关于第六版UNIX操作系统的评论

约翰·莱昂斯
新南威尔士大学计算机科学系

原文John Lions, MC翻译(2018)

这本小册子是为新南威尔士大学的学生制作的，课程分别为6.602B和6.657G；它旨在作为UNIX Level 6操作系统源代码小册子的配套书籍和评论。UNIX 6软件系统由新泽西州Murray Hill贝尔实验室的Ken Thompson和Dennis Ritchie编写，它已在西部电气公司的许可下提供。本文档可能包含一个或多个许可、版权和保密协议所涵盖的信息。本文档的发行仅限于Western Electric的UNIX软件系统许可证持有者。

目录

前言

本书详细阐述近年来出现的一个最有趣的计算机操作系统的内核，这就是UNIX分时操作系统。它运行在DEC公司较大型号的PDP11计算机系统，由贝尔实验室的肯·汤普森和丹尼斯·里奇开发并由ACM通信于1974年7月首次向全世界宣布。我们在教学实践中很快确信UNIX是学生正式学习操作系统的一个有趣版本，其原因简述如下：

• UNIX能在我们已有的硬件系统上运行;
• 它紧凑且易于使用;
• 它提供一套广泛且实用的设施;
• 它本质上很有趣，实际上在许多领域开辟了新的天地。

UNIX的魅力和优点之一是源代码的紧凑性，当仅表示少量外围设备时，它驻留在内核(nucleus)的源代码小于9,000行，这是令人愉快的。人们经常建议1,000行代码表示一个程序的实际限制规模，这样它就能被一个人理解和维护。大多数操作系统要么比UNIX第六版超出1~2个数量级，要么为用户提供一套非常有限的功能；这意味着除了极少数最坚定、最专注及最坚韧的学生之外，其他人都无法理解操作系统的细节；或者，操作系统是相当专用的，学生们对它并没有什么内在的兴趣。

讲授操作系统似乎有三种主要方法。

1. 一般原理法

该方法详细阐述基本原理并引用各种现有操作系统进行说明，其中大多数正好超出学生的直接经验。这是COSINE委员会倡导的方法，然而，我们认为许多学生要么不够成熟、要么经验不足、难以从中获得有益的收获。

2. 模块搭建法

学生自己能搭建一个小型或玩具型的操作系统，毫无疑问，这是一项有价值的工作。如果组织得当，那么它必然包含实际操作系统的复杂性和先进性、并且通常倾向于操作系统设计的一个方面如进程同步。

3. 实例研究法

它最早是由“课程68”系统编译课程推荐的一种方法，这门课程是基于ACM计算机科学课程委员会的一个报告设立的，ACM于1968年3月发布了这份报告。

十年之前，这种主张“在该科目的大部分课程中研究单一系统”的方法是不切实际的，因为安排足够多的学生访问一个合适系统的费用太高。十年之后，经济形势发生重大变化。如果小型机系统成为一个研究对象，那么学习费用就不再是一个决定性的障碍。该方法的一个显著优点是能对现有系统进行详细分析。我们认为：学生有机会从各方面研究一个运行的操作系统，无疑是非常有价值的。除此而外，一个主修计算机科学专业的学生在其职业生涯中应阅读并理解至少一个包含多维度的核心程序，这个任务无疑是有益的。

1976年，我们在澳大利亚新南威尔士大学的操作系统课程中采用UNIX作为案例研究主题，这些笔记最初是为帮助我们的学生研究这两门课程(6.602B和6.657G)而准备的，每学期上一门课程，在进入任何一门课程之前，学生们被认为已学习PDP11架构和汇编语言，并且有机会在早期课程的练习中

使用UNIX操作系统。一般来说，学生似乎发现这两门新课程的任务更重，但是它比此前基于COSINE委员会”“一般原理法“的课程更令人满意。

需要提及有关UNIX系统作者提供的文档，这是在大学校园内复印的，包括两卷A4尺寸纸张，总厚度为3厘米，重量为1250克。

第一个观察结果：这套书(含注释和源代码)能放在在学生的书包中，因此，我们假设这种以新颖、简洁、突发奇想的方式写成的文件数量能满足学生的需求。

第二个观察结果：在获得足够的经验后，我们认为这套书用于参考资料时是非常全面的，然而，还是有足够的余地在其中添加其它教程材料，我们希望这本注释能满足其中的空白。

实际的UNIX操作系统源代码记录在一本单独的配套书中，它是UNIX操作系统源代码(UNIX Operating System Source Code)的实现，它于1976年7月首次打印的。在1975年12月收到UNIX第六版本后，我们特别为本书编选UNIX第六版的源代码。在1976年期间，本书的第一版本以nroff形式分发，并在今年下半年使用“nroff”形式对程序进行了格式化处理。我们最近有机会修改和订正早期的材料，并把它们整合到现在的版本中。

关于源代码的呈现顺序，我们必须尽早做出决定。我们的目的是为计划学习整个操作系统的学生提供一个合理的逻辑顺序。事后看来，仍然可以在细节方面进行大量改进，并且这些改进将在未来的版本中付诸实施。

我们希望本书能使学生们能有趣地学习UNIX分时操作系统并从中获得宝贵的经验。虽然本书的主要目的不是用于参考，但是有些人希望将它用作参考书。我们最后提供的索引应在满足这一水平上的参考材料要求方面有所改进。

由于这些注释涉及到Western Electric Company管理的专有材料，因此它们只能提供给UNIX分时操作系统许可证的持有者，因此无法通过更常用的渠道发布。我们非常受欢迎对这些笔记的改进、批评和建议。

致谢

许多同事和学生都鼓励和支持这些笔记的准备，包括David Carrington、Doug Crompton、Ian Hayes、David Horsfall、Peter Ivanov、Ian Johnstone、Chris Maltby、Dave Milway、John O'Brien和Greg Rose。

Pat Mackie和Mary Powter打印了本书初稿的大部分，而Adele Green在把笔记转移到"nroff"格式方面做出很大贡献。David Millis和新南威尔士大学的出版科大力协助出版机制，Ian Johnstone和澳大利亚管理研究生院为最终草案的编写提供便利。

在整个项目中，我的妻子Marianne给我不懈的道义支持和实践支持以及校对工作。最后感谢Ken Thompson和Dennis Ritchie，他们启动了UNIX的一切开发工作。综上所述，我谨表示衷心的感谢。

我还必须提到"nroff"计划的合作。没有它，这些笔记就永远不会以这种形式产生。然而，它不情愿地产生一些更神秘的秘密，作者的感激确实是混合的。当然，"nroff"本身必须为程序记录员的艺术的未来实践者提供一片沃土。

约翰·莱昂斯
肯辛顿，新南威尔士州，1977年5月

第零部分 简介、处理器、C语言及概述

1 简介

UNIX是PDP-11计算机分时操作系统的名称，它是由美国贝尔实验室的肯·汤普森和丹尼斯·里奇编写，他们两人在1974年7月出版的ACM通讯中对它进行了描述。事实证明，UNIX在运行中是有效、高效和可靠的，截止到1976年底，它已在150多个机器中安装并投入使用。

虽然编写UNIX操作系统的工作量是不容忽视的，但是它仅花了大约10人年的工作量，这与其它主流操作系统的开发工作量相比是微不足道的。例如，截止到1968年，IBM的OS/360的开发连超过5000人年，而TSS/360超过1000人年。

当然，有些操作系统比UNIX更易于理解，然而，我们能断言它们更加简单且试图实现更少的目标。就提供给用户的功能列表而言，UNIX是当今主流的操作系统之一。事实上，UNIX提供的许多功能是其它主流操作系统所缺少的。

1.1 UNIX操作系统

UNIX操作系统源代码文档及其配套文件是介绍UNIX分时操作系统的一份主要文档，即代码在UNIX操作期间常驻在主存中，此代码具有以下主要功能：

- 初始化;
- 进程管理;
- 系统调用;
- 中断处理;
- I/O操作;
- 文件管理。

1.2 实用工具

UNIX剩余的部分更大！它由一组适当定制的程序组成，作为用户程序(User Programs)运行，并且由于缺少更好的术语，我们称它们为实用工具，在该标题下出现许多与操作系统具有很强共生关系的程序，例如

程序：
- shell：命令语言解释器
- /etc/init: 终端配置控制器

文件
- check chmod clri
- df
- mdir mkdir sync mkfs
- du rmdir mkdir sync mkfs umount mount update

应指出上述UNIX实用工具执行的许多功能在其它计算机系统中被视为操作系统的功能；正如在本书定义阐述的，它确实为参照UNIX的其它操作系统做出重大贡献。

关于上述程序的功能和用途的描述，读者能在UPM(Unix Programmer's Manual)中找到，例如，在第I部分的常用程序或第VIII部分中仅由System Manager使用的程序。

1.3 其它文件

这些说明经常引用UNIX程序员手册(UPM)，偶尔参考UNIX文档手册(UNIX Documents Booklet)，并且不断引用UNIX操作系统的源代码，所有这些都与完全理解系统有关。此外，对汇编语言程序的全面研究需要参考DEC(DEC被康柏公司收购后并入惠普公司)出版的PDP11处理器手册。

1.4 UNIX程序员手册

UPM分为八个主要部分，由一个目录和一个KWIC(关键词在上下文)索引预先设定，后者大多非常有用，但偶尔很烦人，因为某些索引材料不存在，而某些现有材料未编入索引。

在本手册的每个部分中，材料按主题名称的字母顺序排列，其中部分编号通常附加到主题名称，因为一些主题出现在多个部分中，例如“CHDIR(I)”和“CHDIR(II)”。

第I节包含命令(commands)，这些命令是由“shell”命令解释器识别或标准用户实用程序的名称;

第II节包含系统调用(system calls)，它们是操作系统例程，它们能从用户程序调用以获取操作系统服务，对操作系统的研究将使学生熟悉大多数系统调用;

第III节包含子程序(subroutines)，它们是能从用户程序调用的库例程；对普通程序员来说，第II节和第III节之间的区别似乎有些武断；第III节的大部分内容与操作系统无关;

第IV节描述特殊文件，它是外设的另一个名称；其中一些是相关的，一些仅仅是有趣的；这取决于你在哪里使用它;

第V节描述文件格式和惯例，本节隐藏许多高度相关的信息;

第VI和VII节描述用户维护程序和子程序，不是特别喜欢UNIX的人(UNIXophile)会忽略这些部分，但它们与操作系统并不是特别相关;

第VIII节描述系统维护，它们不是硬件，而是软件！如果你对如何管理UNIX安装感兴趣，那么这里有很多有用的信息。

1.5 UNIX文档

这是一个有点不相关程度的杂项集：

• 设置UNIX确实属于UPM的第VIII部分(它是相关的);
• UNIX分时系统是原创ACM通信论文的更新版本，它应每月至少重读一次;
• 如果你的UNIX体验有限，那么UNIX for Beginners这本书非常有用;
• 除非你是专家，否则C语言编译器的教程以及C语言汇编程序的参考手册非常有用;
• UNIX I/O系统很好地概述操作系统的许多功能;
• UNIX Summary提供一个检查列表，能用于回答操作系统的功能吗?

1.6 UNIX操作系统源代码

它是贝尔实验室提供的UNIX操作系统的编辑版本。
代码选择假定一个模型系统包括：

• PDP11/40处理器;
• RK05磁盘驱动器;
• LP11行式打印机;
• PC11纸带读取器/打孔器;
• KL11终端接口。

源代码的主要编辑更改如下：

• 文件的呈现顺序已经改变;
• 多个文件中的材料顺序已更改;
• 在非常有限的范围内，代码已经在文件之间传输(事后看来，这是可取的);
• 大约5%的线路以各种方式缩短至少于66个字符(通过消除空白、重新排列注释、分成两行等);
• 引入由一些列下划线字符组成的评论，特别是在程序结束时;
• 通过用空白行填充，把每个文件大小调整为50行的倍数;

源代码以双列格式打印，每列50行，每页(或页面)给出100行。因此行号和表号之间存在方便的关系。

源代码卷的开头包含许多摘要：

• 按目录顺序显示文件的目录及其包含的程序;
• 带有行号的按字母顺序排列的程序列表;
• 带有值的已定义符号列表;
• 交叉参考列表，给出使用每个符号的行号(在C中保留的单词和一些常用符号如“p”和“u”已被省略)

1.7 源代码选择

源代码分为五节，每节主要用于系统的一个主要方面。

我们基本实现每个部分都具有足够的独立性的目的，以便能作为一个单元进行研究，并在其后继者掌握之前进行研究：

第I节包含系统初始化、过程管理、汇编语言例程;

第II节涉及中断、陷阱、系统调用和信号(软件中断);

第III节主要讨论用于程序交换的磁盘操作、基本的面向块的输入/输出、操纵大缓冲池;

第IV节涉及文件和文件系统：创建、维护、操作和销毁文件;

第V节涉及字符特殊文件，这是用于慢速外设的UNIX术语，它由一个通用的、面向字符的缓冲池操作。

各部分的内容在第四章详细介绍

1.8 源代码文件

刚刚描述的五个部分中的每一个都包含几个源代码文件。每个文件的名称都包含一个标识其类型的后缀：
“.s”表示汇编语言陈述的文件;
“.c”表示可执行“C”语言语句的文件;
“.h”表示“C”语言语句的文件，不是用于单独编译，而是在编译时包含在其它“.c”文件中，即“.h”文件包含全局声明。

1.9 笔记应用

这些注释旨在补充源代码中已有的注释，对于理解UNIX操作系统并不重要；没有它们是完全可能的，你应该尝试这样做。

这些笔记是一个助手，它能在困难时帮助你。如果你首先尝试自己阅读每个文件或程序，那么你刚开始的进度可能会更慢、但最终进度会更快。阅读其他人的程序是一门艺术，应该进行学习和实践，因为它是有用的!

1.10 关于编程标准的注释

你会发现UNIX中的绝大部分代码具有非常高的标准。尽管许多章节在开始阅读时是复杂而晦涩的，但是在进一步研究和反思的照耀下，就显得非常明显，这是读者思维飞跃的唯一方式。

出于这个原因，关于编程风格的注释中偶尔会有一些评论；这些评论几乎总是指出：“从近乎完美的常用标准而言，有些代码有明显的错误。”

这些错误是什么引起的？有时，似乎原始代码的失误已被干净利落地打上补丁。显而易见，没有一次不是这样的，“坏”代码实际上已包含一些最初并不明显的细微特征。人类偶尔会有弱点，当然需要一些鼓励。

总而言之，你将发现UNIX操作系统的作者肯·汤普森和丹尼斯·里奇已创建一个强大、完整和有效的程序，你应该钦佩并试图效仿。

2. 基础知识

UNIX运行在由DEC制造的PDP11系列计算机运行。本章简要概述这些计算机的某些特性，特别是PDP11/40。如果读者之前没有熟悉PDP11系列，那么请立即转向DEC出版的“PDP11处理器手册”。PDP11计算机由处理器(CPU)通过称为“Unibus”的双向并行通信线路连接到一个或多个存储器单元和外围控制器组成。

2.1 CPU

该CPU的设计大约为16位字长，用于指令、数据和程序地址，并集成许多高速寄存器。

2.2 CPU状态字

这个16位寄存器有子字段，解释如下：

位描述：
14,15当前模式(00=kernel;）12,13前一模式(11=user;）5,6,7处理器优先级(range 0..7) 4陷阱位
3 N，如果之前的结果为负，那么设置为3 N
2 Z，如果先前的结果为零，那么设置为2 Z
1 V，如果先前的结果溢出，则设置为1 V
0 C，如果前一个操作给出进位，那么设置为0 C

CPU能以两种不同的模式运行：内核态(Kernel Mode)和用户态(User Mode)，内核态在两者中有更多的特权，并由操作系统保留给自己使用，这两种模式的选择决定以下实现：

• 内存管理段寄存器，用于把程序虚拟地址转换为物理地址;
• 实际寄存器用作r6，即栈指针(stack pointer);
• 是否遵守某些指令，例如“halt”。

2.3 通用寄存器

该CPU包含许多16位寄存器，其中8个随时作为“通用寄存器”访问，这些被称为r0、r1、r2、r3，r4、r5、r6和r7。前六个通用寄存器可用作累加器、地址指针或索引寄存器，UNIX中使用这些寄存器的一般约定如下：

r0、r1在表达式求值期间用作临时累加器，用于返回过程的结果，并且在某些情况下用于在过程调用期间传递实际参数;

r2、r3、r4在过程执行期间用于局部变量，它们的值几乎总是存储在程序进入时，并在程序退出时重新存储;

r5用于指向存储在当前栈中的过程激活记录的“动态链”的头指针，它被称为环境指针(environment pointer)。

r6、r7这两个通用寄存器确实具有特殊意义，并且对所有意图都是“特殊目的”：

r6(称为“sp”)用于栈指针，PDP11/40处理器包含两个独立的寄存器，能用于“sp”，具体取决于CPU是处于内核态还是用户态；没有其它通用寄存器以这种方式重复;

r7(也称为“pc”)用于程序指令地址寄存器。

2.4 指令集

PDP11指令集(Instruction Set)包括双、单和0操作数指令，指令长度通常是一个机器字，一些指令被扩展为两个或三个机器字以及附加的寻址信息。对于单操作数指令，操作数通常称为“目标操作数”；对于双操作数指令，两个操作数分别称为“源操作数”和“目标操作数”，我们稍后描述各种寻址模式。

在文件“M40.S”中使用以下指令，即与PDP11/40处理器一起使用的汇编语言支持例程文件。请注意N、Z、V和C是条件代码，即处理器状态字(“ps”)中的位，这些设置为除“bit”、“CMP”和“TST”之外许多指令的副作用，其声明的函数是设置条件码。

adc： add carry, 把C位的内容加至到目的操作数;
add： add the source to the destination, 把源操作数添加到目地操作数;
ash： arithmetic shift, 按移位计数将指定寄存器的内容左移相应次数 (负值意味着右移);
ashc：arithmetic shift combined, 除两个寄存器，与ash指令类似;
asl： arithmetic shift left, 把所有位向左移一位，位0加载到0，位15则加载到C;
asr： arithmetic shift right, 把所有位向右移一位，位15被复制，位0加载到C中;
beq： branch if equal, 若等于，即若Z = 1，则分支;
bge： branch if greater or equal, 若大于或等于，即N = V，则分支；
bhi： branch if higher, 若大于，即C = 0且Z = 0，则分支；
bhis： branch if higher or same, 若大约或相同，即C = 0，则分支;
bic： bit clear, 若源操作数中的位为非0值，则将目的操作数中的相应位清除为0;
bis： bit set, 对源和目的操作数执行或操作，并把结果存储在目的操作数;
bit： 执行源和目的操作数执行逻辑与操作，设置条件代码;
ble: branch if less than, 若大于或等于，即Z = 1或N = V，则分支;
blo： branch if lower, 如低于0，即C = 1，则分支;
bne： branch if not equal, 若不等于(0)，即Z = 0，则分支;
br： branch always, 分支到.-128和.+127之间的一个单元，其中“,”是当前单元地址;
clc： clear C;
clr： clear destination to zero, 清除目的操作数为0;
cmp： compare, 比较源和目的操作数，设置条件代码，若源操作码小于目的操作码，则设置N位;
dec： 从目的操作数的值减1;
div： divide, 存储在rn和r(n+1)的32位二进制补码(n为偶数)除以源操作数，商保留在rn中，余数保留在r(n+1)中;
inc: increment, 在目的操作数加1;
jmp： jump, 跳转到目的地址;
jsr： jump to subroutine, 跳转到子程序，寄存器值的缩小如下：pc, rn, -(sp) = dest., pc, rn
mfpi: move from previous 1 place, 把上一个地址空间中指定字的值压如到当前栈;
mov： move, 把源操作数值复制到目的操作数;
mtpi： move to previous 1 place, 弹出当前栈并把值存储在“上一个”地址空间中的指定机器字中;
mul： multiplly, rn和源操作数相乘，若n是偶数，则积存在rn和r(n+1);
reset：把Unibus上的INIT线设置为10毫秒，其作用是重新定义所有设备控制器的效果;

ror： rotate right, 把目标的所有位向右旋转一个位置，把位0加载到C中，并把先前的C值加载到位15中;
rts： return from subroutine, 从子程序返回，从rn重新加载pc，并从栈重新加载rn;
rtt： return from trap, 从中断或陷阱返回，从栈中重载pc和ps;
sbc: subtract carry, 从目的操作数减去进位;
sob: subtract one and branch, 从指定的寄存器中减1，若其结果不为0，则返回“偏移”字;
sub： subtract, 从目的操作数减源操作数;
swab: swap bytes of word，交换目的操作数中的高、低字节；
tst： test bits, 根据目的操作数的内容设置条件代码N和Z;
wait: 等待处理器闲置并释放Unibus直到发生硬件中断。

与上面说明的字版本指令一样，下列指令的“字节”版本用于汇编文件“m40.s”：
bis inc
clr mov
cmp tst

2.5 寻址模式(Addressing Mode)

PDP11指令集的新颖性和复杂性大部分来自于它所提供的寻址模式，用于定义源和目标操作数。
下面描述在汇编文件“m40.s”中使用的寻址模式。

寄存器模式(Register Mode)：操作数驻留在一个通用寄存器中，例如
```
clr r0
mov rl, r0
add r4, r2
```

在以下模式中，指定的寄存器包含用于定位操作数的地址值。

寄存器延迟模式(Register Deferred Mode)：寄存器包含操作数的地址，例如，
```
inc(rl)
asr(sp)
add(r2)，rl
```

自动增一模式(Autoincrement Mode)：寄存器包含操作数的地址，它产生副作用，即寄存器在操作之后递增，例如：
```
clr(rl)+
mfpi(r0)+
mov(r1)+, r0
mov r2, (r0)+
cmp(sp)+, (sp)+
```

自动减一模式(Autodecrement Mode)：寄存器递减1，然后用于定位操作数，例如，
```
inc - (r0)
mov - (r1), r2
mov(r0)+， - (sp)
clr - (sp)
```

索引模式(Index Mode)：寄存器包含一个值，该值被添加到指令之后的16位字以便形成操作数地址，例如：
```
clr 2(r0)
movb 6(sp), (sp)
movb _reloc(r0), r0
```

```
mov -10(r2), (rl)
```

在该模式下，寄存器是索引寄存器或基址寄存器，后一种情况实际上在汇编文件“m40.s”中占主导地位。上面的第3个例子实际上是寄存器作为索引寄存器的少数用途之一，请注意“reloc”是可接受的变量名。

有以下两种寻址模式，其使用仅限于以下两个示例：

```
jsr pc，(r0)+
jmp * 0f(r0)
```

第一个例子涉及使用自动延迟模式，这发生在第0785、0799行的例程调用中，用于执行的例程的地址能在由r0寻址的字中找到，即涉及两个间接级别；r0作为副作用递增的事实与此用法无关。

第二个例子出现在第1055行、第1066行，这是索引引用模式的一个实例；jump(跳转)的目标地址是机器字的内容，其地址标记为“0f”加上r0的值，它是小的正整数，这是实现多路交换机的标准方式。以下两种模式使用程序计数器作为指定寄存器实现某些特殊效果。

立即模式(Immediate Mode)：这是程序计数器(pc)自动增一模式，因此从程序串中提取操作数，即它成为一个立即操作数，例如以下操作数。

```
add $2, r0
add$ 2, (rl)
bic $17, r0
mov $KISA0, r0
mov $77406, (rl)+
```

相对模式(Relative Mode)：这是程序计数(pc)索引模式，从程序字符串中提取相对于当前程序计数器值的地址，并将其添加到pc值以形成操作数的绝对地址，例如:

```
bic $340, PS
bit  $l, SSR0
Inc SSR0
mov(sp), KISA6
```

请注意索引、索引延迟、立即模式和相对模式的每一个把指令大小扩展一个机器字，而自动增一和自动减一模式的存在以及r6的特殊属性使开发者能方便地把许多操作数存储在栈中、或在存储器中向下增长的后进先出(LIFO)列表中。这就产生许多优点。例如，代码字符串长度更短且更容易编写位置独立代码。

2.6 UNIX汇编程序

UNIX汇编程序是一个没有宏功能的双通汇编程序，其完整描述能在UNIX汇编参考手册(UNIX Assembler Reference Manual)中找到，该手册包含在UNIX Documents中。以下简要说明应该是一些辅助文档：

（a）一个数字串可定义一个数值常数；除非字符串以句点(“,”)结束，当它被解释为十进制数时，它被假定为八进制数。

（b）字符“/”用于表示该行的其余部分是注释;

（c）如果同一行中有两个或两个以上的语句，则必须用分号隔开;

（d）字符“,”用于表示当前位置;

（e）UNIX汇编程序使用字符$和“*”，其中DEC汇编程序分别使用“#”和“@”。
（f）标识符由一组字母数字字符(含下划线)组成；只有前八个字符是重要的，其中第一个字符可能不是数字;
（g）在C程序中出现的变量是已知的全局变量，通过添加由一个下划线组成的前缀进行修改，例如在汇编语言文件“m40.s”的第1025行出现的变量“regloc”，它引用自“trap.c”文件第2677行的变量“regloc”;
（h）有两种语句标签(labels)：名称标签和数字标签；其中，数字标签由一个数字后跟一个冒号组成，它不一定是唯一的；若引用“nf”—其中“n”是数字，则表示通过向前搜索找到的第一个标签“n:”。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”;
（i）形如identifier = expression的赋值语句把值/类型与标识符相关联。例如.=60^，其中运算符’^’传递第一个操作数的值和第二个操作数的类型，在本例中为存储单元“location”;
（j）字符串引号符号为“<”和“>”。
（k）形如.globl x, y, z的语句，使名字“x”、“y”和“z”为外部名;
（l）名称“edata”和“end”是加载器的伪代码变量，它们分别定义数据段的大小且数据段加上bss段。

2.7 内存管理

在PDP11上运行的程序能直接寻址高达64K字节(32K机器字)的存储，这与16位地址大小一致。由于这种方法既经济又合理，因此较大的PDP11型号能配备更大的内存(PDP11/40最多256K字节)以及把16位虚拟(程序)地址转换为18位或更高的物理地址。PDP11/40的存储器管理单元(MMU)比PDP11/45或PDP11/70更简单。

在PDP11/40上，存储器管理单元由两组寄存器组成，它们把虚拟地址映射到物理地址。这些寄存器被称为活动页寄存器(Active Page Registers)或段寄存器(Segmentation Registers)。一组寄存器在CPU处于用户态时使用，而另一组寄存器在CPU处于内核态时使用。更改寄存器的内容就改变映射的详细信息，这种更改的特权由UNIX操作系统保留自用。

2.8 段寄存器

每一组段寄存器包含8对寄存器，每一对寄存器包括页地址寄存器(PAR)和页描述寄存器(PDR)，每一对寄存器控制一页的映射，这意味着页长8K字节(或4K机器字)相应的虚拟地址空间是8页。每一个页面由128个块(block)构成，每一块长度为64个字节(或32个机器字)。块大小(block size)不仅是内存映射函数的粒度容量(grain size)，而且是内存分配的粒度容量。任何虚拟地址不是属于这一页就是属于那一页。

相应的物理地址是通过把页面的相对地址与相应PAR的内容相加生成的，这就形成一个扩展地址(PDP11/40和PDP11/45上的18位;11/70上的22位)，因此每个页地址寄存器对一个页起着重新分配寄存器的作用。每个页划能分为32个机器字边界(?)，它分为高地址部分和低地址部分。每个部分的大小是32个字的整数倍。特别地，一个部分可为空，在此情况下，另一部分就能占用整个页，其中一部分被认为包含有效的虚拟地址，其余部分中的地址被宣布无效。任何引用无效地址的尝试都将被硬件捕获。该方案的优点是物理存储器中的空间仅需要被分配用于页的有效分配物理存储空间。

2.9 页描述寄存器

页描述寄存器定义：

（a）该页低地址部分的大小(存储的数字实际上是32个字块的数量少于一个);
（b）当搞地质是有效部分时设置的位，就称为扩展方向位;
（c）定义“无访问”或“只读访问”或“读/写访问”的访问模式位。
请注意：如果有效部分为空，那么必须通过将访问位设置为“无访问权”。

2.10 内存分配

硬件没有规定应分配物理存储器中与页有效部分相对应的区域，除非它们必须以32字边界开始和结束。这些区域能按任何顺序分配，并且能在任何程度上重叠。

在内存的操作实践中，物理内存区的分配更为严格，我们将在第7章中看到这一点。相关页的区域通常按其页顺序连续分配物理内存，以便一个程序相关的所有段区域包含在物理内存的一个或至多两个区域内。

2.11 状态寄存器

除段寄存器外，PDP11/40还有两个存储器管理的状态寄存器：
SR0：包含中止错误标志(flags)和OS的其它基本信息；特别是当SR0的第0位打开时，启用内存管理;
SR2：在每次取指令开始时加载16位虚拟地址。

2.12 “i”和“d”空间

在PDP11/45和PDP11/70系统中，还有一组附加的段寄存器，使用pc寄存器(r7)创建的地址被称为属于“i”空间，并且由一组不同的段寄存器转换，这些段寄存器用于表示属于“d”空间的剩余地址。这种安排的优点是“i”和“d”空间能占用多达32K字，允许把程序所需的最大空间增加到PDP11/40上可用空间的两倍。

2.13 初始条件

在Unibus上的所有设备重新初始化后首次启动系统时，MMU内存管理单元被禁用且CPU处于内核态。在此情况下，0~56K范围内的虚拟(字节)地址映射到等价的物理地址。然而，虚拟地址空间的最高页面被映射到物理地址空间的最高页面。

PDP11/40或PDP11/45的地址区间：

0160000–0177777

被映射到下列物理地址范围：

0760000–0777777

2.14 专用设备寄存器

物理存储器的高页被保留用于与处理器和外设相关联的各种特殊寄存器。通过以这种方式占用一页内存空间，PDP11设计者能使各种设备寄存器访问，而不需要提供特殊的指令类型。在这一页中，为寄存器分配地址的方法是一种创新，其价值是毋容置疑的。

3. 阅读C程序

学习阅读用C语言编写的程序是在能有效地学习UNIX源代码之前必须克服的障碍之一。与自然语言一样，阅读比写作更容易获得。即便如此，读者仍需要小心以免一些更微妙的地方与你擦肩而过。有两个“UNIX文档”直接与C语言相关：

Dennis Ritchie撰写的C语言参考手册(C Reference Manual)

Brian Kernighan撰写的C语言编程—一本辅助教材(Programming in C – A Tutorial)

你应该现在开始尽可能早地阅读它们，然后不断地重新阅读它们并逐渐理解它们。学习编写C语言程序不是必需的。如果你有机会，那么你应尝试编写至少一些小程序，这确实代表学习编程语言的可接受方式以及你对正确使用以下项(items)的理解。

分号;
“=”和“==”
“{” and “}”
“++”和“--”声明;
Register variables寄存器变量;
“if”和“for”语句

你会发现C语言是一种很方便的语言，它用于访问和操作数据结构和字符串，这是操作系统的主要部分。为适应面向终端的语言，这需要简洁和紧凑的表达，C语言使用一种大字符集并使许多符号如“*”和“＆”频繁地工作。在这方面，C语言能与APL比肩。C语言的许多特性与PL/1相似，但在为结构化编程提供的设施范围内，它远远超出后者。

3.1 某些选定的例子

以下示例直接取自UNIX源代码。

3.2 例一

最简单的过程并不做实际工作，在源代码中出现两次(!)，它们是“nullsys”(2864)和“nulldev”(6577)，原文如此(sic)。

第6577行起始

```
nulldev（）
{
}
```

虽然没有参数，但仍然需要括号“(”和“)”。一对花括号“{”和“}”分隔过程体，它在本例中为空。

3.3 例二

下一个例子有点不那么简单：

第6566行起始

```
nodev()
{
    u.u_error = ENODEV;
}
```

附加语句是一个赋值语句，它以分号结束，分号是语句的一部分，而不是类似Algol语言的语句分隔符。“ENODEV”是定义的符号，即在实际编译前由编译器中的预处理器替换为关联字符串的符号，“ENODEV”在第0484行定义为19。UNIX惯例是被定义的符号是大写形式，而所有其它符号是小写形式。

“=”是赋值运算符，并且“u.u_error”是结构“u”的同一个元素，参见第0419行。请注意点运算符“.”是选择结构元素的运算符，该元素名称是“u_error”，它可作为在UNIX源代码中构造结构元素名称方式的范例：开头是一个区分字母，接着是一个下划线，最后跟这一个名称。

3.4 例三

例三的过程函数非常简单，它把指定数量的单词从一组连续位置复制到另一组，它有三个参数(parameters)，代码参见如下。

第6585行起始

```
bcopy (from, to, count)
int *from, *to;
{
    register *a, *b, c;
    a = from;
    b = to;
    c = count;
    do
        *b++ = *a++;
    while (--cc);
}
```

在以上代码中，我们注意到其中第二行指定前两个变量是指向整数的指针，由于没有为第三个参数提供规范，因此默认情况下假定它是一个整数。

```
int * from，* to;
```

三个局部变量a、b和c已分配给寄存器，因为寄存器更易于访问且引用它们的目标代码更短。“a”和“b”是指向整型的指针，“c”是整型，寄存器声明也能以如下方式表示，这种方式强调寄存器与整型的联系。

```
    register int * a, * b, c;
```

应该仔细研究以“do”开头的三行。如果“b”是“指向整型的指针”，那么类型*b表示指向的整型。因此，为把“a”指向的值复制到“b”指定的位置，我们可写成如下形式：

```
    *b = *a;
```

如果我们用以下形式取代，那么这将使“b”的值与“a”的值相同，即“b”和“a”将指向相同位置，至少在这里，这不是必需的。

```
b = a;
```

把第一个单词从源复制到目标后，我们需要增加“b”和“a”的值以便指向各自集合的下一个单词，这可通过方式完成：

```
b = b+1; a = a+1;
```

但C语言提供更短的符号(当变量名称更长时更有用，即表示成如下形式：

```
b++; a++;
```

现在这里的语句“b++;”和“++b;”之间并没有区别。然而，“b++”和“++b”可用于表达式中的术语，在此情况下它们是不同的。在这两种情况下，保持递增“b”的效果，但是进入表达式的值是“b ++”的初始值和“++b”的最终值。

“−−”运算符遵循与“++”运算符相同的规则，但它每次递减1，因此，“−−c”把表达式作为递减后的值输入。

“++”和“−−”运算符很有用且在贯穿在整个UNIX源代码中。偶尔，你不得不回到第一原则弄清楚它们的用途；另请注意，*b++和(*b)++之间存在差异。

这些运算符适用于指向结构的指针以及指向简单数据类型的指针，当一个已申明引用一个特定结构类型的指针递增时，该指针的实际值增加结构的大小。

我们现在能看到以下行的含义， 它先复制该字，然后指针增一，一次性命中。

```
*b++ = *a++;
```

以上代码中倒数第二行代码

```
while (−c);
```

界定在“do”之后开始的语句集的结0，则重复循环，否则终止。

显然，如果“count”的初始值为负，那么该循环不会正确终止；如果这个可能性很高，那么我们必须修改例程。

3.5. 例四

参数“f”是假定的整数，它被直接复制到寄存器变量“rf”中；这种模式(pattern)变得如此熟悉，因此我们将不再对此进行评论。

第6619行起始

```
getf(f)
{
    register *fp, rf;
    rf = f;
    if (rf < 0 || rf >= NOFILE)
        goto bad;
    fp = u.u_ofile[rf];
```

```
        if (fp != NULL)
        return (fp);
bad:
    u.u_error = EBADF;
     return (NULL);
    }
```

请参见以下三个简单的关系表达式，若为true，则每个值都为1；若为假，则其值为0。第1个测试“rf”的值是否小于0；第2个测试值，若“rf”大于“NOFILE”定义的值；第3个测试值，若“fp”的值不等于“NULL” (定义为0)。

```
    rf < 0    rf >=NOFILE     fp = NULL
```

“if”语句测试的条件是包含在括号中的算术表达式。

若表达式大于零，则测试成功并执行以下语句，例如，如果“fp”具有值001375，那么以下成真

```
    fp！= NULL
```

并且作为算术表达式中的术语，赋予值1，该值大于零，因此执行以下语句

```
    return(fp);
```

从而终止“getf”的执行，并把“fp”的结果作为“getf”的值返回给调用过程。

表达式

```
    rf < 0 || rf >= NOFILE
```

这是两个简单关系表达式的逻辑或“||”。

“goto”语句和相关标签的示例将在后续说明。

“fp”被赋予一个值，该值是距整型数组“u_ofile”的“rf”-th元素的地址，它嵌入在结构“u”中。

过程“getf”向其调用过程返回一个值，这是“fp”(地址)或“NULL”值。

3.6. 例五

该示例与前一个示例之间存在许多相似之处。然而，我们有一个新的概念、一系列结构。令人困惑的是该例的数组和结构都被称为“proc”，然而C语言允许这样做。

第2113行起始

```
wakeup(chan)
{
    register struct proc *p;
    register c, i;

    c = chan;
    p = &proc[0];
    i = NPROC;
```

```
    do {
        if(p->p_wchan == c) {
            setrun(p);
        }
        p++;
    } while(--i);
}
```

它们在表03中以以下形式语句：

第0358行起始

```
struct proc
{
    char p_stat;
    .....
    int p_wchan;
    .....
    proc [NPROC];
```

“P”是指向“proc”类型结构的类型指针的寄存器变量。

```
    p =proc[0];
```

它把“p”分配给数组“proc”的第一个元素的地址，在此上下文中，运算符“＆”表示“地址”。

请注意：如果数组有n个元素，那么该数组元素的下标为0,1，..，(n-1)；同样，我们能把上述陈述更简单地写为

```
    p = proc;
```

“Do”和“while”之间有两个语句，其中第一个能更简单地重写为如下形式：

```
    if (p->p wchan == c) setrun (p);
```

亦即省略原来的花括号，C语言是一种自由形式的语言，因此行之间的文本排列并不重要。

下列语句：

```
    setrun(p);
```

调用过程“setrun”，把“p”的值作为参数传递，所有参数都按值传递。关系

```
    p->p_wchan == c
```

测试“c”的值的相等性和“p”指向的结构的元素“p_wchan”的值。请注意这个写入是错误的

```
    p.p_wchan == c
```

因为“p”不是结构的名称。

第二个语句不能与第一个语句相结合，将“p”增加“proc”结构的大小，无论是什么(编译器能搞清楚)。

为正确地进行计算，编译器需要知道指向的结构类型。如果这不是考虑因素，你会注意到，在类似情况下，“p”将简单地声明为

```
register *p;
```

因为它对程序员来说更容易，编译器也不坚持。

该过程的后半部分可能是等效的，但效率较低

```
.....
i = 0;
do
    if (proc[i].p_wchan == c)
        setrun (&proc[i]);
while (++i < NPROC);
```

3.7. 例六

这个过程只检查是否存在错误，如果未设置错误指示“uu_error”，那么把它设置为通用出错指示(“EIO”), EIO在MAN中解释为Input/Output Error, 实际上应为Error Input/Output; 本示例代码参见如：

第5336行起始：

```
geterror(abp)
struct buf *abp;
{
    register struct buf *bp;

    bp = abp;
    if (bp->b_flags&B_ERROR)
        if ((u.u_error = bp->b_error)==0)
            u.u_error = EIO;
}
```

该过程仅检查是否存在错误，如果未设置错误指示“uu_error”，那么把它设置为一般错误指示“B_ERROR”且其值为04(参见第4575行)。这样仅使用一位(bit)设置，它能用于掩码隔离比特数2(to isolare bit number 2)。

运算符“＆”用在以下表达式中，它是应用于算术值的按位“逻辑与”

```
bp-> b_flags＆B_ERROR
```

如果设置由“bp”指向的“buf”结构的元素“b_flags”的第2位，那么上述表达式大于1。因此，如果出现错误，那么如下表达式进行求值并与0进行比较；

```
（u.u_error）= bp-> b_error）
```

现在这个表达包括一个赋值运算符“=”；表达式的值是在为“bp-> b_flags”赋值之后的“u.u_error”的值。赋值用于表达式的一部分是有用且很常见的。

3.8. 例七

在该示例中，你应注意程序“suser”返回一个用于“if”测试的值，执行取决于此值的三个语，句括在括号“{”and“}”中。

第3428起始

```
stime()
{

    if (suser()) {
        time[0] = u.u_ar0[R0];
        time[1] = u.u_ar0[R1];
        wakeup(tout);
    }
}
```

请注意：对没有参数的过程调用仍然必须用一组空的括号写成。

```
    suser()
```

3.9. 例八

C语言提供条件表达式，因此，如果“a”和“b”是整型变量，那么如下表示就是一个表达式，其值是“a”和“b”中的较大值。

```
    (a> b? a：b)
```

若把“a”和“b”视为无符号整型，则不起作用；因此，应使用以下过程

第6326行起始

```
max(a, b)
char *a, *b;
{

    if (a > b)
        return(a);
    return(b);
}
```

这里的诀窍是声明为字符指针的“a”和“b”被视为无符号整数的比较目的，该程序的函数体能写成如下形式：

```
max(a, b)
char *a, *b;
{
    if (a > b)
        return(a);
    else
        return(b);
}
```

但是，由于“return”本身的的性质，因此这里无须用“else”语句！

3.10. 例九

这里有两个短的过程，它们包含一些不同和陌生的表达式。

请参见第一个例子：

第7679行起始

```
schar()
{

    return(*u.u_dirp++ & 0377);
}
```

请参见以下声明

```
    char *u_dirp;
```

它是“u”结构声明的一部分，“u.u_dirp++”是一个字符指针，因此“*u.u_dirp++”的值是一个字符。当指针增一时，就发生副作用。当字符加载到16位寄存器时，可能发生符号扩展。通过把它与“0377“作按位与计算，我们能消除任何无关的高阶位，因此仅返回一个简单字符。请注意任何以零开头的整数(例如0377)都被解释为八进制整型。

请参见第二个例子：

第1771行

```
nseg(n)
{
    return((n+127)>>7);
}
```

它的返回值是n除以128并向上舍入到下一个最高的“整型”；请注意使用右移位运算符“>>”而不是除法运算符“/”。

3.11. 例十

上面介绍的很多要点都集中在下一个例子中：

第2134行

```
setrun(p)
{
    register struct proc *rp;

    rp = p;
    rp->p_wchan = 0;
    rp->p_stat = SRUN;
    if(rp->p_pri < curpri)
        runrun++;
    if(runout != 0 && (rp->p_flag&SLOAD) == 0) {
        runout = 0;
        wakeup(&runout);
```

```
    }
}
```

通过弄清楚这一过程做了什么，检查你对C语言的理解。你可能需要了解两个附加功能：“&&”是相关表达式的逻辑连接(“and”)，请参见前面介绍的“||”。

请参见最后一个语句，它包含表达式

```
&runout
```

这在语法上是一个地址变量，然而在语义上是一个独特的位模式。这在UNIX中是一个很普遍的实例方法。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. 例十一

请参见本例的代码，这里有一个二元运算符的典型实例。

第4856行起始

```
bawrite(bp)
struct buf *bp;
{
    register struct buf *rbp;

    rbp = bp;
    rbp->b_flags =| B_ASYNC;
    bwrite(rbp);
}
```

该代码片段倒数第二个语句很有意思，因为它能写成如下形式。

```
    rbp->b_flags = rbp->b_flags | B_ASYNC;
```

在该语句中，位掩码“B_ASYNC”被逻辑与编入“rbp->b_flags”；按位或符号“|”是算术值的逻辑分离。这是UNIX中很有用的构造的一个例子，它能为程序员节省大量人力；如果“O”是任何二元运算符，那么有以下语句。

```
x = x O a;
```

其中“a”是表达式，它能更简洁地重写为

```
x = O a;
```

在使用这种结构时，程序员必须注意空白字符的放置，因为

```
x =+ 1;
```

不同于

```
x = +1;
```

请注意以下表示是什么意思?

x =+ 1; ?

3.13. 例十二

此示例引入“for”语句，这是一个很通用的语法结构，“for”语句既强大又紧凑。C语言中的“for”语句的强大功能源于程序员在选择括号之间包含的内容时所拥有的巨大自由。当然，没有什么能把计算限制为整数。其语法结构在C Tutorial的第10页中有详细描述，此处不再重复描述。

第6824行起始

```
ufalloc()
{
    register i;

    for (i=0; i<NOFILE; i++)
        if (u.u_ofile[i] == NULL) {
            u.u_ar0[R0] = i;
            return(i);
        }
    u.u_error = EMFILE;
    return(-1);
}
```

上述“for”语句的Algol程序的等价物是

```
    for i:=1 step 1 until NOFILE-1 do
```

3.14. 例十三

在本例的“for”语句中，指针变量“p”依次遍历数组“proc”的每个元素。

第3949行起始

```
signal(tp, sig)
{
    register struct proc *p;

    for(p = &proc[0]; p < &proc[NPROC]; p++)
        if(p->p_ttyp == tp)
            psignal(p, sig);
}
```

请参见以下”for”语句的原始代码：

```
   for (p=&proc[0];p<&proc[NPROC];p++)
```

但它超过本书UNIX源代码页中的长度！如前所述，在此上下文中使用“proc”作为表达式“＆proc[0]”的替代方法是能接受的。

这种“for”语句在UNIX中几乎随处可见，因此你最好学会识别它，它读为以下伪代码

```
    for p = each process in turn
```

请注意“proc[NPROC]”是数组的(NPROC+1)元素的地址(当然不存在)，即它是数组末端的第一个位置。我们会再次指出，在此前的例子中

```
    I++;
```

意味着整型“i”加1，而在本例中

```
    p++;
```

这表示把p移动指向下一个结构。

3.15. 例十四

这是“while”语句的一个示例，应该与之前遇到的“do..while…”构造进行比较，参见Pascal的“while”和“repeat”语句。

第8870行起始

```
lpwrite()
{
    register int c;

    while ((c=cpass())>=0)
        lpcanon(c);
}
```

该程序的含义是在结果为正时继续调用“cpass”，并把结果作为参数调用给lpcanon。请注意C语言语句中使用“int”。虽然这并不必要，但是它并不总是被省略！

3.16. 例十五

本例是从此前的代码中简化而来的

第5861行起始

```
seek()
{
    int n[2];
    register *fp, t;

    fp = getf(u.u_ar0[R0]);
    .....
    t = u.u_arg[1];
    .....
    switch (t) {

    case 1:
    case 4:
        n[0] =+ fp->f_offset[0];
        dpadd(n, fp->f_offset[1]);
        break;

    default:
        n[0] =+ fp->f_inode->i_size0&0377;
```

```
        dpadd(n, fp->f_inode->i_size1);

    case 0:
    case 3:
        ;
    }
    .....
}
```

请注意两个字的数组“n”的数组语句以及getf的使用(如例4所示)，原文漏掉了第二段省略号。

“Switch”语句根据表达式中的表达式值创建一个多路分支，各个部分都有“案例标签”：

• 若“t”为1或4，则按顺序执行一组操作。
• 若“t”为零或三，则根本不做任何事情。
• 若“t”是其它任何内容，则执行标记为“default”的一组操作。

注意在“switch”语句结束后使用“break”作为对下一个语句的转义。若没有“break”，则在“switch”语句中遵循正常的执行顺序。因此，在“default”动作结束时通常需要“break”。它在这里被安全地省略，因为唯一剩下的案例实际上具有与它们相关联的空行为。

两个非平凡的动作对表示向另一个添加一个32位整数，更高版本的C编译器将支持“长”变量，并使这种代码更容易编写和读取；还要注意表达式中的内容。

```
    FP-> f_inode-> i_size0
```

该语句进行了两级间接操作。

3.17. 例十六

第6672行起始

```
closei(ip, rw)
int *ip;
{
    register *rip;
    register dev, maj;

    rip = ip;
    dev = rip->i_addr[0];
    maj = rip->i_addr[0].d_major;
    if (rip->i_count <= 1)
    switch (rip->i_mode&IFMT) {

    case IFCHR:
        (*cdevsw[maj].d_close)(dev, rw);
        break;

    case IFBLK:
        (*bdevsw[maj].d_close)(dev, rw);
    }
    iput(rip);
}
```

这个例子有许多有趣的功能。

d_major的申明是在struct结构中

```
struct {
       char d_minor;
       char d_major;
}
```

这使得赋给“maj”的值是赋给“dev”的值的hiqh顺序字节。

在本例中，“switch”语句有两个非空(null)的情况，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，应注意这是一个程序调用，其参数是“dev”和“rw”；其次，“bdevsw”(和“cdevsw”)是数组结构，它的“d_close”元素是指向函数的指针

```
     bdevsw[maj]
```

这是结构名，而以下语句

```
     bdevsw[maj].d_close
```

这是该结构的一个元素，恰好是一个指向函数的指针，因此：

```
    *bdevsw[maj].d_close
```

这是一个函数的名称，第一对花括号是“语法糖”，它使编译器处于正确的思维框架！

3.18. 例十七

我们提供以下示例，结束本章的代码评注。

第4043行

```
psig()
{
    register n, p;
    .........

    switch(n) {

    case SIGQIT:
    case SIGINS:
    case SIGTRC:
    case SIGIOT:
    case SIGEMT:
    case SIGFPT:
    case SIGBUS:
    case SIGSEG:
    case SIGSYS:
```

```
        u.u_arg[0] = n;
        if(core())
            n =+ 0200;
    }
    u.u_arg[0] = (u.u_ar0[R0]<<8) | n;
    exit();
}
```

这里“switch”语句选择“n”的某些值，应该对其执行一组动作；另一种方法是写一个很长的“if”语句，例如

```
    if (n==SIGQUIT || n==SIGINS || ...
            ... || n==SIGSYS)
```

这既不清晰，又不高效。请注意向“n”添加八进制常量以及从两个八位值组成16位值的方法。

4. 概述

本章的目的是从整体上研究UNIX源代码，即见林又见树(见树又见林之反义词)。

对源代码的检查将发现它包含大约44个不同的文件，其中主要文件包括如下：

- 两个是汇编语言，它们的名称以“s”结尾;
- 28个是C语言，名称以“c”结尾;
- 14个属于C语言，但不适用于独立编译，名称以“h”结尾。

这些文件及其内容由程序员安排，这大概是为方便程序员的设计开发而不是便利阅读者；在许多方面，文件之间的划分与本文目前讨论的内容无关，可能完全被废除。

如第一章所述，文件分为五个部分，我们尽可能选择大小大致相同的部分，这样就能对强关联的文件进行聚合(cluster)，并分离弱关联的文件。

4.1. 变量分配

PDP11系统架构允许开发者有效访问(access)绝对地址已知的变量或在编译时准确确定这些变量相对于栈指针的地址。它对于变量声明的多个语言级别没有硬件支持，例如Algol或Pascal等块结构化语言。因此，在PDP11上实现的C语言仅支持两个词法级别：全局变量(global variables)和局部变量(local variables)。全局变量是静态分配的，而局部变量在当前栈区或通用寄存器中动态分配的(r2，r3和r4以这种方式使用)。

4.2. 全局变量

UNIX的全局变量的声明绝大部分都被集中在“h”文件集中，除极少数例子以外，这些例外情况是：

a）静态变量“p”(2180)在“swtch”中声明；虽然它以全局方式存储，但是只能从“swtch”过程中访问，实际上“p”是UNIX中局部变量的一个很流行的名称;

b）许多变量如“swbuf”(4721)仅由单个文件中的过程引用并在该文件的开始声明。

全局变量可在引用它们的每个文件中单独声明；然就是加载器(loader)的工作，加载器把程序文件的编译版本链接在一起，以匹配同一变量的不同声明。

4.3. C预处理器

如果开发者必须在每个文件中完整地重复申明全局变量(正如Fortran所要求的那样)，那么程序量就会大幅增加，修改声明充其量只是一种麻烦，而且在最坏的情况下容易出错。UNIX使用C语言编译器的预处理器，它允许大多数全局变量声明仅在少数“h”文件中的每一个中记录一次，因此它能避免以上错误。每当开发者需要声明特定的全局变量时，相应的“h”文件就能“包含”在正在编译的文件中。

UNIX还使用“h”文件作为许多符号名称的标准定义列表的工具，这些符号名称表示常量、可调参数以及用于声明某些结构类型。如果文件bottle.c包含一个过程“glug”，这个名为“gin”的全局变量在文件“box.h”中声明(declaration)，那么以下语句(statement)：

#include “box.h”

必须插入到文件“bottle.c”的开头。当编译文件“bottle.c”时，编译“box.h”中的所有声明，因为它们是在在“bottle.c”中任一过程之前发现的，因此它们在所产生的可重定位模块中被标记为外部的。

当所有对象模块链接在一起时，将在每个文件中找到对“gin”的引用，其中源包含“box.h”；所有这些引用将是一致的，并且加载器将为“gin”分配一个地址空间并相应地调整所有对”gin”的引用。

4.4. 第一节

第一部分包含许多“h”文件和汇编语言文件，它还包含许多与系统初始化和进程管理有关的文件。

4.5. 第一组“.h”文件

Param.h [表01] 不包含变量声明，但包含许多操作系统常量和参数的定义以及3个简单结构的声明。对于定义的常量，请注意使用“仅大写”的约定。

Systm.h [表02; 第19章] 完全由声明组成(结构“callout”和“mount”的定义为副作用)；请注意没有一个变量是显式初始化的，因此所有变量都初始化为0。前三个数组的维度是param.h中定义的参数，因此任何“包含”“systm.h”的文件必须先包含“param.h”。

Seg.h [表03] 包含一些定义和一个声明，用于引用段寄存器，这个文件能并入到“param.h”和“systm.h”，而不会有任何实际损失。

Proc.h [表03;第7章] 包含“proc”的重要声明，它既是一个结构类型，又是一个结构数组。“proc”结构的每一个元素都有一个以“p”开头的名称，并且没有其它变量如此命名，类似的惯例用于命名其它结构的元素；它的前两个元素“p_stat”和“p_flag”的值集具有定义的单个名称。

user.h [表04;第7章] 包含非常重要的“user”结构的声明以及“u_error”的一组定义值；一次只能访问“user”结构的一个实例；这在名称“u”下引用，它位于1024字节区域的低地址部分，称为“每个进程数据区”。

一般而言，本文后面不会详细分析完整的“h”文件，预计读者会不时地参考它们，并会越来越熟悉和理解它们。

4.6. 汇编语言文件

汇编语言有两个文件，大约占源代码的10%，因此开发者必须熟悉这些文件。

Low.s [表05;第9章] 包含用于初始化主存的低地址部分的信息，包括陷阱向量；该文件由名为“mkconf”的实用程序生成，以适应特定安装中存在的外围设备集。

M40.s [表06..14;第6,8,9,10,22章] 包含一组适用于PDP11/40的例程，它们执行以大量C语言不能实现的各种特殊功能，我们将在合适的时候对该文件进行讨论。最大的汇编例程“backup”留给读者研究和练习。当使用PDP11/45和PDP11/70时，应把“m40.s”代换为“m45.s”，但是这里没有提供此文件。

4.7. 第一节中的其它文件

main.c [表15..17;第6,7章] 包含“main”，它执行各种初始化任务以便UNIX能运行；它还包含“sureg”和“estabur”，用于设置用户态(User Mode)的段寄存器。

slp.c [表18..22;第6,7,8,14章] 包含进程管理所需的主要程序，该文件括“newproc”， “sched”，“sleep”和“swtch”。

Prf.c [表23,24;第5章] 包含“panic”和许多其它程序，它们提供一种简单的机制，向操作人员显示初始化消息和错误消息。

malloc.c [表25;第5章] 包含“malloc”和“mfree”，它们的作用是管理内存资源。

4.8. 第二节

第二部分涉及陷阱(trap)、硬件中断(hardware interrupts)、软件中断(software interrupts)。

陷阱和硬件中断造成CPU的正常指令执行序列(顺序)的突然切换，它们提供一种处理在CPU控制外之发生的特殊条件的机制。

Use是“系统调用”机制的一部分，其中一个用户程序执行“陷阱”指令以故意引发陷阱，从而获得操作系统的注意和帮助。软件中断(或称为“信号”)是IPC进程间通信的一个机制，特别是当存在“坏消息”时使用。

reg.h [表26;第10章]定义一组常量，它用于在存储在内核栈时引用先前的用户态(User Mode)寄存器值。

trap.c [表26..28;第12章] 包含“C”程序“陷阱”，它能识别和处理各种陷阱。

sysent.c [表29;第12章] 包含数组“sysent”的声明和初始化，“trap”使用它把适当的内核模式例程与每个系统调用类型相关联。

sysl.c [表30..33;第12和13章] 它是与系统调用(system call)相关的各种例程，它包括“exec”、“exit”、“wait”和“fork”。

sys4.c [表34..36; 第12,13和19章] 包含“unlink”、“kill”等与多个次要的系统调用相关的里程。

clock.c [表37,38;第11章] 包含“clock”例程，它主要处理时钟中断，执行大量的偶发事务管理(housekeeping)和基本统计(basic accounting)。

sig.c [表39..42;第13章]包含处理“信号”或“软件中断”的程序，这些程序为进程间通信(IPC)和跟踪(tracing)提供便利。

4.9. 第三节

第三部分涉及内存(主存储器)和磁盘(磁盘存储器)之间的基本输入/输出操作，这些操作是程序交换活动以及磁盘文件创建和引用的基础；本节还诠释使用和操作大型(512字节)缓冲区的过程。

text.h [表43;第14章] 定义“文本”结构和数组，一个“文本”结构用于定义共享文本段的状态。

text.c [表43,44;第14章] 包含管理共享文本段的过程。

buf.h [表45;第15章] 定义“buf”结构和数组、结构“devtab”以及“b_error”值的名称，所有这些都是管理大(512字节)缓冲区所必须的。

conf.h [表46;第15章] 定义结构数组“bdevsw”和“cdevsw”，它们指定执行逻辑文件操作所需的面向设备的过程。

conf.c [表46;第15章] 如同“low.s”一样，它是由实用工具“mkconf”生成，适应特定安装中存在的外围设备集；它包含控制对基本I/O操作的数组“bdevsw”和“cdevsw”进行初始化。

bio.c [表47..53;第15,16,17章]是“m40.s”之后最大的一个文件，它包含操作大缓冲区和基本块面向I/O的过程。

rk.c [表53,54;第16章] 它是RK11/K05磁盘控制器的设备驱动程序。

4.10. 第四节

第四节主要涉及文件和文件系统。一个文件系统由一组文件、相关的表和目录组成，它们都存放在单个存储设备如磁盘组(a disk pack)；本节主要包括创建过程和访问文件、通过组织并维护文件系统的目录查找文件；它还包括一个新颖的“pipe”类型文件的代码。

file.h [表55;第18章] 定义“文件”结构和数组。

filsys.h [表55;第20章] 定义“filsys”结构，它被复制到“已安装”文件系统上的“超级块”中。

lno.h [表56] 描述“安装”设备上记录的“inode”的结构，由于此文件未包含在任何其它文件中，因此它仅供参考。

inode.h [表 56;第18章] 定义“inode”结构和数组，其中“inode”对于管理文件进程的访问是至关重要。

sys2.c [表57..59;第18,19章] 包含一系列与系统调用相关的例程，包括“read”，“write”，“create”，“open”和“close”。

sys3.c [表60,61;第19,20章] 包含一组与各种次要系统调用相关的例程。

rdwri.c [表62,63;第18章] 包含涉及读写文件的中间级别例程。

subr.c [Sheets 64,65;第18章] 包含更多用于I/O的中间级例程，尤其是“bmap”，它把逻辑文件指针转换为物理磁盘地址。

fio.c [表66 ... 6;第18,19章] 包含用于文件打开、关闭和访问控制的中间级例程。

alloc.c [表69..72;第20章] 包含管理“inode”数组和磁盘存储块中条目分配的过程。

iget.c [表72..74;第18,第19,20章] 包含涉及引用和更新“inode”的程序。

nami.c [表75,76;第19章] 包含搜索文件目录的“namei”程序。

pipe.c [表77,78;第21章] 它是“pipe”的设备驱动程序，它是一种特殊形式的短磁盘文件，用于把信息从一个进程传输到另一个进程。

4.11. 第五节

第五部分是最后一部分，它关注低速面向字符的外设的输入/输出；这些外设共享一个公共缓冲池，该缓冲池由一组标准过程操纵；该组面向字符的外设通过以下方式表现出来：

KL/DL11 交互式终端
PC11 纸带读卡器/打孔机
LP11 行式打印机

tty.h [表79;第23,24章] 定义“clist”结构(用于字符缓冲区队列的列表头)，“tty”结构(存储用于控制单个终端的相关数据)，声明“partab”表(用于控制单个字符到终端的传输)，并定义许多相关参数的名称。

kl.c [表80;第24,25章] 它是终端的设备驱动程序通过KL11或DL11接口连接。

tty.c [表单81..85;第23,24,25章] 包含与附加接口无关的通用程序，用于控制到终端的传输或从终端传输，并考虑到各种终端的特性。

pc.c [表86,87;第22章] 它是PC11纸带读取器/打孔控制器的设备处理器。

lp.c [Sheets 88,89;第22章] 它是LP11行打印机控制器的设备处理程序。

mem.c [Sheet 90] 包含提供对主存的访问过程，就像它是普通文件一样，这段代码留给读者作为练习进行研究。

第一部分、初始化、进程、进程管理

5. 两个文件

本章旨在通过查看第一部分中的两个文件简单地介绍源代码，这两个文件能与其它文件相当好地进行隔离；对这些文件的讨论补充第三章的讨论，并包含一些关于C语言的语法和语义的附加注释。

5.1 文件'malloc.c'

此文件位于源代码的表25中，它包括存储资源的分配和释放，请参见以下实例代码，主存以32字(64字节)为单位，而磁盘交换区单位为256字(512字节)。

malloc(2528) mfree(2556)

对于这两种资源中的每一类，在资源“地图”(“coremap”或“swapmap”)记录可用区列表；指向适当资源“map”指针始终传递给“malloc”和“mfree”，这样例程本身不必知道它们正在处理的资源类型。

“coremap”和“swapmap”中的每一个是在第2515行声明的“map”类型的结构数组，该结构由两个字符指针组成，即两个无符号整数。

“coremap”和“swapmap”的声明在第0203、0204行。这里“map”结构被完全忽略—这是一种令人遗憾的编程快捷方式，因为加载程序没有检测到它。因此“coremap”和“swapmap”中的列表元素的实际数量分别是“CMAPSIZ / 2”和“SMAPSIZ / 2”。

5.2. 列表维护规则

（a）每个可用区域的大小和相对地址(按照与资源相应的单位计算);

（b）每个清单的要素始终按相对地址的增加顺序排列；请注意没有两个列表元素代表连续区 - 把两个区合并为一个更大区域的替代方案总是被采用;

（c）能通过查看数组的连续元素扫描整个列表，从第一个元素开始，直到遇到具有0大小的元素。最后一个元素是“sentinel”，它不是列表中的一部分。

上述规则提供对“mfree”的完整规范以及“malloc”的规范，除一个方面外，它是完整的：我们需要指定当存在多种执行方式时资源分配是如何实际完成的。

“malloc”中采用的方法是一种称为“First Fit”的方法，原因应该是明显的。

作为如何维护资源“map”的说明，假设以下三个资源区是可用的：
- 一个可用区的长度为15，起止范围是[47, 61);
- 一个可用区的长度为13，起止范围是[27, 39];
- 一个可用区的长度是7，在位置65处开始[65:]

然后“map”将包含：

条目	大小	地址
0	13	27
1	15	47
2	7	65

3	0	?
4	?	?

如果收到一个长度为7的存储空间的请求，那么从位置27开始分配，分配后的map将变为：

条目	大小	地址
0	6	34
1	15	47
2	7	65
3	0	?
4	?	?

如果从地址40开始到46的存储区返回到可用列表，则“map”将变为：

条目	大小	地址
0	28	34
1	7	65
2	0	?
3	?	?

请注意：虽然总可用资源确实增加，但是由于合并、元素的数量实际上减少一个。现在让我们转向考虑实际的源代码。

5.3 malloc(2528)

此过程的主体包含一个“for”循环，它搜索“map”数组，直到达到以下两种情况：

（a）或者能到到可用资源清单的结尾;
（b）或者找到能满足当前请求的区域;

2534：“for”语句初始化“bp”以指向资源映射的第一个元素；在后续的每次迭代中，“bp”递增1以指向下一个“map”结构。

请注意以下延续条件：

“bp->m_size”是一个表达式，当引用哨兵(sentinel)时，该表达式变为0；该表达式也能等价地编译为一个更明显的表达式“bp->m_size>0”。

请注意没有对数组结尾进行显式测试(应能证明：假定CMAPSIZ，SMAPSIZ≥2*NPROC，那么不需要后者！）

2535：如果list元素定义的区域至少与请求的区域相等，那么.....

2536：记住该区域第一个单元的地址;

2537：递增存储在数组元素中的地址;

2538：减少存储的元素并把结果与0进行比较，即检查它是否完全适配;

2539：在精确适配的情况下，把所有剩余的列表元素(直到并包括哨兵(sentinel))向下移一个位置。

注意(“bp-l)”指向“bp”引用结构之前的结构;

2542：“while”延续条件不测试“(bp-l)->m_size”和bm->m_size的相等性！测试的值是分配给“(bp->m_size”从“bp->m_size”复制的值)，你不能立即意识到这一点;

2543：返回该区域的地址，return语句代表程序结束，因此非常肯定“for”循环的结束。请注意返回0值意味着“没有运气”，其假设是没有一个有效区域能从零位开始。

5.4 mfree(2556)

该过程把地址“aa”处的大小“size”的区域返回到由“mp”指定的“资源映射”；该过程的程序体由一行“for”语句和此后的多行“if”语句组成。

2564：该行末尾的分号极为重要的，它终止与一个空语句；正如第2394行所述的那样，如果我们把分号设置为单独占一行，那么易读性就更好，如第2394行所述。这种语句表明C语言的能力或不足。请尝试用另一种语言如Pascal或PL/1编译与此语句等价的语句。

在列表中单步执行“bp”，直到遇到一个元素，其地址大于返回区域的地址。
即不是“bp->maddr≤a”
或者表示列表的结尾，即不是“bp-> m size！= 0”;

2565：我们现在已找到应插入新列表元素的元素；这里的问题是：列表是否会增加一个元素或合并是否会保持元素数量相同甚至减少一个?

如果“bp> mp”，那么我们不会试图在列表开头插入。
如果(bp-l)->m_addr + (bp-l)->m_size == a， 那么返回的区域紧靠列表中的前一个元素;

2566：通过返回区域的大小增加前一个列表元素的大小;

2567：返回的区域是否也与列表的下一个元素相邻？如果是这这样……

2568：把列表的下一个元素的大小添加到前一个元素的长度(size);

2569：把所有剩余的列表元素(最多为包含0的元素)向下移动一个位置；请注意：如果第2567行的测试偶然给出“bp->m_size”为0时偶然给出值为真的情况，那么不会造成任何损害;

2576：如果第2565行的测试失败，那么返回此语句-即返回的区域不能与列表中的前一个元素合并，能与下一个元素合并吗，注意检查下一个元素是否为空;

2579：如果返回区域通常为非空的(可能应更早地进行此测试)，那么就把新元素添加到列表中并把所有剩余元素推送到一个位置。

5.5. 结论

这两个过程(procedure)的代码编写得非常紧密。虽然我们几乎不无法删除过程中的冗余以提高运行的时间效率，但是我们能用更清晰的方式编译这两个过程。如果你有同样强烈的感觉，那么请你练习重写“mfree”以便其功能更容易辨别。请注意“malloc”和“mfree”的正确运行取决于“coremap”和“swapmap”的正确初始化。执行此操作的代码发生在第1568行，第1583行的“main”过程中。

5.6文件’prf.c'

该文件位于表23和24中，它包含以下过程：

printf（2340）　printn（2369）
putchar（2386）　panic（2416）
prdev（2433）　deverror（2447）

这些程序之间的调用关系如下所示：

(panic || deverror) -> prdev) -> printf —> printn —> putchar

5.7 printf（2340）

过程“printf”为UNIX操作系统提供一种向系统控制台终端发送消息的方法；它是一种直接、无须操作及不带无缓存的方法；它在初始化期间用于报告硬件错误或即将发生的系统崩溃。这些版本的“printf”和“putchar”在内核态(Kernel Mode)下运行，尽管它们与用户态(User Mode)下运行的C程序调用版本类似，但两者是不同的。后一个版本的“printf”和“putchar“位于库“/lib/libc.a”。这时阅读UPM(UNIX Programmer’s Manual)的”PRINTF(III)“和“PUTCHAR(III)”是有意义的。

2340：在为此过程申明全部参数时，这个程序员的脑袋一定是开了小差；实际上，该过程体仅包含对“x1”和“fmt”的引用。这有助于揭示C语言编程的一个规则：它不强制执行过程调用和过程声明之间的参数匹配，甚至不考虑参数的数目；除此而外，这些参数是以逆序序放在栈上；当“printf”被调用时，“fmt”将比“x1”更接近栈顶。

---------	x2	---------	xl	---------	fmt	---------

“x1”的地址高于“fmt”，但是低于“x2”，因为栈在PDP11中是向下增长的。

2341：“fmt”解释为常量字符指针，该声明几乎等价于以下语句，其不同之处在于“fmt”的值不能改变;

“char * fmt;”

2346：把“adx”设置为指向“x1”，表达式“＆xl”是“xl”的地址。请注意：由于“x1”是栈位置，因此无法在编译时计算此表达式。许多你会发现的表达式-涉及变量或数组的地址都是有效的，因为它们能在编译或加载时求值;

2348：从格式字符串中提取“c”个连续字符;

2349：如果“c”不是’%'，那么.....

2350：如果“c”是空字符('\ 0')，那么表示格式字符串以正常方式结束且“printf”终止;

2351：否则，调用“putchar”把字符发送到系统控制台终端;

2353：看到’%’字符，获取下一个角色(最好不要是’\ 0’!);

2354：如果该字符为’d’、’l’或'o'，那么就调用“printn”作为参数传递“adx”引用的值；另一个取决于“c“的值，若其值为’o’，则该参数值为“8”，否则为“10”(’d’和’l’代码明显相同)；“printn”把一个二进制数按第二个参数所表示的基数转换为一组数字字符;

2356：如果编辑字符为’s’，那么除null终止字符串的最后一个字符之外所有字符都将被发送到终端；在这种情况下，“adx”应指向一个字符指针;

2361：增加“adx”以指向栈中的下一个机器字，即传递给“printf”的下一个参数;

2362：返回第2347行并继续扫描格式字符串；喜欢结构化编程的开发者更愿意用“while(1){”and“}”替换第2347行和第2347行。

5.8 printn(2369)

“printn”以递归方式调用自身以便按所需顺序生成所需的数字。我们有可能更有效地编写这个过程，但是它却难以进一步完善。无论如何，鉴于“putchar”过程的实现，几乎不用在此考虑效率。

假设n=A*b+B，其中A=ldiv(n,b)且B = lrem(n,b)满足0≤B<b为显示n的值，我们需要先显示A的值，然后显示B的值。

对于b=8或b=10，后者是很容易实现：它由单个字符组成；如果A=0，那么前者很容易；递归调用“printn”也很容易；由于A<n，递归调用链一定会终止。

2375：算数运算能很方便地对应于数字转换的字符，其方法是把算术值域字符“0”相加；过程“ldiv”和“lrem”把它们的第1个参数视为无符号整数，在实际除法运算之前把16位值扩展为32位值时，没有符号扩展；它们分别从第1392行和第1400行开始。

5.9 putchar(2386)

该过程作为参数传递的字符传输到系统控制台，它通过一个小例子阐述PDP11计算机上I/O操作的基本功能。

2391：“SW”在第0166行定义为值“0177570”；这是一个只读CPU寄存器的内核地址，用于存储控制台开关寄存器的设置。该语句的含义很明确：获取位置0177570的内容并查看它们是否为0。问题是在C语言中表达这一点。代码

```
if (SW == 0)
```

不会表示这个意思；“SW”显然是一个应解引用的指针值；编译器或许已改为接受

```
if (SW-> == 0)
```

这在语法上是不正确的；通过创建一个具有元素“integ”的伪结构，见第0175行，程序员找到一个满意的解决方法。

无论是在此过程，还是在其它地方，我们能找到类似示例。在硬件方面，系统控制台终端接口由四个16位控制寄存器组成，寄存器在内核地址0177560处从Unibus上获得连续地址，参见第0165行的“KL”的说明。关于寄存器的格式和用法，请参见“PDP11外设手册”第24章。在软件方面，这个界面是未命名的从2313行开始定义的结构，它有4个元素，这4个元素的名字是四个控制寄存器的名字。我们无须分配给它任何实例，因此该结构无命名并不重要。我们感兴趣的是在“KL”所给与的地址上预定义的东西。

2393：当发送器状态寄存器(“XST”)的第7位为0时，系统不做任何操作，因为接口尚未准备好接受另一个字符，这是“busy waiting”的经典案例，此时，CPU无用地反复循环执行一组指令，直到发生一些外部定义的事件。UNIX系统一般不能容忍这种处理能力的浪费，但是它在本例中能被接受；

2395：本语句的必要性与2045行的语句密切相关。

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的当前内容，为发送下一个字符作准备：

2399：当控制状态寄存器的第7位复位时，把要发送的下一个字符移到发送器缓冲寄存器中，这将启动下一个输出操作；

2400：一个“newline”字符需要由“carriage return”字符完成，这是通过对“putchar”的递归调用完成的; 它还抛出一些额外的“delete”字符以便在终端完成回车操作时有任何延迟;

2405：当我们用参数0调用“putchar”的调用，就能有效地重新执行2391~2394代码行；很难理解为程序员为何在这里选择递归调用，而不是简单地重复第2393—2394代码行；即便不考虑清晰度，仅考虑代码效率和紧凑性，递归调用也是不可取的。

2406：恢复发送器状态寄存器的内容，如果先前把位6设置为允许中断，那么就要进行重置。

5.10. panic(2419)

从UNIX操作系统中有许多地方要调用panic过程，例如1605行，当存在这样的情况时，系统的持续运行似乎是不合需要的。UNIX并不自称是一个“容错”或“软故障”系统，因此，在许多情况下，对“panic”的调用可解释为一个很简单的响应。然而，对于更复杂的响应，就需要额外增加大量代码，这与UNIX遵循的“keep it simple”的哲学是背道而驰的。

2419：此语句的作用在第2323行开头的评论中给出;

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

2421：使用格式字符串和一个参数调用“printf”，该参数被传递给“panic”;

2422：“for”语句定义一个无限循环，其唯一的动作是对汇编语言过程“idle”的调用(1284)；“idle”把CPU优先级降为零并执行“wait”，这是一个“do nothing”的指令，它持续无限长时间。当一个硬件中断发生时，系统就终止“wait”指令。一个对“idle”的无限调用指令优于执行一个“halt”指令，因为正在进行的任何I/O活动都被允许完成，系统时钟能继续工作(ticking)。操作员从“panic”中恢复的唯一方法是重新初始化UNIX系统；如果有必要，那么在内存转储(core dump)之后对系统进行初始化。

5.11 prdev(2433)，deverror(2447)

当UNIX系统的I/O操作发生错误时，这两个过程会提供警告消息；在此阶段，它们唯一关注的是使用“printf”的实例。

5.12. 包含的文件

请注意：虽然文件“malloc.c”不包含包含其它文件的请求，但是包含四个单独文件的请求在“prf.c”的开头。细心的读者会注意到：我们假定这些文件在文件层次结构中比“prf.c”本身高一级。第2304行的语句应理解为它被文件“param.h”的全部内容取代。然后，它为“putchar”中出现的标识符“SW”、“KL”和“integ”提供定义。

我们此前注意到“KL”、“SW”和“integ”语句分别出现在第0165行、第0166行和第0175行；若文件“param.h”没有被包含，则毫无意义。在“prf.c”中，包含文件“buf.h”和“conf.h”以便提供“d_major”，“d_minor”，“b_dev”和“b_blkno”语句，这些语句应用在“prdev”和“deverror”。

我们难以发现包含在第四个文件“seg.h”的语句；从代码分析来看，我们并不需要这段代码，“pfr.c”的编译者应为此向读者致歉。在编辑完这一部分源代码时，把“integ”的声明从“seg.h”移到“param.h”或许是一个好主意，证明完毕。

请注意：变量“panicstr”(2328)也是一个全局变量，但由于它未在“prf.c”之外引用，因此其声明未放在任何“.h”文件中。

6. 系统启动

本章阐述UNIX“rebooted”时发生的事件序列，即它在一台空闲的机器中加载和启动的过程。一般来说，对UNIX系统初始化过程的研究本身是有趣的，更重要的是它能把系统的许多重要特征以有序方式呈现出来。当系统崩溃时，我们或许要重启UNIX操作系统。除此而外，有一些很普通的操作上的原因，例如：在隔夜关机之后，我们要经常重新启动系统，

如果假设它是后一种情况，那么我们认为所有磁盘文件都是完整的且不需要识别或处理特殊情况。特别的，我们假设根目录中有一个名为“/unix”的文件，它是操作系统的目标代码；这个文件源于我们正在研究的一组源文件。它们以正常方式编译和链接在一起，形成单个目标程序文件，并存储在根目录中。

6.1. 操作员操作

重新初始化需要操作员在操作控制台上采取行动。

- 终止CPU，把“enable/halt”开关设置为“halt”;
- 使用硬件引导加载程序的地址设置开关寄存器;
- 按下并释放“load address”开关;
- 把“enable/halt”开关移至“enable”;
- 按下并释放“start”开关。

这激活UNIX系统的自举程序(Bootstrap Program)，该程序常驻在CPU的ROM中；引导加载程序加载一个更大的加载程序(位于系统磁盘的#0块)，它寻找并把一个名为“/unix”的文件加载到主存的低地址部分；然后，它把控制转移到已装入#0地址的指令；地址0单元是一条分支转椅指令(第0508行)，它分转移到地址000040单元，其中包含一条跳转(jump)指令(第0522行)，它跳转到标记(label)为“start”的指令，该指令在文件“m40.s”中(第0612行）。

6.2. Start(0612)

0613：测试存储器管理状态寄存器的“enable”位为SR0。如果设置，CPU将一直执行两个指令循环。在启动系统之前，当操作人员激活控制台上的“clear”按钮时，该寄存器通常会清除。专家们提出许多理由说明这种循环的必要性。最有可能的是这样一种情况：在双总线超时错误的情况下，处理器将转移到#0单元且在这种情况下不应允许程序向前执行。

0615：“reset”清除并初始化所有外围设备控制和状态寄存器；现在，系统以内核态(Kernel Mode)运行，并禁用内存管理。

0619：KISA0和KISD0存储在存储空间的高地址部分，它们是第一对内核态的段寄存器的地址，前6个内核描述符寄存器初始化为077406，它说明该端长度为4K机器字，存储控制是读/写。前6个内核地址寄存器分别初始化为0、0200、0400、0600、01000和01200。前6个内核段被初始化(无任何对UNIX实际大小的引用)指向物理内存的前6个4K字段，因此，对于0~0137777范围内的内核地址，内核到物理地址的转换是微不足道的。

0632：“_end”是一个加载器的伪变量，它定义程序代码和数据区的范围；该值向上舍入为64字节的下一个整数倍并存储在第7段的段地址寄存器中(segment #6)。请注意该寄存器的地址存储在“ka6”中，因此该寄存器的内容可作为“*ka6”访问;

0634：对应的描述符寄存器加载一个值(因为“USIZE”等于16)，该段长度为16×32=512字长，其访问控制是对读/写段的描述；把八进制值017向左移动8个位置，然后，与值6做或运算获得值007406;

0641：第8段映射到物理地址空间的最高4K字段。请注意：在禁用存储器管理的情况下，相同的转换已经生效，即32K程序地址空间的最高4K字段中的地址被自动映射到物理地址空间的最高4K字段。我们也许注意到：从这一点开始，所有内核态的分段寄存器保持不变，只有第7个内核分段地址寄存器例外。UNIX明确地操作该寄存器以指向物理存储器中的各种位置。每一个这样的位置都是512字长区间的的起止点，该区间称为每个进程数据区ppda(per process data area)。现在，第7个内核态的地址寄存器被设置为指向一个段，该段将成为进程#0的进程的ppda。

0646：栈指针设置为指向每个ppda进程数据区的最高机器字;

0647：通过把SR0的值从0递增到1，能方便地设置“memory management enable”位；从这一点开始，所有程序地址都转换为内存管理硬件的物理地址。

0649：“bss”是指block started by symbol，它是程序数据区的第二部分，它不是由加载程序初始化的，参见PM中的“A.OUT(V)”。该区域的下限和上限分别由加载器伪变量“_edata”和“end”定义;

0668：处理器状态字PS(processor status word)改变指示先前模式是用户态。这为不属于内核地址空间的物理内存区域的研究和初始化做好准备。这涉及使用特殊指令“mtpi”和“mfpi”(移动到/来自先前的指令空间)以及对用户态的分段寄存器的一些操作;

0669：然后，调用程序“main”(1550)；读者稍后看到“main”调用“sched”，而“sched”永远不会终止。那么，对“start”的最后三条指令(第0670,0671和0672行)是否需要或它有什么作用，这是有些神秘的。对此问题的解答将在后续阐述。在此期间，读者需要思考为什么、这些代码行究竟干什么?

6.3. main(1550)

进入此程序后：

（a）CPU以优先级0、内核态(Kernel Mode)运行，并且前一模式显示为用户态(User Mode);
（b）已设置内核态的段寄存器并启用MMU内存管理单元;
（c）UNIX操作系统使用的所有数据区均已初始化;
（d）栈指针(SP或r6)指向包含“start”的返回地址的机器字。

1559：“main”的第一个动作似乎是多余的，因为“start”执行的初始化命令已把“updlock”设置为0；

1560：对于进程#0，“i”被初始化为超出ppda每进程数据区的第一个32字块的序数;

1562：第一对用户态的段寄存器用于向物理存储器的较高区域提供“moving window”；在窗口的每一个位置尝试用“fuibyte”读取窗口中的第一个可访问机器字；如果无法访问，那么我们假定已到达物理内存的末尾；否则，下一个32字块初始化为0(用“clearseg”(0676))，并把该块添加到可用存储器列表中，然后窗口前进32个机器字。“fuibyte”和“clearseg”都能在“m40.s”中找到，“fuibyte”通常会返回0~255范围内的正值。在异常情况下，若存储位置引用没有响应，则返回值-1，这种阐述方式有点模糊，我们将在第10章中解释。

1582：“maxmem”定义用户程序能使用的最大主存储器容量，它是以下值中最低的：

- 物理可用内存(“maxmem”);
- 可安装的可定义参数(“MAXMEM”)(0135);
- PDP11架构带来的最终限制;

1583：“swapmap”定义交换磁盘(swapping disk)的可用空间，当用户程序从主存换出时可使用该磁盘空间；它被初始化一个大小(长度)为“nswap”的区域，从相对地址“swplo”开始。请注意“nswap”和“swplo”在“conf.c”中初始化(第4697,4698行);

1589：不久将讨论这个和接下来4行代码的重要性;

1599：UNIX设计中假定存在一个系统时钟(system clock)，它以线路频率50Hz或60Hz中断CPU。它有两种可用的时钟类型。第一种是线频时钟(KW11-L)，在Unibus上有一个控制寄存器，它位于地址777546；第二种是可编程实时时钟(KW11-P)，它位于地址777540(1509行,1510行)。UNIX不会假定存在哪个时钟，它第一次尝试读取线频时钟的状态字。如果成功，那么该线频时钟被初始化。而另一个(如果存在)，那么处于闲置状态。如果第一次尝试不成功，那么尝试另一个时钟。如果两次尝试均不成功，那么会出现“panic”调用，并且系统会有效地停止系统并向操作员发送错误消息。先把一对用户态的段寄存器设置为适当值(1599,1600)，然后通过“fuiword”引用时钟状态字，若发生总线超时，则表示不存在该时钟。

1607：语句初始化任一类型的时钟

```
*lks = 0115;
```

该操作的结果是时钟将在20毫秒内中断CPU，该中断可能在任何时候发生；为方便讨论，我们假设在系统初始化完成之前不会发生中断。

1613：“cinit”(82l4)初始化字符缓冲池，参见第23章;

1614：“binit”(5055)初始化大缓冲池，参见第17章;

1615：“iinit”(6922)初始化根设备的表条目，参见第20章。

6.4 进程

进程(process)是一个已不止一次发生的操作系统术语，我们认为进程的合适定义是执行中的程序(a program in execution)。UNIX中进程如何表示的详细资料将在第7章中讨论。我们目前仅关注每个进程包括一个名为“proc”结构，它源于一个被称为proc的数组和一个ppda进程数据区，其中ppda包含一个“u”结构体的副本。

6.5 proc[0]的初始化

结构“proc[0]”的显式初始化从UNIX V6源代码第1589行开始执行，只有4个元素从0的整数初始值改变(请注意core来源于王安博士发明的磁芯内存，至今仍在UNIX中表示内存)：

（a）“p_stat”设置为“SRUN”，表示过程#0准备好运行;

（b）“p_lag”设置为显示“SLOAD”和“SSYS”，前者意味着进程能在内存(core)中找到，即它没有被交换(swap out))到磁盘上；后者意味着它永远不会被换出;
（c）“p_size”设为“USIZE”;
（d）“p_addr”设置为内核(kernel)分段寄存器#6:

我们能看到进程#0已获得“USIZE”块的区，它恰好是ppda每个进程数据区的大小，该区在UNIX操作系统的数据区正式结束(“_end”)后立即开始。已存储该区第一个块的序号以供将来在“p_addr”中引用。该区域在“start”(0661)中被清0，它包含一个名为“u”的用户结构副本。在第1593行，“proc[0]”的地址存储在“u.u_procp”中，即“proc”结构和“u”结构相互链接。

6.6. 故事继续……

1627: “newproc”(1826)将在第7章详细讨论，简而言之，它初始化第2个“过程”结构即“proc[1]”，并在内存(core)中分配第二个ppda每个过程数据区，它是进程#0的ppda副本。这两个过程结构是完全相同的，但是有一点不同：第二个区中“u.u_procp”的值是“&proc[1]”，其中&表示取地址符。我们此刻应注意：在第1889行，有一个调用“savu”(0725)，它在复制之前保存环境的当前值和“u.u_rsav”中的栈指针。同样从第1918行我们能看到“newproc”返回的值将为0，因此第1628~1635行的语句将不会被执行;

1637：调用“sched”(1940)，我们能观察到它包含一个无限循环，因此它永远不会返回!

6..7 sched(1940)

在此阶段，我们仅对第一次输入“sched”时会发生什么感兴趣。

1958：“spl6”是汇编程序的例程(1292)，它把CPU的优先级设置为6，请参见“m40.s”中的“spl0”、“spl4”、“spl5”和“spl7”；当CPU处于第6级时，只有优先级为7的设备才能中断它；因此禁止优先级为6的时钟在该点与第1976行的“spl0”之后的调用之间中断处理器。

1960：通过“proc”进行搜索，其状态为“SRUN”且不是“加载”；进程#0和#1的状态为“SRUN”并被加载；在所有剩余的2193：进程的状态为零，它相当于“undefined”或“NULL”;

1966：搜索失败(“n”仍为-1)，标志“runout”变为非0，它表示没有准备好运行并换出到磁盘上的进程;

1968：“sleep”被称为等待这样的事件，其优先级为“PSWP”(== -100)；当它被唤醒时，就属于“非常紧急”的类别。

6.8. sleep(2066)

2070：“PS”是处理器状态字的地址，CPU状态存储在寄存器“s”(0164, 0175)中;

2071：“rp”被设置为当前进程的数组“proc”中条目的地址(在此阶段仍为“proc[0]”!);

2072：由于“pri”为负数，因此这里采用“else”分支，把当前process(0)的状态设置为“SSLEEP”；请注意“进入睡眠状态”和“环形优先级”的原因。

2093：然后调用“swtch”；

6.9 swtch(2178)

2184：“p”是静态变量(2180)，这意味着其值被初始化为零(1566)且在调用之间被保留；对于第一次调用“swtch”，“p”设置为指向“proc[0]”；

2189：调用“savu”保存“u.u_rsav”中当前进程的栈指针(Stack Pointer)和环境指针(Environment Pointer);

2193：调用“retu”把段#6的内核地址寄存器重置为当作参数传递的值，这导致当前进程发生更改；把栈指针和环境指针重置为适合修订值的当前进程，其执行将立即恢复。

此时对“savu”和“retu”的连续调用组合构成所谓的协程跳转(coroutine jump)，例如在CS上的交换跳转(exchange jump)或/360上的“加载PSW”或在B6700上“移动栈”。

是这次协程跳转是从进程0到进程0(不是很有趣!)；

2201：搜索该组进程以找到状态为“SRUN”且被加载并且“p_pri”为最大值的进程；搜索成功并找到进程#1；请注意进程#0的状态刚从在休眠状态的“SRUN”变为“SSLEEP”，因此它不再满足搜索标准;

2218：由于“p”不是“NULL”，因此不进入空闲循环(Idle Loop);

2228：“retu”(0740)导致协程跳转到process #1，使之成为当前进程；什么是process #1？它是process #0的副本，它在process #1存在时的前一阶段复制；对“retu”调用之前没有调用“savu”，因为实际上已保存必要的信息(在哪里?);

2229：“sureg”是例程1738，把它复制到用户态的段寄存器中，这些值适合于当前进程，它们已存在数组“u.u_uisa”和“u.u_uisd”中；对“sureg”的第一次调用复制0且无任何实际用途；

2240：未设置“SSWAP”标志，因此现在可忽略此神秘部分(2239);

2247：最后“swtch”返回值为“1”；但“return”返回到哪里？不要返回到“sleep”！返回“return”遵循由栈指针和环境指针设置的值；这些(在返回之前)的值等于最近执行“savu(u.u_rsav)”时的有效值；目前process # 1刚开始，它从未执行过“savu”，但是在“newproc”复制process # 0之前，它的值被存储在“u.u_rsav”中，而“newproc”是从“main”调用的；因此，在此情况下，从“swtch”返回到“main”，值为1(请反复阅读该段并确保你能理解!)；

6.10 重访main

到目前为止的故事：process #0以process #1的形式创建自己的副本，已进入眠状态；结果，process #1成为当前进程并返回到“main”，返回值值为1；现在请继续阅读……

1627：现在执行基于“newproc”条件的main()语句；

1628：“expand”(2268)为process #1找到一个更新和更大的存储区(从USIZE*32到(USIZE+1)*32个机器字)，并把原始数据区复制到其中；在此情况下，原始用户数据区仅包含ppda每个过程数据区、0长度数据和栈区域，而原来的存储区被释放;

1629：“estabur”用于设置“prototype”段寄存器，它们被存储在“u.u_uisa”和“u.u_uisd”中，这些值以后供“sureg”使用；作为最后的行动，“estabur”调用“sureg”；“estabur”的4个参数分别是文本、数据和栈区大小以及一个指示符(descriptor)，该指示符用于确定文本和数据区是否应位于单独地址空间中的指示符；在PDP11/40上，这两个区并不分别存放在不同地址空间内；长度均以32机器字为单位;

1630：“copyout”(1252)是一个汇编程序例程，它把内核空间中一个指定的数组复制到用户空间的存储区中；在这里，数组“icode”被复制到用户空间中从0开始的一个存储区;

1635：“return”在此并不特别；从“main”开始，它进入“start”(0670)，其中三个最后指令具有在用户态(User Mode)地址为0处指令的用户态的执行效果，即在“icode”中执行第一条指令的副本，随后执行的指令也是“icode”的指令副本；在这一点上，系统初始化是完整的；process #1正在运行且所有意图和目的都是正常的进程; 其初始形式(几乎)来自编译、加载和执行简单但非平凡的C语言程序：

```
    char *init "/etc/init";
        main ( ) {
        execl (init, init, 0);
        while (1);
}
```

请参见以上C语言程序等价的汇编程序

```
            sys exec
            init
            initp
            br   .
    inip:   init
            0
    init:  </etc/init\0>
```

如果系统调用“exec”失败(例如找不到文件“/etc/init”)，那么该进程进入无限循环，除非发生时钟中断，否则CPU就停止不前；“/etc/init”所执行的功能描述可在UPM的“INIT(VIII)”中找到。

7. 进程(Process)

前一章描述在UNIX操作系统重新启动之后发生的事态发展，并且这样做引入进程概念的许多重要特征。本章的目标之一是回顾并更彻底地重新探讨一些相同的观点。在提供普遍接受的“process”定义方面存在许多严重的困难。这类似于哲学家所面临的困难，他们会回答“生命是什么？”如果轻轻地将更微妙的要点放在一边，那么我们将会很好地合作。

我们此前已定义进程是执行中的程序，在确定预期内容方面做得相当不错。然而，它不适合process＃0在其整个生命周期或process＃1的第一时刻。但是UNIX系统中的所有其它进程都与某些程序文件或其它程序文件的执行明显相关。因此我们在两个层面上把进程引入到操作系统的讨论中。

从上层而言，进程是用于描述整个计算机系统活动的重要组织概念。我们通常很容易把后者视为许多过程的组合活动，每个进程都与特定程序如“shell”或“编辑器”相关联。在Ritchie和Thompson的“UNIX分时系统”论文的第二部分，对这一级别的UNIX进行讨论。在这个层面上，进程本身被认为是系统中的活跃实体，而真正的活动元素、CPU和外设的身份被淹没：进程诞生、生存和死亡；它们以不同的数量存在，它们能获得和释放资源；它们能互动、合作、冲突、分享资源等；

在底层而言，进程是非活动实体，由CPU等活动实体操作，通过允许CPU频繁地从一个进程镜像的执行切换到另一个进程镜像，我们能创建每个进程镜像不断发展的印象，这导致上层解释。我们目前关注的是低级别的解释：进程映镜像的结构、执行细节以及在IPC进程间切换CPU的方法。我们对UNIX操作系统上下文中的进程进行以下观察：

（a）“proc”数组中非空结构的存在暗示一个进程，即一个“proc”结构，其元素“p_stat”是非空的;
（b）每个进程都有一个ppda过程数据区，其中包含“User”结构的副本;
（c）CPU的整个生命周期都在执行一个或另一个进程，除非在指令之间进行休息;
（d）一个进程可能创建或破坏另一个进程;
（e）一个进程能获得并拥有各种资源。

7.1. 进程

Ritchie和Thompson在他们的论文中把进程定义为镜像(image)的执行，其中镜像是伪计算机的当前状态，即抽象数据结构，它可在主存或磁盘上；进程映像涉及两个或三个物理上不同的内存区域：

（1）“proc”结构：包含在内存常驻“proc”数组中且随时能访问;
（2）数据段：由ppda过程数据区组成，与包含用户程序数据、程序文本和栈的段组合;
（3）不总是存在的文本段由仅包含纯程序文本的段组成，包括可重入代码和常量数据。

许多程序没有单独的文本段。如果定义一个，那么在执行同一特定程序的所有进程之间共享一个副本。

7.2 proc结构(0358)

该结构永远驻在主存中，它包含15个元素，其中8个是字符，6个是整型，1个是指向整型的指针。每个元素代表必须随时访问的信息，特别是当进程映像的主要部分已被换出到磁盘时：

- “p_stat”可采用7个值中的一个来定义七个互斥状态，请参见第0381至0387行;

- “p_glag”是6个一位标志的混合(amalgam)，可独立设置, 请参见第0391至0396行;
- “p_addr”是数据段的地址：
    - 如果数据段在主存储器, 那么这是一个块号;
    - 否则，如果数据段已被换出，那么这是一个磁盘记录号;
- “p_size”是数据段的大小，它以块为单位进行测量;
- “p_pri”是当前的进程优先级，这可作为“p_nice”、“p_cpu”和“p_time”的函数不时重新计算；
- “p_pid”：“p_ppid”是唯一标识进程及其父进程的数字;
- “p_sig”、“p_uid”和“p_ttyp”涉及外部通信，即来自进程正常域外的消息或信号;
- “p_wchan”标识符：对于“睡眠”过程(“p_stat”等于“SSLEEP”或“SWAIT”)，睡眠的原因;
- “p_extp”为null或指向“text”数组(4306)项的指针，包含有关文本段的重要统计信息。

7.3 User Structure(0413)

“user”结构的一个副本是每个ppda的基本信息，在任何时候，只能访问一个“user”结构的副本，它的名称为“u”且总能在内核地址0140000处找到，即在内核地址空间的第7页的开头；“user”结构包含的元素多于此处可方便或有用的元素；表04中每个声明附带的评论简洁地表明每个元素的功能。目前你应注意到：

（a）“u_rsav”、“u_qsav”、“u_ssav”，它是用于存储r5，r6值的两个机器字的数组;
（b）“u_procp”，它在“proc”array中给出相应的“proc”结构的地址;
（c）“u_uisa[16]”、“u_uisd[16]”，用于存储页地址和描述寄存器的原型;
（d）“u_tsize”，“u_dsize”，“u_ssize”，它们是文本段大小，两个参数定义数据段大小，用32个字块进行测量。

要考虑的余下元素包括：

- 保存浮点寄存器(不适用于PDP11/40)；
- 用户识别;
- 输入/输出操作的参数;
- 文件访问控制;
- 系统调用参数;
- 统计信息。

7.4. ppda过程数据区

ppda对应于内核地址空间的第7页的有效部分(低地址区)，它长为1024个字节，较低的289字节被“user”结构的实例占用，留下367个字用作内核态的栈区；显然会有与进程一样多的内核态的栈区。当CPU处于内核态时，r5和r6(环境指针和栈指针)的值应保持在0140441至01437777的范围内；超出上限的转换将被捕获为段违规，但是较低限制仅受软件完整性的保护；注意到UNIX系统不使用硬件栈限制选项。

7.5. 段

数据段被分配为物理内存的单个区域，但由三个不同的部分组成：

（a）ppda过程数据区;
（b）用户程序的数据区域: 它进一步划分为程序文本、初始化数据和未初始化数据的区域;
（c）用户程序的栈。

其中(a)的大小始终是“USIZE”块，(b)和(c)的大小以“u.u_dsize”和“u.u_ssize”为单位给出。我顺便指出后两者可能在进程生命周期中发生变化；仅包含纯文本的单独文本段被分配为物理存储器的单个区，而该部分的内部结构在这里并不重要。

7.6. 镜像执行

第7内核态的段地址寄存器确定当前被执行的镜像以及当前进程的标识。如果process #i是当前进程，那么寄存器的值为“proc[i].p_addr”。

我们一般需要区分内核态的执行进程和用户态的执行进程，并分别用术语“kernel process #i”和“user process #i”表示。

如果选择把进程与特定执行栈相关联、而不是与“proc”数组中的元素(条目)相关联，那么我们会把kernel process #i和user process #i视为单独的进程、而不是单个进程不同方面的process #i。

7.7. 内核态执行

我们必须适当地设置第7个内核态的段地址寄存器，而其它内核态的段寄存器在系统启动后不会受到干扰。如前所述，前6个内核态的页面映射到前6个物理内存的页面，而第8个内核态的页面映射到物理内存的最高页面。虽然第7段的地址经常变化，但是其长度始终相同。在内核态下运行时，用户态的段寄存器设置通常是无关紧要的。然而，它们一般按照用户进程正确设置。环境指针和栈指针指向第7页内核态的栈区，该栈区位于“user”结构之上。

7.8. 用户态执行

用户进程的每次激活都在相应的内核进程激活之前和之后进行。无论何时，如果一个进程镜像在用户态下执行，那么与其对应的用户态和内核态的寄存器都将被正确设置，而环境指针和栈指针指向用户栈区。该栈始于第8个用户页面的上半部分，但是它能够向下扩展，例如，占据整个第8页和第7页的一部分或全部。虽然内核态的段寄存器设置相当简单，但是用户态段寄存器的设置则相当麻烦。

7.9. 一个实例

考虑PDP11/40上的一个程序，该程序使用1.7页文本、3.3页数据和0.7页栈区；我们在该例中对小数(decimal fraction)的使用无疑是有点粗糙；虚拟地址集将按如下图所示进行划分：

图1(略)

必须把虚拟地址空间中的两个整页分配给文本段，即使所需的物理区域仅为1.7页；数据和栈区分别需要4页和1页虚拟地址空间以及3.3和0.7页物理内存。整个数据段需要4又八分之一页的物理内存，额外的1/8页用于ppda每个过程数据区”，它对应(不时)到第7个内核态的地址页，见图三。

请注意数据段各部分的顺序且各部分之间没有的未用空间，需要设置用户态的段寄存器以反映下表中的值，其中“t”、“d”分别表示文本和数据段起始地址块编号，请参见图四。请注意第8个地址寄存器的设置，我们通过把“t”和“d”设置为0从而获得存储在数组“u.u_uisa”中的地址原型。

图2~图4(略)

7.10. 设置段寄存器

用户态段寄存器的原型(prototypes)由“estabur”设置，当程序首次启动执行时调用；无论何时，当内存分配发生重大变化时，它就需要被再次调用；这些原型存放在数组“u.u_uisa”、“u.u_uisd”中。每当process #i即将被重新激活时，调用过程“sureg”把原型复制到相应的寄存器中。虽然描述寄存器被直接复制，但是地址寄存器必须被调整以反映所用区域的物理存储器的实际位置。

7.11 estabur(1650)

1654：它执行各种一致性检查，确保文本、数据和栈的请求大小合理；请注意“sep”的非0值表示文本区(“i”空间)和数据区域(“d”空间)的单独映射，这在PDP11/40上是不可能的;

1664：“a”定义相对于任意基地址为0的段地址，“ap”和“dp”分别指向原型段地址和描述符寄存器的集合，每一个集合的前8个意图引用“i”空间。

1667：“nt”测量文本段所需的32个字块的数量；如果“nt”不为零，那么必须为此分配一个或多个页面。

在分配多页的情况下，除最后一页外，其它所有页都由128个块(4096个字)组成且设为只读，它们的相对地址从0开始且连续增加128的相对地址。

1672：如果仍然要分配少于一页(小数计量)的文本，那么就分配下一页的适当部分;

1677：如果分别使用“i”和“d”空格，那么把剩余“i”页的段寄存器标记为空;

1682：因为所有剩余地址都指向数据区(不是文本区域)且相对于该区的起点，因此“a”被重新设置；该区的第一个“USIZE”块保留用于ppda过程数据区;

1703：栈区域从地址空间的顶部向下部地址分配(“downwards”);

1711：如果必须为栈区分配页的一部分，那么就分配该页的有效高地址部分；如果它是向上(upwards)增长的文本和数据区，那么就分配该页的有效低地址部分；这需要描述符中的一个额外位，因此这里是“ED”(“expansion downwards”);

1714：若不使用单独的“i”和“d”空间，则此点仅初始化16个原型寄存器对中的前8个；在此情况下，后8个是从前8个中复制的。

7.12. sureg(1739)

sureg例程被“estabur”(1724)、“swtch”(2229)和“expand”(2295)，以便把原型段寄存器复制到实际的硬件段寄存器中。

1743：从“proc”数组的相应元素中获取数据区的基址；

1744：原型地址寄存器(PDP11/40只有8个)通过添加“a”进行修改并存在硬件段地址寄存器中;

1752：测试是否已分配单独的文本区域，如果已分配，那么就把“a”重置为文本区与数据区的相对地址；注意这个值可能是负数，幸运的是，此时地址是32个机器字块；

1754：紧接着的代码模式类似于例程的开始部分，除了……

1762：一段相当模糊的代码调整地址寄存器的设置，用于不是“可写”的段，它可能是文本段；“estabur”和“sureg”中的代码表明已分几个阶段开发且不如期望的那样优雅；

7.13 newproc(1826)

现在是时候了，让我们仔细看一下创建新进程的过程，它几乎是其创建者的精确复制品。

1841：“mpid”是一个整型值，它在通过值0~32767范围内逐一递增；在创建每个新进程时，创建“mpid”新值以便为该进程提供唯一的区分号；由于值的循环可能最终重复，因此检查该数字是否仍在使用中；如果是这样的话，那么就尝试新的值;

1846：通过“proc”方案搜索空“proc”结构，由“p_stat”表示，具有空值;

1860：此时，“proc”数组的新条目的地址存储为“p”和“rpp”，并且当前进程的“proc”项的地址存储为“up”和“rip”;

1861：把新进程的属性存储在新的“proc”条目中，其中许多都是从当前进程中复制出来的;

1876：新进程继承其父进程的打开文件，增加每个引用计数;

1879：若存在单独的文本段增量，则关联的引用计数；请注意“rip”，“rpp”在这里用于临时引用;

1883：增加父进程的当前目录的引用计数;

1889：在“u.u_rsav”中保存环境的当前值和栈指针，“savu”是第0725行定义的汇编程序;

1890：恢复“rip”和“rpp”的值，暂时把“u.u_procp”的值从适合当前进程的值更改为适合新进程的值;

1896：尝试在主存中找到要创建的新数据段区;

1902：如果主存中没有合适区，那么就必须在磁盘上创建新副本；由于第1891行引入的不一致性，应仔细分析码u.u_procp-> p_addr！= * ka6;

1903：把当前进程标记为“SIDL”以便暂时阻止任何进一步交换它的尝试，即由“sched”(1940)发起）；

1904：使新的“proc”条目保持一致，即设置rpp-> p_addr = * ka6;

1905：在“u.u_ssav”中保存环境的当前值和栈指针;

1906：调用“xswap”(4368)把数据段复制到磁盘交换区；因为第2个参数为零，因此不会释放该数据段创建的主存储区;

1907：把新流程标记为“换出”;

1908：使当前进程恢复到正常状态

1913：在主存中有足够空间，因此存储新“proc”条目的地址并一次一块地复制数据段;

1917：把当前进程的“每个进程数据区”恢复到以前的状态;

1918：返回值为零；显然，“newproc”本身并不足以产生一组有趣且多样化的过程；第12章讨论的程序“exec”(3020)提供必要的附加设施：一个进程须改变其特性以便重生。

8. 进程管理

进程管理涉及在多个进程之间共享CPU和主存，这些进程被视为这些资源的竞争者；在初次启动资源的进程时或其它原因，我们需要经常作出重新分配资源的决定。

8.1. 进程切换

一个进程通过调用“retu”(0740)的“swtch”(2178)，它在激活过程能挂起自身即放弃CPU。如果一个过程已达到一个不能立即进行的点，就能做到这一点。该过程称为“sleep”(2066)，它称为“swtch”；或者，准备恢复到用户态的内核进程将测试变量“runrun”，如果这是非0，那么就暗示具有更高优先级的进程已准备好运行，则内核进程将调用“swtch”。“swtch”搜索“proc”表，查找“p_stat”等于“SRUN”且“SLOAD”位设置为“p_flag”的条目。从中选择“p_pri”值最小的过程，并把控制转移给它。

通过使用“set-pri”(2156)程序，系统经常为每个进程重新计算“p_pri”的值；很显然，“set-pri”使用的算法具有重要影响；一个称为“sleep”并挂起自身的进程可能会被另一个进程重新转变为“准备运行”状态，这通常发生在处理中断期间，当处理中断的过程通过“wakeup”(2113)上的调用直接或间接地调用“setrun”(2134)时；

8.2. 中断(Interrupts)

应该注意硬件中断(参见第9章)并不直接导致“swtch”或其等价的调用；硬件中断导致用户进程恢复到内核进程；如上所述，在中断处理完成后，可调用“swtch”作为恢复用户态的替代方法。如果一个内核进程被中断，那么在处理完中断后内核进程就会在它停止的地方重新开始。这一点对于理解UNIX如何避免与代码的临界区(critical sections)相关的陷阱很重要，本章末尾将讨论这些陷阱。

8.3. 程序交换(Program Swapping)

计算机的内存容量一般不能一次同时容纳所有进程的镜像(images)，因此内存中的一些数据段必须被换出(swap out)，即在写入特殊的磁盘家交换区(disk swapping sections)。在磁盘上，过程镜像相对难以访问，并且肯定是不可执行的，因此必须通过交换镜像定期更改主存储器中的进程镜像集。

关于交换的大多数决定是通过“sched”(1940)程序做出的，该程序在第14章中详细讨论。“sched”由process # 0执行，它完成初始任务后，把其时间用于双重角色：一是公开作为“调度程序”，即正常的内核进程；而二是隐蔽地作为“swtch”的中间进程(第7章已讨论)。由于程序“sched”永远不会终止，因此kernel process # 0永远不会完成其任务，因此它不会出现user process # 0的问题。

8.4. 作业(Job)

在UNIX中没有作业的概念，至少在更传统的，面向批处理的系统中理解该术语的意义上。

任何过程都可以随时“分叉”自己的新副本，基本上没有延迟，因此创建了相当于新工作的东西。因此，工作调度，工作班等都是非事件。

8.5. 汇编程序(Assembler Language)

接下来的三个进程用汇编器编写，并在CPU优先级设置为7的情况下运行。这些进程不遵守正常的进程条目约定，因此在进程进入和退出过程中，r5和r6(环境和栈指针)不会受到干扰。正如我们已指出的那样，“savu”和“retu”能组合起来产生协同跳转的效果，第三个过程“aretu”，后跟“return”语句产生非本地“goto”的效果。

8.6 savu(0725)

该进程由“newproc”(1889,1905)，“swtch”(2189,2281),“expand”(2284)，“trapl”(2846)和“xswap”(4476,4477)调用；r5和r6的值存入其地址作为参数传递的数组中。

8.7 retu(0740)

该过程由“swtch”(2193,2228)和“expand”(2294)调用，它重置第7个内核段地址寄存器，然后从新访问的“u.u rsav”副本中重置r6和r5；我们注意到它位于“u”的开头；

8.8 aretu(0734)

该过程由“sleep”(2106)和“swtch”(2242)调用，它从作为参数传递的地址重新加载r6和r5;

8.9 swtch(2178)

“swtch”被“trap”(0770,0791)、“sleep”(2084,2093)、“expand”(2287)、“exit”(3256)，“exit”(4027)和“xalloc”(4480)调用。此进程的独特之处在于其执行分为三个阶段，它们通常涉及三个独立的内核进程。这些进程中的第1个和第3个分别称为“retiring”和“arising”的进程，process＃0始终是一个中间进程，它也可能以上两个进程。请注意“swtch”使用的唯一变量是寄存器、全局或静态(全局存储)。

2184：静态结构指针“p”定义搜索“proc”数组以便找到要激活的下一个进程的起点。其使用减少对“proc”数组中早期输入进程所显示的偏差；如果“p”为null，则把其值设置为“proc”数组的开头；这应仅在第一次调用“swtch”时发生;

2189：对“savu”(0725)的调用保存环境和栈指针的当前值(r5和r6);

2193：“retu”(0740)重置r5和r6，尤为重要的是它重置内核地址寄存器6，对“scheduler”的数据段进行寻址;

2195：第2阶段开始：从该行到第2224行代码仅由kernel　process＃0执行；它有两个嵌套循环，除非找到可运行进程，否则它不会退出；在空闲(slack)期间，CPU大部分时间都在执行第2220行，它仅受到中断(如时钟中断)的干扰;

2196：重置“runrun”标志，它指示一个比当前进程更高优先级的进程已准备就绪；“swtch”将寻找最高优先级进程;

2224：一个产生进程(Arising Process)的优先级标注在全局变量“curpri”中，以便将来引用和比较;

2228：对“retu”的另一个调用，它把r5、r6和第7个内核态的地址寄存器复位为适用于“产生进程”的值;

2229：第3阶段开始：“sureg”(1739)用所存储的原型对生成的寄存器重置用户态的硬件段寄存器；

2230：从这里开始的评注并不令人鼓舞，我们将在本章结尾再次讨论这一点;

2247：如果检查代码，那么你就会发现没有一个调用“swtch”的程序直接检查此行的返回值。

8.10 setpri(2156)

由于第一次激活子进程的方式，只有调用对此值感兴趣的“newproc”的过程！

2161：根据下列公式计算进程优先级
priority = min(127, (time used + PUSER + p_nice))
其中：
（1）使用的时间=累计CPU时间(通常是自上次交换过程以来)的滴答(tick)数除以16，即测量单位是1/3秒测量；稍后讨论时钟中断时，我们阐述更多相关内容;
（2）PUSER == 100;
（3）“p_nice”计算进程优先级的偏置值，它通常为正，因此降低进程的有效优先级；请注意UNIX的某些约定令人困惑：优先数(p_pri)越小，优先级越高；因此，-10的优先级高于优先数100。

2165：如果刚刚重新计算的进程优先级低于当前进程，那么就要设置重新调度标志；第2165行的测试结果令人惊讶，特别是与2141行进行比较时，我们把这个问题留给读者思考：为什么这不是一个错误；我们提示查看“setpri”上调用的参数；

8.11 sleep(2066)

当一个内核态进程选择挂起自身时，系统将调用此过程(来自代码中的近30个不同位置:有两个参数：
要睡觉的原因;
•优先考虑过程后的过程
被唤醒
如果该优先级为负，则通过“信号”的到达不能从睡眠中唤醒该过程。 “信号”将在第十三章中讨论。
2070：保存当前处理器状态以保留输入处理器优先级和之前的模式信息;
2072：如果优先级是非负的，则对“等待信号”进行测试;
2075：一个小的关键部分从这里开始，其中过程状态被改变，参数存储在通常可访问的位置（即在数组“proc”中）。
此代码至关重要，因为可以通过“唤醒”（2113）查询和更改相同的信息字段，这经常被中断处理程序调用;
2080：当“runin”非零时，调度程序（进程＃0）正在等待将另一个进程交换到主存储器中;
2084：对“swtch”的调用表示未知程度的延迟，在此期间可能发生了相关的外部事件。因此，对“issig”（2085）的第二次测试并非无关紧要;
2087：对于负优先级“休眠”，其中进程通常等待释放系统表空间，不允许出现“信号”偏转c

2087：对于负优先级“休眠”，其中该过程通常等待释放系统表空间，不允许出现“信号”偏转活动过程。

8.12 wakeup(2113)

该进程是“sleep”进程的一个补充，它仅搜索所有进程的集合、寻找任何因某特别原因(作为参数“chan”给出)“sleep”进程，而且通过“setrun”调用单独重新激活这些进程。

8.13 setrun(2134)

2140：过程状态设置为“SRUN”，它现在通过“swtch”和“sched”把该进程视为再次执行的候选者;

2141：如果引发的进程比当前进程更重要(优先级更低!)，那么就重新安排标志“runrun”被设置为以后的引用；

2143：如果“sched”正在休眠，那么等待进程“swap”；如果新引发的进程在磁盘上，那么唤醒“sched”；因为事实证明“sched”是唯一一个用“chan”(与“&runout”等价)调用sleep的过程，因此第2145行能被下列形式的递归调用替换：

```
    setrun(&PROC[0];
```

或者更好的是，仅仅通过

```
    rp = &proc[0];
    goto sr;
```

其中“sr”是一个要插入第2139行开头的标签。

8.14 expand(2268)

该程序开始时的注释(2251)说明关于程序需要说的大部分内容，唯一需要补充之处是与内存不足时要执行的换出操作有关的问题；请注意“expand”并不特别关注用户数据区或栈区内容。

2277：如果扩张(expan)实际上是收缩(contraction)，那么从高地址端释放多余部分;

2281：“savu”把r5和r6的值存储在“u.u_rsav”中;

2283：如果没有足够的主存.....

2284：环境指针和栈指针再次记录在“u.u_ssav”中；但是，请注意，由于没有新过程(procedure)进入且没有累积栈增长，因此所记录的值与第2281行相同;

2285：“xswap”(4368)把第1个参数指定进程的内核镜像复制到磁盘；由于第2个参数不为0，因此被数据段占用的主存区返回到可用空间列表；然而，计算(computation)继续使用主存的相同区，直到下一次调用“swtch”中的“retu”(2193)为止；请注意:“swtch”中第2189行的“savu”调用将在磁盘镜像后在“uu_rsav”中存入新值；由于核心映射(core map)已被正式放弃，因此它没有实际用处；

2286：“SSWAP”标志设置在进程的proc数组元素中；它不会被换出(swap out)，因此这个效果不会丢失;

2287：当调用“swtch”时，系统挂起(suspend)仍在在旧存储区运行的进程；当调用“xswap”时，标志“SLOAD”被关闭，这时“swtch”无法选择立即重新激活进程；只有把磁盘镜像再次复制到内核(core)后，才能再次激活该进程；“swtch”执行“return”命令，返回到“expand”过程；

8.15 重访swtch

当“swtch”被重新激活时，该进程将发生什么变化？它会在“swtch”中成为“arising”进程吗?

2228：栈和环境指针从“u.u_rsav”重新存储；请注意指向“u”的指针也是指向“u.u_rsav”的指针(0415)，但是…

2240：如果核心映射被换出，例如通过“expand”.....

2242：栈和环境指针的值已不能继续使用，它们已被重置。当前有一个问题：存储在第2284行的“u.u_ssav”中的值是否与第2281行中“u.u_rsav”中存储的值相同，它们是怎样变得不同？据推测，这是“你不应理解的”(第2238行)……显然应研究“xswap”…这个谜底终于在第15章被揭开……同时，你或许希望亲自研究它以便更快地理解第2238行。

8.16 临界区(Critical Sections)

如果两个及以上进程对同一组数据进行操作，那么该组进程的输出可能取决于各进程之间的相对同步状态，这通常被认为是极不可取的且要不惜一切代价避免。其解决方法通常是在每个进程执行的代码中定义临界区。程序员有责任识别出这些临界区并确保在任何时候不会有多个进程同时执行一个包含特定数据集的代码段。

在UNIX系统中，用户态的进程不共享数据，因此不会以这种方式发生冲突。然而，内核态的进程能共享访问各种系统数据且可能会发生冲突。在UNIX中，中断不会导致进程产生直接的副作用。只有内核态的进程通过对“sleep”显式调用并在临界区中间挂起(suspend)自身的情况下，才需要引入由一组进程观察到的显式锁变量。即便如此，测试和设置锁的操作通常也不必是原子的。

某些临界区的代码是由中断处理程序执行的。为保护其输出受某些中断处理影响的代码段，在进入临界区之前，CPU优先级会暂时提高到足以延迟此类中断，直到确保达到安全性为止；然后，CPU将降低其优先级。当然，UNIX中有一些中断处理代码应遵循的约定，这将在第9章中讨论。顺便提一下，读者应注意到UNIX Level 6操作系统所采用的策略仅适用于单处理器(Uniprocessor)，并且完全不适用于多处理器(Multiprocessor)。

第二部分 Trap、中断、系统调用和进程管理

第二部分涉及陷阱、硬件中断和软件中断。陷阱和硬件中断会把突然切换引入CPU的正常指令执行序列。这提供一种机制，用于处理在CPU即时控制之外发生的特殊情况。一种称为系统调用(system call)的机制使用这种设施，由此，用户程序能执行“trap”指令以故意引起陷阱并因此获得操作系统的注意和帮助。软件中断(software)或信号(signal)是IPC进程间通信的机制，特别是当存在“坏消息”时。

9. 硬件中断和陷阱

与许多其它计算机一样，PDP11计算机存在一种中断(interrupt)机制，它允许外围设备设的控制器(CPU外部的设备)在适当的时间中断CPU，同时请求操作系统服务；同样的机制已有效且方便地应用于陷阱(trap)，这些陷阱是CPU内部的事件，它包括到硬件错误、软件错误以及来自用户程序的服务请求。

9.1 硬件中断

中断的作用是把CPU从其正在进行的操作中转移出来，并将其重定向以执行另一个程序，在硬件中断期间：

- CPU把当前处理器状态字(PS)和当前程序计数(PC)保存在其内部寄存器中;
- 然后从位于主存低地址区的两个连续机器字重新加载PC和PS，这两个机器字中第一个的地址称为中断的向量位置(vector location);
- 最后把原始PC和PS值存放在新的当前栈中，它是内核栈还是用户栈取决于PS的新值；

不同外围设备具有不同的向量位置；而一个特定设备的实际向量位置由硬件接线决定，并且它们很难改变；此外，为各种设备选择向量位置有一个很好的惯例。因此，在发生中断后，由于程序计数器(PC)已重新加载，因此CPU执行的指令源也已更改。

新源应该是与引起中断的外围设备控制器相关的过程。此外，由于PS也已更改，CPU的模式可能已更改。在UNIX系统中，初始模式是用户态(User Mode)或内核态(Kernerl Mode)，但是在中断之后，其模式始终是内核态；请注意UNIX系统模式的改变意味着：
（a）内存映射的变化，注意为避免混淆，向量位置总是被解释为内核态的地址；
（b）栈指针的变化，请回忆一下，栈指针SP或r6是唯一一个为每种模式复制的特殊寄存器，这意味着在CPU的模式更改后，即使没有重新加载，栈指针值也会发生变化！

9.2 中断向量

对于我们的样品系统，表9.1列出所选的代表性外围设备以及它们的传统硬件定义的向量位置和优先级。

表9.1中断向量位置和优先级

9.3 中断处理程序

在这个UNIX源代码选择中，有7个称为中断句柄(interrupt handlers)的过程，当且仅当作为中断的结果执行。“clock”将在第11章中详细讨论，其它将与其相关设备的代码进行讨论。

clock
rkintr
klxint
klrint
（3725）pcrint（8719）
（5451）pcpint（8739）
（8070）lpint（8976）
（8078）

9.4 优先级

中断立即发生不是在外围设备控制器请求它的时候，而是在CPU准备好接受它的时候。我们希望一个低优先级服务的请求不应中断具有更高优先级的活动。PS的第7位至第5位确定8个级别之一(标记为0到7)的CPU优先级。此外，每个中断由硬件接线确定的相关优先级。只要CPU优先级大于或等于中断优先级，就会禁止中断。

中断后，CPU优先级根据存储在向量位置的PS确定，它不必与中断优先级相同。中断优先级由硬件决定，而UNIX系统能随时改变向量位置的内容。如果你有好奇心，就会注意到PDP11硬件把可能的中断优先级限制为4、5、6和7，这意味着Unibus不支持级别1、2和3。(3835)在第7级(最高级)运行。

9.5 中断优先级

在UNIX系统中，中断处理例程在启动时与中断的优先级相同。这意味着：在处理中断期间，来自同一优先级设备的第2个中断将被延迟，直到CPU优先级降低。有两种方式降低CPU优先级，一种是通过专门为此目的而执行“spl”过程中的一个(1293行~1315行)，另一种是通过重载CPU状态字如从中断返回。

在中断处理期间，可暂时提高CPU优先级以保护某些操作的完整性。例如，面向字符的设备，例如如纸带读取器/打孔器、行式打印机的中断处于第4级，它们的中断句柄(interrupt handlers)调用“getc”(0930)或“putc”(0967)，它们把CPU优先级暂时提升到第5级，同时操纵字符缓冲区的队列。控制台电传打字机的中断处理程序使用“timeout”功能，这涉及一个队列，该队列也由时钟中断句柄操纵，它在第6级运行。为防止可能的干扰，“timeout”过程(3835行)在第7级的最高级别运行。

一般而言，在CPU优先级低于中断优先级的情况下运行中断处理程序是没有意义的，因为这会在完成第一个中断处理之前冒着相同类型的第二个中断的风险，即使是来自同一设备也是如此。这可能是最不方便的、最糟糕的和灾难性的。然而，时钟中断句柄就是这样做的，它每秒处理一次任务，因此它有很多额外的工作要做。

9.6 中断处理程序的规则

如上所述，中断句柄需要注意CPU优先级的操作，避免允许其它的中断太快发生；同样需要注意其它中断不要过度延迟，以免降低整个系统的性能，特别是，当发生中断时，当前活动的进程很可能是对这种中断的发生不感兴趣的进程，因此请考虑以下情形：

用户进程#m处于活动状态并启动I/O操作，它执行一个trap指令并转换到内核态；内核进程#m启动所需操作，然后调用“sleep”以挂起自身等待操作完成…

一段时间之后，当某个其它进程(如用户进程#n)处于活动状态时，操作完成并发生中断；进程#n恢复为内核态，内核进程#n处理中断，即使它可能对操作没有兴趣或事先知道，也是如此。

一般而言，内核进程#n将包括唤醒进程#m作为其活动的一部分，但情况并非总是如此，例如发生错误并重试操作的地方。

显然，外围设备的中断句柄不应引用当前的“u”结构，因为它不可能是一个合适的“u”结构；如果它被临时交换到磁盘上，那么相应的“u”结构很可能无法访问。同样，中断句柄不应调用“sleep”，因为这样挂起的进程很可能是一写无辜的进程。

9.7 陷阱(Traps)

陷阱就像中断一样是由相同的硬件机制处理的事件，因此它由类似的软件机制处理。然而，陷阱与中断终究是不同的，因为陷阱是CPU内部事件，而不是外部事件的结果。在其它系统中，术语内部中断和外部中断更加有力地作出这种区别。陷阱可由于硬件或电源故障而意外发生、或可预测且可重复地发生，例如，执行一个非法指令或一个陷阱指令。

陷阱始终由CPU识别，不能以低优先级中断的方式延迟。如果你愿意，那么陷阱的中断优先级能设置为8。陷阱指令被有意插入到用户态的程序中，并通过执行指定服务请求引起UNIX系统的注意；该机制是系统调用功能的一部分。与中断一样，陷阱导致从向量位置重载PC和PS及在当前栈中保存PC和PS的旧值。表9.2列出各种陷阱类型的向量位置。

矢量位置	陷阱类型	优先级
004	总线超时	7
010	非法指令	7
014	bpt-trace	7
020	iot	7
024	电源故障	7
030	仿真器陷阱指令	7
034	陷阱指令	7
114	11/10奇偶校验	7
240	编程中断	7
244	浮点错误	7
250	段违例	7

表9.2 陷阱向量位置和优先级

我们应把表9.1和9.2的内容与表05的文件“low.s”进行比较。如前所述，该文件在每次安装时连同文件“conf.c”一起生成，参见表46 ；它作为一个实用程序“mkconf”的产品，反映安装的实际外设集。

9.8 汇编语言中的陷阱

从“low.s”来看，陷阱和中断都是由软件单独进行处理的。然而，一个详细研究表明调用和陷阱分别在文件“m40.s”的单个代码序列的不同入口点，参见第0755行和第0776行，第10章将详细阐述这个序列。在执行该序列期间，对C语言过程进行调用以便进一步的特定处理。在中断的情况下，C过程是一个特定于某个设备控制器的中断句柄。

在有陷阱的情况下，C过程是另一个称为陷阱的过程。是的，陷阱(trap)这个词的确是用得太频繁啦！在系统错误的情况下，最有可能有限调用“panic”；在系统调用的情况下，将通过“trapl”(2841)间接调用一个合适的系统调用过程。

9.9 Return

在完成对一个中断或陷阱的处理后，代码遵循一个的公共路径，以“rtt”指令(0805)结束；这将从当前的内核栈重载PC和PS，以便恢复在中断或陷阱之前存在的CPU环境。

10. 汇编程序的陷阱例程

本章的主要目的是检查在处理中断和陷阱时涉及的“m40.s”中的汇编语言代码。该代码在第0750行和第0805行之间找到，它们有两个入口点，包括“trap”(0755)和“call”(0766)。几条不同的相关路径通过这段代码，我们将追溯这些实例。

10.1 陷阱和中断的来源

第一节讨论了三个发生陷阱或中断的地方：

(a) “main”(1564)重复调用“fuibyte”，直到返回一个负值，这将在总线超时错误(bus timeout error)后发生，并在此之后遇到一个向量位置4(第0512行)的陷阱;

(b) 时钟已设置为运行，并将在每个时钟周期(Clock Tick)产生一个中断，即一个时钟周期为16.7或20毫秒;

(c) 进程#1即将执行一个“陷阱”指令，它是基于“exec”的系统调用的一部分。

10.2 fuibyte(0814)、fuiword(0844)

“main”同时使用“fuibyte”和“fuiword”；由于前者不重要但是更复杂，我们把它留给读者阅读，并专注于阐述后者。当系统以内核态运行时，它调用“fuiword”(1602)，其中一个参数是用户态地址空间的地址。例程的函数是获取(fetch)相应机器字的值并把它作为结果返回(在r0中左侧)。若发生错误，则返回值为-1。请注意：与“fuibyte”不同，“fuiword”存在一个模糊之处，即返回值-1不一定是错误指示，而是用户态空间中的实际值；请你相信它在“main”中的使用方式无关紧要。此外，代码不区分总线超时错误和段错误。该例程按照如下方式执行：

0846：参数移至r1;

0848：调用“gword”;

0852：当前PS存储在栈中;

0853：优先级提高到7，禁用中断;

0854：位置nofault(1466)的内容保存在栈中;

0855：“nofault”加载“err”程序的地址;

0856：“mfpi”指令用于从用户态空间获取机器字；

如果没有出错，这个值将留在内核栈上。

0857：该值从栈传输到r0;

0876：恢复“nofault”和PS的先前值;

现在，我们假设“mfpi”指令出错，并且发生总线超时。

0856：“mfpi”指令将被中止，程序计数器(PC)将指向下一条指令(0857)，并且把发生通过向量位置4的陷阱;

0512：新的程序计数器(PC)具有“陷阱”的值，新的PS将表示为：

present mode = kernel mode,
previous mode = kernel mode,
priority = 7;

0756：执行的下一条指令是“trap”的第一个指令，它把CPU状态字保存在当前栈顶之外的两个字，但是它在这里是不合适的；

0757：“nofault”包含“err”地址且为非0;

0765：把1移动到SR0，重新初始化存储管理单元(MMU);

0766：“nofault”的内容在栈顶部移动，覆盖此前的内容，即“gword”中的返回地址;

0767：“rtt”不是返回到“gword”，而是返回到“err”的第一个机器字;

0880：“err”恢复“nofault”和PS，跳过返回到“fuiword”；在r0中放置-1，然后直接返回到调用例程。

10.3 中断

假设时钟已中断CPU，两个时钟的向量位置100和104具有相同信息，程序计数器(PC)设置到标记为“kwlp”(0568)位置的地址，PS设置为包含以下英文信息。请注意：无论从向量位置获取的值如何，PS都将包含一个真实的先前模式。

present mode = kernel mode
previous mode =kernel or user mode
priority = 6

0570：向量位置包含新的PC值，该值是标记为“kwlp”语句的地址；该指令是通过r0“调用”的子程序调用；

0802：r1从栈恢复;

0803：从栈中删除PS的副本;

0804：从栈中恢复r0的值;

0805：最后“rtt”指令返回到被中断的内核态程序;

如果此前的模式是用户态，则不能确定中断的例程将立即恢复;

0788：在专用中断例程(在此情况下为时钟)返回后，检查(“runrun> 0”)以便查看是否有比当前进程优先级更高的进程准备就绪；如果决定允许当前进程继续，那么重要的是它不会中断，因为它在“从中断返回”指令之前恢复其寄存器。因此，在测试之前，CPU优先级提高到7(第0787行)，从而确保在用户态恢复之前不再发生中断，然而，另一个中断可能随后立即发生；如果“runrun>0”，那么另一

个更高优先级的进程正在等待，CPU优先级被重置为0，并允许采用任何未决中断；然后调用“swtch”(2178)，允许更高优先级的进程继续进行；当进程从“swtch”返回时，程序循环返回以便重复测试；以上讨论显然扩展到所有中断；唯一与时钟中断有关的部分是调用一个专用例程“clock”。

10.5 用户程序陷阱

系统调用机制允许用户态的程序调用UNIX系统以便获得帮助，它包括用户态的程序执行256个种“trap”指令中的任何一种；“version”是指令字中低阶字节的值。

0518：在用户态的程序中执行陷阱指令导致向量位置34发生陷阱，这导致程序计数器(PC)加载标签“trap”的值(第0512和第0755行)；新设置的进程状态指令PS(Process Status)指示：

当前模式=内核模式；
先前模式=用户模式；
优先级= 7

0756：被执行的下一条指令是“trap”的第一个指令，它把CPU状态字保存在栈中超出当前栈顶的两个字；在更改PS进程状态指令之前，尽快保存PS非常重要，因为它包含定义发生的陷阱类型信息；“move”的某种非常规的目标是提供与中断处理的兼容性，以便能进一步使用相同的代码;

0757：“nofault”为零，因此不执行分支;

0759：内存管理状态寄存器仅在需要时存储，内存管理单元(MMU)被重新定义；

0762：使用r0使子程序条目“调用”；这会把r0的旧值整齐地存在栈中，但不会存储返回地址；新值是接下来要输入例程的地址，在本例中为“trap.c”(2693)文件中的“trap”例程;

0772：把栈指针调整为指向已经包含PS副本的位置；

0773：CPU优先级设置为0;

从这里开始遵循与中断相同的路径。

10.6 内核栈(kernel stack)

C版本中的“trap”过程或图10.1中所示的专用中断处理之一时内核栈的状态。列(2)和(3)分别给出栈字相对于标记为“r0”和“tpc”字的栈位置的位置。第(1)和(2)栏定义或解释文件“reg.h”的内容，参见表26。“dev”、“sp”、“r1”、“nps”“r0”、“pc”和“ps”依次是“trap”(2693)和程序声明中使用的参数名称和时钟“(3725)。

请注意，在进入C版本的“trap”陷阱或其它中断处理程序之前，寄存器r2、r3、r4和r5的值尚未保存在栈中。这些操作是通过调用“csv”(lg20)执行的，它们在每个编译过程开始时由C编译器自动包含。对“csv”的调用形式等同于以下汇编程序指令：

jsr r5，csv

这将r5的当前值保存在栈中，并把它替换为C过程中下一条指令的地址。

1421：r5的这个值被复制到r0中;

1422：栈指针的当前值被复制到r5中;

请读者注意：此时，r5指向包含前一个r5值的栈位置，即它指向一个指针链的开头，每个过程一个线程栈；当C程序退出时，它实际上返回到“cret”(1430)，其中r5的值用于恢复栈，r2、r3和r4用于它们的早期条件，即它们在进入程序之前，因此r5通常被称为环境指针。

11. 时钟中断(Clock Interrupts)

程序“clock”(3725)处理来自线频时钟(型号为KW11-L型，中断向量地址为100)或可编程实时时钟(型号为KW11-P型，中断向量地址为104)的中断；UNIX要求至少其中一个应处于可用状态；如果两者都存在，那么仅使用线频时钟。无论使用哪个时钟，系统都以线频产生中断，若电源频率为50Hz，则每20毫秒产生一次中断。时钟中断的优先级为6，高于典型系统上的任何其它外围设备。因此，一旦时钟控制器请求中断，“clock”的启动通常会有很小的延迟。

11.1 clock(3725)

时钟(clock)打破了外围设备处理程序的大多数规则：它确实引用当前的“u”结构且在某些时候以低优先级运行；如果此前的执行尚未完成，那么就缩减其活动； “clock”函数包含以下一般管理(housekeeping)

- 更新显示寄存器（仅限PDP11/45和11/70）；
- 保持各种统计值如时间、累计处理时间和执行情况;
- 按照计划唤醒按固定时间间隔休眠的进程;
- 核心交换活动每秒启动一次。

3740：显示(display)是PDP11/40上的no-op操作;

3743：数组“callout”(0265)是“callo”类型的“NCALL”(0143)结构的”callo”(0260)；“callo”结构包含三个元素：增量时间、参数和函数的地址；当函数元素不为null时，则在指定时间后使用提供的参数执行该函数；对于正在研究的这个系统，以该方式执行的唯一函数是“ttrstrt”(8486)，它是一个句柄，参见第25章;

3748：如果列表的第一个元素为null，则整个列表为空;

3750：“callout”列表按照期望的执行顺序排列；记录时间是事件之间的时钟周期数；除非第一次(下一个事件之前的时间)已经为0，这意味着其执行已到期，否则此时间应减1；如果该时间已被计数为0，则下一次递减，除非它也已为0，即减少列表中的第一非0时间；所有具有0次的前导条目(或项目)表示已到期的操作，这些操作实际上是稍后进行的;

3759：检查以前的处理器数据，如果优先级不为零，那么绕过下一部分，执行那些到期的操作;

3766：将处理器优先级降低到5，现在可能发生其它6级中断;

3767：搜索“callout”数组，查找到期的操作并执行它们;

3773：将尚未到期的操作的条目移动到数组的开头;

3787：无论先前的CPU优先级如何，执行从此处到第3797行的代码，优先级为5或6;

3788：如果此前模式为用户态，则递增用户时间计数器；如果正在累积执行配置文件，那么请调用“incupc”(a895)在用户态的直方图中输入程序计数器(PC)；“incupc”是用汇编语言编写的，大概是为提高效率和方便性；有关其用途的描述能在UPM的“PROFIL(II)”找到，另见程序“profil”(3667);

3792：若此前模式不是用户态，则为进程添加系统内核的时间计数器；刚描述的代码执行系统的基本时间计算；对于某些进程，每个时钟周期会导致“u.u_utime”或“u.u_stime”递增；“u.u_utime”和“u.u_stime”都在“fork”(3322)中初始化为0；它们的值在“wait”(3270)中查询(interrogate)；在32K滴答(约10小时)后，这些值将变为负值！

3795：“p_cpu”用于确定进程优先级，它是一个字符值，始终被解释为0~255正整数；当它移动到特殊寄存器时，将发生符号扩展，因此255就会变为-1；对该值加1，结果为0；对0减1，该值再次为-1，并以无符号255保存。请注意：在引用“p_cpu”的其它位置(2161,3814)，在把值传送到特殊寄存器后，前8位被掩码;

3797：增加“lbolt”，如果超过“HZ”，即超过一秒或更长时间……

3798：如果CPU以前没有以非0优先级运行，那么就要做大量的事务处理;

3800：用“HZ”减少“lbolt”;

3801：增加日累计器的时间;

3803：随后的事件(events)可能需要花一些时间，但是它们可能会被合理地中断以便服务其它外围设备；因此，CPU优先级低于所有设备优先级，即低于4；但是，在过程“clock”的激活完成前，现在或许存在另一个时钟中断；通过把CPU优先级设置为1而不是0，“clock”的第二次激活也不会尝试从第3804行执行代码。请注意优先级1在功能上与优先级0相同;

3804：如果当前时间(以秒为单位测量)等于“tout”中存放的值，那么通过“sleep”系统调用唤醒所有已选择暂停一段时间的进程，即通过过程“sslep”(5979)；“tout”存储下一个进程被唤醒的时间；如果存在多个这样的进程，那么将被扰乱的余数必须在它们之间重置“tout”；此类进程的数量变得很大，虽然这种机制很有效，但是效率不高；在此情况下，需提供类似于“callour”数组(见第3767行)的机制;事实上，合并两种机制有多大困难，不利因素是什么;

3806：当“time[1]”的最后两位为0时，即每4秒重置调度；标记“runrun”并唤醒所有等待“lightning bolt”的东西；“lbolt”代表每4秒钟发生的一般事件以启动杂项管理；它由“pcopen”(8648)使用;

3810：对于所有当前定义的过程：将“p_time”增加到最大值127，它只是一个字符变量；通过“SCHMAG”(3707)减少“p_cpu”但不允许它变为负数；注意，如前所述(第3795行)，“p_cpu”被视为0~255范围内的正整数；若CPU优先级当前设置为抑制值，则重新计算；请注意“p_cpu”通过“setpri”(2156)进入进程优先级计算“p_pri”；“swtch”(2209)使用“p_pri”选择哪个进程，从处于内核(“SLOAD”)和准备运行(“SRUN”)的进程中，接下来应得到CPU的注意；“p_time”用于衡量进程在内核或交换到磁盘的时间长度(单位为妙)；“p_time”由“newproc”(1869)，“sched”(2047)和“xswap”(4386)设置为0； “sched”(1962,2009)使用它确定换入或换出的进程；

3820：如果调度程序正在等待重新排列，那么就将它唤醒；因此，调度决策的正常速率是每秒一次;

3824：如果中断之前的模式是用户态，那么把“r0”的地址存储在标准位置；如果已收到该进程的“signal”，则为合适的行动调用“psig”(4043)；

11.2 timeout(3845)

该进程在’callout“数组中构造新条目；在该系统中，仅从例程“ttstart”(8505)调用它，传递进程“ttrstrt”(3486)；请注意“ttrstrt”调用“ttstart”，这可能会调用“timeour”，这是一种完全不同的关系；此外，请注意：大多数“timeout”在优先级为7时运行，以便避免时钟中断。

12. 陷阱和系统调用

本章涉及系统处理陷阱的方式，特别是系统调用；有许多条件可能导致CPU“trap”陷阱；其中许多是非常明显的错误条件包括硬件或电源故障，并且UNIX不会尝试任何复杂的恢复过程；我们关注的最初焦点是在文件“trap.c”中的主要过程。

12.1 trap(2693)

第10章讨论调用该进程的方式，汇编程序“trap”例程执行某些基本的管理任务(Housekeeping Tasks)设置内核栈；当调用此过程时，一切看起来都是合适的；“trap”进程运行就像通过另一个C过程以正常方式调用以下7个参数：

dev、sp、rl、nps、r0、pc、ps

顺便提一下，这里有一个特殊考虑因素，通常所有传递给C程序的参数都是按值传递(passed by value)的。若程序随后改变其参数的值，则不会直接影响调用过程；若“trap”或中断句柄更改其参数的值，在恢复“先前模式”寄存器时，则提取新值并把它反射回来。

在通过在“trap”陷阱之后，立即捕获CPU状态字的值，并屏蔽除低5位以外的所有值，从而获得“dev”的值；在进行以上操作之前，我们已用一个适当向量位置(地址))中包含的原型设置CPU状态字。如果向量位置的第2个机器字是“br7 + n;”，例如第0516行，那么“dev”的值将是n。

2698：“savfp”保存浮点寄存器，对于PDP11/40，这是一个无操作(no-op);

2700：如果先前模式是用户态，那么通过添加八进制值020修改“dev”的值(2662);

2701：存储r0的栈地址记录在“u.u_ar0”中以供未来参考；随后各种寄存器值能表示为“u.u_ar0 [Rn]”;

2702：现在有一个多路“switch”，它取决于“dev”的值；此时我们可观察到UNIX把陷阱分为三类，具体取决于此前的CPU模式和陷阱的来源：

（a）内核态;
（b）用户态，不是由于“trap”指令造成的;
（c）用户态，由于“trap”指令造成的。

12.2 内核态陷阱

陷阱是出乎意料的且只有一个异常，系统对该异常引发“panic”；执行的代码是“switch”语句中的“default”。

2716：Print(打印)；

- 第7个内核段地址寄存器的当前值，即当前每个进程数据区的地址;
- 地址“ps”，在内核栈中;
- 陷阱类型编号;

2719 “panic”，它没有返回值；浮点运算仅由用户态程序使用，而不是由UNIX操作系统使用；由于PDP11/45和11/70上的浮点操作是异步处理的；当发生浮点异常时，CPU或许已切换到内核态处理中断；因此，偶尔可预期内核态浮点异常陷阱，这是当前用户态程序的关注点；

2793：调用“psignal”(3963)设置一个标志，表明发生浮点异常;

2794：返回，这提出一个有趣的问题：“为什么内核态和用户态的浮点异常处理稍有不同？”

12.3 用户态陷阱

首先考虑一个陷阱，这个陷阱不是由于执行而生成的；它被视为可能的错误，除核心转存(core dump)的可能性之外，UNIX操作系统不提供任何规定；然而，使用用户态的程序本身可能已预料到并为此做好准备。这方面制定和实施的规定是下一章的主题。在此阶段，主要要求是发送信号“signal”说明陷阱已经发生。

2721：系统处于用户态时发生总线错误，将“i”设置为值“SIGBUS”(0123);

2723：“break”导致“switch”语句中的分支出现在第2818行;

2733：除注意到的一个特殊情况外，非法指令的处理在此级别与总线错误相同;

2739：

2743：

2747：

2796：

请参看第2721行的评论，注意“4+USER”(电源故障)和“7+USER”(编程中断)的情况由“default”情况(第2715行)处理。

2810：这表示需要UNIX操作系统辅助扩展用户态栈区的情况；汇编程序例程“backup”(1012)用于重建在执行导致陷阱指令之前存在的情况；“grow”(4136)用于进行实际扩展；“backup”进程并非易事，其理解涉及仔细考虑PDP11架构的各个方面，因此我们把它留给对此感兴趣的读者；正如PDP11/40所述，“backup”或许并不总是成功，因为CPU没有保存足够的信息解决所有可能性；

2818：调用“psignal”(3963)设置适当的“信号”；请注意此声明只能从包含“break”语句的“switch”的情况中获得;

2821：“issig”检查“signal”是否刚刚发送到用户程序，或刚刚在某个较早的时间发送给用户程序且还没有被处理过;

2822：“psig”执行适当的操作；“issig”和“psig”都将在第13章中详细讨论;

2823：重新计算当前进程的优先级。

12.4 系统调用

用户态程序使用“trap”指令作为系统调用机制的一部分调用UNIX操作系统以便获得帮助。由于“trap”指令有许多可能的版本，请求的协助类型可被编码为“trap”指令的一部分，作为系统调用一部分的参数以不同方式从用户态程序传递：

（a）通过特别寄存器r0;
（b）作为程序中嵌入的一组词语, “trap”指令后面的字符串;
（c）作为程序数据区中的一组单词，这是间接调用。

间接系统调用比直接系统调用的开销更大，当参数是数据相关且在编译时无法确定时，就需要间接调用。如果只有一个数据相关参数通过r0传递，那么有时可避免间接调用；在选择哪些参数应通过r0传递时，系统设计者会以经验为指导，因为这种模式不符合最小的惊人规律。C编译器不会对系统调用给出特殊的识别，但会以与其它过程相同的方式处理它们。当加载器解决未确定的引用时，它使用包含实际“trap”指令的库例程满足它们。

2752：错误指示灯复位;

2754：检索导致陷阱的用户态指令，除最不重要的6位之外，所有指令都被屏蔽掉；结果用于从结构数组中选择一个条目“sysent”，所选条目的地址存储在“callp”中;
2755：零系统调用(zeroth)是一个间接系统调用，其中传递的参数实际上是系统调用参数序列的用户程序数据空间中的地址；注意“fuword”和“fuiword”的单独使用；两者之间的区别在PDP11/40上是不重要的，但是，最重要的是在具有单独的“i”和“d”地址空间的机器；

2760：“i=077”模拟最后一次系统调用(2975)上的调用，这导致对“nosys”(2855)的调用并导致错误条件，这对于用户态通常是致命的程序;

2762：

2765：“sysent”指定的参数数量是用户态程序员提供的实际数量；如果通过r0传输一个参数，那么该数量减去1；参数从用户数据区或指令区复制到五元素阵列“u.u_arg”中；从“sysent”(表29)看来，“u _area[]”似乎有4个元素就足够 - 这是为将来的扩充留有余地吗;

2770：第一个参数的值被复制到“u.u_dirp”中，它似乎主要用作方便的临时存储位置;

2771：使用所需系统例程的地址调用“trapl”，请注意第2828行的注释;

2776：发生错误时，设置旧CPU状态字中的“c位”，参见第2658行，并通过r0返回错误编号。

12.5 系统调用处理程序

读者可在表29中的文件“sysent.c”中查看全套完整的系统调用，但更相关的是，这些将在UPM手册的第II部分中详细讨论。处理系统调用的过程主要在文件“sysl.c”、sys2.c“、sys3.c”和“sys4.c”中找到，其中两个重要的普通过程是“nullsys”(2855)和“nosys”(2864)，它们位于文件“trap.c”中。

12.6 文件’sysl.c'

该文件包含5个系统调用的过程，其中3个立即被考虑，2个(“rexit”和“wait”)被推迟到第10章；此文件中的第一个过程也是我们遇到的第一个系统调用“exec”。

12.7 exec(3020)

该系统调用#11执行一个程序的进程改变为执行不同程序的进程，请参见UPM手册的EXEC(II)，这是最长的系统调用之一。

3034：“namei”(6618)把第一个参数(它是指向定义新程序名称的字符串的指针)转换为“inode”引用，“inode”是文件引用机制的基本部分，我们将在第19章中详细讨论“namei”(6618);

3037：等待当前“exec”的数量太大，参见第3011行的评论;

3040：“getblk(NODEV)”导致从缓冲池中分配512字节缓冲区，该缓冲区暂时用于存储在核心(core)，即当前在用户数据区(ppda)中的信息以及启动新程序所需的信息；请注意，“u.u_arg”中的第2个参数是指向此信息的指针;

3041：如果文件不可执行，那么“access”返回非0结果；第二个条件检查文件是目录还是特殊字符文件；似乎通过提前进行此测试，例如在第3036行之后，能提高代码的效率;

3052：把用户空间中的参数集复制到临时缓冲区中;

3064：如果参数字符串太大而无法放入缓冲区，那么请退出错误;

3071：如果参数字符串中的字符数为奇数，那么添加一个额外的空字符;

3090：指定文件的前四个机器字(8个字节)被读入“u.u_arg”，这些机器字的解释在3076行开始的注释中表示，更全面的解释在UPM手册的A.OUT(V)中表示；注意准备读取操作的“u.u_base”，“u.u_count”，“u.u_offset”和“u.u_segflg”的设置;

3095：如果文本段不受保护，那么把文本区域大小添加到数据区域大小，并把前者设置为0;

3105：检查程序是否具有纯文本区域，但程序文件已被其它程序作为数据文件打开；如果是这样，那么请退出错误;

3127：当达到这一点时，执行新程序的决定是不可撤销的，即不再有机会返回原始程序并设置错误标志;

3129：“expand”在这里实际上意味着一个主要的收缩，仅限于ppda每个过程数据区;

3130：如果需要，那么“xalloc”负责分配并链接到文本区;

3158：把存放在缓冲区中的信息复制到新程序的用户数据区的栈中;

3186：内核栈的地址(location)设为0，内核栈包括用户态寄存器先前值的副本；除r6和设置在第3155行的栈指针以外;

3194：减少“inode”结构的引用计数;

3195：释放临时缓冲区;

3196：唤醒在第3037行等待的任何其它进程。

12.8 fork(3322)

对“exec”的调用通常在“fork”之前；”fork”的大部分工作是由“newproc”(1826)完成的，但在调用后者之前，“fork”会对“proc”数组中的一个插槽进行独立搜索，并把该位置记为“p2”(3327)；“newproc”也会独立搜索“proc”；据推测它总是找到与“fork”相同的空槽，因为它不会报告值；为什么这一点没有混淆?

3335：对于这个新进程，“fork”返回父进程标识的值并初始化各种统计参数;

3344：对于父进程，“fork”返回子进程标识的值，并用一个字跳过用户态的程序计数器(PC)；请注意最终返回到C程序的值与上面的略有不同，请参阅UPM手册FORK(II)。

ssig(3614)

kill(3630)

12.9 sbreak(3354)

该过程实现系统调用#17，其在UPM手册BREAK(II)中描述；程序开头的注释使一个以上的读者感到困惑：显然标识符“break”在C程序中使用，留下一个封闭的程序循环，其方式与此处的预期完全不同(更改大小)程序数据区。“sbreak”与程序“grow(4136)”有明显相似之处，但与后者不同，它只是明确调用，实际上可能导致数据区的收缩和扩展，这取决于新的理想容量。

3364：计算数据区的新容量(32个字块);

3371：检查新的数据区容量是否与内存分段约束一致;

3376：数据区正在缩小；把栈区向下复制到以前的数据区；调用“expand”释放多余部分;

3386：调用“expand”增加总的区域；把栈区复制到新增部分，并清除该栈此前占用的区域。

在第13章中描述在“SysL.C”中还包括的下列步骤：

rexit (3205)　　wait (3270)

exit (3219)

12.10 文件’sys2.c'和'sys3.c'

“sys2.c”和“sys3.c”主要关注文件系统和I/O，这两个文件位于操作系统源代码的第四部分。

12.11文件’sys4.c'

该文件中的所有过程都实现系统调用，以下过程将在第13章进行描述

ssig(3614)　kill(3630)

由于以下程序很简单，我们留给读者娱乐和编辑：

getswit（3413）sync（3486）
gtime（3420）getgid（3472）
stime（3428）getpid（3480）
setuid（3439）nice（3493）
getuid（3452）times（3656）
setgid（3460）profil（3667）

以下描述与文件系统有关的以下过程：

unlink（3510）
chdir（3538）
chmod（3560）
chown（3575）
smdate（3595）

13. 软件中断

本章主要关注文件“sig.c”的内容，这个文件出现在表39至42中。该文件引入一种IPC进程间通信的机制。特别地，它规定一个用户态的进程被另一个进程中断，而后者的中断原因包括动作中断、转移、终止、错误或操作员的动作。在本次讨论中，我们有意使用术语“软件中断”而不是“信号”。

我们回避后者的原因是：它在UNIX环境中获得新的内涵，这与普通英语的用法有很大的不同。UNIX识别20(“NSIG”, 第0113行)位不同类型的软件中断，其中13个具有标准名称和关联，读者可通过阅读UPM手册的SIGNAL(II)而发现。中断类型#0被解释为“无中断”。

在每个进程的ppda内有一个“NSIG”字的数组“u.u信号”。每个字对应一个不同的软件中断类型，并定义在进程遇到这种软件中断时应采取的操作，具体请参见表54。中断类型＃9(“SIGKILL”)是特别区分的，因为UNIX确保“uu_signal[9]”在进程完成之前保持为0。如果进程因此而中断，它将总是终止自己。

13.1 预期

除了刚提到的“u.u_signal[9]”以外，每个进程能设置数组“u.u_signal[]”的内容。因此，我们能够预期将来的中断并采取适当措施。用户态的程序员通过“signal”系统调用完成这个操作，请参见UPM手册的“SIGNAL(II)”。

例如，如果用户点击其终端上的“delete”键，那么就会产生#2中断；如果程序员希望忽略这类软件中断，那么就应在C程序中执行系统调用把“u.u_signal[2]”设置为1。

“signal(2,1)”

13.2 因果关系

一个“proc”结构的进程包含一个“proc”“p_sig”(0363)条目，我们通过把其值设置为与某个中断相对应的类型编号，即从1到“NSIG”-1之间的一个值，就能很简单地为该进程引起一个中断。

即使受影响的进程及其ppda已被交换到磁盘，我们也总是能直接访问“p_sig”。显然，这种机制进仅允许每个进程在任一时刻仅被中断一次。除非其中一个中断属于#9类型，否则这个最新中断总是优先。

13.3 效果

软件中断不会立即产生影响。如果受影响的进程当前正在运行，那么其影响可能仅在微小延迟之后发生，或者，如果受影响的进程被挂起且已被换出，那么可能在相当长延迟之后发生。中断指示的操作总是由受影响的进程自身造成的，因此仅在受影响进程处于活动状态时发生。

当执行用户定义过程的效果时，内核态进程会调整用户态栈，使其看起来已执行进程并在执行第一条指令前立即中断(硬件风格)，然后系统以一般方式从内核态返回到用户态。经过以上调整后，下一个被执行的用户态指令就是指定过程的第一条指令。

13.4 追踪

UNIX的软件中断工具已扩展为一种机制，它能提供强大但有些低效的进程管理，其中父进程能监视一个或多个子进程的进展。

13.5 程序

与软件中断相关程序之间的相互关系乍一看有点令人困惑，因此在全力投入前简要介绍这些程序是值得的。
.....

13.6 A. 预期

“ssig”(3614)实现系统调用#48(“signal”)以设置数组“u.u_signal”中一个元素中的值。

13.7 B. 因果关系

“kill”(3630)实现系统调用#37(kill)，它对由其进程标识号定义的进程引起一个指定的中断。

“signal”(3949)对从指定终端控制和/或启动的所有进程造成一个指定的中断。

对“psignal”(3963)实现对“p_sig”的设置，而“kill”(3649)、“singal”(3955)、“trap”(2793,2818))和pipe“(7828)”实现对“psignal”的调用。

13.8 C 效果

“sleep”(2085)、“trap”(2821)和“clock”(3826)调用“issig”(3991)，询问是否存在应用正在等待发生的活动进程的突出和不可忽略的软件中断。

“psig”(4043)被调用以实现中断触发的动作，它是在无论何时“issig”返回一个非0结果条件下条用的；一个例外是“sleep”，因为它稍微复杂一些。

“core”(4094)被“psig”调用，其调用条件是一个核心转储(core dump)被一个中断进程指示。

“grow”(4136)被“psig”调用以扩大用户态的栈区。

“grow”(4136)由“psig”调用以扩大用户态的栈区，这在在需要时进行的。

“exit”(3219)终止当前的活动进程。

13.9 D 跟踪

“ptrace”(4164)实现#26系统调用。

“stop”(4016)被“issig”(3999)调用，用于追踪被允许的进程，其中这个监督父进程具有“look-see”的作用。

“procxmt”(4204)一个是从“stop”(4028)调用的过程，在父进程的请求下(behest)，子进程执行与追踪相关的某些操作。

13.10 ssig(3614)

这个过程实现“signal”系统调用。

3619：如果中断原因超出范围或等于“SIGKILL”(9)，那么就退出错误;

3623：捕获“u.u_signal [a]”中的初始值，作为系统调用的结果返回;

3624：将“u.u_singal”的元素设置为所需要的值…

3625：如果当前原因的中断未决，那么就请取消它；目前尚不清除这一步骤为什么是必要的、甚至是可取的？有什么建议吗？

13.11 kill(3630)

这个过程实现“kill”系统调用，使指定类型的软件中断另一个指定进程。

3637：如果“a”不为0，那么是识别要中断的进程号的进程；如果“a”为0，那么来自与当前进程相同终端的所有进程都将被中断;

3639：依次考虑“proc”表中的每个条目，如果该项符合以下任何一个条件，那么就拒绝它：

它是当前进程(3640)；
它不是指定进程(3642);
没有指定一个特定进程(“a”==0)，但是它没有相同的控制终端或它是两个初始进程之一(3644)；
用户不是“super user”且用户身份不匹配(3646);

3649：对通过上述测试的任何进程，请调用“psignal”更改“p_sig”。

13.12 信号(3949)

对于每一个进程，如果它由指定的终端控制(用“tp”表示)，那么就用“psignal”命中它。

13.13 psignal(3963)

3966：如果“sig”太大，则拒绝这个调用；但是它不负值，为什么在这种情况下不检查？在把该参数传递给“psignal”前，kill”不检查该参数；无可否认，“kill”命令只导致“sig”的值为正……;

3971：如果“p_sig”的当前值未设置为“SIGKILL”，那么就覆盖它；如果一个进程被彻底杀掉，那么就无法恢复它;

3973：这里似乎有一个错误…对于“p_stat”读取“p_pri”…若效果不太好，则提高过程的优先级;

3975：若进程正在等待非内核事件—即它是有正优先级“sleep”(2066)，则把其设置为再次运行。

13.14 issig(3991)

3997：如果“p_sig”不为0，那么……

3998：如果“tracing”标志打开，则调用“stop”，这个主题将在稍后恢复;
4000：如果“p_sig”为0，则返回值为0；这显然是多余的，但在测试上是必要的，因为“stop”会把“p_sig”重置为副作用;

4003：如果“u.u_signal”的相应元素中的值是偶数(可能为0)，那么就返回非0值;

4006：否则返回0值；有关“issig”调用频率的评论发生在第3983至3985行，因此需要作出一些澄清；对“issig”至少调用一次，每次执行“trap”的一部分，但只有一个中断例程(“clock”，每秒只调用一次“issig”)；当“pri”为正时，“sleep”(2073,2085)在调用“swtch”前后调用“issig”。

13.15 psig(4043)

仅当“issig”发现“u.u_signal[n]”具有偶数值时，才调用这个过程。如果找到该值(4051)为非0，那么把它作为必须执行的用户态函数的地址。

4054：重置“u.u_signal[n]”，但非法指令或跟踪陷阱的中断除外;

4055：计算要插入用户态栈的两个字中较低位的用户空间地址…

4056：调用“grow”以检查当前用户态的栈大小，并在必要时将其向下扩展;

4057：将“trap”或硬件中断(在“时钟”中断情况下)捕获的CPU状态寄存器和程序计数器(PC)的值放入用户栈，并更新r6、r7和CPU状态字的记忆值；返回到用户态后，在指定程序开始时恢复执行；当此进程返回时，恢复最初中断的进程;

4066：如果“u.u_signal [n]”为0，那么对于列出的中断类型，通过“core”程序生成内核镜像;

4079：将值存储在“u.u_arg[0]”中，这个值由记忆值r0的低位字节和“n”组成，记录中断类型以及内核镜像是否被成功创建;

4080：调用“exit”进程以终止自身。

13.16 core(4094)

此过程把可交换程序镜像复制到用户当前目录中名为“core”过程的文件中。对此过程的详细说明必须等到涉及I/O，它在已经涵盖在文件系统这一章的第三节和第四节的材料。

13.17 grow(4136)

此过程的参数“sp”定义应包含在用户态栈的字地址。

4141：如果栈已经扩展得足够远，则简单地返回0值；请注意，此测试依赖于2的补码算法的特性，如果两者都有| SP | > 215和| u.u_size * 64 | > 215，可采取延长栈的决定在这个时刻可能是错的;

4143：计算栈容量递增，它包含新栈点加上20 * 32字边距;

4144：检查这个值实际上为正，即我们没有处理第4141行的测试失败；

4146：检查新栈容量是否与内存的段约束冲突(如果冲突，那么“establishur”就设置“u.u_error”)，并重置段寄存器原型;

4148：获取一个新的放大数据区，把栈段(每次32个字)复制到新数据区的高地址端，并清除当前成为栈扩展的段;

4156：更新栈大小“u.u_ssize”并返回“successful”结果。

13.18 exit(3219)

当一个进程要终止时，就调用这个过程(procedure);

3224：重置“tracing”标志;

3225：把数组“u.u_signal”中的所有值(包括“u.u_signal[SIGKILL]”)设置为1，以便将来执行“psig”后不再执行“issig”;

3227：调用“close”(6643)以关闭进程已打开的所有文件；在大多数情况下，“closing”进涉及到递减引用计数;

3232：减少当前目录的引用计数;

3233：设置进程与任何文本段的连接;

3234：需要一个地址存储“每个进程”信息，直到父进程能查看它；磁盘交换区域中的一个块(256个字)是一个方便的地址;

3237：找到合适的缓冲区(256个字)和… 3238：复制“u”结构的下半部分，进入缓冲区;

3239：把缓冲区写入交换区;

3241：把进程占用的内核空间输入空闲列表；当然，该空间仍在使用过程中，但是，在任何其它进程再次进入空闲列表前，该内核空间停止使用，因此这个操作无法提前完成；读者稍后会看到“getblk”和“bwtite“都能调用”sleep“，在此期间，可能发生各种情况；有鉴于此，如果我们在第3226行后插入以下语句，那么它或许是合理的。

```
expand(USIZE);
```

3243：将进程状态设置为“zombie”，证明完毕；

3245：剩下的代码搜索“proc”数组以便找到父进程并唤醒它，使任何子进程处在监护状态下；如果它们已停止进行追踪，那么就释放它们；最后，对于该进程而言，代码包括最后一次调用“swtch”；在继续考虑跟踪之前，有两个与“exit”密切相关的例程目前能方便地进行处理。

13.19 rexit(3205)

此过程实现退出系统调用#1，它只需收回(salvage)用户提供参数的低位字节并把它保存在“u.u_arg[0]”中。它位于“u”结构的下半部分，即作为“zombie”写入交换区的部分。

13.20 wait(3270)

对于每一个“exit”的调用，应该由一个焦虑的父进程或者祖先进程对“wait”进行一个匹配的调用；实现”wait”系统调用的后一过程的主要作用是使父进程或祖先进程找到并处置“zombie”子进程；“wait”也有一个辅助功能，它寻找已停止跟踪的子进程，这是下一个主要话题。

3277：搜索整个“proc”数组，寻找子进程；如果不存在子进程，则退出错误(第3317行);

3280：如果子进程是一个“zombie”进程，那么就采取以下措施：
- 保存子进程的标识数，并向父进程报告;
- 从磁盘交换区读回256字记录，并释放磁盘交换空间;
- 重新初始化“proc”数组条目;
- 积累各种统计；
- 保存“u_arg [0]”值，并报告回到父进程。

3300：子进程是否处于停止状态；如果是这样，那么就等待跟踪讨论；

3313：如果查找到一个或多个子进程，但没有一个处于“zombie”或“stop”状态，那么就转入“sleep”状态，然后再进行查询。

13.21 根踪

通过软件中断功能的修改和扩展提供跟踪工具，简而言之，如果父进程正在跟踪子进程的进度，那么每当子进程遇到软件中断时，父进程就有机会作为对中断的总响应的一部分进行干预。

父进程的干预可能涉及询问子进程数据区的值，包括ppda每个过程数据区域；在一定约束条件下，父进程也能改变这些数据区内的值。

软件中断来自父进程、用户(通过在终端输入“kill”或“delete”命令)、子进程(通过其指令或其它故障)。子进程与父进程之间的通信是一种正式的方式：

（1）子进程经历软件中断并停止;
（2）等待的父进程发现已停止的子进程(第3301行)，然后恢复，随后……
（3）父进程可执行“ptrace”的系统调用，该系统调用的效果在系统定义结构“ipc”(3939)中为子进程留下一个请求消息;
（4）当子进程被唤醒时，父进程就开始休眠;
（5）子进程在“ipc”中读取消息并对它进行操作，例如把它的一个值复制到“ipc.ip_data中;
（6）当父进程被唤醒时，子进程就开始休眠;
（7）父进程检查该行动的结果，例如“ipc”中的记录;
（8）步骤(3)至(7)可连续重复多次。

最后，父进程可允许子进程继续正常执行，它可能不知道已发生软件中断。

关于追踪设施的讨论载于UPM手册的PTRACE(II)部分，对于“Bug”段落中提到的功能限制列表，我们能添加一下关于效率的评论：

- 应有一种机制把一个块信息(如一次最多256个机器字)从子进程传送到父进程；反之，则不必如此;
- 应有一个适当的协程程序(类似于“swtch”)以便在子进程和父进程之间实现快速的控制转移。

13.22 stop(4016):

若设置跟踪标志(“STRC”, 0395)，则该进程由“issig”(3999)调用。

4022：如果父进程是进程1(即“/etc/init”)，那么就调用“exit”(第4032行);

4023：否则通过“proc”查找父进程…唤醒父进程……宣告自己停止并且……调用“swtch”；注意不要调用“sleep”；读者需要问自己这是为什么;

4028：如果已重置跟踪标志或程序或“procxmt”的结果为真，那么就返回“issig”;

4029：否则重新开始。

13.23 wait(3270) — 续

3301：如果子进程已停止并且...

3302：如果未设置“SWTED”标志，即父进程最近没有通知这个子进程…

3303：为“aidememoire”设置“SWTED”标志，设置“u.u_ar0[R0]”和“u.u_ar0[R1]”，以便把子进程的状态字返回给父进程;

3309：设置“SWTED”标志，这意味着父进程通过至少连续两次等待而不对“ptrace”进行任何干预，对子进程不感兴趣；因此重置“STRC”和“SWTED”标志并释放子进程；注意使用“setrun”而不是“wakeup”补充“swtch”(4027)上的调用。

13.24 ptrace(4164)

该过程实现“ptrace”系统调用#26。

4168：“u.u_arg2]”对应于C程序调用序列中的第一个参数；如果这个参数为0，那么子进程要求父进程执行跟踪，设置“STRC”标志并返回；请注意：这段代码处理子进程在追踪上被要求采取的唯一显式操作；这里并没有给出真正的理由：为什么这个(追踪)行动应由子进程而不是父进程执行；从安全角度来看，最有可能的是，子进程只有在父进程同意时才是可追踪的；如果子进程要求被追踪、但是被父进程忽略，那么子进程可能被无限期地阻塞；或许一个最好的解决方法是仅在父进程和子进程的显式行动后才设置“STRC”标志；

4172：搜索“proc”表，寻找一个停止的进程；匹配这个给定的进程识别号; 当前进程的子进程;

4181：如果它目前正在使用，那么就等待“ipc”结构变得可用;

4183：把参数复制到“ipc”…

4187：重置“SWTED”标志，然后……

4188：让子进程回到“ready to run”状态;

4189：休眠直到“ipc.ip_req”为非正(4212);

4191：提取要返回到父进程的值，检查错误，解锁IPC，并唤醒等待IPC的任何进程；请注意第4182和第4190行的“sleep”在本质上是不同的原因，并且它可通过在第4190和第4213行用“＆ipc.ip_req”替换“＆ipc”实现良好的区分效果；

13.25 procxmt(4204)

该过程由子进程执行，但是其受到父进程在“ipc”结构中留下数据的影响。

4209：如果当前进程错误地设置“ipc.ip_lock”，那么肯定会忽略“ipc”的其余部分；在“stop”(4027)调用“swtch”之后，通过“setrun”上的三个调用之一重启子进程，使“STRC”和“SWTED”标志处于指示的状态；

exit（3254）set wait（3310）reset ptrace（4188）set

4211：存储“ipc.ip_req”值，然后重置后者，接着唤醒父进程，最后按指示选择下一个操作；UPC手册中的PTRACE(II)充分解释以上各种操作，其中一个限定条件是示例1、2和4、5的文档有误，即分别是指令空间“I”和数据空间“D”，而不是数据空间“D”和指令空间“I”。

第三部分 程序交换、IO、硬盘驱动和缓冲控制

第三节涉及主存和磁盘存储器之间的基本I/O操作，这些操作是程序交换活动以及磁盘文件的创建和引用的基础，本节还介绍使用和操作大型(512字节)缓冲区的过程。

14. 程序交换

与其它分时操作系统(timesharing)和多道程序设计系统(multiprogramming)一样，UNIX使用程序交换(program swapping)，也被称为滚进/滚出(rollin/roll-out)，共享几个主进程中主要物理内存的有限资源。

挂起的进程有选择性地被换出(swapping out)，这是通过把数据段(含ppda)写入磁盘交换区实现的；然后把被占用的内存区重新分配给其它进程，这很可能从交换区换入(swapping in)。

大多数换出的决定和所有换入决定都是通过“sched”程序做出的。换入通过基于“swap”(5196)的一个直接调用call(2034)实现，换出通过调用基于“xswap”(4368)一个调用call(2024)实现。

对于那些喜欢思考早期版本OS架构的骨灰级专家而言，最初“sched”直接调用”“swap”而不是“xswap”从而实现进程换出。一个额外的过程(在文件“text.c”找到的几个过程之一)是共享的文本段(text segment)的实现所必需的。

评估文本段特性需要多少额外代码是有益的：“text.c”有四个过程“xswap”、“xalloc”、“xfree”和“xccdec”，它们操纵一个名为“text”的数组结构并在文件“text.h”中声明。 “sysl.c”和“slp.c”还添加附加代码。

14.1 文本段(Test Segment)

文本段是仅包含纯代码和数据的段，亦即在整个程序执行期间保持不变的代码和数据，这样它们能在执行相同程序的多个进程之间共享。

当系统的多个用户同时执行相同程序时，由此产生的空间节省可能相当可观。有关文本段的信息必须存放在一个中心位置，因此存在文本数组，共享文本段的每个程序都把保存指向一个在“u.u_textp”中相应的文本数组元素的指针。

文本段存储在代码文件的开头，第一个开始执行的程序导致在交换区中创建文本段的副本；当随后没有留下引用文本段的程序时，文本段吸收的资源就被释放；只要没有运行程序引用的文本段，就会释放交换区。

这些状态中的每个数字分别由“x_ccount”和“x_count”表示，数字的递减由例程“xccdec”和“xfree”处理；当计数达到0时，它还负责释放资源。只要程序被换出或终止，就会调用“xc-cdec”；只要程序终止，就会通过“exit”调用“xfree”。

14.2 sched(1940)

进程#0执行“sched”，当该进程不等待由它自己启动的输入/输出操作完成时，它会花费大部分时间等待下列情况之一：

A. (runout)没有任何交换过程准备就绪，因此它就什么也不做；我们可通过调用“wakeup”、“newproc”或“expand”调用的“xswap”改变这种情况。

B. (runin)至少有一个进程被交换出来并准备就绪，但它没有超过3秒和/或目前在主存中没有任何进程处于非活动状态或已存在超过2秒；通过“clock”或“sleep”调用时间的流出可以改变这种情况。

当以下任何一种情况终止时：

1958：当CPU以优先级6运行时，时钟不能中断并改变“p_time”的值，搜索准备运行且已被换出最长时间的进程;

1966年：如果没有这样的进程，那么情况A成立;

1976：搜索足够大小的主存以便保存数据段；如果相关文本段也必须存在但目前不在主存中，那么该区域将增加文本段的大小;

1982年：如果有主存有足够大的区域，该计划将转到(branch)“found2”(2031)；请注意程序不处理文本和数据段有足够空间但在主存储器不同区的情况，扩展代码以覆盖这种可能性是否值得;

1990：搜索主存中的进程，但不是调度程序或已锁定进程(即已被换出的进程)，其状态为“SWAIT”或“SSTOP”但不是“SSLEEP”—即该进程正在等待优先级较低的事件或在跟踪期间已停止，请参阅第13章；如果找到这样的进程，请转到第2021行，以便换出镜像；请注意：对于“proc”条目在“proc”数组中较早的进程，这里似乎存在偏差;

2003：如果要换入的进程镜像出现的时间少于3秒，那么情况B成立;

2005：搜索已加载但不是调度程序或已锁定进程，其状态为“SRUN”或“SSLEEP”，即准备运行，或等待高优先级事件，并且已在主存中最长的时间;

2013：如果要换出的进程镜像在主存中的存储时间不到2秒，那么情况B成立；这里的常数2(也就是2003行的3)有点武断；由于某些原因，程序员已偏离UNIX中通常命名这些常数的做法以强调它们的起源;

2022：过程镜像被标记为未加载，并使用“xswap”换出(4368)；请注意：此处未设置“SSWAP”标志，因为换出的进程不是当前进程，参见第1907和第2286行;

2032：如有必要，将文本段读入主存，请注意交换过程的参数包括：

- 磁盘交换区域内的地址;
- 主存储地址(32字块的序数);
- 大小，要转移的32个字块的数量；
- 方向指示器（“B READ == 1”表示“磁盘到主存”);

2042：交换数据段并……

2044：把磁盘交换区释放到可用列表，记录主存地址，设置“SLOAD”标志，并重置累计时间指示器。

14.3 xswap(4368)

4373：如果未提供“oldsize”数据，请使用存储在“u”中数据段的当前大小;

4375：在磁盘交换区中找到进程数据段的空间；请注意磁盘交换区域按512个字符块分配;

4378：“xccdec”(4490)被条件地调用以，便减少与文本段相关联的计数，该计数引用该文本段在主存中进程的数量；如果计数变为0，那么文本段占用的主存区将简单地返回到可用空间；没有必要把它复制出来，因为正如我们将要看到的，磁盘交换区域中已存在一个副本;

4379：正在换出过程时设置“SLOCK”标志，这是为防止“sched”尝试换出已处于换出过程中的过程；如果交换最初是由一些例行程序开始的，而不是“sched”，例如“expand”;

4382：释放主存镜像，除非“newproc”调用“xswap”;

4388：如果设置“runout”，“sched”正在等待“交换”的东西，因此将它唤醒。

14.4 xalloc(4433)

当启动新程序时，“exec”(3130)调用“xalloc”处理文本段的分配或链接；参数“ip”是指向代码文件“模式”的指针;在此调用时，“u.u_arg [1]”包含文本段大小，它以以字节为单位。

4439：如果没有文本段，那么就立即返回;

4441：通过“text”数组查看未使用的条目和文本段的条目；如果能找到后者，那么进行簿记并转到“out”(4474);

4452：安排把文本段复制到磁盘交换区域；初始化未使用的文本，并在磁盘交换区获取空间;

4459：将进程占用的空间更改为足以包含ppda和文本段的空间;

4460：在读取代码文件之前，需要调用“estabur”设置用户态的段寄存器;

4461：UNIX进程一次只能启动一个I/O操作；因此，可把I/O参数存储在“u”结构中的标准位置，即“u.u_count”、“u.u_offset[]”和“u.u_base”;

4462：八进制值020(十进制16)是代码文件的偏移量;

4463：把信息读入用户态地址空间中从0开始的区域;

4464：把代码文件的文本段读入到当前数据段; 值得注意的是，只要情况开始变得复杂，处理文本段的代码就很保守；例如，当没有更多文本条目可用时，“panic”(4451)似乎是一种很极端的反应；然而，对文本数组空间慷慨的策略很可能比更好所需的代码开销更小，关键是读者怎么看?

14.5 xfree(4398)

“xfree”由“exit”调用(3233)，当进程被终止时及当进程被变形时，由“exec”(3128)处理进程。

4402：将“proc”条目中的文本指针设置为“NULL”;

4403：如果它现在为0，那么就减少主存储器数量……

4406：如果文本段未被标记为保存…

4408：放弃磁盘交换区中文本段的镜像;

4411：调用“iput”(7344)减少“inode”引用计数，并在必要时删除它。

“ISVTX”(5695)定义UPM手册的CHMOD(I)中提到的“粘性位”掩码；如果设置了此位，即使没有正在运行的程序引用它，那么允许文本段的磁盘副本保留在磁盘交换区中，期望很快将再次需要它；这是常用程序如“shell”或编辑器。

15. 基本I/O

在对UNIX系统的I/O主题进行详细描述之前，读者需要完全理解有三个文件的内容。

15.1 文件'buf.h'

该文件对“buf”(4520)和“devtab”(4551)结构进行申明；结构“buf”的实例被声明为’bfreelist(4567)，并作为数组“BUF”(!)(4535)的“NBUF”元素。

结构“buf”可能被错误命名，因为它实际上是缓冲区头或缓冲区控制块；适当的缓冲区域分别赋值并声明(4720)为：

“char buffers[NBUF] [514];”

从“buf”数组到“buffers”数组的指针由过程“binit”设置。

结构“buf”的其它实例被称为“swbuf”(4721)和“rrkbuf”(5387)，没有514个字符缓冲区与“bfreelist”或“swbuf”或“rrkbuf”相关联。

“buf”结构可分为三个部分：
（a）标志(flags)： 这些传达状态信息并包含在一个机器字中，用于设置这些标志的掩码在第4572至4586行中被定义为“B WRITE”、“B READ”等。
（b）列指针(list pointer)：两个双向链表的前向和后向指针，我们分别称为“b”列表和“av”列表。
（c）I/O参数：它是与实际数据传输相关的一组值。

15.2 devtab(4551)

“devtab”结构包括5个机器字，其中最后4个是前向和后向指针。对于每种块类型的外围设备，在设备处理程序中声明“devtab”的一个实例；在我们该模型系统中，唯一的块设备是RK05磁盘，“rktab”在5386行被声明为“devtab”结构；“devtab”结构包含设备的一些状态信息，并用于以下表头：

（a）与设备相关联的缓冲区列表，同时在“av”列表中;
（b）该设备的未完成I/O请求列表。

文件“conf.h”声明：
- 另一种是把整数分解为两个部分的方法(“d_minor”和“d_major”)；请注意“d_major”对应于“hibyte”(0180);
- 两个结构数组;
- 两个整型变量“nlkdev”和“nchrdev”。

两个结构数组“bdevsw”和“cdevsw”在“conf.h”中被声明但未标注或初始化；这些数组的初始化在文件“conf.c”中执行。

15.4文件'conf.c'

该文件与“low.s”一起在每次安装时(通过程序“mkconf”反映实际安装的外设集)单独生成；在我们的例子中，“conf.c”反映我们模型系统的代表性设备。）

该文件初始化以下内容：

bdevsw（4656）swapdev（4696）
cdevsw（4663）swplo（4637）
rootdev（4635）nswap（4698）

15.5 系统生成

UNIX安装中的系统生成主要包括：

- 使用适当的输入运行“mkconf”；
- 重新编译输出文件，被创建为“c.c”和“l.s”；
- 使用修订的目标文件重新加载系统。

这个过程在UNIX中仅需几分钟，而在一些操作系统中需要几个小时；请注意“bdevsw”和“cdevsw”在其它地方的“conf.c”中定义不同，即作为指向返回整数值的函数指针的一维数组。这悄然忽略这样一个事实，例如，“rktab”不是一个函数，并且依赖于链接程序而不是过于仔细地查询它正在执行的工作性质。

15.6 swap(5196)

在深入研究文件“bio.c”的所有细节前，检查前面介绍的一个例程即“交换”既是有益的，又是方便的。缓冲区“swbuf”被声明为控制交换I/O，它必须与其它活动共享对磁盘的访问；没有缓冲区元素与“swbuf”相关联；相反，程序占用或被占用的核心区用于数据缓冲区。

5200：为了方便和经济性，“swbuf”中的标志地址被转移到寄存器变量“fp”；

5202：测试“B_BUSY”标志，如果它已打开，那么交换操作已经进行，因此设置“B_WANTED”标志，并且该过程必须通过“sleep”调用等待；请注意行5202到5205上的代码循环以优先级6运行，即比磁盘中断优先级高1；你能明白为什么这是必要的吗？在什么条件下设置“B_BUSY”标志？

5206：标志设置为反映：
- “swbuf”正在使用中(“B_BUSY”);
- 物理I/O意味着与用户数据段之间的大量传输(“B_PHYS”);
- 操作是读还是写(“rdflg”是swap的一个参数);

5207：初始化“b_dev”字段；据推测，它可能在初始化期间执行过一次，而不是每次使用“swbuf”时，即在“binit”中执行;

5208：“b_wcount”被初始化，请注意其中的负值和被32相乘的方法;

5210：硬件设备控制器需要一个完整的物理地址(PDP11/40上的18位)； 32字块的块号必须转换成两部分：低地址10位向左移6位并存储为“b_addr”，其余6位高地址存储为“b_xmem”；在PDP11/40和PDP11/45上，6位种仅有2位是重要的;

5212：乍一看！把“swapdev”向右移动8个位置以获取主设备编号；使用结果索引“bde-vsw”；从这样选择的结构中，提取策略例程并以“swbuf”的地址作为参数执行；

5213：解释为什么这个“spl6”的调用是必要的；

5214：等到I/O操作完成，请注意“sleep”的第一个参数实际上是“swbuf”的地址；

5216：如果有的话，唤醒正在等待“swbuf”的进程；

5218：把进程或优先级重置为0，从而允许任何挂起的中断“happen”；

5219：重置“B_BUSY”和“B_WANTED”标志。

15.7 竞争条件

“swap”的代码有许多有趣的特性，特别是当几个进程一起运行时，它在微观世界中显示竞争条件的问题，请考虑在5206行设置的情形：

— 在进程A中没有进行交换；
— 进行交换操作，最初用“flags”表示“swbuf.b_flags”；
— flags == null；
— 进程A没有在第5204行延迟，启动其I/O操作并在第5215行进入休眠状态，我们现在有：
flags == B_BUSY | B_PHYS | rdflg

假设在I/O操作正在进行时，进程B也启动交换操作，它也是从执行”“swap”开始的，但是发现设置“B_BUSY”标志，因此它设置“B_WANTED”标志(5203)并且也进入休眠状态(5204); 我们现在有

flags == B_BUSY | B_PHYS | rdflg | B_WANTED

最后，I/O操作完成，进程C接受一个中断并执行“rkintr”，它调用(5471)“iodone”，而后者调用(5301)“wakeup”，以便唤醒进程A和进程B；“iodone”也设置“B_DONE”标志并重置“B_WANTED”标志，如下所示：

flags == B_BUSY | B_PHYS | rdflg | B_DONE

接下来会发生什么？这取决于重新激活进程A和进程B的顺序；因为它们具有相同的优先级“PSWP”，哪个进程先执行类似于一个掷硬币(toss-up)的随机过程；

示例(a)：进程A先行，设置“B_DONE”，因此不再需要休眠；“B_WANTED”被重置，因此没有唤醒。过程A整理(5219)，并留下“swaps”与

flags == B_PHYS | rdflg | B_DONE

进程B现在运行且能够立即启动其I/O操作。

示例(b)：进程B先行，它发现“B_BUSY”开启，因此重新打开“B_WANTED”标志，再次进入休眠状态，离开

flags == B_BUSY | B_PHYS | rdflg |
B_DONE | B_WANTED

进程A再次启动，如通情况(a)一样，但是，这次发现“B_WANTED”，所以除其它杂项之外，它必须调用“wakeup”(5217)；进程B终于被再次唤醒，整个调用链完成。

示例(b)显然比示例(a)效率低得多，这似乎是对5215行的简单改变

```
sleep(fp, PSWP-1);
```

因此，示例(a)几乎没有任何开销，并确保案例(b)从未发生过！

读者应研究在各个点上的情况以便提高CPU优先级的必要性：例如，如果省略行5201且当进程A的操作发生在I/O完成中断时，进程B刚刚完成第5203行，然后“iodone”关闭“B_WANTED”，并在进程B即将进入休眠之前执行“wakeup”……永远！这无疑是一个糟糕的场景。

15.8 可重入

还要注意上面的假设，即进程A和进程B能同时执行交换；所有UNIX进程通常都是可重入的(re-entrant)，这意味着系统能同时执行多个进程。 如果不允许重新入，UNIX系统将如何改变?

15.9 对于未初始化

我们现在回来完成第8章中关于“aretu”和“u.u_ssav”中提出的调用链事宜：在设置“uu_ssav”(2284)以后，“expand”调用(2285)“xswap”，后者调用(4380)“swap”，接着后者调用call(5215)“sleep”，然后后者调用(2084)“swtch”，再接着后者重置“uu_rsav”(2189)。因此，实际上“u.u_rsav”最终被重置为适合于比“u.u_ssav”更深的四个过程调用的值。

15.10 补充阅读

UNIX之父丹尼斯·里奇撰写的文章“The Unix I/O System非常贴切。

16. RK磁盘驱动程序

RK磁盘存储系统采用包含单个磁盘的可移动磁盘盒，该磁盘盒安装在具有移动的读/写磁头驱动器内。指定为RK11-D的设备由一个磁盘控制器和一个驱动器组成。我们指定额外的驱动器为RK05，总共7个，能添加到单个RK11-D中。

对超过8个驱动器的要求将需要具有不同UNIBUS地址集的附加控制器。此外，必须修改文件“rk.c”中的代码以便处理两个或更多控制器的情况。这种情况最不可能，因为对于大量在线磁盘存储的要求将更经济地提供，例如，由RP04磁盘系统。

图表(略)

平均总访问时间为70毫秒。对于多驱动子系统，一个驱动器的搜索可能与另一个驱动器的读取或写入重叠。但是，UNIX不使用此功能，因为硬件控制器中一次存在的错误。

在启动数据传输时，设置RKDA、RRBA和RKC，然后设置RKCS。完成后，状态信息在RKCS、RRER和RKDS中获得。发生错误时，UNIX只需调用“deverror”(2447)即可在系统控制台上显示RKER和RKDS，而无需进行任何分析。在设备驱动程序报告错误之前，操作最多重复十次。

“PDP11外设手册”中完整描述的寄存器格式在几个点上反映在程序代码中，以下摘要足以描述UNIX使用的功能：

16.1 文件’rk.c'

该文件包含特定于RK磁盘系统的代码，即RK“设备驱动程序”。

16.2 rkstrategy(5389)

例如，策略程序的功能是处理读取和写入请求，“swap”(5212)调用这个策略例程。

5397：除PDP11/70系统之外，这里的测试和调用“mapalloc”是“no-op”;

5399：从这里到第5402行的代码似乎是不必要的绕弯子！请参阅下面的“rkaddr”讨论；如果块编号太大，那么就设置“B_ERROR”标志并报告完成;

5407：把缓冲区链接到控制器的FIFO列表，该列表是单链接的，使用“buf”结构的“av_forw”指针，并在“rktab”中有头尾指针；第一步后可能不允许来自磁盘设备的中断;

5414：如果RK控制器当前未激活，则通过“rkstart”(5440)上的调用将它唤醒，该调用检查有事可做some thing to do(5444)，把控制器标记为busy(5446)并调用ters：“devstart”(5447)作为参数指针传递给第一个入队缓冲区头；RKDA磁盘地址寄存器的地址，传递的值实际上是0177412，参见第5363行和第5382行；由“rkaddr”计算的磁盘地址; 0，这在我们的讨论中并不重要并可能会被忽略；

16.3 rkaddr(5420)

此过程中的代码包含一个特殊功能，用于扩展到多个磁盘驱动器的文件。UPM手册中的RK(IV)中描述了这个功能，但是它的作用似乎受到限制；“rkaddr”返回的值被格式化为直接传输到控制寄存器RKDA。

16.4 devstart(5096)

调用RK磁盘时，此过程会依次把适当的值加载到寄存器RKDA、RKBA、RKWC和RKCS中，在此阶段只需要计算最后一个值。虽然外观凌乱，但是计算是直截了当的。请注意“hbcom”为0、“rbp->bxmem”包含物理内核地址的两个高位；RKCS的加载初始化磁盘控制器，即操作现在完全在硬件的控制下。

“devstart”返回“rkstart”(5448)、返回“rkstrategy”(5416)，重置CPU优先级并返回“swap”(5213)，…

16.5 rkintr(5451)

调用此过程处理RK磁盘操作完成时发生的中断。

5455：检查误报！

5459：如果设置，那么检查错误位…

5460：调用“deverror”(2447)在系统控制台终端上显示一条消息;

5461：清除磁盘控制器的内部寄存器和……

5462：等到这个完成，通常是几微秒(milisecond);

5463：如果操作重试次数少于十次，请调用“rkstart”再试一次，否则放弃并报告错误;

5469：将重试计数设置回0，从“actf”列表中删除当前操作，并通过调用“iodone”完成操作;

5472：这里无条件地称为“rkstart”。如果不需要调用，因为“actf”列表为空，那么“rkstart”将立即返回(5444)；

16.6 iodone(5018)

此例程主要涉及块I/O操作完成时的资源返回，它有以下三个操作：

- 释放Unibus地图，适用于11/70，如果适用;
- 设置“B_DONE”标志;
- 若I/O是异步则释放缓冲区，否则重置“B_WANTED”标志并唤醒等待I/O操作完成的任何进程。

17 缓冲区操作

在本章中，我们将详细介绍文件“bio.c”，它包含用于操作缓冲区头和缓冲区的大多数基本例程(4535,4720)。各个缓冲区标题由设备编号“b_dev”(4527)和块编号“b_blkno”(4531)标记；注意后者被声明为无符号整数。缓冲区标题可同时链接到两个列表中，“av”列表和各种“b”列表都是双重链接，以便于在任何时候插入和删除。
“b”列表： 每个设备控制器一个列表，它将与该设备类型相关联的缓冲区链接在一起;
“av”列表：这是一个缓冲区列表，可从当前使用中分离并转换为备用；

17.1 标志

如果暂时从“av”列表中撤回缓冲区，则会引发其“B_BUSY”标志。如果缓冲区的内容正确反映存放或应存方在磁盘上的信息，那么会引发“B_DONE”标志。如果引发“B_DELWRI”标志，那么缓冲区的内容比相应磁盘块的内容更新，因此必须先写入缓冲区才能重新分配。

17.2 类似缓存的内存

读者能看出UNIX对大缓冲区操作方式类似于连接到计算机主存硬件高速缓存的操作，例如PDP11/70中实现方式。缓冲区不会分配给任何特定的程序或文件，除非每一次只有非常短的间隔。以这种方式，开发者能在大量程序和文件之间有效地共享相对少量的缓冲区。信息留在缓冲区中直到需要缓冲区，即如果最近只有部分缓冲区被改变，那么就避免立即“write through”。

读取或写入与缓冲区大小相比较小的记录的程序不会被过度惩罚。最后，当程序终止且文件关闭时，请确保程序缓冲区被正确刷新的问题，这个问题已被解决。有一个须实际考虑的领域：如果决定“何时写入”仅留给操作系统，那么一些缓冲区可能不会被写出很长时间。因此，有一个实用程序，它每分钟运行两次并强制无条件地写出所有这些缓冲区。这能限制突然发生的系统崩溃可能造成的损坏

17.3 clrbuf(5038)

这个例程将缓冲区的前256个字(512字节)清0。请注意：传递给“clrbuf”的参数是缓冲区头的地址，“clrbuf”由“alloc”调用(6982)。

17.4 incore(4899)

此例程搜索已分配给特定(设备和块号)对的缓冲区，它搜索循环“b”-list，其头部是设备类型的“devtab”结构；如果找到缓冲区，那么返回缓冲区头的地址；“incore”被称为“breada”(4780,4788)。

17.5 getblk(4921)

该例程执行与“incore”相同的搜索，但进一步的是，如果初始搜索不成功，则从“av”列表(可用列表)分配缓冲区；通过调用“notavail”(4999)，缓冲区从“av”列表中删除并标记为“B_BUSY”；“getblk”对其参数更加怀疑而不是“incore”。它被称为
exec（3040）

exit（3237）
bread（4758）
breada（4781,4789）
smount（6123）
writei（6304）
iinit（6928）
alloc（6981）
free（7016）
update（7216）

4940：此时，通过搜索“b”列表找到所需的缓冲区或者是“B_BUSY”，在这种情况下必须调用“sleep”(4943)，否则它被占用(4948);

4953：如果找不到所需的缓冲区，如果“av”-list为空，则为“av”-list设置“B_WANTED”标志并转到“sleep”(4955);

4960：如果“av”-list不为空，那么选择第一个成员；如果它代表“延迟写入”，那么安排将其异步写出(4962);

4966：“B_RELOC”是在UNIX开发过程中遗留的(见4583);

4967：此处的代码直到4973无条件地从“b”列表中删除缓冲区以获取它的当前设备类型，并把它重新插入到新设备类型的bn列表中；由于这通常是“no-op”，即新旧设备类型将是相同的，因此似乎需要插入测试：

if（bp-> b_dev == dev）

在执行第4967至4974行之前。

请注意“dev == NODEV”(-1)的调用的特殊处理；这些调用在没有第2个参数的情况下完成，参见例如3040。“bfreelist”用作“NODEV”的“b”列表的“devtab”结构。

17.6 brelse(4869)

该过程将缓冲区作为参数传递，并把它链接回“av”-list；任何等待特定缓冲区或任何可用缓冲区的进程都会被唤醒；请注意：由于两个进程都在等待，因此两个“sleep”(4943,4955)具有相同的优先级。请注意，由于两个“休眠”(4943,4955)具有相同的优先级，如果两个进程正在等待—一个用于特定缓冲区，一个用于任何缓冲区，这是易如反掌的。

通过给予第二优先权(例如通过偏向一个)，应更令人满意地解决竞争。这种改变的缺点是它可能在某些特殊情况下导致僵局。如果发生错误，例如在将信息读入缓冲区时，缓冲区中的信息可能不正确。第4883行的赋值确保随后不会错误地检索缓冲区中的信息。例如，设置“B_ERROR”标志。通过“rkstrategy”(5403)和“rkintr”(5467)。如果要了解为什么发生这种情况，那么请考虑在磁盘I/O操作完成时缓冲区发生的情况：

5471 “rkintr”调用“iodone”;

5026 “iodone”设置“B DONE” flag;

5028 “iodone”调用“brelse”;

4387“brelse”重置“B_WANTED”，“B_BUSY”和“B_ASYNC”标志，但不重置“B_DONE”标志;

……

4948“getblk”找到缓冲区并调用“notavail”; 5010“notavail”设置“B_BUSY”标志;

4759“bread”调用“getblk”，找到“B_DONE”标志设置并退出。

请注意：缓冲区标题通过“notavail”从“av”-list中删除，并由“brelse”返回；缓冲区标题通过“getblk”从一个“b”列表移动到另一个列表。

17.7 binit(5055)

该过程由“main”(1614)调用以初始化缓冲池。设置空双向链接的循环列表：

- 对于“av”列表，“bfreelist”是其头部;
- “b”列表：表示空设备(“dev == NODEV”)，“bfreelist”又是头部;
- 每个主要设备类型的“b”列表。

对于每个缓冲区：
- 缓冲区标题链接到设备“NODEV”(-1)的“b”列表中;
- 缓冲区的地址在标题中设置(5067);
- 缓冲区标志设置为“B_BUSY”，这似乎不是必需的(5072);
- 缓冲区标题通过“brelse”(5073)调用链接到“av”列表;

块设备的数量记录为“nblkdev”，用于检查“getblk”(4927)，“getmdev”(6192)和“openi”(6720)中“dev”的值；检查“bdevsw”(4656)表明“nblkdev”将被设置为8，而值1才是真正需要的；这个结果可通过“编辑”获得如下：

```
/ 5084 / m / 5081 /“nblkdev = i;
/ 5083 / m / 5077 /“i ++
```

17.8 bread(4754)

这是从块设备读取的标准过程，它被以下函数调用。“getblk”找到一个缓冲区，如果设置“B_DONE”标志，则不需要I/O。

wait（3282）
breada（4799）
statl（6051）
smount（6116）
阅读（6258）
writei（6305）
bmap（6472,6488）namei（7625）

iinit（6927）
alloc（6973）
ialloc（7097）
iget（7319）
iupdat（7386）
itrunc（7426,7431）

17.9 breada(4773)

与“bread”相比，这过程有一个附加参数，它仅由“readi”(6256)调用。

4780：检查是否已把所需块分配给缓冲区；它可能还不能使用，但至少它是存在的;

4781：如果没有启动必要的读操作，那么就不等待它完成;

4788：四处寻找“read ahead”预读块；如果它不存在，那么就分配buffer(4789)；如果缓冲区已准备，那么就释放它(4791);

4793：“read ahead”预读块未就绪，因此启动异步读操作;

4798：如果缓冲被分配，那么就给当前块调用“bread”包装它，否则……

4800：等待从第4785行开始的操作完成。

17.10 bwrite(4809)

这是写入块设备的标准过程。它被以下过程调用
“”exit”(3239)、“free”(7021),
“bawrite”(4863), “update” (7221)
“getblk”(4963)、“iupdat” (7400)
“bflush”(5241)

请注意“writei”调用“bawrite” (6310)!

4820：如果未设置“B_ASYNC”标志，则在I/O操作完成之前，程序不会返回;

4823：如果设置“B_ASYNC”，但未设置“B_DELWRI”(注意“flag”在4816行设置)，那么就调用“geterror”(5336)检查错误标志；如果设置“B_DELWRI”且出现错误，那么把错误指示发送到正确的过程是很困难的；“geterror”上的调用(4824)仅为报告与写入操作启动相关的错误。

17.11 bawrite(4856)

该过程由“writei”(6310)和“bd-write”(4845)调用；“writei”调用“bawrite”或“bdwrite”，具体取决于要写入的块是完全还是部分填充。

17.12 bdwrite(4836)

该过程由“writei”(6311)和“bmap”(6443,6449,6485,6500和6501!)调用。

4844：如果设备是磁带驱动器，请不要延迟写入...保持一切顺序;

4847：设置“B_DONE”、“B_DELWRI”标志，然后调用“brelse”，把缓冲区链接到“av”列表。

17.13 bflush(5229)

该过程由“update”(7201)调用，其由“panic”(2420)、“sync”(3489)和“sumount”(6150)调用。“bflush”在“av”列表中搜索延迟写入块并强制它们异步写出。

请注意当“notavail”调整“av”列表的链接时，在遇到每个延迟写入块之后重新启动搜索，在CPU优先级6运行；此外，请注意，由于“bflush”仅通过“update”调用，“dev”等于“NODEV”，因此可在第5238行进行简化。

17.14 physio(5259)

调用该例程来处理“原始”输入/输出，即忽略正常512字符块大小的操作。“physio”被“rkread”(5476)和“rk-write”(5483)调用，它们在数组“cdevsw”(4684)中显示为条目。

原始I/O不是UNIX的基本功能，对于磁盘设备，它主要用于复制整个磁盘并检查整个文件系统的完整性，参见例如UPM手册的ICHECK(VII)，这里能方便地读取整个磁道，而不是单个块时间。

注意“strat”的声明(5261)。由于使用实际参数，例如“rkstrategy”(5389)没有返回任何价值，这种形式的声明真有必要吗?

第四部分 文件和管道

第四部分包括文件和文件系统，文件系统是组织在单个存储设备如磁盘包上的一组文件和关联的表和目录。本部分介绍创建和访问文件，通过目录查找文件以及组织和维护文件系统的方法，它还包括一个称为“pipe”的新颖文件的代码。

18. 文件访问和控制

每种操作系统的很大一部分都与数据管理和文件管理有关，UNIX也不例外。第四部分源代码包含13个文件，前4个包含各种其它例程所需的通用声明：

“file.h”描述文件数组的结构;
“filsvs.h”描述已安装文件系统超级块的结构;
“ino.h”描述安装设备上记录的“inodes”的结构;
“inode.h”描述“inode”数组的结构;

接下来的两个文件“sys2.c”和“sys3.c”包含系统调用的代码。“sys1.c”和“sys4.c”在第2节中介绍。接下来的五个文件“rdwri.c”、“subr.c”、“fio.c”、“alloc.c”和“iget.c”共同提供文件管理的主要例程，并在面向I/O的系统调用和基本的I/O例程之间提供一个链接。文件“nami.c”包括搜索目录以将文件路径名转换为“inode”引用。最后，在概念上，“pipe.c”是管道的(虚拟)设备驱动程序。

18.1 文件特征

UNIX文件在概念上是一个命名的字符串，存储在任何一个外设或主存中，并能通过适用于通常外围设备的机制访问。请注意UNIX文件没有相关的记录结构，但是，在文件中插入换行(newline)字符定义类似于记录的子串。UNIX允许有效的文件名唯一地确定文件的所有相关属性，引入了文件独立于设备的概念，并把设备无关性的思想带到其逻辑极端，导致了UNIX一切皆文件的思想。

18.2 系统调用

为文件操作明确提供以下系统调用：

图(略）

18.3 控制表

数组“file”和“inode”是文件访问机制的基本组件。

18.4 file(5507)

数组“file”被定义为一个数组结构，简称为文件。如果一个”file”中的某个元素“f_count”为0，那么我们认为“file”数组的元素为是未分配的。

每个“open”或“creat”系统调用在文件数组中分配一个元素，这个元素的地址存放在一个调用进程数组“u.u_ofile”的对应元素中，这个元素被它的对应元素进行索引；当索引完成后，这个调用数组被传递回用户态进程；由“newproc”创建的多个后代进程继承父进程“u.u_ofile”数组的内容。

“file”的每个元素包括一个计数器“f_count”，以便确定引用它的当前进程数。“f_count”计数器的递增是通过“newproc”(1878)、“dup”(6079)和“falloc”(6857)实现的；如果文件无法被“openl”(5836)打开，那么其递减是通过“closef”(6657)实现的。

“file”元素的“f_flag”(5509)指出文件是否处于打开状态以供读取和/或写入，或者，它是否是一个管道“pipe”，有关“pipe”的进一步讨论将在第21章进行。“file”结构也包含一个“f_inode”(5511)指针，它指向“inode”表的一个条目和一个32位整数“f_offset”(5512)的一个条目，后者是指向一个文件字符的逻辑指针。

18.5 inode(5659)

“inode”被定义为一个结构数组，如果其引用计数“i_count”为0，那么我们认为“inode”的元素是未分配的。在每个时间点，“inode”数组包含每个文件的一个单一条目(或项)，该条目可被正常的I/O操作引用、或正在被实现、已被实现且设置粘滞位(stiky bit)、或是某个进程的工作目录。多个“file”表的条目可以指向单个“inode”条目，该inode条目描述这个文件的一般特征。

18.6 资源请求

每个文件都需要专有的系统资源。当一个文件存在但未以任何方式引用时，它需要(a)-(c)三个基础项、(d)为某种目的引用项以及(e)-(f)打开要读写的资源：

（a）一个目录条目，它包括在该目录文件中的16个字符;
（b）一个磁盘“inode”条目，它包括存放在磁盘中某个表的32个字符;
（c）0、1或多个磁盘存储块，每块512个字符；
（d）一个核心“inode”条目，即“in-ode”数组中的32个字符;
（e）一个“file”数组条目，含8个字符;
（f）用户态程序中“u.u ofile”数组中的条目，每个文件一个机器字，指向文件数组条目;

我们必须建立一套资源请求的运行机制，有序地分配和解除分配上述资源。为此，我们在下表中给出所包含主要过程的名称：

图：

18.7 打开文件

当一个程序希望引用已存在的文件时，它必须打开该文件以创建通向文件的桥梁。请注意：在UNIX系统中，进程通常会继承其父进程或祖先进程中打开的文件，因此一个进程经常需要的所有文件都已隐式打开。如果该文件尚不存在，那么就必须创建该文件。我们将首先研究第2个案例。

18.8 creat(5781)

5786：“namei”(7518)把路径名转换为一个“inode”指针，“uchar”是一个过程名，它从用户态的程序数据区逐个字符地重新获得路径名;

5787：要么一个空“inode”指针指示一个错误，要么这里不存在该路径名的文件;

5788：有关出错的条件，请参阅UPM手册中的CREAT(II)部分;

5790：“maknode”(7455)通过调用“ialloc”创建一个核心“inode”，接着把它初始化，然后把它输入到相应的目录中；请注意粘滞位的显式重置；

18.9 openl(5804)

该过程被“open”(5774)和“creat”(5793,5795)调用，它传递第3个参数“trf”的值，其参数值分别是0、2和1，其中值2表示不存在所需名称文件。

5812：当“trf”的值为0时，第2个参数“mode”可取值01(“FREAD”)，02(“FWRITE”)或03(“FREAD|FWRITE”)，否则只能取02；如果已存在所需名称的文件，那么就通过调用access(6746)检查所需活动模式的访问权限，这可能为“u.u error”设置为副作用;

5824：如果文件正在被创建，那么就通过调用“itrunc”(7414)删除它先前的内容；当我们把测试条件修改为“(trf == 1)”后，就能改善此处的代码质量，请验证一下这种方法。

5826：“prele”(7882)用于解锁“inode”；你可能会问：“inode”是否被锁定，为什么；

5827：注意“falloc”做的第一件事情是(6847)调用“ufalloc”(6824);

5831：“ufalloc“留下用户文件识别数“u.u_ar0[R0]”；为何该语句放在此处而不是在5834行之后;

5832：假如需要任何特定设备的操作，我们就执行一个调用链，首先调用“openi”(6702)，接着为特定文件调用句柄(handlers)；如果它针对的是磁盘文件，那么就不采取行动；

5839：如果在进行文件数组输入时出错，那么通过“iput”调用释放“inode”条目。

我们认识到文件打开的职责分布在各项任务中。“file”表的条目由“falloc”和“openl”进行初始化；“inode”表的条目由“iget”、“ialloc”和“maknode”进行初始化。

请注意：“ialloc”清除了新分配的“inode”的i_addr”数组，而“itrunc”对预先存在的“inode”执行相同的操作；因此在“creat”系统调用之后，没有与文件相关的磁盘块，它现在被归类为“small”类型。

18.10 open(5763)

我们现在转而考虑一种情况——一个程序希望引用一个已存在文件。我们调用“namei”(5770)，它的第2个参数为0，表示要查找的一个命名文件；“u.u_arg[0]”包含用户态空间定义的文件路径名的字符串的地址；由于用户编程约定与内部数据表示之间是不匹配的，因此我们应将“u.u_arg[1]”的值递增1。

18.11 重新审视openl

由于“trf”的值目前为0，我们检查访问权限(5813)但不处理现有文件(5824)。这有点令人不安，除了对“falloc”(5827)的调用以外，没有直接调用任何其它资源分配的例程。当然，对于一个现有文件，我们不必分配目录条目、磁盘“inode”的条目和磁盘块。如果需要，核心“inode”条目是作为调用“namei”的副作用被分配额的，但是……在哪里初始化?

18.12 close(5846)

“close”系统调用能显式地切断一个用户程序和一个文件之间的连接，因此我们把“close”视为与“open”相反的系统调用。用户程序的文件标识通过r0传递给“close”，它的值首先由“getf”(6619)验证，然后删除“u.u_ofile”的条目，最后在“closef”上进行调用。

18.13 closef(6643)

“closef”被“close”(5854)和“exit”(3230)调用，后者更常见，因为大多数文件不是显式地关闭，而是在用户程序终止时隐式地关闭。

6649：如果一个文件是管道，那么就重置管道模式，并唤醒正在等待管道的任何进程，无论是信息还是空间，这都是适用的;

6655：如果这是引用文件的最后一个进程，那么就调用“closei”(6672)对特定文件做特殊的结束处理，然后调用“iput”;

6657：递减“file”条目的引用计数；如果它当前为0，那么就不再分配该条目。

18.14 input(7344)

“closei”的最后一个操作是调用“iput”；实际上，我们只要切断与核心“inode”的连接并递减引用计数，就能从许多地方调用“iput”例程。

7350：如果此时引用计数的值为1，那么将释放“inode”；当这种情况发生时，我们应锁定“inode”;

7352：如果文件的链接数为0或更少，那么就要取消文件分配，见下文;

7357：“iupdat”(7374)更新记录在磁盘“inode”上的访问和更新的时间;

7358：“prele”解锁“inode”，为什么要在这里以及第7363行调用“prele”?

18.15 删除文件

新文件一旦被打开，就会自动作为永久文件输入到文件目录中；随后，在关闭该文件时，它不会被自动删除。如在第7352行所见，当核心“inode”条目的字段“i_nlink”为0时，将发生删除行为。首次创建文件时，该字段被“maknode”(7464)初始化为1，其递增或递减是由系统调用“link”(5941)或“unlink”(3529)实现的。

一个程序在终止前，我们应启动“unlink”系统调用删除它创建的临时工作文件。请注意：“unlink”调用本身不会删除该文件，它只能在引用计数(“i_count”)即将递减到0—(zero(7350,7362))时发生。为最大限度地减少与程序或系统崩溃时存在的临时文件问题，程序员应遵守以下惯例

（a）在打开临时文件(temporary files)后，应立即对它们执行“unlink”操作;
（b）临时文件应始终放在“tmp”目录中；通过把进程的标识号合并到文件名中，能分别对各个临时文件生成唯一的文件名(unique file names)，请参见“getpid”(3480)。

18.16 读取和写入

在详细检查代码之前，走读该代码调用的缩写摘要是有意义的，它是某个用户进程执行一个读取(read)系统调用时发生的，请参见如下代码。

```
... read (f, b, n); /*user program/
            {trap occurs}
2693 trap
            {system call #3}
5711 read()
5713     rdwr (PREAD);
```

用户进程对系统调用的执行导致以内核态运行“trap”的激活；“trap”识别系统调用#3，并通过“trapl”调用例程“read”，该例程调用“rdwr”。

```
5731 rdwr
5736 fp = getf（u.u_ar0 [R0];
5743 u.u_base = u.u_arg [0];
5744 u.u_count = u.u_arg [1];
5745 u.u_segflg = 0;
5751 u.u_offset [1] = f2-> f offset [1];
5752 u.u_offset [0] = fp-> f offset [0];
5754 readi（fp-> f inode）;
5756 dpadd（fp-> f offset,
                    u.u_arg [1] -u.u_count）;
```

“rdwr“包含的许多代码对“read”和”write“操作都是通用的；它调用“getf”(6619)把用户进程提供的文件标识转换为文件数组中一个条目的地址。请注意系统调用中第1个参数与其余2个参数的传递方式是不相同的。

我们把“u.u_segflg”设置为0，它指出操作目标位于用户态的地址空间中；在使用一个作为“inode”指针的参数调用“readi”之后，通过把请求传送的字符数减去未传送的剩余数量(在“u.u_count”中保留)，添加到文件偏移量，从而执行最终记帐。

```
 6221 readi
 6239 lbn = lshift(u.u_offset，-9);
 6240 on = u.u_offset [1]＆0777;
 6241 n = min(512 - on，u.u_count);
 6248 bn = bmap(ip，lbn);
 6250 dn = ip-> i_dev;
 6258 bp = bread(dn，bn);
 6260 iomove(bp，on，n，B READ);
```

6261 brelse(bp）；

“readi”把文件偏移量转换为两个部分：一个是逻辑块号“lbn”，另一个是块内索引“on”。要传输的字符数是“uu_count”和块中剩余字符数的最小值，在此情况下必须读取附加块(未显示)以及文件中剩余字符数(此示例未显示)。

“dn”是存储在“inode”中的设备号，“bn”是设备磁盘上的实际块号，由“bmap”(6415)使用“lbn”计算。对“bread”的调用找到了需要的块，必要时把它从磁盘复制到核心(core)。“iomove”(6364)传输适当的字符到目标并执行计数操作。

18.17 rdwr(5731)

“read”和“write”执行类似的操作，并共享很多代码。两个系统调用“read”(5711)和“write”(5720)，立即调用“rdwr”：

5736：将用户程序文件标识转换为文件表中的指针;

5739：检查操作read或write是否与打开文件的模式一致;

5743：使用适当的参数在“u”中设置各种标准位置;

5746：“pipe”从一开始就得到特殊处理；

5755：适当地调用“readi”或“writei”;

5756：更新文件偏移量，并把返回给用户态程序的值设置为实际传输的字符数。

18.18 readi（6221）

6230：如果没有要转移的字符，那么就什么都不做;

6232：设置“inode”标志以指示已访问“in-ode”;

6233：如果文件是一个字符特殊文件，那么调用相应设备读取程序，把设备标识作为参数传递;

6238：开始循环传输数据，每次最多为512个字符，直到(6262)遇到不可恢复的错误条件或已转移所请求的字符数;

6239：“lshift”(1410)“u.u_offset”数组中的两个字级联起来，向右移动9位，并截断为16位；这将定义要引用文件的逻辑块号;

6240：“on”是块内的字符偏移量;

6241：“n”最初确定为块中“on”之外的字符数和请求传送的数字最小值；注意“min”(6339)把它的参数视为无符号整数;

6242：如果文件不是特殊块文件，那么…

6243：把文件偏移量与当前文件容量进行比较;

6246：把“n”重置为所请求字符的最小值和文件中的其余字符;

6248：调用“bmap”把文件的逻辑块编号转换为其主机设备的物理块编号；不久之后，“bmap”会有更多内容；当前，请注意“bmap”把“rablock”设置为副作用;

6250：把“dn”设置为“inode”中的设备标识;

6251：如果文件是特殊的块文件，那么…

6252：从“in-ode”条目的“i_addr”字段设置“dn”；据推测，这几乎总是与“i_dev”字段相同，为什么要这里不同;

6253：将预读块设置为下一个物理块;

6255：如果文件的块显然是按顺序读取的话.....

6256：调用“breada”读取所需的块并启动读取预的读块;

6258：否则仅读取所需的块;

6260：调用“iomove”把信息从缓冲区传输到用户区;

6261：把缓冲(buffer)返回到“av”列表。

18.19 Write

6303：如果写入的字符数少于一个块(512的字符数)，那么就必须读取缓冲区的先前内容以便保留适当的部分，否则仅获取任何可用的缓冲区;

6311：没有预先写入(write ahead)功能，但对于最终字符未更改的缓冲区，存在延迟写入(delayed write);

6312：如果文件偏移量当前指向超出已记录文件字符的末端，那么文件显着增长了;

6318：为什么再次设置“IUPD”标志是必要的/需要的，请参见第6285行。

18.20 iomove(6364)

这个过程开头的注释阐述了大部分需说明的内容；“copyin”、“copyout”、“cpass”和“passc”可分别在第1244、1252、6542和6517行找到。

18.21 bmap(6415)

“bmap”函数的一般描述可在UPM手册的File SYSTEM(V)的第2页上找到。

6423：不支持超过2**15个块的文件，字符数为2**24)的;

6427：从“small”小文件算法开始，小文件是指不超过8个块的文件，即最多4096个字符;

6431：若块号为8或更多，则小文件必须转换为大文件。请注意这是“bmap”的副作用，当且仅当“writei”调用“bmap”时才出现；“bmap”永远不会被“readi”调用，请参阅第6245行。因此，全部文件都是以小文件形式开始使用的，并且从未显式地改变为大文件。我们也要注意这种改变是不可逆的!
6435：“alloc”(6956)从设备的空闲列表中分配设备“d”上的块，然后为该块分配一个缓冲区，并返回一个指向缓冲头的指针;

6438：把“inode”的“i_addr”数组中的8个缓冲区地址复制到缓冲区中，然后清除;

6442：“i_addr[0]”被设置为指向为延迟写入而设置的缓冲区;

6448：文件仍然很小，必要时获取下一块;

6456：注意“rablock”的设置;

18.22 剩余部分

请读者自己分析以下程序：
seek（5861）statl（6045）
sslep（5979）dup（6069）
fstat（6014）owner（6791）
stat（6028）suser（6811）

19. 文件目录和目录文件

正如我们此前所见，关于单个文件的许多重要信息都包含在“inode”表中。如果文件当前能访问或正在被访问，那么相关信息就保存在内存(core)的“inode”表中。如果文件位于磁盘上(更常见的是，在某些文件系统卷上)且当前不可访问，则相关的“inode”表是记录在磁盘上的表(文件系统卷)。值得注意的是，“inode”表中没有关于文件名称的任何信息，它存储在目录文件中。

19.1 目录数据结构

每个文件应至少有一个文件名。一个文件可能有不止一个不同的文件名，但是，两个不同的文件无法共享同一个文件名，即每个名称必须定义为唯一的文件。一个文件名可由多个分量或部分(component)组成，在写入时，一个文件名的各分量用反斜杠(“/”)分隔，其分量的顺序很重要，即“a/b/c”与“a/c/b”不同。

如果文件名分为两部分：一个起始部分称为主干(stem)，最后部分称为结尾(ending)，那么两个具有相同主干的文件名通常以某种方式联系起来。例如，两个相关的文件名可驻留在同一磁盘上，可属于同一用户等。用户通过引用文件名对文件进行初始引用，例如，在”open”系统调用。UNIX系统的一个重要功能是把文件名解码为相应的“inode”条目。

为了实现这个功能，UNIX系统创建并维护一个目录数据结构，它等价于一个有命名边的有向图(Directed Graph)。这个有向图在最纯粹的形式中是一棵树，即它有单个根节点，从根节点到任何其它节点之间恰好只有一条路径。该有向图是通过合并一组或多组叶节点而获得的一个网格(lattice)。这种图结构在UNIX是很常见的，但是在其它操作系统中不是这样的。

在此情况下，虽然在根节点和任何内部节点之间仍然只有一条路径，但是在根节点和叶节点之间可能有多条路径。叶节点是没有后继者的节点，并对应于数据文件(data files)。内部节点是有后继者的节点，并对应于目录文件(directory files)。文件名是从根节点与文件对应的节点之间的路径边名称获得的，因此它一般被称为路径名(pathname)；如果一个文件有多个路径，那么它就有多个名称。

19.2 目录文件

目录文件与非目录文件在许多方面没有区别。但是，目录文件包含用于查找其它文件的信息，因此其内容受到周密地保护，并且仅由操作系统操控制。在每个文件中，信息存储在一个或多个容量为512个字符的块中。目录文件的每一块分为32*16的字符结构，每个结构由一个16位“inode”的表指针和一个14个字符的名字组成。指针“inode”指向与目录引用文件相同的磁盘或文件系统卷上的“inode”表。若“inode”值为0时，则表示这是定义目录的一个空闲条目，我们稍后将详细介绍。引用目录的过程概述如下：

namei（7518）搜索目录
link（5909） 创建一个备用名称
wdir（7477） 写目录条目
unlink（3510） 删除一个文件

19.3 namei(7518)

7531：“u.u_cdir”定义进程当前目录的“inode”，一个进程在创建时(“newproc”,1883)继承其父进程的工作目录；我们可用“chdir”(3538)系统调用更改当前的工作目录；

7532：请注意“func”是“namei”的一个参数，且它始终要么是“uchar”(7689)、要么是“schar”(7679);

7534：调用“iget”(7276)，它的主要操作包括：

- 等到“inode”相应的“dp”不再被锁定;

- 检查相关文件系统是否仍然挂载;
- 增加引用计数;
- 锁定“inode”;

7535：可接受多个斜杠，即“////a///b /”与“/a/b”相同;

7537：任何试图替换或删除当前工作目录或根目录的尝试立即被退回!

7542：标签“cloop”标记一次程序循环的开始，它一直延伸到第7667行；每次循环分析该路径名的一个分量，即由空字符、一个或多个斜杠终止的字符串；请注意名称可由许多不同字符构成(7571);

7550：已成功到达该路径名的末尾，返回“dp”的当前值;

7563：目录搜索权限的编码方式与其它文件的执行权限相同;

7570：在尝试把一个名字与一个目录条目匹配之前，把名字复制到更容易访问的位置；请注意长度大于“DIRSIZ”字符的名字将被截去;

7589：“u.u_count”设置为该目录中的条目数;

7592：标签“eloop”标志着程序循环的开始，该循环延伸到第7647行；每次循环处理一个目录条目;

7600：如果目录已被线性搜索而未匹配提供的路径名分量，那么就必定存在一个错误，除非存在以下条件：
（a）这是路径名的最后一个部分，即“c =='\0'”;
（b）要创建文件，即“flag == 1”;
（c）用户程序对目录具有“write”权限;

7606：在“u.u_pdir”中记录新文件目录的“inode”地址;

7607：如果此前已遇到((7642))过某个新目录的一个合适插槽，那么将值存储在“u.u_offset[1]”中；否则，为“dp”指定“inode”设置“IUPD”标旗(或标志)，但这是为什么;

7622：在谨慎地释放任何此前保存的缓冲区后，在合适的时候，从目录文件中读取一个新块；注意使用“bread”，请读者思考为何不是“breada”;

7636：把目录条目的8个机器字复制到数组“u.u_dent”中。在比较之前进行复制的原因是模糊不清的！这实际上能更有效吗；完全复制整个目录的原因对这些评注的作者而言是相当困惑的;

7645：这种比较有效地使用单个字符指针寄存器变量“cp”；若逐字进行比较，则循环将更有效;

7647：“eloop”循环由以下任何一个语句终止：
```
return(NULL);    (7610)
goto out;        (7605, 7613)
```

由于它实现一次成功的匹配，使得程序不必跳转到“eloop”(7647);

7657：如果要删除(“flag==2”)名称、该路径名已完成且用户程序对该目录的“write”有访问权限，那么就返回指向目录“inode”的指针；暂时保存设备标识(为什么不在寄存器“c”?)并调用“iput”(7344)解锁“dp”、递减“dp”上的引用计数并执行任何后续处理;

7664：重新验证“dp”以指向下一级文件的“inode”;

7665：因为目录说该文件存在，“dp==NULL”不应发生；但是，“inode”表溢出且有可能发生I/O错误；有时在一次系统崩溃后，文件系统可能会处于不一致状态。

19.4 一些注释

“namei”是一个关键的过程(procedure)，它在UNIX似乎很早就已实现并经过彻底的调试，然后就基本保持不变。“namei”与系统其它部分之间的接口相当复杂，仅仅因为这一个原因，它就不会赢得年度最佳过程奖。“namei”被12种不同的过程调用了13次：

图(省略)

我们能够看出：
（a）有两个来自“链接”的调用;
（b）调用可分为四类，其中第一类是迄今为止最大的;
（c）在最后两类中，每个类别只有一个代表;
（d）特别是，只有一个包含例程“schar”的调用，该例程序总是针对核心文件(core file)的；核心文件通常在系统收到特定信号时由操作系统生成，信号既可由程序执行过程中的异常触发，又可由外部程序发送，其结果是生成某个进程的内存转储(core dump)文件，后者包含此进程当前的运行栈信息；如果把这种情况作为一种特殊情况处理，例如第2个参数的值为3，那么就能删除“uchar”和“schar”。

“namei”可能以各种方式终止：

（a）如果有错误，那么就返回“NULL”值并设置变量“u.u_error”；大多数错误导致标签“out”(7669)分叉以便正确维护inode的引用计数(7670)；如果在“iget”(7664)中发生错误，那么就不需要这样做;
（b）如果“flag==2”，即其调用来自“unlink”，那么在正常情况下返回的值是命名文件(7660)的父目录的“inode”指针;
（c）如果“flag==1”(即调用来自“creat”、“link”或“mknod”，且若文件尚不存在，则创建一个文件)且如果指定的文件不存在，然后返回“NULL”值(7610)；在此情况下，一个指向新文件目录的“inode”的指针存放在“u.u_pdir”(7606)中；另请注意，在此情况下，一个“u.u_offset”指针指向一个空目录的条目或目录文件的末尾;
（d）如果在其它情况下，文件存在，那么就返回该文件的“in-ode”指针(7551)；“inode”被锁定，引用计数已递增；随后需要调用“iput”消除这些副作用。

19.5 link(5909)

此过程实现系统调用，该调用把现有文件的新名称输入到目录结构中，该过程的参数是该文件已存在的名称和新名称;

5914：查找现已存在的文件名;

5917：如果文件已有127个不同的名称，那么就就不要理会并立即退出;

5921：如果现有文件是一个目录，那么只有超级用户才能重命名它;

5926：解锁现有文件“inode”．当“namei”的第一次调用执行“iget”(7534,7664)时，这是锁定的；在什么条件下解锁该“inode”的失败是灾难性的？尽管有可能，但是现有文件在搜索新名称时遇到目标的可能性是很小的，最有可能的情况是：系统尝试重建已存在的备用文件名或别名;

5927：在目录中搜索第2个名称，目的是创建一个新条目;

5930：存在具有第2名称的现有文件;

5935：“u.u_pdir”被设置为对“namei”(5928)调用的副作用；检查该目录是否与文件在同一设备上;

5940：写一个新的目录条目，参见下文;

5941：增加文件的“link”计数。

19.6 wdir(7477)

此过程在目录中输入新名称，它被i“link”(5940)和“maknode”(7467)调用，调用参数是一个指向内存“inode”的指针。目录条目的16个字符被复制到结构“u.u_dent”中并从那里写入目录文件。请注意“u.u_dent”的先前内容将是目录文件中最后一个条目的名称；该过程假定已搜索目录文件，已分配dlrectory文件的“inode”且已适当地设置“u.u_offset”的值。

19.7 maknode(7455)

从“core”(4105)、“creat”(5790)和“mknod”(5966)调用此过程，在此前调用“namei”且第2个参数为1之后，显示没有指定名称的文件存在。

19.8 unlink(3510)

该过程实现系统调用unlink(3510)，这个调用从目录结构中删除文件名；删除对文件的所有引用后，该文件本身也将被删除。

3515：搜索具有指定名称的文件，如果它存在，那么就返回指向直接父目录的“inode”的指针;

3518：解锁父目录;

3519：获取指向文件本身的“inode”指针;

3522：除超级用户以外，禁止取消链接(unlink)目录;

3528：重写该目录的条目，把“inode”值设置为0;

3529：减少“link”计数；请注意没有尝试减小尺寸目录下面的“high water”标记。

19.9 mknod(5952)

该过程实现同名的系统调用，它只能由超级用户执行，如UPM手册的MKNOD(II)部分所述，该系统调用用于为特殊文件创建“inode”。

“mknod”也解决了目录来自哪里的问题？传递给“mknod”的第2个参数被使用，没有修改或限制设置“i_mode”；请与“creat”(5790)和“chmod”(3569)进行比较。这是“inode”被标记为目录的唯一方式。

在此情况下，传递给“mknod”的第3个参数必须为0。该值被复制到“i_addr[0]”，这适用于特殊文件；如果非0，它将被“bmap”(6447)不加批判地接受。在第5969行之前做插入一面一个语句测试可能是谨慎的，而不是无限期地依赖超级用户不出错。

```
if（ip-> i_mode＆（IFCHR＆IFBLK）！= 0）
```

19.10 access(6746)

“exec”(3041)、“chdir”(3552)、“core”(4109)、“openl”(5815,5817)、“namei”(7563,7664,7658)调用这个过程以便检查文件的访问权限。第2个参数“mode”等价于“IEXEC”、“IWRITE”和“IREAD”中的任何一个，它们的8进制值分别是0100、0200和0400。

6753：如果文件位于已安装为只读的文件系统卷上，或该文件作为一个正在执行程序的文本段，那么就不允许它具有“write”权限;

6763：除非在3个许可组中至少一组是可执行的，否则超级用户可能不执行文件；在任何其它情况下，它总是被允许访问;

6769：如果用户不是文件的所有者，那么把“m”向右移动3个位置，以便按照组的权限操作…如果组不匹配，则再次移动“m”;

6774：比较“m”和访问权限；请注意这里存在异常；如果文件的模式为0077，那么所有者根本无法引用它，但其他人都引用；过插入语句能令人满意地改变这种情况：

```
M = |（m |（m >> 3））>> 3;
```

在第6752行之后，用第6764,6765代替第6764行

```
if（m＆IEXEC &&（m＆ip-> i_mode）== 0）
```

20. 文件系统

大多数计算机系统中不止一个外设用于文件的存储；现在有必要讨论一些与UNIX管理整套文件和文件存储设备有关的问题。首先，我们给出一些定义：

文件系统：一种集成在一个面向块的存储设备上的具有分级目录系统的文件集合;

存储设备：可存储信息的设备，尤其是磁盘包或磁带(本文是DEC磁带)等;

访问设备：用于把信息传送到存储设备或从存储设备传送信息的机制;

只有在访问设备中处于存储状态时，才能够访问存储设备；在此情况下，对存储设备的引用是通过对访问设备引用进行的；

(a) 信息被记录为可寻址的存储块，每块512个字符块，每个字符能独立读取或写入；请注意IBM兼容磁带不满足这个条件；
(b) 设备上记录的信息符合某些一致性标准：

Block#1被格式化为超级块(super block), 请读者参见下文；block＃2到＃(n+1)，其中n记录在超级块中；该快包含一个“inode”表，它引用存储设备记录的所有文件且不引用任何其它文件；记录在存储设备的目录文件仅引用同一存储设备上的文件，即文件系统卷构成一组自包含的文件、目录和“inode”表；如果UNIX系统正式识别出访问设备中已存在存储设备，那么就安装文件系统卷。

20.1 超级块(5561)

超级块(super block)总是在存储设备上记录为block #I，而block #0总是被忽略并可用在与UNIX有关的杂项。超级块包含用于分配资源的信息，即存储块和记录在文件系统的“inode”表中的条目；在安装文件系统卷时，超级块的副本保留在核心中并在此更新；为防止存储设备副本变得过时，其内容会定期写出。

20.2 安装表(0272)

安装表(Mount Table)是指包含每个已安装文件系统卷的条目，每个条目定义安装文件系统卷的设备，指向存储设备中超级块缓冲区的指针以及“inode”指针，该安装表引用如下：

由“main”(1615)调用的iinit(6922)为根设备创建一个条目；
smount(6086)是一个系统调用，它允许其它设备进入；
iget(7276)搜索安装表，如果它遇到一个设置为“IMOUNT”旗标的“inode”；
getfs(7167)搜索“mount”表，查找并返回指向特定设备中超级块的指针；
定期调用update(7201)并搜索安装表，以查找应从核心表写入文件系统卷上维护表中的信息；
sumount(6144)是一个系统调用，它从安装表中删除条目。

20.3 iinit(6922)

该例程由“main”(1615)调用以初始化根设备的安装表的条目。

6926：调用根设备的“open”例程，请注意“conf.c”(4695)定义“rootdev”；

6931：把根设备超级块的内容复制到与任何特定设备无关的缓冲区：

6933：安装表中的首项为0条目分配给根设备；3个元素中仅两个被明确初始化；第2个是“inode”指针，将永远不会被引用;

6936：存储在超级块中的锁被显式重置；当最后一次把超级块写入文件系统卷时，可能已设置这些锁;

6938：根设备安装为“writable”状态;

6939：系统根据超级块的记录时间设置其当前时间和日期的概念；如果系统已经停止相当长的一段时间，那么计算机操作员将需要重置时间；

20.4 安装

从操作角度而言，安装(mounting)一个文件系统卷包括把它放入合适的访问设备、准备设备及输入命令，然后在shell中输入诸如以下参数在内的命令。

''/etc/mount/dev/rk2/rk2''

shell创建(fork)一个进程，执行“mount”系统调用，把两个指针作为参数传递给两个文件名。

20.5 smount(086)

6093：“getmdev”解码第一个参数，以便查找面向块的访问设备;

6096：系统重置“u.u_dirp”，准备调用“namei”解码第2个文件名，请注意“u.u_dirp”由“trap”设置为“u.u _arg [0]”(2770);

6100：检查第2个参数指定的文件是否满足以下条件：当前没有其进程正访问该文件；且该文件不是特殊文件，例如块或字符;

6103：搜索安装表以查找一个空条目(“mp->m bufp==NULL”)或已为该设备创建条目；安装数据结构在第0272行定义;

6111：“smp”应指向安装表中的一个合适条目;

6113：执行适当的“open”例程，其参数为设备名称和read/write旗标；如前所见，关于RK05磁盘的“open”例程是一个“no-op”操作;

6116：从设备读取block #1，这个块是超级块;

6124：把超级块从与“d”相关联的缓冲区复制到与“NODEV”关联的缓冲区；在卸载设备之前，第二个缓冲区不会被再次释放;

6130：“ip”指向第2个命名文件的“inode”，这个“inode”现在被标记为“IMOUNT”；其作用是在强制“iget”(7292)忽略文件的正常内容，而文件系统卷则被安装；在实践中，第2个文件是专门为该目的创建的一个空文件；

20.6 注释

1.一个已安装设备的“read/write”状态仅取决于提供给“smount”的参数；系统没有试图检测硬件的“read/write”状态；若磁盘已被“write protect”且未安装“read only”，则系统就不会出错。

2.安装过程不对已安装文件的系统卷执行任何类型的标签检查，这在文件系统卷很少重新排列的情况下是合理的。然而，在频繁和重新安装卷的情况下，一些验证已安装正确卷的方法似乎是可取的。此外，如果文件系统卷包含敏感信息，那么可能还需包含某种形式的密码保护。超级块(5575)中存在用于存储名称和空间的空间和加密密码。

20.7 iget(7276)

该过程由“main”(1616,1618)、“unlink”(3519)、“ialloc”(7078)和“namei”(7534,7664)调用，其中两个参数—设备和设备上文件的“inode”编号，一起唯一地标识文件。“iget”返回对内存“inode”表的条目的引用。当调用“iget”时，首先搜索核心“inode”表以便查看内存“inode”表的文件是否已存在。如果没有，那么“iget”创建一个。

7285：搜索内存“inode”表…

7286：如果已指定文件的条目，存在……

7287：如果它被锁定，那么就进入休眠状态;

7290：重试一次，注意：因为条目可能已经消失，整个表需从头开始再次搜索；

7292：如果IMOUNT标志开启……这是很有可能的，我们将推迟到后面讨论;

7302：如果未设置“IMOUNT”标志，那么就增加“inode”引用计数、设置“ILOCK”标志及返回指向“inode”的指针;

7306：记住“inode”表中的第一个槽；

7309：如果“inode”表已满，那么就向操作员发送一条消息，并以出错而退出;

7314：此时，即将在“inode”表中创建一个新条目;

7319：读取包含文件系统卷“inode”的块；注意使用“bread”而不是“readi”，假设“inode”信息在block＃2中开始，并且有效“inode”数字从1开始(而不是0)的约定;

7326：此时，如果因读取操作出错，就不向系统的其余部分报告;

7328：复制相关的“inode”信息，此代码隐式使用文件“ino.h”(工作表56)的内容，它在系统的任何地方都没有显式引用；

现在让我们回到未完成的事项：

7292：发现需要要设置“IMOUNT”标志，当安装文件系统卷时，该标志由“smount”设置;

7293：搜索“mount”表以查找指向当前“inode”的条目；虽然该表的搜索开销并不是很大，但是似乎能方便地将后退指针存储在“inode”的“i_astr”字段中，这样能节省时间和代码空间；

7396：把“dev”和“ino”重置为已安装文件系统卷的设备号和根目录的“inode”号，重新开始这个过程；显然，“namei”(7534,7664)调用“iget”，因此该技术允许把安装文件系统卷的整个目录结构集成到预先存在的目录结构中；如果我们暂时忽略目录结构与树结构的可能偏差，那么就有这样一种情况，即现有树的页节点被整个子树结构替换；

20.8 getfs(7167)

除作者的注释以外，关于这个程序没什么需要说的。此过程称为
access（6754）ialloc （7072）
alloc（6961）ifree （7138）
free（7004）iupdat（7383）

请注意“n1”、“n2”的巧妙用法，它们被声明为字符指针—即无符号字符；它允许仅在第7177行对这两个变量进行单方面测试；

20.9 update(7201)

从广义上讲，这个过程确保文件系统卷上的信息保持最新；该过程的注释从第7190行开始，它以逆序描述三个主要子函数。“update”是“sync”系统调用(3486)的全部事务，它通过一个“sync”的shell命令调用。或者，有一个标准的系统程序连续运行，它唯一的函数是每30秒调用一次“sync”,请参见UPM手册的UPDATE(VIII)。在卸载一个文件系统卷之前，“sumount”(6150)调用“update”；除此而外，“panic”(2420)调用“update”是活动停止前系统的最后一个操作。

7207：如果另一个“update”在执行，那么返回即可;

7210：搜索安装表;

7211：对于每个安装的卷……

7213：除非文件系统最近未被修改、超级块被锁定、或卷已安装“read only”...

7217：更新超级块，把它复制到缓冲区，并把缓冲区写入卷;

7223：如果需要，那么就搜索“inode”表，并锁定每个非空(non-null)条目并调用“iupdat”以更新卷的“inode”条目;

7229：允许再次执行“update”开始;

7230：“bflush”(5229)强制执行任何“delayed write”块。

20.10 sumoun(6144)

该系统调用从安装表中删除已安装设备的条目；此调用确保在从物理访问设备上移除存储设备之前，正确终止从设备进出的流量。

6154：在安装表中搜索相应的条目;

6161：在“inode”表中搜索该设备文件的任何未完成条目；如果存在任何此类错误，那么请退出错误，并且不要更改安装表项;

6168：清除“IMOUNT”旗标。

20.11 资源分配

我们现在把注意力转向管理单个FSV(文件系统卷)的资源；在“bmap”请求下，通过“alloc”从空闲列表中分配存储块；该存储块在“itrunc”命令下自由返回到空闲列表，其中，“itrunc”被“core”、“openl”和“openl”调用。

FSV的“inode”表中的条目由“ialloc”、“maknode”和“pipe”调用，该表条目由“ifree”取消，而“ifree”由“iput”调用。FSV的超级块是资源管理程序的核心，“super block”(5561)包含以下信息：

- 容量信息(可用资源总量);
- 最多100个可用存储块的列表;
- 最多100个可用“inode”条目的列表;
- 锁定以控制对上述列表的操纵;
- 标志(flags);
- 上次更新日期。

如果在文件系统卷上可用“inode”条目在内存的列表已耗尽，那么就读取和搜索FSV的整个表并重建该列表；反之，如果可用“inode”表溢出，那么就会立即忘记额外列表，直到以后被再此发现。

一个列表中的可用存储块(数量)使用一个不同的策略，这些块被排列成最多一百个块的组。除第一组外，每组中的第一块用于存储属于上一组块的地址；最后一组是不完整的，其中包含的块地址存放在超级块中。

第一个块编号列表中的第一个条目为0，它充当哨兵(sentinel)的角色；由于整个列表受制于LIFO(后进先出)的规则，因此，如果在列表中发现块号为0，那么就表示它实际为空。

20.12 alloc(6965)

每当需要一个新的存储块保存文件的一部分时，就由“bmap”(6435,6448,6468,6480,6497)调用。

6961：把设备名称转换为指向超级块的指针;

6962：如果设置“s flock”，那么一个可用块的列表目前正由另一个进程更新;

6967：获取下一个可用存储块的块号;

6968：如果该列表中的最后一个块号为0，那么整个列表当前为空;

6970：“badblock”(7040)用于检查从该列表中获得的块号是否合理;

6971：如果超级块中的可用块列表当前为空，那么刚查到的块将包含下一组的地址；

6972：设置“s flock”，在“超级块”中可用块的列表被补充之前，延迟任何其它进程以获得“无空间”指示；

6975：确定要复制的列表的有效条目数;

6978：重置“s flock”，唤醒等待该锁的任何进程;

6982：清除缓冲区，以便在默认情况下记录在文件中的所有信息都为0;

6983：设置“modified”标志以确保通过”update”(7213)写出超级快；

20.13 itrunc(7414)

该过程由“core”(4112)、“openl”(5825)和“iput”(7353)调用；在前2种情况下，文件的内容即将被替换；在第3种情况下，文件将被放弃。

7421：如果文件是字符或块特殊文件，那么就不做任何事;

7423：向后搜索存储在“inode”中的块号列表;

7425：如果文件很大，那么就需要间接提取；对于编号为7或更高的块，需要进行双重间接提取;

7427：以逆序引用缓冲区的所有257个元素；请注意这似乎是引用缓冲区字符#512、#513的唯一地方；由于它们可能包含0，它们对计算并没什么贡献；如果用“510”代替“512”且再次出现在第7432行，那么就会导致全面改善(?);

7438：“free”把单个块返回到该可用列表;

7439：这是在第7427行上的“for”语句的结束；同样，从7432开始到7435结束的语句;

7443：清除“i_addr[]”中的条目;

7445：重置容量信息，并把“inode”标记为“update”。

20.14 free(7000)

该过程被“itrunc”(7435,7438,7442)调用以便把简单存储块重新插入设备的可用列表中。

7005：不清楚为何在此处以及在程序结束时(第7026行)设置“s fmod”旗标，读者有什么建议；

7006：遵守锁定协议;

7010：如果该设备此前没有空闲块，通过设置一个包含block #0条目的元素列表恢复该情况，随后把该值解释为列表末尾的哨兵;

7014：如果超级块中的可用列表已满，那么就把它写入到FSV并设置“s flock”;

7016：获取一个缓冲区，它是与当前空闲列表输入的块相关的;

7019：把超级块列表的内容复制到缓冲区(前面是有效块数的计数)，编写缓冲区，取消锁定及唤醒任何等待的进程；

7025：把返回的块添加到可用列表中。

20.15 iput(7344)

这个程序是UNIX中最受欢迎的程序之一，它从近30个不同的地方调用，其使用被读者频繁地观察到。本质上，它简单地为用于参数传递的“inode”的引用计数进行递减操作，然后调用“prele”(7882)重置“inode”锁，并执行任何必要的唤醒。“iput”有一个重要的副作用，如果引用计数即将减少到0，那么就指示资源释放；如果其链接数也为0，那么这可能只是内存“inode”或两者及文件本身。

20.16 ifree(7134)

这个过程由“iput”(7355)调用，以便将FSV的“inode”返回到超级块中维护的可用列表。如上所述，如果此列表已满或已使用“s ilock”使得列表被锁定，那么就简单地丢弃该信息。

20.17 iupdat(7374)

这个过程由“statl”(6050)、“update”(7226)和“iput”(7357)调用以便修改FSV的特定“inode”条目。如果没有标记相应的内存“inode”(“IUPD”或“IACC”)，它就什么都不做; “IUPD”旗标可由其中一个设置。

图：

“IACC”标志可由以下之一设置
readi（6232）
writei（6285）
maknode（7462）
pipe（7751）

这个旗标由“iput”(7359)进行重置。

7383：如果FSV已安装为“read only”，那么就忘记它；

7386：读取包含FSV的“inode”条目的适当块；正如此前在“iget”中观察到的，请注意使用“bread”而不是“readi”；假设“inode”表从block #2开始，且约定有效的“inode”数字从1开始;

7389：从内存“inode”复制相关信息;

7391：如果合适，更新最后一次访问时间;

7396：如果合适，更新上次修改的时间;

7400：把更新后的块写回到FSV。

21 管道(Pipes)

我们在此简要阐述一下管道的概念，管道“pipe”是一个先进先出(FIFO)的字符列表，它是由UNIX操作系统作为另类文件管理的。一组进程可写入“pipe”，而另一组可从同一“pipe”读取”，因此它主要用于IPC进程间通信。管道利用一种过滤器的概念，过滤器是一个程序，它读取一个输入文件并把它转

换为一个输出文件。UNIX操作系统通过管道把两个或以上程序链接在一起，为其用户提供一套令人惊叹的全面而先进的设备。

21.1 pipe(7723)

一个管道是由一个系统调用管道过程而创建的。

7728：为根设备分配一个“inode”;

7731：分配文件列表条目;

7736：记住文件列表条目为“r”，并分配第2个文件表条目;

7744：在R0和R1中返回用户文件标识;

7746：完成文件数组中的条目

21.2 readp(7758)

在UNIX操作系统中，管道文件与其它文件的不同之处在于：管道保留两个单独的文件偏移量，一个用于“read”操作，另一个用于“write”操作，“write”偏移实际上与文件长度相同。

7763：传递给“readp”的参数是一个指向文件数组条目的指针，从中可提取一个“inode”指针;

7768：“plock”(7862)确保一次仅操作一次“read”或者“write”;

7776：如果管道已满(或文件的有效部分已达到容量限制)，一个希望写入管道的进程被阻塞，那么它将在“ip-> i模式”中设置“IWRITE“标志表示其困境。

7786：休眠前释放锁;

7787：“i_count”是指向“inode”文件表条目的数量；若该值小于2，则“writers”组中必然不存在进程;

7789：等待输入的进程将引发“IREAD”标志；由于管道不能同时充满和空置，因此在任何时候都不应设置一个“IWRITE”或“IREAD”旗标;

7799：“prele”解锁文件并唤醒等待管道的任何进程。

21.3 write(7805)

该过程的结构在许多方面与“readp”结构相呼应。

7828：请注意：当一个“writer”发现没有“reader”时，它就收到一个“signal”，以防它没有监视到“write”操作的结果；与此类似，一个“reader”接收一个0字符计数作为读取结果，这是标准的end-of-file提示。

7835：管道的容量不允许超出“PIPSIZ”规定的字符数。只要“PIPSIZ”(7715)不大于4096，文件就不会转换为一个“large”文件；从访问效率的角度来看，这个规定是非常有必要的；请注意：“PIPSIZ”

限制“writer”偏移指针的值；如果“read”偏移指针距写偏移指针不远，那么该管道的真实容量可能非常小。

21.4 plock(7862)

如果需要，在等待后锁定“inode”；该过程由“readp”(7768)和“writep”(7815)调用。

21.5 prele(7882)

解锁“inode”并唤醒任何等待的进程；除“readp”和“writep”之外，这个过程被其它几个函数(特别是“iput”)调用。

第五部分 字符和交互式终端

第五部分是最后一部分，它是最后一部分但并非最不重要，它关注的是较慢的面向字符外设的I/O；这些设备共享一个公共缓冲池，该缓冲池由一组标准过程操作；面向字符的外设集如下所示：

- KL/DL11交互式终端；
- PC11纸带读取器/打孔器；
- LP11行式打印机

22. 面向字符的特殊文件

面向字符的计算机外设的传输速度相对较慢，一般是每秒<1000个字符，且通常包括可变长度较短记录的字符传输。顾名思义，一个设备句柄(handler)是一个设备和通用系统之间接口的软件部分，它通常是识别一个特定设备特性的软件的唯一部分。

开发者应尽可能合理地为许多相似类型的设备编写驱动程序，并且在适当情况下，同时为多个这样的设备提供服务。交互式终端是指带有键盘输入、串行打印机或可视化输出的设备，在克服了许多困恼后，它们被转换(coerce)到单个设备驱动，因为读者可在详细阅读“tty.c”文件期间进行判断。

在字符设备中应用的标准UNIX处理句柄使用“putc”和“getc”过程，这些过程在标准缓冲池中存储和检索字符，我们将在第23章中对它们进行更详细的描述。有关设备控制器硬件和设备的更完整信息，请参阅“PDP11外设手册”。

22.1 LP11行式打印机驱动程序

这个驱动程序能在文件“lp.c”中找到(Sheets 88,89)，它的大部分复杂性包含在程序“lpcanon”(8879)中。这个程序涉及正确处理特殊字符，这是我们希望首先研究的一个问题。在开始时，读者可忽略“lpcanon”，假设基于它的所有调用(第8859,8865,8875行)都被“lpout-put”(8986)的类似调用所取代。“lpcanon”对行式打印机的字符扮演着最终过滤器的作用，它处理代码转换、特殊格式字符等。

22.2 lpopen(8850)

当打开行式打印机文件时，系统遵循正常的调用顺序：“open”(5774)调用“open1”(5832)，后者调用“openi”(6716)，后者在字符特殊文件情况下调用“cdevsw[..].d_open”；在行式打印机的情况下，后者把(4675)转换为“lpopen”。

8853：如果另一个行式打印机文件已打开或行式打印机未就绪，例如电源关闭、没有纸张、打印机鼓门打开、温度过高或操作员已离线切换打印机，那么就采取错误退出；

8857：设置“lp11.flag”表示文件已打开，打印机具有换页功能，每行缩进8个字符；

22.3 注释

（a).“lp11”是从第8829行开始定义的七机器字结构，该结构的前三个机器字实际上构成“clist”类型结构(7908)；只有第一个元素在“lp.c”中被显式操作，接下来的两个由“putc”和“getc”隐式使用。

（b).“旗标”是该结构的第四个要素，剩下的三个元素是“mcc”、“ccc”和“mlc”，它们分别表示最大字符计数、当前字符数、最大行数。
（c）行式打印机控制器在UNIBUS上有两个寄存器。

行式打印机状态寄存器(“lpsr”)

bit 15: 存在错误条件时设置，参见上文;
bit 7： 当打印机控制器准备好接收下一个字符时，设置“DONE”；
bit 6：设置“IENABLE”，它允许“DONE”或“ERROR”引发一个中断;

行式打印机数据缓冲寄存器(“lp-buf”)

第6位到第7位保存要打印字符的7位ASCII代码，该寄存器是“write only”。

8858：设置行式打印机状态寄存器中的“enable interrupts”位;

8859：向打印机发送换页“form feed”或新页“new page”字符，确保后续字符将在新页面上开始；如上所述，在这个阶段，我们忽略“lpcanon”并假设第8859行是简单的“lpoutput(FORM)”；“lpcanon”的工作是抑制除第一个换页之外的所有换页和换行，以避免浪费纸张;

22.4 lpoutput(8986)

这个打印作为参数；用一个字符调用过程；
8988：“lp11.cc”是一个计数器，它的值表示等待发送到行式打印机的字符数，如果该值足够大(“LPHWAT”,8819)，那么就休眠一段时间以免泛滥字符缓冲池;

8990：调用“putc”(0967)把字符存放在安全的地方；“putc”及其同伴“getc”的功能是第23章要讨论的主题；请注意未检查“putc”是否成功存储字符；字符缓冲区中可能没有空格；在实践中，这似乎不是一个大问题，但读者会感到好奇；

8991：把CPU的优先级提到足够高的水平以便禁止行式打印机的中断，调用“lpstart”，然后降低CPU的优先级。

22.5 lpstart(8967)

当行式打印机准备就绪且在安全位置仍存有字符时，请继续向打印机控制器发送字符。假设控制器正在为一条完整行构建一组字符，“DONE”位的重置速度将比CP向控制器发送字符的速度更快。然而，一旦启动打印周期，“DONE”位将不会再次复位为100毫秒的顺序，这取决于打印机的速度。请注意：在这一系列数据传输期间，中断将被禁止，因此每当“DONE”位置1时，“lpint”就不会进入该行为，除非CPU优先级再次降低时在末尾可能出现一次。

22.6 lpinit(8976)

调用这个过程处理行式打印机的中断；如上所述，CPU忽略大多数可能的中断；CPU接受的中断将与其中任何一个相关联，请参见如下细节。

（a）完成印刷周期；
（b）打印机在一段时间后准备就绪，设置“error”位；
（c）一系列字符翻转中的的最后一次转移。

8980：再次启动，把字符传输到打印机缓冲区;

8981：若等待发送的字符数为0或正好为“LPLWAT”(8818)，则唤醒等待向打印机传输字符的进程。

后一种情况有点令人费解，因为它仅会偶尔得到满足。当然，如果列表中的字符数量变低，那么就开始重新填充。如果“lp-start”执行一系列不间断的传输(至少通过“lpint”)，那么字符的数量可从大于“LPLWAT”的值变为小于它的值，但是在此过程中没有进行这个测试。

相应的，直到列表完全清空之前，一个等待进程程不会被唤醒。其结果是可能经常延迟下一个打印周期的启动，从而允许打印机在低于其额定速度下运行。该问题的一个解决方案是彻底改变行式打印机的缓冲策略；一个不太剧烈的变化会涉及到创建一个新的旗标“lp11.wflag”，用类似的方法替代第8981、8982行。

```
if（lp11.cc <= LPLWAT && lp11.wflag）
  {wakeup（&lp11）;
        lp11.wflag = 0
}

{    lp11.wflag ++;
     sleep(&lp11, LPRI);

}
```

22.7 lpwrite(8870)

这是一个作为写入系统调用而调用的过程，这意味着在这个过程中执行一个调用链，最终实现这个过程。在这个调用链中，“write”(5722)调用“rdwr”(5755)，后者调用“writei”(6287)，后者调用“cdevsw[...].d_write”（4657），最后，它被转换为“lpwrite”。

“lpwrite”获取用户区中记录的空终止字符串的非空字符，并通过“lpcanon”一次一个地传递给“lpoutput”。导致调用此过程的过程调用列表类似于“lpopen”。输出换页字符以清除当前页面，并重置打开标志。

22.8 讨论

为了把一个字符串发送到打印机，程序执行一个调用链，先调用“lpwrite”一次或多次，接着调用“lp-canon”，再接着调用“lpoutput”。若在任何时候存储太多字符，则进程在“lpoutput”中休眠。“lpoutput”迟早将继续，把字符存储在缓冲区中；若有可能，它调用“lpstart”把一个字符串发送到打印机控制器。

当有更多字符可供发送时以及来自打印机的中断时，都会调用“lpstart”。对“lpstart”的大多数调用实际上都没有实现。在打印机刚完成一个打印周期时，“lpstart”偶尔能把整串字符发送到打印机控制器。

22.9 lpcanon(8879)

这个过程对发送到行式打印机的字符进行解释，并做各种修改、插入和删除的操作，因此它实际上扮演一个过滤器的作用。

8884：从此行到第8913行的代码段与字符转换有关，如果一个完整的96个字符集不可用时，我们就使用64个字符集进行字符转换。

由于打印机的能力一般不随时间而变化，因此定义的变量“CAP”(8840)必须一次性设置在一个特定的安装中。如果编译器有“druthers”，那么基于(lp11.flag和CAP)的运行时测试可被基于CAP的编译时测试代替；如果CAP变为0，那么第8913行的整段代码可编译为空。

本代码预先考虑可能安装两个或更多不同类型的打印机的情况，即便如此，这里也存在不一致的问题。一方面，使用“CAP”、“IND”和“EJECT”；另一方面，使用“EJLINE”和“MAXCOL”。实际上，不同尺寸的表单在单个打印机上是很常见的，因此最后2个参数不应是常量，而应是动态可设置的。

8885：小写字母表是通过增加常数来转换的的，它被方便地定义为“A’-’a”；

8887：某些剩余字符是特殊字符，其打印方式是在为字符基础上增加一个减号，例如“{”(8889)印为“{ - ”;

8909：“similiar”字符通过递归调用“lpcanon”输出的，它使“lp11.ccc”按一递增而产生副作用;

8910：递减当前字符数计数(与“后退空格”字符的效果相同和…

8911：准备输出减号;

8915：从这里开始的“switch”语句延伸到行，在垂直和水平间距中的某些字符被赋予特殊的解释，并具有延迟的通信;

8917：对于水平制表符，当前字符数向上舍入到8的下一个倍数，不要立即输出任何空白字符;

8921：对于换页或换行字符，如果：
（a）打印机没有“页面恢复”功能;
（b）现行线不为空;
（c）自上次换页字符以来已经完成一些行，然后.....

8926：增加完成的行数;

8927：如果当前页面上已完成足够的行且打印机具有换页功能，那么就把新行字符转换为换页。

8929：输出字符，如果是换页，那么重置已完成行数为0;

检查此代码将显示：

（a）任何以换页开始的换页或换行串，若发送给具有换页功能的打印机，则减化为单一换页;
（b）发送到没有换页功能的打印机的换页字符导致新行开始，但在没有注释的情况下传递。

8934：对于回车，并注意换页和换行，把当前字符数重置为0或八，取决于“IND”，并返回;

8949：对于所有其它字符……

8950：若收到一串退格(真实或人为)和/或回车，则输出一个回车并把最大字符数重置为0;

8954：若计数不超过最大行长度，则输出空白字符以把最大字符数计入当前字符数；或许这两个变量更准确地称为实际字符数和逻辑字符数;

8959：输出实际字符。

22.10 对于有空的读者：一个建议

读者可观察到：用于复打或下划线字符的退格引入一个单独的打印周期，并且在这些特征在频繁使用的条件下，打印机的有效输出率可能会大为降低。如果这被认为是一个严重的问题，就要重写“lpcanon”以确保在此情况下每一行不超过2个打印周期。

22.11 PC-11纸带读取器/打孔驱动器

该驱动程序可在Sheets 86和87的文件“pc.c”中找到，一方面，因为有类似于“lpcanon”的例程它比行式打印机驱动程序更简单；另一方面看，它从I/O的角度来看比较复杂，因为其输入和输出设备能同时独立地激活。这个设备的操作描述包含在丹尼斯·里奇的The UNIX I/O System文档中。读者或许会注意到某些特殊功能：

（1）一个进程仅能为读取打开文件一次，但没有限制写进程(writer)的数量;
（2）与行式打印机驱动程序相比，该例程更多地关注错误条件，然而，其处理不详尽;
（3）“passc”(8695)知道需要多少个字符，并且当达到“足够”时就返回一个负值;
（4）当且仅当“pcclose”确认设备已打开输入时，它谨慎地清除输入队列中的任何剩余字符。

23. 字符处理

字符特殊设备的缓冲通过一组4个字块提供，每个字块提供六个字符的存储。原型存储块是“cblock”(8140)，它包含一个字指针(类似的结构)以及六个字符。包含字符计数器加上头和尾指针的类型“clist”(7908)的结构用于“cblock”类型块列表的标头(headers)。

当前不使用的“cblock”通过它们的头指针链接到一个列表，其头部是指针“cfreelist”(3149)；列表最后一个元素的头指针具有“NULL”值。“cblock”列表为字符列表提供存储。过程“putc”可用于添加字符到该列表尾部，“getc”用于从该列表头部移除字符。图23.1到23.4描述当字符被删除和添加时列表的开发。

图23-1(省略)， 图23-2(省略)

在开始时，该列表假定包含14个字符“efghijklmnopqr”，请注意头指针和尾指针指向字符。如果第一个字符“e”被“getc”删除，那么图23.1中描绘的情况就变为图23.2。字符计数已经递减，头指针已经前进一个字符位置。如果从列表的头部删除另一个字符“f”，那么情况就如图23.3所示。字符数已减少；第一个“cblock”不再包含任何有用的信息并已返回“cfreelist”；头指针现在指向第二个“cblock”中的第一个字符。

当前的问题是：““第1种和第2种情况之间的差异是如何产生的，以便所采取的行动总是一致的?”如果你还未猜到答案，那么它其实涉及到指针地址模块8的值。由于在二进制计算机很容易除以8，因此每个“cblock”选择六个字符的原因应该是也很明显。在图23.3和图23.4之间的变化显示一个字符添加到列表。由于图23.3中的最后一个“cblock”已满，因此从“cfreelist”获得一个新“cblock”并链接到“cblock”列表中，字符数和尾指针已适当调整。

23.1 cinit(8234)

该过程由“main”(1613)调用一次，它把一组字符缓冲区链接到空闲列表“cfreelist”，并计算字符设备类型的数量。

8239：“ccp”是数组“cfree”中第一个字的地址(8146);

8240：将“ccp”舍入到下一个最高的八个，并标出“cblock”大小的部分，注意不要超过“cfree”的边界；请注意一般会定义如“NCLIST-1”这样的块，而不是“NCLIST”;

8241：将“cblock”的第一个字设置为指向空闲列表的当前头部；请注意“c_next”在第8141行定义且“cfreelist”的初始值为“NULL”;

8242：更新“cfreelist”以指向列表的新头部;

8244：计算字符设备类型的数量；当引用表46的“cdevsw”，就会看到“nchrdev”被设置为16，而一个更合适的值是10。

23.2 getc(0930)

此过程被以下函数调用

flushtty（8258,8259,8264）
canon（8292）
ttstart（8520）
ttread（8544）
pcclose（8673）
pcread（8688）
pcstart（8714）
lpstart（8971）

使用一个单一参数，这是“clist”结构的地址。

0931：把参数复制到r1，并在栈中保存把初始处理器状态字和r2的值;

0934：把处理器的优先级设置为5，高于字符设备的中断优先级; ture(即字符数); 此结构的第2个单词(即指向头部字符的指针)移动到r2;

0936: r1指向一个“clist”结构的第一个机器字，即字符计数；把该结构的第2个机器字移动到r2，即指向头字符的指针；

0937：如果列表为空(头指针为“NULL”)，则转到行第0961行;

0938：把头部字符移动到r0，并递增r2作为副作用;

0939：屏蔽r0以消除任何扩展的负号;

0940：将更新的头指针存回“clist”结构，这可能需要稍后改动;

0941：减少字符数，如果它仍然是正数，请转到第0947行;

0942：列表现在为空，因此把头尾符号指针重置为“NULL”，并转到第0952行;

0947：看看r2的3个最低有效位，若它们的值为非0，则跳转到第0957行并立即返回调用例程;

0949：这时r2指向“cblock”以外的下一个字符位置；把存储在“cblock”第一个字的值(在r2-8处)移动到“clist”的头指针处，且下一个“cblock”的地址也存储在以上第1个字中；请注意r1在第0941行有副作用地递增；

0950：存储的最后一个值需要按2递增加，参见图23.2和23.3;

0952：这时由r2确定的“cblock”将返回“cfreelist”；要么r2指向“cblock”，要么超出它；在此递减r2以便r2指向“cblock”;

0953：重置r2的3个最低有效位，留下一个指向“cblock”的指针;

0954：把“cblock”链接到“cfreelist”;

0957：从栈中恢复r2和PS的值并返回;

0961：此时已知列表为空，因为遇到“NULL”的头指针，确保尾指针也是“NULL”；

0962：把-1移至r0，当列表为空时，返回结果。

23.3 putc(0967)

此过程被以下函数调用

canon（8323）
ttyinput（8355,8358）
ttyoutput（8414,8478）
pcrint（8730）
pcoutput（8756）
lpoutput（8990）

这里有两个参数：一个字符和一个“clist”结构的地址。

由于“getc”和“putc”具有相关的功能，这两个过程的代码在许多方面是相似的。因此，“putc”的代码不被详细检查，而是留给读者思考。值得注意的是：如果需要新“cblock”且“cfreelist”为空，那么“putc”可能会失败，在此情况下，返回一个非0值(第1002行)而不是0值(第0996行)。请注意这里讨论的“getc”和“putc”程序与UPM手册的“GETC(III)”和“PUTC(III)”中的程序没有直接关系。

23.4 字符集

UNIX使用一套完整的ASCII字符集，该字符集显示在UPM手册ASCII(V)中。由于该字符集一般被认为是没有问题的，但是它并不总是合理的，这里的一些评论似乎是有序的。ASCII是美国信息交换标准代码英文单词的首字母缩写。

23.5 控制字符

128个ASCII字符中的前32个是非图形的，用于控制传输或显示的某些方面，UNIX显式使用或识别的控件字符参见下图。请注意其中最后两个属于代码的最后96个字符或图形部分。

图：控制字符集合

23.6 图形字符

有96个图形字符。其中2个是空格和删除，它们在屏幕上都是不可见的，并且可用控制字符进行分类；图形字符可以分为3组，每组32个字符，可粗略地表征为

- 数字和特殊字符
- 大写字母字符
- 小写字母字符。

当然，由于只有26个字母字符，后两组也包括一些特殊字符；特别是，最后一组包括以下六个非字母字符：

140'
173 {
174 |
175}
176~
177

反撇号左支撑
竖条
右支撑
代字号删除

23.7 图形字符集

支持所有ASCII图形符号设备如行式打印机或终端通常被称为支持96ASCII字符集，尽管实际上只涉及94个图形。支持所有ASCII图形符号的设备，除了在最后一组32字符以外，据说支持64ASCII字符集。这种设备缺少小写字母和以上列出的符号，即“～”，“{”，“|”和“\}”。请注意“delete”字符，因为它不是在屏幕上不可见的字符，因此它仍然得到支持。后一组中的设备可被称为“仅大写”。有时，一些图形符号可能是非标准的，例如， “←”字符取代“_”字符，虽然通常不是致命的，但是它们有可能造成方便。

正如读者所知，UNIX更倾向于通过“小写”字母的角度观察世界。从“仅大写”的终端接收的字母字符在从大写到小写的接收(指令)时立即被转换。如果小写字母前面有一个反斜杠，则随后把小写字母转换回大写字母。对于这种终端的输出，大写和小写字母字符都映射为大写字母。而行式打印机和终端的约定是不同的，因为：

(a)行式打印机的水平对齐通常很重要，它能在没有很大困难的情况下打印复合和重打字符，在此情况下使用减号;

(b)对于计算机终端而言，水平对齐并未被认为如此重要；提供超载字符的退格在大多数VDU视频显示器上不起作用; 因为相同的图形约定用于I/O时，符号应尽可能方便地输入。

23.8 maptab(8117)

该数组用于在一个终端之前由一个单反斜杠“\”进行字符转换。004(eot)、’ # ’和’@‘这三个字符总是有特殊含义，因此需要用(单)反斜杠断言，无论何时按字面解释它们。这三个字符位于“maptab”的自然位置，即它们在ASCII表中的位置。因此，例如’ # ’的代码为043和

maptab [043] == 043。

“maptab”中的其它非空字符涉及来自“仅大写”设备的输入字符的转换，并且不出现在它们的自然位置中，而是出现在与它们等价字符的位置，例如， “\{”出现在“{”的自然位置，因为“\{”将被解释为“{”等。请注意有关字母字符的情况，只有在记住任何反斜杠被识别之前，字母字符都被转换成小写字母时，这才是可解释的。

23.9 partab(7947)

该数组由256个字符组成如“maptab”；令人遗憾的是UNIX操作系统源代码手册中省略“partab”的初始化；这当然是必要的，一直到现在都是如此：

```
    char partab [] {

0001,0201,0201,0001,0201,0001,0001,0201,
0202,0004,0003,0205,0005,0206,0201,0001,
0201,0001,0001,0201,0001,0201,0201,0001,
0001,0201,0201,0001,0201,0001,0001,0201,
0200 0000,0000,0200,0000,0200,0200,0000,
0000 0200,0200 0000,0200,0000,0000,0200,
0000,0200,0200 0000,0200,0000,0000,0200,
0200,0000,0000,0200,0000,0200,0200,0000,
0200,0000,0000,0200,0000,0200,0200,0000,
0000,0200,0200,0000,0200,0000,0000,0200,
0000,0200,0200,0000,0200,0000,0000,0200,
0200 0000,0000 0200,0000,0200,0200,0000,
0000 0200,0200 0000,0200,0000,0000,0200,
0200,0000,0000,0200,0000,0200,0200,0000,
0200,0000,0000,0200,0000,0200,0200,0000,
0000,0200,0200,0000,0200,0000,0000,0201
    };
```

“partab”的每个元素都是一个8位字符，使用适当的位掩码(0200和0177)，可解释为两部分结构：

比特 7 比特 3-5 比特 0-2
奇偶校验位;
不使用。始终为零; 代码号。

当计算机传输字符时，奇偶校验位将附加到7位ASCII码，从而形成具有偶校验的8位代码。代码号由“ttyoutput”(8426)使用，以便把字符分类为7个类别之一，用于确定在传输下一个字符前应发生的延迟。这对于机械打印机尤其重要，因为它需要时间从线路的末端返回等。

24. 交互式终端

本书剩余的任务在本章和下一章完成，这些任务主要是考虑控制交互式终端(或终端)的代码。用户可使用各种各样的终端，并且能同时把多种不同类型的终端连接到一台计算机上。除形状、大小和颜色的非本质特征以外，不同类型终端的特征主要包括：

(a) 传输速度：例如ASR33电传打字机为110波特，DECwriter为300波特，可视显示器(VD)为2400波特或9600波特;

(b) 图形字符集: 特别是完整的ASCII图形集和64图形子集;

(c) 传输奇偶校验：奇数，偶数，无或无操作;

(d) 输出技术：串行打印机或可视显示器;

(e) 杂项：组合回车/换行，字符，半双工终端(输入字符不需回显)，识别标签字符;

(f) 某些控制功能的特征延迟，例如，在单个字符传输时间内，可能无法完成回车等。

除可用和使用的各种终端之外，还有各种硬件设备可用于把终端连接到PDP 11计算机。例如：

DL11/KL11
单线，异步接口; 13标准，传输速率在40到9600波特之间;

DJ11
16线，异步，缓冲串行线路复用器; 11，速度在75到9600波特之间，可在四个线路组中选择;

DH11
16线，异步，缓冲，串行线路复用器; 14速，可单独选择; DMA传输

上述每个接口都有完双工或半双工模式；处理5、6、7或8级代码；生成奇数、偶数或无奇偶校验；并生成1、1.5或2位的停止代码；除上述异步接口之外，还有许多同步接口，例如DQ11，每个接口都有自己的控制特性并需要一个单独的操作系统设备驱动程序。

在这些文件之间可共享的共同代码被收集到单个文件“tty.c”中，用户能在表81到表85中找到这个文件。表79的文件“tty.h”中收集了一组通用定义。例如，表80包含文件“kl.c”，它构成一组DL11/KL11接口的设备驱动程序；该设备驱动程序总是需要存在的，因为一个KL11接口总是包含在操作员控制台终端的系统中。

24.1 ‘tty’结构(7926)

无论使用什么类型的硬件接口，“tty”的一个实例与系统每一个终端的端口相关联；在本章上下文中的端口是一个附接终端线的地方。因此，DL11只提供一个端口，而DJ11提供多达16个端口；“tty”结构由16个机器字组成，包括：

表24.1交互式终端

读者应仔细研究表79中的信息，在选择此处检查的代码时，下面列出的某些项目不会以任何必要的方式被引用。

t_char
t_speeds
（7940）NLDELAY
（7941）TBDELAY
（7974）
（7975）

24.3 初始化

“tty”结构的初始化是设备驱动程序中各种“open”例程负责的，例如“klopen”(8023)。表24.1的B组中的项目可通过“stty”系统调用改变，并可通过“gtty”系统调用询问当前值。

这些描述包含在UPM手册的“STTY(II)”和GTTY(II)中。这些调用由“stty”shell命令调用并在STTY(I)描述。因为“stty”和“gtty”系统调用需要文件描述符作为参数，所以它们只能应用于一个“open”字符特殊文件。

这两个系统调用共享大量的通用代码,我们将跟踪执行“stty”以下的进度，并向读者留下类似的“gtty”执行跟踪。

24.4 stty(8183)

这个过程实现“stty”系统调用，它使用作为指针提供的参数把3个用户参数信息复制到“u.u_arg [..]”，然后调用“sgtty”。

24.5 sgtty(8201)

8206：获取一个指向文件数组条目的有效指针;

8209：检查文件是否为特殊字符;

8213：为设备类型调用一个合适的“d_sgtty”例程，参见表46；请注意“d_sgtty”例程是行式打印机和纸带读取器/打孔器的“nodev”。

24.6 klsgtty(8090)

这是一个“d_sgtty”例程的例子，它调用“ttystty”传递一个指针给适当的“tty”结构作为参数。

24.7 tysty (8577)

一个源自“STTY”的调用的第2个参数为0；

8589：立即清空与终端关联的所有队列，它们很可能包括一些无意义的信息;

8591：重置速度信息，这在有DH11接口的情况下是有用的，但对当前的代码选择关系不大；重置“erase”字符和“kill”字符；“kill”在这里表示扔掉当前输入行；请注意：如果这些字符分别从“#”和“@”的值中改变，那么就不会对“maptab”进行相应的更改，它们也不应这样做；

8593：重置定义某些相关终端特征的标志，请参见表79：

标志位
XTABS 1
LCASE 2
回声3
CRMOD 4
RAW 5

如果设置...
终端不能正确解释水平制表符;
终端只支持64个字符的ASCII子集;
终端以全双工模式运行，输入字符必须回显;

输入后，回车由换行代替;输出时，换行由回车和换行代替;
输入字符将完全按照接收的方式发送到程序，无须“erase”或“kill”处理或调整反斜杠字符。

此外，在发送下一个字符之前和在选择发送的字符之后，应选择由“ttyoutput”(8373)的延迟位。

8,9换行;
10, 11个水平标签;
12, 13回车;
14垂直标签或换页(feed form)。

24.8 DL11/KL11终端设备句柄

文件“kl.c”构成通过DL11/KL11接口连接到系统终端的设备句柄；这个组至少有一个成员——操作员的控制台终端；因此这个句柄将始终存在。每个DL11/KL11硬件控制器提供一个异步串行接口，把单个终端连接到PDP 11系统。有关此接口的更详细信息，请参阅“PDP11外设手册”。

24.9 设备寄存器

每个DL11/RL11单元都有一组4个寄存器，在UNIBS上占用4个连续的机器字；UNIX将类型为“klregs”(8016)的结构映射到每个寄存器组。

接收器 状态寄存器(klrcsr)

Bit 7: 接收者完成；一个角色已被转移到接收器数据缓冲寄存器;
bit 6: 接收器中断“enable”；在设置时，每次设置第7位都会产生中断;
bit 1: 数据终端就绪；

bit0: 读取器启用，只写；当设置时，bit 7清0；

接收器数据缓冲寄存器(klrbuf)
bit 15： 当设置时，错误指示；
Bit 7-0: 收到的字符，只读

发送器状态寄存器(kltcsr)
bit 7: 传输器就绪
当数据加载到发送器数据缓冲器时清除，并在后者准备好接收另一个聊天器时设置;

bit 6: 发送器中断enable
当设置时，只要设置bit 7，就会产生中断

发送器数据缓冲寄存器(kltbuf)
bit 7-0: 传输的数据，只写

24.10 UNIBUS地址

接收器状态寄存器始终是从一个4字边界开始其最低地址，以下地址均为18位的八进制地址。

Diagram(省略)

除操作员的控制台界面有自己的标准UNIBUS位置，接口分为两组(由于这里不相关的原因)。按照惯例，在每个组内，寄存器从最低地址开始的连续位置中分配。

24.11 软件考虑

我们为一个特定的安装设置“NKL11”(8011)去定义前两组中的接口数量；相应地，设置“NDL11”(8012)去定义第三组的接口数量。如果一些硬件变更改变了接口的实际数量，那么这些变更就必须通过修改和重新编译“kl.c”反映在软件中，并重新链接到UNIX操作系统。我们看到“klopen”为每个接口计算接收器状态寄存器的正确内核态的地址(16位)，并把它存储(8044)到适当的“tty”结构的“t_addr”元素中。

24.12 中断向量地址

第一个接口的向量地址是060和064，它们分别用于接收器和发送器中断。附加的DL11/KL11接口具有始终至少为0300的向量地址，并根据考虑可能存在的其它接口的规则分配这些地址。

中断双字节的第2个字是“新处理器状态”字，该字的五个低位可任意选择且实际上用于定义次要设备编号，参见类似用途区分各种陷阱—表05。系统把新的处理器状态字的掩码版本作为参数“dev”提供给中断处理例程，参见第8070行。

24.13 源代码

我们现在对文件“kl.c”(表80)和“tty.c”中的代码进行详细研究(表81–表85). 我们首先把“open”和“close”终端视为字符特殊文件和中断处理；然后，我们在下一章将查看从终端接收的数据；最后，我们把数据传输到终端。我们已经讨论过“klread”(8062)、“klwrite”(8066)和“klsgtty”(8090)，这里就不再重复。

24.14 klopen(8023)

调用此过程以将“open”终端作为字符特殊文件。对于在系统中处于活动状态的每个终端，通常由程序“/etc/init”进行此调用。由于子进程继承父进程的打开文件，因此通常不需要其他进程再次“open”该设备。请注意没有尝试同时停止将终端作为打开文件的两个不相关的进程。

8026：检查次要设备号;

8030：找到一个合适的“tty”结构;

8031：如果打开文件的进程没有关联的控制终端，那么为该角色指定当前终端；请注意存储引用是“tty”结构的地址；

8033：把终端设备号存储在“tty”结构中;

8039：计算终端和存储的适当设备寄存器集的地址;

8045：如果终端尚未“open”，那么对“tty”结构进行一些初始化。

8046：“t_state”设置显示文件为“open”，这样，如果文件被第二次打开，那么接下来的三行将不被执行，可能撤消“stty”系统调用的效果: “t_state”也设置为显示“CARR ON”(执行)。这是一个软件标志，它显示终端在逻辑上被启用，而不管终端的真实硬件状态。如果为终端重置“CARR ON”，系统应忽略来自终端的所有输入；这似乎并不完全正确，这一点将在以后再次讨论;

8047：标准终端被假设为无法解释水平制表符，仅支持64字符ASCII子集，以全双工模式运行，并要求回车和换行字符以提供正常的新行处理；这可能是一个33型电传打字机;

8048：根据UNIX约定设置“erase”和“kill”字符;

8051：接收器控制状态寄存器初始化为模式“0103”，使终端准备就绪，启用读取和接收器中断;

8052：初始化发送器的控制状态寄存器，以便在接口准备好接收另一个字符时产生中断；请注意：“open”例程不区分文件打开中的只读、只写或读写三种情况。

24.15 klclose(8055)

8057：在这种结构的数组中找到适当“tty”结构的地址，“kl11”(8015)；在表80第2列的所有程序中都能观察到这种操作，并且应注意其相关性；

8058：“wflushtty”(8217)允许终端的输出队列“排空”，然后刷新输入队列;

8059：“t_state”被重置，因此“ISOPEN”和“CARR ON”不再成立。

24.16 klxint(8070)

响应发送器中断执行该过程，它应与“pcpint”(8739)和“lpint”(8976)进行比较。请注意，参数“dev”是中断向量中的“新处理器状态”字的掩码版本，保留低位5位；如果向量已正确初始化，那么将正确识别次要设备编号；第8074行测试的第2部分将在下一章最后讨论。

24.17 klrint(8078)

响应接收器中断执行该过程，它与“pcrint”(8719)相比并不容易，尽管相似之处肯定存在。

8083：从接收器数据缓冲寄存器读取输入字符;

8084：为下一个字符启用接收器;

8085：注释阐述了“硬件故障”，最好地相信它;

8086：把字符传递给“ttyinput”以便把它插入适当的原始输入队列。

25 文件“tty.c”

这最后的一章终于揭开交互式终端处理程序的复杂性，它主要包括：

（a）处理“erase”和“kill”字符;
（b）在输入和输入期间转换字符，输出仅用于大写的终端;
（c）在“回车”等各种特殊字符后插入延误。

我们在前一章已阐述例程“gtty”(8165)，“stty”(8183)，“sgtty”(82a1)和“ttystty”(8577)。

25.1 flushtty(8252)

这个过程的目的是规范化与特定终端关联的查询，其效果是立即终止向终端的传输，并丢弃任何累积的输入字符。

8258：扔掉“熟”输入队列中的所有内容;

8259：输出队列同上;

8260：唤醒等待提取的任何进程，来自原始输入队列的字符;

8261：输出队列同上;

8263：提高处理器优先级以防止来自终端的中断…

8264：刷新原始输入队列；

8265：“分隔符计数”正确设置为0；“flushtty”被“wflushtty和“ttyinput”(8346,8350)调用：

（a）终端未以原始模式运行，并且从终端接收“exit”或“delete”字符；
（b）原始输入队列增长得不合理，大概是因为没有进程从终端读取输入;

它们都已被发送，因此这个过程一直等到到终端字符队列为空，然后，调用“flushtty”清理输入队列。“wflushtty”被“klclose”调用(3053)，这种情况不经常发生，实际上仅当所有引用终端的文件都关闭时，即仅在用户注销时才会发生；在调整终端环境参数之前，它也被“ttystty”(8589)调用。

25.3 字符输入

对于一个从终端输入程序的请求，有一系列过程调用延伸到“ttread”.....

25.4 ttread(8535)

8541：检查终端是否处于逻辑活动状态;

8543：如果“熟”输入队列中有字符或“canon”(8274)的调用是成功的…

8544：从“熟”输入队列传输字符，直到它为空或已转移足够的字符，以满足用户的要求。

25.5 canon(8274)

该过程由“ttread”(8543)调用，以便将字符从原始输入队列传送到“cooked”输入队列；在处理“erase”和“kill”字符之后且仅在大写终端的情况下，处理转义字符，即前面带有字符“\”的字符。如果“cooked”输入队列不再为空，则“canon”返回非0值。

8284：如果原始输入队列中的分隔符数为0，那么…

8285：如果终端在逻辑上不活动，那么就返回;

8286：否则就去休眠；请注意，此上下文中的分隔符是所有分隔符的字符，八进制值为377，并由“ttyinput”(8358)插入;

8291：设置“bp”指向工作数组的第3个字符“canonb”;

8292：开始一个循环，延伸到第8318行，每个循环从原始队列中删除一个字符;

8293：如果字符是分隔符，那么将分隔符计数减1并退出循环，即转到第8319行;

8297：如果终端未以原始模式运行...

8298：如果前一个字符，注意它是“bp[-1]”符号，而不是反斜杠'\'，那么就执行第8299行至8307的代码，否则执行从第8309行开始的代码；前一个字符不是反斜杠；

8299：如果字符是“erase”并且.....

8300：如果要擦除至少一个字符，那么就备份指针“bp”;

8302：从第8292行开始的循环的下一个循环开始;

8304：如果字符是“kill”，那么扔掉当前行累积的字符，并返回到第8290行;

8306：如果字符是“eot”(004)，它通常在终端处生成为“control-D”，那么就忽略它(且不要将其置于“canonb”中)，并开始下一个循环；若此字符出现在一行的开头，则随后“ttread”(8544)将在“cooked”输入队列中找不到任何字符，即它将读取0长度记录，然后导致程序接收正常的“end-of-file“提示；

前一个字符串是反斜杠

8309：如果“maptab [c]”位非0且“maptab[c]==c”或终端仅为大写，那么...

8310：如果最后一个字符不是反斜杠('\')，那么就用“maptab[c]”替换“c”并备份“bp”，这样反斜杠就会被删除。

准备好的字符

8315：将“c”移到“canonb”的下一个字符中，如果此数组现已满，那么就退出循环；

行已完成

8319：此时，输入行已组装在数组“canonb”中;

8322：把“canonb”的内容移到“cooked”输入队列，并返回“successful”的结果。

25.6 注释

(A) “bp”在“canonb”的第3个字符处开始(8291)的原因可在第8310行找到。

(B)处理反斜杠的一些微妙之处仍没有立即显现出来，读者无疑会在实际使用UNIX时会遇到它。由于“maptab[c]”对于“c ==‘\’”为0，而8进制值为134，所有反斜杠都会复制到“canonb”中；如果要声明它后面的字符，那么要随后覆盖单个反斜杠(如’＃’或’@‘或eot(004)的情况；或者，如果字母字符的情况只为大写的终端改变，那么一个反斜杠将被完全覆盖。

25.7 ttyinput(8333)

“canon”从原始输入队列中删除字符；首先，它们被“tty-input”放在那里，当从硬件控制器接收到输入字符时，就用“klrint”(8087)调用它们；传递给“ttyinput”的参数是一个字符和对“tty”结构的引用。

8342：如果字符是回车且终端仅以回车操作，而不是一对回车和换行，那么把字符改为新行;

8344：如果终端不在原始模式下运行，且字符为“quit”或“delete”(7958)，那么就调用“signal”(3949)向每个具有终端的进程发送一个软件中断，刷新与终端关联的所有队列，并返回;

8349：如果原始输入队列增长过大，那么刷新终端的所有队列并返回；初看起来这似乎有点严格，但是这通常都是需要的;

8353：如果终端具有有限的字符集，并且字符是大写字母，则将其转换为小写字母;

8355：把字符插入到原始输入队列;

8356：如果终端以原始模式运行，或该字符是“new line”或“eot”，那么……

8357：唤醒等待终端输入的任何进程，在原始队列中放置一个分隔符(全部)并增加分隔符计数的提示；这是“putc”可能失败的一个点(当没有缓冲区空间)被明确识别；这里发生的故障能解释为何8316行的测试有时会成功。

8361：最后，如果输入字符要被回传，即终端以全双工模式运行，那么把该字符的副本插入到输出队列，并安排将其发送(“ttstart”)回终端。

25.8 字符输出-ttwrite(8550)

当要将输出发送到终端时，这个过程通过“klwrite”(8067)来调用。

8556：如果终端处于非活动状态，则不执行任何操作;

8558：为每个要传输的字符循环...

8560：虽然还有足够数量的字符排队等待传输到终端.....

8561：调用“ttstart”以防万一向终端发送另一个字符的时候;

8562：在这里设置“ASLEEP”标志(也在“wflushtty”(8224)中)是没有意义的，因为从不查询，并且在文件关闭之前从不重置;

8563：休眠，同时，中断处理程序将从输出队列中排出字符，并把它们发送到终端;

8566：调用“ttyoutput”把字符插入输出队列，并安排发送;

8568：再次调用给“ttstart”，祝好运。

25.9 ttstart

每当有理由尝试把下一个字符发送到终端时，就会调用此过程，它常常毫无用处。

8514：见8499行的评论，此代码与此无关;

8518：如果控制器没有准备就绪，即发射器stalus寄存器的第7位未设置，或尚未经过必要的延迟，那么就不执行任何操作;

8520：从输出队列中删除一个字符；如果“c”为正，那么队列不为空，正如预期的那样……

8521：如果“c”小于“0177”，那么它就是要发送的字符……

8522：从数组“partab”的相应元素设置奇偶校验位后，把“c”写入发送器的数据缓冲寄存器，以便启动硬件操作;

8524：否则(“c”>0177)把字符插入到输出队列，以便发信号通知延迟；调用“timeout”(3845)在标注列表中输入一个条目；其结果是在“c＆0177”时钟滴答之后启动“ttrstrt”(8486)的执行；可看出“ttrstrt”再次调用“ttstart”，且操作“TIMEOUT”标志(8524,8491)确保在此期间启动“ttstart”的另一次执行，那么就执行同一个终端，它将(8518)返回而不做任何事情。

25.10 ttrstrt(8486)

请参阅上面第8524行的评论。

25.11 ttyoutput(8373)

此过程在源代码中有更多注释，因此需要的解释少于其它一些注释；注意使用递归(8392)生成一个空格字符串来代替制表符，其它递归调用在第8403和8413行。

25.12 具有受限字符集的终端

8400：“colp”指向一串字符对；如果要输出的字符与这些对中的任何一对的第二个字符匹配，那么字符将被反斜杠替换，后跟该对的第一个字符;

8407：小写字母通过增加常数转换成大写字母；请注意：这里的转换应该与输入反向问题的处理进行比较；这里有一个算法，它明确地交换空间(没有表格类似于“maptab”)的时间(通过8400行的字符串串行搜索)；在“canon”中能采用节省空间的方法，但是那里的问题更加复杂。

8414：把字符插入输出队列，如果是偶然的话，“putc”因缺少缓冲空间而失败，不要担心插入任何后续延迟，或更新系统对当前打印列的想法;

8423：把“colp”设置为指向“tty”结构的“t_col”字符，即“*colp”具有一个值，该值是刚打印的列序号;

8424：把“ctype”设置为与输出字符“c”对应的“partab”元素;

8426：屏蔽掉“ctype”的有效位，并把结果用作切换索引;

8428：(示例0) 常见情况！ 增加“t_col”;

8431:(示例1)非打印字符，该组由ASCII字符集的第1、第2和第4个8位字节组成，加上“so”(016)，“si”(017)和“del”(0177)，不要增加“t_col”;

8434：(示例2) 退格，减去“t_col”，除非它已经为0;

8439: (示例3) 换行，显然“t_col”应该设为0，主要问题是计算在发送另一字符前应发生的延迟。对于Model 37电传打字机，这取决于打印机在页面上的进展程度；选择的值至少为十分之一秒或6个时钟周期，并且可能与((132/16)+3)/60=0.19秒一样多；对于VT05，延迟为0.1秒；对于DECwriter，延迟为0，因为终端包含缓冲存储器并具有双速追赶打印模式;

8451: (示例4) 水平制表符，它把“t_flags”的第10、11位的值赋给“ctype”;

8453：对于唯一识别的非平凡情况(“c”==1或模型37电传打字机)，计算下一个制表位的位置数(通过8454行的模糊计算)；如果结果是4列或更少，那么将它视为0;

8458：舍入“*colp”(即“colp”指向的值)到下一个8减1;

8459：将“*colp”增加为8的精确倍数;

8462 :(示例5）垂直运动，如果把位14设置在“t_fags”中，那么使延迟尽可能长，即0177或127个时钟周期，即仅超过2秒;

8467 :(示例6）回车，把“t_flags”的第12、13位的值赋给“ctype”;

8469：对于第1类，允许延迟5个时钟周期;

8472：对于第2类，允许延迟10个时钟周期;

8475：将“*colp”(打印的最后一列)设置为0；在离开文件“tty.c”之前，有两个问题需要进一步检查。

25.13 A “TTLOWAT”测试(8074行)

在“klxint”的第8074行，测试是否重新启动等待向终端发送输出的任何进程；如果字符数为0或者等于“TTLOWAT”，那么测试成功。如果字符数在这些值之间，那么在队列完全为空之前不执行唤醒，很可能在输出到终端的进程中存在中断。由于在实践中经常观察到输出流的暂时中断，如果不出更多的刺激，那么人们可合理地询问“是否有任何方式能把字符数从大于”TTLOWAT“变为低于它的任何方式。在第8074行没有检测到这一点?

很明显是有的，因为每次调用“ttstart”都能递减少该队列的长度，并且只有一个这样的调用后跟着测试。如果对“ttrstrt”(8492)或“ttwrite”(8568)的任何一个调用“ttstart”恰好越过边界，那么就导致延迟。这种情况发生的概率既小又有限，因此该事件可能在任何合理的长输出序列中被观测到。在另外两种情况下，调用“ttstart”似乎令人满意。队列在“ttwrite”(8561)处于最大范围，并且在“ttyinput”(8363)处有一个前面的“tty-output”调用，它通常但不总是向输出队列添加一个字符。

25.14 B 无效终端

当终端的最后一个特殊文件关闭时，调用“klclose”(8055)并重置(8059)“ISOPEN”和“CARR　ON”标志。然而，接收器控制状态寄存器的“read　enable”位不被复位，因此仍然能接收输入字符且通过“klrint”(8078)把它存储在终端的原始输入队列中(8087)和“ttyinput”(8333)，它不测试“CARR　ON”标志以查看是否终端是逻辑连接的。这些字符可能会累积很长时间并堵塞字符缓冲区存储空间。只有当原始输入队列达到256个字符(“TTYHOG”,8349)时，才会丢弃此队列的内容。因此，似乎应在第8058行之前的“klclose”中包含禁用读取器中断的语句。

25.15 就这样，伙计们...

既然你作为读者，在长期痛苦、精疲力尽时，读完这本书，你一定会毫不费力地完成最后剩下的文件“mem.c”(第90页)。在本章中，我们结束对UNIX操作系统源代码的讨论。当然，还有更多的设备驱动程序供你耐心的检验，事实上，整个UNIX分时操作系统的源代码几乎没有被触及到，因此这不是真的结束。

第六部分、练习

26. 建议的练习

任何操作系统设计都涉及系统设计者的许多主观和临时判断。在UNIX源代码的许多地方，你会发现自己想知道：”为什么他们这样做呢？如果我改变它会怎么样?” 以下练习表达其中一些问题。在更深入的研究之后，你能通过对源代码的检查回答一些问题。其他人需要一些实验性的探测和测量，通过终端对文件“/dev/kmem”的只读访问将证明是有益的；还有一些人确实需要构建和测试操作系统的实验版本。

26.1 第1节

1.1 设计对“malloc”(2528)的更改以实现最佳拟合算法。

1.2 重写“mfree”(2556)程序，使读者更容易辨别其功能。

1.3研究数组“coremap”和“swapmap”(0203,0204)大小是否足够；当“NPROC”增加时，“CMAPSIZ”和“SWAPSIZ”应如何变化?

1.4 证明“malloc”和“mfree”一起正确地解决内存混淆问题。

1.5 通过监视“coremap”的内容，请估算主存储器的使用效率，并估算经常压缩主存“使用区域”的开销，以便降低内存碎片；因此，决定是否值得将当前的内存分配方案扩展到包含内存压缩。

1.6 在设置前六个内核页面描述寄存器时，UNIX不使用可用的所有硬件保护功能，例如：一些只包含纯文本的页面可以是只读的。设计对代码的更改，以最大限度地利用可用的硬件保护。

1.7编译程序

```
        char *init “/etc/init”;
        main ( ) {
        execl (init, init, 0);
        while (1);
}
```

并将结果与数组“icode”(1516)的内容进行比较。

1.8 调查内核态栈区所需的大小，表明所提供的367字区域是足够的。

1.9 如果主存由几个独立的内存模块组成，其中一个模块不是最后一个模块，那么“main”将不包括内存模块之外的内存模块在内存映射“coremap”的可用空间列表中；请设计一些简单的更改为“main”处理这种情况；该系统的其它部分是否还需要修改?

1.10 重写例程“establishur”(1650)和“sureg”(1739)以便它们在PDP11/40上尽可能高效地工作；这些惯例多久用于实践？尝试实施改进版本真的值得吗?

1.11 调查启动新进程所涉及的间接开销，在不同条件下对一组不同大小的程序执行一系列测量。

1.12 评估肯·汤普森打算作为修订调度算法基础的以下方案：

为每个进程保留一个数字“p”，存储为“p_cpu”。每个时钟周期“p”增加1，该进程发现正在执行。因此“p”累积CPU使用率。在每一秒，“p”的每个值被其值的五分之四替换为最接近的整数。这意味着“o”具有以0和方程的解k=0.8*(k+HZ)的解，即4*HZ。如果HZ是50或60且“p”是整数，则“p”能存储在一个字节中。

1.13　始终通过直接线性搜索搜索“proc”表；随着表大小的增加，搜索开销也会增加；在“nPRC”为300的时，研究改进搜索机制的替代方案。

26.2 第2节

2.1 详细说明系统如何对处理器处于内核态时发生的浮点陷阱做出反应。

2.2当进程终止时，会把“zombie”记录写入磁盘，然后由父进程回读，请设计一个方案，将必要的信息传递给父进程，从而避免两个I/O操作的开销。

2.3 文档“备份”(1012)

2.4　使用“shell”设置一组异步进程相对容易，这这会使你的终端淹没无用的输出。试图逐个停止这些进程可能是一个问题，因为它们的识别号可能是未知的。使用命令“kill　0”通常是一种绝望的行为。请设计一种替代方案，例如使用诸如“kill –99”之类的消息，这将是有效的，但更具选择性。

2.5 设计一种协同跳转形式，它能使控制在被跟踪的程序与其父进程之间更有效地传递。

26.3 第3节

3.1 重写“sched”过程以避免使用“goto”语句。

3.2　修改“sched”，如果单个大区域不能立即可用，那么程序的文本段和数据段可能会分配在单独的主存中。

3.3　如果系统崩溃且必须重新启动(reboot)，那么崩溃时未写出的缓冲区内容将丢失。但是，如果采用核心转储(core　dump)，那么就能获取缓冲区的内容，从而能使磁盘的内容保持最新；请你概述执行该计划的一个独立计划，你认为它会有多有效?

3.4 解释为什么在第4720行上声明的缓冲区是514个、而不是512个字符长。

3.5 解释如果可用的大缓冲区太少，可能会出现死锁情况；假设无法增加缓冲区的数量，你建议采取哪些措施缓解此问题。

第四部分

4.1 设计一种用于标记文件系统卷的方案，并在安装卷时检查这些标签。

4.2 讨论在UNIX下支持ANSI标准标签磁带的问题，并提出一个解决方案。

4.3 设计一种为文件提供索引顺序访问的方案。

4.4“stickybit”的出现(参见PM手册的“CHMOD(I)证实连续分配文件的所有空间中存在一些剩余的优点，请讨论使“连续文件”更普遍可用的优点。

4.5 设计一种测量管道效率的技术，应用该技术并报告您的结果。

4.6 根据以下方案设计对“pipe.c”的修改，这将使管道更有效：每当读取指针大于512时，旋转“inode”中的非空块编号，并把读取和写入指针递减512。

第五部分

5.1 通过监视空闲缓冲区的数量或以其它方式，确定你安装时提供的字符缓冲区数量是否足够。

5.2 执行测量和/或实验，以确定字符缓冲块是否将更有效地利用，如果每个块是由4个或8个字符组成，而不是6个。

5.3 重新设计行式打印机驱动程序，在最小化打印周期数的意义上更有效地处理套印和退格。

5.4文档“mmread”(0916)和“mmwrite”(9042)。

一般情况

6.1 改变OS使用的主存空间的最简单方法是改变“NBUF”。如果禁止这样做，请你建议最好的方法：

（a）减少500字所需的空间；（b）额外使用500字。

6.2讨论C作为系统编程语言的优点，它缺少哪些功能？还是有多余的功能?

罗斯跋

本书首次出现是以行式打印机列表的形式，当时是1976年，新南威尔士大学把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·莱昂斯副教授讲授，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。学校购置多种DEC公司的PDP-11计算机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，ACM计算机协会通信协会发表里奇和汤姆森的一篇论述“UNIX分时系统”的论文。由于该系统是在几台PDP-11计算机上运行，因此实际上是能自由使用的。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且能在CDC Cyber电脑系统上使用。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。当时按照学校的意见，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，因为莱昂斯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对世界上其它大学的意义一样。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，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，用于承接每天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UNIX 的发展。《莱昂斯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1970年代后期和1980年代初期的成功发展作出巨大贡献。

有一个事实可用来衡量《莱昂斯UNIX源代码分析》一书取得何等成功，这就是它们的大量影印本的四处流行。我在1987年访问过一家小型计算机公司，居然在该公司的书架上发现《莱昂斯UNIX源代码分析》的影印本。我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，纵然是近乎无法想象的，但依然通行无阻。

约翰·莱昂以其特有的方式写作《莱昂斯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。其它课程牵涉的都是小程序，即学生能够管理的程序，而UNIX操作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，一般人难得一见；UNIX操作系统是是写得很好的程序，大家都有机会去看它的源代码。在以后的几年中，教学生们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v6的核心系统的结构完美、程序简约。直到今天，尚未发现其它程序能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代码。

莱昂斯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《莱昂斯UNIX源代码分析》的新版写这篇后记，我感到非常愉快和荣幸。这项任务本应由约翰本人承担。但不幸的是约翰的健康状况欠佳，不允许他做这件事情。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