书名莱昂氏 UNIX 源代码分析
原书书名：Lions’Commentary on UNIX 6th Edition with Source Code
作者：(澳)　John Lions
译者：尤晋元
书号：ISBN 7-111-08018-1
估价：58 元
页数：378 页
出版日期：2000 年 7 月 1 日

内容简介：

本书由上、下两篇组成。上篇为 UNIX V6 的源代码，下篇是莱昂先生对 UNIX 操作系统版本 6 源代码的详细分析。该书最早于 1976 年在作者当时所在的澳大利亚新南威尔士大学内部流通，但是由于有关公司希望保守在 UNIX 内核方面的商业机密，该书多年来从未正式出版。但本书一直受到 UNIX 黑客的关注，私下广为流传于世，是一部杰出、经典之作。在各方有识之士的努力下，该书终于于 1996 年正式出版，此次中文版的推出在国内计算机图书出版界尚属首次。全书语言简洁、透彻；作为一本自学 UNIX 的教材，二十多年来一直鼓舞和激励着计算机和高级操作系统方面的专业人员和学生。。

作者简介：

Lions，1937 年生于澳大利亚的悉尼。是悉尼大学应用理学学士，英国剑桥大学控制工程博士，曾受雇于多家公司，担任顾问及技术主管。1972 年开始在澳大利亚新南威尔士大学执教，直至 1995 年因为健康原因退休。Lions 先生是 ACM 会员，曾担任《澳大利亚计算机杂志》主编，同时他还是"澳大利亚 UNIX 系统用户组"的终身会员。

译者简介：

尤晋元，上海交通大学计算机科学及工程系教授、博士生导师、系主任，中国计算机学会理事，上海计算机学会软件工程专业委员会主任。在科研教学方面，他的主要工作领域是：操作系统设计与实现，分布对象计算，移动计算，构件/构架技术，Java 技术。他曾编著、翻译过多本与操作系统，特别是 UNIX 系统有关的著作，深受读者欢迎。

专业人士对本书的评价：

正式出版约翰·莱昂编著的《莱昂氏 UNIX 源代码分析》标志着一个悠长故事的圆满结束。在代码中你也可以观察到其基本结构，该结构沿用了很长时间，而且能够包容在计算环境中发生的巨大变化。在莱昂的分析中，你可以觉察到新鲜的、经常提出问题的立场，其中的很多词语和思想都很适宜于教育和学习。本书已经教育了一代人，它是计算机界中复印数量最多的一本书稿。将此书公开出版是件大好事。
——丹尼斯·M·里奇

我对 1997 年中的那一天仍然是记忆犹新，那时我接到邮寄来的此书的第一份草稿，开始时我对此书并无很多期望，但是最后却是逐字逐句进行了仔细阅读。20 年之后，此书依旧是对一个实际操作系统工作的最好分析说明。
——肯·汤姆森

(汤姆森和里奇因为开发和实现 UNIX 操作系统而共同获得 1983 年 ACM 美国计算机协会图灵奖)。

有一个事实可以用来衡量<<莱昂氏 UNIX 源代码分析>>一书取得了何等成功，这就是它们的大量非法影印本的四处流行。我在 1987 年访问过一家小型计算机公司，居然在一个书架上发现了<<莱昂氏 UNIX 源代码分析>>的影印本，从其扉页上的答案推断，它至少是从经过四次复制的影印件上拷贝下来的，象这样的作法，纵然是近乎非法的，但依然通行无阻。约翰。莱昂以其特有的方式写作本书，并且重新组织操作系统这门课程的教学，其中包含着多种理由。在原书的绪论中，对多数理由已作了说明。但是有一条理由，在出版物中一向很少被人提及，这或许并非一条不太重要的理由。这就是那时构成计算机科学教学计划的所有课程中，除硬件课程之外，都教学生如何编写程序和调试程序。但是没有一门课程要求学生学会"阅读"程序。其他课程牵扯的多是小程序，即学生能够管理的程序，而 UNIX 核心系统显然是一个非常大的程序。对于这种情况，约翰以不无讽刺的口吻说："他们看到仅有的大程序是那些由他们自己编写的程序；而这一个至少是写的很好的程序。"在以后的几年中，教学生们去阅读和考察代码变成人们接受的事实。
现在回头去看这本书，我发现提出这些目标现在仍很合适。UNIX 版本 6 核心系统的结构完美、程序简约。直到今天，尚未发现其他程序能够展示如此高超的软件工程技艺。UNIX 处处显现出设计者的见识和独具匠心，简明扼要而易于理解，兼收并蓄各种折中和无需优化的低效代码，莱昂取得的部分成就，就是认识到 UNIX 的这种特质，并把它们融入自己的书中。

——格里格·罗斯(曾就读于新南威尔士大学，毕业后创立了两家软件公司，其中一家经营至今。他在 IBM 的沃森研究中心作过一年访问学者，其后在澳大利亚计算与通信研究所和斯特林软件公司任职，并加盟高通公司。格里格是澳大利亚 UNIX 系统用户组的前总裁和 USENIX 的董事。)

目录：

第 1 章　绪　　论

UNIX 是在 PDP11 计算机上运行的一个分时系统的名字，它由贝尔实验室的 Ken Thompson 和 Dennis Ritchie 编写。《ACM 通信》的 1974 年 7 月号刊载了他们的论文，对该系统进行了说明。

事实说明 UNIX 功能丰富、高效和可靠，到 1976 年底为止，它已安装在 150 多台机器上并得到应用。

直至 UNIX 的 Level Six System,编写 UNIX 共投入的人力约为 10 人年，与其他系统相比较，这是非常小的一个数字。(例如，到 1968 年，OS/360 已投入 5000 人年，而另一个 IBM 操作系统 TSS/360 则投入了 1000 多人年。)

UNIX 的强大能力主要来自于它的简练和直接，UNIX 由两位在同一办公室工作的人员编写，他们用了一种称作为 C 的程序设计语言，还受到 PDP11 非常有限的存储容量和计算能力的限制。

不仅是 UNIX 的能力令人注目，其可理解性也是大多数其他系统难以比拟的。为了较深入地理解 UNIX，必须掌握的资料数量并不太多。与之相比较，那么任一个人想要完全理解 OS/360 及其后继者都是不太可能的，因为它们过于复杂。在对任一主流操作系统作较大修改前，一个人需要先花费数月时间对该系统进行学习。

自然，有些系统比 UNIX 更易于理解，但是可以断言这些系统会简单得多，而且它们企图达到的目标也非常有限。如果关心的是向用户提供的功能类型和数量，那么 UNIX 就位于其中。事实上，UNIX 提供了很多某些非常著名的主流操作系统所缺少的功能。

1.1　UNIX 操作系统

《莱昂氏 UNIX 源代码分析》的目的是详细地说明我们从 UNIX 分时系统中选出的部分，这部分代码在 UNIX 运行时常驻内存，我们将其称作为“UNIX 操作系统”。这部分代码提供下列功能：

?系统初启。
?进程管理。
?系统调用。
?中断处理。
?输入/输出操作。
?文件管理。

1.2　公用程序

UNIX 的其余部分(它比内核大得多!)由一组剪裁适当的程序组成，它们以“用户程序”方式运行，常被称为“公用程序”。

在此标题下，有一组与操作系统关系特别紧密的程序，例如：

“shell”(命令语言解释程序)
“/etc/init”(终端配置控制程序)

以及一组文件系统管理程序，例如：

check　　du　rmdir
chmod　mkdir　　sync
clri　mkfsumount
df　mount　　update

应当指出的是：由上述程序执行的很多功能在其他计算机系统中是由操作系统提供的，而且在操作系统中占了相当大的比重。

上述程序的功能性说明及使用方法包含在《UNIX 程序员手册》UNIX Programmer’s Manual。

UPM 的第一部分(公共使用的程序)和第三部分(仅由系统管理员使用的程序)中。

1.3 其他文档

本书中的注释频繁地引用《UNIX 程序员手册》(UPM)，偶而也引用《UNIX 文档》(UNIX Documents)手册，而对《UNIX 操作系统源代码》(“UNIX Operating System Source Code”)，则几乎时时处处都要引用。

所有这些文档都关系到对 UNIX 系统的完整理解。另外，为了学习汇编误语言例程，需要参考 DEC 出版的《PDP11 处理机手册》PDP11 Processor Handbook)。

1.4 UNIX 程序员手册

《UNIX 程序员手册》(UPM)分成八个主要部分，在它们之前还有一个目录和一个 KWIC(Key Word In Context,上下文中的关键词)索引。KWIC 是极其有用的，但有时也使人烦恼，某些被索引到的材料并不存在，而某些现存的材料却设有索引到。

在该手册的每一部分中，材料按主题名的字母顺序编排。在主题名后按惯例附加了宅所在部分的编号，某些主题会出现在几个部分中，例如，“CHDIR(Ⅰ)”和“CHDIR(Ⅱ)”。

第一部分(SectionⅠ)包含了各种命令，它们或者是由“shell”命令解释程序所识别的，或者是标准用户公用程序的名字。

第二部分(SectionⅡ)包含“系统调用”(System calls)。系统调用是用户程序可以调用以获得操作系统服务的操作系统例程。学习 UNIX 后，我们会对大多数系统调用相当熟悉。

第三部分(SectionⅢ)包含“子例程”(subroutines)。这些子例程是可由用户程序调用的序例程。对一般程序员而言，第二和第三部分的区别并不很重要。第三部分的大多数与操作系统并无关系。

第四部分(SectionⅣ)说明“特殊文件”(special file)，在 UNIX 中将外部设备称为“特殊文件”。某些特殊文件是相关的，有些则关系不大。这取决于你所处的位置。

第五部分(SectionⅤ)说明“文件格式和约定”(File Formats and Conventions)。大量与理解 UNIX 紧密相关的信息都可在这一部分中找到。

第六部分(SectionⅥ)和第七部分(SectionⅦ)说明“用户维护的”程序和子例程。UNIX 的爱好者不会忽略这两个部分，但是这两部分和操纵系统并没有密切的联系。

第八部分(SectionⅧ)说明“系统维护”(软件部分，不包括硬件部分)。如果你对如何管理 UNIX 特别有兴趣，那么这一部分就包含了大量你所需的信息。

1.5 UNIX 文档

本文档手册是相关程度不等的文集，其主要内容包括：

?构造和安装 UNIX 实际上属于 UPM 的第八部分(这部分内容是有关的)。

?UNIX 分时系统是原先发表在《ACM 通信》上论文的修改版。这篇文章极其重要，应当每月至少重读一次。

?UNIX 入门，如果你在 UNIX 方面缺少经验，那么这是非常有用的。

?除非你是 C 语言等方面的专家，那么“C”语言导引、编辑程序以及“C”语言和汇编语言的参考手册都是非常有用的。

?UNIX I/O 系统提供了 UNIX 系统很多特征和功能的综述。

?UNIX 摘要提供了一张检查表，该检查表在回答“一个操作系统是做什么的?”此类问题是有用的。

1.6 UNIX 操作系统源代码

这是对 Bell 实验室所提供的 UNIX 系统代码经编辑处理后形成的一个版本。

该代码假定的硬件“模型”系统包括：

PDP11/40 处理机。

PK05 磁盘驱动器。

LP11 行式打印机。
PC11 纸带阅读/穿孔机。
KL11 终端接口。
对 Bell 实验室提供的源代码所作的主要更改是：
?各文件的编排顺序已经更改。
?在许多文件中材料的先后顺序已经更改。
?代码在文件之间作了极少量的迁移(现在回顾起来当时还应当做更多的此类代码迁移)。
?大约 5%的代码行以各种方式缩短成少于 66 个字符(删去空格、对注释重新安排、分成 2 行等)。
?增加了一些由下划线组成的行，特别在程序过程的结尾处。
?每个文件的长度都调整为 50 行的整数倍，原先文件的不足部分用空白行增补。
?每行的行首加了一个 4 位数标识该行的编号，以便于交叉引用。
在源代码篇的开始部分有若干摘要性内容，包括：
?UNIX 操作系统过程分类索引：其中各过程都带了行号。
?UNIX 操作系统文件及过程：显示各文件及文件中包含的过程出现的前后顺序。
?UNIX 操作系统定义的符号列表各定义符号都带它们的值。
?UNIX 操作系统源代码交叉引用列表：给出了每个符号引用处的行号。其中省略了“C”的保留字和若干通用符号，例如“p”、“u”。

1.7 源代码中各部分

源代码分成五部分，每个部分致力于系统的一个主要方面。
这种划分的意图是使每个部分尽可能自包含，无需掌握后继部分就能作为一个单元学习每一部分，这一目的已经基本达到。各部分的主要内容是：
第一部分：处理系统初启和进程管理。它也包含了所有汇编语言例程。
第二部分：处理中断、陷入(Trap)、系统调用和信号(signal),信号又称为软件中断(software mterrapt)。
第三部分：主要处理程序换进/换出(swapping)、基本磁盘操作，以及面向块的输入/输出。它也处理块缓冲池的资源管理和使用。
第四部分：处理文件和文件系统，包括文件创建、维护、使用和删除。
第五部分：涉及“字符特殊文件”(“character special files”)。UNIX 将低速外部设备称为字符特殊文件，这类设备共用面向字符的缓冲池。
各部分的内容将在第 4 章中更加详细地列出。

1.8 源代码文件

上述五个源代码部分的每一个都由若干源代码文件组成。每个文件的名字都带一个标识其类型的后缀：
“.s”表示这是包含汇编语言语句的文件。
“.c”表示这是包含可执行“C”语言语句的文件。
“.h”表示这是包含“C”语言语句的文件，但该文件并不单独进行编译，而需包含在其他“.c”文件中一起进行编译，亦即“、h”文件包含了全局说明。

1.9 分析的使用

本书包含的分析力图对已存在于源代码中的注释进行补充，它对理解 UNIX 操作系统并不是绝对必须的。在阅读 UNIX 源代码时可以不阅读本书，而且鼓励你尽可能这样做。
在阅读 UNIX 源代码时，如果遇到难点而不能前进，那么你就可以把这本“分析”当作拐杖而使用它，从而得到帮助。如果你完全独立地阅读每一个文件或过程，那么在开始阶段可能相当慢，但是进展会越来越快。阅读他人编写的程序是一种应当学习并加以实践的艺术，这

是一门有用的艺术。

1.10　对程序设计水平的一条注释

你将发现大部分 UNIX 代码具有很高的水平。很多部分在你开始阅读时似乎很复杂也不清晰，但在进一步加以分析和回顾后，你会发现它们的思路很完美而且可能是唯一的编写方法。由于这一原因，所以在本“分析”中偶而对程序设计风格所做的评论几乎都是针对一些稍稍偏离通常的高水平之处而言的。

造成这些小问题的原因是什么呢?部分原因是对原先的代码加了补钉。有些我们找到的“坏”代码事实上往往与采用的巧妙方法有关连，而这种情况在开始阅读时并不能清楚地意识到。当然也确实具有可进一步斟酌之处。但是作为一个整体，你会发现 UNIX 的作者创造了一个强有力的、有机结合成一个整体的、功能非常丰富的程序，对此应当赞美并加以仿效。

第 2 章　基 础 知 识

UNIX 运行在由数字设备公司(DEC)制造的 PDP11 系列计算机的较大型号机上。本章对这些计算机，特别是 PDP11/40 机的某些重要特征提供概括性的摘要。

如果读者以前并不熟悉 PDP11 系列计算机，那么应当先读 DEC 出版的“PDP11 处理机手册”。一台 PDP11 计算机由下列几部分组成：一个处理器(也称为 CPU)、与之相连接的一个或多个内存部件以及若干外设控制器，将这三部分连接起来的是一根称之为“Unibus”(单总线)的双向平行通信线。

2.1　处理机

处理机是按 16 位字长的指令、数据和程序地址设计的，它包含一组高速寄存器。

2.2　处理机状态字

这是一个 16 位寄存器，它分成若干位段，它们的意义说明如下：

位	说　明
14，15	当前状态(00＝核心态)
12，13	前状态(11＝用户态)
5，6，7	处理机优先级(范围 0－7)
4	陷入法
3	N 位，若上次结果为负则设置此位
2	Z 位，若上次结果为零则设置此位
1	V 位，若上次操作产生溢出则设置此位
0	C 位，若上次操作产生进位则设置此位

处理机可以在两种不同模式下操作：核心态和用户态。与用户态相比，核心态具有较高优先权，它由操作系统保留供自身使用。模式选择决定了下列事项：

?使用哪一组存储管理段寄存器，这些寄存器的作用是将程序虚地址翻译成物理地址。

?使用哪一个寄存器作为 r6，r6 是“栈指针”。

?某些指令，例如 halt(停机)，是否可以执行。

2.3　通用寄存器

处理机包含了一组 16 位寄存器，其中有 8 个是任何时候都可存访的“通用寄存器”。这些寄存器被称之为：

r0、r1、r2、r3、r4、r5、r6 以及 r7。

前 6 个通用寄存器可用作为累加器、地址指针或变址寄存器。UNIX 使用这些寄存器的惯例是：

?r0、r1 在表达式求值时用作临时累加器；在过程返回时存放返回值；在过程调用的某些情况下可用来传递实参。

?r2、r3 和 r4 在过程执行时可用作局部变量。在过程调用入口处存储这些寄存器的值，在退出过程时则恢复这些寄存器值。
?r5 用作过程激活记录动态链的链首指针，该动态链存放在当前栈上。r5 被称为“环境指针”。
最后两个通用寄存器具有专门的意义和作用：
?r6(也称为“sp”)用作栈指针。PDP11/40 处理机有两个寄存器，分别在核心态和用户态下用作“sp”。而其他通用寄存器则没有这种双份使用方式。
?r7(也称为“pc”)用作程序计数器，亦即指令地址寄存器。

2.4 指令集

PDP11 指令集包括双、单和零操作数指令。指令长度通常是一个字，某些指令则扩充为两个或叁个字以包括附加的寻址信息。
对于单操作数指令，其操作数通常称为“目的操作数”；在双操作数指令中，两个操作数分别被称为“源操作数”和“目的操作数”。后面将说明各种寻址方式。
文件 m40.s 是针对 11/40 处理机的汇编语言支持例程文件。下列指令在该文件中得到应用。
注意，N、Z、V、C 是处理机状态字(“ps”)中的条件码位。很多指令以副作用设置这些位，而“bit”、“cmp”和“tst”指令的主要功能则是设置这些条件码位。

adc 将 C 位的内容加至目的操作数。
add 将源操作数加至目的操作数。
ash 按移位计数将指定寄存器的内容左移相应次数(负值表示右移次数)。
ashc 除涉及两个寄存器外，与 ash 指令相同。
asl 所有位向左移 1 位。第 0 位装入 0，而第 15 位的原值则装入 C。
asr 所有位向右移 1 位。第 15 位保持原先值，而第 0 位的原先值则装入 C。
beq 若等于，也就是若 Z=1 则转移。
bge 若大于或等于，也就是若 N=V 则转移。
bhi 若高于，也就是若 C＝0 以及 Z＝0 则转移。
bhis 若高于或相同，也就是若 C＝0 则转移。
bic 若源操作数中的位为非零值，则将目的操作数中的相应位清除为零。
bis 对源和目的操作数执行或操作，将结果存入目的操作数。
bit 对源和目的操作数执行逻辑与操作，设置条件码。
ble 若大于或等于，也就是若 Z=1 或 N=V 则转移。
blo 若低于(0)，也就是若 C＝1 则转移。
bne 若不等于(0)，也就是若 Z=0 则转移。
br 转移到.－128 和.＋127 之间的一个单元，其中，表示当前单元的地址。
clc 清除 C。
clr 将目的操作数清为 0。
cmp 比较源和目的操作数，设置条件码位。若源操作数的值小于目的操作数值，则设置 N 位。
dec 将目的操作数的值减 1。
div 存在 rn 和 r(n+1)(其中，n 是偶数)中的 32 位二进制补码整数被除以源操作数。商存在 rn 中，而余数则存在 r(n+1)中。
inc 将目的操作数的值加 1。
jmp 跳转至目的地址。
jsr 跳转至子程序。寄存器值则按下列方式从左向右移动：
pc、 rn、－(sp)=dest.、pc、rn
mfpi 将前地址空间中指定字的值压入当前栈

mov　将源操作数值复制至目的操作数。
mtpi　从当前栈弹出一个字，将其值存入前地址空间中的指定字中。
mul　将 rn 的内容和源操作数相乘。若 n 是偶数，则将积存放在 rn 和 r(n+1)中。
reset　将 Unibus 中的 INIT 线设置成 10ms。其作用是重新启动所有设备控制器。
ror　将目的操作数中各位循环右移 1 位。第 0 位的原先值装入至 C，而 C 的原先值装入至第 15 位。
rts　从子程序中返回。从 rn 中重装 pc，从栈中重装 rn。
rtt　从中断或陷入返回。从栈中弹出重装 pc 和 ps。
sbc　从目的操作数中减进位位。
sob　从指定寄存器值中减 1。若结果非 0，则转移回“位移”(offset)字。
sub　从目的操作数中减源操作数。
suab　交换目的操作数中的高、低字节。
tst　按目的操作数的内容设置条件码，N 和 Z。
wait　使处理器空转，释放 Unibus，直至发生一硬件中断。
与上面说明的字版本指令一样，下列字节版指令也用于 m40.s 文件中：
bis　inc
clr　mov
cmp tst
2.5　寻址方式
PDP11 指令集的新颖和复杂性很多来自于它所提供的多种寻址方式,这些寻址方式可以用于规定源和目的操作数。
下面说明“m40.s”中使用的寻址方式。
2.5.1　寄存器方式
操作数驻留在一个通用寄存器中，例如：
clr　r0
mov r1,r0
add　r4,r2
在下列方式中，指定的寄存器包含一地址值，用其定位操作数。
2.5.2　寄存器延迟方式
寄存器包含操作数的地址，例如：
inc　(r1)
asr　(sp)
add　(r2),r1
2.5.3　自动增 1 方式
寄存器包含操作数地址。作为一副作用，在操作后，寄存器值增 1，例为：
clr　(r1)+
mfpi (r0)+
mov (r1)+,r0
mov r2,(r0)+
cmp (sp)+,(sp)+
2.5.4　自动减 1 方式
寄存器值先减 1，然后用来定位操作数，例如：
inc　－(r0)
mov －(r1),r2

```
mov (r0)+,－(sp)
clr　－(sp)
```

2.5.5　变址方式

操作数地址由两部分组成，一部分是寄存器包含的值，另一部分是紧跟指令后的一个 16 位字值，两者相加得到操作数地址，例如：

```
clr　2(r0)
movb　　6(sp),(sp)
movb　　－reloc(r0),r0
mov －10(r2),(r1)
```

在这种寻址方式中，寄存器可被视为变址寄存器或基地址寄存器。m40.s 采用后一种看法。上面的第三个例子是使用寄存器作为变址寄存器的少数几个实例中的一个。(注意，“－reloc”是一个可接受的变量名。)

有两种寻址方式，它们只用于下列两个例子中：

```
jsr　pc,*(r0)+
jmp　*0f(r0)
```

其中，第一个例子涉及“自动增量延迟”方式的使用。(这用于例程“call”中的 0785 和 0799 行。)企图执行的子程序的地址在 r0 指向的地址字中找到，也就是说涉及到两级间接寻址。在此种使用中 r0 增 1 的副作用并无任何实际影响。

第二个例子出现在 1055 和 1066 行，它是“变址延迟”方式的一个例子。“jump”(“跳转”)的目的地址是一个字的内容，该字的地址值是 0f 加上 r0 的值(一个小的正整数)。这是实现多路选择(multi-way switch)的标准方法。

下列两种方式用程序计数器作为指定的寄存器，以达到某种特定的作用。

2.5.6　立即方式

这是 pc 自动增 1 方式。操作数从程序串中取得，也就是这是一个立即操作数，例如：

```
add　$2,r0
add　$2,(r1)
bic　$!7,r0
mov $KISA0,r0
mov $77406,(r1)+
```

2.5.7　相对方式

这是 pc 变址方式。相对于当前程序计数器值的地址从程序串中取得，并与 pc 值相加得到操作数的绝对地址，例如：

```
bic　$340,ps
bit　$1,SSR0
inc　SSR0
mov (sp),KISA6
```

应当引起注意的是，“变址”、“变址延迟”、“立即”以及“相对”方式都将指令延长了一个字。

由于存在“自动增 1”和“自动减 1”方式，再加上 r6 的特殊属性使得我们可以方便地将许多操作数存放在栈或 LIFO(后进先出)列表中，在存储器中它向下生长。从此可以得到很多优点：代码串的长度缩短；易于编写位置独立的代码。

2.6　UNIX 汇编程序

UNIX 汇编程序是一个两遍汇编程序，它不具有宏功能。《UNIX 汇编程序参考手册》对其作了完整的说明。

下面的一些简要注解应对读者有所帮助：

1) 一数字字符串可定义一数值常数。若该数字字符串以“.”终止，则被解释为十进制数，否则为八进制数。

2) 字符“／”用于表示该行的后继部分是注释。

3) 若在同一行中包含有多条语句，则相互间应当用分号分隔。

4) 字符“.”用于表示当前位置。

5) 在 DEC 汇编用“#”和“@”之处，UNIX 汇编分别改用了“$”和“*”。

6) 标识符由若干字母数字字符组成(包括下划线)。只有前 8 个字符是有意义的，而且第一个字符不能是数字字符。

7) 在“C”程序中出现的全局变量各在汇编程序中加了一个前缀，它由单个下划线组成。例如，在汇编语言文件“m40.s”中第 1 025 行上出现变量“－regloc”，它引用“trap.c”文件中 2 677 行的变量“regloc”。

8) 有两类语句标号：名字标号和数值标号。后者由一个数字后面跟一个冒号组成，这种标号无需是唯一的。如若引用 nf,其中“n”是一个数字，则表示要引用的是向前搜索遇到的第一个标号“n：”。

如若引用“nb”，则表示要引用的是向后搜索遇到的第一个标号“n”。

9) 下列形式赋值语句

标识符＝表达式

将值和标识符的类型相结合。例如：

.＝60^.

其中，操作符“^”传递第一个操作数的值以及第二个操作数的类型(在此例是“存储单元”。

10) 字符串引用符号是“＜”和“＞”。

11) 下列形式的语句

.global x,y,z

使名字“x”、“y”和“z”为外部名。

12) 名字“－edata”和“－end”是装入程序的伪变量，它们分别定义了数据段以及数据段加上 bss 段的长度。

2.7 存储管理

在 PDP11 上运行的程序，其直接寻址最大可为 64K 字节(32K 字)，这与 16 位地址字长相一致。考虑到经济因素，较大型号的 PDP11 机可以配置较多的存储器(PDP11/40 最多可配置 256K 字节)。PDP11 机又包括了存管部件(memoy management unit)，它将 16 位虚地址(或称程序地址)变换成 18 位或更多位的物理地址。PDP11/40 的存管部件较 PDP11/45 或 PDP11/70 的简单。

PDP11/40 的存管部件由两组寄存器组成，它们将虚地址映照为物理地址。这些寄存器被称之为“活动页寄存器”或者“段寄存器”。一组寄存器在处理机处于用户态时使用，另一组则在核心态时使用。更改这些寄存器的内容也就更改了虚、实地址的具体映照。进行这种更改的能力作为一种特权由操作系统保留自用。

2.8 段寄存器

每组段寄存器包含 8 对寄存器，其中每一对的组成是：“页地址寄存器”(PAR)和“页说明寄存器”(PDR)。

每一对寄存器控制一页的映照，而页长为 8K 字节(4K 字)，程序虚地址空间包含 8 页。

每一页又分成 128 块(block)，每块长度为 64 字节(32 字)。块长是存储映照的基本单位，也是存储分配的基本单位。

任一虚地址不是属于这一页，就是属于那一页。相应物理地址的产生方式是：将页内相对地

址与相应 PAR 的内容相加，这构成了一个位数扩展的地址(PDP11/40 和 11/45 是 18 位；PDP11/70 是 22 位)。于是每个页地址寄存器对相应页起重定位寄存器的作用。
每一页可被分成两部分，一部分被称为上部(高地址部分)，另一部分则被称为下部(低地址部分)，它们以 32 字的整数倍数作为边界而分隔，因此每一部分的长度都是 32 字的整数倍。特别地，其中一个部分可以为空，在此情况下，另一部分就可占用整个页。两个部分中只有一部分包含有效虚地址。在另一部分中的地址则被说明为无效的。如果企图引用一无效虚地址，则将被硬仲捕捉到。这种方案的优点是只需为页的有效部分分配物理存储空间。

2.9 页说明寄存器

页说明寄存器规定：
1) 该页低地址部分的长度(实际存储的是 32 字块数－1)；
2)“扩充方向位”，当高地址部分是有效部分时该位被设置为 1；
3) 存取方式字段，它定义“不存取”、“只读”或“读/写”。
注意，若有效部分为零，则应将存取方式字段设置为“不存取”。

2.10 存储分配

页的有效部分应当对应于物理存储器的某一区间，但是硬件并不规定此种存储区间的分配方式(除了这种区间应当在 32 字边界上开始和结束外)。这些区间可以以任何顺序进行分配，而且可以相互重迭。
实际上，物理存储器的分配是相当有规律的，我们将在第 7 章中看到这一点。相关页对应的物理存储区间通常是按页号顺序连续分配的，这样就使得一道程序的各段最多映照到物理存储器中的两个区间。

2.11 状态寄存器

除段寄存器外，PDP11/40 还有两个存储管理状态寄存器：
SR0 包含异常终止出错标志和其他有关操作系统的重要信息。特别地，当 SR0 的第 0 位设置时，存储管理起作用。
SR2 在每条指令存取操作的开始处装入 16 位虚地址。

2.12 “l”和“d”空间

PDP11/45 和 11/70 系统增加了段寄存器组。pc 寄存器(r7)引用的地址被称为属于“i”空产，它由一组段寄存器翻译为物理地址；其余址被称为属于“d”空间，它由另一组段寄存器翻译。这种安排的优点是“i”和“d”空间可分别为 32K 字，于是可分配给程序的最大地址空间较 PDP11/40 扩大了 1 倍。

2.13 启动条件

当系统第一次启动时，存管部件被禁止操作，处理机处于核心态。
在这种环境下，变化范围在 0～56K 字节的虚地址映照为相同的物理地址值。但是，虚地址空间的最高页映照为物理地址空间的最高页，在 PDP11/40 或 11/45 中，地址区间
0160000－0177777
被映照为下列物理地址区间
0760000－0777777

2.14 专用设备寄存器

物理存储器的最后一页保留用于处理机及各外部设备的各专用寄存器。这种处理方式使存储器空间减少了 1 页，但是却不再需要提供特殊的指令类型来存取各设备寄存器。
为寄存器在此页中分配地址的方法是一种创新，其价值是不容置疑的。

第 3 章 阅读“C”程序

学习阅读用“C”语言编写的程序是必须的，否则就无法阅读 UNIX 的源代码。

与自然语言相类似，读比写要容易一些。即使如此，你仍然需要非常细心地学习那些巧妙之处。
与“C”语言直接相关的“UNIX 文献”有两种，它们是：
《C 参考手册》，作者是 Dennis Ritchie
《C 语言程序设计导引》，作者是 Brian Kernighan
从现在开始，你就应当阅读这些文献，并且要尽可能一次又一次地反复阅读它们，这样才会愈来愈深入地了解 C 语言。
学习 UNIX 源代码并不要求你能够编写“C”程序。但是，如果有机会，那么你应当至少编写一些小的 C 语言程序。这是一种学习程序设计语言的良好方法，使用这种方法你会较快地理解如何正确使用下列的“C”语言组成部分：
分号；
“＝”和“＝＝”；
“｛”和“}”；
“＋＋”和“－－”；
说明；
寄存器变量；
“if”和“for”语句；
等等
你将发现“C”是一种在存取和处理数据结构和字符串方面非常方便的语言，而这种操作占了操作系统相当大的部分。作为一种面向终端的要求简明而紧凑表达式的语言，“C”使用一种大字符集，并且使很多符号，例如“*”、“&”等得到很频繁的使用。在这方面“C”语言可与 APL 相比。
“C”语言的很多特征令人想到 PL/1，但“C”在结构化程序设计方面提供的设施比 PL/1 要强得多。

3.1 某些选出的例子

下面的例子都直接选自 UNIX 源代码。

3.2 例 1

最简单的过程并不做任何实际工作，在源代码中这种过程出现了两次，它们是“nullsys”(2864)和“nulldev”(6577)。
其中并没有任何参数，但是一对园括号“(”和“)仍旧是需要的。一对花括号“｛”和“}”界定了过程体，在本例中过程体为空。

3.3 例 2

下面一个例子内容稍多一些：
其中增加了一条赋值语句。该语句用一分号终止，它是语句的组成部分，而非 Algol 类语言中的语句分隔符。
“ENODEV”是一个定义的符号，在实际编译之前由编译程序的预处理器将这种符号代换成一个与其相结合的字符串。ENODEV 在 0484 行被定义为 19。UNIX 与此相关的惯例是：定义符号由大写字符组成，而其他符号则使用小写字符。
“＝”是赋值算符，“u.u_error”是结构“u”的一个元素(参见 0419 行)。注意，“.”算符用于从一个结构中选出一个元素。该元素名是“u_error”。在 UNIX 中，结构元素命名通常都用这种方式—开头是一个区分符，然后是一个下划符，最后是一个名字。

3.4 例 3

此过程的功能非常简单：它将指定的字数从一组连续单元复制到另一组连续单元。
w 此过程有 3 个参数。第 2 行

int *from,*to;
说明前两个变量是指向整型的指针。因为对第三个参数没有进行说明，按系统默认这是整型。三个局部变量 a、b 和 c 被指定为寄存器型。寄存器易于存取，引用它们的目标码也比较短。a 和 b 是指向整型的指针，而 c 是一个整型。寄存器型说明也可写成为：
register int *a,*b,c;
这强调了寄存器型与整型的关连。
对以“do”开始的 3 行语句应当仔细观察。如果“b”是指向整型的指针，那么：
*b
表示被指向的整型数。于是，为了将“a”所指向的值复制到 b 所指向的单元，我们可使用下面的语句：
*b=*a;
如果我们将此写成：
b=a;
那么这将使 b 的值与 a 值相同，也就是“b”和“a”将指向同一位置。至少在这里，这并不是所要求的。
从源到目的复制了第一个字后，需要使“b”和“a”的值增 1，以分别指向下一个字。为此可写成
b=b+1;a=a+1;
但是“C”提供了一种简短的表达法(在变量名较长时，这种方法更加有用。)：
b++;a++;
或者
++b;++a;
在这里“b++”和“++b”并无区别。
但是，“b++”和“++b”可用作表达式中的一部分，在此种情况下，它们的作用就会不同。在两种情况下，对 b 增 1 的作用是相同的，但对于 b++而言，进入表达式的值是其初始值，而对于++b 则进入表达式的值是其最后值。
除减 1 外，“－－”算符遵守与“＋＋”相同的规则。于是，“－－C”使进入表达式的值是 C 减 1 后的值。
“＋＋”和“－－”算符是非常有用的，在 UNIX 源代码中到处都用到这两个算符。注意，在
*b++和(*b)++
之间是有区别的。
如同应用于简单数据类型一样，这两个算符也可用于指向结构的指针。如果一个指针说明为引用一个特定的结构类型，那么对其实施增量操作，该指针的实际值增加结构的长度。
现在，我们可以观察下列行的作用：
*b++=*a++;
它先复制了字，然后指针值增 1，简简单单完成了这一切。do_while 语句的结尾是：
while(--c);
在园括号中的表达式“－－C”先求值，然后被测试(测试的是减 1 后的值)。若其值非 0，则重复此循环，否则终止循环。
显然，如果 count 的初始值为负，那么此循环决不会正常终止。如果可能产生此种情况，那么就必须对这种程序进行修改。
3.5　例 4
参数“f”按系统默认为整型，并直接复制到寄存器型变量“rf”中。这种模式将得到普遍应用，

因此读者会对此熟悉起来，我们在此也就无需再多作说明。)
下面三个是简单关系表达式：
rf＜0 rf＞=NoFILE fp!=NULL
若它们为真，则取值 1，若为假，则取值 0。第一个表达式测试“rf”的值是否小于 0；第二个表达式测试“rf”值是否大于或等于 NOFILE”的定义值，第三个表达式测试“fp”的值是否不等于“NULL”(“NULL”定义为 0)。
由“if”语句测试的条件是包含在园括号中的算术表达式。如若此表达式大于 0，则该测试成功，然后执行后面的语句。于是，若“fp”的值是 001375，那么
fp!=NULL
为真，作为算术表达式中的一项，其值取为 1。因为此值大于 0，因此将执行语句：
return(fp);
从而终止“getf”的执行，并将“fp”的值作为“getf”的执行结果返回给调用过程。
表达式
rf＜０‖rf＞＝ＮＯＦＩＬＥ
是两个简单关系表达式的逻辑或。
“goto”语句及相关标号的一个实例将在后面说明。
u-ofile 是嵌入在结构 u 中的一个整型数组，其第 rf 个元素的值赋予“fp”。
过程 getf 将一个值返回给调用过程。该值或者是“fp”值(这实际上是一个地址)或者是 NULL。
3.6　例 5
本例与上一个例子有许多相似之处，但是本例中有一个新概念—一个结构数组。本例中使人感到有些困惑的是数组和结构都被称为“proc”(是的，“C”语言允许这样做)。该结构和数组说明为下列形式：
“p”是指向一结构的指针型的寄存器变量，该结构的类型为“proc”。
p=&proc〔0〕;
将 proc 数组第一个元素的地址赋予“p”。其中的算符“&”在这里的意义是“的地址”。
注意，若一数组有几个元素，则这些元素的下标号为 0、1、“(n-1)。“C”语言允许将上面的语句写成如下更简单的形式：
p=proc;
在“do”和“while”之间有两条语句。其中，第一条语句可改写成下列较简单的形式：
if(p－＞p_wchan==C)setrun(p);
亦即，省略了原来的花括号，“C”是一种自由格式语言，行之间的文本安排并不重要。
下列语句
setrun(p);
调用过程“setrun”，它将“p”的值作为参数传送给 setrun。(所有参数都传送值。)
下列关系
p－＞p_wchan==C
测试“c”的值和“p”所指向结构的元素“p_wchan”的值是否相等。注意，如果将此写成下列形式，则错了：
p·p_wchan==c
因为“p”并不是一结构的名字。
“do”和“while”之间的第二条语句是 p++;它将“p”的值增加“proc”结构的长度。(编译可以计算出此值。)
为了正确地进行这种计算，编译需知道“p”所指向的结构种类。当不需要作这种考虑时，你将发现在类似情况中，将把“p”简单地说明为

register *p;
这对程序员来说会容易一些，而编译程序则并不坚决要求这样做。
这个过程的后面部分可以等效地写成下列形式，但其效率会差一些：

```
……
i=0;
do
   if(proc〔i〕.p_wchan==c)
      setrun(Qproc〔i〕);
while(++i＜NPROC);
```

3.7　例 6
此过程检查是否已出错，若出错指示器“u.u_error”没有设置，则将其设置为通用出错指示(“EIO”)。
“B_ERROR”的值为 04(请查看 4575 行)，所以只设置 31 位，可将其用作屏蔽字以提取第 2 位的值。用在下列表达式

```
bp－＞b_flags & B_ERROR
```

中的算符“&”对两个算述值实施按位逻辑与。
如果“bp”指向的“buf”结构的元素“b_flags”的第 2 位已设置，则上述表达式的值大于 1。
如果已有一个错，则表达式

```
(u.u_error=bp－＞b_error)
```

被求值，然后与 0 相比较。在此表达式中包括了一赋值符“＝”。在“bp－＞b_error”的值赋予“u.u_error”后，该表达式的值是 u.u_error 的值。
将赋值用作表达式的一部分是非常有用的，而且是常用的。
3.8　例 7
在本例中，应当注意过程“suser”返回一值，然后“if”语句对其进行测试。若测试结果为真，则执行在“{”和“}”中的三条语句。
注意，在不带参数调用一过程时，仍旧应当写一对空园括号，例如：

```
suser( )
```

3.9　例 8
“C”提供了一条件表达式。若“a”和“b”是整型变量，那么：

```
(a＞b？a:b)
```

是一个表达式，其值是 a 和 b 中的较大者。
但是如果“a”和“b”被视为不带符号整型数，则此表达式并不能正常工作。为此可使用下一过程
其中的窍门是将“a”和“b”说明为指向字符的指针，在进行比较时起无符号整型的作用。
该过程体也可写成如下形式：
但是由于“return”本身的性质，使得“else”在这里并无作用。
3.10　例 9
下面是两个短过程，它们包含了不同的、看来有些陌生的表达式。第一个是：
其中，u_dirp 在结构 u 中说明为

```
char *u_dirp;
```

“u.u_dirp”是一字符指针。因此“*u.u_dirp++”的值是一字符。(指针增量是副作用。)
当将一字符装入 16 位寄存器中时，将进行符号位扩展。将其与 0377 作按位与，则额外的高位都被消除，于是只返回一字符值。
注意，以 0 开始的任一整型数(例如 0377)都被解释为八进制整型数。

第 2 个例子是：
其返回值是：“n 除以 128，然后取整为下一个整数”。
注意，使用右移操作“》”优于除操作“／”。
3.11　例 10
上面说明过的很多要点集合在下一例子中：
请思考这一过程做了些什么以检查你对“C”的理解程度。
关于此过程可能有两个附加的特征需要了解：
?“&&”是用于关系表达式的逻辑与。(“前面引进的“‖”则是关系表达式中的逻辑或。)
?最后一条语句包含表以式
& runout
在语法上这是一个地址变量，但在语义上只是一个独特的位模式。这是一个在 UNIX 中处处都要使用的方法实例。程序员由于某种特殊的目的需要一独特的位模式。只要它是独特的就满足了要求，其精确值则并无意义。对此问题的一个解决方法是使用一个适当的全局变量的地址。
3.12　例 11
最后第 2 条语句值得引起注意，它原本可以写成为下形式：
rbp－＞b_flags=rbp－＞b-flag｜B_ASYNC;
在此语句中，位屏蔽“B_ASYNC”与“rbp－＞b_flags”相“或”。符号“｜”是算术值的逻辑或。
这是一个在 UNIX 中非常有用的结构的实例，这种结构可以节省程序员的很多劳动。如果图是任一二进制算符，则：
x=xa;
(其中，“a”是一表达式)可被改写成更加简洁的如下形式：
x=x*a;
使用这种结构的程序员应当注意空白字符的位置。因为
x=+ 1;
不同于
x= +1;
请考虑下列表达式的意义：
x=+1；
此表达式具岐义性，其结果与编译的具体实现有关。
3.13　例 12
本例引入了“for”语句，这是一个非常通用的语法结构，功能强，表达紧凑。
“for”语句的结构在“C Tutorial”的第 10 页中有清晰的描述，在此不再复述。
“for”语句等效的 Algol 结构是：
for i:=1step1　until NOFILE-1 do
“C”语言“for”语句的强大能力来自于它给予了程序员很大的自由，可以选择在花括号中到底放些什么。
3.14　例 13
在本例的“for”语句中，指针变量“p”逐个指向“proc”数组中的每一个元素。
实际上，原先的代码是：
for(p=&proc〔0〕;p＜& proc〔NPROC〕;p++)
但是这超过了本书 UNIX 源代码页中的行长。正如前面所指出的，用“proc”代替表达式“& proc〔0〕”在此上下文中是可以接受的，因此将原先的代码改写成现今的形式。

这种“for”语句在 UNIX 中几乎随处可见，所以应当仔细学习它。将其读成
for p=each process in turn(顺序指向每一进程)
注意，“& proc〔NPROC〕”是“proc”数组第 NPROC＋1 个元素的地址(当然该元素实际并不存在)，也就是说，这是该数组所占用存储区后的第一个单元的地址。
我们再次指出：在前一个例子中
i++
表示“向整型数 i 加 1”，而在本例中
p++
表示“将 p 移动指向下一个结构”。
3.15　例 14
这是一个“while”语句的例子，应当将此语句与前面介绍过的“do...while...”结构进行比较(请对照 Pascal 的“while”和“repeat”语句)。
该过程的意义是：
当“cpass”的返回结束为正时，不断调用它，并将该结果作为参数调用“lpcanon”。
注意，在变量“C”的说明语句中使用了“int”，虽然这并不必要，但有时也并不省略!
3.16　例 15
下面的例子是从原先的代码中简化出来的：
注意对两个字的数组“n”的说明方式以及“getf”的使用(它出现在例 4 中)。
依赖于括号中表达式的值，“switch”语句作多路转移选择。每个可转移部分都有“case 标号”。
如果“t”的值是 1 或 4，则顺序执行一组操作。
如果“t”的值是 D 或 3，则不执行任何操作。
如果“t”的值是其他，则执行标号“default”后的一组操作。
注意，用“break”跳出“switch”语句，从而执行紧跟 switch 语句的下一条语句。不使用“break”，则按“switch”语句中的正常执行序列执行。
于是，在“default”操作的尾端通常要使用“break”。因为“default”后的几个 case”实际并不执行任何操作，所以在这里省略“break”是安全的。
这两组操作实施一个 32 位整数与另一个的办法。“C”编译的稍后版本将支持“long”(“长”)类型变量，这将使这种代码更易于编写和阅读。
也应注意在表达式
fp－＞f_inode→i_size0
中进行了两级间址操作。
3.17　例 16
在本例中有一些有趣的特征。
对“d_major”的说明是：
这使得赋予“maj”的值是赋予“dev”值的高字节。
在本例中，“switch”语句只有两个非空 case，没有“default”。两个非空 case 中的操作，例如
(*bdevsw〔maj〕.d_close)(dev,rw);
初看起来很难理解。
首先应当注意到这是一个过程调用，参数为“clev”和“rw”。
其次，“bdevsw(以及“cdevsw”)是结构数组，其“d_close”元素是一指向函数的指针，亦即：
bdevsw〔maj〕
是结构名，而
bdevsw〔maj〕.d_close
是该结构的一个元素，它是一个指向函数的指针，于是：

*bedsw〔maj〕.d_close
是一函数的名字。第一对园括号是“语法上的糖衣”，它帮助编译程序正确了解它所处理项的位置。
3.18　例 17
我们以本例结束本章
其中，“switch”依据“n”的值进行转移位置选择，然后执行一组操作。
当然也可以用很长的“if”语句来代换“switch”语句，例如：
if(n==SIGQIT ‖ n==SIGINS ‖ …
　　… ‖ n==SIGSYS)
这既不清晰，效率也不高。
注意一个八进制数与“n”相加的方法，以及用两个 8 位值如何构成一 16 位值。

第 4 章　概　　述
本章的目的是对源代码作为整体进行综述，亦即先看森林，后看树木。
查看源代码将发现它由 44 个文件组成，其中：
?两个是汇编语言文件，它们的名字带后缀“.s”；
?28 个是用“C”语言编写的，它们的名字带后缀“.C”；
?14 个是用“C”语言编写的，但并不准备单独编译，名字带后缀“.h”。
这些文件以及它们的内容是由编写它们的程序设计人员为使他们自己使用方便而安排的，并没有考虑读者阅读的便利。在很多方面，文件之间的划分与本书的讨论无关。
如同在第 1 章中已提及的，所有这些文件已被组织成五个部分。在组织这些部分时所遵循的基本点是：尽可能使各部分的长度大致相近；将强相关的文件分在同一部分中；将弱相关的文件分到不同部分中。
4.1　变量分配
PDP11 系统结构支持对变量进行高效存取，其条件是这些变量的绝对地址已知，或者它们相对于栈指针的地址可在编译时完全确定。
PDP11 在硬件方面并无对变量说明的多词法级的支持，在块结构语言，例如 Algo(或 Pascal，则有变量说明的多词法级支持。由于“C”在 PDP11 上实现，所以它只支持两个词法级：全局(global)和局部(local)。
对全局变量进行静态分配；对局部变量则在为前栈中或在通用寄存器(r2、r3 和 r4 用于此种方式)中进行动态分配。
4.2　全局变量
除极个别例子外，对全局变量的说明都已集中到各个“.h”文件中。例外包括：
1) 在“swtch”中说明的静态变量“p”(2180)，它以全局方式存储，但只能在“swtch”过程中存取。(在 UNIX 中，“p”是一个普通使用的局部变量名。)。
2) 有些变量，例如“swbuf(4721)，仅由同一文件中的若干过程引用，于是在该文件的开始部分对它们进行说明。
可在引用全局变量的每个文件中分别对它们进行说明。然后，装入程序(loader)在将编译好的各程序文件连接到一起时，将每一变量的各个说明相匹配。
4.3　“C”预处理程序
如果全局性说明在每个文件必须全部重复(正如在 Fortran 中所要求的那样)，那么程序量会显著增加，而且对一个说明进行修改是非常枯燥无味的，而且非常容易出错。
UNIX 使用了“C”编译的预处理程序设施，从而避免了这些困难。这种设施允许将大多数全局变量的说明记录在少数几个“.h”文件之中，每个全局变量记录一次。

任何时候若需一特定全局变量的说明，则只要将相应“.h”文件包含(“include”)在将被编译的文件中。
UNIX 也使用“.h”文件作为载体存放标准符号名的列表以及某些结构类型的说明。符号名常代表常数和可调整的参数。
例如，若文件“bottle.c”包含一过程“glug”，而该过程引用一名为“gin”的全局变量，该全局变量在文件“box.h”中说明，那么下列语句应插入到文件“bottle.c”的开始处：
#include “box.h”
当编译该文件时，所有在“box.h”中的说明都被编译，因为它们是在“bottle.c”中的任一过程之前发现的，所以在所产生的可重定位模块中它们被标记为外部的。
当将所有目标模块连接到一起时，在其源文件中包括了“box.h”的每个文件中都将发现对“gin”的引用。所有这些引用将是一致的，装入程序将为“gin”分配一个所需的地址空间，并相应调整所有对“gin”的引用。

4.4 第一部分

第一部分包含了很多“.h”文件和汇编语言文件。它也包含若干涉及系统初启和进程管理的文件。

4.4.1 第 1 组“.h”文件

param.h,不包含变量说明，但包含很多操作系统常数和参数的定义，以及三个简单结构的说明。有关约定是对于定义的常数只使用大写字母。
systm.h〔第 19 章〕，主要由说明组成，其中作为副作用定义了“Callout”和“mount”结构。注意，其中没有一个变量是显式地赋予初值的，它们的初值都赋为 0。
头三个数组的长度是定义在“param.h”中的参数。因此，任一包含“systm.h”的文件，必须在包含“systm.h”之前先包含“param.h”。
seg.h，包含几个定义和一个说明，在引用段寄存器时使用这些定义和说明。此文件实际上可并入“param.b”和“systm.h”中。
proc.h〔第 7 章〕，包含重要的“proc”的说明。proc 是一个结构类型，也是这种结构类型的一个数组。proc 结构的每一个元素的名字都以“p-”开始，没有其他变量是这样命名的。类似的惯例用于命名其他结构的各元素。
proc 结构中头两个元素“p_stat”和“p_p_flag”的各种相关值都有单独的名字，它们也在“proc.h”中定义。
user.h〔第 7 章〕，包含非常重要的“user”结构的说明，还包含一组与“u_error”相关的定义值。在任一时刻只有一个“user”结构的实例是可存取的。该实例以名字“u”引用，它位于长度为 1024 字节的“每个进程数据区”的低地址部分。
一般而言，本书在此后不会对“.h”文件进行详细而全面的分析。希望读者能反复地参阅它们，这样就会不断增加对它们的熟悉和理解程度。

4.4.2 汇编语言文件

有两个汇编语言文件，大约占源代码中的 10%。应当对这些文件有较高的熟悉程度。
low.s〔第 9 章〕，包含初始化内存低地址部分的有关信息，包括陷入矢量。此文件是由名为“mkconf”的公用程序生成的，以适应在特定配置中具有的外部设备集。
m40.s〔第 6、8、9、10、22 章〕，包含一组与 PDP11/40 相适应的例程，它们执行大量“C”不能直接实现的特殊函数。我们将在适当的时间和位置对此文件的各部分进行介绍和讨论。(最大的汇编语言过程“backup”已作为一个练习留给读者分析。)
当使用 PDP11/45 或 PDP11/70 时，应将“m40.s”代换成“m45.s”但在这里我们没有提供此文件。

4.4.3 在第一部分中的其他文件

main.c〔第 6、7 章〕，包含“main”，它执行多种初始化任务以使 UNIX 运行。它也包含“sureg”和“esfabur”，它们设置用户态段寄存器。
slp.c〔第 6、7、8、14 章〕，包含进程管理所需的各主要过程，包括：“newproc”、“sched”、“slcep”和“swtch”。
prf.c〔第 5 章〕，包含“panic”和若干其他过程，它们提供了一种简单的机制向操作人员显示初始化消息和出错消息。
malloc.c〔第 5 章〕，包含“malloc”和“mfree”，它们的功能是对存储资源进行管理。

4.5 第二部分

第二部分涉及陷入(trop)、硬件中断(hardware inTerrupt)和软件中断(software interrupt)。
陷入和硬件中断造成 CPU 正常指令执行序列的突然切换。这提供了一种处理在 CPU 控制外产生的特殊条件的机制。
这种设施也用作“系统调用”机制的一部分，在“系统调用”机制中，一用户程序可以执行一条“trap”指令以故意造成一次陷入，从而取得操作系统的注意和帮助。
软件中断(或称“信号”)是一种特殊的进程间进行通信的机制，在产生“坏消息”时使用。
reg.h〔第 10 章〕，定义了一组常数，当前用户态寄存器值存放在核心态栈中而且需要引用它们时，使用这一组常数。
trap.c〔第 12 章〕，包含“C”过程“trap”，其功能是识别并处理各类陷入。
sysent.c〔第 12 章〕，包含数组“sysent”的说明及初值表，“trap”使用此数组按系统调用类型找到在核心态下执行的相应例程的入口地址。
sys1.c〔第 12、13 章〕，包含多个与系统调用相关的例程，包括：“exec”、“exit”、“wait”和“fork”。
sys4.c〔第 12、13、19 章〕，包含“unlink”、“kill”及若干其他非主要系统调用的例程。
clock.c〔第 11 章〕，包含“clock”例程，其主要功能是处理时钟中断，并且进行偶发事务及会计事务的大量处理。
sig.c〔第 13 章〕，包含处理“信号”或“软件中断”的过程。这些为进程通信及跟踪提供了能力。

4.6 第三部分

第三部分涉及内存(主存)和磁盘存储器之间的基本输入/输出操作。这些操作对程序换进/换出以及磁盘文件的创建和引用活动是基础性的。
本部分也包含使用和处理大缓存(512 字节)的各个过程。
text.h〔第 14 章〕，定义“text”结构和数组。系统使用一个“text”结构定义一共享正文段(sbared text segment)的状态。
text.c〔第 14 章〕，包含管理共享段的各个过程。
buf.h〔第 15 章〕，定义“buf”结构和数组、“devtab”结构以及“b-error”可包含的各种值的符号名。为管理大缓存，所有这些都是必须的。
conf.h〔第 15 章〕，定义结构数组“bdevsw”和“cdevsw”，它们指定了面向设备的各处理过程，为执行针对设备的逻辑文件操作，需调用这些过程。
conf.c〔第 15 章〕，与“low.s”相似，“conf.c”是由公用程序“mkconf”生成的，以适应特定系统配置的各种外部设备。它包含“bdevsw”和“cdevsw”数组的初始化值表，这两个数组中设置的各个处理过程控制基本 I/O 操作。
bio.c〔第 15、16、17 章〕，是“m40.s”之后的最大一个文件，它包含了处理大缓存以及面向块的基本 I/O 的各个过程。
rk.c〔第 16 章〕，是针对 RK11/RK05 磁盘控制器的设备驱动程序。

4.7 第四部分

第四部分与文件及文件系统有关。
文件系统由一组文件和相关的表，以及目录组成，所有这些都组织存放在单个存储设备，例如磁盘包上。
本部分包含下述功能：
?创建和存取文件。
?通过目录定位文件。
?组织并维护文件系统。
本部分也包括与特种文件“pipe”有关的代码。
file.h〔第 18 章〕，定义“file”结构和数组
filsys.h〔第 20 章〕，定义“filsys”结构，在装配和拆卸文件系统时，从超级块(super block)复制至该结构或反之。
ino.h，说明记录在“已装配”设备上的“索引节点”(“inode”)结构。此文件并不被任何其他文件包括，仅仅是为提供信息而存在。
inode.h〔第 18 章〕，定义“inode”结构和数组。在管理进程对文件的存取时，“inode”起关键性作用。
sys2.c〔第 18、19 章〕，包含一组与系统调用相关的例程，这些系统调用是“read”、“wrik”、“creat”、“open”和“close”。
sys3.c〔第 19、20 章〕，包含一组与多个非主要系统调用相关的例程。
rdwri.c〔第 18 章〕，包含与读/写文件有关的中间层次例程。
subr.c〔第 18 章〕，包含与 i/o 有关的中间层次例程，特别是“bmap”，它将逻辑文件指针翻译成物理磁盘地址。
fio.c〔第 18、19 章〕，包含与文件打开、关闭和存取控制有关的中间层次例程。
alloc.c〔第 20 章〕，包含若干管理文件系统 inode 和磁盘块资源的过程，它们负责分配“inode”数组表目项和磁盘存储块。
iget.c〔第 18、19、20 章〕，包含与更新和引用“inode”有关的各过程。
nami.c〔第 19 章〕，包含“namei”过程，它搜索文件目录。
pipe.c〔第 21 章〕，是“pipe”(“管道”)的“设备驱动程序”，“pipe”是特殊形式的短磁盘文件，用于从一个进程向另一个进程传输信息。

4.8 第五部分

第五部分是最后一部分。它与低速面向字符的外部设备的输入/输出相关。
这类设备共享一个公共缓冲池，该缓冲池由一组标准的过程进行管理和操作。
面向字符的外设用下列设备作为实例：
KL/DL11 交互式终端
PC11 纸带阅读/穿孔机
LP11 行式打印机
tty.h〔第 23、24 章〕，定义“clist”结构(用作字符缓冲队列的队首)和“tty”结构(存放控制某单个终端的相关数据)，说明“partab”表(用于控制单个字符向终端的传输)，也定义很多相关参数的符号名字。
kl.c〔第 24、25 章〕，是经 KL11 或 DL11 接口相连终端的设备驱动程序。
tty.c〔第 23、24、25 章〕，包含若干独立于接口的公共过程，它们控制向/从终端的传输，也考虑到各种终端的特殊性。
pc.c〔第 22 章〕，是 PC11 纸带阅读/穿孔控制器的设备处理程序。
mem.c，包含一组存取主存的过程，它们将主存视作为普通文件。这部分代码作为一个练习留给读者进行分析。

第 5 章　两 个 文 件

本章从易处着手介绍第一部分中的两个文件，它们可与本书其他部分较清晰地隔离出来。对这些文件的讨论补充了第 3 章中对“C”语言的概括介绍，包括了若干与“C”语言语法和语义有关的注解。

5.1　文件 malloc.c

此文件仅由两个过程组成：

malloc(2528),mfree(2556)

这两个过程涉及两类存储资源的分配和释放，这两类存储资源是：

?主存：它以 32 个字(64 字节)为单位。

?盘交换区(disk swep area)：它以 256 字(512 字节)为单位。

对于这两类资源的每一类，各有一些资源图(“coremap”或“swapmap”)记录可用区列表。指向相应资源图的一个指针作为参数传递给“malloc”和“mfwe”，于是这两个过程无需了解它们正处理的资源的类型。

“coremap”和“swapmap”都是类型为“map”的结构数组，“map”在 2515 行说明。该结构由两个字符指针，亦即无符号整型组成。

对“coremap”和“swapmap”的说明分别位于 0203 行和 0204 行。在这里，完全忽略了“map”结构—这是一种程序设计捷径，由于装入程序并不检测这一点，所以可以进行这种形式的说明。在“coremap”和“swapmap”中的实际列表元素数分别是“CMAPSIZ/2”和“SMAPSIZ/2”。

5.11　列表维护规则

1) 每个可用存储区由其长度和相对地址定义(按相应资源的单位计算)。

2) 每个列表中的各元素总是按相对地址从低到高排列。任何两个相邻列表元素所描述的可用存储区如果构成一个连续可用区，则总是立即将它们合并成一个区。

3) 从第一个元素开始顺序逐个查看就能扫描整个列表，当查看到一个零长元素时结束扫描。此最后一个元素并不是有效的列表部分，而是该列表有效部分的结束标达。

上述规则提供了对“mflee”完整的规格说明，提供了对“malloc”完整规格说明，但有一个方面例外。此例外方面是：

当有多种方式可进行资源分配时，我们须说明在具体实现时选择哪一种。

“malloc”所采用的方法是“首次适配”算法(“First Fit”)。采用这种算法的理由读者以后会体会到。

下面举一个例子说明资源图是为何维护的，假定下列 3 个资源区是可用的：

?一个可用区的长度为 15，在位置 47 处开始，在 61 处结束。

?一个可用区的长度为 13，在位置 27 处开始，在 39 处结束。

?一个可用区的长度为 7，在位置 65 处开始。

于是，资源图应当包含：

项	长　　度	地　　址
0	13	27
1	15	47
2	7	65
3	0	??
4	??	??

如果接到一个长度为 7 的存储空间请求，则从位置 27 处开始分配，分配后资源图变成：

项	长　　度	地　　址
0	6	34

1 15 47
2 7 65
3 0 ??
4 ?? ??
如果从地址 40 开始到 46 结束的存储区返回至可用区列表，则资源图变成：
项 长 度 地 址
0 28 34
1 7 65
2 0 ?
3 ?? ??
注意，虽然可用存储资源的总量增加了，但是由于 3 个可用存储区(其中 1 个是刚释放的区)合并，该列表中的元素数反而减少 1。
现在让我们转过头来考虑实际源代码。
5.1.2 malloc(2528)
此过程体由一“for”循环组成，它搜索“map”数组，直至：
1) 到达可用资源列表的尾端；或
2) 搜索到一个区，其长度能满足当前请求。
2534：“for”语句首先对“bp”赋初值，使其指向资源图的第一个元素。在每一次循环时，“bp”增 1 以指向下一个“map”结构。
注意：循环继续条件“bp－＞m_size”是一表达式，当引用资源图结束标志元素时，该表达式值为 0。该表达式也可等效地写成更明显的形式“bp－＞m_size＞0”。
还要注意：对数组的结尾没有进行显式测试。(假定 CMAPSIZ，SMAPSIZ＞＝2*NPROC，那么这种测试是不必要的!)
2535：若该列表元素定义了一个区域，其长度至少与所要求的相等，则……
2536：记住该区第一个单元的地址。
2537：增加存放在该数组元素中的地址。
2538：减少存放在该数组元素中的长度，并将结果与 0 比较(也就是检查是否确好适配)。
2539：如果精确适配，则将所有后续元素(直至并包括结束标志元素)都下移一位置。
注意：“(bp-1)”指向“bp”所引用的前一个结构。
2542：“while”的继续条件并不测试“(bp-1)－＞m_size”和“bp－＞m_size”是否相等。
被测试的值是由“bp－＞m_size”赋予“(bp-1)－＞m_size”的值。
(你如果没有立即识别这一点，也是可以宽恕的。)
2543：返回该区域的地址。return 语句结束了“malloc”过程，因此非常肯定也就结束了“for”循环。
注意：返回值 0 意味着“不幸”。这基于下列事实：没有一个有效区会在单元 0 处开始。
5.1.3 mfree(2556)
此过程将一个存储区返回给由“mp”指定的“资源图”，该区的长度为“size”，其起始地址为“aa”。此过程的体由一行“for”语句和占多行的一条“if”语句组成。
2564：本行尾端的分号是特别重要的，它终止了一条空语句。(如果如同 2394 行那样使此分号单独占 1 行，则会更清晰一些。)
此语句例示了“C”语言的能力或其晦涩。请试看用其他语言，例如 Pascal 或 PL/1,编写与此语句等效的代码。
步进“bp”直至遇到一元素：
?该元素的地址大于所返回区的地址，亦即不满足条件：

“bp－＞m_addr＜＝a”，或者
?该元素是列表结束标志元素，亦即它不满足条件：
“bp－＞m_size!=0”;
2565：我们已找到在其之前应插入新列表元素的元素。问题是：该列表是增大一个元素，还是由于合并，该列表仍保持原先的元素数，甚至其元素数减少 1?
如果“bp＞mp”，那么我们就不会在列表开始处插入。如果
(bp-1)－＞m_addr+(bp-1)－＞m_size==a
那么正被返回的存储区与列表中前一元素描述的存储区紧相邻接。
2566：将前 1 列表元素的长度增加正被返回区的长度。
2567：正被返回区是否与列表中下一个元素紧相邻接。
如果是这样……
2568：将下一列表元素的长度加至前一元素的长度。
2569：将所有余下的列表元素向下移动 1 个位置，直至并包括该列表结束标志元素。
注意：若 2567 行的测试在“bp－＞m_size”为 0 时意外地产生结果为真的情况，那也不会造成任何损害。
2576：若 2565 行的测试失败，则执行此语句，亦即正被返回的区不能与列表中的前一元素区相合并。
这能否与下一元素区合并呢?注意本语句中对下一元素是否为空(null)的检查。
2579：假定正被返回的区非空(在进入本过程时就应对此作检查)，则在列表中加一个新元素，然后将所有余下的元素向上(也就是向数组下标增大方向)移动一个位置。

5.1.4 结论

这两个过程的代码编写得非常紧凑。几乎没有可以删除之处以提高运行时的效率。但是，有可能将这些过程写得更清晰一些。
如果你对此抱有同感，那么作为练习请改写“mfree”使其功能更易于辨别出来。
也应注意，“malloc”和“mflee”的正确功能依赖于“coremap”和“swapmap”的正确初始化。执行此种初始化的代码在过程“main”中(1568 行，1583 行)。

5.2 文件 prf.c

此文件包含下列过程：

printf	(2340)	panic	(2416)
printn	(2364)	prdev	(2433)
putchar	(2386)	deverror	(2447)

这些过程之间的调用关系示于图 5-1。
图 5-1

5.2.1 printf(2340)

过程“printf”给操作系统提供了一种向系统控制台终端发送消息的方法，这种方法直接、不复杂、不带缓存。在系统被启、报告硬件出错和系统突然崩溃时使用此过程。
此处的“printf”和“putchar”在核心态下运行，它们类似于但并不等同于由“C”程序调用的库函数“printf”和“putchar”，后者在用户态下运行，位于库“/lib/libc.a”中。此时阅读 UPM 中的“PRINTF(III)”和“PUTCHAR(III)”是有益的。
2340：当程序员为此过程说明各参数时，他一定是走了神。事实上该过程体只引用“X1”和“fmt”。
从此可以看出“C”程序设计在过程调用和过程说明时参数匹配的规则，该规则是不进行两者之间参数的匹配检查，两者的参数数甚至也可以不同。
参数以逆序放到栈上。当调用 printf 时，“fmt”与“X1”比较，其位置更接近于栈顶，其他参

数的情况依此类推(见图 5-2)。
“X1”的地址高于“fmt”，但低于“X2”，其原因是在 PDP11 中栈向下生长，亦即向低地址方向扩展。
2341：“fmt”可解释为常数字符指针。此说明也(几乎)等同于：
“char *fmt;”
其区别是：这里的“fmt”值是不能更改的。
2346：将“adx”设置成指向“X1”。表达式“&X1”是“X1”的地址。注意，“X1”占用的是栈单元，所以在编译时不能对此表达式求值。
随处可见的很多涉及变量或数组的表达式地址是很有效的，它们可以在编译或装入时求值。
2348：从格式字符串顺次取出字符并送入寄存器类变量“C”。
2349：若“C”不是“%”则……
2350：若“C”是－null 字符(“＼0”)，则表示格式字符串以正常方式结束，于是“printf”终止。
2351：否则调用“putchar”将该字符发送至系统控制台终端。
2553：见到一个“%”字符。取下一个字符(最好取得的字符不是“／0”!)。
2354：若此字符是“d”、“l”或“o”，则调用“printn”，传送给它的参数是两个，一个是“adx”所引用的值，另一个则取决于“c”的值，若其值为“o”，则该参数值为〞8〞，否则为〞10〞。(从此可见对于“d”和“l”的代码是完全一样的。)
“printn”将一个二进制数按第二个参数所表示的基数转换成一组数字字符。
2356：若格式编辑字符是“s”，则以 null 终止字符串中的所有字符(除 null 外)都被传送至终端。在此种情况下，adx 应当指向一字符指针。
2361：增加“adx”，使其指向栈上的下一个字，亦即指向传送给〞printf〞的下一个参数。
2362：回到 2347 行，继续扫描格式字符串。热衷于结构化程序设计的程序员将会把 2347 行和本行代换成：
“while”(l)｛“　　and　　”｝

5.2.2　printn(2369)

为了按所要求的顺序产生相应数字字符，“printn”递归调用自身。将本过程的代码编写得更有效一些还是有可能的，但不会是更加完善。(无论如何，从实现“putchar”的观点分析，效率在这里几乎是不必考虑的。)
假设 n=A*b+B,其中：
?A=ldiv(n,b)而且，
?B=lrem(n,b),0＜＝Ｂ＜b。
那么，为了显示 n 的值，我们需要先显示 A 的值，其后跟随 B 的值。
对于 b=8 或 b=10，后者是容易做到的：它只是一个单一字符。如果 A=0，那么前者是容易的。如果递归调用“printn”，那么总能达到 A=0 的情况。因为 A＜n,递归调用链一定会终止。
2375：对应于数字的算术值可方便地变换成相应的字符表示，方法是将算术值与字符‘0’相加。
过程“ldiv”和“Lrem”将它们的第一个参数处理为一个无符号的轻型数(亦即，在实际除法操作之前将 16 位值扩展为 32 位时，不进行符号位扩展)。这两个过程分别从 1392 行和 1400 行开始。

5.2.3　putchar(2386)

本过程将作为参数的字符传送至系统控制台。这是一个小的实例，从中可以观察到 PDP11 计算机 I/O 操作的基本特征。
2391：“SW”在 0166 行定义的值是“0177570”。这是一个只读处理器寄存器的地址，在该寄存器中存成控制台开关寄存器的设定值。

此语句的意思是清晰的：取得 0177570 单元的内容，然后检查它们是否为 0。问题是将其表示成“C”语言形式。代码

if(SW==0)

并没有实现上述要求。很清楚，SW 是一个指针的值，应当用其进行间接访问。编译程序是否能接收下列形式的语句呢?

if(SW－＞==0)

但这在语法上是不正确的。在 0175 行说明了一个伪结构，它是一个元素“integ”，使用此结构程序员找到了一个解决上述问题的方法。

在这一过程和其他地方可以找到许多其他类似的实例。

使用硬件术语进行描述，系统控制台终端接口由 4 个 16 位控制寄存器组成，它们在单总线上连续编址，其起始地址是核心态地址 0177560(参见 0165 行对 KL 的说明)。关于这些寄存器的说明和使用方法请参见《PDP11 外设手册》第 24 章。

使用软件术语进行描述，该接口是一个在 2313 行至 2319 行定义的无名结构，它具有四个元素，它们的名字是 4 个控制寄存器的名字。因为对这种结构无需分配任何实例，所以该结构没有名字并无妨碍。(我们关心的是在 KL 所给予的地址上预定义的东西。)

2393：在发送器状态寄存器(“XST”)的第 7 位为 0 时，不做任何事情，其原因是该接口并没有为接收下一个字符准备就绪。

这是一个“忙等待”(“busy waiting”)的经典实例，在这里处理机无用地反复循环执行一组指令，直至发生某个外部事件。这种处理机能力的浪费通常是不可容忍的，但在本特例中却可接受。

2395：本语句的作用与 2405 行的语句紧密相关。

2397：保存发送器状态寄存器的当前内容。

2398：清除发送器状态寄存器，为发送下一个字符作准备。

2399：清控制状态寄存器的第 7 位，将要发送的下一个字符送入发送器缓冲寄存器这启动了下一个输出操作。

2400：一个“新行”符需伴随一“回车”符，这由递归调用“putchar”实现。

也加进了 2 个额外的“删除”字符，其作用是提供在终端上完成回车操作所需的延迟。

2405：对“putchar”以一个参数 0 进行调用有效地造成 2391 行至 2394 行的再执行。

很难看出为什么程序员在这里选用递归调用，而不选用简单地重复 2393 行和 2394 行。即使不考虑清晰程度，仅考虑代码的效率和紧凑性方面，在这里使用递归调用法也是有害而无益的。

5.2.4 panic(2419)

UNIX 操作系统中有很多位置都要调用本过程。当继续系统的操作似乎是并无希望时将调用本过程。

UNIX 并不自称是一个“容错”或“个别部件发生故障时仍能可靠工作但性能有所下降”的系统。在很多情况下，针对某些很明显的问题调用“panic”是一种行之有效的解决方法。比较复杂的解决方法需要增加大量的代码，这与 UNIX 遵循的哲学“尽量使其简化”是背道而驰的。

2419：本语句的作用在 2323 行开始的注释中说明。

2420：“update”使所有大的块缓存的内容都写到相应块设备上。见第 20 章。

2421：以一个格式字符串和一个参数调用“printf”，该参数是传送给“panic”的。

2422：此“for”语句构成了一个无限循环，该循环中的唯一动作是调用汇编语言过程“idle”(1284)。

“idle”将处理器优先级降为 0，然后执行“wait”。这是一条不做任何事情的指令，它持续无

限长时间。当发生一硬件中断时，“wait”终止。
对“idle”进行无限次调用优于执行一条“halt”指令，其原因是允许正在进行的 I/O 活动得以完成，并且系统时钟能继续工作。操作员可以采用的从“panic”中恢复的唯一方法是重新启动系统(如果愿意，则可在内存转储操作之后)……

5.2.5 prdev(2433)、deverror(2447)

在 I/O 操作中出错时，这两个过程提供警告消息。在此阶段，它们的主要益处是作为使用“printf”的实例。

5.3 包含的文件

应当注意的是：“malloc.c”文件并不要求包含其他文件，但“prf.c”却要求在其开始处包含 4 个文件。
细心的读者会注意到，在文件系统的层次结构中，这 4 个文件所在的层次较“prf.c”高一层。2304 行语句可被理解为在其位置上代换成“param.h”文件的全部内容。这样也就增补了标识符“SW”、“KL”以及“integ”的定义，它们都在“putchar”中出现。
如前所述，“KL”、“SW”和“mtey”的说明分别位于 0165、0166 和 0175 行，但是如果在“prf.c”中没有包含“param.h”，那么它们对“prf.c”中的各过程是没有任何意义的。
“prf.c”中包含了“buf.h”和“conf.h”文件，它们提供了“d_major”、“d_minor”、“b_dev”和“b_blkno”的说明，在“prdev”和“deverror”中使用到它们。
包含第 4 个文件的理由较难发现。从代码本身分析这并不需要，“prf.c”的编写者应当为此对读者表示歉意。在编辑这一部源代码时，很可能将“integ”的说明从“seg.h”搬到了“param.h”。注意，变量“panicstr”(2328)是全局型的，但在“prf.c”之外并不引用它，所以没有将其说明放在“.h”文件中。

第 6 章 系 统 初 启

当 UNIX 重新启动，也就是在一台空闲机器上装入和初启时将发生一系列事件，本章的目的就是考虑这一系列事件的序列。
对初启过程本身的研究是非常有趣的，但更重要的是它能将系统的很多重要特征以有序的方式展现出来。
在系统崩溃之后可能需要重新启动操作系统。由于某些非常普通、操作上的原因，例如过夜之前的关机，也要频繁地重新启动操作系统。如果我们考虑后一种情况，那么可以认为所有磁盘文件都是完好无损的，没有什么特殊情况需要识别和处理。
我们尤其可以设想在根目录中有一个称为“/unix”的文件，它是 UNIX 操作系统的目标代码。该文件起始于一组我们正在研究学习的源文件。将这些源文件分别编译，然后连接到一起构成了一个单一目标文件，然后存放在根目录下的 unix 文件中。

6.1 操作员的动作

重新启动要求操作员在处理机控制台上执行下列操作：
?将“enable/halt”(“启动/停机”)开关设置到“halt”位置，这样就停止了处理机。
?将硬件引导装入程序的地址设置到开关寄存器。
?按下然后释放“装入地址”开关。
?将“enable/halt”开关设置到“enable”位置。
?按下然后释放“start”(“启动”)开关。这样就激活了常驻在处理机中 ROM 里的引导程序。
引导装入程序装入一更大的装入程序(从系统盘的＃ 0 块)，该程序查找并将称为“/unix”的文件装入到内存的低地址部分。然后，它将控制转移到已装入＃ 0 地址的指令。
地址 0 单元中是一条转移指令(0508 行)，它转移到 000040 单元，其中包含一条跳转(jump)

指令(0522 行)，它跳转到标号为“start”的指令，该指令在文件“m40.s”中(0612 行)。

6.2 Start(0612)

0613：测试存储管理状态寄存器的“启动”位 SR0。各该位设置，则处理机将一直执行两条指令的循环。在启动系统之前，当操作员触发在控制台上的“清除”(“clear”)按钮时，该寄存器通常被清除。

对于这一循环的必要性已提出了很多理由。最主要的一条是：在双总线超时错情况下，处理机将转移到＃0 单元，在这种情况下不应当允许程序向前执行。

0615：“reset”(“清除”)清除位，启动所有外部设备控制和状态寄存器。

现在，系统将运行在核心态方式，而存储管理部件则未启动。

0619：RISA0 和 KISD0 处于物理存储空间的高地址部分，它们是第一对核心态段寄存器的地址。前 6 个核心态说明寄存器被赋予的初值是 077406，它说明该段长度为 4K 字，存取控制是读/写。

前 6 个核心态地址寄存器被赋予的初值分别是 0、0200、0400、0600、01000 和 01200。

对前 6 个核心态段说明和地址寄存器赋初值的结果是：它们分别指向了物理存储器的前 6 个 4K 字段。于是，对于在 00137777 范围内的核心态地址，将它们翻译至物理地址是轻而易举的。

0632：“－end”是装入程序的一个伪变量，它定义程序代码和数据区的范围。此值被归整到下一个 64 字节的整倍数并存放在第 7 个段(段＃6)地址寄存器中。

注意：此寄存器的地址存放在“ka6”中，所以此寄存器的内容可以用“*ka6”存取。

0634：第 7 段说明寄存器装入的值表示：该段长度是 16*32=512，其存取控制字段是读/写。将 8 进制 017 左移 8 个位置，然后与 6 相“或”则得值 007406。

0641：第 8 段映照为物理地址空间的最高 4K 字段。

应当注意，在存储管理并未启动情况下，此种释译已经实施，亦即 32K 程序地址空间中最高 4K 字段的地址自动映照为物理地址空间中的最高 4K 字段。

我们可以注意到，从此点开始核心态段寄存器除一个附件都将保持不变，该附件是第 7 个核心态段地址寄存器。

UNIX 对该寄存器进行显式操作使其指向物理存储器中经常变动的一个位置。每一个这种位置都是一个 512 字长区间的起括点，该区间称之为“每个进段数据区”(perprocess data area,ppda)。

现在第 7 核心态地址寄存器被设置为指向一个段，该段将成为＃ 0 进程的 ppda。

0646：将栈指针设置为指向 ppda 的最高字。

0647：使 SR0 值从 0 增为 1，这样就方便地设置了“存储管理启动”位。

从此开始，所有程序地址都由存储管理硬件释译为物理地址。

0649：“bss”指的是程序数据区的第二部分，装入程序对该部分并不赋初值(参见 UPM 中的“A.OUT(V)”)。此部分的低、高地址分别由装入程序的伪变量“－edata”和“－end”定义。

0668：更改处理机状态字(PS)以指明“前状态”是“用户态”。这为检查和初始化非核心态地址空间外的物理存储器作好了准备。(这涉及到使用专用指令“mtpi”和“mfpi”(至/从前指令空间作移动)，以及对用户态段寄存器的处理。)

0669：调用“main”过程(1550)。

后面将观察到：“main”调用“sched”，而“sched”则不会终止。那么，“start”的最后 3 条指令(0670、0671 和 0672 行)是否需要，又有什么用处呢?这有些令人迷惑。对此问题的答案将在后面说明。读者可以先推敲一下“为什么”、“这些行究竟是做什么的?”。

6.3 main(1550)

在进入此过程时：

1. 处理机在优先级 0、核心态运行，而前状态为用户态。
2. 核心态段寄存器已被设置、存储管理单元已启动。
3. 操作系统使用的所有数据区已初始化。
4. 栈指针(SP 或 r6)指向一个字，其中包含返回至“start”的地址。

1559：因为“start”执行的初始化中已将“updlock”设置为 0，所以“main”的第一个动作似乎是多余的。

1560：“i”被赋初值为＃ 0 进程 ppda 区后第一个 32 字块块号。

1562：第一对用户态段寄存器用作在物理内存较高区域的“移动窗口”(“moving window”)。在该窗口的每一个位置，都试图读该窗口中的第 1 个可存取的字(用“fuibyte”)。如果这一读操作失败，则表示已到达物理存储器的尾端。否则将窗口中的 32 个字都赋初值为 0(用“clearseg(0676))，并将该块加至可用存储器列表中，然后将窗口前移 32 个字。

“fuibyte”和“clearseg”都位于“m40.s”中。在正常情况下，“fuibyte”将返回 0~223 之间的一个正值。但是，在异常情况下，亦即对该存储单元的访问没有得到响应时，“fuibyte”将返回－1。(产生这种结果的方式还是有点含糊不情，在第 10 章中将对此作解释。)

1582：“maxmem”规定了可供用户程序使用的主存最大总量。它是下列两个值中的较小值：

?物理上可用的存储器(“maxmem”)。

?一个装置可定义的参数(“MAXMEM”)(0135)。

最后的限制取决于 PDP11 系统结构。

1583：“swapmap”定义了在交换磁盘(swapping disk)上的可用空间，当用户程序被换出主存时，可使用该磁盘空间。初始化后这是一个连续的单一区域，其长度是“nswap”，起始相对地址是“swplo”。注意，“nswap”和“swplo”在“conf.c”(4697 行和 4698 行)中初始化。

1589：很快就会对此行及下 4 行的作用进行讨论。

1599：UNIX 设计假定系统中有一个时钟，它以线频(亦即，50Hz 或 60Hz)中断处理机。

有两种可供使用的时钟类型：一种是线频时钟(KW11-L)，其控制寄存器在单总线上的地址是 777546；另一种是可编程实时时钟(KW11-P)，其地址是 777540(1509 行、1510 行)。

UNIX 并不假定系统中存在哪一种时钟。首先它试图读线频时钟的状态字。若成功则初启该时钟，而另一个(若存在)则弃而不用。若不成功，则试探另一个时钟。若对两者的检测都没有成功，则调用“panic”，它等效于停止系统，并向操作员传送一条出错消息。

先将一对用户态段寄存器设置为适当值(1599，1600 行)，然后用“fuiword”引用时钟状态字，若产生总线超时错，则表示不存在该时钟。

1607：时钟用下列语句初始化

```
*lks=0115;
```

此操作的结果是：经 20ms 后该时钟将向处理机提出中断请求。该中断可在任何时刻发生，但是为了讨论的方便，可以假定在完成系统启动之前，不会发生中断。

1613：“cinit”(8234)初始化字符缓存池。参见第 23 章。

1614：“binit”(5055)初始化大缓存池(亦即块缓存池)。参见第 17 章。

1615：“iinit”(6922)为根设备(root device)初始化装配表的表目项。参见第 20 章。

6.4 进程

“进程”是一个已多次出现的术语。目前与我们目的相适应的进程定义可以是：正在执行的程序。

UNIX 中进程表示的细节将在下一章中讨论。现在我们想提请读者注意的是：每个进程都包含一取自“proc”数组的“proc”结构和一个“每个进程数据区”(“ppda”)，它包括一“u”结构的拷贝。

6.5 proc〔0〕的初始化

从 1589 行开始对“proc〔0〕”进行初始化。proc〔0〕中各元素在先前已被赋初值 0，现在只改变其中 4 个元素的值：

1)“p_stat”被设置为“SRUN”，这意味着＃ 0 进程已为运行准备就绪。

2)“p_flag”被设置为“SLOAD”和“SSYS”。前者意味着该进程在内存中(没有将它换出到磁盘上)，后者意味着决不应将此进程换出到磁盘上。

3)“p_size”被设置为“USIZE”。

4)“p_addr”被设置为核心态段地址寄存器＃ 6 的内容。

将会观察到＃ 0 进程已获得长度为“USIZE”(单位：块)的区(就是“ppda”的长度)，其起点紧跟操作系统数据区的结尾处(“－end”)。

该区第一块的序号数存入“p_addr”中以供后用。“start”(0611)中曾将该区清为 0，现包含称为“u”的“user”结构的一个副本。

在 1593 行，proc〔0〕”的地址存入“u、u_procp”，亦即“proc”和“u”结构相互关连。

1627：下一章将详细讨论“newproc”(1826)。

简而言之，“newproc”初始化第 2 个“proc”结构(“proc〔１〕”)，并在内存中分配第 2 个“ppda”。这里＃ 0 进程“ppda”的一个副本，其区别只有 1 项，第 2 个“ppda”中的“u.u-procp”的值是“&proc〔１〕”。

这里应当注意的是：1889 行中调用了“savu”(0725)，在构造第 2 个“ppda”副本之前，它将环境和栈指针的当前值保存在“u.u-rsav”中。

从 1918 行我们可以看到，“newproc”的返回值是 0，所以将不会执行 1628 行至 1635 行的语句。

1637：调用“sched”(1940)，可以先大致观察一下该过程，它包含一无限循环，所以决不会返回!

6.6 sched(1940)

在此阶段我们所关心的只是，当第一次进入“sched”时会发生些什么。

1958：“spl6”是一个汇编例程(1292)，它将处理机优先级设置为 6(请对照“m40.s”中的“spl0”、“spl4”、“spl5”和“spl7”)。

当处理机处于优先级 6 级时，只有优先级为 7 级的设备才能中断它。从此开始到调用“spl0”(1976 行)之间，优先级为 6 级的时钟就被禁止对处理机的中断。

1960：搜索“proc”数组，寻找其状态为 SRUN 并且未“装入”内存的进程。

＃ 0 和＃1 进程的状态为“SRUN”并且已“装入”内存，其余所有进程的状态为 0，它等效于“未定义”或“NULL”。

1966：该搜索失败(“n”仍为－1)。使“runout”标志值非负，表明没有进程，它们准备好运行并且“已换出”到磁盘上。

1968：调用“sleep”(等待此种事件发生)，调用时提供的优先权参数为“PSWP”(==100)，该优先权在此进程被唤醒后起作用，“PSWP”处在“非常紧急”的优先类中。

6.7 sleep(2066)

2070：“PS”是处理机状态字的地址。地址机状态存放在寄存器“S”中(0164，0175)。

2071：“rp”被设置为当前进程在“proc”数组中表目项的地址，也就是使“rp”指向当前进程的 proc 结构(此时，当前进程仍旧是“proc〔０〕”)。

2072：“pri”为负，所以转移至“else”部分，将当前进程(＃０)状态设置为“SSLEEP”。进入睡眠状态的原因以及唤醒后的优先权皆记入该进程的“proc”结构(“proc〔０〕)中。

2093：调用“swtch”。

6.8 swtch(2178)

2184：“p”是一个静态变量(2180)，这意味着其值被初始化为 0(1566)，而且在两次调用之间

其值保持不变。在第一次调用“swtch”时，将“p”设置为指向“proc〔0〕”。
2189：调用“savu”，它将当前进程的栈指针和环境指针保存到“u.u-rsav”中。
2193：调用“retu”：
1) 按传递过来的参数值设置核心态＃6 段地址寄存器(这造成当前进程的更改)。
2) 按更改过的当前进程设置栈和环境指针，该当前进程即将恢复执行。
此处对“savu”和“retu”的顺次调用的组合构成了“合作例程跳转(“coroutine jump”)(请对照 Cyber 系统的“交换跳转”(“exchauge jump”)，IBM/360 机上的“装入 PSW”或 B6700 上的“移动栈”(“Move Stack”)。
2201：搜索进程集以找到一进程，其状态是“SRUN”并且已装入内存，而且在所有这些进程中其“p_pri”值最小。(进程“p_pri”值小，则其优先权高。)
该搜索成功并且找到的是＃1 进程。(＃0 进程的状态刚从“SRUN”变为“SSLEEP”，所以它不再满足搜索寻找条件)。
2218：因为“p”非空，所以不进入空闲(idle)循环。
2228：“retu”(0740)造成合作例程跳转至＃1 进程，它成为当前进程。
＃1 进程是什么呢?它基本上是＃0 进程的复制品。
在调用“retu”之前并未调用“savu”，事实上在此之前已经保存了所需的信息。(在什么地方?)
2229：“sureg”是一例程(1738)，它将相应于当前进程的值复制到用户态寄存器中。这些值在早先已存入“u.u_uisa”和“u.u_uisd”数组。
最早一次“sureg”调用复制 0，它并无实际作用。
2240：“SSWAP”标志没有设置，所以现在可以忽略这一段令人迷惑的程序(2239)。
2247：最后，“swtch”返回值 1。但是该“return”究竟返回到何处呢?不是返回到“sleep”!(Not to “sleep”!)
在“return”之前用栈指针和环境指针进行一组调用寄存器值的设置。这些值恰好在返回之前)等于最近一次“savu(u.u_rsav)”执行时的值。
现在＃1 进程刚刚启动，它虽然从未执行过“savu”，但在复制＃0 进程之前(由“main”调用“newproc”实现)，相应值已存放到了“u.u_rsav”中。
于是，在这种情况下，从“swtch”就返回到“main”，返回值为 1。(对此请再查看一次，保证你对此已理解了!)

6.9 再回到 main

叙述到这里，其基本情况是：＃0 进程创造了一个它的复制品＃1 进程，＃0 进程则进入睡眠状态。结果是：＃1 进程成为当前进程，并返回到“main”，其返回值是 1。现在让我们继续阅读代码。
1627：现在执行“main”中的此条语句，它是一条 if 语句，其条件表达式是“newproc( )”。
1628：“expand”(2268)为＃1 进程找到一块新的较大区(从 USIZE*32 字扩大至(OSIZE+1)*32 字)，然后将原来的数据区复制到其中。
在此种情况下，原先的用户数据区仅由“ppda”组成，没有数据和栈区。现在释放原先的区。
1629：“estabur”用于设置“原型”段寄存器组的值，它们被存放在“u.u_uisa”和“u.u_uisd”，这些值以后由“sureg”使用。作为其最后的动作，“estabur”调用“sureg”。
“estabur”有 4 个参数，它们分别是正文、数据和栈区的长度，以及一个指示标志，用其说明正文和数据区是否应分别位于不同的地址空间。(在 PDP11/40 机上，这两个区并不分别存放在不同地址空间。)长度单位都是 32 字。
1630：“copyout”(1252)是一个汇编语言例程，它将核心空间中一指定长度的数值复制到用户空间区。在这里，“icode”数组复制到用户空间从地址 0 开始的一个区域中。
1653：此“return”并无特殊之处。它从“main”返回至“start”(0670)，其中最后 3 条指令的

效果是：在用户态执行用户态地址空间中位于＃0 地址的指令，也就是执行“icode”中第一条指令的副本。随后执行的指令也是“icode”中指令的副本。
到达这一点时，系统初始化全部完成。
1＃进程正在运行，在全部意义上它都是一个正常的进程。它的初始形式是下列“C”程序径编译、装入和执行后形成的：
其等效的汇编语言程序是：
如果系统调用“exec”失败(例如不能找到文件“/etc/init”)，则该进程进入一无限循环，除非发生时钟中断，否则处理机就停止不前。
“etc/init”所执行的功能在 UPM 的“INIT(Ⅷ)中说明。

第 7 章　进　　程

上一章跟踪了操作系统被引导后的启动过程，其中介绍了一些进程概念的重要特性。本章的目的是回过头去透彻地再揭示这些特性。
提出一种广为接受的“进程”定义是极其困难的。这与请哲学家回答“什么是生命?”这样的难题类似。如果我们不是钻在某些牛角尖里，那么面临的其他问题就比较容易解决。
前面已经给出了进程定义，“正在执行的一道程序”，这在相当范围内是一个不错的定义。但是，对于＃0 进程的整个生命期和＃1 进程的开始阶段，该定义却并不合适。系统中的所有其他进程则清清楚楚地与程序文件中的某一个或几个相关。
可以分两个层次对操作系统进程进行讨论。
在较高的层次上，“进程”是一个重要的组织概念，用其说明一个计算机系统作为一个整体的活动。将计算机系统视作若干进程的组合活动是适当的，每一个进程与一道特定的程序相结合，例如“sbell”或者“编辑程序”。Ritchie 和 Thompson 的论文“The UNIX Time-Sharing System”的后半部分就在此层次上对 UNIX 进行了讨论。
在这一层次上，进程本身被视作系统中的活动实体，而真正的活动部件本体；处理机和外部设备则被消隐，不引起人们的注意。进程诞生、生长，然后死亡；它们存在的数量在不断变化；它们可以获得并释放资源；它们可以交互作用、合作、冲突、共享资源等等。
在较低的层次上，进程是不活动的实体，它们依靠活动实体，例如处理机才起作用。借助于频繁地使处理机从一个进程映像的执行切换到另一个，就可以产生一种印象：每一个进程映像都连续发生变化，这就导致较高层次上的解释。
我们现在所关心的是较低层次上的解释：进程映像(process image)的结构、执行的细节以及在进程之间切换处理机的方法。
在 UNIX 环境中，关于进程可以观察到下列各点：
1) 在“proc”数组中的一个非空结构意味着一个进程的存在，非空“proc”结构指的是其元素“p_stat”非空。
2) 每个进程都有一个“每个进程数据区”(“ppda”)，其中包含“user”结构的一个副本。
3) 处理机在其生命期中不是执行此进程就是执行其他进程(除在两条指令之间的停顿外)。
4) 一个进程可创建或破坏另一个进程。
5) 一个进程可获得并占用各种资源。

7.1　进程映像

Ritchie 和 Thompson 在他们的论文中定义一个“进程”为一“映像”的执行，其中“映像”是-伪-计算机(pseudo-computer)的当前状态，亦即一个抽象数据结构，它存放在内存或磁盘上。
进程映像涉及 2 或 3 个物理上不同的存储区：
1)“proc”结构，它被包含在常驻内存的“proc”数组中，任一时刻对其都可存取。

2) 数据段，它由“每个进程数据区”、用户程序数据、(可能的)程序正文和栈组成。
3) 正文段，它并不总是存在，如果存在则由仅包含纯程序正文的段组成，亦即由可再入代码和常数数据组成。
很多程序没有单独的正文段。若有单独的正文段，那么很多执行同一程序的进程就可共享一个副本。

7.2 proc 结构(0358)

proc 结构常驻内存，它包含 15 个元素，其中 8 个是字符型，6 个是整型，1 个是整型指针。每个元素都提供任何时刻都能存取的信息，特别当进程映像的主要部分已被换出到磁盘上时：

ァ兒_stat”可取 7 个值中的 1 个，这 7 个值定义了 7 种互斥的状态。请参见 0381~0387 行。
ァ兒_flag”是 6 个 1 位标志的混合物，这 6 个标志位可单独设置。请参见 0391~0396 行。
ァ兒_addr”是数据段的地址：
■ 若该数据段在内存中，则这是一个块号。
■ 若该数据段已被换出至磁盘上，则这是一磁盘记录号。
ァ兒_size”是数据段的长度，单位是块。
ァ兒_pri”是当前的该进程优先级。经常对其重新计算，计算式是“p_nice”、“p_cpu”和“p_time”的函数。
ァ兒_pid”、“p_ppid”是数值，它们唯一的标识一个进程及其父进程。
ァ兒_sig”、“p_uid”、“p_ttyp”涉及与外部的通信，亦即来自进程正常域外的消息或“信号”。
ァ兒_wchan”为“睡眠”进程(“p_stat”或者为“SSLEEP”，或者为“SWAIT”)，标识其睡眠的原因。
ァ兒_textp”或者为 null，或者是指向“text”数组(4306)中一项的指针，该 text 数组项包含了与该正文段有关的重要统计信息。

7.3 user 结构(0413)

“user”结构的一个副本是每个“ppda”的重要组成部份。在任一时刻，只有一个“user”副本是可存取的。该副本的名字是“u”，总是位于核心态地址 0140000，亦即核心态地址空间第 7 页的起始地址。
“user”结构的很多元素在此处还不便于介绍，暂时也不是很有用。源代码第 4 页上的注释大致说明了“user”中每个元素的作用。
此刻，应当注意：
1)“u_rsav”、“u_qsav”和“u_ssav”皆为两个字的数组，被用来存放 r5、r6 的值。
2)“u_procp”指向在“proc”数组中相应的“proc”结构。
3)“u_uisa〔16〕”、u_uisd〔16〕”存放页地址和说明寄存器的原型。
4)“u_tsize”、“u_dsize”、“u_ssize”分别是正文段、数据段和栈段的长度，单位是 32 字块。
余下的元素与下列内容有关：
?保存浮点寄存器(并非针对 PDP11/40)。
?用户标识。
?输入/输出操作的参数。
?文件存取控制。
?系统调用参数。
?帐号信息。

7.4 每个进程数据区

“每个进程数据区”对应于该心地址空间第 7 页的有效部分(低地址部分)。其长度是 1024 字节。较低的 289 个字节由“user”结构的一个实例占用，余下的 367 字用作为核心态栈区(显

然，有多少进程就有多少核心态栈。)
当处理机处于核心态时，环境和栈指针 r5 和 r6 应指向下列地址范围：
0140441～01437777
超过上界将以段建例自陷，但是下界仅由软件设计的周密考虑而保证不会越出。(应注意的是：UNIX 没有使用硬件栈限选项。)

7.5 段

数据段被分配占用单一物理存储区，但是它包含 3 个不同的部分：
1) 一个“每个进程数据区”。
2) 用于用户程序的数据区。这可进一步划分成程序正文区、初始化数据区和不赋初值数据区。
3) 用于用户程序的栈。
其中，1)的长度总是“USIEE”块。2)和 3)的长度由“u.u_dsize”和“u.u_ssize”给出(单位：块)。在进程生存期中后两部分的长度可能更改。
对仅包含纯正文的单独正文段分配一单个物理存储区。该段的内部结构在此处并不重要。

7.6 映像的执行

第 7 核心态段地址寄存器的设备决定了当前正被执行的映像，因此也就决定了当前进程。若＃i 进程是当前进程，则该地址寄存器的值为“proc〔i〕.p_addr”。
经常希望将正在核心态执行的进程与正在用户态执行的同一进程两者区分开来。我们将使用术语“核心态进程＃i”和“用户态进程＃i”分别表示“正在核心态执行的进程＃i”和“正在用户态执行的进程＃i”。
如果我们选择使进程与特定执行栈相关连，而不是与“proc”数组中一项相关连，那么我们就是把核心态进程＃i 和用户态进程＃i 视为单独的进程，而不是单个进程＃i 的两个不同方面。

7.7 核心态执行

第 7 核心态段地址寄存器的内容必须随当前运行进程的改变而改变。而其他核心态段寄存器在系统启动后就不会再改变。正如前面已经指出的，前 6 个核心态页映照至前 6 个物理存储器页，同时，第 8 个核心态页映照至物理存储器的最高页。第 7 段的地址虽然经常变化，但其长度始终相同。
在核心态下运行时，用户态段寄存器的具体设置通常是并无关系的。但是在正常情况下它们总按用户进程正确设置。
环境和栈指针指向在第 7 页中的核心态栈区，该栈位于“user”结构之上。

7.8 用户态执行

每次在激活一用户进程时，在其前、后总是伴随着相应核心态进程的一次激活。与此相对应，无论何时，只要一进程映像正在用户态执行，那么与其相对应的用户态和核心态寄存器就会被正常设置。
环境和栈指针指向用户态栈区。该栈从第 8 用户页的高地址处开始，但可向下扩充，例如占用整个第 8 页，第 7 页的一部分或全部等等。
核心态段寄存器的设置相当简单，而用户态段寄存器的设置则相当繁锁。

7.9 一个实例

考虑在 PDP11/40 上运行的一道程序，其正文部分为 1.7 页，数据部分为 3.3 页，栈区为 0.7 页。(在本例中为方便起见使用了小数，与实际情况稍有差别。)在虚地址空间中的安排示于图 7-1 中：
在虚地址空间中必须为正文段分配 2 整页，而物理存储区则只需 1.7 页(见图 7-2)。
数据和栈区分别要求使用虚地址空间中的 4 页和 1 页，而在物理地址空间中则分别要求使用

3.3 页和 0.7 页。
整个数据段要求使用四又八分之一页物理存储器。额外的八分之一页用于“每个进程数据区”，它对应于第 7 核心态地址页(见图 7-3)。
注意数据段各部分的顺序，在各部分之间没有未使用区。
需设置用户态段寄存器以反映下表中的值，其中，“t”和“d”分别表示正文和数据段的起始地址(单位：块)：

页	地址	长度	注释
1	t+0	1.0	只读
2	t+128	0.7	只读
3	d+16	1.0	
4	d+144	1.0	
5	d+272	1.0	
6	d+400	0.3	
7	?	0.0	未用
8	d+400	0.7	向下扩展

注意第 8 个地址寄存器的设置。将“t”和“d”设置为 0，则获得存放在数组“u.u_uisa”中的地址原型。

7.10 设置段寄存器

用户态段寄存器的原型值是由“estabur”设置的，当一通程序第一次执行，以及存储分配发生重要变化时都要调用该过程。此原型存放在数组“u.u_uisa”和“u.u_uisd”中。
无论何时只要将再激活＃i 进程，则调用“sureg”，它将原型数据按需进行适当处理后送入相应寄存器。说明寄存器只要直接复制，但地址寄存器值则必须作调整以反映物理存储区中的实际使用位置。

7.11 estabur(1650)

1654：对一致性作各种检查以保证所要求的正文、数据和线的长度都是合理的。
注意“sep”的非 0 值意味着正文区(“i”空间)和数据区(“d”空间)的分别映照。PDP11/40 并不支持此种功能。
1664：“a”定义了相对于基地址为 0 的段地址。“ap”和“dp”分别指向原型段地址和说明寄存器组。
这些设置的前 8 个力图针对“i”空间，后 8 个则针对“d”空间。
1667：“nt”测量正文段所需的 32 字块数。若 nt 为非 0，则需为正文段分配 1 页或多页。
若分配多于 1 页，则除最后 1 页外，其他各页长度皆为 128 块(4096 字)，而且是只读，它们的相对地址从 0 开始，各页顺序增加 128。
1672：若正文段尚余留部分小于 1 页，则为其分配下一页的所需部分，并设置相应段地址和说明寄存器。
1677：如分开使用“i”和“d”空间，则将余下“i”页的段寄存器标记为 null。
1682：因为所有余下的地址引用数据区(非正文区)并且相对于此区的起点，所以使“a”复位，而此区开始的“USIZE”块保留用于“每个进程数据区”。
1703：从地址空间顶向低地址(向下)分配栈区。
1711：若需为栈区分配页的一部分，则分配该页的高地址部分。(对于向上扩展的正文和数据区，则分配页的低地址部分。)这要求在说明寄存器中设置一额外位，即“ED”位(向下扩展”。)
1714：若不使用分开的“i”和“d”空间，那么到此点为止，只对 16 个原型寄存器对中的前 8 个赋予初值。在这种情况下，后 8 个从前 8 个中复制。

7.12　sureg(1739)

“estabur”(1724)、“swtch”(2229)和“expand”(2295)调用此例程以便将原型段寄存器值复制到实际的硬件段寄存器。

1743：从“proc”结构的适宜元素取得数据区的基地址。

1744：各原型地址寄存器(对于 PDP11/40，有 8 个原型地址寄存器)的值加上“a”，然后存放到相应硬件段地址寄存器。

1752：测试是否已分配分开的正文区，若是则将“a”设置为正文区对数据区的相对地址。(注意，此值可能是负值!地址的单位是 32 字块)

1754：此段代码的样式类似于本例程的开始部分，例外是……

1762：这是一段其作用不易看清楚的代码，它对非可写，亦即正文段的地址寄存器的值进行调整。

“estabur”和“sureg”代码带有经多次修改的痕迹，它们不像我们所希望的那样精致巧妙。

7.13　newproc(1826)

现在到时候了，让我们仔细观察一下创建新进程的过程，创建的新进程几乎是它们创建者的精确复制器。

1841：“mpid”是一个整型值，它在 0 至 32767 范围内逐一递增。创建一进程时，要在步进过程中找到一个“mpid”的新值，从而为该新进程提供了一个区别于所有其他现存进程的整型数。在寻找“mpid”新值时可能进入循环，于是步进得到的值或许会与现存正在使用的值重复，因此要不断进行测试。

1846：在“proc”数组中搜索以找到一个空“proc”结构(若 p_stat 的值为 null，则该“proc”结构为空)。

1860：到达此点时，新找到“proc”项的地址存放在“p”和“rpp”中，而当前进程“proc”项的地址则存放在“up”和“rip”中。

1861：将新进程的属性存放到它的“proc”结构中。其中很多是从当前进程处复制的。

1876：新进程继承其父进程的打开文件。对于每一个打开文件增加其引用计数。

1879：如果正文段是共享纯正文段，也就是与数据段分开，则增加相关的引用计数。注意，在此处暂时借用了“rip”和“rpp”。

1883：增加父进程的当前目录的引用计数。

1889：将环境和栈指针的当前值保存到“u.u_rsav”中。“savu”是定义在 0725 行处的汇编语言例程。

1890：恢复“rip”和“rpp”的值。暂时更改“u.u_procp”的值，使其从对应于当前进程的值更改为对应于新进程的值。

1896：试着在内存中找到一个区域，以便在此区域中创建新数据段。

1902：如果在内存中没有一个合适的区域，则应当在磁盘上构造新副本。由于在 1891 行引入的不一致性，

应当仔细地分析下一段代码。

u.u_procp－＞paddr!=*ka6

1903：将当前进程标记为“SIDL”，以阻止“sched”将其映像换出到磁盘上。

1904：设置

rpp－＞p_addr=*ka6;

使新进程“proc”项协调一致。

1905：将环境和栈指针的当前值保存至“u.u_ssav”。

1906：调用“xswap”(4368)，将数据段复制到磁盘交换区。因为第二个参数为 0，所以不释放该数据段占用的主存区。

1907：将新进程标记为“已换出”。
1908：使当前进程恢复到正常状态。
1913：在内存中有足够空间，所以保存新“proc”项的地址，并一次一块地复制数据段。
1917：将当前进程的“每个进程数据区”恢复至它的以前状态。
1918：返回，返回值为 0。
显然，“newproc”本身还不是以产生有益并变化的进程集。将在第 12 章中讨论的“exec”过程提供所需的附加能力：使一个进程改变自己的角色从而获得新生。

第 8 章　进 程 管 理

进程管理关系到在多个进程之间共享处理机和主存，这些进程可被视为是这些资源的竞争者。
在初启使用资源的进程时，或者由于其他原因，经常要作出重新分配资源的决定。

8.1　进程切换

一个进程调用 swtch(2178)(swtch 又调用“retu”(0740))可挂起自身，也就是释放处理机。
例如若一个进程已到达它不能超过的某个点，这时它就要调用“sleep”(2066)，而 sleep 则调用“swtch”。
另外，一个在核心态下运行的进程，当它将要转入用户态之前，会测试变量“runrun”，如果其值非 0，则意味着更高优先权的进程已为运行准备就绪。此时该核心态进程也将调用“swtch”。
“swtch”在“proc”表中搜索其“p_stat”等于“SRON”，其“p_flag”中设置了“SLOAD”位的各进程，并从中选择其“p_pri”最小的进程，然后将控制转移给被选中的进程。
对于每一个进程的“p_pri”值，系统要经常重新进行计算，为此目的使用的过程是“setpri”(2156)。显然，“setpri”使用的算法具有重要影响。
调用“sleep”挂起自身的进程可由另一个进程唤醒而返回到准备运行状态。在进行中断处理时经常发生这种情况，此时处理中断的进程直接调用“setrun”(2134)或先调用“wakeup”(2113)，然后由“wakeup”调用“setrun”。

8.2　中断

应当注意一个硬件中断(请参见第 9 章)并不会直接造成对“swtch”或其等效例程的调用。一个硬件中断使一个用户进程转变为一核心进程，正如上面刚刚提到的，在它处理完中断后可能不返回到用户态而是调用“swtch”。
若一核心态进程被中断，那么在中断处理结束后，该核心态进程一定回到它被中断的点继续运行。这一点对理解 UNIX 如何避免很多与“临界区”(“critical sections”)相关的陷井是很重要的。本章最后部分将对临界区进行讨论。

8.3　程序交换

一般而言，内存容量并不足以让所有进程的映像同时驻留其中，于是某些数据段就是被“换出”(“swapped out”)，写到磁盘上一个被称之为磁盘交换区(disk swapping area)的区域中。
存放在磁盘期间，进程映像相对而言是不可存取的，而且确实是不可执行的。因此，驻在内存中的进程映像集应当定期更改，其方法是将内存中的某些进程映像换出，将存放在磁盘交换区中的某些进程映像换进。
关于进程映像换进换出的很多决策都是由过程“sched”(1940)作出的，在第 14 章将对此过程进行详细分析。
“sched”是由＃0 进程执行的。＃0 进程完成其初始任务后起两种作用。一种作用是调度进程(“scheduler”)，也就是一个正常的核心态进程；另一种是在不知不觉之中担任“swtch”的中间进程(已在第 7 章中讨论了“swtch”)。因为过程“sched”决不终止，所以核心态＃0 进程也

决不会完成其任务，随之而来的结果是：不会有在用户态下运行的＃0 进程。

8.4　作业

在 UNIX 中没有作业的概念，至少是没有传统批处理系统中作业的概念。

任一进程在任一时刻都可创建(“fork”)它自身的一个副本，而且基本上没有多少延迟，因而也就创建了一个新作业的等价物。UNIX 中没有作业调度、作业类之类的活动和概念。

8.5　汇编语言过程

下面 3 个过程是用汇编语言编写的，而且在处理机优先级设置为 7 级时运行。这些过程并不遵循一般的过程入口约定，在进入和退出这些过程时，不影响 r5 和 r6(环境和栈指针)。

正如在上一章中已提及的，“savu”和“retu”两者组合起来起合作例程跳转的作用。第 3 个例程是“aretu”，若其后跟随一条“return”语句，则起非本地“goto”的作用。

8.6　savu(0725)

此过程由“newproc”(1889，1905)、“swtch”(2189，2281)、“expand”(2284)、“trap1”(2846)和“xswap”(4476，4477)调用。

将 r5 和 r6 的值存放到一个数组中，该数组的地址作为一个参数传递至“savu”。

8.7　retu(0740)

此过程由“swtch”(2193，2228)和“expand”(2294)调用。

它复位第 7 个核心态段地址寄存器，然后从“u.u_rsav”最新可存取副本中复位 r6 和 r5(注意，u.u_rsav 位于“u”的开始处)。

8.8　aretu(0734)

此过程由“sleep”(2106)和“swtch”(2242)调用。它从作为参数传递过来的地址处重装 r6 和 r5。

8.9　swtch(2178)

“swtch”由“trap”(0770，0791)、“sleep”(2084，2093)、“expand”(2287)、“exit”(3256)、“stop”(4027)和“xalloc”(4480)调用。

“swtch”是一个非常特殊的过程，它分三段执行，一般而言，涉及 3 个不同的核心态进程。其中，第 1 第第 3 个进程分别被称为“退休”(“retiring”)进程和“上升”(“arising”)进程。第 2 个进程总是＃ 0 进程，它也可以是“退休”或“上升”进程。

注意，“swtch”使用的变量都是寄存器型、全局或静态(全局存放)变量。

2184：静态“proc”结构指针“p”规定了搜索“proc”数组的起点，搜索的目的是找到下一个要激活的进程。将“p”定义为静态变量就可以在两次调用“swtch”之间保持其值，也就保持了搜索的偏置值。若“p”值为 null，则将其值设置为“proc”数组的起始地址。这只在第 1 次调用“swtch”时才发生此种情况。

2189：调用“savu”(0725)保存环境和栈指针。

2193：“retu”(0740)复位 r5 和 r6，而最重要的是将核心态地址寄存器＃ 6 设置为调度进程的数据段地址。

2195：第 2 段从此开始：

从此行开始到 2224 行的代码只由核心态进程＃ 0 执行。其中包含了两层嵌套循环，除非找到一可运行进程，否则不会从其中退出。

在系统没有准备就绪且映像在内存的进程时，处理机消耗其大部分时间执行 2220 行。只有发生中断时(例如时钟中断)才会终止执行此行代码。

2196：清“runrun”标志，该标志指示一个较当前进程具有更高优先权的进程已为运行准备就绪。“swtch”将寻找最高优先权进程。

2224：将“上升”进程的优先数存放在全局变量“curpri”中，以便将来引用和比较。

2228：对“retu”的另一次调用，它将 r5、r6 和第 7 个核心态地址寄存器设置为与“上升”进程相适应的值。

2229：第 3 段从此开始：
“sureg”(1739)用为“上升”进程存储的原型值复位用户态硬件段寄存器组。
2230：从此行开始的注释并不令人鼓午。在本章将结束处，我们还会返回到此处。
2247：经仔细检查后，你会发现没有一个调用“swtch”的过程直接检查此行的返回值。
只有调用“nmwproc”的过程才对此值有兴趣。所创建子进程的第一次激活与此值有关。
8.10 setpri(2156)
2161：按下面的公式计算进程优先数：
优先数=min{127,(使用的时间+PUSER+P_nice)}
其中：
1) 使用的时间=进程最近一次换入以来累计使用的 CPU 时间(滴答)除以 16，亦即使用时间的测量单位是约 1/3 秒。(当讨论时钟中断时，还将对此进行讨论。)
2) PUSER==100。
3)“p_nice”是计算进程优先数的偏置值。p_nice 值通常为正，因此减少了进程的有效优先权。注意，UNIX 中的一条易于使人迷惑的惯例是：优先数(p_pri)愈小，优先权愈高。于是优先数－10 的优先权高于优先数 100 的优先仅。
2156：如果刚计算优先数进程的优先权低于当前进程，则设置再调度标志。
2165 行测试的含意令人惊讶，特别是与 2141 行进行比较时。我们将此留给读者思考，以找到为什么这不是一个错误的理由。
8.11 sleep(2066)
当一个核心态进程要将自己挂起时调用此过程。(在代码中大约有 30 处调用此过程。)此过程有两个参数：
?要进入睡眠状态的原因。
?一优先数，该进程以后被唤醒后起作用。
若此优先数为负，则一个信号的到达不会使该进程不进入睡眠状态。第 13 章将讨论“信号”。
2070：保存当前处理机状态，包括处理机优先级和前状态信息。
2072：若优先数非负，则测试是否接收到信号。
2075：从此开始一个小的临界区，其中改变进程状态，调用此过程的两个参数则存放在通常可存取的单元中(在“proc”结构中)。
因为这几个信息字段可被“wakeup”(2113)查询和更改，所以这一代码段是临界区，“wakeup”经常由中断处理程序调用。
2080：若“runin”标志非 0，则调度进程(＃ 0 进程)正在睡眠，以等待将另一进程换进内存。
2084：对“swtch”的调用提供了一个未知长度的延迟，(亦即从此次放弃处理机到下一次被再次切换占用处理机之间的时间长度是不确定的。)在此期间可能发生相关的外部事件。因此对“issig”(2085)的第二次测试是合适的。
2087：对于负优先数“睡眠”，进程典型地等待系统表空间的释放，为了加快此种活动的进展速度，不允许一“信号”的发生。
8.12 wakeup(2133)
此过程与“sleep”互补。它简单地搜索进程集，寻找由于指定的原因(作为参数“chan”给出)而睡眠的所有进程，并调用“setrun”分别激活它们。
8.13 setrun(2134)
2140：将进程状态设置为“SRUN”。“swtch”和“sched”将把此进程视为可再次执行的候选者。
2141：若该进程的优先数低于当前进程，亦即它比当前进程更重要，则设置重新调度标志“runrun”，以便以后引用。

2143：若“sched”正睡眠，等待一个进程“换入”，而且刚被唤醒的进程位于磁盘上，则唤醒“sched”。
因为“sched”是唯一以“chan”为“&runout”调用“sleep”的过程，2145 行可以代换成下列形式的递归调用：
setrun(&proc〔0〕);
或者代换成更好一些的下列代码：
rp=&proc〔0〕;
goto sr;
其中，sr 是要插在 2139 行开始处的标号。

8.14 expand(2268)

在本过程开始处(2251)的注释对此过程作了几乎全部必要的说明，唯一需要补充之处与内存不足时要进行的换出操作有关。
注意，“expand”并不特别关注用户数据区或栈区的内容。
2277：若扩充实际上是收缩，则从高地址端释放多余的存储区。
2281：调用“savu”将 r5 和 r6 的值存入“u.u_rsav”。
2283：如无足够内存可供使用…
2284：环境指针和栈指针再次存放至“u.u_ssav”中。但是，应当注意：因为没有进入新过程，栈也并未增长，所以存放到“u.u_ssav”中的值与 2281 行的相同。
2285：“xswap”(4368)将其第 1 个参数所指定进程的内存映像复制到磁盘上。
因为第 2 个参数值非 0，所以将该进程数据段占用的内存区释放回可用空间列表中(即“coremap”中)。
但是，在“swtch”中再次调用“retu”之前，仍继续使用内存中的同一区。
注意：在已经复制了磁盘映像后，“swtch”中在 2189 行所调用的“savu”将新值存入“u.u_rsav”(因为内存映像已被正式放弃，所以这并无实际用处)。
2286：在该进程“proc”结构的“p_flag”中增加设置“SSWAP”标志。(“proc”结构不会被换出，因此“SSWAP”标志随时可以检查。)
2287：调用“swtch”，于是仍在其原占用区运行的进程将自身挂起。因为调用“sxwap”使“SLOAD”标志被清除，所以“swtch”不会立即再激活此进程。
只有该进程在磁盘上的映像再次被复制到内存中后，才可能再次激活它。“swtch”执行的“return”，将控制转移调用“expand”的过程。

8.15 再回到 swtch

当一个进程再次被激活，也就是变成了“swtch”中的“上升”进程时将发生什么呢?
2228：从“u.u_rsav”中恢复栈和环境指针(注意，指向“u”的指针也是指向“u.u_rsav”的指针(0415))，但是……
2240：若内存映像已被换出(例如被“expand”换出)……
2242：栈和环境指针的值已经不再能继续使用，从“u.u_ssav”中复位它们的值。
问题是“如果在 2284 行存入“u.u_ssav”中的值与 2281 行存入“u.u_rsav”的值相同，那么它们又怎样会变成不同了的呢?
对此问题的回答需要细致的分析、推敲，也有相当难度，因此才会有 2238 行的注释“并不期望你能理解这一类。”请先将“xswap”分析透彻，而最后的答案将可在第 15 章中找到。现在你可能急于弄清楚这一问题，那么请立即加入“2238”俱乐部。

8.16 临界区

若 2 个或多个进程对同一数据集进行操作，那么这些进程的组合输出将依赖于它们的相对同步(依赖于它们对同一数据集操作的前后顺序)。这通常是非常不希望发生的，应想方设法加

以避免。解决方法一般是在由每个进程执行的代码中定义“临界区”(组织临界区是程序员的责任)。程序员应当保证在任一时刻不会有几个进程同时执行存取特定数据集的代码段。
在 UNIX 中用户进程不共享数据，所以并不会产生这种形式的冲突。但是核心态进程会存取多种系统数据，于是会产生冲突。
在 UNIX 中，中断并不直接造成进程的改变。只有核心态进程显式调用“sleep”，才可能将它们自己挂起在一个临界区中，此时确实需要引进一显式的锁变量。甚至在这种情况下，对锁变量的测试和设置操作也无须作为一个原子操作执行。
某些临界区代码是由中断处理程序执行的。为了保护其输出会受到某些中断处理影响的代码段，在进入临界区之前将处理机优先级暂时提升到足够高，这样就可以推迟中断，当该代码段退出临界区后才降低处理机优先级。当然，有一些中断处理程序应当遵循的约定，我们将在第 9 章中对此进行讨论。
顺便要提及的是 UNIX 所采用的策略只对单处理机系统有效，对多处理机系统则是完全不适用的。

罗　斯　跋

本书首次出现是以行式打印机列表的形式，那是 1976 年，新南威尔士大学将这种打印材料分发给选修“6.602A—计算机系统 I”课程的学生，每次印发一章。这门课程由约翰·莱昂副教授讲授，而我就是学生中的一员。正好是我写这篇评价文章的 20 年之前，莱昂一方面忙于编写这本《UNIX 操作系统源代码分析》，重新编排源代码，一方面讲授这门课程。
在 1974 年底到 1975 年初这段时间，新南威尔士大学的计算机中心处于从 IBM360/50 系统转移到 CDC Cyber/72 系统的过程中。Cyber 计算机系统允许使用远程批处理终端提交作业，而在小范围使用卡片阅读机和行式打印机。学生们使用计算机不必再到计算中心场地。学校购置了多种 DEC PDP-11 计算机来实现这种远程访问功能，而没有采用昂贵的 CDC 计算机。在电子工程学院安装了一台更大的 PDP-11/40，这台机器拥有 2.5MB 容量的磁盘驱动器，其内存(后来)达到 104KB。
1974 年 7 月《计算机协会通信》发表了里奇和汤姆森的一篇论文，论述“UNIX 分时系统”。由于该系统是在几台 PDP-11 计算机上运行，所以实际上是可以自由使用的。由一位讲师编写使用 UNIX 分时系统的讲义。在同一时期，尼克劳斯·韦思发明的 Pascal 成为流行的编程语言，并且可以在 CDC Cyber 系统上使用。具有讽刺意味的是，使用 Pascal 的讲义是由莱昂写成的，而 Pascal 的维护和授课却是由肯·罗宾逊承担。按照学校的意见，肯的工作由莱昂接替，而肯就是编写使用 UNIX 的讲义的人。
PDP-11 在几年内成为在澳大利亚发展 UNIX 的焦点，因为莱昂与一批教员和学生力图开发利用这台计算机，把它用于教学工作和提交远程作业。围绕应该在 PDP-11 上使用什么操作系统的争执，竟然引起一次停课事件和计算机中心主任的辞职。这件事却提高了计算机科学系在新南威尔士大学组织层次中的地位。
UNIX 对新南威尔士大学有着重要意义，正像 UNIX 对世界上其他大学的意义一样。那时，学校的教师、研究人员和学生都能学习编译原理、数据库，等等，然而对于操作系统却几乎不能做任何需要“动手”的工作。使用实时操作系统的实时计算机被锁在机房中，用于承接一天 24 小时的处理作业。UNIX 改变了这种局面。UNIX 系统的全部文档，加上约翰编写的源代码和分析，“能够轻而易举地放进每个学生的书包”。几年以后，约翰巧妙地将这种变化称为“固入 4BSD”。
对于学生的学习和实验而言，使用 UNIX 系统的好处是可达到性和可检验性。世界上由研究机构增进 UNIX 发展的情况出现变化，这一变化大大促进了 UNIX 的发展。《莱昂氏 UNIX 源代码分析》一书使得用 UNIX 作试验变得非常容易，对于 UNIX 在 70 年代后期和 80 年代

初期的成功发展作出了巨大贡献。
有一个事实可以用来衡量《莱昂氏 UNIX 源代码分析》一书取得了何等成功，这就是它们的大量非法影印本的四处流行。我在 1987 年访问过一家小型计算机公司，居然在一个书架上发现了《莱昂氏 UNIX 源代码分析》的影印本，从其扉页上的答案推断，它至少是从经过四次复制的影印件上拷贝下来的。像这样的作法，纵然是近乎非法的，但依然通行无阻。
约翰·莱昂以其特有的方式写作《莱昂氏 UNIX 源代码分析》，并且重新组织操作系统这门课程的教学，其中包含着多种理由。在原书的绪论中，对多数理由已作了说明。但是有一条理由，在出版物中一向很少被人提及，这或许并非一条不太重要的理由。这就是那时构成计算机科学教学计划的所有课程中，除硬件课程之外，都教学生如何编写程序和调试程序。但是没有一门课程要求学生学会“阅读”程序。其他课程牵涉的都是小程序，即学生能够管理的程序，而 UNIX 核心系统显然是一个非常大的程序。对于这种情况，约翰以不无讽刺的口吻说：“他们看到仅有的大程序是那些由他们自己编写的程序；而这一个至少是写得很好的程序。”在以后的几年中，教学生们去阅读和考察代码变成人们接受的事实。
现在回头去看这本书，我发现提出这些目标现在仍很合适。UNIX 版本 6 核心系统的结构完美，程序简约。直到今天，尚未发现其他程序能够展示如此高超的软件工程技艺。UNIX 处处显现出设计者的见识和独具匠心，简明扼要而易于理解，兼收并蓄各种折衷和无需优化的低效代码。莱昂取得的部分成就，就是认识到 UNIX 的这种特质，并把它们融入自己的书中。
应约为约翰的《莱昂氏 UNIX 源代码分析》的新版写这篇后记，感到非常愉快和荣幸。这项任务本应由约翰本人来承担。但不幸的是约翰的健康状况欠佳，不允许他来做这件事情。我希望在这里能够表达出他的一些感受。作为了解他的人和他的学生，我深切地感谢这 22 年来他所给予我的友谊和教诲。
格里格·罗斯
于悉尼，1996 年 2 月
(罗斯曾就读于新南威尔士大学，毕业后创立了两家软件公司，其中一家经营至今。他在 IBM 的沃森研究中心作过一年访问学者，其后在澳大利亚计算与通信研究所和斯特林软件公司任职，并加盟高通公司。格里格是澳大利亚 UNIX 系统用户组的前总裁和 USENIX 的董事。)

奥 德 尔 跋
我首次拜读《莱昂氏 UNIX 源代码分析》(原书分为两册)时的情景，至今仍然历历在目。我记得走进俄克拉荷马州立大学计算机科学系办公室，查看我的邮箱。我翻出了投进邮箱的最新一期《UNIX 新闻》。我立即停止搜索收到的其余邮件而专注于这本杂志，伫立在一排文件柜的前面，靠在文件柜的台面上阅读，在杂志第一页的中央刊有一篇文章，宣称澳大利亚有人写了一本《莱昂氏 UNIX 源代码分析》的书。甚至提到更好的消息，说他们正在发售这本书，而且以一个研究生可以买得起的价格收费。余下的最大问题是要找到一张汇率表，把澳大利亚的定价换算成美元(在这种事情上支付本国货币的做法是天真的)，之后，我随意地寄出一张支票，这件事几乎就被遗忘了。
大约在一个月之后，从新南威尔士大学寄来了一个包裹，系秘书替我保存起来。令她惊奇的是这个来自万里之外的包裹却带有精致的包装。最初，我同样迷惑不解，但打开包裹后，露出两本薄薄的装饰精美的书：一本《UNIX 操作系统分析》，红色封面，另一本《UNIX 操作系统源代码版本 6》，桔黄色封面，两本书都出于同一作者约翰·莱昂。
过了几天以后，我在计算机科学系办公室再度露面。我的教授、计算机科学系主任怀疑我到哪里去了，所以注意我，因为平常我出入系办公室同留在宿舍的时间至少是同样多。我手中拿着那两本书，笑着对他说：“我在读这些书，”“现在我知道了很多它的原理!”自然这是指 UNIX 版本 6。那是 1977 年，差不多是在 20 年前。

不久之前我查看了原来的哪些拷贝，并惊奇于有那么多的事情发生了变化，但并非诸事如意。C 语言曾经是一种非常优美的编程语言，它以一种雅致、奇妙的形态呈现在页面上。后来由于不良造型陈规和周全备至的类型定义，给语言带来损害。当今的 C 语言，由于追求它的全部功能改善和可移植性，似乎已经变成了许许多多的点和细长的边，其中旧的语言已成平缓的曲线，围绕着使用一种奇特结构的那些概念塑造着自己。

系统的其余部分亦属如此。诚然，我欣赏现代的 tty 处理程序，它能够从最简陋的终端上抽取出人类工程学上的有用行为，我也喜欢联网子系统，它能将我的击键声通过 ISDN 线路传递到我正用来写这篇文章的计算机上，而其性能比我从一台本地的 9600 波特的终端上获得的更好。但是，最重要的是我确实欣赏文件系统，它在受到无端的停机侵害面前，不会报以自行销毁。

但是书中存在的那个系统，完整地摆在那里。不错，今天的车座垫更加舒适，变速箱换上了同步离合器，然而系统的灵魂依旧，而约翰·莱昂所作的绝妙注释正好揭示了当初由肯和丹尼斯所注入的内在美。他的注释是简洁而深刻的，映照出代码的原形，解开了我的脑海中疑惑好奇之处。

几年以后，我应邀出席在天津举行的英国 UNIX 用户组会议(一次真正的会议)。在途中逗留在一位朋友家里，他在会前的周末约请了几位与会的客人。到达伦敦后，立即赶往 Farnborough，我步入房舍时受到其他客人的迎接。然后我被介绍给约翰·莱昂。我一时有些茫然，仅能说出一句话："您是约翰·莱昂?"他羞怯地微笑着回答："您或许是指那些书吧?不错，我就是。"此时此刻，我能想到要做的一切就是对他表示感谢，是他推动我走上一条职业的和冒险的道路，这是比我所能想像的更为复杂的人生历程。那个周末的其余时间全部花在游览伦敦的建筑物上，跟随约翰进行"库克船长式的巡游"。约翰曾就读于剑桥大学，并多次造访过伦敦。这次旧地重游，访问了多处他昔日经常光顾的地方。他那儒雅的教授品格一直是奇特的。

两年前，我怀着极其愉快的心情，参加了为约翰颁发 USENIX 终身成就奖的活动，这个奖项是为表彰他推动了一代 UNIX 高手走上职业化的道路，他们中的一些人在不同的公司份演着重要角色，而那些公司正是凭借那本桔黄色小书中的代码取得了商业上和智慧上的成功。那本小红书中的表述言词，则以其深刻的见解帮助人们从根本上形成自己的思维方式。

敬爱的约翰，每一个学生都会有良师益友，帮助从根本上塑造他们的思想和确定他们的人生道路，我本人虽然没有坐在教室里聆听过您一小时讲课，却要请您理解我的深切感谢，感谢您的著作对我不可磨灭的影响。

迈克·奥德尔

(奥德尔曾经从俄克拉荷马走到劳伦斯伯克利国家实验室，在那里他同德比·希勒在一个办公室。在历经从伯克利到华盛顿特区、再到科罗拉多州斯普林斯的漫游之后，他最终返回特区。他曾是《计算系统》杂志的首任主编和 USENIX 协会的副总裁。现任 UUNET 技术公司副总裁和首席科学家。)

古德哈特跋

我第一次见到莱昂先生是在 1989 年，那时我刚从英国移居澳大利亚。自从那次相见之后，我就从他的友情中感到巨大的快慰。至于要说同他打交道，大约要追溯到 70 年代后期。当时一位朋友将莱昂的两本小册子的最初影印本交给我，至今我还保存着它们。从某种意义上说，这些卷成滚筒状的使用过无数次和已经褪色的影印本，比起约翰新近送给我的原始小册子的复制品更重要，更具感情上的特征。如同那个年代学习计算机科学的许多学生一样，约翰的小册子成为我所保存的 UNIX 文献中不可或缺的一部分，而该书本身不过 3/4 英寸那么厚，是依据从 Western Electric 公司的许可证条款复制的文档。

莱昂副教授在新南威尔士大学执教 23 年之后于 1995 年 7 月退休。他早些时候是在电子工程系教书，后来才到计算机科学系，在那个时期，他对这两个系作出了巨大的贡献。在澳大利亚学术界，约翰在专业领域发挥着重要的作用。他担任《澳大利亚计算机杂志》主编多年，曾负责建立澳大利亚的第一个 UNIX 系统。他于 1975 年创建了澳大利亚 UNIX 用户组(AUUG)，而在 1976 年编著了本书。

了解那个时期约翰所开展的工作的意义是很重要的：对于 70 年代学习计算机科学的学生来说，对于操作系统中若干复杂的问题是不在学习范围之内的，如像进程调度、系统安全性、进程同步、文件系统，以及一些其他的高级概念，同时也很难组织教学。可供作为学生进行案例分析的任何东西，并非都能得到。那时计算机科学系的学生，是通过穿孔卡片穿孔和收集打印输出的折叠纸获得训练的，如此等等。

在那个年代，人们大体上是把计算机操作系统看成是一大堆庞杂而难于理解的专用代码。当然，UNIX 系统的出现改变了这种看法，即使如此，直到莱昂的著作问世之前，学习计算机科学的学生们并不能从校园的 UNIX 源代码许可证中获得实际好处。学校所用的操作系统充其量可以作为案例分析使用。不幸的是，小册子的发布受到限制，并且从未获得正式出版，但在世界各地，学习操作系统的学生中不使用其复印本的人是很少的。

我们今天来看约翰的著作，其重要意义与其说在于其实用方面，倒不如说在于追忆过去—当然，除非你并不想了解它。现今的 UNIX 系统同旧时的系统已经有了根本性的差别，系统的形态和规模看起来大得多，复杂程度远远超过了版本 6 的精巧的核心系统，也就是约翰所注释的 10 000 行代码。不仅如此，人们也很难再被推到 PDP-11/40 那样的环境，不得不用 RK05 盘组作为系统的引导介质。然而，你要得到的是来自贝尔实验室的第一个也是仅有的一个 UNIX 源代码的公开发行版本。像约翰所作的那样，一个操作系统一旦经过以一个程序的实际大小进行解释，就变成可以理解的，并且可以单独维护—这是现在的 UNIX 系统不能奢谈的。令人难以置信的是，这本优秀的 UNIX 系统的著作竟然在存在了 20 年之后才能出版!

如果 UNIX 系统的兴起是源于它在大学中的广泛使用，那么约翰的小册子则是其取得成功的直接原因。因此，我们要感谢约翰对 UNIX 系统的成功所作的贡献，以及他对我们今天所目睹的计算机系统技术的演进所作的贡献。

博尼・M・古德哈特

古德哈特是《UNIX Curses explained》一书的作者，《The Magic Garden Explained》及其“习题解答”的作者之一。他在悉尼的 Tandem 计算机公司任职。)

克 林 森 跋

自从我开始在 UNIX 系统上工作以来的 20 年中，计算技术已经有了重大变化。我要使读者获得某些有关《莱昂氏 UNIX 源代码分析》的印象，以表明这本书对于我们这些为系统使用开辟道路的人说，何以成为如此重要的著作。

我开始使用 UNIX 进行工作时，大约是在莱昂的那本书写成之前一年左右，这个 UNIX 系统后来称为 UNIX 版本 6。我曾是英国坎特伯雷郡肯特大学计算机科学系的一名讲师。我们在一台 PDP-11/40 上安装了 UNIX 系统，经过短时间的试验性运行之后，随即转入全天运行。我那时为支持 UNIX 系统的使用提供“服务”。我们没有磁带驱动器，UNIX 系统分布在两个 RK05 磁盘组上。其中，在一个 RK05 磁盘组上存放二进制系统代码、UNIX 核心系统源代码和系统用户手册的文档页面，而另一个 RK05 磁盘组则包含整个系统的源代码。一个 RK05 的存储容量将近 2.5MB，刚好可以容纳下整个系统。在开始阶段，我们不得不卸下第二个磁盘组，在机器上给用户留下工作空间。很久以后，我们获得了一个旧的 RP02，它的容量非常大，达到 20MB。

系统的大部分文档是由源代码编制成的。如果需要对系统作某些修改，那么就要先找到要作

修改的源代码。源代码中的绝大部分用 C 语言编制，是很容易理解的。C 语言本身也被写成文档，非常简明扼要，是按 C 语言参考手册编写的。我本人就是主要通过阅读源代码和修改这些代码学习 C 语言的。这不失为学习语言的一种好方法，你可以找到某些处理过程的例子，并按自己的使用要求对它加以修改。我是在同今天的新系统结缘之后才舍弃了这种方法。自然，在极端的情况下，还需要检查由编译程序生成的汇编语言。

但是，如果你安装了系统的某种新版本，并且不能工作，那么你必须对它作出修改。如果你是独立负责系统，没有 Usenet 新闻把寻找帮助的请求广播出去，不能用电子邮件求教于人，也无求助电话可以利用。总之，你处于孤立无援的境地。这种情况下，就必须了解你作出的每一处修改，这样才能保证修改的正确性。

有几件事情有助于系统的使用。首先，代码的编写几乎总是很好的。程序采用模块结构，所以在作某个修改时不需去了解程序的所有方面。程序几乎总是用 C 预编译器定义常数，所以很少会出现硬编码奇异数值常数而使解释无法进行的情形。每个程序都不大，只做好一件事情，这使得它们很容易理解。但是，如果你从总体上不了解系统，就不要去搞乱代码。

UNIX 核心系统适合于这种模式。它的大部分是用 C 语言写成，只用少量汇编语言胶合程序将系统同硬件连接。系统的模块化使各个部分之间的接口清晰。很多人是通过编写设备驱动程序入手去改动系统核心的，我本人也不例外。

有一些文档讨论应该如何编写设备驱动程序。丹尼斯·里奇写过一篇文章“UNIX I/O 系统”，解释设备驱动程序的工作原理。在由 USENIX 和 O'Reilly 出版的《4.4 BSD 程序员补充文档手册》中可以找到这篇文章。

需要使用设备驱动程序有几个原因。首先，是为了实现“先用再改”的需要。我为了把一台打印机加进系统，通过一个串行口连接打印机，该接口把打印机上的忙信号线加到中断信号线上，后者通常用于对接口传送电话铃声。这是我写的第一个设备驱动程序，更确切地说，是通过一些其他的驱动程序而建立的驱动程序。

第二，需要为新添置的硬件增加驱动程序。我曾经为一个早期的软盘驱动器写过一个驱动程序。后来又替剑桥大学环形网写了一个驱动程序，该网用来把校园的各计算机系统连接起来。最后，系统中存在若干我们确实不喜欢的缺陷。例如，终端接口存在的问题。当你删去一个字符时，它不从屏幕上消失，而回应一个字符“＃”。为改正这个缺陷，就需要对终端设备驱动程序从根本上进行修改。我也把许多时间花费在提高系统的性能上面。对 RP02 作过一些工作，试图通过在磁盘的中心启动文件系统使寻道时间减少到最低限度。

在新建立起来的 UNIX 用户组中，设备驱动程序开始成为受关注的问题。常把可以交流的软件保存在 RK05 磁盘组上，在用户之间流传。新出现的终端驱动程序就是一个绝好的例子，它最初是由波士顿儿童博物馆的用户写成，后来在英国经过三个人修改，就是在斯万塞的埃姆里，，在格拉斯哥的吉姆·麦基，在坎特伯雷的我本人。我们都必须了解代码，保证它能工作，并安装在自己的系统中。我们都必须在出现问题时采取措施。直到很久之后，我们才开始合作，生成一个统一的工作版本，并交由英国 UNIX 系统用户组(UKUUG)传播。

建立 Usenix 和 UKUUG 的初衷是为寻求帮助的人们提供一个场所，或者更进一步，直接提供这种帮助。在 Usenix 通信上公布了莱昂在 1977 年编写《莱昂氏 UNIX 源代码分析》的消息，Usenix 通讯后来被称为《UNIX 新闻》。同年晚些时候，UKUUG 通信也刊载同样消息。我们从澳大利亚订购了该书的 3 套复印本。

《莱昂氏 UNIX 源代码分析》一书使我获得一次检验自己的机会，看看自己已从源代码演绎出什么东西。我仔细记下系统中我还不理解的那些部分。我认为在书寄到时，我最需要了解的是如何管理进程，分叉/执行进程是如何工作的，以及进程调度，等等。该书解释了著名的注解：/＊你不需要了解此处＊／约翰按照贝尔实验室发布的形式编制源代码文档，使所有人都能理解他的书。约翰的目的是建立一种教学文献。该书确实需要读者用心阅读源代码，

一行一行地读，并理解正在发生的事情。这就意味着该书与你结合在一起，而不仅仅是文献而已。它试图使你理解系统要干什么。结果，书中充满着指明通过什么途径理解系统的问题。该书也指出代码中存在的缺陷，提出在哪些地方可以采用处理问题的更好方法。书中也提出具有挑战性的问题：试用一种更好的方法写出这行代码。

《莱昂氏 UNIX 源代码分析》使我在安装其他系统修改时具有信心，那些修改保存在磁带上，由约翰在澳大利亚的研究小组送到英国来。有些系统修改是由肯·汤姆森制作的，如像管道设备驱动程序的管线部分。对于了解新代码具备什么功能，总是需要遵循一些规则。如果新代码使系统中断，我就对它进行修改。约翰的著作在有些时日曾是我生活中的重要组成部分。约翰的书并未在全世界流行，至少没有正式流行。贝尔实验室在很短时间内接管了该书的发行工作。有一次，接到贝尔实验室通知，说明提供此书，同时允许学术性机构拥有一份拷贝。此后我得到了几本书。但是在那段时间，不少人说只是见到注书的影印本。流传过一些戏言，说该书一直居拷贝之冠。

约翰的著作对于想要了解 UNIX 系统的许许多多的人提供帮助和信息，它使学习 UNIX 变得非常容易。不仅如此，它使你所搜集的信息是可靠的。我确信，由这本书所哺育的一代 UNIX 精英，当他们回顾过去的时候，会对它的作者表示莫大的感激和崇高的敬意。

彼得·克林森

于英国坎特伯雷郡肯特大学

1996 年 3 月

(克里森是英国最早的 UNIX 用户之一，是他把《UNIX 新闻》介绍到英国。)

雷 杰 斯 跋

我第一次听说约翰先生的《莱昂氏 UNIX 源代码分析》是在 1977 年，那是在北卡罗来纳大学把 UNIX 版本 6 移植到 PDP-11/45 之后不久。该系统对我们来说仍然是很神秘的，所以我们急切地想了解这一传闻，就是新南威尔士大学有人编写了一本对 UNIX 进行全面分析的书。不久我们得到一个经过 5 次复制的影印本，同时有一个不留姓名的人打算用整个晚上的时间在复印室复制版本 6，而这是第 1 页上注明禁止的。

与此同时，发生了四件重要的事情。第一，我们发现了一段使我们鼓舞而不是烦恼的软件程序；第二，我们获得了对该计算机软件的文字批评材料；第三，在我们计算机科学的教学中，通过实际阅读一个完整的操作系统而取得明显的进展；第四，我们冲破了法律的约束。

我们先前曾讲授过一个操作系统，在某种程度上那是只能由大批程序员承担的系统。一旦写出系统，不允许进行分析，因为它由数百万行汇编语言指令组成。所幸的是肯·汤姆森和丹尼斯·里奇证明这是错误的，产生了 UNIX 版本 6 的核心系统，其代码不超过 10 000 行，不仅允许对它进行分析，而且要求作这样的分析。

阅读这本书的初学者可能获得这样一种印象，UNIX 是一个复杂的系统，还存在很多缺陷。但是必须强调，它是仅有的一个实际操作系统，适合用来作这样的分析。UNIX 是最容易理解的操作系统，出入在盘面上，一张盘即可容纳，尽管带有很多缺陷，它却完全超过了它以前的那些系统，同时也是那些后来者所不及的系统。虽然 UNIX 版本 6 可能被功能更强的 UNIX 新版本所超越，但是它在每一代码行提供的处理能力方面将是独一无二的。

整整一代 UNIX 核心的杰出人物都从这本书中学习，在 AT&T 和 Western Electric 公司对这份材料的发布和在教学中的使用实施限制以后，则从流行的各种地下版本中学习。我使用“黑客”这个词来指那些孜孜不倦的富有创造性的程序编写者，而不是后来所说的随意非法闯入各种计算机系统的黑客。然而事实毕竟是我们破坏了法律，占有和传播了由汤姆森，里奇和莱昂写成的这一特殊的技术文集。虽然我们宁愿对这种行为进行申诉，并把自己想像成某种维护电子世界自由权利的斗士，但这不是在进行任何真正的审判。Western Electric 公司只在

试图保护 UNIX 系统源代码的商业神密，而 UNIX 的某种成功则使 Western Electric 公司的努力遇到日益增大的困难。

最终，由于《莱昂氏 UNIX 源代码分析》一书具有如此大的价值，Western Electhr 公司不得不放宽限制，允许持有 UNIX 源代码许可证的每一个客户获得一个拷贝，但严禁再进行任何形式的复制。但有了这个让步，人们必须打听这本书的现存情况，并向 Western Electhr 公司提出正式请求，自然包括翻箱倒柜，查明自己拥有一份实在的 UNIX 源代码的许可证。

本书对于我们这些学习 UNIX 的人的巨大价值，以及围绕其传播所进行的暗中活动，使本书增添了神秘的色彩。由于在学校的操作系统教学课堂上不能公开讨论这本书，我们中的一些人就在夜间聚会，到一间无人的教室进行讨论。成为一种地下活动的积极分子，这在我的一生中是仅有的一次。

虽然《莱昂氏 UNIX 源代码分析》并不代表 UNIX 系统的最新实现技术，但是它提供给计算机科学系的学生们一种机会，读到由杰出的开拓者们写成的产品代码。在我从事 20 年的编程工作之后，我仍然把自己看成是学习计算机科学的学生中的一员。越过肯和丹尼斯两位大师的肩膀，去窥视他们所开展的工作，这是第一次也是最重要的一次机会。必须强调，这是最终产品代码；UNIX 版本 6 除了应用于计算机科学的研究领域之外，还在许多应用系统中使用。我们在教科书见到的代码实例可能是精致的，但编者往往忽略了实际编程时所面临的种种困难，例如错误处理和中断处理。

除了感谢丹尼斯和肯表达自己的系统结构思想，感谢约翰将其整个地翻译成一种甚至更高级的语言，我们也要感谢 AT&T 贝尔实验室的管理者们，他们使得这些事情得以实现。历史上不乏这样的先例，最基本的技术进展落到没有回报的境地。UNIX 面临的最大问题，也是半导体和激光技术遇到的问题。贝尔实验室再一次为我们提出 UNIX 这一如此重要的技术，而在本质上它是属于全世界的。过去我总是感激 AT&T 没有对它发明的晶体管实施基本专利条款，而今我要感激他们在 UNIX 源代码的发布上仍然如往昔一样慷慨无私。但是我常常为持有约翰的《莱昂氏 UNIX 源代码分析》影印本而感到内疚，尽管它对于我接受教育和我的职业生涯来说如同 UNIX 本身对我那样地重要。现在我翘首以盼这本书的面世。

彼得·斯于 IBM 沃森研究中心

1995 年 2 月

(雷杰斯现在 NetSpeak 公司从事多媒体因特网应用开发工作。在作为计算机科学家的头 10 年中，他投身于多项不同机器上独具 UNIX 特色的开发，曾为卡罗来纳微电子中心建立了用在工作站上的 VLSI CAD 应用系统。此后，他积极参与用 Prolog 语言实现逻辑程序设计的项目，以及在 IBM 沃森研究中心进行演绎数据库的应用研究。)

作 者 小 传

John Lions 1937 年出生于澳大利亚悉尼。1959 年就读于悉尼男子高级中学，后来在悉尼大学获得应用数学理学士学位。从那里得到去英国剑桥大学深造的奖学金，在剑桥参与了 EDSACⅡ的研究项目，并于 1963 年获得控制工程博士学位。随后，他受雇于加拿大多伦多的 KCS 公司，任该公司的顾问，1967 年受聘于加拿大新斯科夏省戴尔毫斯大学，任副教授和计算中心主任。1970 年，Lions 到了美国，在洛杉矶以系统设计师的身份参加 Burroughs 公司 B5000 的设计工作。他于 1972 年回到澳大利亚，应聘到新南威尔士大学执教，担任高级讲师。

1974 年 5 月，John 从美国《计算机协会通信》上看到 Ritchie 和 Thompson 两人合写的论文“UNIX 分时系统”，这篇带有浓厚经验主义色彩的文章引起了他对 UNIX 的强烈兴趣，并鼓动新南威尔士大学计算机科学系的同事们去申请一份 UNIX 系统的许可证。校园内的一批人随之把注意力转到新建立的 UNIX 计算机系统上，因为这套系统同当时使用的 1BM 360/50

或 Control Data 公司的 Cyber 72-26 系统相比，能够提供更佳的性能和具备更好的连接性。UNIX 甚至可能使行式打印机按照其额定打印速度工作!由于各个方面表现出来的兴趣，推动 John 在澳大利建立了 UNIX 系统用户组。

John 在 UNIX 系统上的持久兴趣促使他把这个操作系统用到教学中。1976 年他为学生写了一本小册子，取名为“莱昂氏 UNIX 操作系统源代码，第 6 版本”，其中包含了 UNIX 版本 6 的几乎全部代码，这个版本运行在 DEC 公司的小型计算机 PDP-11/40 上，只有少数设备驱动程序没有收进去。次年，他补充了一组解释性的注释，目的是为他的学生介绍代码。这些注释最终演变为著名的《莱昂氏 UNIX 源代码分析》

John 写信给 USENIX(学术界 UNIX 用户协会)的 Mel Ferentz,Loce Katz 和其他一些成员，提议将他写的那些注释的拷贝给他们使用。经过同 Western Eteetric 公司进行专利许可谈判后，他把注释分发给 UNIX 系统的其他许有证持有人。随后就有源源不断的订单从海外发来，每一次，他接到贝尔实验室发来的一份 200 个拷贝的批量订单。

在 1978 年的一天晚上，John 接到 Doug Mcliroy 从贝尔实验室打来的电话，表示顾意承接那些注释的分放工作。John 欣然应允，因为这样一来就可以使他不再勉为其难，得以从打印、包装和发运这样一些工作中解脱出来 —在此之前，他已经分发了大约 600 份拷贝!

1928 年夏秋之间，John 造访了座落在新泽西州墨累山庄的贝尔实验室，这是应 Bekdey Taglle 之邀到那里休年假，并作为技术小组成员在那里工作。不久之后，他被聘任为新南威尔士大学计算机科学系的副教授。1983 年，John 再次到贝尔实验室休假，而第三次去那里休年假是在 1989 年，这也是最后一次。休假期间他准备了对 UNIX 的另外一组注释，同他前次的注释相似，这次是基于“方案 9”(Plan 9)的。所写的文档限制在贝尔实验室内部使用，定名为《方案 9：第 1 卷：核心源代码》和《方案 9：第 2 卷：综述与评注》。在往后的几年中，Jshn 继续从事他的学术研究工作，并更专注于计算机网络。

John 是“计算机协会”的会员和“澳大利亚计算机协会”的高级成员。在 1982 年至 1987 年间，John 曾担任《澳大利亚计算机杂志》主编，他也是“澳大利 UNIX 系统用户组”的终身会员。

John Lions 在新南威尔士大学服务 23 年之后，由于健康原因，在 1995 年 7 月退休。

(这篇 John Lions 的小传由 Berny Goodheat 执笔，写作中得到 John 的妻子 Marianne Lions 和他在新南威尔士大学的学生们的帮助。