了，修改 d_size = 12。

注意，写入的数据在 1050#数据块占据的缓存块中，还没写回磁盘。

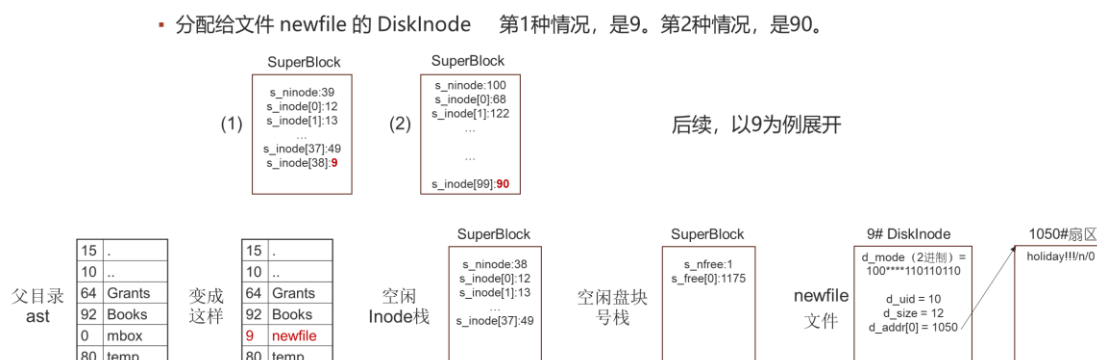
新建的 9#DiskNode 在内存 Inode 中，也还没写回磁盘。

那，什么时候写回磁盘呢？关闭文件的时候，9#DiskNode 写回磁盘。1050#数据块是

延迟写的，关闭文件时也未必会写回磁盘。

文件系统会丢数据。这是延迟写必需付出的代价，必需用其它技术来弥补。

下图，是程序执行完毕后，与本程序相关的所有文件系统元数据和文件数据块。



4、系统调用 Link(), 为文件建立新的硬链接

系统调用 Unlink()

`link("name1", "name2");` // 在name2所在的父目录文件中添加一个引用name1文件的目录项。

`link("mbox", "newfile")`

- 线性目录搜索，找到 name1 文件的 DiskNode 号 n
- name2的父目录有写权限嘛？
 - 没有，退出
- 父目录文件中有同名文件，name2？
 - 有，出错返回。
 - 没有，父目录文件中分配一个空闲目录项，写入 DiskNode 号 n 和 name2的最后一个路径名分量
- 修改 n# DiskNode
 - d_link ++，置脏标记
- 必要时修改name2父目录文件的DiskNode。。。如果文件长度增加

