操作系统系列丛书

Practical
UNIX Programming
A Guide to Concurrency,
Communication, and Multithreading

实用UNIX编程

（美） Kay A. Robbins
Steven Robbins 著

刘宗田 孙志勇 秦宗贵 等译

机械工业出版社
China Machine Press

Prentice
Hall

操作系统系列丛书

实用UNIX编程

（美） Kay A. Robbins
Steven Robbins 著

刘宗田 孙志勇 秦宗贵 等译

机械工业出版社
China Machine Press

本书提供了在UNIX环境下的编程技术，特别为进程管理、并发和通信中许多基本概念提供了编程指导。本书用大量实例和图示阐述了UNIX操作系统中的抽象概念，为UNIX C编程人员提供了很好的参考。本书可作为计算机专业本科生和研究生的教材或参考书，也可作为自学UNIX操作系统的参考书。

本书版权登记号：图字：01-1999-2346

图书在版编目(CIP)数据

实用UNIX编程/（美）罗宾斯（Robbins, K. A. ），（美）罗宾斯（Robbins, S.）著；刘宗田等译. - 北京：机械工业出版社，1999.10

(操作系统系列丛书)

书名原文：Practical UNIX Programming, A Guide to Concurrency, Communication, and Multithreading

ISBN 7-111-07381-9

Ⅰ.实… Ⅱ.①罗… ②罗… ③刘… Ⅲ.UNIX操作系统-程序设计 Ⅳ.TP316

中国版本图书馆CIP数据核字（1999）第34703号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码100037）

责任编辑：吴 怡

北京昌平第二印刷厂印刷 · 新华书店北京发行所发行

1999 年 10 月第 1 版第 1 次印刷

787mm × 1092mm 1/16 · 28.75 印张

印数：0 001-5 000 册

定价：38.00 元

译 者 序

对于初学者来说，理解操作系统的基本原理和各种抽象的术语是非常困难的。无论教科书的编写者怎样用心描述，无论课堂上的教师如何费力讲解，这些看不见摸不着的操作系统内部的运行机制总不能在学生的头脑中建立起清晰的概念。也许指导学生动手实现一个操作系统是让学生真正明了这些概念的最彻底的方法，但操作系统又不像其他小型的应用程序，可以在一个小时或者十几个小时内由三、五个学生分工完成。而且也很难把它分解成一个一个的互相独立的部分，通过逐个实现而完成总的实现。

特别是由于在操作系统中引入了多进程、多线程、并发以及资源共享等复杂机制，如何安全地实现和使用这些功能就成为非常重要的问题了。网络的出现又把操作系统的内涵极大地扩展了，从逻辑意义上讲，整个网络（包括Internet）上的任务协同与在一台物理机器上的任务协同变得等同了。程序员所设计的程序由以单任务模式为主转变为以多任务模式为主，以往无须考虑的问题现在必须认真对待了。也就是说，没有对操作系统内部机制的深入理解，程序员仅凭学会一两门编程语言对管理操作系统和开发应用程序是远远不够的。

UNIX操作系统是具有多进程、多线程、并发和资源共享机制的最流行的系统之一，几乎在各类机型上都有它的版本。学习UNIX编程不仅有现实的应用需要，而且具有典型意义。

本书既从基本概念上进行了系统的阐述，又从实验的角度考虑，设计了大量的有助于澄清概念的练习和例题。无论作为计算机专业教材还是作为自学参考书籍都是非常适用的。

本书由刘宗田、孙志勇、秦宗贵、谢志鹏、邵堃、姜川、刘莹、李连生等翻译，由刘宗田等整理，袁兆山等校。参加本书翻译和整理工作的还有李心科、陈凯明、邢大红、孙慧杰、潘飚、冯鸿、许东、姜桂华等。

由于时间和水平限制，难免有翻译错误或不妥之处，请读者批评指正。

1999年8月

前　言

计算机系统正迅速地由终端存取的大型单处理器主机向多处理器工作站的网络演化。类似并发、通信和多线程的思想已经从研究界走入了商界。应用程序员必须理解这些概念，而本书的目的正是为了使读者逐条地理解这些概念。

本书使用了非传统意义上的自己动手(hands-on)的方法。在操作系统传统意义上的“自己动手”方法中，程序员可以实现一个简单的操作系统或者修改一个现有的操作系统以增加功能。自己动手方法提供了对基本操作系统设计的一个深入的理解，但是这对于程序员来说是难以独立进行下去的。当这种方法被用在中等水平的大学时，讲师们将花费相当多的课时来论述实现的细节——而剩下较少的时间来全面地论述基本原理。此外，传统意义上的自己动手方法通常并不能提供关于高级的同步和通信概念的实际编程经验。另一个方法是理论上的表述，这种非实践性(hands-off)课程论述了更多的素材，但是并不能使读者在实践中深入地理解这些概念。

本书通过论述标准UNIX下的编程，进而在操作系统的自己动手和非实践性方法之间的沟壑上架起了一座桥梁。专业程序员可以独立地使用本书或是作为指南与其他参考书（如Stevens的《Advanced Programming in the UNIX Environment》）一起使用，从而获得对操作系统以及系统编程的更加全面的理解。学生可以将本书作为传统教科书（如Silbers © hatz和Galvin的《Operating Systems Concepts》或Tanenbaum的《Modern Operating Systems》）的伴侣手册使用，来学习操作系统。

练习和工程使得本书独具特色。实际上，本书是作为一本工程业务手册开始的。在初步展开之后，我们发现，进行工程所需的材料分散在许多地方——经常能在一些提供了许多细节但缺乏概念性纵览的参考书中找到。本书于是演化成了一本依赖于最新UNIX标准的独立参考书。

本书组织成4篇，每篇包含一些主题章和一些工程章。主题章有许多例子和形如“试一下这个”或“会怎样”的简短练习。主题章以一个或多个练习节结束。本书为进程管理、并发和通信中的许多基本概念提供了编程练习。这些编程练习满足了在传统科学课程中进行实验室实验的需要。为了完全地理解这些概念，我们有必要去使用它们。为了逐步展开，练习都进行了详细地说明，而且它们大多可以在100行代码内实现。

工程章通过开发更广泛的应用而合并了来自多个主题章的材料。工程在两个层次上起作用。除了阐明编程思想外，工程可使我们理解那些与应用有关的高级主题。这些工程分阶段设计，而且大多数完整实现有几百行长。由于大量的代码无需重写，程序员可以集中于理解概念而不是调试代码。为了简化编程，我们制作了可用于网络通信的库。

表I-1总结了本书的组织结构——15章总共分成4篇。主题章与工程章是相互独立的，读者在第一遍阅读本书时可以跳过工程章。

通览本书有多条路径。第1篇的3个主题章是本书其他部分必需的前提。在第1篇的主题章之后，读者可以以任何次序来阅读第2篇到第4篇，但稍后几章结尾处关于交互（如线程与信号如何交互）的讨论例外。

表I-1 本书结构表

篇	主题章	章号	工程章	章号
第1篇基础知识	并发	1		
	程序和进程	2		
	文件	3	工程：令牌环	4
第2篇异步事件	信号	5	工程：计时器	6
			工程：解剖shell	7
第3篇并发	临界区和信号量	8		
	POSIX线程	9		
	线程同步	10	工程：简化的并行虚拟机	11
第4篇通信	客户机-服务器通信	12	工程：Internet广播	13
	远程过程调用	14	工程：元组空间	15

我们假定本书的读者虽然是优秀的C程序员但未必是UNIX C程序员。读者应当熟悉C编程和基本数据结构。对于不熟悉UNIX的读者，附录A覆盖了程序开发的要点。UNIX程序员可能已懂得许多在第2章和第3章中可以找到的材料，但是这里的论述是很详细的，而且后面的工程将大量地依赖于它。

如果你未曾成功地开发一些使用这些概念的程序的话，作为读者，你不应当认为阅读了一章便等于是理解了。对于专业程序员，主题章结尾处的练习为该素材提供了一个小型的自己动手的专题论文。使用本书作为操作系统课程的讲师一般可在一个学期课程期间选择几个练习加上一个较大的工程。每个工程都有许多的变种，因此这些工程可以在多个学期中使用。

本书包含了许多标准函数的提要。指定函数的相关标准出现在提要方框的右下角。通常情况下，ISO C是POSIX.1和Spec 1170的一个子集。在某些情况下，不同标准的所需头文件列表是不同的，有些头文件对于某些标准是可选的。在这些情况下，提要方框列出了所有相关的头文件。

本书所涉及的内容很广泛，由于篇幅所限，本书只讨论至此，我们欢迎你的评论和建议。可以通过pup@vip.cs.utsa.edu发送电子邮件给我们。关于本书的信息可以在WWW站点http://vip.cs.utsa.edu/pup上得到。通过该WWW站点或通过在目录pub/pup中对vip.cs.utsa.edu的匿名文件传输可以得到本书包括的所有代码。

注：本书中多次提到的

《Advanced Programming in the UNIX Environment》，Stevens著

《Modern Operating Systems》，Tanenbaum著

《The C Programming Language》，Kernighan & Ritchie著

中译本即将由机械工业出版社出版。——出版者注

目　录

第二篇 异步事件

第三篇 并 发

第四篇 通 信

第五篇 附 录

第一篇 基础知识

第1章 并 发

[本章提要] 并发指的是在同一时间片内对资源的共享。这通常意味着多个进程共享同一个CPU(即并发地执行)、共享内存，或共享一个I/O设备。如果并发处理不适当，就会使程序莫名其妙地失败，即使过去在相同的输入下它们曾看上去运行得很好在这种条件下也会出现这种结果。操作系统管理着共享资源，而且在过去，程序员允许操作系统处理并发的各个方面。对于现今的复杂程序，情况就不同了，它们需要在现代的计算机上高效而健壮地运行。在一种新的体系结构中，并发控制向系统设计员呈现出新而重要的内涵，而多处理器桌面机器和分布式系统则是该体系结构的一些实例。本章介绍了并发问题，并为并发环境下UNIX系统上的编程提供了一些准则。

在近50年内，计算机能力在计算速度、内存和块存储容量、电路复杂性、硬件可靠性及I/O带宽等方面有着几何级的增长。在过去10年中，伴随着单CPU上复杂的指令流水线、桌面计算机上多CPU的使用以及网络连接中的激增，该增长一直在继续。

通信和计算能力的惊人增长引发了商业软件的重大变革。过去，大型数据库和其他商业应用在与终端相连的主机上执行，现在则分布在更小型、更便宜的机器上，终端已经让位于具有图形用户接口和多媒体能力的桌面工作站。在该领域的另一端，独立运行的个人计算机已在向使用网络通信演化。电子表格应用程序不再是一个支持单用户的孤立程序，因为一个电子表格的更新可能导致其他关联应用程序(例如，用图表表示数据或执行销售计划)的更新。像协同编辑、会议和白板这样的应用程序推动了群组操作(group work)和交互。计算趋势正在向复杂数据的共享、应用程序的实时交互、智能的图形用户接口以及复杂的数据流(包含音频和视频以及文本)迈进。

所有这些发展依赖于通信和并发。通信是由一个实体向另一个实体的信息传递。并发是同一时间片内资源的共享。当两个程序在同一个系统上执行以便它们的执行能及时地进行交叉存取时，它们便是在共享处理器资源。程序也可以共享数据、代码和设备。并发实体是程序或其他抽象对象内的执行线程。并发也可以在单CPU系统中发生。事实上，现代操作系统的一个重要工作是来管理计算机系统的并发操作。

处理并发并不容易，因为并发程序并不总是像期待的那样运行，而且典型的并发程序缺陷并不以固定的规则出现(在一百万次的执行中，问题可能仅出现一次)。实际上也并不存在一种替代物可以代替关于这些概念的实际经验。本章从并发可能出现的最简单的情形开始描述并发的例子，本书的其他部分将详述这些思想并提供一些具体的例子。

1.1 多道程序设计和多重任务处理

操作系统管理着系统资源——处理器、内存、以及包括键盘、显示器、打印机、鼠标、

磁盘、CD-ROM和网络通信在内的I/O设备。操作系统看上去以一种复杂的令人费解的方式运行，而这种方式是由于外围设备的特性而引起的，特别是它们相对于CPU或处理器的速度。表1-1以纳秒为单位给出了典型的处理器、内存和外设的速度。第三列显示了减速一亿倍后的速度，它给出了缩放到人类表达方式中的速度。每秒一个操作的比例时间粗略地等于50年前老式机械计算器的速率。所引用的速度是一个活动(动态发展)的对象，但是趋势是处理器速度的指数增长导致了处理器和外设之间的渐增的性能差异。

表1-1 计算机系统构件的典型时间

项 目	时 间	减速一亿倍
处理器周期	10纳秒(100兆赫兹)	1秒
高速缓存访问	30纳秒	3秒
存储器访问	200纳秒	20秒
上下文切换	100 000纳秒(100微秒)	166分钟
磁盘访问	10 000 000纳秒(10毫秒)	11天
时间片	100 000 000纳秒(100毫秒)	116天

磁盘驱动器的存取速度已有所提高，但它们的旋转机械的本性限制了它们的性能而使磁盘存取时间并没有取得指数级的缩短。处理器时间和磁盘存取时间之间的悬殊在继续增长，现在对于100MHz的处理器，比率大约为1:1 000 000。

进程是程序执行中的一个实例。第2章讨论了程序成为进程的机制。上下文切换时间是从执行一个进程切换到另一个进程所花费的时间。时间片(quantum)粗略地是一个进程必须让另一个进程运行之前而分配给它的CPU时间量。在某种意义上，键盘前的用户也是一个外围设备。熟练的打字员可以每100微秒击键一次。这个时间和进程调度时间片有着相同的数量级。

练习1-1

调制解调器是一个设备，它允许计算机通过电话线和另一台计算机进行通信。调制解调器的典型速度为2 400bps、9 600bps、14 400bps和28 800bps，其中bps意为“每秒比特数”。假定它采用8位来传输一个字节，那么为了传输足够的字符以填满具有25行、每行80个字符的屏幕，请估算一下在每种比特率下所需的时间。现在假设由1024 × 768像素的数组组成的一个图形显示。每个像素具有一个颜色值，可以是256种可能的颜色之一。假定这样一个像素值可以通过调制解调器在8位中传输，传输中没有使用压缩，为了传输足够的像素以填满该图形显示，估算一下在每种比特率下将花费的时间。对于一个14.4kbps的调制解调器，在一个2 400bps的调制解调器可以填满文本屏幕的时间内，如果要使它填满一个图形屏幕，必须要有多大的压缩率？

表1-2 在特定速度的调制解调器连接上填满文本屏幕和图形屏幕的估计时间的比较

速度(bps)	文本时间(秒)	图形时间(秒)
2 400	6.67	2 621
9 600	1.67	655
14 400	1.11	437
28 800	0.56	218

答： 文本显示具有80 × 25=2000个字符，因此必须传输16 000个字节。图形显示有1 024 × 768=786 432个像素，因此必须传输6 291 456个字节。表1-2给出了不同速度下传输文本和图形所用的时间，可见压缩率必须高于65方可。表1-2中的估计并没有计入压缩或通信协议所花费的时间。

从表1-1中可以看出，执行磁盘I/O的进程并不是非常高效地使用CPU：10纳秒对10毫秒，或者按人类的表示方法，1秒对11天。正因为这种时间的悬殊，所以大多数现代操作系统采用多道程序设计。多道程序设计意味着多个进程可以预备执行，操作系统可以从这些就绪进程中选取一个来执行。当一个进程需要等待资源时(如，击键或磁盘存取)，操作系统在该进程停止前将保存再继续执行该进程所需的所有信息，并选择另一个就绪进程来执行。由于资源请求(如read或write)将导致一个操作系统请求(一个系统调用)，并且在系统调用期间操作系统将执行切换到其他进程的代码，我们很容易便可看出这是如何工作的。

UNIX不仅可以进行多道程序设计，而且还是分时共享的，即该操作系统看上去是在同时执行多个进程。如果仅有一个CPU，则在一特定的时间，仅有一个进程的一条指令可以正在执行。但由于人类时间级要比当代计算机的时间级慢几百万倍(甚至几十亿倍)，因此操作系统可以在进程间快速切换，看上去就像是在同时执行多个进程。

下面的类比可以加深对该概念的理解。假设一个杂货店有多个结帐柜台(进程)，但只有一个结帐员(CPU)。结帐员对一个顾客的一个又一个购物(指令)进行结帐，直到需要一个价格检查(一次资源请求)时为止。此时，结账员并不是等待价格检查而无所事事，他将转移到另一个结帐柜台并且对来自另一个顾客的购物进行结账。只要那儿有顾客(进程)等待结帐，结帐员(CPU)将一直在忙碌。结帐员是高效的，但是顾客可能不愿意在这样一家商店购物，因为当那儿有一个大的购单而还没有进行价格检查时可能要等待很长的时间。

现在假定结帐员为一个顾客结帐购物的时间最多为30秒(时间片)。当结帐员开始为某个顾客处理时，一个30秒的定时器即被启动了。如果定时器期满，即使无需价格检查，结帐员也将转移到另一个顾客。这就是分时共享。如果结帐员足够地快，情况几乎等同于在每个结帐柜台都有一个较慢的结帐员。假设对这样一个结帐柜台的录像并以100倍的正常速度回放，就好像结帐员正在同时处理多个顾客。

练习1.2

对应于表1-1中的1秒的处理器周期，假设结帐员每秒能够对一次购物进行结账。对应于该表，在转移到一个正在等待的顾客之前，结帐员在一个顾客上所花的最多时间可能是多少?

答：这个时间就是在表中被缩放到116天的时间片。程序在一个时间片可以执行几十亿条指令，比平均顾客购货的货物数稍大一些。

如果与切换之间的时间(CPU工作时间)相比，从一个顾客转移到另一个顾客的时间(上下文切换时间)很短，结帐员则能够有效地应付顾客。分时共享的一个缺点是在顾客之间切换会浪费时间，但是它的优点是在价格检查时不浪费结帐员的时间，而且小购单的顾客不必长时间等待大购单顾客。

如果只有一个结帐柜台，且同时有多个顾客挤在这个结帐柜台周围，则该类比将更加贴近操作系统中的情况。为了从顾客A切换到顾客B，结帐员保存登记带的内容(上下文)并且将它恢复到上一次处理顾客B时的内容。如果现金登记簿有多个磁带并且能同时保存多个顾客定单的内容，则上下文切换时间可以减少。实际上，一些计算机系统具有专门的硬件来同时保存多个上下文。

多处理器系统具有多个处理器，它们访问一个共享的内存。在对多处理器系统的结帐类

比中，每个顾客具有一个单个的登记带，并且那儿有多个漫游结帐的结帐员，他们直接为未被服务的顾客结账。许多杂货店都有这种工作人员。

1.2 应用层次上的并发

并发既在软件层次上也在硬件层次上发生。并发发生在硬件层次上，是因为多种设备同时操作，处理器具有内部并行机制而同时在几条指令上工作，并且系统具有多个处理器，系统可以通过网络通信进行交互。在信号处理、I/O和处理的重叠、通信以及在不同进程或同一进程内的线程之间的资源共享中，并发发生在应用程序层次上。本章的描述正是基于这3种基本类型的并发。本节从硬件层次上提供了对并发的总的看法。

1.2.1 中断

在传统机器层次上的单个指令的执行缘于处理器指令周期的作用。在一特定的时间内，处理器执行程序计数器中地址所对应的程序(现代处理器具有内在的并行机制如流水线以减少执行时间，但是这里的讨论并不考虑那种复杂性)。因为处理器除了执行指令周期外，还要控制外围设备，因而，并发发生在传统机器层次上。一个外设可以产生一个称为中断的电子信号来设置处理器中的硬件标志。中断的检测是指令周期自身的一部分。在每个指令周期上，处理器检测硬件标志以查看是否有某个外围设备需要注意。如果处理器检测到一个中断已经发生，它保存程序计数器中的当前值并装入一个新值，该值是一个称为中断服务例程的特殊函数的地址。

如果实体不能确定事件发生的时间，那么这个事件对于该实体便是异步的。由外部硬件设备产生的中断对于系统上运行的程序通常是异步的。这些中断并不总是发生在程序执行中的相同点上，因此程序必须给出相同结果而不管它在何处被中断。相反地，错误事件(如被0除)则是同步的，因为它总是在一个特殊指令的执行过程中发生(如果相同的数据提供给该指令的话)。

尽管中断服务例程可以是被中断程序的一部分，而中断服务例程的处理对于并发则是一个不同的实体，因为中断对于该进程是异步的。称为设备驱动程序的操作系统例程通常处理由外围设备所产生的中断。然后这些驱动程序通知相关进程某个事件已经发生了。

操作系统使用中断来实现分时共享。大多数机器具有一个称为定时器的设备，它在一个指定的时间间隔后产生一个中断。操作系统在设置程序计数器运行程序之前启动这个定时器。当定时器到时时，便产生一个中断使得CPU去执行定时器中断服务例程。该中断服务例程将操作系统代码的地址写进程序计数器中，然后再将控制交给操作系统。当某个进程以上述这种方式失去CPU时，我们便说它的时间片已经终止了(到时了)。操作系统将该进程放置到就绪运行的进程队列中，然后该进程便在那儿等待另一个运行的时机。

1.2.2 信号

信号是事件的软件通知。通常，信号是操作系统对于中断(一个硬件事件)的响应，例如，*ctrl+c*键对处理键盘的设备驱动程序产生一个中断。这个驱动程序通过发送信号来通知适当的进程。操作系统也可向进程发送信号以向它通知一个I/O操作的完成或一个错误。

当导致信号的事件发生时，信号便产生了。一旦信号产生，操作系统将为该进程设置一

个对应于该信号的标志。信号可以同步产生，也可以异步产生。如果信号由接收它的进程或线程产生，则它是同步产生的。一个非法指令或被0除的执行可能产生一个同步信号。如击ctrl+c键则将产生一个异步信号。信号(第5章)可以被用于计时器(第6章)、终端程序(7.5节)、作业控制(7.7节)、异步I/O(5.9节和9.1.2节)或程序监控(5.11节)。

如果接收到信号的进程执行了该信号的处理程序，信号便被捕获了。捕获信号的程序至少有两个并发部分：主程序和信号处理程序(5.6节)。如果信号处理程序修改了一些外部变量，而这些变量也可能被主程序在其他某些地方修改过，要正确执行程序，则要求这些变量被保护，否则会出现问题。

1.2.3 输入和输出

操作系统必须能协调各种具有不同特殊操作时间的资源，因而需有专门应用程序来直接处理这个问题以确保高效的执行。当一个程序等待磁盘访问完成时，处理器可以执行上百万条其他操作。分时共享环境下，在当前进程等待I/O请求完成时，操作系统通常选择另一个进程来运行。通过使用异步I/O(5.9节和9.1.2节)或专用线程(第9章)而非普通的阻塞I/O，该进程在等待过程中可以做别的工作。当然，使用并发时必须综合考虑附加性能及其额外程序设计。

当应用监控两个或更多的输入通道(如，来自网络上不同来源的输入)时，一个类似的问题出现了。即如果使用标准的阻塞I/O，当应用开始从一个输入源读数时，它将等待该输入，即使其他来源的输入可用，它也无法处理来自其他来源的输入。第9章提出了处理多源问题的5种方法：a)使用非阻塞I/O的轮询；b)使用异步I/O；c)使用select；d)使用poll(尽管名字是poll，但是并不使用轮询)；e)使用线程。

1.2.4 线程和资源共享

要在UNIX中实现并发执行，一个传统的方法是用户使用fork系统调用来创建多个进程。这些进程通常需要以某种方式来协调它们的操作。在最简单的情况下它们可能只要协调它们的终止点，但进程的终止问题也比看上去的复杂得多。第2章论及了进程结构和管理，并介绍了UNIX的系统调用fork、exec和wait。10.2和10.3节论述了进程终止的敏感问题。

具有共同祖先的进程可以使用一个称为管道(第3章)的简单机制来进行通信。没有共同祖先的进程可以使用信号(第5章)、FIFO(3.9节)、信号量(8.2节)、共享地址空间(第9章)或消息(第12章)进行通信。

执行的多线程可以提供在一个进程内部的并发。当某个程序执行时，CPU使用程序计数器来确定下一条将执行的指令，产生的指令流被称为程序的执行线程，它是该程序的控制流。如果两个不同的执行线程在一个时间片内共享同一个资源，则必须使这些线程互不干扰。多处理器系统增加了应用程序间和应用程序内并发和共享的机会。在多处理器系统上，当一个多线程应用程序同时有不止一个执行线程处于活动状态时，来自同一进程的多条指令可能在同一时间内执行。

现在还没有一个关于使用线程的标准，而程序员也不愿意使用线程，因为每个厂商的线程包是不同的。随着每个新操作系统的发行，线程包都可能经历一些变化。最近，在POSIX.1标准中融入了线程标准，第9章和第10章将深入讨论。

1.2.5 分布式计算网络

另一个重要的趋势是计算在网络上的分布。并发和通信联合起来形成了新的应用。客户机-服务器模型是使用最广泛的分布式计算模型。这种模型中的基本实体是管理资源的服务器进程和请求访问共享资源的客户机进程(进程可以既是客户机又是服务器)。客户机进程通过向服务器发送请求来共享资源，服务器将执行代表客户机的请求并向客户机发送回应。基于客户机-服务器模型的应用的典型例子有文件传输(ftp)、电子邮件和文件服务器等。开发客户机-服务器应用需要对并发和通信有深入的理解，而实现客户机-服务器模型的基本机制便是第14、15章所讨论的远程过程调用或RPC。

另一种分布式计算的模型是基于对象的模型。系统中的每一个资源可看作是一个具有消息处理接口的对象，所有共享资源都以统一的方式访问。基于对象的模型分享了面向对象编程的优点，均允许受控的增量开发和代码重用。对象框架定义了代码模块之间的交互，而对象模型自然地体现了保护的概念。许多实验性的分布式操作系统，包括Argus、Amoeba、Mach、Arjuna、Clouds和Emerald，都是基于对象的。基于对象的模型需要对象管理器来追踪系统中的对象位置。

一个真正的分布式操作系统的可选方法是提供应用程序层，这些应用程序层运行于常见的独立操作系统之上以利用网络的并行机制。并行虚拟机(PVM)是一个软件包，它允许多个不同种类(heterogeneous)的UNIX工作站像一台并行机那样工作，以解决大型计算问题。PVM通过网络来管理和监控分布于不同工作站上的任务。第11章开发了一个PVM简化版本的调度器，而第15章为Linda(一种商用分布式编程语言)的简化版本开发了元组空间的实现。

1.3 UNIX标准

不同厂商的UNIX操作系统总存在系统层次的差异，这颇伤人脑筋。本文遵循三个重要的标准——ANSI C、POSIX和Spec 1170。C语言已经被美国国家标准协会(ANSI)和国际标准化组织(ISO)进行了标准化。

在系统层次上，电气电子工程师协会(IEEE)制定了一系列的称为POSIX(Portable Operating System Interface)的标准。POSIX使操作系统和用户之间的接口标准化，以便应用程序可以跨平台移植。POSIX标准也称为IEEE Std. 1003，例如，有关系统应用程序接口的标准也称为IEEE Std. 1003.1或POSIX.1。表1-3总结了一些POSIX标准。

适应POSIX的实现必须支持POSIX.1的基础标准。POSIX.1的许多方面都不属于基础标准，而被定义为对基础标准的扩展。如果在一个应用的unistd.h中定义了相应于某个扩展的符号，则该应用将支持该扩展。表1-4列出了这些符号以及它们相应的POSIX扩展。

除了UNIX外，POSIX还包含其他的操作系统，但即便是从UNIX的角度上看也存在一些混乱。UNIX有两个主要的流派：System V和BSD。System V由最初的AT&T UNIX演化而来，BSD UNIX则是由加州大学伯克利分校开发，其目标是提供一个具有复杂网络支持的开放系统。看上去，大多数商业UNIX版本都在System V Release 4和一个正在形成的称为Spec 1170的标准上进行标准化。而Spec 1170是由X/Open Foundation制定的。本书侧重使用POSIX、Spec 1170和System V Release 4标准。对于像网络服务之类的POSIX标准没有包括的项目，将遵循Spec 1170标准。对于两个标准都没有包括的项目，本书执行System V Release 4标准。

表1-3 部分POSIX标准

标 准	公布时间	说 明
POSIX.1	1990	系统应用程序接口(API)[C语言]
POSIX.1b	1993	API Amendment 1: 实时扩展[C语言](亦称POSIX.4)
POSIX.1c	1995	API Amendment 2: 线程扩展
POSIX.2	1992	外壳与实用工具
POSIX.3	1991	POSIX一致性测试方法
POSIX.3.1	1992	POSIX. 1 一致性测试方法
POSIX.4	1993	现称为POSIX.1b
POSIX.5	1992	POSIX.1[ADA语言]
POSIX.6	制定中	安全性
POSIX.7	制定中	系统管理
POSIX.7.1	制定中	打印管理
POSIX.7.2	制定中	软件安装与管理
POSIX.7.3	制定中	用户(组)管理
POSIX.8	制定中	透明数据访问
POSIX.9	1992	POSIX.1[FORTRANL语言]
POSIX.12	制定中	网络服务

表1-4 POSIX.1编译时的符号常量，若常量在unistd.h中定义则系统支持相应的POSIX.1扩展

符 号	POSIX.1扩展
_POSIX_ASYNCHRONOUS_IO	异步输入与异步输出
_POSIX_FSYNC	同步存储
_POSIX_JOB_CONTROL	工程控制
_POSIX_MAPPED_FILES	存储器转换文件
_POSIX_MEMLOCK	进程存储锁定
_POSIX_MEMLOCK_RANGE	排列存储锁定
_POSIX_MEMORY_PROTECTION	存储保护
_POSIX_MESSAGE_PASSING	信息传送
_POSIX_PRIONITIZED_IO	优先输入和输出
_POSIX_PRIORITY_SCHEDULING	处理过程
_POSIX_REALTIME_SIGNALS	实时信号
_POSIX_SAVED_IDS	进程已存储用户ID
_POSIX_SEMAPHORES	号志监控模块
_POSIX_SHARED_MEMORY_OBJECTS	共享存储器对象
_POSIX_SYNCHRONIZED_IO	同步输入与同步输出
_POSIX_TIMERS	计时器
_POSIX_VERSION	199309L(用于相容实现)

1.4 UNIX中的编程

用专门设计的支持并发的语言来编程是学习并发的一种方法。本书通过标准C和UNIX编程来对并发进行深入探讨。本节将给出在UNIX下开发系统程序的一些简要的准则。附录A为系统程序员提供了UNIX的一些基础知识。

通过调用一些系统调用，程序可以请求一些操作服务(如输入和输出)。系统调用是直接进入内核的入口点。内核是运行于特权模式下的软件模块——意味着它们可以完全地存取系统资源。系统调用是操作系统内核的一个本质部分(有时，术语“操作系统”和“内核”可以相互换用)。