第19章 文件目录和目录文件

正如我们已了解到的，与各单个文件有关的重要信息存放在“ inode ”表中。若该文件是当前可存取的，或者是正被存取的，那么相关信息存放在内存“ inode ”表中。如果一个文件在磁盘上(更一般而言，在某个“ 文件系统卷 ”中)，并且当前不可存取，那么相关的“ inode ”表存放在磁盘上(文件系统卷中)。

19.1 文件名

引人注意的是在“ inode ”表中缺少文件名信息。这种信息存放在目录文件中。

每个文件应当至少有1个名字。一个文件可以有几个不同的名字，但是两个不同的文件不能共用一个名字，亦即每个名字只能规定一个文件。

一个名字可以由多个部分组成。在书写名字时，名字中的各部分名或分量名 (component) 由斜线（/）分隔。一个名字中各分量名出现的顺序是有意义的，亦即，“ a/b/c ”不同于“ a/c/b ”。

如果将文件名分成两部分：一个起始部分或“ 主干 ”（stem）以及一最后部分或“ 尾端 ”（ending），那么若两个文件的名字主干部分相同，则通常它们会具有一定的关系。它们可能驻留在同一磁盘上，可能属于同一用户等。

19.2 目录数据结构

开始时用户用文件名引用所需的文件，例如在“ open ”系统调用中。操作系统的一个重要功能是将文件名变换成相应的“ inode ”项。为了实现这一点， UNIX创建并维护一目录数据结构。此结构等效于以名字为边的有向图。

该结构的最纯粹形式是树，亦即，它有一个根节点，在根节点和任一节点之间只有一条路径。更通常一点在UNIX中，该结构(图)是一个格(在其他操作系统中则不一定如此)。把一个或几个叶节点图结合起来，就可从树结构得到格结构。

在此情况下，在根和任一内部节点之间仍旧只有一条路径，但在根和任一叶节点之间都可能有若干条路径。叶节点是没有下层节点 (或称后继节点)并且对应于数据文件的节点。内部节点是具有下层节点并且对应于目录文件的节点。

由根节点至对应于某个文件节点的路径包含多条边，这些边的名字按序组织起来就构成了文件名。由此，这种文件名通常被称之为“ 路径名 ” (pathname)。如果一个文件有多条这种路径，那么该文件就有多个名字。

19.3 目录文件

目录文件在很多方面与非目录文件并无区别。但是，目录文件中所包含的信息可被用来

定位其他文件，因此其内容受到严格保护，只能由操作系统独自处理。

每个文件的信息都存放在1个或多个512字符块中。目录文件的每一块分成32×16字符结构。每个结构中包含1个16位“ inode ”表指针，以及一个14个字符组成的名字。该“ inode ”指针指向位于同一磁盘(或文件系统卷)上的“ inode ”表。目录文件及其引用的各文件也存放在同一磁盘上(在以后将对此作更多说明)。若该“ inode ”指针值为0，则表示这是目录中的一个空闲项。

下列过程都与目录有关：

- namei (7518) 搜索目录。
- link (5909) 创建另一个文件名。
- wdir (7477) 写目录项。
- unlink (3510) 删除一文件名。

19.4 namei(7518)

7531：“ u.u_cdir ”规定了相应进程当前目录的“ inode ”。一个进程在刚出生时（newproc, 1883)继承其父进程的工作目录。可以用“ chdir ”(3538)系统调用更改进程自身的当前工作目录。

7532：注意，“ func ”是“ namei ”的一个参数，它只能是“ uchar ”(7689)或“ schar ”(7679)。

7534：调用“ iget ”(7276)，其主要操作是：

- 等待，直至对应于“ dp ”的“ inode ”不再被锁住。

安装的。

即 ////a///b/ 等同于/a/b）。

工作目录或根目录，则立即跳转至本过程结束处！

程序循环的开始处，该循环的尾端在 7667行。每次循环分

，由一个null字符，或由一个或多个斜线结束的一个字符

不同的字符构造成的 (7571)。

的终点。返回“ dp ”的当前值。

对目录文件而言则被解释为“ 搜索 ”权限。

个目录项相匹配时，先将它复制到一个能更快存取的单元

部分被截去。

该目录中的目录项编号数。

程序循环的开始处，该循环的尾端在 7647行。每次循环处

目录，但是没有找到所给出路径分量名的匹配项，则除下

1) 这是该路径名的最后一个分量名，亦即“ c=='\0' ”。

2) 该文件是应被创建的，亦即“ flag==1 ”。

3) 用户程序对该目录具有“ 写 ”权限。

7606：将新文件所在目录的“ inode ”地址记入“ u.u_pdir ”。

7607：如果以前已经遇到了一个可存放新目录项的目录文件区 (7642)，则将此区域的位置值存入“ u.u_offset〔1〕”，否则将IUPD标志设置至“ dp ”所指向的“ inode ”中(为什么?)。

7622：在合适时，在谨慎地释放某个以前持有的缓存后，读目录文件的一个新块 (注意，使用的是bread，为什么不是breada呢?)。

7636：将目录项的8个字复制到“ u.u_dent ”数组中。在比较之前进行这种复制的原因并不清楚！是不是提高了效率？

7645：这一比较有效地使用了单个字符指针寄存器变量“ cp ”。如果一个字一个字地进行比较，此循环的效率会更高一些。

7647：“ eloop ”循环由下列语句之一终止：

“ `return(NULL);` ”(7610)

“ `goto out;` ”(7605,7613)

一次成功的匹配，使得无需跳转至“ eloop ”(7647)。

7657：如果此名字是要被删除的（ flag==2),该路径名已经结束，用户程序对此目录具有“ 写 ”权限，那么返回指向该目录“ inode ”的指针。

7662：暂时保存设备标识 (为什么不保存在寄存器型变量 c中呢?)，然后调用“ iput ”(7344)，它解锁“ dp ”，对“ dp ”的引用计数减1，接着进行后续处理。

7664：使“ dp ”指向下一层次文件的“ inode ”。

7665：因为该目录指明该文件存在，所以条件“dp==NU
表溢出以及i/o错则可能发生，有时在一次系统崩溃后，文件

19.5 一些注释

“ namei ”是一个关键性的过程，似乎在UNIX的实施中
地测试和排错，然后就基本不变。在“ namei ”和UNIX系
的，单单由于这一条理由，就不能对“ namei ”授予“ 本年

在12个不同过程中对“ namei ”进行了13处调用，它们

行	例 程	
3834	exec	ocbar 1
3543	chdir	ocbar 1
5778	open	ocbar 1
5914	link	ocbar 1
6033	stat	ocbar 1
6097	rmount	ocbar 1
6108	getmdev	ocbar 1
6976	owner	ocbar 1
5786	creat	ocbar 1
5928	link	ocbar 1

(续)

行	例 程	参 数
5050	mknod	uchar 1
3515	unlink	uchar 2
4181	core	schar 1

从中可见：

1) “ link ”中有2处调用“ namei ”。

2) 可将这些调用分成4类，其中第1类最大，它包含了8处“ namei ”调用。

3) 最后2类都只有一处“ namei ”调用。

4) 特别的，与“ schar ”例程有关的“ namei ”调用只有1处，它总是针对名为“ core ”的文件。(若将此处理为一种特例，例如将第 2个参数值取为3，那么在上表中就可以将 uchar和schar删去。)

“ namei ”可以以多种方式终止：

1) 如果已出错，则返回“ NULL ”，并且设置了变量“ u.u_error ”。(大多数错误造成的结果是跳转至标号out(7669)，这使得对inode的引用计数正确地被保持 (7670)。若在iget(7664)中出错，则不必如此)；

2) 如果“ flag==2 ”(亦即，这是来自unlink的调用)，那么在正常情况下的返回值是命名文件目录的“ inode ”指针(7660)；

3) 如果“ flag==1 ”(亦即，此调用来自 creat、link或mknod，并且若命名文件现在并不存

并不存在，则返回“ NULL ”值(7610)。在此情况下，新值存放在“ u.u_pdir ”中(7606)。(也请注意，在此情况下，页，或者指向目录文件的尾端。)

已存在，则返回指向该文件“ inode ”的指针 (7551)。该1。后面需调用“ iput ”以取消这些副作用。

它将一个现存文件的新名字加至目录结构中。此过程的参新名字。

不同的名字，则设置u.u_error后立即退出。

那么只有超级用户才可以为它增加一新名字。

”。在本过程开始处调用“ namei ”，而它又调用“ iget ”上锁。

”时产生的错误会造成灾难呢?在搜寻新名字时发现该现存是不可能，但也是极少有的。最可能的情况是：系统力图是它已经存在。

5927：为第2个名字搜索该目录，其意图是创建一新目录项。

5930：有一个具第2个名字的文件。

5935：“ u.u_pdir ”是作为“ namei ”调用(5928)的副作用而设置的。检验该目录与该文件驻留在同一设备上。

5940：写一新目录项(参见下面的说明)。

5941：增加该文件的“ 连接 ”计数。

19.7 wdir(7477)

此过程将一个新名字写入一目录中。它由“ link ”(5940)和“ maknode ”(7467)调用，调用参数是一指向内存“ inode ”的指针。

该目录项的16个字符复制到结构“ u.u_dent ”中，然后再写入目录文件。(注意，u.u_dent的先前内容将成为该目录文件最后一项的名字。)

此过程假定此目录文件已被搜索过，已为该目录文件分配“ inode ”，并且“ u.u_offset ”的值已设置为适当值。

19.8 maknode(7455)

“ core ”(4105)、“ creat ”(5790)和“ mknod ”(5966)以第2个参数为1调用“ namei ”，在其执行中发现指定名字的文件并不存在。在此之后，它们即调用本过程。

19.9 unlink(3510)

本过程实现系统调用“ unlink ”，它在目录结构中删除
用皆被删除时，该文件本身也将被删除。

3515：搜索指定名字的文件，若其存在，则返回指向其

3518：解锁父目录。

3519：对文件本身取得其内存“ inode ”指针。

3522：除超级用户外，不能对目录实施“ unlink ”(解除

3528：以“ inode ”指针值为0重写该目录项，亦即使该

3529：使“ link ”(连接)数减1。

注意，这里并不减小目录文件的长度。

19.10 mknod(5952)

本过程实现同名系统调用，这只能由超级用户执行。正
的那样，此系统调用的主要功能是为特殊文件创建“ inode

“ mknod ”也解决了“ 目录来自何方？”的问题。传送给
置“ i_mode ”，而且对其不进行任何修改和限制。(请与cre
这是使一个“ inode ”能够具有目录标志的惟一方法。

对于这种情况，传送给“ mkond ”的第3个参数值一定

中(这对于特殊文件而言是合适的)，如果此值非0，那么“ bmap ”会不加检查地接受它(6447)。在5969行之前插入下面一条测试语句可能更妥当一些，这样就不致于无限地依赖超级用户的绝对正确性：

```
if (ip→i—mode  &(IFCHR & IFBLK!)=0)
```

19.11 access(6746)

此过程由“ exec ”(3041)、“ chdir ”(3552)、“ core ”(4109)、“ openl ”(5815、5817)、“ namei ”(7563、7664、7658)调用，其功能是检查对一个文件的存取许可权。第 2个参数应当取下列3种值之一，它们是：“ IEXEC ”、“ IWRITE ”和“ IREAD ”，它们的8进制值分别是：0100、0200和0400。

6573：如果此文件位于安装为“ 只读 ”的一个文件系统卷中，或者该文件是一个执行程序的正文段，那么对此文件就不允许进行写操作。

6763：除非在3个“ 权限 ”组中至少有一组表示该文件是可执行的，否则超级用户不能执行该文件。在任一其他情况下，超级用户的存取操作都能得到许可。

6769：如果此用户不是该文件的属主，则将“ m ”右移3位，以便按组权限进行检查。如果此用户亦非同组用户，则再将“ m ”右移3位；

6774：将“ m ”与该文件的存取权限进行比较。

注意，这里可能会遇到一种异常情况：如果一个文件的“ 方式 ”是 0077，这表示文件主不能对其进行任何存取操作，但是其他任一用户则可进行所有存取操作。这种情况可以满意地得到改变，其方法是：

```
;
列语句：
>i_mode)==0)
```