HOPE
中国科学院希望高级电脑技术公司

UNIX

UNIX SYSTEMV
高级编程指南

- 本书介级UNIX内核与用户程序间的接口—系统调用
- 3500行的C代码例子
- 高级编程技巧与方法

Macro JRoch Kind著
陈捍东　编译

UNIX系统高级编程指南

陈振东 编译

中国科学院希望高级电脑技术公司

一九九一年元月

■ 北京市新闻出版局
准印证号：3155—90155
■ 订购单位：北京8721信箱资料部
■ 邮　　码：100080
■ 电　　话：2562329
■ 传　　真：01—2561057
■ 乘　　车：320、332、302路
车至海淀黄庄下车
■ 办公地点：希望公司大楼一楼
往里走101房间

前　言

本书的内容是Unix系统调用——介于Unix内核和在其上运行的用户程序之间的接口。那些只同命令，如外壳、文件编辑程序、和其它应用程序打交道的人也许乎无需知道系统调用，但关于系统调用的透彻的知识对Unix程序却是必须的。系统调用是进入内核工具如文件系统、多任务机制和进程间通信原的唯一途径。

系统调用定义了UNIX是什么。任何别的东西——子程序如命令——都是建立于此基础之上的。许多这些新的高水平程序产品使Unix名声大振，而这些程序也可以在任何现代操作系统上进行编程。当人们把Unix描述为精巧、简单、有效、可信赖和可移植时，人们指的不是命令（有些命令根本不属此类），而是指内核。

学习Unix 系统调用有多困难呢？当我1973年第一次开始编程时，那是一点都不难。当时的Unix——及其程序员使用手册——仅是现在的规模和复杂度的零头，那时的使用手册里没有任何编程的例子，但所有的源代码都是联机可用的并且象外壳程序或观察系统调用是怎样工作的编辑器这样的程序当时都是很容易读懂的。也许最重要的是，那时周围有更多有经验的人可请教。甚至Dennis Ritchie和KenThompson，Unix的发明者，也会抽出时间帮助我。

今天有志气的Unix程序员面临着比我当时更艰巨的挑战。Unix传播得如此广泛以至专家不愿接近它了，许多运行Unix的计算机只许使用目标码，所以源代码是不可得的。当今的系统调用是1973年的两倍，而使用手册的质量与由Ritchie和Thompson亲手写所有系统调用书面记录的时期相比已明显下降了。

本书的目的是让任何有经验的程序员象我当初一样容易地学习Unix 系统调用，然后巧妙方便地应用它们。本书包括3500行的C代码例子。我不仅从战术（怎样使用系统调用）上写，也试图包括进战略（为什么和什么时候使用它们）。并且书中有很多非正式建议，这些建议是基于我过去十二年里Unix编程的经验提出的。

当今Unix有许多版本，其中最重要的有五种：

1．SystemV，AT&T的最新版本，其系统调用是SystemⅢ系统调用的超级集，

2．SystemⅢ，先于System V（System IV未发行过）。

3．Version 7，来自贝尔实验室研究机构的最新版（1979年）。此版已不再销售了，但它却形成了许多现行销售的Unix派生版的基础，它可能是被理解得最好的AT&T版本，因为大多数Unix书籍都是关于Vevsion 7的。

4．Berkeley 4.2BSD，来自加利福利亚大学，是Version 7的拓宽的变种，它的许多系统调用似乎是Version 7和SystemⅢ的融合，但却有名成熟的差别，新增加60多条系统调用。

5．Xenix，Microsoft公司的产品，最新版，基于SystemⅢ，本书称之为Xenix Ⅲ。虽然Xenix只是许多有竞争力的商品化系统中的一种，但我把它单列出来加以强调，因为它是流行的是微机版本，而且——大等重要的是——它由

IBM PC／AT 支持，这几乎就保证了运行XenixⅢ的计算机比运行别的组合版本的计算机更多。

写出能运行于这三种版本之上的Unix程序是很理想的，除我特别注明外，这本书中所讲的适用于SystemⅢ，SystemⅤ，和XenixⅢ——并适于许多它们之中任一种的重新组合的商品名（如PC/IX和UnipPus）。除XenixⅢ少数无意义的系统调用外，所有这三种版本的系统调用都包括在本书内。我也给出了写可移植到Vevsion 7和Berkeley 4.2上BSD的程序的提示。本书不含那60个左右4.2BSD独有的系统调用，但本书多数内容也可用于该版本。不在AT&T许可的系统范围内如Idris，Coherent，和Regulus等是Unix的精确的替代，本书也适于它们，但我未写入关于它们的具体的资料。总之这本书将帮助你学习编写Unix的任何版本的程序——一旦明白了概念，你将很容易找到具体的细节。

看本书时你也想把你的Unix手册放在近旁，那样你就可以检查我所讲的与你的具体系统手册上的差别。另外，本书也使你熟悉手册的结构和措词，从长远观点看，使用手册是你的主要参考，所以无论如何你迟早要与它讲和。

一个叫作／usr／group的Unix工作机构提出了一个Unix标准，包括系统调用的详细说明。除了一个系统调用*lockf*（记录上锁）外，它实质上是SystemⅢ的一个子集（而且因此也是SystemⅤ和XenixⅢ的子集），但它因太不完整而对可移植性无太大帮助——这个标准只能在各种不同版本一致处达成统一。具体地说，它略去了终端I/O的细节和所有进程间通信机制如消息，信号灯，和共享存贮。贯穿本书有此标准的参考和评价。

本书是高级Unix指南，读者应已熟悉Unix，至少作为一个用户，应会C程序语言（见第1章中进一步了解C）。第1章是关于核心工具的简介，并包括入我将在本书使用的术语。如果你还不对这些作好准备，请从我在书目中介绍的一两本书开始。

我已把系统调用按功能分组，第2到第4章描述文件和终端I／O的系统调用。第5章是关于进程（多任务）的。第6，7章是关于进程间通信，第8章是关于信号，我将不宜放在别处的系统调用归入第9章。附录A是进程属性的完整清单，附录B概括了例子中使用的标准子程序，我把它们包括进来是为方便那些想在床上或飞机上看本书而又没带Unix使用手册的读者们。

我推测很多读者都不会按顺序读此书，特别是那些已是Unix程序员的读者，书中的很多部分在各部都可找到所需。对那些想更有条理地学习Unix的读者，我依次列出了一些文献，把未介绍到的主题的参考压缩到最小范围。我没有完全删除书目，因为顺序读本书的读者也会愿意指引到相互参照。但在任何情况下，往前看的进度只会在第一次看时更快些，而不会影响对后面章节的理解。开始时认真阅读第一章会有很大帮助。

本书是操作系统技术方面的高水平读物。它可作研究生的教材。除了理论准则外，本书对于任何想真正了解在实际中事情是怎么完成的也是很有价值的。自然，它比教课书上讲的要复杂，但有关关于Unix——的简洁的简便性和有效性的极好的融合——是很难用几句概括抓住的。你需了解细节才能欣赏到Unix对操作系统技术的贡献。

我在第2章到第9章后编写了些练习，几乎所有练习都需要进行Unix编程。如果有个可供使用的Unix系统且响应时间也足够快，那么除少数例外，这些练习应只花几个小

时就可完成。如果能在代码通读时观察这些程序是最好的了（在排错和测试以后）。Unix是如此丰富，以至于两个程序员对同一问题的解法大相径庭。分享这些不同的解法至少会同只用一种方法解同样的有益。虽然我是本书的唯一作者，但Unix却不是我设计出的。多年来，我的许多贝尔实验室的同事赐教于我，尤其是Bill Burnette, Rudd Canaday, Don Carter, Ted Dolottn, Alan Glasser, Rich Graveman, Dick Haight, Paul Jensen, John Linderman, Terry Lyons, John Mashey, Dennis Ritchie, Bill Roome, Ken Thompson, Larry Wehr, 及Peter Weinberger。他们中的任何人都与此书无任何直接关系；所有的观点——和难免的错误——都是我的，但我的技术和方法的成功都源于他们。

Brian Kerrighan，也应列入前面的名单，从一开始就与本书有关，他详细地阅读了我的原稿和计划，他帮我同Prentice Hall出版社订了合同，并看了最后的手稿。

Microsoft公司慷慨地提供了一套完整的Xenix Ⅲ的资料，在此对他们的帮助深表感谢。由于他们的帮助，使我能把最新资料编入，这对大批使用IBM PC／AT的程序员大有裨益。

我编写并在PC/IX下调试了本书的大部分程序，PC/IX是由Interactive System公司将System Ⅲ移植到IBM PC／XT上的一个版本。Brian Lucas和他的同事们值得为我曾用过的最有效、可靠和配有最完善资料的Unix系统而受到称赞。它在5000美元一台的XT机上运行得惊人的好。

我对Patvicia Henry，我的编辑，和她的在Prentice Hall的下属为顺利快速出版此书而付出的辛劳表示感谢。同他们一起工作是令人愉快的。

最后，得感谢Dennis Ritchie和Ken Thompson，他们向一代程序员表明复杂是可以克服。许多人为Unix奉献了多年，但Dennis和Ken的贡献奠定了基础，明了的基本原理使Unix保持为简巧的操作系统。

Macro J Roch kind

目　　录

第一章　基本概念

1.1　介绍

本章带你快速浏览一遍Unix内核提供的工具。我们将不过多涉及通常随Unix而来的用户程序（命令），如*ls*，*ed*，和*sh*。关于这些的讨论远离了本书的宗旨。我们将也不过多涉及内核的内部（如文件系统是怎样实现的）。

我们将在定义之前用一些术语，如进程，因为我们假定你已大致知道其含义了。如果有过多的东西你听着生疏，那么你应在继续看之前先进一步熟悉 Unix。如果你不知道进程是什么，那么你必须更进一步熟悉Unix，一本好的可读的书是 Kernighan 和 Pike 写的《Unix 编程环境》（《*The Unix ProgrammingEnvironment*》）。书目中列出了一些其它书也可供你参考。我们假定你已会C 编程了；如果不会，也会有许多关于C的书。

1.2　文件

有三类Unix文件：普通文件，目录，和特殊文件

1.2.1　普通文件

普通文件含有数据字节，它们被装在一线性数组中。任何字符或字符序列都是可读或可写的。读写都始于文件指针指向的字节位置，文件指针可被置为任何值（甚至超出文件尾）。普通文件存在磁盘上。

向一个文件中间插入字节（把文件分为两半）或从文件中删除字节（消除间隔）都是不可能的。随着字节被写到文件尾，文件变得越来越大，写一次大一字节。一个文件可被删为零字节长度，但却不能缩为中间大小2。当要进行任何这些不可能的文件时——例如，在文件编辑中——你只能是写3个全新的文件。这样也较安全。

两个或多个进程可以同时读写同一文件。其结果依赖于各个输入／输出请求发出的顺序，一般而言结果不可预测。虽然过去一直是非有效机制（见2.5），但直到最近，Unix 仍未能提供控制并发访问的有效机制。如今一些Unix 版本提供文件封锁和信号灯（见第七章）。

普通文件没有名字，它们有叫做i-数的号。i-数是i-节点数组索引，保存在盘上每一含有Unix文件系统的区域的前面。每个i-节点含有一个文件的重要信息。有趣的是，这些信息既不包括名字也不包括数据字节。它包括如下内容：文件类型（普通，目录，或特殊）；连结表（很快将解释）；专有的用户和组标识符；三类访问允许，对所有者，组，及其它；字节大小；最后存取，最后修改时间，及状态变化（当i-节点本身最后被修改时）；当然，还有指向含有文件内容的磁盘块的指针。

1.2.2　目录

由于用i-数指示文件不方便，就提供了目录以便用名字。实际上，目录几乎总是用于存取文件。i-数只是当文件系统遭损坏后修复时才用。

每个目录有一个两列的表，名字占一列，它的对应i-数占另一列。一个 名字／i-节点对叫一个连结。当Unix内核被告知用名字存取文件时，它自动地在目录中查找i-数。然

后它找到对应的i-节点，i-节点含有更多的文件信息（如谁可存取该文件）。如果数据本身将被存取，那么i-节点告诉到盘上何处找到数据。

目录实际上作为普通文件存贮，但在i-节点中把它们标记为目录。所以，在一个目录中对应于一个具体名字的i-节点可能是另一个目录的i-节点，这使得用户用Unix著名的分级结构来组织他们的文件。一个路径如*memo/july/smith*指示内核去获取当前目录的i-节点以便找到它的数据，在那些数据字节中找到*memo*，取相应i-数，获取那个i-节点以找到*memo*目录的数据字节，在数据字节中找到*july*，取相应i-数，获取i-节点以找到*july*目录的数据字节，找*smith*，最后取出*memo/july/smith*相联系的相应的i-节点。

循着一条相对路径（始于当前目录的路径），内核怎样知道从哪儿开始呢？内核只是简单地追踪每一进程的当前目录i-数。当一进程改变其当前目录时，它必须提供一个到新目录去的路径。这条路径产生一个i-数，然后这个i-数做为当新目录的i-数存起来。

一条绝对路径由一个/开始并始于根目录。内核只简单地把i-数2保留给根目录。在第一次构造文件系统时这就建立了。有一个系统调用用于改变进程的根目录（对i-数为非2的目录），但很少这样做。

因为目录的两列结构直接由内核使用(内核只在极少数情况下关心文件的内容)，又因为非法目录能很容易地毁坏整个Unix系统，所以，虽然一个有适当允许的程序可读一目录，但一个程序（既使是高级用户运行的）不能向一目录写入。相反，程序使用一个特殊的系统调用集修改一个目录。总之，唯一合法操作是增加或删除一个连结。

两个或多个连接可能处于相同或不同的目录里，使用相同的i-数。这意味着同一文件有不止一个名字。因为用给定的路径存取文件时只能找到一个i-数，所以不会模棱两可。i-数也许是通过另一路径找到的，但这无关紧要。当一个连接从一个目录里删除后，并不是立即就知道i-节点和相关数据字节是否可抛弃。这就是为什么i-节点含有一个连结计数器。删除一个i-节点的连接只是连接计数器减少一个数；当计数器减为零时，内核才放弃这个文件。

没有结构上的原因解释为什么目录不能象普通文件那样有多重连接。但由于这样做会使扫描整个文件系统的命令的编程复杂化，所以内核禁止这样做。

1.2.3 特殊文件

一个特殊文件或是某类设备（如磁带驱动器或通信线）或是个FIFO（先进先出队列），FIFO是用于进程间传输数据的一种机制，我们将在这里回顾设备特殊文件，在1.8中阐述FIFO。

设备特殊文件有两类：块和字符。块特殊文件遵循一特别模式：该设备有一固定大小块的数组（一般每块为512字节），并有一个用作高速缓存的内核缓冲区池以加速输入/输出。字符特殊文件完全不必遵循什么规则。它们可做很小量（字符）或很大量（磁盘磁道）的输入/输出。

同一种物理设备既可有块特殊文件又可有字符特殊文件，实际上，对磁盘就常常是这样。普通文件和目录由文件系统通过块特殊文件存取，以受高速缓存之益。有时，主要是在数据库应用中，需要更多的直接存取。数据库管理系统可以完全避开文件系统而用

一个字符特殊文件存取磁盘文件（但与文件系统用的区域不同）。许多Unix都有系统一字符特殊文件用于此目的，在进程地址空间和磁盘空间之间用DMA（直接存贮器存取）直接传输数据，其结果是运行情况有极大的改进。另一好处是较好的错误判别，因为高速缓存不碍事。

特殊文件有一i-节点，但该i-节点不指向磁盘上任何数据字节。相反地，这一部分i-节点含有一设备号。它是一个表的索引，该表用于内核查找一个叫做设备驱动程序的子程序集。

当执行一个系统调用在特殊文件上执行一操作时，一个适当的设备驱动程序就被调用了。随后发生的就完全由设备驱动程序决定了；由于设备驱动程序运行于内核，并且与用户进程不同，它能存取——或修改——内核任何部分，任何用户进程，计算机本身的任何寄存器或存贮器（如段寄存器）。添加一新设备驱动程序到内核是较容易的，这就提供了一种除了仅仅做新输入／输出设备接口外还可做别的事的方法。最流行的办法是把Unix 当作黑盒子来作其设计者从不打算让它作的事。例如，文件和记录封锁可以用（已经用）伪设备驱动程序实现。

1.3 程序和进程

程序是存于磁盘上普通文件中的指令和数据集。在其i-节点中该文件被标明为可执行的，并且文件内容按内核建立的规则排列（内核关心文件内容的另一情形）。

用户可用选择的任何方法创建可执行文件。只要内容遵循了规则并且文件被标明为可执行的，程序就可执行。实际上，多数用户这样做：首先，把用某种程序语言写的（一般为C）源程序敲入一个普通文件（经常指的是文本文件，因为它被编成文本行了）。其次，创建称为目标文件的含有源程序的机器语言解释的另一普通文件。这个工作由编译程序或汇编程序（它们本身也是程序）完成。如果这个目标文件是完整的（没有丢失的子程序），那么它就被标记为可执行的并且如果已标记了就可运行。如果不完整，则就用连结程序（Unix 行话称之为“装载程序”）来把该目标文件同以前建立的其它目标文件连接在一起，这些目标文件可能是从一称为库的目标文件集中取出的。除非连接程序找不到它要找的东西，其输出是完整的可执行的。

为了运行一个程序，首先要请求内核创建一新进程，它是程序在其中执行的环境。一个进程用三个段组成：指令段3，用户数据段和系统数据段。程序是用于初始化指令和用户数据段的。初始化之后，进程和它运行的程序之间不再有任何固定的联系。虽然现代的程序员通常不修改指令，但却右实修改数据。另外，进程可获得不出现于程序中的资源（更多的内存，打开的文件，等等）。

几个并发运行的进程可以是由同一程序初始化而来。但这些进程之间都没有功能的联系。内核可以设法让这些进程共享指令段以节省存贮，但由于这些段是可读的，所以涉及到的别的进程察觉不到共享。

一个进程的系统数据包括各种属性，如当前目录，打开文件描述符，累计 CPU时间，等等。本章后面几节将涉及这些问题。一个进程不可以直接地存取或修改它的系统数据，因为它在进程地址空间之外。相反，有各种不同系统调用用于存取或修改属性。

内核替当前执行进程创建一个进程，当前执行进程变成为新子进程的父进程。子进程继承了大部分父进程的系统数据属性。例如，如果父进程有任何文件打开着，子进程也将让它们打开着。正如我们将看到贯穿本书，这种继承性对Unix的操作是绝对重要的。

1.4 信号

内核可以给进程发信号。信号可由内核自己产生，从一个进程发向它自身，发自另一进程，或替用户发。

一个源于内核的信号的例子是段违反信号，当一个进程企图访问其地址空间以外的存贮区时发此信号。一个信号由进程发向自身的例子是闹钟信号：进程设置时钟，当闹钟停止后，该信号就发送。由一个进程向另一进程发信号的例子是终止信号，当几个相关联的进程中的一个进程决定终止整个进程家族时发此信号。最后，源于用户的信号是中断信号，用户按中断键（通常是DEL）时发送给用户创建的所有进程。

大约有19种信号（某些Unix 版本有或多或少）。除一种信号（取消信号，常常是致命的）外，对所有信号而言，进程在收到信号后可控制所发生的事。它可接受系统设置的动作，其结果是进程的终止；它可以忽略信号；或者它可捕捉信号并在信号到来时执行子程序。信号类型（从1到19的整数）作为参数被传送给该子程序。但没有直接的方法使子程序确定是谁送的信号。当信号一处理了程序返回时，进程由断点恢复执行。

有两个内核来定义的信号，它们由应用程序用于自己的目的。

1.5 进程-标识符和进程组

每一进程都有一进程-标识符，它是个整数。任何时候它们都保证是唯一的。除一个进程外，其余每个进程都有一父进程。这个例外是进程0，它由内核自己创建和使用，用于交换。

一个进程的系统数据也记录其父-进程-标识符，它的父进程的进程-标识符。一个进程因父进程终止而成了孤儿进程，它的父-进程-标识符就变为1。这是初始化进程（init）的进程-标识符，该进程是所有其它进程的祖先。换句话说，初始化进程抚养所有孤儿进程。

有时程序员选择实现一个象一组相关进程这样的子系统以代替一个单个进程。例如，一个复杂的数据库管理系统可能被分为几个进程以获得额外的磁盘输入/输出并行。Unix内核允许这些相关进程组成一个进程组。

进程组成员之一是进程组领导。组中每一成员把进程组领导的进程-标识符作为它的进程-组-标识符。内核提供了一个系统调用可向指定进程组的每一成员发信号。典型地，它可用于把进程组作为一个整体而终止，但任何信号都可用这种方法传送。

任何进程都可以从其进程组中辞去，或通过将其进程-组-标识符变成与其进程-标识符相同的方法而成为进程组领导，然后产生子进程以使新进程组完美。所以一个单个的用户可以运行，比如说，由10个进程组成的，比如说，三个进程组。

一个进程组可以有一个控制终端，它是进程组领导打开的第一个终端设备。通常，这个控制终端是用户进程由其装入的终端。当一个新进程组形成后，新进程组中的进程不再拥用控制终端。

终端设备驱动程序可能发送由终端来的中断，退出，和挂起信号给每一进程，该终

端是这些进程的控制终端。除非采取了预防措施，这样做，如挂起一个终端，将终止用户的所有进程。为防止此事发生，进程可设法忽视挂起信号（这就是*nohup*命令所做的事）。

当进程组领导因某种原因终止后，除非被捕捉到或被忽视，挂起信号就发向具有相同控制终端的所有进程，并也终止它们。这一特性使不能物理挂起的硬件设备能象那些物理挂起的硬件设备那样动作。这样，当一用户注销后（终止外壳，它通常是进程组领导），所有的东西都为了下一用户而清除，就象用户真的被挂起一样。

总之，有三种与每一进程相关联的进程-标识符：

进程-标识符	唯一定义该进程的正整数。
父-进程-标识符	该进程的父进程的进程-标识符。
进程-组-标识符	进程-组领导的进程-标识符。如等于进程-标识符，则该进程是进程组领导。

1.6 允许

用户-标识符是一与口令文件（*/etc/passwd*）中用户注册名相联系的正整数。当用户注册时，*login*命令把此标识符作为第一个创建的进程，注册外壳的用户-标识符。此外壳的子孙进程继承这个用户-标识符。

用户也被组织成组（不要与进程组搞混淆了），组也有标识符，叫组-标识符。用户的注册组-标识符由口令文件中取得并且当作他或她的注册外壳的组-标识符。

组定义在组文件里（*/etc/group*）。注册后，用户可改变到他或她是其成员的另一组里，这改变处理请求的进程的组-标识符（通常是外壳，通过 newgrp 命令），之后，该组-标识符由所有子孙进程继承。

这两个标识符被称为真实用户-标识符和真实组-标识符，因为它们代表了真实用户，注册的人，与每一进程相关联的还有另两个标识符：有效用户-标识符和有效组-标识符。这两个标识符通常与相应的真实标识符相同，但它们也可不同，正如我们将很快看到的那样。现在，我们假定真实的和有效的标识符是相同的。

有效标识符总是用于决定允许，真实标识符用于计数和用户-到-用户的通信。一个标明用户的允许；一个标明用户的身份。

每一文件（普通，目录，特殊）在其i-节点中都有一所有者用户-标识符和一所有者组-标识符。这个i-节点中还含有三类三比特位的允许(一共9比特位)。每类中有一位用于读允许，一位用于写允许，一位用于它执行允许。如果许可则允许位为1，否则为0。三类中，一类用于所有者，一类用于所有者组，一类用于其它（公用）。下面是比特位分配情况（比特位0是最右位）：

Bit	*Meaning*
8	owner read
7	owner write
6	owner execute
5	group read
4	group write
3	group execute
2	others read
1	others write
0	others execute

允许位常用一个八进制数指定。例如八进制数775意味着所有者和组的读，写和执行允许，其它只有读和执行允许。ls命令可象*rwxrwxr-x*这样显示这类允许组合；用二进制表示是111111101；用八进制将是775。

允许系统决定一个给定的进程是否可在一给定的文件上执行期望的动作（读、写或执行）。对于普通文件来说，其动作的含意是明显的。对目录来说，读的含意是明显的，因为目录存于普通文件中（例如，ls命令读一目录）。目录上的"写"允许意味着能发一系统调用修改目录（添加或删除一个连结）。"执行"允许意味着能在路径中使用目录（有时称为"寻找"允许）。对特殊文件，读和写意味着能执行*read*和*write*系统调用。如果有什么事隐含的话，这由设备驱动程序的设计者决定。特殊文件上的执行允许是毫无意义的。

允许系统用下面的规则决定允许是否被许可。

1．如果有效用户-标识符是零，则马上许可允许（有效用户是超级用户）。

2．如果进程有效用户-标识符和文件用户-标识符相匹配，则所有者类的比特位用于查看动作是否被允许。

3．如果进程有效组-标识符和文件组-标识符相匹配，则组类比特位将被使用。

4．如果用户-标识符和组-标识符都不匹配，则进程属于"其它"且第三类比特位将被使用。

有其它的一些动作，它们是可被称为"改变i-节点"的只有所有者或超级用户才可做的动作。这包括改变文件的用户-标识符和组-标识符，改变文件的允许，和改变文件的存取和修改时间。作为一种特殊情况，文件的写允许可把它的存取和修改时间设置为当前时间。

偶尔，我们想让一个用户暂时承接另一用户的特许权。例如，我们执行*passwd*命令改变口令时，我们想让有效用户-标识符是*root*（根）（超级用户的传统注册名）的标识符，因为只有根才能向口令文件写入。这可能通过把*root*（根）变成*passwd*命令的所有者而完成（也即，含有*passwd*程序的普通文件），然后置位*passwd*命令的i-节点中的另一允许位，此位叫置-用户-标识符位。执行一个该位已置位的程序可把有效用户-标识等改变为含有此程序文件的所有者。由于它们是有效的而非真实的决定允许的用户-标识符，这就允许用户暂时承接其它用户的允许。置-组-标识符的用法类似。

因为两类用户标识符（真实的和有效的）都由父进程被继承到子进程，所以用置-用户-标识符的功能可能用一个有效用户-标识符运行很长时间。

潜在的漏洞存在着。假设你这样做：拷贝*sh*命令到你自己的目录下（你将是这个拷贝的所有者）。然后用*chmod*置位置-用户-标识符位，并用*chown*把文件的所有者改为*root*（根）。现在执行你的*sh*拷贝，并利用*root*（根）的特权！幸好，这个漏洞很久以前就堵住了。如果你不是超级用户，则改变文件所有者会自动清除置-用户-标识符位和置-组-标识符位。

1.7 其它进程属性

除已提到的外，还有几个别的有趣的属性记录在进程的系统数据段里。进程打开的每个文件（普通文件，目录，或特殊文件）都有一个打开的文件描述符（从0到19的整

数），进程创建的每个通道有两个打开的文件描述符。子进程不继承其父进程的打开表件描述符，而是继承它们的拷贝。虽然这样，但它们仍是指向同一系统范围打开文件文的索引，处于其它东西之中的该表意味着父、子进程共享同一文件指针。

进程优先级由内核的调度程序使用。任何进程都可通过*nice*系统调用降低其优先级，超级用户进程可通过同一系统调用升高其优先级。从技术上讲，nice设置一个叫作"nice"值的属性，它只是实际优先级计算机中的一个因素。

进程的文件大小极限可以（并且通常是）低于系统范围限制，这将防止糊涂的或未开化的用户写出一个乱撞的文件。超级用户进程可以升高它的极限。

如果跟踪标志已置，则子进程在接受到信号时仅仅停止而不做通常应做的事（如终止）。它是可重新执行的，它的父进程将由入到子进程地址空间并可能重启动之。跟踪标志只由调试程序使用（如*adb*和*sdb*）。这一设施的使用复杂得可怕；值得庆幸的是，由于调试程序已写出来了，所以我们大多数人不必担心使用它了。

1.8 进程间通信

在System Ⅲ之前的Unix系统中，进程可通过共享的文件指针，信号，进程跟踪，文件和通道来相互通信。对System Ⅲ，又新添了FIFO队列（命名为通道）。对System V，新添了信号灯，消息，和共享存贮。正如我们将看到的，这九种机制中无一完全令人满意。这正是为什么有九种的原因。

共享的文件指针极少用于进程间通信。理论上讲，一个进程可以把文件指针指向一个文件的某些虚位置，而另一进程可查明这个指针指向了何处。这个位置（一个界于，比如0到100间的数）即为通信数据。因为进程必须相关联才能共享文件指针，所以它们也可只用通道。

有时当需要一个进程激励另一个进程时要用到信号。例如，无论何时一个打印文件被假脱机输出，打印假脱机程序都可向实际打印进程发信号。但信号不能传递足够的信息以有助于大多数应用。另外，一个信号中断接收进程，使得程序比接收进程就绪时就可通信的更复杂。信号主要用于终止进程。

用进程跟踪，父进程可以控制子进程的执行。由于父进程可以读，写子进程的数据，所以它俩可以自由通信。进程跟踪只由调试程序使用，因为它对一般应用而言太复杂且不安全。另外父进程和子进程也可由一通道通信。

文件是进程通信的最简单的形式。例如，人们可以用运行*ed*的进程写文件，而用运行*nroff*的进程把它格式化。但是文件在两进程并发运行时不方便，原因有二：第一，读进程可能跟踪写进程之外，看到一个文件结束标志，从而认为通信已结束（这可由一些巧妙的编程处理）。第二，两进程通信越久，文件变得越大。有时进程通信达数日或数周之久，传输数百万字节数据。这将很快使文件系统资源耗尽。

使用一空文件作为信号灯也是一种传统的Unix技术。这将利用Unix创建文件的方法上的一些特性。有关细节在2.5中给出。

通道解决文件的同步问题。通道不是一类文件；虽然它有i-节点，但它无连结。读写一个通道有点像读写文件，但二者有着显著区别：如果读在写之前，则读进程只是等待更多的数据。如果写在前可读，写进程就一直等到读进程有机会赶上为止，所以内核