UNIX系统
高级程序设计

段小航　刘京志　王凌　编译

中国铁道出版社

(京)新登字 063 号

内 容 简 介

本书通过大量的实例讲述了 UNIX 操作系统的高级使用方法及其使用技巧和策略，对可移植性问题给出了建议性的忠告。全书共分九章：第一章讲述了基本概念；第二章介绍了基本的文件输入和输出；第三章介绍了高级的文件输入和输出；第四章讲述了终端的输入和输出；第五章讲述了有关进程的一些系统调用；第六章介绍了基本的进程间通讯；第七章讲述了高级的进程间通讯；第八章为信号；第九章讲述了其它常用的一些系统调用。附录 A 以表格形式列出了系统 V 的进程属性；书中用到的标准子例程在附录 B 中说明。

本书可作为大学高年级及研究生学习和掌握 UNIX 操作系统的参考书，也可作为研究单位和软件开发部门进行 UNIX 系统高级程序设计的参考资料和培训教材。

UNIX 系统高级程序设计

段小航 刘京志 [illegible] 译

中国铁道出版社出版、发行

(北京市东单三条[illegible]号)

责任编辑 殷小燕 封面设计 翟达

各 地 新 华 书 店 经 售

中国铁道出版社印刷厂印

开本：850×1168 毫米 1/32 印张：12 字数：315 千

1991 年 12 月 第 1 版 第 1 次印刷

印数：1— 3500 册

ISBN7-113-01176-4/TP · 118 定价：7.25 元

编译者序

UNIX 操作系统自从诞生以来，以其简洁、可移植性好、功能强和效率高等特点受到广泛的重视，已推广到从微型机到大型机的各种计算机上，并取得了巨大的成功。与 UNIX 标准化有关的各种国际组织，如/usr/group、UI(UNIX International)、X/open 和 OSF(开放软件基金会)等也正在积极从事 UNIX 的标准化工作。从长远来看，UNIX 的原理和思想对今后操作系统的发展将会起着深远的影响。

目前，见之读者的有关 UNIX 操作系统方面的书籍很多，但大多是讲述 UNIX Shell 层的使用、UNIX 的结构和算法及 UNIX 操作系统的组成。而涉及如何使用 UNIX 核心提供的功能，拓展其应用范围，以便充分发挥其潜能的书籍却不多。在大量的基于 UNIX 系统的应用开发中，尤其是高级的应用中，需要深入了解 UNIX 核心提供的接口系统调用及其有关问题。为此我们编译了 Marc J. Rochkind 的《Advanced UNIX Programming》一书。作者 Rochkind 从事 UNIX 程序设计已有十几年的历史，并在贝尔实验室工作过，具有丰富的实践经验。该书打破了 UNIX 有关书籍的常规体例，从用户的应用考虑，由浅入深、循序渐进地讲述了 UNIX 系统调用的使用技巧和策略。书中还配以大量和完整的实例，并对每个实例都有详尽的描述，使读者能通过上机实践来掌握和使用较复杂的系统调用。另外，在每章的最后都给出了本章所讲述的系统调用的可移植性建议和忠告，这对编写 UNIX 应用程序的开发人员来说是非常有益的。

在编译过程中，我们根据需要对原书结构做了某些调整并增加了一些新的内容，对书中所列实例都尽可能上机加以验证。如果

本书能对您的应用开发工作起到一定作用或帮助您解决了实际中所遇到的一些问题，我们将感到无比欣慰。

本书第一章至第四章由刘京志编译；第五章至第七章由段小航编译；第八、九两章、附录A和附录B由王凌编译。

限于编译者的时间和水平，书中难免有错误和不妥之处，恳请读者批评指正。

一九九〇年十一月

目　录

第一章　基本概念

1.1 引　言

为了能更好地理解本书所叙述的内容，本章我们将对 UNIX 所涉及的概念及提供的工具作一个简单但却是全面的介绍。本章介绍的内容不涉及太多的 UNIX 常用命令，如 ls，ed 和 sh，对这些命令的讨论不属于本书的讨论范围。我们也不涉及太多的 UNIX 核心内部的情况(比如文件系统是如何实现的)。

本章的介绍相当于对 UNIX 的基本概念做一次复习。我们假设读者已经粗略地知道了 UNIX 所使用的术语，比如进程的含义。如果读者对这些术语不熟悉，那么在阅读本书之前，需要先熟悉一下 UNIX(如果你连进程的概念都不知道，那么首先必须要熟悉一下 UNIX!)。这里推荐一本较好的书——《The UNIX Programming Environment》(UNIX 程序设计环境)，作者是 Kernighan 和 Pike。我们还假定读者已经懂得如何使用 C 语言编程，如果不会，也有许多有关 C 语言的书可读。

1.2 文　件

UNIX 中有三种类型的文件，即普通文件、目录文件和特别文件。

1.2.1 普通文件

普通文件用一线性数组实现，其内容为数据字节。文件中的任何字节或字节序列都可以读或写。读和写操作是从文件指针所指的字节位置开始进行的，文件指针可置成任何值(甚至超过文件的

末尾)。通常,普通文件存放在磁盘上。

一般说来,不能在文件的中间插入任何字节,也不能从文件的中间删除字节。当在文件末尾写上若干个字节时,文件变大,且一次只能写一个字节。文件可被缩短至0字节,但不能被缩小到任何中间大小[1]。当需要进行这些不可能的操作时,例如在正文编辑中,写一个全新的文件也许更安全一些。

两个或更多的进程可以同时读和写同一个文件。其结果依赖于各个输入/输出请求出现的顺序,且一般来说,这一结果是不可预测的。到1985年为止,UNIX仍未提供有效的机制来控制并发性的访问,尽管有一些无效的机制可供使用(见2.5节)。现在,UNIX的某些版本提供了文件锁定和信号灯装置(见第七章),用以改善对文件的并发访问问题。

普通文件没有名字,它们有个称之为i—number的编号,即i节点号。i节点号是到一个i节点(i—node)数组的索引。它存放在含有UNIX文件系统的磁盘区的前头。每个i节点都含有有关文件的重要信息。有趣的是,这一信息不包括文件的名字和数据字节,它只包括下列信息:文件类型(普通文件、目录文件和特别文件)、链接计数(在后面将简要说明)、文件主和同组用户的标识符(以下简称ID)、用于文件主、同组用户和其它用户的三组存取权限、以字节为单位的文件大小、最近的一次存取、修改及状态改变时间(当i节点本身最近一次被修改时),当然,还有指向含有文件内容的磁盘块的指针。

1.2.2 目　　录

由于通过i节点号来引用文件很不方便,所以提供了目录以便能用名字来引用文件。实际使用中几乎总是通过目录来存取文件的。只有当文件系统被破坏后,需要对其进行修复时才使用i节

〔1〕 在Xenix 3中可用chsize系统调用来做这项工作。

点号。

每个目录含有两列信息:一列表示名字,另一列表示它所对应的 i 节点号。一个<名字,i 节点>对被称为一条链接。当告诉 UNIX 核心要通过名字访问某个文件时,核心自动地在目录中查找,以便找出相应的 i 节点号。通过 i 节点号得到对应的 i 节点,该 i 节点含有有关文件的更多的信息(比如谁可以存取这个文件)。如果数据本身要被访问,则 i 节点告诉在磁盘的什么地方可以找到这些数据。

实际上,目录是作为普通文件存贮的。但在 i 节点中,它们被标记为目录。相对于某个目录中的一个特定名字的 i 节点可能是另一个目录的 i 节点。这就允许用户按照我们所熟知的 UNIX 层次结构来安排他们的文件。比如,memo/july/smith 这样一条路径。首先,UNIX 核心通过得到当前目录的 i 节点来寻找它的数据字节,从这些数据字节中找到 memo,取出对应的 i 节点号,根据这一 i 节点号得到相应的 i 节点,并从中找到 memo 目录的数据字节,再从这些数据字节中找到 july,取出 july 的 i 节点号,根据这一 i 节点号得到它的 i 节点,从中找到 july 目录的数据字节。最后,从这些数据字节中找到 smith,取出它的 i 节点,这就是与 memo/july/smith 相联系的 i 节点。

对相对路径来说(从当前目录开始),UNIX 核心是如何知道从哪儿开始呢?核心只是简单地为每个进程记录当前目录的 i 节点号。当一个进程改变它的当前目录时,它必须提供到新目录的一条路径。这条路径指向一 i 节点号,该 i 节点号作为新的当前目录的 i 节点号来保存。

绝对路径以"/"开头,以根目录作为起点。核心为根目录保留 i 节点号 2,这是在文件系统第一次构造时建立的。有一个系统调用可以改变进程的根目录(其 i 节点号不为 2),但很少这样做。

由于目录的两列信息结构由 UNIX 核心直接使用,而且因为一个无效目录能很容易地破坏整个 UNIX 系统,所以尽管在适当

的权限下可以读一个目录，但程序不能(即使是由超级用户运行的)向任一目录写入。程序可以通过一组专门的系统调用来对目录进行修改。总之，唯一合法的操作是添加或删除一条链接。

在相同或不同目录下的两个或多个链路可以引用同一个 i 节点号。也就是说，同一个文件可以有多个名字。当通过一给定路径访问一文件时，因为只能找到一个 i 节点号，所以不会产生二义性。这个 i 节点号也可以通过其它路径找到，但这无关紧要。可是，当从一个目录中删去一条链路时，是否 i 节点和与之相关的数据字节也被删除，还不能立即知道。这就是为什么 i 节点中含有一链接计数的缘故。删除连接于某个 i 节点的一条链路，只是把链接计数减 1，只有当计数减至 0 时，UNIX 核心才放弃这个文件。

在结构上没有任何原因能够说明为什么目录不能象普通文件那样也有多重链接。但有一点是可以肯定的，那就是目录的多重链接将使得扫描整个文件系统的命令的设计工作复杂化，所以UNIX 核心把它看作是非法的。

1.2.3 特别文件

一个特别文件或是某种类型的设备(比如磁带驱动器或通讯线)，或是一 FIFO(先进先出队列)。FIFO 是一种用于在进程间传递数据的机制。本节我们将先复习设备特别文件，在 1.8 节我们再复习 FIFOs。

有两种设备特别文件：块设备和字符设备。块设备特别文件遵循一特殊的模型：该设备含有一组固定大小的块(通常为 512 字节/块)，并且有一用于提高 I/O 速度的核心缓冲池作为高速缓冲存贮器。字符特别文件不遵循任何规则。它们可按非常小的块(字符)进行 I/O，也可按非常大的块(磁道)进行 I/O。

同一个物理设备既可以有块特别文件也可以有字符特别文件。事实上，磁盘就是这样一种设备。文件系统访问普通文件和目录是通过块特别文件来完成的，以得益于缓冲区的好处。有时，比

如在数据库应用中，需要更直接的访问，数据库管理系统可能完全绕过文件系统而使用字符特别文件去存取磁盘(但不是文件系统所使用的区域)。大多数 UNIX 系统都有一字符特别文件用于在一个进程的地址空间和磁盘之间使用 DMA(直接存贮器访问)直接传递数据，结果是成数量级地改善其运行性能。字符特别文件的另一个优点是能检测到更多的错误，因为缓冲区(高速缓冲存贮器)无法做到这点。

每个特别文件都有一个 i 节点，但它并不指向磁盘上的任何数据。相反，i 节点中包含的是设备号，这个设备号是核心所使用的一个表的索引，UNIX 核心利用此表可以找到该设备的设备驱动程序。

当一个系统调用在某一特别文件上执行一操作时，就引用相应的设备驱动子例程。至于会发生什么情况，则完全由设备驱动程序的设计者来决定。因为驱动程序是在核心内部运行的，而不是作为一个用户进程，因此，它可以访问，或许也可以修改核心的任何部分、任意的用户进程以及计算机本身的任一寄存器或内存(比如段寄存器)。把新的设备驱动程序加到核心相对来说比较容易，这不仅便于与新类型的 I/O 设备进行接口，而且还为做许多其它事情提供了便利。比如，文件和记录的锁定可以(并且已经)用一个伪设备驱动程序来实现。

1.3 程序和进程

程序是指令和数据的集合，它存放在磁盘上的一个普通文件里。在它的 i 节点中该文件被标记为可执行的，文件的内容是按照 UNIX 所设定的规则来安排的。

用户可以用任一种方法创建可执行文件。只要文件的内容符合 UNIX 的规则并且文件被标记为可执行的，那么就可以运行这一程序。在实际中，大多数用户是这样做的：首先，将用某种程序设计语言(通常为 c 语言)编写的源程序输入到一个普通文件中(通

常称作正文文件，因为它是按正文行的形式安排的）。然后，创建另一个称作目标文件的普通文件，它是由源程序转换过来的，含有转换好的机器语言。这项工作由编译器或汇编程序（它们本身也是程序）来完成。如果该目标文件是完好的（未丢失子例程），则就被标记为是可执行的，并可以运行。如果不完好，则使用连接程序（用UNIX 术语称做“loader”）把该目标文件与其它原先已创建的目标文件连接起来，或者与称为库的“目标文件集”中的文件连接起来。除非连接程序找不到所要找的东西，否则它的输出是完好的和可执行的。

为了运行一个程序，首先要求 UNIX 核心创建一个新进程，这是程序执行所需的环境。一个进程由三段组成：指令段[1]、用户数据段和系统数据段。程序用于初始化指令和用户数据。初始化后，进程和正在运行的程序之间就不再有任何固定的联系了。虽然现代程序设计人员一般不修改指令，但还要修改数据。另外，进程可以获得程序中所没有提供的资源（如更多的内存，打开文件等）。

几个同时运行的进程可以由同一个程序初始化得到，然而这些进程之间并没有功能上的联系。UNIX 核心能够通过安排这样的进程来共享指令段以节省内存，但因为这些段只可读，所以，参与的进程不能检测到这种共享。

一个进程的系统数据含有该进程的属性，比如当前目录、打开的文件描述字、累积使用的 CPU 时间等。在本章的后几节将讨论这些问题。进程不能直接存取和修改它的系统数据，因为系统数据放在进程的地址空间之外。相反，有各种系统调用可用来访问或修改这些属性。

通过核心为一个当前正在执行的进程创建一个进程，则当前正在执行的进程就变为新创建的进程的父进程。子进程继承了父

---

〔1〕 在 UNIX 术语中，“指令段”又叫做“正文段”，但为避免造成混淆，我们仍称之为指令段。

进程的大多数系统数据属性。例如，如果父进程有什么打开的文件，那么子进程也同样拥有这些打开的文件。正象我们在本书自始至终所看到的那样，这种继承方式构成了 UNIX 操作的基础。

1.4 信　　号

核心可以向进程发信号。信号可由核心本身产生，或从一个进程发送给它自己，也可由另外一个进程发送，或者代表用户发送。

核心产生信号的一个例子是段违例信号。当进程试图存取它的地址空间以外的存贮单元时，发送这一信号。进程向自己发送信号的例子是报警时钟信号。进程设置时钟，当时钟走完时，发送这一信号。从一个进程向另一个进程发送信号的例子是终止信号，当几个相关的进程中的一个进程决定终止整个进程家族时，发送这一信号。用户发送信号的例子是中断信号，当用户按下中断键(通常为 DEL 键)时，这一信号发送给他所创建的所有进程。

UNIX 中有大约 19 种信号(某些 UNIX 版本或多于 19 种，或少于 19 种，但相差不多)。对于所有这些信号(除了 Kill 信号，它是致命的)，当进程接收到信号时，它可以控制将要发生的动作。它可以接受缺省动作，使得该进程终止，也可以忽略该信号，或者捕俘这一信号并执行一个子例程。信号类型(从 1 到 19 的一个整数)作为一个参数被传递到该子例程。可是，没有任何直接的方法使子例程能确定是谁发送这一信号的。当信号处理子例程返回时，进程在断点处恢复执行。

有两种信号核心未定义。可以由应用程序根据其意图来使用。

1.5 进程标识符和进程组

每个进程都有一个进程 ID，用一个正整数表示。在任何时刻 ID 都是唯一的。除一个进程外，其它进程都有一个父进程，这个例外的进程就是进程 0，它由核心创建和使用，用于进行对换(swapping)。

进程的系统数据也记录了其父进程的 ID。如果一个进程由于其父进程终止而成为孤儿，则它的父进程的 ID 就变为 1。这是初始化进程(init)的 ID，它是所有其它进程的祖先。换句话说，初始化进程收养所有的孤儿进程。

有时，程序员选择一组相关进程代替一单一进程来实现某个子系统。例如，一个复杂的数据库管理系统可能要分成几个进程以获取对磁盘 I/O 的并行处理。UNIX 核心允许这些相关的进程组成一个进程组。

进程组中的一个成员为组长，其它成员都把组长的进程 ID 作为它的进程组 ID[1]。UNIX 核心提供了一个系统调用来给指定的进程组的每个进程发送信号。最典型的是用来终止作为一个整体的整个进程组。用这种方法也可发送其它任何信号。

任一进程都可脱离它的进程组。通过把它的进程组 ID 设置成它自己的进程 ID，使它成为自己这个组的组长，然后产生子进程形成新的组。例如，单个用户可以正在运行 10 个进程，而这 10 个进程又可以分成三个进程组。

一进程组可以有一个控制终端，该终端是由组长打开的第一个终端设备[2]。通常，用户进程的控制终端是用户注册进入的终端。当形成一个新的进程组时，该组内进程不再有控制终端。

终端的设备驱动程序向每个进程发送来自控制终端的中断、退出和挂起信号。除非采取预防措施，否则，挂起的终端将终止所有的用户进程。为了防止这种事情的发生，进程可以忽略挂起信号(由命令 nohup 完成)。

当进程组的组长由于某种原因终止时，将给在同一控制终端上的所有进程发送挂起信号，除非这些进程或者捕俘或者忽略这

---

〔1〕 不要同进程的组 ID 相混淆，见 1.6 节。

〔2〕 系统Ⅲ和系统Ⅴ手册有时引用 tty—group ID(该组长的进程 ID)代替一个控制终端。