UNIX技术·培训丛书之五

UNIX高级编程

上海电子计算机厂科技服务部培训部

UNIX 高级编程

赵东风　译
胡传国　校

上海电子计算机厂科技服务部培训部

前 言

本书的主题是 UNIX 系统调用——UNIX 内核及其顶层运行的用户程序之间的接口。仅仅与命令，如 Shell、正文编辑程序及其它应用程序打交道的用户不需要十分了解系统调用。但是对于 UNIX 程序员来说，彻底了解系统调用则是最基本的。系统调用是访问诸如文件系统、多任务机构及进程间通信原语之内核设备的唯一方法。

系统调用定义了 UNIX。其它的任何东西——例行子程序和命令——都是建筑在这一基础之上的。而这些高级程序的新产品的不断推出使 UNIX 名声大震。这些高级程序也可以在其它任何时髦的操作系统上进行编程。当人们把 UNIX 描绘成一个精致、简便、可靠、有效的操作系统时，人们所指的绝不是它的命令（有些命令根本不具有这些特点），而是它的内核。

学习 UNIX 系统调用有多么困难呢？当人们在一九七三年第一次开始编制 UNIX 程序时，并不感到十分困难。那时的 UNIX 及其程序员手册的大小和复杂程度只是当今的几分之一。那时，手册中没有任何编程的实例，但是所有的源代码都排列成行，阅读程序十分容易，就象 Shell 或编辑程序了解系统调用如何工作一样容易。

今天 UNIX 如此广泛的传播，以致不需要什么专家在你身边。大多数运行 UNIX 的计算机都只提供目标代码，命令的源代码是得不到的。现在系统调用的数量是 1973 年时的二倍。而手册的质量比 UNIX 发明者当时对系统调用所做的书面记录的质量有着明显的下降。手册中到处充斥着象下列这样奇异的段落：

If the set-user-ID mode bit of the new process file is set (see chmod (2)), exec sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and the real group-ID of the new process remain the same as those of the calling process.

（若设置了新进程文件的"设置-用户-标识符"方式位，exec 就把该新进程的有效用户标识符设置成该新进程文件的拥有者标识符。与此相仿，若新进程文件的设置组标识符的方式位已被设置，则该新进程的有效组标识符被设置成该新进程文件的组标识符。新进程的实际用户标识符和实际组标识符保持和调用进程的一样。）

一个有经验的程序员第一次看到这样的段落时，还能明白它是什么意思；但新手一定会感到困惑不解。直至今天，这种状况还没有改变。

本书的宗旨就是要使任何富有实践经验的程序员都能轻松自如地学习掌握 UNIX 的系统调用，并能灵活地应用它们。本书中包含了累计 3500 多条 C 语言程序代码的实例。它不仅论及了编程的手法（如何使用系统调用），而且还试图包括一些策略方面的问题（为什么使用它们以及何时使用它们）。另外，作者还根据自己几十年从事 UNIX 程序设计的经验，提供了一些信息和劝告。

当今，UNIX 有许多版本，下列五种是最为重要的：

1）系统 V，AT&T提供的最新版本。它的系统调用是在系统 III 的基础上又增加了一些

系统调用的超集（superset）。

2）系统Ⅲ，它是在系统Ⅴ之前问世的一种版本（系统Ⅳ没有发行）。

3）版本7，它是贝尔实验室研究机构于1979年推出的最新版本。该版本发行不久，便成为其它许多也广泛流传的 UNIX 派生系统的基础。该版本大概是 AT&T 最易被人们理解的版本，因为大多数关于 UNIX 的书籍都论及了版本 7（version7）。

4）美国加州大学研制的Berkeley 4.2 BSD，它是版本7的修改和扩充。它的许多系统调用看起来就象是版本 7 和系统Ⅲ的混合体。但是又有着许多不同之处。它增加了 60 多条系统调用。

5）Xenix，它是Microsoft公司的产品。它的最新版本是在系统Ⅲ的基础上推出的，本书中称为 Xenix3。尽管 Xenix 只是许多具有竞争性的已经商品化的操作系统之一，我还是把它单独列出加以特别强调。这是因为它是最流行最广泛使用的微型计算机的操作系统版本。

最为重要的是，它得到了IBM公司的支持，所有IBM PC AT都运行该系统。这就保证了运行 Xenix3 的计算机的数量比运行其它版本的计算机的总和还要多。

希望大家编写在这五种版本上都能运行的 UNIX 程序。除非另加说明，本书中的所有内容都适用于系统Ⅲ、系统Ⅴ和 Xenix3，以及众多采用其它商标名称的系统（如 PC／IX 和 UniPlus 等）。它们只不过是这五种版本之一的改装本而已。除了 Xenix3 中的一些无意义的系统调用外，本书包含了这三种版本中的全部系统调用。另外，还对如何编写可以移植到版本 7 和 Berkeley4.2 BSD 上的程序给予了必要的提示。本书中没有包含近 60 条只适用于 4.2 BSD 的系统调用，但几乎本书中的所有系统调用都适用于 4.2 BSD。因为非基于 AT&T 的系统，例如 Idris、Coherent 和 Regulus 等是 UNIX 的准确的衍生系统，所以，本书同样也适用于它们。无论如何，本书将有助于你学习如何编写任何 UNIX 版本的程序——一旦您明白了概念，要再进一步深入掌握其细节就不是一件很难的事了。当您阅读本书时，最好把您的 UNIX 手册放在身旁。这样，您就可以核对一下，看看（在书中）所提供的与您的特殊系统所采用的方法有何不同。此外，您还可以熟悉 UNIX 手册的组织结构和语言表达方法。从长远来说，手册总是您的基本参考书，因此，迟早您总要与手册相一致。

一个称作／usr／group 的 UNIX 工业组织提议了一个包含系统调用规范的 UNIX 标准，实质上，它是系统Ⅲ（因此也是系统Ⅴ和 Xenix3）的一个子集，另外再增加一个（用于记录锁定）lockf 系统调用。但是，它太不完善，对可移植性没有多大帮助——因此，（标准化）委员会只能一致认为它在各种版本上已经是统一的。特别是，它省略了终端 I／O 和较新的进程间通信机制——例如消息、信号灯和共享存贮器——的细节。本书通篇都给出了参考书和对标准（化）的评述。

本书是一本"高级"的 UNIX 书籍，您肯定已经熟悉 UNIX，至少作为一个用户，您对 C 程序设计语言一定比较了解（读完本书后，您将更进一步了解它）。本书的第一章对内核的设施进行了概括性的温习，并介绍了本书中所要用到的技术术语。如果您感到您对第一章还准备不足，则请您从选读一、二本本书最后推荐的参考书目中的书籍着手。

本书按照系统调用的功能把它们编排成功能组。第二到第四章描述用于文件和终端 I／O 的系统调用；第五章是关于进程（多任务处理）方面的描述；第六、七章关于进程间通信；第八章是关于信号的描述；我把不适合放在任何一组的系统调用安排在第九章中。附录 A 是一个完整的进程属性表；而附录 B 汇总了实例中所用到的所有标准子程序。把它们包含在本书中，完全是为那些喜欢在床头阅读或在飞机上阅读而身边无 UNIX 手册的读者

着想的。

考虑到许多读者，特别是那些已经是 UNIX 程序员的读者，可能会不逐章地阅读本书。本书中安排了一些互相参照条目，对于那些按顺序学习 UNIX 的读者，则按顺序进行介绍，最大限度地减少读者参阅尚未介绍过的科目，但是，还不能完全避免这种现象的出现，因此，按顺序进行学习的读者有时也会发现他们也使用了相互参照条目。但是，在任何情况下，首次阅读时，在不影响阅读进度的前提下"偷阅"总是可以遵循的。当您开始阅读本书时，首先仔细认真地阅读第一章将使您收益非浅。

本书也可作为大学高年级或毕业班学生学习操作系统教程的一本很好的辅助教材。学生们不仅可以从本书学到基础理论，而且能够从中了解到实际生活中，工作是如何进行的。自然，UNIX 实际所进行的大部分工作并不象教科书所宣扬的那样完美无瑕。但是，关于 UNIX 的一些东西——例如关于它的简易性与强大的功能的完好结合——笼统地加以评价，很难引起人们的注意。关于 UNIX 对操作系统技术的贡献，您需要阅读有关详细的评价材料。

在第二到第九章的最后部分，还给出了一些练习。几乎所有这些练习都要求进行一些 UNIX 编程。只要 UNIX 系统能够很快通过，且有足够响应时间，这些练习题，除个别的外，只要花费几小时就能完成。最好在对这些程序进行查错和测试后再通过会话作代码 (reviewed in code-walk-through sessions) 复查。UNIX 是多产的，以致两程序员解同一问题可能提出两种完全不同的解答方法。共享这些解答方法至少象获得一个解答一样有价值。

多年来，作者得到了贝尔实验室的许多同事们的指教，特别是 Bill Burnette、Rudd Canaday、Don Carter、Ted Dololtta、Alan Glasser、Rich Graveman、Dick Haight、Evan Ivie、Paul Jensen、John linderman、Terry lyons、John Mashey、Dennis Ritchie、Bill Roome、RenThompson、larry Wehr 和 Peter Weinberger 等同事给了很多帮助。这些同事对本书没有直接做过任何工作，其中的见解和难免的错误都属于笔者本人，但是作者在本书中所传递的许多技术和策略思想是他们开创的。

Brian Kernighan——他的名字也可以列入上述名单之列——从一开始就参与了本书的工作。是他，仔细地审阅作者最初的建议和构思；是他帮助作者与出版社签订了合同；也是他审阅了作者的最终手稿。

Microsoft 公司非常慷慨地为我提供了 Xenix3 的全套资料。对他们的无私帮助表示由衷的感谢。作者已把最新的信息写入本书中，它将给使用 IBM PC AT 的无数程序员带来巨大的好处。

作者是在 PC/IX 系统下写完本书的手稿，并对大多数程序进行了测试的。PC/IX 是 Interactive Systems 公司把系统 III 移植到 IBM PC XT 上的一种版本。Brian Lucas 和他的同事们完全应该为我曾使用过的有效、可靠、资料齐备的 UNIX 系统 I 而感到自豪。它在 $5000XT 上的性能是令人惊奇的。

最后还要感谢 Dennis Ritchie 和 Ken Thompson 两位先生，他们向一代程序员表明：复杂性是可以避免的。多年来，还有许多其他人为 UNIX 的发展做出了贡献。但是，是 Dennis 和 ken 的扎实的理论基础和清晰明了的表达使 UNIX 继续成为最精致的操作系统。

编者

91.5

目　录

第一章　基本概念

1.1 引论

本章将带你到 UNIX 核心所提供的设备上进行一次旋风式旅行。我们既不想过多地论及通常伴随 UNIX 的用户程序（命令），例如 ls，cd 和 sh，关于它们的讨论超出了本书的范围；也不想涉足于核心的内部（例如文件系统是如何实现的）。

我们这里所说的旅行的含义是温故知新。我们在定义诸如进程一类术语之前便使用它们，因为我们假定您对这些术语的含义已经有了粗略的了解。若您对这些术语感到很生疏，则请您在着手研读本书之前，先熟悉一下 UNIX（如果您还不知道什么叫做进程，那么您的确需要先熟悉 UNIX！）。Kernighan 和 Pike 所著的《UNIX 编程环境》一书便是可供你阅读的佳作。本书中的参考书目列出了一些可供你选读的书籍。同样，我们还假定您已经懂得如何用 C 语言编程。若您还不知道如何用 C 语言进行程序设计，则同样有许多关于 C 语言的书籍可供您研读。

1.2 文件

UNIX 文件有三种：普通文件、目录文件和特别文件。

1.2.1 普通文件

普通文件含有几个组织成一个线性数组的数据字节。可以读／写这些数据字节的任一字节或字节序列。读和写从文件指针所确定的字节位置开始。它可以被设置成任一值（甚至是文件尾以外的值）。普通文件存放在磁盘上。

不能够在一个文件中间插入字节（扩展文件），或者从一个文件中间删除字节（闭合间隙）。因为字节都是被写到一个文件的尾部，所以文件逐渐变大，写入一次扩大一个字节。一个文件可以被缩短到零字节长度，但不能被缩小到中等大小（而在 Xenix3 中用 chsize 系统调用就可以做到这一点）。当需要进行这种不能进行的操作时，例如在正文编辑中，则要写一个全新的文件。这也是可靠的方法。

两个或多个进程可以同时读写同一个文件。其结果取决于各个 I／O 请求产生的顺序，而且通常是不可预测的。直到目前为止，UNIX 还没能提供控制同时访问文件的有效手法，虽然已经有了一些无效的手段（请阅 2.5 节）。UNIX 的某些版本现在提供了文件锁定和信号灯的技巧或手段（见第七章）。

普通文件没有文件名，而有称作 i 号的数。i 号是 i-节点数组中的下标，存放在装有 UNIX 文件系统的每个盘区的前端。每个 i-节点都含有关于文件的重要信息。有趣的是，该信息既不包括文件的名称，也不包括数据字节。它包括：文件类型（普通文件、目录文件和特别文件）；（需要简要说明的）链数；文件拥有者的用户和组标识符；三组访问权限；文件拥有者的访问权限，文件所属组的组内成员的访问权限及其它用户的访问权限；以字节表

示的文件大小；最后一次访问的时间和最后一次修改的时间以及状态的变化（当 i-节点本身是最后一次修改时）；当然还包括指向磁盘中含有文件内容的磁盘块的指针。

1.2.2 目录文件

由于通过 i-number 访问文件较麻烦，因此提供了允许使用（文件）名字的目录文件。在实践中，目录总是用来存取文件。只有当一个文件系统被破坏后修补该文件系统时才使用 i 号。

每个目录由一个两列组成的表格所构成。一列中是文件名字，另一列里是其对应的 i 号。一个名字／i-节点对称作一个链（link）。当告诉 UNIX 核心要通过文件名来访问文件时，它就自动地到目录中查找 i-number（i 号）。然后便可获得其对应的 i-node（i-节点），它含有更多的关于该文件的信息（例如关于谁可以访问它的信息）。如果要存取数据本身，则 i-node 告诉你到磁盘的何处可以找到该数据。

实际上目录也是作为一个普通文件来存贮的，只是在 i-node 中把它标记成目录，因此，对应于某个目录中特定（文件）名字的 i-node 也可能是另一目录的 i-node。这就使得用户能够把他们的文件安排成层次结构，在层次结构这方面 UNIX 是非常著名的。一个路径，例如 memo／july／smith，指示核心取得设置其数据字节的当前目录的 i-node，在这些数据字节中间找到 memo，取对应的 i-number，获得定位 memo 目录数据字节的 i-node；找到数据字节中的 july，取对应的 i-number，获得定位 july 目录数据字节的 i-node，找到 smith，最终，取得对应的 i-node 就是和 memo／july／smith 相关联的 i-node。

遵循相对路径（即从当前目录开始的路径），核心是怎么知道从哪里开始呢？它只是简单地跟踪每个进程的当前目录的 i-number。当一个进程改变其当前目录时，它必须提供一条通向新目录的路径。该路径引导到一个 i-number，然后该 i-number 就作为新的当前目录的 i-number 保存起来。

一个绝对路径的首部带有一个斜杠／，并且从根目录开始。核心只为该根目录保留 i 号 2。这是在首次构造文件系统时建立的。有一个系统调用去改变一个进程的根目录（至不为 2 的 i 号），但这种事很少进行。

因为核心直接使用目录的双列结构（核心很少注意文件的内容），以及一个无效目录（文件）极易破坏整个 UNIX 系统，所以不可以把程序（即使是超级用户所运行的程序）写入目录。尽管它如果获得许可，就可以读一条程序。而是一个程序可以使用专门一组系统调用来修改目录。最后，唯一的合法行为是增加或去掉一个链。

在相同或不同的目录中，可能有两个或两个以上的 link 涉及同一个 i-number，这就是说：同一个文件可能有一个以上的名字。但在通过给定路径访问文件时不会产生二义性的现象。因为只能找到一个 i-number。当然，也可以通过其它路径找到它（一个 numbers），但那是与之不相干的。然而，当一个 link 从一个目录中拿掉时，尚不能立即知道 i-node 及相关的数据字节是否也要去除掉。这就是为什么 i-node 要包含一个链计数（link count）的原因。把一个 link 移到一个 i-node 仅仅使链计数减 1。当链计数减到零时，核心便放弃该文件。对于目录和普通文件来说，不能有多个 link，这并不是结构上的原因。但是，它使扫描整个文件系统的命令的编程工作复杂化，因此核心剥夺了它的存在。

1.2.3 特别文件

特别文件是某些设备类型（例如磁带驱动器或通信线路）或者是 FIFO（先进先出队列）。它是用于进程之间传送数据的机制。在此，我们将首先重温一下设备特别文件，在 1.8 节中将再讨论 FIFO。

有两种设备特别文件：块型（block）特别文件和字符型特别文件。块型特别文件遵循一特定模式：设备包含一组固定长度的字符块（通常每块 512 字节）和一个核心缓冲池。这些核心缓冲区用作高速缓冲器，以提高 I/O 的速度。字符特别文件全然不遵循任何规则，它们可以以最小的字符块为单位进行 I/O，或者以最大的块（磁盘的磁道）为单位进行 I/O。

同一个物理设备可以同时有块型和字符型两种特别文件。实际上，对于磁盘来说才真正这样。文件系统通过块型特别文件访问普通文件和目录，以提高高速缓冲存贮器的效率。在应用数据库的初始阶段有时更需要直接存取方式。数据库管理程序可以绕过整个文件，使用字符特别文件存取磁盘（但不是存取文件系统所使用的同一盘区）。大多数 UNIX 系统都备有能够用 DMA（直接内存存取）直接在进程地址空间和磁盘之间传送数据的字符特别文件。其结果使性能有了数量级的改善，另一收益是使错误检测效果更好，因为高速缓冲存贮器不妨碍它。

特别文件有一个 i-node，但是磁盘上没有 i-node 所要指向的任何数据字节，而是 i-node 部分含有一个设备号（device number）。这个设备号就是放入表中的索引（号），核心用它来寻找称作设备驱动程序的例行子程序集。

在为完成特别文件上的操作而执行系统调用时，就要调用适当的设备驱动子程序。然后将会发生什么完全取决于设备驱动程序的设计者。因为驱动程序在内核部分运行，且不作为一个用户进程，所以它可以存取——可能还能够修改核心的任意部分、任一用户进程以及计算机本身的任何寄存器或内存贮器（例如段寄存器）。把新的设备驱动程序添加到内核部分是相对容易的，这样就提供了一种手段。利用这种手段除了能和各种新的 I/O 设备接口外，不可以做许多事情。这是让 UNIX 做一些它的设计者从不企望它做的一些事情的最流行的方法。例如，可以用一种伪设备驱动程序来实现文件和记录锁定。

1.3 程序和进程

程序是保存在磁盘上普通文件中的指令和数据的集合。在文件的 i-node 中标明该文件是可执行的。文件的内容按照核心所制定的规则排列（这是核心关注文件内容的另一种情况）。

用户可以用他们选择的任何方法建立可执行的文件，只要文件的内容遵守核心所制定的规则，并且标明是可执行的，程序就能运行。在实践中，大多数用户选用下列方法：首先，把用某种程序设计语言（通常用 C 语言）编写的源程序键入普通文件（因为文件排列成一行行的文本，所以常常把它称作文本文件（text file））中。其次，建立另一个称作目录文件的普通文件，它包含将源程序翻译成机器语言的工作。这一工作由编译程序或汇编程序（它们本身就是一种程序）来完成。若该目录文件是完整的（没有丢失子程序），则标记成是可执行的，且可正常运行。否则使用链接程序（linker）（在 UNIX 术语中称作装入程序

(loader)）把该目标文件和先前建立的其它文件，也可能和取自称作文件库（libraries）的目标文件集的其它文件装配起来。除非链接程序（linker）找不到它所要找的东西，否则它的输出就是完整且可执行的。

为了运行一个程序，内核首先请求建立一个新的进程，该进程就是程序在其中执行的环境。一个进程由三个段组成：指令段、用户数据段和系统数据段。指令和用户数据的初始化工作需要使用程序。初启以后，进程和它所运行的程序之间就不再有任何固定的联系。尽管现代程序员在正常情况下不需修改指令，但数据还是要修改的。此外，进程还可能获得程序中没有叙述的资源（更多的内存及打开的文件等）。

几个同时运行的进程可以由同一程序初启。然而，这几个进程之间并没有什么函数关系。核心（kernel）可以通过安排这些进程共享指令段来节省内存。但是，由于这些指令段是只读指令，因此，它所包含的进程不能觉察出这种共享。

一个进程的系统数据包括诸如当前目录、打开文件说明符和 CPU 累计时间等属性。这些将在本章后面的一些章节中加以讨论。一个进程不能直接存取或修改它的系统数据，因为这将超出它的地址空间。而是有各种各样的可以存取或修改属性的系统调用。核心建立一个进程，代表当前可执行的进程，该进程将成为新的子进程的父进程。子进程继承父进程的大多数系统数据属性。例如，如果父进程有一个文件是打开的，则该文件对于子进程也是打开的。这种继承性对于 UNIX 操作而言是绝对基本的东西。从本书的通篇中均可看出这点。

1.4 信号

核心可以给一个进程发送一个信号。该信号可以是核心自己生成的，可以是该进程本身发送的，可以是别的进程发送的，也可以是以用户的名义发送的。

核心生成的信号的一个例子是当一个进程试图访问超出其地址空间的内存时发出的违背分段原则信号。由进程发送给自己的信号的一个例子是闹钟信号；一个进程设置时钟，当进行报时时，便发出信号。一个进程发送给另一进程的信号的例子是当几个相关进程中的一个进程决定结束整个系列时发送的终止信号；最后，用户生成的信号的例子是中断信号，当用户按中断键（一般为 DEL）时，用户给所有进程发送中断信号。

大约有 19 种信号（某些 UNIX 版本多几种，有的版本少几种）。对于 kill 信号以外的所有信号而言，一个进程能够控制接收该信号后所发生的事情。它可以接收一个缺省动作，其结果是终止该进程。它可以忽略该信号，或者可能捕捉到信号，并且在信号到达时执行一个子程序。信号的类型（整数 1～19）作为一个参数传送给这个子程序。但是，这个子程序没有直接方法来确定是谁送的信号。当信号处理子程序返回时，进程从中断处重新执行。

有两个信号没有被核心定义。应用程序为了其自身目的可能使用这两个信号。

1.5 进程标识符（process-ID）和进程组

每个进程都有一个进程标识符（process-ID），它是一个正整数。在任何情况下，都必须确保它是唯一的。每个进程都有且只能有一个父进程。0 进程例外，它是由核心自己用以交换而建立和使用的。

进程的系统数据也记录它的父进程的进程标识符——parent-process-ID。如果一个进

程因为其父进程已经结束而成为孤儿进程，则它的父进程标识符（parent-process-ID）变为1。这个就是初启进程（init）的进程标识符（process-ID），它是其它所有进程的祖先。换言之，初启进程收养所有的孤儿进程。

有时程序员实现一个子系统作为一个相关进程组，而不是作为一个单一进程。例如，一个复杂的数据库管理系统可能被分割成几个进程，以增加磁盘I/O的附加并发性。UNIX核心允许把这些相关进程组织成一个进程组。

进程组成员中的一个成员为组长（group leader）。进程组的每个成员都把组长的进程标识符（process-ID）作为它的进程组标识符（process-group-ID）（请注意，不要和一个进程的组标识符（group-ID）相混淆。见1.6节）。核心提供一个系统调用，把一信号发送给指定进程组的每个成员。这个方法典型地被用来结束作为一个整体的整个组。但是任何信号都可以用这一方法传播出去。

任何一个进程都可以从其进程组退出，把它的进程组标识符（process-group-ID）变成和它自己的进程标识符（process-ID）相同，从而成为自己组的组长，然后在其新组的周围大量孵生子进程。因此，譬如说，一个单一用户就可以分成三个进程组运行十个进程。

一个进程组可以有一个控制终端（control terminal），它就是进程组长首先打开的那个终端设备。正常情况下，用户进程的控制终端通常是用户在它上面登录的那个终端。当一个新的进程组组成后，该新组中的进程就不再有控制终端。

终端设备驱动程序把来自一个终端的中断信号、退出信号和挂起信号发送给把该终端视为控制终端的每个进程。例如，除非采取预防措施，否则，挂起一个终端将终止用户的所有进程。为了防止这种事情发生，一个进程可以设法把挂起信号忽略（这就是nohup命令要做的事）。

当一个进程组长由于某个原因而终止时，将给所有带有同一控制终端的进程发送一个挂起信号，该挂起信号也将终止它们这些进程，除非该挂起信号被捕捉或被忽略。这一特性使得硬接线终端（hard-wired terminal）极易被挂起，终止其它进程。这样，当一个用户注销（logs off）时（终止shell时，它通常是进程组组长），就把下面用户的所有东西都清除掉了，就象用户实际被挂起一样。

总之，有三种进程标识符（process-ID）与每个进程相关：

进程标识符（process-ID）	只标识这一进程的正整数
父进程标识符（parent-process-ID）	该进程的父进程的进程标识号。
进程组标识符（process-group-ID）	进程组长的进程标识号。如果它等于进程标识符（process-ID），则该进程是组长。

1.6 权限

用户标识符（user-ID）是一个与口令文件（/etc/passwd）中用户注册名相关联的正整数。在用户注册时，login命令就把这个ID，即建立的第一个进程的用户标识符（user-ID）变成注册shell，由这个shell衍生的所有进程都继承这个用户标识符（user-ID）。

用户也同样编成一个一个的组（注意不要和进程组相混淆）。该组也有标识符（ID），称作组标识数（group-ID）。从口令文件（password file）处取一个用户注册的组标识号，

并把它变成用户的注册外壳的组标识符.

组是在组文件（/etc/group）中被定义的. 在注册时，用户可能变成另一组的成员，这就改变了处理请求的进程的组标识号（通常是 shell，通过 newgrp 命令），然后这个组标识符被所有繁衍的进程所继承.

以上两个标识号（ID）称作实际用户标识符（real user-ID）和实际组标识符（real group-ID）. 因为它们是实际用户即注册人的代表. 还有另外两个与每个进程相关的 ID：有效用户标识符（effective user-ID）和有效组标识符（effective group-ID）. 通常这两个 ID 和其对应的实际 ID 是相同的. 但是，它们也可以不相同. 正如我们不久将看到的那样. 但是现在我们假定实际 ID 和有效 ID 是相同的.

有效 ID 已经被用于确定是否许可；实际 ID 用于计数和用户对用户的通讯. 一个表明用户是否被允许；一个表明用户的同一性.

每个文件（普通文件、目录文件和特别文件）在其 i-node 中都有一个文件拥有者用户标识符（owner user-ID）和一个文件拥有者组标识符（owner group-ID）. i-node 还含有三个三位的权限集（总共 9 位）. 每个权限集中有一位是读权限位，一位是写权限位，还有一位是执行权限位. 若权限位的值为 1，则表示允许；若权限位的值为 0，则表示不允许. 有三个集：一个文件拥有者集，一个文件拥有者组集和一个共用（其它用户）集. 下面是位的分配（0 位是最右边的一位）.

位	意　义
8	文件拥有者读
7	文件拥有者写
6	文件拥有者执行
5	组　读
4	组　写
3	组执行
2	其他用户读
1	其他用户写
0	其他用户执行

权限位经常用八进制数来表示. 例如，八进制数 775 表示允许文件拥有者和（用户）组读、写和执行，只允许其它用户读和执行. ls 命令将把这些权限的集合表示成 rwxrwxr-x；用二进制数表示为：111111101；用八进制数表示，将为 775.

权限系统决定一个给定进程是否能够在给定的文件上完成所要完成的动作（读、写、操作）. 对于普通文件，这些动作的含义是显而易见的；对于目录文件，读的含义是很清楚的. 因为目录存放在普通文件中（例如，ls 命令读一个目录），允许在目录文件上"写"意味着能够发出系统调用来修改目录文件（增加或减少 link）. 允许"执行"意味着能够在路径上使用目录（有时称作"检索"权限. 对于特别文件而言，允许读和写意味着能够执行 read（读）和 write（写）系统调用. 如果有隐含的话，它隐含什么就取决于设备驱动程序的设计者. 特别文件上的执行权限是无意义的.

权限系统用下列算法来决定是否允许：

1) 如果有效 user-ID 为零，则立即允许（有效用户为超级用户）.

2) 如果进程的有效user-ID和文件的user-ID相匹配，则用文件拥有者的位集来确定动

作是否被允许。

3) 如果进程的有效group-ID和文件的group-ID相匹配，则用组的位集来判断动作是否被允许。

4) 如果user-ID和group-ID都不相匹配，那么该进程为"其他用户"。并用第三位集来判定动作是否被允许。

还有其它工作，可以把它称作"改变 i-node"，但这些工作只能被文件拥有者用户和超级用户来完成。这些工作包括改变文件 user-ID 或 group-ID。改变文件的权限（位）及文件的存取时间或修改时间。作为一种特殊情况，允许在文件上写就允许把存取时间和修改时间设置成当前时间。

有时我们需要一个用户暂时优先于其它用户。例如，当我们执行 passwd 命令来改变我们的口令时，我们总是希望有效用户标识符（user-ID）是 root（超级用户的传统注册名）的有效用户标识符，因为仅有 root 可以写入口令文件。这项工作通过下列方法来完成：首先把 root 变成 passwd 命令的文件拥有者（owner）（即含有 passwd 程序的普通文件），然后打开 passwd 命令的 i-node 中的另一个称作 set-user-ID 位的权限位（permission bit）。执行带有该位的程序，把有效用户标识符变成含有该程序的文件的拥有者（用户标识符）。因为它是一个决定权限的有效的用户标识符（user-ID）而不是实际用户-ID，所以它允许一个用户暂时获得别人的允许。以类似的方法使用 set-group-ID。

由于两个 user-ID（实际 user-ID 和有效 user-ID）都是由父进程遗传给子进程，因此能够很长时间地用 set-user-ID 性能运行带有有效 user-ID 的文件。

这里有一个潜在的漏洞。假定你进行下列工作：把 sh 命令拷贝到你自己的目录中（你将是该拷贝的拥有者）然后用 chmod 打开 set-user-ID 位，用 chown 把该文件的拥有者（owner）位变成 root。现在执行你的 sh 拷贝，并具有 root 优先级！幸好，这个漏洞早就被堵塞了。如果你不是超级用户，改变一个文件的拥有者（位）就自动地清除 set-user-ID 位和 set-group-ID 位。

1.7 其它的进程属性

除了已经提及的这些属性外，在进程系统数据段中还记录着一些其它有趣的属性。

进程已经打开的每个文件（普通文件、目录文件或特别文件）都有一个打开文件描述符（0～19 整数），进程建立的每个管道（pipe）文件都有两个打开文件描述符。子进程不继承其父进程打开文件描述符，而宁可继承它们的拷贝。尽管如此，它们是放入同一系统范围（system wide）打开文件表中的索引。除其它事情外，它还意味着父和子共享同一文件指针。

内核调度程序使用进程的优先级。每个进程都可以通过系统调用来提高它的优先级。从技术上来说，nice 设置一个称作 nice value 的属性，它只是计算实际优先级的一个因子。

一个进程的文件长度范围可以（通常是）小于系统范围的限值。这是为了防止不文明的用户或混乱的用户编写失控的文件。超级用户进程可以提高它的限值。

若跟踪标记（trace flag）是打开的，子进程在接收到信号时就不进行它的正常工作（例如终止）而只是停下来。可以推测，它的父进程将在该子进程的地址空间四周游荡并可能重新起动子进程。跟踪标记（trace flag）仅仅被调试程序（如 adb 和 sdb）使用。这一机制使

用起来十分复杂。幸运的是，因为排错程序已经编写好，我们大多人不必为使用它而操心。

1.8 进程间通信

在系统Ⅲ（SYSTEMⅢ）以前的 UNIX 系统中，进程可以通过共享文件指针、信号(signal)、进程跟踪（process tracing）、文件和管道相互通信。系统Ⅲ中又增加了 FIFO(命名的 pipes)，系统 V（System V）中又增加了信号灯（semaphores），消息（message）和共享内存等手段。正如我们将要看到的那样，这几个机制没有一个是完全满意的，这就是为什么一定要九个的原因。

共享文件指针很少用于进程间的通信。从理论上来讲，一个进程将把文件指针置于文件中的某个假定位置，然后第二个进程将发现它指向那里。单元位置（譬如说 0 和 100 之间的一个数）可以是一个通信数据。因为只有相关联的几个进程才能共享一个文件指针，所以它们还不如使用管道。

当一个进程正好需要指向另一进程时，有时就要使用信号（signal）。例如，每当一个打印文件假脱机输入输出时，打印假脱机输入输出程序就可能给实际打印进程发一信号。但是，在大多数应用中，signal（信号）不传递足够的有用信息。另外，signal（信号）还要中断接收进程，使程序设计比接收器就绪后就能获得通信时的（程序设计）更加复杂。signal(信号）主要被用于正好要终止进程的情况下。

用进程跟踪（process tracing）方法，父（进程）可以控制其子（进程）的执行。由于父可以读写子的数据，因此两者可以自由通信。进程跟踪只由排错程序使用，因为对于一般应用，它显得太复杂太不可靠。况且，父和子通过管道可以更容易地进行通信。

文件（File）是进程间通信的最通用的方式。例如，某人可以用进程运行 ed 来写一个文件，然后用进程运行 nroff 使其格式化。但是，如果两个进程同时运行，则文件（File）通信方式是不方便的。其原因有二：第一，读者的速度可能超过写者，读者看到文件尾(end of file）便认为通信已经结束（这可以通过巧妙的程序设计来加以处理）。第二，较长的两个进程的通信，使得文件变得较大。有时进程通信持续几天或数星期，传递数以百万计字节的数据。这将迅速消耗文件系统。

使用一个空文件作为信号灯（semaphore）也是一个传统的 UNIX 技术。它利用了 UNIX 建立文件方法中的某些特色。这将在 2.5 节中进行更详细的介绍。

pipe（管道）解决了文件的同步问题。pipe 不是一种文件类型。尽管它也有 i-node，无 link（链）连接在它上面。读写一个 pipe（管道）和读写一个文件有某些相同之处，但也有一些明显不同之处。若读者跑在写者的前头，读者就将等待写者以读到更多数据；若写者的速度远远超过读者，写者就处于睡眠状态直到读者赶上它，这样内核就没有太多的排队数据。最终，一个字节数据一旦被读入，它就一去不复返。因此，通过管道（pipe）连接的长运行进程不会填满文件系统。

shell 用户对管道 pipe 是很熟悉的。他们可以键入类似于下列的命令行，看一看他们有多少文件：

```
ls | ws
```