第1章　硬件基础与软件基础

1.1　硬件基础

操作系统必须和作为其基础的硬件系统紧密地协同工作。操作系统需要只有硬件能提供的特定服务。为了完全理解Linux操作系统，需要了解它下层的硬件基础知识。本节将简短介绍该硬件：现代PC。

当以Altair 8080机器的图解作为封面的1975年1月份的《大众电子》杂志印刷时，一场“革命”开始了。家庭电子爱好者仅花397美元就可以组装出一台以早些时候的电影“星际旅行”中的一个目的地而命名的Altair 8080。它的Intel 8080处理器和256字节的存储器而没有屏幕和键盘用今天的标准看来是多么弱小。它的发明者Ed. Roberts创造了“个人计算机”一词来描述自己的新发明，但今天PC一词被用来指几乎任何你不需帮助就可以得到的计算机。从这个定义上说，甚至一些具有强大能力的Alpha AXP系统也是PC。

狂热的黑客们看到Altair的潜力并开始为它写软件和建造硬件。对于这些早期的先行者来说，它代表着自由：不用在巨大的批处理大型机系统上运行和被“精英们”监视的自由。许多被这种新东西——一台可以放在家中厨房里桌子上的计算机迷住的大学辍学者一夜之间而暴富。许多硬件出现了，在某种程度上都不相同，而软件黑客很乐意为这些新机器写软件。然而IBM坚实地建造了现代PC的模型，它们1981年发布IBM PC并于1982年早期开始销售给客户。它有Intel 8088处理器、64KB内存(可扩充至256KB)、两个软盘和一个25行80字符的彩色图形适配器(CGA)，这在今天的标准看来仍不很强大但却销售得很好。接着是1983年的IBM PC-XT，有了“奢侈”的10MB字节的硬盘。不久，许多诸如Compaq这样的公司开始生产IBM PC兼容机，PC的体系结构成为一个事实标准。这个事实标准有助于许多的硬件公司在一个不断增长的市场中一起竞争，从而保持价格很低，使消费者受益。这些早期PC的许多系统结构特征一起保持到当今的PC。例如，即使是最强大的基于Intel Pentium Pro的系统启动时也运行于Intel 8086的寻址模式下。当Linus Torvalds开始写后来成为Linux的东西时，就选择了最普遍和合理价格的硬件，Intel 80386 PC。

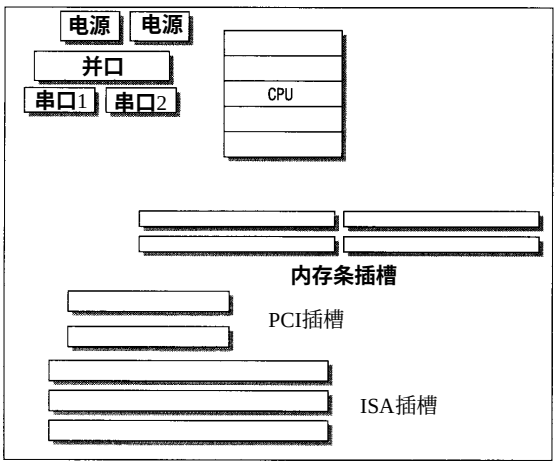


图1-1-1　典型的PC主板

从PC的外面来看，最明显的部件是机箱、键盘、鼠标和图形监视器。机箱前面是一些按钮、一个显示数字的小显示器和一个软驱。现在的大多数系统有 CD-ROM，并且如果你觉得有必要保护数据的话，还可以有一台磁带驱动器作备份用。这些设备被统称做外设。

尽管CPU在总体上控制系统，它并非唯一的智能设备。所有的外设控制器，比如 IDE控制器，都具有一定的智能。在 PC内部，有一块主板(见图1-1-1)，上面有CPU或称微处理器、内存条插槽和一些ISA或PCI外设控制器的插槽。有些控制器，如 IDE磁盘控制器可以直接建在系统主板上。

1.1.1 CPU

CPU或叫微处理器，是计算机系统的心脏。微处理器通过从内存中读取指令并执行进行计算、逻辑操作以及数据流管理。在早期计算中微处理器的功能部件是分离的 (物理上很大的)单元。就是那时创造了中央处理单元 (Central Processing Units)的术语。现代的微处理器把这些部件组合在蚀刻于很小的硅片上的集成电路中。 CPU、微处理器(microprocessor)、处理器(processor)三个词在本书中通用。

微处理器操作由0和1组成的二进制数据，这些 0和1对应于电子开关的打开或关闭。如十进制的42表示“ 4个10和2个1 ”，一个二进制数是表示 2的幂的一串二进制数。在这里幂是指一个数乘以自身的次数。 10的1次幂(10^1)是10，10的2次幂(10^2)是10×10，10的三次幂(10^3)是10×10×10，依此类推。二进制 0001是十进制1，二进制0010是十进制2，二进制0011是3，二进制0100是4，等等。这样，十进制 42的二进制就是101010即($2+8+32$或$2^1+2^3+2^5$)。在计算机程序中通常不用二进制表示数据，而用另一种基数，十六进制表示。在这种表示下，每个数位表示一个16的幂。因为十进制数只有 0到9，数10到15用字母A、B、C、D、E、F表示成单个数位。例如，十六进制 E是十进制14，十六进制2A是十进制42(两个16加上10)。用C语言的表示方法 (正如在本书通篇中所做的)，十六进制要加前缀“ 0x ”；十六进制数 2A被写作0x2A。

微处理器可以进行算术运算，如加、减、乘、除和逻辑运算，如“ X是否比Y大? ”。

处理器的执行被外部时钟所驱动。这个时钟，即系统时钟，产生规则的时钟脉冲到处理器，而处理器在每一个时钟脉冲做一些工作。比如，处理器可以在每个时钟脉冲执行一条指令。处理器的速度用系统时钟跳动的速度来描述。一个 100MHz的处理器每秒钟将收到 100 000 000个时钟脉冲。用时钟脉冲来描述 CPU的能力有误导性，因为不同的处理器在一个时钟脉冲期间完成不同量的工作。但是，在其它所有东西都一样时，速度更快的时钟意味着计算能力更强的处理器。处理器执行的指令都很简单，比如像“ 将存储器 X位置的内容读到Y寄存器 ”。寄存器是微处理器的内部存储区，用来存储数据和在其上面执行操作。执行的操作可能会引起停下它正在做的东西并跳转到存储器中其它地方的某条指令。这些微小的组成单元赋予当今的微处理器几乎无穷的能力，它们能够每秒钟执行数百万条甚至上十亿条指令。

指令在执行前必须先从存储器中取出。指令自身可以引用存储器中的数据，该数据必须被从存储器中取出并在适当的时候存回去。

微处理器内部的寄存器的大小、数量和类型完全取决于微处理器的类型。 Intel 80486处理器和Alpha AXP处理器就有不同的寄存器集；首先， Intel的是32位宽，而Alpha AXP的是64位宽。一般说来，任何微处理器都会有一定数量的通用寄存器和少量的专用寄存器。大多数处

理器有以下的专用寄存器：

程序计数器(Program Counter,PC)：该寄存器包含将被执行的下条指令的地址。每当一条指令取出后PC的值将被自动增量。

栈指针(Stack Pointer,SP)：处理器必须能够存取大量的外部随机读 /写存储器(RAM)，以存储临时数据。栈就是一种在外部存储器中方便地存储和恢复数据的方式。通常处理器有专门指令让你把值压到栈上，并在晚些时候将它们弹出。栈工作于后进先出 (Last In First Out, LIFO)的基础上。也就是说，如果你压两个值，即 X和Y到栈上，然后弹出一个值，将会得到后压进的Y的值。

有些处理器的栈朝存储器顶端向上增长，而另一些朝存储器底端即基端向下增长。有的处理器支持这两种，比如ARM。

处理器状态(Processor Status,PS)：指令可能产生结果；比如“ 寄存器 X的值是否大于寄存器Y的值 ”将产生真或假作为结果。 PS寄存器保存这种和其它的当前处理器的状态信息。例如，大部分处理器至少有两种操作模式，核心 (或管理)模式和用户模式。 PS寄存器中保留有识别当前操作模式的信息。

1.1.2 存储器

所有系统都有一个存储器层次结构，在这个层次结构的不同层上有不同速度和大小的存储器。速度最快的存储器就是我们所知道的高速缓存。就像听起来的那样，它是用来暂时保留或缓存主存储器内容的存储器。这种存储器速度很快但也很贵，所以大多数系统有少量的片上(on-chip)缓存和稍多的系统级(板上)缓存。有的处理器用一个缓存保存指令和数据，但其它的处理器有两个缓存，一个指令缓存和一个数据缓存。 Alpha AXP处理器就有两个内部缓存：一个是数据的(D-Cache)，一个是指令的(I-Cache)。外部缓存(B-Cache)将两者合在了一起。最后是主存储器，相对于外部缓存来说是很慢的。相对于 CPU片上缓存，主存慢得就像爬一样。

高速缓存和主存储器必须保持同步 (一致)。也就是说，如果主存储器的一个字保存在高速缓存的一个或多个位置，则系统必须要保证高速缓存和主存储器的内容是相同的。高速缓存一致性的工作一部分由硬件完成，一部分由操作系统完成，许多主要的系统任务也是这样，要求硬件和软件紧密配合来达到目标。

1.1.3 总线

系统主板上的单个部件通过被称为总线 (bus)的许多系统连接通路相连。系统总线在逻辑功能上分为三类：地址总线、数据总线和控制总线。地址总线用来为数据传送指明存储器位置(地址)。数据总线保持传送的数据。数据总线是双向的，它允许数据读入到 CPU和从CPU写出。控制总线包括各种各样的线路用来在系统中传送定时和控制信号。存在许多种总线，像ISA和PCI就是连接外设到系统的常用总线方式。

1.1.4 控制器和外设

外设是实在的设备，像图形卡或磁盘。它们受系统主板上的控制器芯片或插到主板上的控制器卡的控制。IDE磁盘用IDE控制器芯片控制、SCSI磁盘用SCSI磁盘控制器芯片控制等等。

这些控制器通过一组总线连接到 CPU及相互连接。大部分现在制造的系统使用 PCI和ISA总线连接主要的系统部件。控制器是像 CPU一样的处理器，它们可以被看作 CPU的智能化助手。CPU对系统整体进行控制。

所有的控制器都不相同，但通常都有一些寄存器控制它们。在 CPU上运行的软件必须能够读写这些控制寄存器。一个寄存器可能包含描述出错状态的信息。另一个可能被用作控制目的，来改变控制器的模式。总线上的每个控制器可以被 CPU单独寻址，这样软件的设备驱动程序能够写到它的寄存器中以控制它。 IDE ribbon就是个很好的例子，它赋予你单独访问总线上每个驱动器的能力。另一个不错的例子是 PCI总线，允许每个设备(如图形卡)独立地被访问。

1.1.5 地址空间

系统中连接CPU和主存的总线与连接 CPU和系统的硬件外设的总线是分开的。硬件外设所占用的存储器空间被总称为I/O空间。I/O空间本身可以再细分下去，但我们现在先不用考虑那么多。CPU能够存取系统空间存储器和 I/O空间存储器，而控制器自身只有在 CPU的帮助下间接地访问系统存储器。从设备的角度，比如软盘控制器，它只能看到自己的控制寄存器所在的空间(ISA)，而不能看到系统存储器。典型情况是， CPU有分开的指令访问存储器和I/O空间。例如，可能有一条指令要“ 从 I/O地址0x3f0读一个字节到寄存器 X中。”CPU就是这样控制系统的硬件外设 —— 通过读写它们在I/O空间中的寄存器。在PC体系结构发展的这么多年里，一般的外设(IDE控制器、串口、硬盘控制器等)的寄存器在什么地方 (地址)已经成为习惯。 I/O空间地址0x3f0正好是一个串口(COM1)的控制寄存器地址。

有时控制器需要直接读或写系统存储器中的大量数据，例如用户数据被写到硬盘时。在这种情况下，直接存储器访问 (Direct Memory Access,DMA)控制器将被使用以允许硬件外设直接访问内存，但这种访问是处在 CPU的严格控制和管理之下的。

1.1.6 时钟

所有的操作系统都需要知道时间，所以当代 PC都包含一个特殊的外设叫实时时钟 (Real Time Clock,RTC)。它提供两样东西：一个可靠的日期时间和一个准确的定时间隔。 RTC有自己的电池，所以当PC断电的时候它继续运行，这就是为什么你的 PC总是知道正确的日期和时间的原因。间隔定时器允许操作系统准确地调度必需的工作。

1.2 软件基础

程序是完成特定任务的计算机指令集合。程序可以用汇编，一种很低级的计算机语言写成，也可以用高级的、与机器无关的语言比如 C语言写成。操作系统是一个特殊的程序，使用户能够运行像表格或字处理这样的应用程序。本节介绍基本编程原理并给出操作系统的功能和目标的一个概述。

1.2.1 计算机语言

1. 汇编语言

CPU从主存取出并执行的指令对于人是根本不能理解的。它们是机器代码，精确地告诉

机器干什么。十六进制数0x89E5是一条Intel 80486指令，将ESP寄存器的内容拷贝到EBP寄存器中。为最早的计算机发明的软件工具之一是汇编器，一个输入人可读的源文件并把它汇编成机器代码的程序。汇编语言显式地处理寄存器和对数据的操作，并且它们是针对特定微处理器的。Intel X86微处理器的汇编语言与Alpha AXP微处理器的汇编语言有很大差别。下面的Alpha AXP汇编代码展示了一个程序可能执行的操作：

```
        ldr r16, (r15)          ：第一行
        ldr r17, 4(r15)         ：第二行
        beq r16,r17,100         ：第三行
        str r17, (r15)          ：第四行
100:                            ：第五行
```

第一个语句(第一行)从寄存器15保存的地址装载寄存器16。下一条指令从存储器下一个位置装载寄存器17。第三行比较寄存器16和寄存器17的内容，如果它们相等，就转移到标号100。如果寄存器中的值不等则程序继续执行第4行，将寄存器17的内容存到存储器。如果寄存器确实给相同内容则不必存储任何数据。汇编程序冗长、难写而又易于出错。Linux内核只有很少一部分是为了高效而用汇编语言写的，那些部分是针对特定微处理器的。

2. C语言和编译器

用汇编语言写大型程序是困难而费时的工作。它很容易产生错误，并且产生的程序不可移植，被限定在一个系列的微处理器上。使用像C [7, C Programing Language]这样的机器无关语言要好得多。C使得你可以用逻辑算法和操作的数据来描述程序。称为编译器的特殊程序读进C程序并把它翻译成汇编语言，再从它产生针对特定机器的代码。好的编译器能够产生接近优秀汇编程序员所写的那样高效的汇编指令。大部分Linux内核是用C语言写的。下面的C程序片断：

```
if (x != y)
        x = y ;
```

执行和前面例子的汇编代码一模一样的操作。如果变量x的内容和变量y的内容不相同，那么y的内容将被拷贝到x。C语言被组织成例程，每个例程执行一件任务。例程可以返回C语言所支持的任何值或数据类型。像Linux内核这样的大型程序包括许多独立的C源程序模块，每个模块有自己的例程和结构。这些C源程序代码模块在一起组合成像文件系统处理这样的逻辑功能。

C支持许多类型的变量，一个变量就是一个可以用符号名字引用的存储器位置。在上面的片断中x和y就引用存储器的位置。程序员不管变量被放在存储器中什么地方，那是连接器(linker)所关心的。有些变量包括不同类型的数据、整数、浮点数，还有一些是指针。

指针是包含其它变量的地址即它在存储器中位置的变量。考虑一个变量x，可能位于内存中地址0x80010000处。你可以有一个指针px指向x。px可能位于内存中地址0x80010030处。px的值是0x80010000：x的地址。

C允许你将相关的变量捆在一起成为数据结构。例如：

```
struct {
        int i ;
        char b ;
} my_struct ;
```

是一个叫做my_struct的数据结构，它包含两个元素，一个叫 i的整数(32位数据存储)和一个叫b的字符(8位数据存储)。

3. 连接器

连接器是一个程序，它将几个目标模块和库连接在一起形成一个单一的、一致的程序。目标模块是编译器或汇编器输出的机器代码，包含机器可执行的代码和数据及使连接器把模块们组装在一起形成一个程序的信息。比如一个模块可能包含一个程序的所有数据库功能而另一个则包含其命令行参数处理功能。如果在一个模块中引用的例程和数据结构确实存在于另一模块中的话，连接器将安排好目标模块间的引用。 Linux内核就是由它的许多组成目标模块连接在一起形成的单一大型程序。

1.2.2 什么是操作系统

一台计算机如果没有软件只不过是一堆散发热量的电子器件。如果硬件是计算机的心脏的话，软件就是它的灵魂。操作系统是一些允许用户运行应用程序的系统程序的集合。操作系统抽象了系统的真实的硬件，提供给用户及其应用程序一个虚拟机器。在很大程度上，软件体现系统的特征。大部分 PC能够运行一个或多个操作系统，每个都有不同的外观和感觉。Linux由一些功能独立的部分组成，它们一起构成了操作系统。 Linux一个很重要的部分就是内核本身，但是，它离开shell和库也是没有用的。

为了开始理解什么是操作系统，考虑一下你敲入一个简单的命令后发生了什么：

```
$ ls
Mail            c               images          perl
docs            tcl
$
```

$符号是注册Shell(如果是bash的话)输出的一个提示符，它意味着正在等待用户输入一些命令。键入ls，键盘驱动程序会识别出这些字符被敲入了。键盘驱动程序将它传给 Shell，由它寻找一个具有相同名字的可执行映像来处理该命令。它在 /bin/ls找到映像，然后调用内核(kernel)服务，把ls可执行映像装入虚拟内存并开始执行。 ls映像调用内核的文件子系统来查找有什么文件存在。文件系统可能使用缓存的文件系统信息或利用磁盘设备驱动程序从磁盘上读取该信息。它甚至可能引起网络驱动程序同一个远程机器交换信息以获得本系统访问的远程文件的细节(文件系统可以通过网络文件系统即 NFS远程安装)。不管该信息通过什么方式得到，ls将输出该信息，由图形驱动程序在屏幕上显示出来。

上面的这些内容看起来很复杂，但它表明即使是最简单的命令也能说明操作系统事实上是合作的功能的集合，它给予用户一个系统的一致的视图。

1. 存储器管理

在资源有限的情况下，比如存储器，操作系统需要做的很多事情就是冗余。操作系统的许多基本技巧之一就是使少量的物理存储器用起来就像许多存储器一样。这些表面上的大量存储器就是虚拟存储器。其思想是系统上运行的软件被“ 欺骗 ”，认为自己在大量存储器中运行。系统把存储器分成容易处理的页面，在运行时，把这些页面交换到内存上。因为有另一个技巧 —— 多进程的存在，所以软件却感觉不到这一点。

2. 进程

一个进程可以被想象成一个运行的程序，每个进程都是一个运行特定程序的独立实体。

如果你查看一下Linux系统上的进程，就会发现有许多进程。例如，在机器上敲入 ps将显示下列进程：

```
$ ps
  PID TTY STAT  TIME COMMAND
  158 pRe 1     0:00 -bash
  174 pRe 1     0:00 sh /usr/X11R6/bin/startx
  175 pRe 1     0:00 xinit /usr/X11R6/lib/X11/xinit/xinitrc --
  178 pRe 1 N   0:00 bowman
  182 pRe 1 N   0:01 rxvt -geometry 120x35 -fg white -bg black
  184 pRe 1 <   0:00 xclock -bg grey -geometry -1500-1500 -padding 0
  185 pRe 1 <   0:00 xload -bg grey -geometry -0-0 -label xload
  187 pp6 1     9:26 /bin/bash
  202 pRe 1 N   0:00 rxvt -geometry 120x35 -fg white -bg black
  203 ppc 2     0:00 /bin/bash
 1796 pRe 1 N   0:00 rxvt -geometry 120x35 -fg white -bg black
 1797 v06 1     0:00 /bin/bash
 3056 pp6 3 <   0:02 emacs intro/introduction.tex
 3270 pp6 3     0:00 ps
$
```

如果机器中有许多CPU，那么每个进程就能（至少理论上能）在不同的CPU上运行。不幸的是只有一个CPU，所以操作系统又得使用技巧，把每个进程依次运行一段很短的时间。这一段时间就是我们所知的时间片 (time-slice)。这个技巧叫作多进程或调度，它骗使每个进程都以为自己是唯一的进程。进程相互之间受到保护，所以当一个进程崩溃或出错时不会影响其它的进程。操作系统通过给每个进程一个独立的、只有它自己能访问的地址空间来达到这个目的。

3. 设备驱动程序

设备驱动程序构成Linux内核的主要部分。像操作系统的其它部分一样，它们在高特权的环境下操作，如果它们出错可能引起灾难。设备驱动程序管理操作系统及其控制的硬件设备之间的交互。例如，文件系统在写文件块到 1DE磁盘上时使用一个通用块设备接口。驱动程序进行细节操作和设备相关的操作。设备驱动程序针对它们驱动的特定的控制器芯片，所以如果你的系统中有一块NCR810 SCSI控制器的话，就需要有NCR810 SCSI驱动程序。

4. 文件系统

Linux像UNIX一样，系统使用逻辑上独立的文件系统而不是实际的设备标识符（比如驱动器名或驱动器号）来进行文件访问， Linux的每个新文件系统都被安装到根文件系统的某个目录上(比如/mnt/cdrom)，这样这个新文件系统就被合并到单一的根文件系统树中。 Linux最重要的特征之一就是支持多种不同的文件系统。 Linux上最流行的文件系统是 EXT2文件系统，它也是大部分发布的Linux都支持的文件系统。

文件系统提供给用户一个系统硬盘上的文件和目录的一个合理的视图，而不管文件系统的类型和底层物理设备的特征如何。 Linux透明地支持许多不同的文件系统（如MS-DOS和EXT2)，并把所有安装的文件和文件系统表示成一个集成的虚拟文件系统。这样，一般说来，用户和进程不需要知道一个文件是哪种文件系统的一部分，而只管使用就是了。

块设备驱动程序把不同类型的物理块设备 (如IDE和SCSI)之间的差别隐藏起来，并且，对

每个文件系统来说，物理设备只是数据块的线性集合。不同的设备会有不同的块大小，例如软盘通常为512字节，而IDE设备通常为1024字节；同样，这对系统的使用者是隐藏的。一个EXT2文件系统不管保存在什么设备上看起来都一样。

1.2.3 内核数据结构

操作系统必须保持许多关于系统当前状态的信息。随着系统中事件的发生，这些数据结构也要被改变以反映当前现实。例如，当一个用户登录系统时，一个进程可能被创建。内核必须创建一个表示这个进程的数据结构并把它链接到表示系统中其它所有进程的数据结构上。

通常这些数据结构存在于物理内存中，并且只能被内核及其子系统访问。数据结构包含数据和指针，其它数据结构或例程的地址。

放在一起，Linux内核使用的数据结构看起来会很迷惑。每个数据结构有自己的用途。尽管有些是被几个内核子系统使用，但它们比乍一看起来要简单得多。

理解Linux内核关键是理解它的数据结构及 Linux内核用它们所完成的功能。本书把对Linux内核的描述建立在其数据结构的基础上。本书通过算法、完成功能的方法以及对数据结构的使用等来讨论每个内核子系统。

1. 链表

Linux使用一些软件工程技巧来把数据结构链接在一起。在许多情况下它使用链接的(linked)或链状的(chained)数据结构。若每个数据结构描述某个事物的单个实例或出现，内核必须能够找到所有的实例。在链表中一个根指针包含表中第一个数据即元素的地址，而每个数据结构包含一个指针指向表中下一个元素。最后一个元素的下一个指针为空，这意味着它是表中的最后一个元素。双向链表包含一个指针指向表中下一个元素，同时包含一个指针指向表中前一个元素。使用双向链表使得在表的中间添加或删除元素变得容易，尽管需要更多访存操作。这是一个典型的操作系统折衷：以内存访问换取 CPU周期。

2. 散列表

链表是一种把数据结构链在一起的简便方法，但查找链表的效率会很低。如果你正在搜索一个特定元素，可能看完整个表才找到所需要的。 Linux使用另一种技巧 —— 散列(hashing)来解除这种限制。一个散列表是一个指针的数组或向量。数组或向量就是在内存中一个换一个的事物的简单集合。一个书架(上的书)可以说是一个书的数组。数组通过索引来访问，索引就是数组的偏移。进一步拿书架作类比，你可以通过它在书架上的位置来描述每本书，比如你可能要第5本书。

散列表是数据结构指针的数组，而它的索引是通过这些数据结构中的信息导出的。如果你用一个数据结构描述一个村子的人口，那么就可以人的年龄作为索引。为了找到一个特定的人的数据就可以用他的年龄作为人口散列表的索引，然后沿着指针找到包含该人的细节的数据结构。不幸的是一个村的许多人很可能具有相同的年龄，所以散列表中的指针成为指向数据结构链或表的指针，每个数据结构描述同年龄的一个人。搜索这些短的链仍比搜索全部数据结构快得多。

因为散列表加速了对经常使用的数据结构的访问， Linux经常使用散列表来实现高速缓存，高速缓存是需要快速访问的信息，并且通常是可以得到的完整信息集合的一个子集。数据结构被放进高速缓存并保留在那里，因为内核要经常访问它们。高速缓存有一个缺点就是它们

在使用和维护上比简单链表或散列表更复杂。如果数据结构能在高速缓存中找到 (即高速缓存命中)，那一切都好。否则，所有相关的数据结构都要被搜索，并且，如果该数据结构确实存在，它就必须被加入到高速缓存中。在向高速缓存中加入新数据结构时，一个老的数据结构可能会被淘汰出去。Linux必须决定淘汰哪一个，而危险在于淘汰的数据结构可能正是 Linux 下一个所需要的。

3. 抽象接口

Linux内核经常抽象其接口。接口是按特定方式操作的例程和数据结构的集合。例如所有的网络设备驱动程序必须提供一定的例程，在这些例程中操作特定的数据结构。在这种方式下可以有使用专用代码的低层的服务 (接口)的通用层代码。网络层是通用的，它被遵守标准接口的设备专用代码支持。

通常这些低层在启动时向高层注册 (register)。这种注册通常涉及向一个链表中加入一个数据结构。例如，内核中每个文件系统在启动时向内核注册自己；或者，如果你在使用模块，当该文件系统首次被使用时注册。通过查看文件 /proc/filesystems你可以发现哪些文件系统注册了自己。注册数据结构通常包含函数指针。它们就是完成特定任务的软件函数的地址。再一次以文件系统注册为例，每个文件系统在注册时传给内核的数据结构包含文件系统专用例程的地址，当文件系统被安装时，这些例程将被调用。

第2章 内存管理

内存管理子系统是操作系统最重要的部分之一。从早期计算开始，系统的内存大小就难以满足人们的需要。为了解决这个问题，可利用虚拟内存。虚拟内存通过当需要时在竞争的进程之间共享内存，使系统显得有比实际上更多的内存空间。

虚拟内存不仅仅使机器上的内存变多，内存管理子系统还提供以下功能：

- 大地址空间　操作系统使系统显得它有比实际上大得多的内存。虚拟内存可以比系统中的物理内存大许多倍。
- 保护　系统中每个进程有自己的虚拟地址空间。这些虚拟地址空间相互之间完全分离，所以运行一个应用的进程不能影响其他的进程。同样，硬件的虚拟内存机制允许内存区域被写保护。这样保护了代码和数据不被恶意应用重写。
- 内存映射　内存映射用来把映像和数据文件映像到一个进程的地址空间。在内存映射中，文件的内容被直接链接到进程的虚拟地址空间。
- 公平物理内存分配　内存管理子系统给予系统中运行的每个进程公平的一份系统物理内存。
- 共享虚拟内存　尽管虚拟内存允许进程拥有分隔的(虚拟)地址空间，有时你会需要进程共享内存。例如系统中可能会有几个进程运行命令解释 shell bash。最好是在物理内存中只有一份bash拷贝，所有运行bash的进程共享它；而不是有几份 bash拷贝，每个进程虚拟空间一个。动态库是另一个常见的几个进程共享执行代码的例子。

共享内存也可以被用作进程间通信(IPC)机制，两个或更多进程通过共有的内存交换信息。Linux支持Unix(tm) System V的共享内存IPC。

2.1 虚拟内存抽象模型

在考察Linux支持虚拟内存所使用的方法之前，考察一下抽象模型会有所帮助。

当处理器运行一个程序时，它从内存中读取一条指令并解码。在解码该指令过程中它可能需要取出或存放内存某个位置的内容。处理器然后执行该指令并移动到程序中下一条指令。这样处理器总是访问内存来取指令或取存数据。

在虚拟内存系统中以上所有的地址都是虚拟地址而不是物理地址。处理器基于由操作系统维护的一组表中的信息，将虚拟地址转换成物理地址。

为了使这种变换容易一些，虚拟内存和物理内存都被分为合适大小的块叫做“页(page)”。这些页都有同样的大小。它们可以不具有同样大小，但那样的话系统将很难管理。 Alpha AXP系统上Linux使用8KB字节大小的页，Intel x86系统上使用4KB字节大小的页。这些页中每一个都有一个唯一的号码：页帧号(Page Frame Number, PFN)。在这种分页模型中，一个虚拟地址由两部分组成：一个偏移和一个虚拟页帧号。如果页大小是 4KB字节，虚拟地址的11：0位包含偏移，12位及高位是虚拟页帧号。每当处理器面临一个虚拟地址时，它必须析取出偏移和虚拟页帧号。处理器必须将虚拟页帧号转换成物理的页帧号，然后在该物理页中正确的偏移

位置上进行访问。为了完成这些处理器要使用页表。

图1-2-1展示了两个进程的虚拟地址空间，进程 X和进程Y，每个都有自己的页表。

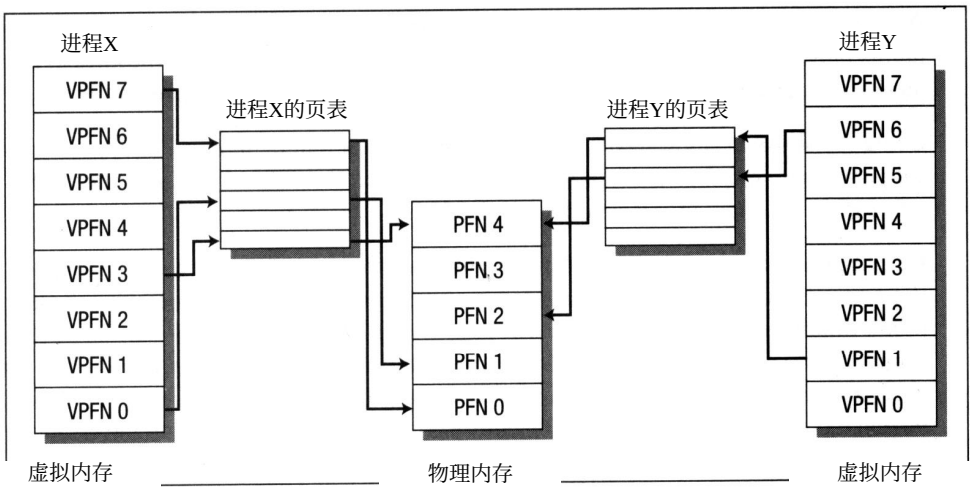


图1-2-1 虚拟地址到物理地址映射的抽象模型

这些页表将每个进程的虚拟页映射到内存中的物理页。图 1-2-1显示进程X的虚拟页帧号0映射到内存中物理页帧号 1，进程Y的虚拟页帧号1映射到物理页帧号4。理论上的页表中每一项包含下列信息：

- 有效标志，用来指示该页表项是否有效。
- 本项所描述的物理页帧号。
- 访问控制信息。它描述该页可以被怎样使用。它是否可以被写？是否包含可执行代码？

页表用虚拟页帧号作为偏移来访问。虚拟页帧号 5将是表中第6个元素(0是第1个)。

为了将一个虚拟地址转换成物理地址，处理器首先必须得到虚拟地址页帧号和在该虚拟页中的偏移。通过选取页大小为 2的幂，这可以很容易地屏蔽和移位得到。再看一下图 1-2-1，设页大小是0x2000字节(即十进制8192)，Y进程虚拟地址空间中一个地址为 0x2194，则处理器将把该地址转换为虚拟页帧号 1的偏移0x194。

处理器使用虚拟页帧号作为进程页表的索引来检索它的页表项。如果该偏移处页表项有效，处理器将从该项取出物理页帧号。如果该页表项无效，说明处理器访问了虚拟内存中不存在的区域。在这种情况下，处理器不能解析该地址，并且必须把控制传给操作系统来解决问题。

处理器如何通知操作系统一个正确的进程试图访问一个没有有效转换的虚拟地址，这是依处理器不同而不同的。无论如何处理器能够处理它，这被称作“ 页故障 (page fault) ”。操作系统被告知故障的虚拟地址和故障原因。

如果访问的是有效的页表项，处理器取出物理页帧号，并将它乘以页的大小以得到物理内存中该页的基地址。最后，处理器将偏移加到所需的指令或数据的地址。

再以上面的例子为例，进程 Y的虚拟页帧号 1映射到物理页帧号 4，起始于0x8000(4×0x2000)。加上0x194字节的偏移，最后得到物理地址 0x8194。

通过用这种方式映射虚拟地址和物理地址，虚拟内存能够以任何次序被映射到系统物理页。例如，在图1-2-1中进程X的虚拟页帧号0被映射到物理页帧号1，而虚拟页帧号7被映射到物理页帧号0，尽管在虚拟内存中它比虚拟页帧号 0要高。这说明了虚拟内存的一个有趣的副作用：虚拟内存页不必以任何特定次序出现在物理内存中。

2.1.1 请求调页

因为物理内存比虚拟内存小得多，操作系统必须小心以高效地利用物理内存。一种节约物理内存的方法是只装载被执行的程序当前正在使用的虚拟页。例如：一个数据库应用程序可能被运行来查询一个数据库。这种情况下，并非数据库的全部都需要被装入内存，而只是装入那些被检查的数据记录。如果数据库查询是一个搜索查询，那么把处理添加新记录的代码从数据库程序中装载进来是毫无意义的。这种只在被访问时把虚拟页装入内存的技巧叫请求调页。

当进程试图访问一个当前不在内存中的虚拟地址时，处理器将不能为被引用的虚拟页找到页表项。例如：图 1-2-1中，进程X的页表中没有虚拟页帧号 2的页表项，所以当 X试图从虚拟页帧号2中读一个地址时，处理器将不能把该地址转换成物理地址。这时处理器通知操作系统发生了页故障。

如果故障的虚拟地址无效，这意味着，该进程试图访问一个不该访问的地址。或许应用程序出了些错误，比如写内存中的随机地址。这种情况下操作系统将终止它，保护系统中其他进程不受此恶意进程破坏。

如果故障的虚拟地址有效，但它所引用的页当前不在内存中，操作系统就必须从磁盘上的映像中把适当的页取到内存中，相对来说，磁盘访问需要很长时间，所以进程必须等待一会儿直到该页被取出。如果还有其他可以运行的进程，操作系统将选择其中一个来运行。被取出的页会被写到一个空闲的物理页帧，该虚拟页帧号的页表项也被加入到进程页表中。然后进程从出现内存故障的机器指令处重新开始。这次进行虚拟内存访问时，处理器能够进行虚拟地址到物理地址的转换，进程将继续执行。

Linux使用请求调页把可执行映像装入进程虚拟内存中。每当一个命令被执行时，包含该命令的文件被打开，它的内容被映射到进程虚拟空间。这是通过修改描述进程内存映射的数据结构来完成的，被称作“ 内存映射 ”。然后，只有映像的开始部分被实际装入物理内存，映像其余部分留在磁盘上。随着进程的执行，它会产生页故障， Linux使用进程内存映射以决定映像的哪一部分被装入内存去执行。

2.1.2 交换

如果一个进程想将一个虚拟页装入物理内存，而又没有可使用的空闲物理页，操作系统就必须淘汰物理内存中的其他页来为此页腾出空间。

如果从物理内存中被淘汰的页来自于一个映像或数据文件，并且还没有被写过，则该页不必被保存，它可以被丢掉。如果有进程再需要该页时就可以把它从映像或数据文件中取回内存。

然而，如果该页被修改过，操作系统必须保留该页的内容以便晚些时候再被访问。这种页称为“ 脏(dirty) ”页，当它被从内存中删除时，将被保存在一个称为交换文件的特殊文件中。

相对于处理器和物理内存的速度，访问交换文件要很长时间，操作系统必须在将页写到磁盘以及再次使用时取回内存的问题上花费心机。

如果用来决定哪一页被淘汰或交换的算法(交换算法)不够高效的话，就可能出现称为“抖动”的情况。在这种情况下，页面总是被写到磁盘又读回来，操作系统忙于此而不能进行真正的工作。例如，如果图1-2-1中物理页帧号1经常被访问，它就不是一个交换到硬盘上的好的候选页。一个进程当前正使用的页的集合叫做“工作集 (working set)”。一个有效的交换算法将确保所有进程的工作集都在物理内存中。

Linux使用“最近最少使用(Least Recently Used, LRU)”页面调度技巧来公平地选择哪个页可以从系统中删除。这种设计中系统中每个页都有一个“年龄”，年龄随页面被访问而改变。页面被访问越多它越年轻；被访问越少越年老也就越陈旧。年老的页是用于交换的最佳候选页。

2.1.3 共享虚拟内存

虚拟内存使得几个进程共享内存变得容易。所有的内存访问都是通过页表进行，并且每个进程都有自己的独立的页表。为了使两个进程共享—物理页内存，该物理页帧号必须在它们两个的页表中都出现。

图1-2-1显示了共享物理页帧号4的两个进程。对进程X来说是虚拟页帧号4，而对进程Y来说是虚拟页帧号6。这说明了共享页有趣的一点：共享物理页不必存在于共享它的进程的虚拟内存的相同位置。

2.1.4 物理寻址模式和虚拟寻址模式

操作系统本身运行在虚拟内存中没有什么意义。操作系统必须维持自己的页表是件非常痛苦的事情。大多数多用途处理器在支持虚拟寻址模式的同时支持物理寻址模式。物理寻址模式不需要页表，在这种模式下处理器不会进行任何地址转换。 Linux内核就是要运行在物理寻址模式下。

Alpha AXP处理器没有特殊的物理寻址模式。相反，它把地址空间分成几个区，指定其中两个作为物理映射地址区。这种核心地址空间被称为 KSEG地址空间，它包括所有oxfffffc0000000000以上的地址。为了执行链接在KSEG中的代码(定义为内核代码)或存取其中的数据，代码必须执行于核心模式下。Alpha上的Linux内核被链接成从地址oxfffffc0000310000开始执行。

2.1.5 访问控制

页表项中也包含访问控制信息。因为处理器要使用页表项来把进程虚拟地址映射到物理地址，它可以方便地使用访问控制信息来检查并保证进程没有以其不应该采用的方式访问内存。

有许多原因导致限制访问内存区域。有些内存，比如那些包含可执行代码的，自然地就是只读内存；操作系统不应该允许进程写数据到它的可执行代码中。相反，包含数据的则可以被写，但是试图把它当作指令来执行就会失败。大部分处理器至少有两种执行模式：用户态(用户模式)和核心态(核心模式)。我们不希望核心代码被用户执行或核心数据结构被访问，除非处理器运行在核心态下。

访问控制信息保留在PTE(页表项)中，并且是处理器相关的。图1-2-2显示了Alpha AXP 的PTE。其各字段意义如下：

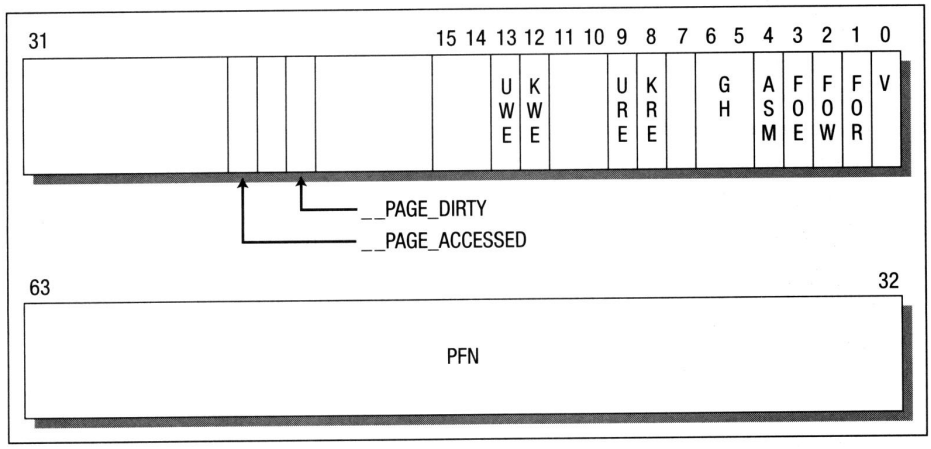


图1-2-2 Alpha AXP页表项

- V 有效性。如果置位则PTE有效。
- FOE 执行时故障，当试图执行本页中的指令时，处理器报告页故障并将控制传给操作系统。
- FOW 写时故障，当试图写本页时发出如上页故障。
- FOR 读时故障，当试图读本页时发出如上页故障。
- ASM 地址空间匹配，当操作系统想要仅清除转换缓冲区中一些项时用到。
- KRE 运行于核心态的代码可以读此页。
- URE 运行于用户态的代码可以读此页。
- GH 粒度暗示，当用一个而不是多个转换缓冲区项映射一整块时用到。
- KWE 运行于核心态的代码可以写此页。
- UWE 运行于用户态的代码可以写此页。
- PFN 页帧号。对于V位置位的PTE，本字段包括物理页帧号；对于无效PTE，如果本字段非0，则包含本页在交换文件中的位置的信息。

下面两个位在Linux中被定义并使用：

- _PAGE_DIRTY 如果置位，此页需要被写到交换文件中。
- _PAGE_ACCESSED Linux用来标识一个页已经被访问过。

2.2 高速缓存

如果你使用上面的理论模型实现一个系统，它可以工作但效率不高。操作系统设计者和处理器设计者都试图从系统中得到更高的性能。除了把处理器、内存等做得更快以外，最好的方法就是维持对有用信息和数据的高速缓存，以使一些操作更快。 Linux使用几种与内存管理有关的缓存：

- 缓冲区缓存 缓冲区缓存包含被块设备驱动程序使用的数据缓冲区。这些缓冲区是固定大小的(比如512字节)并包含从块设备中读出的或要写入块设备的信息块。块设备是只能通过读写固定大小的数据块来访问的设备。所有的硬盘都是块设备。

缓冲区缓存通过设备标识符和想要的块号来索引，并用来快速地寻找一块数据。块设备只能通过缓冲区缓存访问。如果数据能在缓冲区缓存中找到则不用从物理块设备 (如磁盘)中去读数据，从而可以很快地完成访问。

- 页缓存　用来加速对磁盘上的映像或数据的访问。它每次缓存一个文件中一页的逻辑内容并通过文件和文件内偏移来访问。所有的页都从磁盘读到内存，它们被缓存在页缓存中。
- 交换缓存　只有被修改的(脏的)页被存到交换文件中。只要这些页被写入交换文件以后没有被修改，那么下一次当该页被交换出去时就没有必要把它写到交换文件中，因为它已经在交换文件中了。该页可以简单地被淘汰掉。在交换负担严重的系统中这可以省去许多不必要的、费时的磁盘操作。
- 硬件缓存　一个普遍实现的硬件缓存，位于处理器内：页表项缓存。在这种情况下，当处理器需要时并不总是直接读取页表项，而是把页的转换信息缓存起来。这就是转换旁视缓冲器(TLB)，它们包含系统中一个或多个进程的页表项的缓存副本。

当引用虚拟空间地址时，处理器将试图找到一个匹配的 TLB项。如果能找到，它就可以直接将虚拟地址转换为物理地址，并对数据进行正确的操作。如果处理器找不到匹配的 TLB项，就必须要得到操作系统的帮助。它通过通知操作系统发生了 TLB失效来请求帮助。有一个系统相关的机制被用来将该异常传送到操作系统中完成相应操作的代码。操作系统为地址映射产生一个新的 TLB项。当异常清除后，处理器将再次尝试转换虚拟地址。这次将正常工作，因为现在TLB中有一个有效项为该地址进行转换。

使用缓存的缺点(无论是硬件的还是其他的)：为了节约开销Linux必须用更多的时间和空间来维护这些缓存，如果这些缓存崩溃，系统也将垮掉。

2.3 Linux页表

Linux假定系统中有三级页表。所访问的每级页表包含下一级页表的页帧号。图 1-2-3显示了一个虚拟地址如何被分解成几个字段，每个字段包含一个特定页表的偏移。为了将一个虚拟地址转换成一个物理地址，处理器必须取出每一个字段的内容，把它变换成包含页表的物理页的偏移并读出下一级页表的页帧号。这个过程被重复三次，直到包含该虚拟地址的物理页的页帧号被找到。然后虚拟地址的最后一个字段、字节偏移，被用来查找页中的数据。

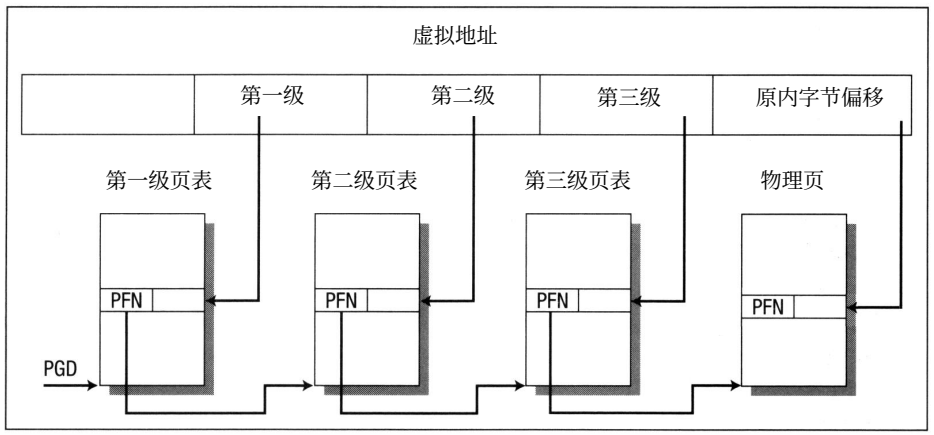


图1-2-3　三级页表

Linux运行的每一个平台都必须提供转换宏，使得内核可以遍历一个特定进程的页表。这样，内核不必知道页表项的格式以及它们如何组织。这是一种很成功的方法： Linux为Alpha处理器提供三级页表，而对Intel x86处理器，只提供两级页表，但使用相同的页表处理代码。

2.4 页分配和回收

系统中有许多对物理页的请求。例如，当一个映像被装载入内存时操作系统需要分配页。当映像结束执行并卸载时这些页又将被释放。物理页的另一个使用是保存内核专用的数据结构，比如页表本身。用来进行页分配和回收的机制和数据结构或许是维持虚拟内存子系统高效的最关键的一点。

系统中所有物理页都用 mem_map数据结构描述，它是一个在启动时被初始化的mem_map_t结构的列表。每个mem_map_t，描述系统中一个物理页。一些重要的字段有 (仅对内存管理而言)：

- count 本页使用者计数。当该页被许多进程共享时计数将大于 1。
- age 描述本页的年龄，用来判断该页是否为淘汰或交换的好的候选。
- map_nr men_map_t描述的物理页的页帧号。

Free_area向量被页分配代码用来寻找和释放页。整个缓冲区管理设计是基于这个机制的支持，并且对于代码来说，页大小和处理器所使用的物理分页机制是无关的。

Free_area的每个元素包含页块的信息。数组中第一个元素描述一页，下一个描述两页的一块，再下一个描述 4页的一块，依此以 2的幂增长。 List元素用作一个队列头并含有指向mem_map数组中page数据结构的指针，空闲的页块在这里排成队列。 map是指向一个位图的指针，该位图跟踪被分配的该大小的页组。如果第几个块空闲，则位图中第几位将被置位 N。

图1-2-4展示了free_area结构。元素0有一个空闲页(页帧号0)，而元素2有两个空闲的4页块，第一个开始于页帧号4而第二个开始于页帧号56。

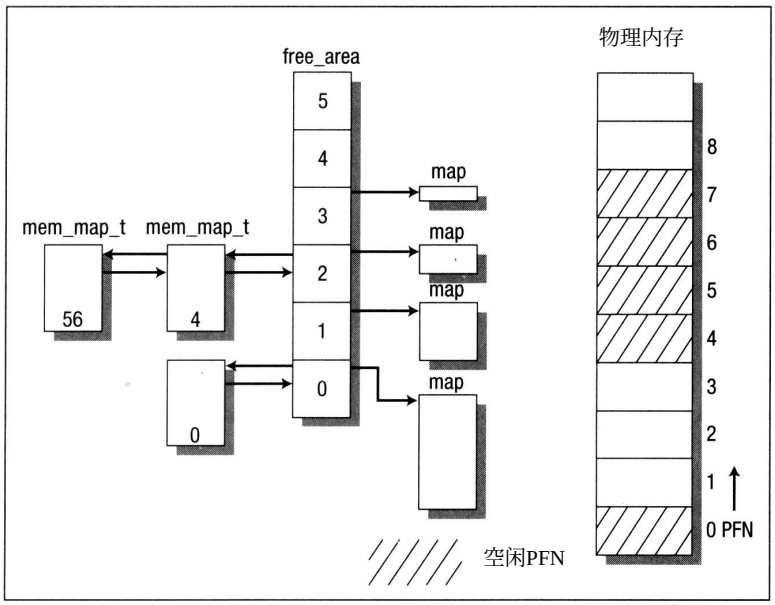


图1-2-4 free-area数据结构

2.4.1 页分配

Linux使用Buddy算法2 来高效地分配和回收页块。页分配代码试图分配一个或多个物理页的一块。页面分配是以 2的幂大小的块来进行，这意味着它可能分配 1页块、2页块、4页块等。只要系统中有足够的空间页来满足请求 (nr_free_pages > min_free_pages)，分配代码将搜索free_aree来寻找请求大小的一个页块。例如，数组中元素 2有一个内存映射来描述系统中 4页长的块的空闲和分配。

分配算法首先搜索请求大小的页块。它沿 free_area中list元素处排成队列的空闲页面链进行。如果没有所请求大小的页块是空闲的，下一个大小的块 (两倍于所请求的大小)将被查找，这个过程一直进行到 free_area整个被检查过或找到一个页块。若找到的页块比请求的大，则它必须被分开直到得到合适大小的一块。因为块大小都是 2的幂，所以分块的过程很容易，只需把块等分。空闲的块排到合适的队列中，被分配的块返回给调用者。

例如，图1-2-4中如果一个2页块被请求，第一个 4页块(开始于页帧号 4)将被分成两个2页块。第一个，也就是开始于页帧号 4的将被作为分配的页返回给调用者；而第二块，开始于页帧号6，将作为空闲的2页块排进free_area数组的元素1的队列中。

2.4.2 页回收

页块的分配可能会把内存分段，把大的空闲页块分开成小的。而页回收代码在可能的时候把页重新组合成大的空闲页块。事实上页块大小很重要，因为它使得可以容易地把块组合成更大的块。

当一个页块被释放时，将检查相邻的或称伙伴的同样大小的块是否空闲。如果是，它和新释放的页块被组合在一起形成一个新的下一个大小的空闲页块。每次两个页块被组合成更大的空闲页块时，页回收代码将试图再将该块组合成还要大的块。这样，空闲页块可以任意大，只要内存使用允许。

例如，在图1-2-4中，如果页帧号 1被释放，那它将和已经空闲的页帧号 0组合起来，并作为2页空闲块排进free_area的元素1的队列中。

2.5 内存映射

当一个映像被执行时，该可执行映像的内容必须被放到进程的虚拟地址空间。对于可执行映像链接到的共享库也是如此。可执行文件并非真正地被读到物理内存，而只是链接到进程的虚拟内存。然后，随着运行的应用对程序部分的引用，该映像被从可执行映像读到内存。这种将一个映像链到一个进程的虚拟地址空间的技术也被称为内存映射 (memery mapping)。

每个进程的虚拟内存都通过一个 mm_struct数据结构表示。它包含当前正执行的映像 (如bash)的信息以及一些指向 vm_area_struct数据结构的指针。每个 vm_area_struct数据结构描述虚拟内存区的开始和结束、进程对该内存的访问权以及对该内存的一组操作。这些操作是处理这个虚拟内存区时 Linux必须使用的一组例程。例如，当进程试图访问此虚拟内存，却发现(通过页故障)该内存并没有真正地在物理内存中时，虚拟内存操作之一将进行正确的动作。这就是nopage操作。nopage操作在Linux请求调一个可执行映像的页进内存时使用。

当一个可执行映像被映射到进程虚拟地址空间时，一组 vm_area_strnct数据结构将被产生。

每个vm_area_struct数据结构表示可执行映像的一部分；是可执行代码，或是初始化的数据 (变量)，以及未初始化数据等。 Linux支持一些标准虚拟内存操作并且当 vm_area_struct数据结构被创建时，相应的虚拟内存操作集将和它们关联起来。

2.6 请求调页

当一个可执行映像被内存映射到进程的虚拟内存时，它就可以开始执行。就在刚开始将该映像装入物理内存时，它就很快会访问一个还未在物理内存中的虚拟内存区 (图1-2-5)。当进程访问一个没有有效页表项的虚拟地址时，处理器将向 Linux报告页故障。页故障描述了发生页故障的虚拟地址以及引起页故障的内存访问类型。

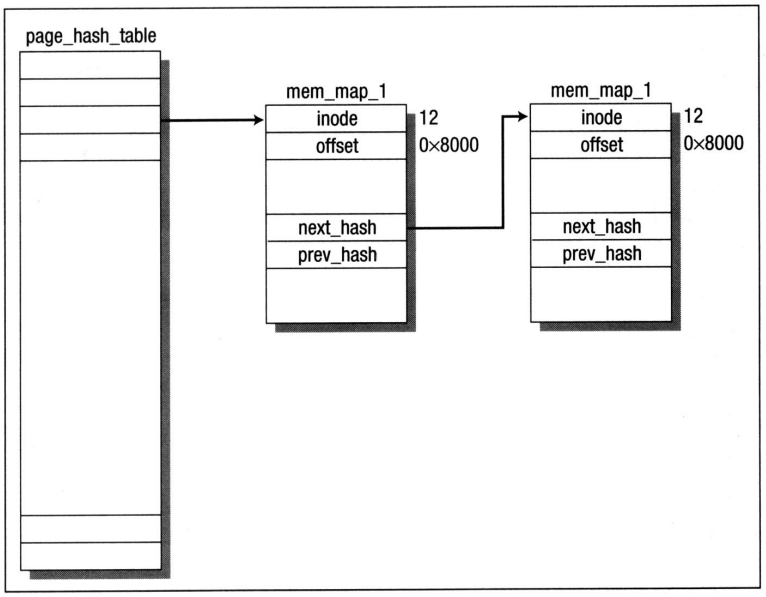


图1-2-5 虚拟内存区

Linux必须找到表示发生页故障的内存区域的 vm_area_struct数据结构。因为查找vm_area_struct数据结构是高效处理页故障的关键，它们被一起链接到一个 AVL(Addsen_Vebskii和Landis)树结构中，如果不存在发生故障的虚拟地址的vm_area_struct数据结构，那么该进程就访问了一个非法的虚拟地址。Linux将通知该进程，发送一个SIGSEGV信号，并且如果该进程没有此信号的处理器就会被终止。

Linux然后检查发生的页故障的类型，与本虚拟内存区所允许的访问类型比较。如果进程以非法方式访问内存，比如写一个只读区，它同样将会被通知发生了内存错误。

现在Linux判定页故障是合法的，则必须处理。 Linux必须区分在交换文件中的页和在磁盘上其他地方并且是可执行映像的一部分的页。它通过查看故障虚拟地址的页表项来区分。

如果该页的页表项无效但不空，则页故障是当前保存在交换文件中的一页。对于 Alpha AXP页表项来说，这些就是有效位没有置位但 PFN字段值又非0的页表项。这种情况下， PFN字段包含一些信息指明该页在交换文件中哪里 (以及哪个交换文件)被保存。交换文件中的页如何处理将在本章后面描述。

并非所有的vm_area_struct都有一个虚拟内存操作集，即使有，有的也可能没有 nopage操

作。这是因为默认(缺省)情况下Linux将通过分配一个新物理页并创建一个有效页表项来完成访问。如果一个虚拟内存区没有nopage操作，Linux将使用默认方式。

一般的Linux nopage操作用在内存映射的可执行映像上，并使用页缓存把请求的页装入物理内存。

不管怎样，所请求的页都将被装入物理内存，进程页表被更新。或许需要一些硬件相关的动作来更新这些项，特别是当处理器使用了转换旁视缓冲器时。在此页故障被处理之后，它可以被清除掉，进程在引起故障虚存访问的指令处重启动。

2.7 Linux页缓存

Linux页缓存的作用是加速对磁盘上文件的访问。内存映射的文件每次读取一页，并且这些页就保存在页缓存中。图1-2-6显示了由page_hash_table(一个指向mem_map_t数据结构的指针)组成的页缓存。Linux中每个文件由一个VFS inode数据结构标识(在第7章中描述)，并且每个VFS inode都是唯一的，完全描述一个且唯一的一个文件。页表的索引就由文件的 VFS inode和文件内的偏移导出。

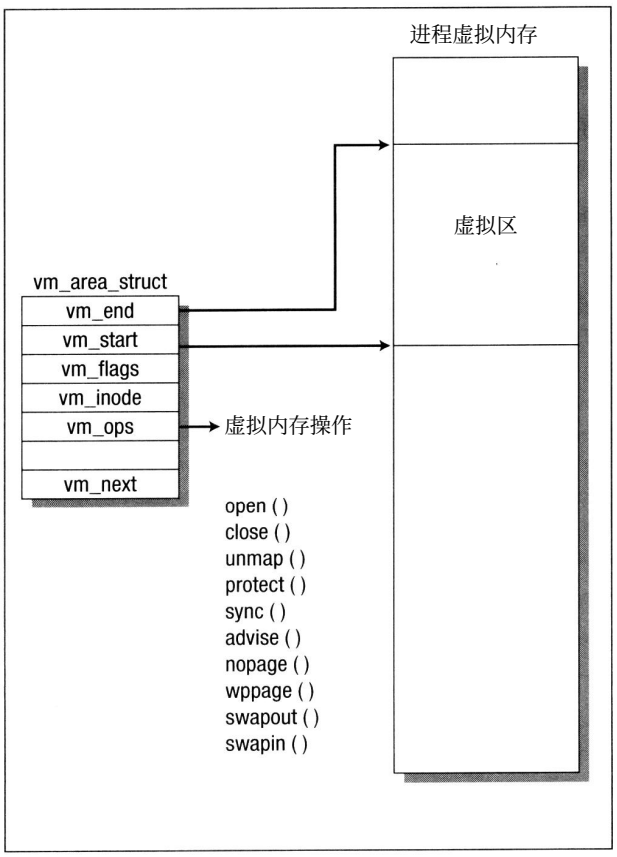


图1-2-6 Linux页缓存

每当一页要从内存映射文件读出时(比如当在请求调页时需要被读回内存)，该页通过页缓存被读出。如果该页在缓存中，一个代表该页的 mem_map_t数据结构指针被返回给页故障处

理代码，否则该页必须从保存相应映像的文件系统中被读到内存。 Linux分配一个物理页然后从磁盘文件中读取该页。

随着映像的读入和执行页缓存不断增长，当页不再被需要时会被移出缓存，比如说一个映像不再被任何进程使用时。随着 Linux不断使用内存，它可能开始缺少物理页，这时 Linux将减少页缓存的大小。

2.8 页换出和淘汰

当物理内存变得短缺时， Linux内存管理子系统必须试着释放物理页。这项任务落在内核交换守护进程(kswapd)头上。内核交换守护进程是一种特殊类型的进程 —— 一个内核线程。内核线程是没有虚拟内存的进程，它们在核态下的物理地址空间中运行。内核交换守护进程的命名有一点不准确，因为它并不仅是把页面交换到系统交换文件中。它的作用是确保系统中有足够空闲页来保持内存管理系统高效运行。

内核交换守护进程 (kswapd)在启动时由内核初始化进程启动，并等待内核交换定时器周期性超时。每当定时器超时，交换守护进程就去看系统中空闲页的数目是否太少了。它使用两个变量， free_pages_high 和free_pages_low来决定是否应释放一些页。只要系统中空闲页数目多于free_page_high，内核交换守护进程就什么也不做，它再次“睡眠”直到其定时器再次超时。在进行这次检查时，内核交换守护进程把当前正被写出到交换文件的页也计算在内。它保持一个这种页的计数于nr_async_pages中，每当有一页排队等待被写出到交换文件时，则递增；而当交换设备的写完成后递减。 Free_pages_low和free_pages_high在系统启动时被置值并与系统中物理页数有关。如果系统中的空闲页数少于 free_pages_high，甚至少于free_pages_low，则内核交换守护进程将试用三种方法来减少系统中使用的物理页数：

- 减少缓冲区和页缓存的大小。
- 交换出System V共享内存页。
- 换出及淘汰页。

如果系统中空闲页数低于 free_pages_low，内核交换守护进程将在下次运行前试着释放 6页，否则将试着释放3页。上面三种方法将依次被使用直到足够的页被释放。内核交换守护进程会记住上次它是用哪种方法来释放物理页。每当运行时它将开始试着用上次成功的方法释放页。

当释放完足够页后，交换守护进程再次睡眠直到定时器超时。如果内核交换守护进程释放页的原因是系统中空闲页数少于 free_pages_low，那么它仅睡眠通常一半的时间。一旦系统中空闲页多于free_pages_low，内核交换守护进程将在检查前睡眠长些。

2.8.1 减少缓冲区和页缓存大小

保存在页缓存和缓冲区缓存中的页是被释放到 free_area 向量中的很好的候选。页缓存 (包含内存映射文件)，可能包含着不必要的占据系统内存的页。缓冲区缓存与此类似，它包含将从物理设备中读出或写入的缓冲区，也可能包含不需要的缓冲区。当系统中物理页用完后，从这些缓存中淘汰页相对要容易些，因为它不需要写物理设备 (不像把页交换出内存)。淘汰这些页除了使对物理设备和内存映射文件的访问变慢外，没有其他太多不良作用。然而，如果从这些缓存中进行页淘汰是公平进行的，那么所有的进程将受到相同影响。

每次内核交换守护进程试图压缩这些缓存时，就检查 mem_page向量中的一块页面来看是否有页面能被淘汰出物理内存。各内核交换守护进程在强制交换时，所检查的那块页面将很大，也就是系统中的空闲页数已经降到非常低了。页面块用循环方式检查，每次试图压缩缓存时都将检查不同的一块页面。这也被称为时钟算法，因为就像时钟的分针一样，每次检查整个mem_map页向量的几页。

所检查的每一页都要看一下它是否被缓存在页缓冲区或缓冲区缓存中。需要注意，这时不考虑淘汰共享页并且一个页不能同时位于这两种缓存中。如果该页不在任何这两种缓存中则检查mem_map页向量的下一页。

页面被缓存在缓冲区缓存中 (或说缓冲区所在的页被缓存)来使缓冲区分配和回收更高效。内存映射压缩代码试着释放包含在正在检查的页面中的缓冲区。如果所有的缓冲区都被释放了，那么包含它们的页就被释放了。如果被检查的页位于 Linux页缓存中，它将被移出页缓存并释放。

当这次尝试释放了足够多的页后，内核交换守护进程将等待直到它下一次定期被唤醒。因为被释放的页不是任何进程虚拟内存的一部分 (它们是被缓存的页)，所以不需要更新任何页表。如果没有足够的缓存页被淘汰，则交换守护进程将试着换出一些共享页。

2.8.2 换出System V共享内存页

System V共享内存是一种进程间通信机制，它使得两个或多个进程可以共享虚拟内存用以在它们之间传递信息。进程如何以这种方式共享内存将在第 4章中更详细描述，现在只需要知道 每个 System V共享内存区通过一个 shmid_ds数据结构描述。它包含一个指向vm_area_struct数据结构列表的指针，每个数据结构被一个共享虚拟内存区的进程所用。Vm_area_struct数据结构描述此System V共享内存区位于每个进程的虚拟内存的什么地方。该System V共享内存区的所有vm_area_struct数据结构通过vm_next_shared和vm_prev_shared指针链接在一起。 shared_ds数据结构还包含一个页表项列表，它们中的每个描述一个虚拟共享页映射到的物理页。

内核交换守护进程在换出 System V共享内存页时同样使用时针算法。每次运行时它都记住最近换出了哪个共享虚拟内存区中的哪一页。它通过两个索引完成这种功能，一个是shmid_ds数据结构集的索引，另一个是这个 System V共享虚拟内存区中页表项列表的索引。这将保证它公平地牺牲System V共享虚拟内存区。

因为对于一个给出的System V共享内存页来说，其物理页帧号包含在所有共享了虚拟内存区的进程的页表中，内核交换守护进程必须修改所有这些页表来表明现在该页不再位于内存中而是保存在交换文件中。对于每个交换出去的共享页，内核交换守护进程找到每一个共享进程的页表中的页表项 (通过vm_area_struct数据结构中的一个指针)。如果此System V共享内存页的这个进程页表项有效，则把它变成无效且换出该页表项并把该 (共享)页的使用者计数减1。一个交换的系统V共享页表项格式包含一个 shmid_ds数据结构集的索引和一个此 System V共享内存区页表项的索引。

如果所有共享进程的页表被修改后页的使用计数为 0，则共享页就可以被写到交换文件中。该System V共享内存区的shmid_ds数据结构所指向的页表项列表将被一个换出页表项替代。一个换出页表项是无效的但包含一个打开的交换文件集的索引和一个该文件中的偏移，从偏

移处可以找到换出页，在此页需要被读回物理内存时将使用这些信息。

2.8.3 换出和淘汰页

交换守护进程依次查看系统中每个进程来看它是否是交换出的好的候选。好的候选是指那些可以被换出(有些不能)的进程，并且它有一个或多个页可以被换出或从内存中淘汰。仅当页中的内容不能从另一种方法获得时，页面才会被从物理内存交换出到系统的交换文件中。

一个可执行映像的许多内容来自于映像的文件并且可以容易地从文件中重读出来。例如，一个映像的可执行指令永远不会被映像修改，所以从来也不会被写到交换文件中。这些页可以简单地被扔掉；当再次被进程引用时，它们将从可执行映像中被读入内存。

一旦将被交换的进程被定位，交换守护进程查看它所有的共享内存区来寻找未共享并且锁住的区域。Linux并不是把所选中进程的全部可交换页换出，而是只移出少量的页。页面如果被锁住就不能被换出或淘汰。

Linux交换算法使用页年龄。每一个页有一个计数器 (保存在mem_map_t数据结构中)以提供给内核交换守护进程一些该页是否值得换出的信息。页面在不用时会变老而被访问时将重新年轻；交换守护进程只换出老的页面。一个页被分配时的缺省动作是给它一个初始年龄 3。每当它被访问时，它的年龄被加 3直到最大值20。每次内核交换守护进程运行时，要使页面变老：把它们的年龄减 1。这些缺省动作可以被改变，为此它们 (及其他有关交换的信息)被保存在swap_control数据结构中。

如果页面是老的 (age=0)，交换守护进程将进一步处理之。“ 脏 ”页是可以被换出的页。Linux使用PTE一个体系结构相关的位来用这种方式描述页 (见图1-2-2)。然而，并非所有“ 脏 ”页需要被写到交换文件中。进程的每个虚拟内存区可以有自己的交换操作 (由vm_area_struct中vm_ops指针所指向)并被使用。否则，交换守护进程将在交换文件中分配一页并把该页写出到设备上。

该页的页表项将被一个标记为无效但包含该页在交换文件中位置信息的节点所代替。这些信息是交换文件中该页被保存处的偏移和使用哪个交换文件的指示信息。无论使用什么交换方法，原先的物理页将通过放回到 free_area中而释放。干净(或说不“ 脏 ”)的页可以被扔掉并放回free_area中来重用。

如果足够的可交换进程页被换出或淘汰了，交换守护进程将再次睡眠。下次它醒来时将考虑系统中下一个进程。这样，交换守护进程一点一点地移走每个进程的物理页直到系统再次达到平衡。这比交换出整个进程要公平得多。

2.9 交换缓存

在把页面换出到交换文件时， Linux尽量避免写页。有的时候一页既在交换文件中又在物理内存中。当一个页面被换出内存并在又一次进程访问时被读进内存，则会出现这种情况。只要内存中的页没有被写，交换文件中的副本就保持有效。

Linux使用交换缓存来跟踪这些页。交换缓存是一个页表项列表，每个代表系统中一物理页。这是一个被换出的页的页表项并描述该页保存在哪个交换文件中以及在交换文件中的位置。如果一个交换缓存项非空，它代表保存在交换文件中并且未被修改的一页。若该页在以后被修改(通过写入)，该项将被从交换缓存中移去。

当Linux需要换出一物理页到交换文件中时，它将参考交换缓存，并且若有该页的一个有效项，就不需要写该页到交换文件中。这是因为内存中这一页在最近一次由交换文件中读出后还没被修改过。

交换缓存中的项是换出的页的页表项，它们被标识为无效但包含一些信息使 Linux可以找到正确的交换文件及该文件中正确的页。

2.10 页换入

存入到交换文件中的脏页可能再次被需要，例如当一个应用程序要写一段内容保存在换出的物理页的虚拟内存区时。访问一个不在物理内存中的虚拟内存页将导致页故障的发生。页故障是处理器在通知操作系统，它不能将一个虚拟地址转换到物理地址。在这里是因为当页被换出时描述该页虚拟内存的页表项被标识成无效。处理器不能处理虚拟地址到物理地址的转换，所以它把控制传回操作系统，并同时描述发生页故障的虚拟地址和故障原因。这些信息的格式以及处理器如何将控制传给操作系统是处理器相关的。处理器相关的页故障处理代码必须找到vm_area_struct数据结构，该数据结构描述发生页故障处的虚拟内存区。它通过查找进程的vm_area_struct数据结构来进行，直到找到包含故障地址的那个。这是一段对时间要求很严格的代码，并且进程的 vm_area_struct数据结构被组织成使这种查找耗费尽量少的时间。

执行完处理器相关动作并发现故障虚拟地址是位于合法的虚拟内存区中，页故障处理过程对于可运行Linux的所有处理器，就成为通用并且适用的。通用的页故障处理代码查找故障虚拟地址的页表项。如果找到的页表项从属于一个换出的页， Linux必须把该页换回物理内存。被换出的页表项格式是处理器相关的，但所有处理器都把这些页标识为无效，并把在交换文件中定位该页所需的信息放进页表项中，为了把该页换入物理内存， Linux需要使用这些信息。

这时， Linux知道了故障的虚拟地址，并有一个页表项包含该页被交换出到何处的信息。vm_area_struct数据结构可能含有一个指针指向一个例程，用于把它描述的虚拟内存区中任何页换回物理内存，这是它的swapin操作。如果有此虚拟内存区的swapin操作， Linux将使用它。事实上，它正是 System V共享内存页的处理方式：因为它需要特殊处理，来适应换出的 System V共享页的格式与普通换出页格式的差异。也可能没有 swapin操作，这时Linux将认为它是一个普通页，不需要特殊处理。它将分配一个空闲物理页并从交换文件中读回被换出的页。该页在交换文件中的位置信息(及哪个交换文件)将从相应无效页表项中获得。

如果引起页故障的访问不是写访问，则该页被留在交换缓存中，并且其页表项不被标识成可写。如果该页以后被写，将发生另一个页故障，并且那时该页将被标识为“脏”，其页表项从交换缓存中被移去。若该页没被写并且它需要再次被换出时， Linux可以免去把该页写到交换文件，因为它已经在交换文件中了。

如果引起把页从交换文件读回的访问是写操作，则此页被从交换缓存中移出，并且其页表项被标识为“脏”和“可写”。

第3章 进 程

本章描述了什么是进程以及Linux内核如何创建、管理和删除系统中的进程。

进程执行操作系统中的任务。一个程序是保存在一个磁盘的可执行映像中的机器代码指令和数据的集合，并且，实际上是一个被动实体；一个进程可以被认为是一个执行中的计算机程序。它是一个动态实体，总是随着机器代码指令随处理器的执行而处于变化之中。除了程序的指令和数据，进程还包括程序计数器和所有 CPU寄存器，以及含有例程参数、返回地址和保存的变量等临时变量的进程栈 (stack)。当前正执行的程序或说进程，包含所有处理器当前的行为。Linux是一个多进程操作系统。进程是独立的任务，有自己的权利和责任。若一个进程崩溃并不会引起系统中另一个进程崩溃。每个单独的进程运行在自己的虚拟地址空间，并且只能通过安全的内核管理机制和其他进程交互。

在进程的生存期(lifetime)内将使用许多系统资源。它将使用系统的 CPU来运行自己的指令并使用系统的物理内存来保存自己和自己的数据；它将打开和使用文件子系统中的文件并直接或间接地使用系统中的物理设备。 Linux必须跟踪进程本身和它拥有的系统资源，来保证它能公平地管理该进程和系统中其他进程。如果一个进程独自占有系统中的大部分物理内存或其CPU，对系统中其他进程来说将是不公平的。

系统中最宝贵的资源是CPU，通常只有一个。Linux是个多进程操作系统，它的目标是在每一时刻都有一个进程运行在系统的每个 CPU上，来极大化CPU利用率。若进程比CPU多(通常是这样)，其余的进程在它们可以运行前必须等待一个 CPU变空闲。多进程基于这样一个简单的思想：一个进程执行直到它必须等待，通常是等待一些系统资源；当它获得这个资源后，就可以再次运行。在单进程系统中，比如 DOS，CPU将简单地闲置等待，等待的时间将被浪费。在多进程系统中内存里同时保存多个进程。每当一个进程需要等待时，操作系统从该进程夺走CPU并给予另一个更有价值的进程。选择哪一个进程是下一次运行最合适的进程是调度器(scheduler)的事，Linux使用一些调度策略来保证公平。

Linux支持几种不同的可执行文件格式， ELF是一种，Java是另一种；并且它们要透明地管理，因为进程要使用系统的共享库。

3.1 Linux进程

为了使Linux可以管理系统中的进程，每个进程通过一个 task_struct数据结构表示 (任务(task)和进程(process)是Linux中两个互相通用的术语)。Task向量是一个指向系统中所有task_struct数据结构的指针数组。这意味着系统中最大进程受限于 Task向量的大小，缺省情况下它有512项。当进程被创建时，从系统内存中分配一个新的 task_struct并把它加入到task向量中。为了便于查找，当前运行的进程由 current指针来指向。

除了普通类型的进程， Linux还支持实时进程。这些进程必须对外部事件反应迅速 (从而得到名字“ 实时 ”)，并且它们被调度器区别于普通用户进程对待。尽管 task_struct数据结构很大很复杂，但它的字段可以被分成几个功能区。

• 状态　随着进程执行，它根据其环境改变状态。Linux进程有以下状态：

a. 运行　进程或者正在运行(它是系统中的当前进程)或者已经准备好运行(等待被分配到系统的一个CPU上)。

b. 等待　进程正在等待一个事件或一个资源。Linux区分两种类型的进程；可中断的和不可中断的。可中断等待进程可以被信号中断，而不可中断等待进程直接等待硬件条件，并且在任何环境下都不会被中断。

c. 停止　进程停止了，通常是通过接收一个信号。一个正在被调试的进程可以处于停止状态。

d. 死亡　这是一个被终止的进程，由于某种原因，仍在 task向量中有一个task_struct数据结构，正像其名字那样，它是一个死去的进程。

• 调度信息　调度器需要这些信息以公平地决定系统中哪个进程最值得运行。

• 标识　系统中每个进程有一个进程标识。进程标识不是 task向量的索引，它只是一个简单的数。每个进程还有用户和组标识，这些被用来控制本进程对系统中文件和设备的访问。

• 进程间通信　Linux支持经典的UNIXTM IPC机制如信号、管道和信号灯，以及 System V IPC机制如共享内存、信号灯和消息队列。Linux支持的IPC机制在第4章描述。

• 链接　在Linux系统中没有哪个进程与其他进程完全独立。除了初始进程，系统中每个进程都有一个父进程，新的进程不是被创建，它们是从先前的进程被复制 (或说克隆)。每一个表示一个进程的task_struct都有指针指向其父进程及其兄弟进程 (和它具有相同父进程的进程)以及它自己的子进程。你可以用 ptree命令查看Linux系统中运行的进程之间的家族关系：

```
init(1)-+-crond(98)
        |-emacs(387)
        |-gpm(146)
        |-inetd(110)
        |-kerneld(18)
        |-kflushd(2)
        |-klogd(87)
        |-kswapd(3)
        |-login(160)—bash(192)—emacs(225)
        |-lpd(121)
        |-mingetty(161)
        |-mingetty(162)
        |-mingetty(163)
        |-mingetty(164)
        |-login(403)—bash(404)—pstree(594)
        |-sendmail(134)
        |-syslogd(78)
        `-update(166)
```

另外系统中所有的进程都保存在一个双向链表中，它的根是 init进程的task_struct数据结构。这个链表使得 Linux可以查看系统中每一个进程，它需要这样来为一些命令如 ps或kill提供支持。

• 时间和时钟　内核记住进程的创建时间以及在它生存期内消耗的 CPU时间。每次时钟

“滴答”时，内核更新保留在jiffies中的时间量，表示当前进程花费在系统和用户态下的时间。Linux还支持进程相关的间隔定时器。进程可以用系统调用来设置定时器，以便当定时器超时时给进程自己发送信号。定时器可以是一次触发的或周期性多发的。

- 文件系统　进程在需要的时候可以打开和关闭文件；进程的task_struct包含每个打开的文件的描述符指针以及两个VFS索引节点的指针。每个VFS索引节点唯一地描述文件系统中的一个文件或目录，并提供一个底层文件系统的统一接口。 Linux对文件系统的支持将在第7章描述。两个VFS索引节点指针第一个指向进程的根目录，第二个指向其当前的或称pwd目录。pwd从UNIX命令pwd——打印工作目录(print working directory)而来。这两个VFS索引节点使自己的count字段(field)递增来表示一个或多个进程在引用它们。这就是为什么不能删除被一个进程设成 pwd的目录的原因。同样的原因，也不能删除它的子目录。
- 虚拟内存　大多数进程有一些虚拟内存(内核线程和守护进程没有)，并且Linux必须跟踪虚拟内存如何映射到系统物理内存。
- 处理器相关上下文　一个进程可以被认为是系统当前状态的总和。每当一个进程运行时，它要使用处理器的寄存器、栈等等，这是进程的上下文 (context)。并且，每当一个进程被暂停时，所有的CPU相关上下文必须被保存在该进程的 task_struct中。当进程被调度器重新启动时其上下文将从这里恢复。

3.2 标识符

Linux像所有UNIXTM系统一样使用用户和组标识来检查对系统中文件和映像的访问权限。Linux系统中所有文件都有所有权和授权，这些授权描述系统中用户对该文件或目录有什么访问权限。基本的授权是读、写和执行并被赋予三种用户：文件的所有者、属于一个特定组的进程和系统中所有的进程。每种用户可以有不同的授权，例如一个文件可以有授权使得其所有者可以读写，文件的组成员可以读而系统中其他进程没有任何访问权。

组是Linux给一组用户(而不是一个用户或系统中所有进程)赋予访问文件或目录特权的方式。例如，你可以为一个软件项目的所有用户创建一个组，并可以设定只有他们能够读写该项目的源代码。一个进程可以属于几个组 (缺省最大值是32个)，并且它们被保存在每个进程task_struct中的groups向量中。只要进程所在的一个组有对某文件的访问权限，该进程就拥有对该文件的适当的组权利。

一个进程task_struct中有4对进程和组标识符：

- uid、gid　进程运行所代表用户的用户标识符和组标识符。
- 有效uid、gid　有些程序把从执行进程来的uid和gid改变成自己的(作为属性保存在描述可执行映像的VFS索引节点中)。这些程序被称为setuid程序，并且它们很有用，因为它提供一种限制访问某些服务的方式，特别是那些代表其他用户运行的进程，比如一个网络守护进程。有效uid和gid是来自setuid程序的而其uid和gid保持不变。当内核检查特权时就检查有效uid和gid。
- 文件系统uid和gid　这些通常和有效uid、gid相同，并在检查文件系统访问权利的时候被使用。它们在NFS安装的文件系统中需要，在那里用户模式下的 NFS服务器需要像一个特定进程一样访问文件。这种情况下只有文件系统uid和gid被改变(而不是有效uid和gid),

这样可以防止恶意用户向NFS服务器发送kill信号。Kill信号将被送到一个有特定有效uid和gid的进程。

- 保存的uid和gid　这是POSIX标准所要求的并被那些通过系统调用改变进程 uid和gid的进程使用。它们用来在初始uid和gid被改变期间保存真正的uid和gid。

3.3 调度

所有的进程都是部分运行在用户模式下，部分运行在系统模式下。这些模式怎样被底层硬件支持是随硬件不同而不同的，但通常都有一个安全的机制来从用户模式进入系统模式以及再返回。用户模式拥有的特权比系统模式少得多。每次进程进行一次系统调用时，它从用户模式切换到系统模式然后继续执行，这时内核代表进程执行。在 Liunx 中，进程不能抢先当前的、正在运行的进程，它们不能停止其运行以便使它们自己能运行。每个进程当必须等待某个系统事件时，会释放它正在其上运行的 CPU。例如，一个进程可能需要等待从一个文件中读入一个字符，这种等待发生在系统调用中。在系统模式下，进程使用一个库函数来打开和读取文件，接着进行系统调用来从打开的文件中读取字节。在这种情况下，等待的进程将被暂停而另一个更有价值的进程将被选中运行。

进程总是在进行系统调用，所以经常需要等待。即使如此，若一个进程执行直到它等待才让出CPU的话，就仍可能使用了不合适的 CPU时间，所以Linux使用抢先调度。在这种方案中，每个进程被允许运行少量的时间 (200ms)当这段时间用完后另一个进程被选中来运行，而原先的进程要等待一会直到它可以再次运行。这段运行的少量时间被称为时间片 (time-slice)。

调度器必须从系统中可运行的进程中选取出最值得运行的进程。可运行的进程是指只等待CPU来运行的进程。Linux使用一个合理的基于简单优先权的调度算法来从系统中现有的进程中选择。当选中一个新进程来运行，它将保存当前进程的状态，处理器相关寄存器和其他上下文被保存在进程 task_struct数据结构中。然后它恢复新进程的状态 (同样这是处理器相关的)来运行并把系统控制交给该进程。调度器为了在系统中可运行进程间公平地分配 CPU时间，为每个进程在task_struct中保存有如下信息：

- policy　将被应用于本进程的调度策略。有两种 Linux进程：普通的和实时的。实时进程拥有比其他进程都要高的优先级。如果有一个实时进程准备好了运行，它总是先运行。实时进程可以有两种高度策略：“ 轮转(round robin) ”法和 “ 先进先出(first in first out) ”。在轮转法调度中每个可运行的实时进程依次运行；而在先进先出法中，实时进程按它们进入运行队列的顺序依次运行，并且该顺序永不会改变。
- priority　调度器将给予进程的优先级。它也是进程被允许运行时可以运行的时间量 (在jiffies中)。可以通过系统调用的方法和 renice命令来改变进程的优先级。
- rt_priority　Linux支持实时进程，并且它们在调度时拥有比系统中其他非实时进程更高的优先级。这个字段使调度器可以给每个实时进程一个相对优先级。实时进程的优先级可以用系统调用改变。
- counter　此进程允许运行的时间量(在jitties中)在第一次运行时被置成priority的值，然后在每个时钟 “ 滴答 ” 中递减。

调度器可以从内核中几个地方运行。它可以在把当前进程放到等待队列中后运行，也可以在一个系统调用结束并且刚好一个进程从系统模式返回进程模式时运行。另一个需要运行

的原因是系统定时器刚好把当前进程的counter置为0，每次调度器运行时它作以下工作：

- 内核工作　调度器运行Bottom Half控制程序，并处理调度器任务队列。这些轻量内核线程在第9章中详细描述。
- 当前进程　在另一个进程被选择运行前必须处理当前进程，如果当前进程的调度策略是轮转法，则它被放到运行队列尾；如果任务是可中断的，并且从最近一次被调度后收到了一个信号，则它的状态变为RUNNING。

如果当前进程时间用完了，那么其状态变为RUNNING。

如果当前进程是RUNNING，则它保持该状态。

既不是RUNNING也不是可中断的进程将从运行队列中移出。这意味着当调度器寻找最有价值的进程来运行时，它们不会被考虑。

- 进程选择　调度器查看运行队列中的所有进程来寻找最有价值的来运行。如果有实时进程(有实时调度策略的进程)那么它们将得到比普通进程更高的优先。也就是如果系统中有可运行的实时进程的话，在别的普通进程运行之前总是先运行这些实时进程。当前进程，已经消耗了一部分自己的时间片(其counter被减去了)，如果系统中有其他相同优先级进程的话将处于不利地位；事实上也应该是这样。如果有几个具有相同优先级的进程，则最靠近运行队列头的被选中。当前进程将被放到运行队列的尾部。在一个平衡的有许多相同优先级进程的系统中，每个进程将依次运行，这就是所谓的轮转法调度。然而，因为进程会等待资源，它们的运行顺序会改变。
- 切换进程　如果最有价值运行的进程不是当前进程，那么当前进程必须被暂停并使新进程运行。当进程运行时它正使用CPU和系统的寄存器和物理内存。每次它调用一个例程为它在寄存器中传递参数，并且可能把保存的值压到栈上，比如返回到调用例程的地址。所以，当调度器运行时它是在当前进程的上下文中运行。它将处于一个特权模式即核心模式，但运行的仍是当前进程。当该进程被暂停时，其所有的机器状态，包括程序计数器(PC)和处理器的所有寄存器，必须被保存在进程的task_struct数据结构中。然后，新进程的所有机器状态必须被装入。这是一个系统相关的操作，没有CPU用相同的方法完成此任务，但是通常有一些完成该动作的硬件帮助。

进程的切换发生在调度的最后。因此，被保存的原先进程的上下文是系统硬件上下文的一个快照(snapshot)，因为在调度最后还是在这个进程中运行。同样，当新进程上下文被装入时，它也是调度末尾情况的一个快照，包括此进程的程序计数器和寄存器内容。

如果先前进程或新进程使用虚拟内存，则系统的页表可能需要更新。这个动作又是体系结构相关的。像Alpha AXP这样使用转换旁视表或缓存的页表项的处理器，必须清空这些缓存的属于先前进程的表项。

多处理器系统中的调度

在Linux世界中有多个CPU的系统很少，但却已经做了许多工作来使　Linux成为一个SMP(Symmetric Multi-Processing,对称多处理器)操作系统。也就是一个可以在系统中CPU之间平等地均衡负载的操作系统。这项均衡工作并不比调度器中的明显。

在一个多处理器系统中，希望所有处理器都忙于运行进程。每个处理器在其当前进程用尽它的时间片后或需要等待系统资源时将独立地运行调度器。　SMP系统中要注意的第一件事

是系统中不只一个空闲进程。在单处理器系统中空闲进程是 task向量中第一个任务；在SMP系统中每个CPU有一个空闲进程而你可以有多于一个的空闲 CPU。另外，每个CPU有一个当前进程，所以SMP系统必须为每个处理器跟踪当前和空闲进程。

在SMP系统中每个进程的task_struct包含进程正在运行的处理器号 (processor)和它上次运行的处理器号 (tast_processor)。一个进程每次被选中后可在不同的 CPU上运行，但可以用processor_mask来把一个进程限制在一个或多个处理器上。如果位 N被置位，则此进程可以在处理器N上运行。当调度器选择一个新的进程运行时 ,它不会考虑那些没有在其processor_mask中对应于当前处理器的位置位的进程。调度器同样给予上次运行在本处理器上的进程轻微的优先，因为当移动一个进程到不同的处理器上时经常会有性能开销。

3.4 文件

图1-3-1显示了系统中每个进程有两个数据结构描述文件系统相关信息。第一个 —— fs_struct，包含指向此进程VFS索引节点的指针和umask。umask是新文件被创建的缺省模式，它可以通过系统调用来改变。

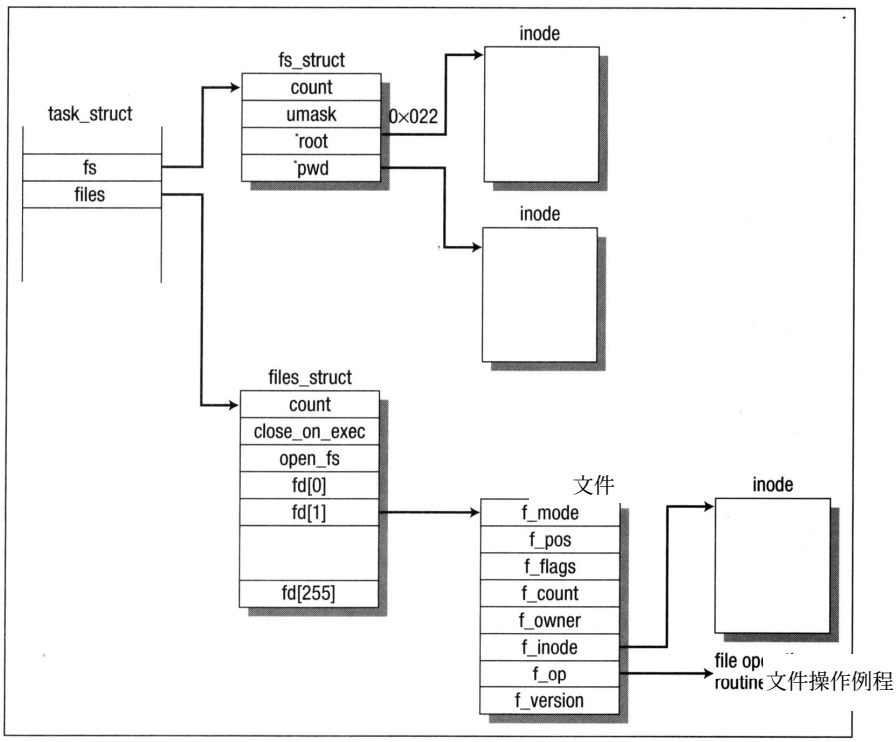


图1-3-1 进程的文件

第二个数据结构 —— Files_struct，包含此进程正在使用的所有文件的信息。程序从标准输入设备(standard input)读入而写出到标准输出设备(standard output)。任何错误消息应该输出到标准错误设备(standard error)。这些设备可以是文件、终端输入 /输出或一个真实的设备，但就程序而言它们都被当作文件对待。每个文件有自己的描述符并且 files_struct中包含至多256

个file数据结构的指针，每个数据结构描述一个被此进程使用的文件。 f_mode字段描述该文件是以什么模式创建的：只读、读写还是只写。 f_pos保存文件中下一个读或写将发生的位置。f_inode描述文件的VFS索引节点，而f_ops是一个例程地址向量的指针，每个代表一个想施加于文件的操作的函数。例如，有一个写数据函数。这个接口的抽象非常强有力并允许 Linux支持广泛的文件类型。后面我们会看到Linux中管道就是用这种机制实现的。

每次一个文件被打开时， files_struct中的空闲file指针之一就被用来指向新的 file结构。Linux进程期望在启动时有三个文件描述符被打开了，它们就是标准输入设备、标准输出设备和标准错误设备，并且通常它们是从创建此进程的父进程继承得来的。所有对文件的访问是通过传递或返回文件描述符的标准系统调用进行的。这些描述符是进程 fd向量的索引，所以标准输入设备、标准输出设备和标准错误设备分别对应文件描述符 0、1和2。每次对文件访问都是使用file数据结构的文件操作例程以及VFS索引节点来达到需求。

3.5 虚拟内存

一个进程的虚拟内存包含来自多处的可执行代码和数据。首先是被装入的程序映像，比如一个命令如fs。与其他所有可执行映像一样，这个命令由可执行代码和数据组成。映像文件包含把可执行代码和相关联的程序数据装入进程虚拟内存中所需要的所有信息。其次进程在其处理过程中可以分配(虚拟)内存来使用，比如保存它读取的文件的内容。这新分配的虚拟内存需要被链接到进程已有的虚拟内存以便使用。第三， Linux进程使用公共用途代码库，比如文件处理例程。每个进程有自己的库副本是没有意义的， Linux使用可以被几个运行的进程同时使用的共享库。这些共享库中的代码和数据必须被链接到此进程的虚拟地址空间，以及其他共享这些库的进程的虚拟地址空间。

在一个给定的时间段中一个进程不会使用包含在其虚拟内存中的全部代码和数据，它可能仅使用在特定情况下使用的代码(如初始化时或处理特定事件时)，也可能只使用共享库中一部分例程。将全部这些代码和数据装入物理内存将是浪费：它们不可能被同时使用。随着系统中进程数的增多，这种浪费被成倍地扩大，系统将非常低效地运行。事实上， Linux使用一种称为请求调页(demand-Paging)的技术：只有当进程要使用时其虚拟内存才被装入到物理内存。所以，不是直接把代码和数据装入物理内存， Linux内核只修改进程的页表，标识出虚拟内存页存在但不在内存中。当进程想要访问代码或数据时，系统硬件将产生页故障并把控制交给Linux内核来解决。因此，对于进程地址空间中的每一个内存区， Linux都需要知道该虚拟内存来自何处，以及如何把它装入内存以解决页故障。

Linux内核需要管理所有这些虚拟内存区，并且每个进程虚拟内存的内容通过其task_struct中指向的一个mm_struct数据结构来描述。进程的mm_struct数据结构还包含装入的可执行映像的信息和一个指向进程页表的指针。它包含指针指向一个 vm_area_struct数据结构列表，每个vm_area_struct代表此进程中的一个虚拟内存区。

这个链表是按虚拟内存中上升顺序排列，图 1-3-2显示了一个简单进程的虚拟内存布局，以及管理它的内核数据结构。因为这些虚拟内存区来自多处， Linux 使vm_area_struct指向一个虚拟内存处理例程的集合(通过vm_ops)来抽象接口。这样进程所有的虚拟内存可以用统一的方法处理，而不管内存管理的底层服务如何变化。例如当进程试图访问不存在的虚拟内存时将有一个例程被调用，这就是页故障的处理方法。

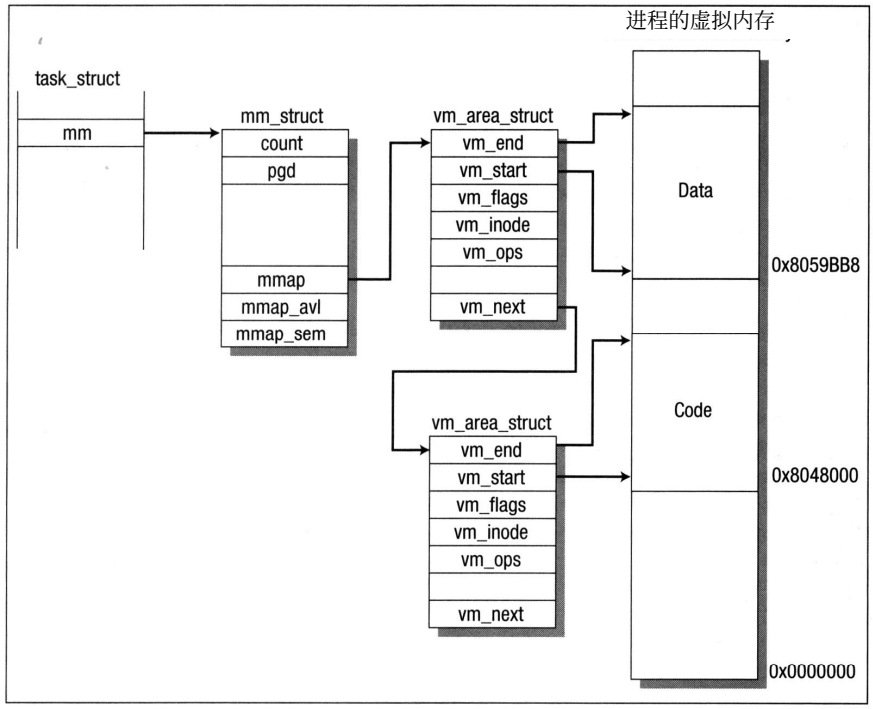


图1-3-2 进程的虚拟内存

随着Linux为进程创建新的虚拟内存区以及解决对不在系统物理内存中的虚拟内存引用的问题，进程的vm_area_struct数据结构集合将不断被 Linux内核访问。这使得查找到正确的vm_area_struct所花费的时间对系统性能很重要。为了加快这种访问， Linux同样把vm_area_struct数据结构组织成一个 AVL(Adelson-Velskii和Landis)树。这种树被组织成每个vm_area_struct(或节点)有一个左指针和一个右指针指向其邻接 vm_area_struct结构。左指针指向一个具有比自己低的起始虚拟地址的节点，而右指针指向一个具有比自己高的起始虚拟地址的节点。为了找到正确的节点， Linux从树的根节点开始沿每个节点的左、右指针进行直到它找到正确的 vm_area_struct。当然，世界上没有免费的午餐，在这种树中插入一个新的vm_area_struct将花费额外的处理时间。

当一个进程分配虚拟内存时， Linux并不真正为它保留物理内存。它只是创建一个新vm_area_struct数据结构来描述虚拟内存，这个结构被链入进程的虚拟内存列表。当进程试图写一个位于新分配虚拟内存区域的虚拟地址时，系统将产生页故障。处理器试图转换该虚拟地址，但是因为没有此内存的页表项，它将放弃并产生一个页故障异常，留给 Linux内核来解决。Linux查看被引用的虚拟地址是否位于当前进程的虚拟内址空间。如果是 Linux创建适当的PTE并为此进程分配一页物理内存。代码或数据可能需要从文件系统或交换硬盘上读入物理内存。然后进程可以从引起页故障的那条指令处重启，并且因为这次内存物理地址存在，所以它可以继续执行。

3.6 创建进程

当系统启动时它在核心模式下运行，并且在某种意义上，只有一个进程 —— 初始进程。像

所有进程一样，初始进程有一个机器状态，用栈、寄存器等表示。当系统中其他进程被创建执行时，这些将被保存在初始进程的task_struct数据结构中。在系统初始化的结尾，初始进程启动一个内核线程(称为init)，然后就空闲等待，什么也不做。每当没有其他事情时调度器就运行这个空闲的进程。空闲进程的task_struct是唯一非动态分配的，它是在内核建造时静态定义，并且很奇怪地被称为init_task。

Init内核线程或进程具有进程标识符1，因为它是系统中第一个真正的进程。它完成一些系统的初始配置(如打开系统的控制盘以及安装根文件系统)，然后执行系统初始化程序——/etc/init、/bin/init或/sbin/init之一，至于使用哪一个则由系统而定。 Init程序使用/etc/inittab作为一个脚本文件来创建系统的新进程，这些新进程自己可以继续创建新进程。例如 getty进程在用户试图登陆时，可以创建一个login进程。系统中所有进程都是init内核线程的后代。

新进程是通过克隆旧进程或说克隆当前进程而创建。新任务通过一个系统调用 (fork或clone)而创建，克隆过程发生在核心模式下的内核中，在系统调用的末尾会有一个新进程等待调度器选中它并运行。一个新的task_struct数据结构被从系统物理内存分配，以及一页或多页物理内存作为克隆的进程的栈(用户的和核心的)。一个进程标识符可能会被创建：一个在系统进程标识符集合中唯一的标识符。然而，克隆的进程完全有理由保存其父进程的标识符。新的task_struct新加入到task向量并且老的(当前)进程的task_struct的内容被复制到克隆出的task_struct中。

当克隆进程时，Linux允许两个进程共享资源而不是有两份独立的副本。这种共享可用于进程的文件、信号处理器和虚拟内存。当资源被共享时，它们的 count字段将被递增，以便在两个进程都结束访问它们之前Linux 不会回收这些资源。举例来说，如果克隆的进程将共享虚拟内存，其task_struct将包含指向原先进程的 mm_struct的指针并且该mm_struct将把其count字段递增以表明当前共享该页的进程数。

克隆一个进程的虚拟内存是很有技巧性的。一个新的 vm_area_struct集合以及它们拥有的mm_struct数据结构，还有被克隆的进程的页表必须被产生出来。在这时没有进程的任何虚拟内存被复制。复制将是一件很困难和冗长的工作，因为一些虚拟内存在物理内存，一些在进程正在执行的可执行映像中，还可能有一些在交换文件中；相反， Linux使用称为“写时复制(copy on write)”的技巧，这意味着仅当两个进程试图写它时虚拟内存才会被复制。任何虚拟内存只要没被写，即使它可以被写，也将在两个进程间共享而不会引起任何危害。只读的内存(比如可执行代码)总是被共享。为了使“写时复制”工作，可写区域将其页表项标识为只读并且vm_area_struct数据结构描述为它们被标识成”写时复制”。当一个进程试图写该虚拟内存时将产生一个页故障。正是在此时 Linux将产生该内存的一个副本，并修改两个进程的页表和虚拟内存数据结构。

3.7 时间和定时器

内核保持跟踪一个进程的创建时间以及它在生存期中消费的 CPU时间。每个时钟“滴答”一下，内核就更新jiffies中当前进程花费在系统模式和用户模式下的时间量。

除了这些计数定时器外，进程支持进程相关的间隔定时器。进程可以使用这些定时期，在每次它们超时时给自己发送各种各样的信号。 Linux支持三种类型的间隔定时器：

- 真实的 这种定时器按真实时间跳动，当它超时时发送给进程一个 SIGALRM信号。

- 虚拟的 这种定时器仅在进程运行时才跳动，当它超时时发送给进程一个 SIGVTALRM 信号。
- 描述的 这种定时器在进程运行时和系统代表进程执行时都将跳动，当它超时时将发送 SIGPROF信号。

一种或全部间隔定时器都可以运行， Linux将所有必需信息保存在进程的 task_struct数据结构中。可以进行系统调用来设置这些间隔定时器，以及启动、停止它们或读取其当前的值。虚拟和描述定时器以相同方式处理。每个时钟“ 滴答 ”，当前进程的间隔定时器被递减并且如果它们超时，就发送相应的信号。

真实定时器有些不同， Linux将使用第9章中描述的定时机制。每个进程有自己的 time_list 数据结构，并且当真实间隔定时器运行时，它被排队到系统的定时器列表。当定时器超时时，定时器底层部分处理器把它移出队列并调用间隔定时器处理器。这将产生 SIGALRM信号并重启间隔定时，把它加入到系统定时器队列尾部。

3.8 执行程序

在Linux中，像UNIX中一样，程序和命令通常由命令解释器执行。命令解释器是一个像其他进程一样的用户进程，它被称为shell。Linux中有许多shell，一些最流行的如sh、bash和tcsh。除了一些内建的命令如cd和pwd之外，一个命令就是一个可执行二进制文件。对于每个输入的命令，shell在保存于PATH环境变量中的进程的搜索路径目录中寻找一个名字匹配的可执行映像。如果文件找到了，它就被装入并执行。shell使用上面描述的fork机制克隆自身，然后新的子进程用刚找到的可执行映像文件的内容替换它正执行的 shell二进制映像。通常 shell等待命令结束，或说等待子进程退出。你可以通过把子进程推到后面来使 shell再次运动。敲入comtrol_z，将使一个SIGSTOP信号发送到子进程，使它停止。然后使用 shell命令bg把它推到后面， shell发送一个SIGCONT信号重新启动它，它将留在后面直到结束或需要做终端输入或输出。

可执行文件可以有许多种格式甚至是一个脚本文件。脚本文件必须被识别出来并运行适当的解释器来处理，例如， /bin/sh解释shell脚本。可执行目标文件包含可执行代码和数据，以及足够的信息使操作系统可以把它们装入内存并执行。 Linux中最常用的目标文件格式是ELF，但是，理论上Linux足够灵活以处理几乎任何目标文件格式。

像对文件系统一样， Linux支持的二进制格式或是在内核建造时内置进内核的，或者可以作为模块被装入。内核保持一个所支持的二进制格式列表 (见图1-3-3)，当试图执行一个文件时，依次试用每一种二进制格式直到有一种成功。通常Linux支持的二进制格式是a_out和ELF。可执行文件不必完全被装入内存，而是使用了一种称为请求调页的技术。随着每一部分可执行映像被进程使用，它被读入内存。未被使用的部分映像可以从内存中淘汰。

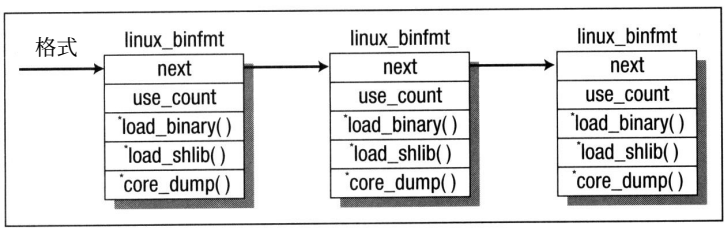


图1-3-3 注册的二进制格式

3.8.1 ELF

由Unix系统实验室设计的ELF(Executable and Linkable Format，可执行和可链接格式)目标文件格式已牢固树立了Linux中最常使用的格式的地位。尽管和其他目标文件格式如 ECOFF和a.out相比有轻微性能开销，但ELF被认为更灵活。ELF可执行文件包含可执行代码，有时被称为正文(text)，以及数据(data)。可执行映像中的表格描述程序应该怎样被放到进程的虚拟内存中。静态链接映像被链接器(ld)或链接编辑器建造成一个单一的映像，包含运行此映像所需的所有代码和数据。映像还指明此映像在内存中的布局，以及此映像中第一条执行的代码的地址。

图1-3-4显示了一个静态链接的可执行映像的布局。它是一个简单的打印“ hello world ”，然后就退出的程序。

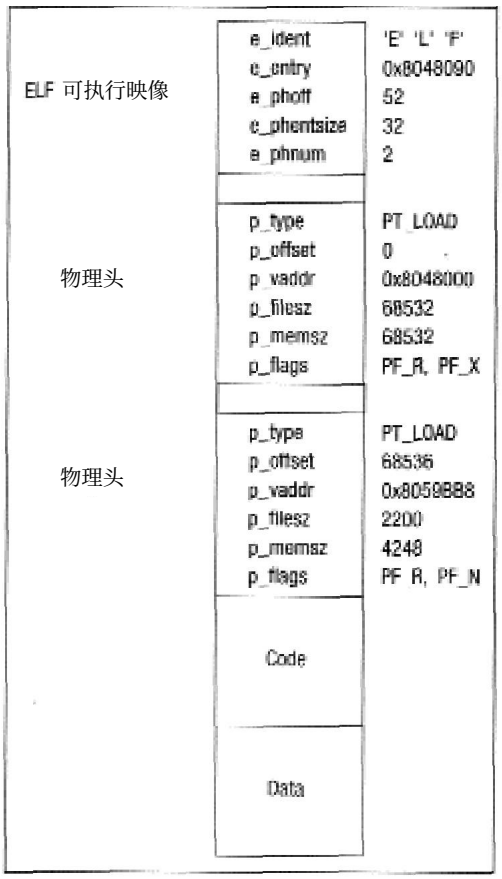


图1-3-4 ELF可执行文件格式

头信息说明它是一个 ELF映像，在文件开始 52字节处(e_phoff)有两个物理头(e_phnum为2)。第一个物理头描述映像中的可执行代码。它从虚拟地址0x8048000处开始，共有 65532字节。这是因为它是一个静态链接和映像，包含了输出“ hello world ”的printf( )调用的全部库代码。这个映像的入口点，即程序的第一条指令，不是映像的开始处而是在虚拟地址 0x8048090(e_entry)处，代码紧接第二个物理头开始。这个物理头描述了程序的数据并将装入到虚拟内存中地址0x8059BB8处。这段数据既是可读又是可写的。读者将注意到该数据在文件中的大小是2200字节(p_filesz)，而它在内存中的大小是4248字节。这是因为前面的 2200字节包含预初始化的数据，而后面 2048字节包含的数据将被执行的代码初始化。

当Linux把一个ELF可执行映像装入进程的虚拟地址空间时，它并不实际地装入映像。它设置好虚拟内存数据结构，进程的 vm_area_struct树及其页表。当程序被执行时，页故障将引起程序的代码和数据被取到物理内存中，程序中未被使用的部分将永远不会被装入内存。一旦ELF二进制格式装入器发现该映像是一个有效的ELF可执行映像，它就把进程的当前可执行映像从其虚拟内存中冲掉。因为这个进程是被克隆的映像(所有的进程都是)，这个老的映像是其父进程正在执行的程序，比如是命令解释器 bash，把老的可执行映像冲掉，将淘汰老的虚拟内存数据结构并重置进程的页表。它同时清除任何建立的信号处理器，并关闭任何打开的文件。最后进程已经为新的可执行映像作好准备。不管可执行映像是什么格式，相同的信息都将被设置到进程的mm_struct中。其中有映像的代码和数据的起始和结束指针。这些值在读取ELF可执行映像的物理头时可以得到，它们描述的程序段被映射

到进程的虚拟地址空间。也就在这时 vm_area_struct数据结构被设置，进程的页表被修改。Mm_struct数据结构也含有指向将被传到此程序的参数的指针以及指向此进程环境变量的指针。

ELF共享库

另一方面，一个动态链接映像并不包含运行时需要的所有代码和数据。一部分被保存在运行时被链接到映像的共享库中。在把共享库链接到映像时，动态链接器也要使用 ELF共享库的表格。Linux使用几种动态链接器：ld.so.1、libc.so.1和ld-linux.so.1。所有这些都可以在/lib下找到。库中包含通常使用的代码如语言子例程。如果没有动态链接，所有程序将需要有自己的这些库的副本，并需要更多的磁盘空间和虚拟内存。在动态链接中。每个引用的库例程信息都被包括在 ELF映像的表格中。这些信息指示动态链接器如何定位库例程以及如何把它链到程序的地址空间中。

3.8.2 脚本文件

脚本文件是需要解释器来运行的可执行文件。 Linux有许多可用的解释器，例如 wish、perl及命令shell如tcsh。Linux使用标准UNIX的习惯：用脚本文件的第一行包含解释器的名字。所以，一个典型的脚本文件将像下面这样开始：

```
#!/usr/bin/wish
```

脚本二进制装入器试着为脚本找到解释器。它通过试着打开在脚本第一行中提到的可执行文件来进行。如果能够打开，装入器将有一个其 VFS索引节点的指针，它可以继续让其解释脚本文件。脚本文件的名字作为参数 0(第一个参数)，而其他参数依次排列(原先的第一个参数成为新的第二个参数，等等)。装入解释器是用与 Linux装入其所有可执行文件一样的方式完成。Linux依次试用每种二进制格式直到一种成功。这意味着在理论上可以有几种解释器和二进制格式，使得Linux二进制格式处理器成为很灵活的软件。

第4章　进程间通信机制

内核用于协调进程间相互通信的活动。 Linux支持一部分进程间的通信 (Inter-Process Communication,IPC)机制。信号和管道是两种 IPC机制，但Linux也支持UNIX™ system V的IPC机制

4.1　信号机制

信号机制是UNIX系统使用最早的进程间通信机制之一，主要用于向一个或多个进程发异步事件信号，信号可以通过键盘中断触发、也可以由进程访问虚拟内存中不存在的地址这样的错误来产生。信号机制还可以用于 shell向它们的子进程发送作业控制命令。

系统内有一组可以由内核或其他的进程触发的预定义信号，并且这些信号都有相应的优先级。你可以使用 kill命令(kill-1)列出系统支持的所有信号。在作者的 Intel硬件平台的Linux系统上会产生如下的结果：

```
 1) SIGHUP       2) SIGINT     3) SIGQUIT     4) SIGILL
 5) SIGTRAP      6) SIGIOT     7) SIGBUS      8) SIGFPE
 9) SIGKILL     10) SIGUSR1   11) SIGSEGV    12) SIGUSR2
13) SIGPIPE     14) SIGALRM   15) SIGTERM    17) SIGCHLD
18) SIGCONT     19) SIGSTOP   20) SIGTSTP    21) SIGTTIN
22) SIGTTOU     23) SIGURG    24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF   28) SIGWINCH   29) SIGIO
30) SIGPWR
```

Alpha AXP硬件平台的Linux系统支持的信号数量与前面的不同。进程可以选择忽略上面的大多数信号，但 SIGSTOP和SIGKILL是不可忽略的。其中 SIGSTOP信号，使进程停止执行；而SIGKILL信号使进程中止。对于其他情况，进程可以自主决定如何处理各种信号：它可以阻塞信号；如果不阻塞，也可以选择由进程自己处理信号或者由内核来处理。由内核来处理信号时，内核对每个信号使用相应的缺省处理动作，例如：当进程收到 SIGFPE信号(浮点异常)时，内核的缺省动作是进行内核转贮 (core dump)，然后中止该进程。信号之间不存在内在的相对优先级。如果对同一个进程同时产生两个信号的话，它们会按照任意顺序提交给该进程，并且对同种信号无法区分信号的数量。例如：进程无法区别它收到了 1个还是42个SIGCONT信号。

Linux使用存贮在每个进程 task_struct结构中的信息实现信号机制， 它支持的信号数受限于处理器的字长，具有 32位字长的处理器有 32种信号，而像 Alpha AXP处理器有64位字长，最多可以有64种信号。当前未处理的信号记录在 signal域中，并把阻塞信号掩码对应位设置为阻塞状态。但对 SIGSTOP和SIGKILL信号来说，所有的信号都被设置为阻塞状态。如果一个被阻塞的信号产生了，就将一直保持未处理状态，直到阻塞被取消。 Linux中还包括每个进程如何处理每种可能信号的信息，这些信息被记录在 sigaction数据结构的矩阵中，由每个进程的task_struct指向sigaction矩阵。这些信息中包括处理信号例程的地址或者通知 Linux该进程选择

忽略信号还是由内核处理信号的标志。进程通过系统调用改变缺省信号的处理过程，这些系统调用会改变对应信号的sigaction结构和阻塞掩码。

并不是系统中的每个进程都可以向其他的进程发消息，只有内核和超级用户可以做到这一点。普通的进程只能向同一进程组或具有相同的 uid和gid的进程发送信号。信号可以通过设置task_struct结构signal域中相应中的位来产生。如果一个进程没有阻塞信号，正处于可中断的等待信号状态中，当等待的信号出现时 ,系统可以通过把该进程的状态变成运行状态 ,然后放入候选运行队列中的方法来唤醒它。通过上面的方法在下次调度时 ,进程调度器会把该进程作为候选运行进程进行调度。如果需要缺省处理的话， Linux可以优化信号的处理，例如：当出现SIGWINCH(X window焦点改变信号)信号时，如果没有进程的信号处理例程可以调用的话，系统会使用缺省处理过程。

信号产生后，并不立即提交给进程 ,它必须要等到进程再次被调度运行时。每当进程从系统调用中返回时，系统都会检查进程的 signal域和blocked域，以确定是否出现某些未阻塞的信号。这看起来非常不可靠，但实际上系统的每个进程都在不断地做系统调用，如向终端写字符。进程可以选择挂起在可中断的状态上，等待某一个它希望的信号出现， Linux的信号处理程序为当前每个未阻塞的信号查找 sigaction结构。

如果一个信号被设置为按缺省动作处理，那么内核会处理它。 SIGSTOP信号的缺省处理是把当前进程的状态改为停止状态，然后运行进程调度器选择一个新进程运行。 SIGFPE信号的缺省处理动作是对该进程进行内核转贮，然后中止该进程。相反，进程也可以指定自己的信号处理例程。在 sigaction结构中存贮有这个信号处理例程的地址，当信号产生时，这个例程就会被调用。内核必须调用信号处理例程，但如何调用是与处理器相关的。在调用信号处理例程时，所有的CPU必须要考虑到下面的几个问题：当前进程正在核态中运行，准备返回到用户态，而对信号处理例程的调用是由内核或系统例程来完成的。这个问题可以通过对栈和进程寄存器进行操作来解决，系统把进程的程序计数器置为进程信号处理例程的地址，并把例程的参数加到函数调用桢中或通过寄存器来传递参数。当进程重新开始运行时，信号处理例程就像被正常调用了一样。

Linux兼容POSIX标准，进程在某个信号处理例程被调用时，能指出哪些信号可以被阻塞。这意味着在调用进程信号处理例程时需要改变阻塞掩码，当信号处理例程结束时，阻塞掩码必须要恢复到初始值。因此 Linux增加了一次对清理例程的调用。清理例程按照信号处理例程的调用栈来恢复初始的阻塞掩码。在几个信号处理例程都需要被调用时， Linux也提供了优化方案。Linux把这些例程压入栈中，这样每当一个处理例程退出时，下一个处理例程立即被调用,当所有处理例程都完成后清理例程被调用。

4.2 管道

Linux shell通常支持重定向操作。例如对命令：

```
$ ls | pr | lpr
```

管道操作把列出目录中所有文件 ls命令的标准输出重定向为分页命令 pr的标准输入,接着pr命令的标准输出又被管道操作重定向为 lpr命令的标准输入 (lpr命令的作用是在缺省的打印机上打印)。管道是单向的字节流，它可以把一个进程的标准输出与另一个进程的标准输入连接起来。Linux的shell负责建立进程间的这些临时性的管道，而进程根本不知道这些重定向操作，

仍然按照通常的方式进行操作。

在Linux系统中，管道用两个指向同一个临时性 VFS索引节点的文件数据结构来实现。这个临时性的VFS索引节点指向内存中的一个物理页面。图 1-4-1表明每个文件数据结构包含指向不同文件操作例程向量的指针。一个例程用于写管道，另一个用于从管道中读数据。从一般读写普通文件的系统调用的角度来看，这种实现方法隐藏了下层的差异。当写进程执行写管道操作时，数据被复制到共享的数据页面中 ;而读进程读管道时，数据又从共享数据页中复制出来。Linux必须同步对管道的访问，使读进程和写进程步调一致。为了实现同步， Linux使用锁、等待队列和信号量这三种方式。

写进程使用标准的写库函数来写管道。使用文件操作库函数要求传递文件描述符来索引进程的文件数据结构集合。每个文件数据结构代表一个打开的文件或是一个打开的管道。Linux写系统调用使用代表该管道的文件数据结构指向的写例程，而写例程又使用代表该管道的VFS索引节点中保存的信息来管理写请求。

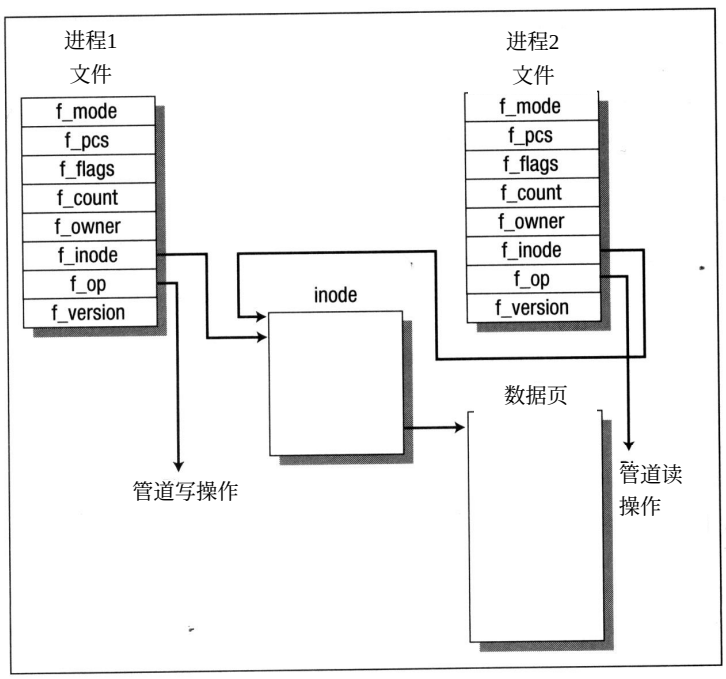


图1-4-1 管道

如果有足够大的空间把所有的数据写入管道中，并且该管道没有被读进程锁定，那么Linux为写进程锁定管道，把待写的数据从进程空间复制到共享数据页中。如果管道被读进程锁定或者没有足够大的空间存放数据，那么当前的进程被强制进入睡眠状态，放在管道对应的索引节点的等待队列中，然后系统调用进程调度器来选择合适进程进入运行状态。睡眠的进程是可中断的,它可以接收信号；也可以在管道中有足够大空间来容纳写数据或在管道被解锁时，被读进程唤醒。写数据完成后，管道的 VFS索引节点被解锁。系统会唤醒所有睡眠在读索引节点等待队列中的读进程。

从管道中读数据的过程与向管道中写数据非常相似。进程可以做非阻塞的读操作，但它依赖于打开管道的模式。进程使用非阻塞读时，如果管道中无数据或者该管道被锁定，读系

统调用会立即返回出错信息。通过这种办法，进程可以继续运行。另一种处理是进程在索引节点的等待队列中等待写进程完成。一旦所有的进程都完成了管道操作，管道的索引节点和共享数据页会立即被释放。

Linux也支持命名管道(named pipes)。因为这种管道遵循先进先出的规则，所以它也被称为FIFO(先进先出)管道。普通的管道是临时性的对象，而FIFO管道是通过mkfifo命令创建的文件系统中的实体。只要有适当的权限，进程就可以自由地使用FIFO管道。但FIFO管道的打开方式与普通管道有所不同：普通管道(包括两个文件数据结构：对应的VFS索引节点以及共享数据页)在进程每次运行时都会创建一次，而FIFO是一直存在的，需要用户打开和关闭。Linux必须处理读进程先于写进程打开管道、读进程在写进程写入数据之前读入这两种情况。除此之外，FIFO管道的使用方式与普通管道完全相同，都使作相同的数据结构和操作。

4.3 套接字

4.3.1 System V的进程间通信机制

Linux支持最早在UNIX System V中出现的三种进程间通信机制。它们是消息队列、信息量和共享存储器。这些System V的进程间通信机制使用相同的认证方法，即通过系统调用向内核传递这些资源的全局唯一标识来访问它们，Linux使用访问许可的方式核对对System V IPC对象的访问，这种方式与文件访问权限的检查十分相似。

System V IPC对象的访问权限是由该对象的创建者通过系统调用来实现的。Linux的每种IPC机制都把IPC对象的访问标识作为对系统资源表的索引，但访问标识不是一种直接的索引，而是由索引标识通过某些运算来产生的对象索引。

Linux系统中所有代表System V IPC对象的数据结构中都包括ipc_perm数据结构，在ipc_perm结构中有拥有者和创建者进程的用户标识和组标识、该对象的访问模式以及IPC对象的密钥。密钥的用处是确定System V IPC对象的索引标识。Linux系统中支持两种密钥：公共密钥和私有密钥。如果IPC对象的密钥是公共的，那么系统中的进程在通过权限检查后就可以得到System V IPC对象的索引标识。但要注意System V IPC对象不是通过密钥而是通过它们的索引标识来访问的。

4.3.2 消息队列

消息队列允许一个或多个进程向队列中写入消息，然后由一个或多个读进程读出(见图1-4-2)。Linux系统维护一个消息队列的表。该表是msgque结构的数组，数组中每个元素指向一个能完全描述消息队列的msqid_ds数据结构。一旦一个新的消息队列被创建，则在系统内存中会为一个新的msqid_ds数据结构分配空间，并把它插入到数组中。

每个msqid_ds结构都包含ipc_perm数据结构以及指向进入该队列的消息的指针。除此之外，Linux还记录像队列最后被更改的时间等队列时间更改信息。msqid_ds结构还包括两个等待队列；一个用于存放写进程的消息，另一个用于消息队列。

每次进程要向写队列写入消息时，系统都要把它的有效用户标识和组标识与该队列的

ipc_perm数据结构中的访问模式进行比较。如果进程可以写队列。那么消息会从进程的地址空间复制到一个 msg数据结构中，然后系统把该 msg数据结构放在消息队列的尾部。由于Linux限制写消息的数量和消息的长度，所以可能会出现没有足够的空间来存放消息的情况。这时当前进程会被放入对应消息的写等待队列中，系统调用进程调度器选择合适的进程运行。在该消息队列中有一个或多个消息被读出时，睡眠的进程会被唤醒。

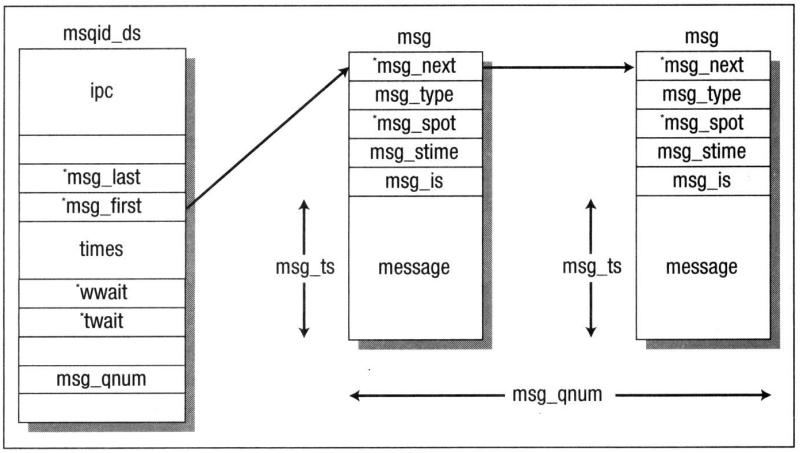


图1-4-2 System V IPC 消息队列

从队列中读消息的过程与前面相似，进程对写队列的访问权限会再次被核对。一个读进程可以选择获得队列中的第一个消息而不考虑消息的类型，还是读取某种特别类型的消息。如果没有符合要求的消息的话，读进程会被加入到该消息的读等待队列中，系统唤醒进程调度器调度新进程运行。一旦有新消息被写入消息队列。睡眠的进程被唤醒，并再次运行。

4.3.3 信号量

最简单的信号量是内存中的一个区域，它的值可以被多个进程执行 test_and_set操作(一种具有原子性的系统调用，用于测试某一地址的值然后再更改它)。test_and_set操作对每个进程来说是不可中断的，即具有原子性的操作。一旦一个进程执行该操作，其他的任何进程都不能打断它。Test_and_set操作的结果是对当前信号量的值进行增量操作，但增量可以是正的，也可以是负的。根据test_and_set操作的结果，进程可能会进入睡眠状态，等待其他进程改变信号量的值。信号量能用于实现关键段操作 (关键段指一段关键的代码段，同一时间内只有一个进程能执行该段操作)。

假如有许多相互协作的进程从一个数据文件中读取或写入记录，你会要求对文件的访问应是严格相互同步的。这样可以在文件操作的代码外面，使用两个信号量操作，并把信号量的初始值置为1。第一个操作是测试并减少信号量的值；第二个操作是测试并增加信号量的值。当第一个进程访问文件时，它会减少信号量的值，使信号量的值变为 0，这样第一个进程可以成功的进行文件操作了。这时若有另一个进程要访问文件而去减小信号量的值，信号量的值变为－1，从而这个进程被挂起，等待第一个进程完成数据文件的操作。当第一个进程完成文件操作时，它会增加信号量的值，使其再次变为 1。现在系统会唤醒所有的等待进程，这时第二个要访问文件进程的减1操作会成功。

System V的每个IPC信号量对象都对应一个信号量数组，在 Linux中用semid_ds数据结构来表示它(见图1-4-3)。系统中所有的 semid_ds数据结构都被一个叫 semary的指针向量指向。在每个信号量数组中都有 sem_nsems域，这个域由sem_base指向的sem数据结构来描述。所有允许对System V IPC信号量对象的信号量数组进行操作的进程，都必须通过系统调用来执行这些操作。在系统调用中可以指出有多少个操作。而每个操作包含三个输入项：信号量的索引、操作值和一组标志位。信号量索引是对信号量数组的索引值，而操作值是加到当前信号量值上的数值。首先 Linux会测试是否所有的操作都会成功 (操作成功指操作值加上信号量当前值的结果大于0，或者操作值和信号量的当前值都是 0)。如果信号量操作中有任何一个操作失败，Linux在操作标志没有指明系统调用为非阻塞状态时，会挂起当前进程。如果进程被挂起了，系统会保存要执行的信号量操作的状态，并把当前进程放入等待队列中。 Linux通过在栈中建立一个sem_queue数据结构，并填入相应的信息的方法来实现前面的保存信号量操作状态的。新的 sem_queue数据结构被放在对应信号量对象的等待队列的末尾 (通过使用sem_pendinging和sem_pending_last指针)，当前进程被放在sem_queue数据结构的等待队列中，然后系统唤醒进程调度器选择其他进程执行。

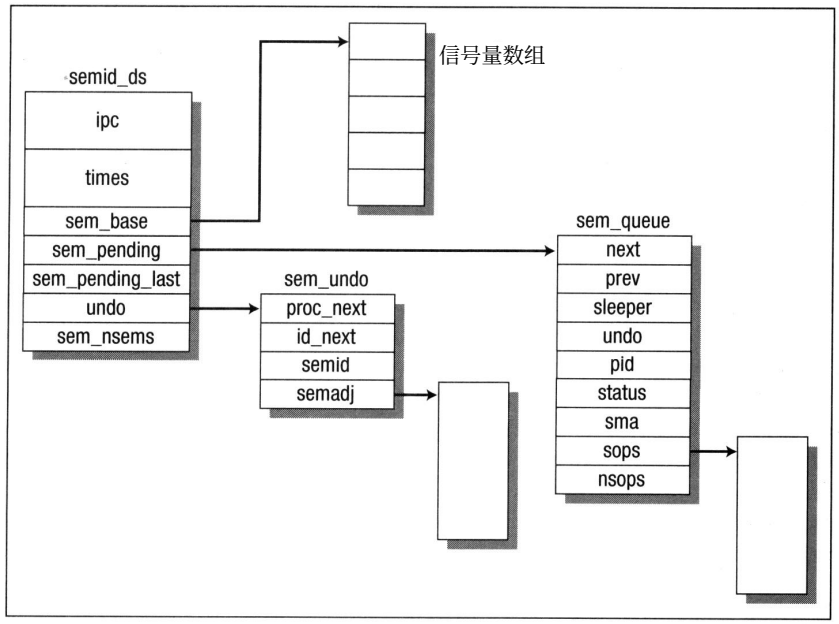


图1-4-3 System V IPC 信号量

如果所有的信号量操作都成功了，那么当前进程就不必挂起了。 Linux会继续运行当前进程，对信号量数组中的对应成员执行相应的操作。接着 Linux会查看那些处于等待状态被挂起的进程，以确定它们是否能继续持行信号量操作。 Linux会逐个查看等待队列中的每个成员，测试它们现在能否成功地执行信号量操作。如果有进程可以成功地执行了 ，Linux会删除未完成操作列表中对应的sem_queue数据结构，对信号量数组执行信号量操作，然后唤醒睡眠进程，将其放入就绪队列中。 Linux不断地查找等待队列，直到没有可成功执行的信号量操作并且也没有可唤醒的进程为止。

但信号量存在着死锁(deadlock)的问题，当一个进程进入了关键段，改变了信号量的值后，

由于进程崩溃或被中止等原因而无法离开关键段时，就会造成死锁。 Linux通过为信号量数组维护一个调整项列表来防止死锁。主要的想法是在使用调整项后，信号量会被恢复到一个进程的信号量操作集合执行前的状态。调整项被保存在 sem_undo数据结构中，而这些sem_undo数据结构则按照队列的形式放在 semid_ds数据结构和进程使用信号量数组的 task_struct数据结构中。

每一个单独的信号量操作都要求建立相应的调整项。 Linux为每个进程的每个信号量数组至多维护一个sem_undo数据结构。如果还没有为请求的进程建立调整项，那么当需要时，系统会为它创建一个新的 sem_undo数据结构。 sem_undo数据结构被加入到该进程的 task_struct数据结构和信号量数组的 semid_ds数据结构的队列中。一旦对信号量数组中某些信号量执行了相应的操作，那么该操作数的负值会被加入到该进程 sem_undo结构调整项数组的与该信号量对应的记录项中。因此，如果操作值是 2的话，那么－2就被加到该信号量的调整项中。

当进程被删除时，退出时 Linux会用这些sem_undo数据结构集合对信号量数组进行调整。如果信号量集合被删除了，那么这些 sem_undo数据结构还存在于进程的task_struct结构的队列中，而仅把信号量数组标识标记为无效。在这种情况下，信号量清理程序仅仅丢掉这些数据结构而不释放它们所占用的空间。

4.3.4 共享存储区

共享存储区允许一个或多个进程通过在它们的虚地址空间中同时出现的存储区进行通信。虚地址空间的页是通过共享进程的页表中的页表项来访问的。共享存储区不需要在所有进程的虚存中占有相同的虚地址。像所有的 System V IPC对象一样，共享存储区的访问控制是通过密钥和访问权限检查来实现的。一旦某一内存区域被共享了，系统就无法检查进程如何使用这部分内存区域。因此系统必须使用 System V信号量等其他的机制来同步对存储器的访问。

每个新创建的共享存储区由shmid_ds数据结构来表示，并被记录在shm_segs向量中(见图1-4-4)。shmid_ds数据结构中包含共享存储区的大小、当前使用该共享存储区的进程数目以及共享存储区如何映射到进程地址空间等信息。共享存储区的创建者设置对该共享存储区的访问许可权限，并确定它的密钥是公用的还是私有的。如果一个进程有足够的访问权限，就可以将共享存储区锁定到物理存储区域上。

每个想访问共享存储区的进程必须先通过系统调用，将该共享存储区连接到它的虚地址空间中。这个操作会创建一个描述该进程共享存储区的 vm_area_struct数据结构。进程既可以指定共享存储区放在它的虚地址空间的位置，也可以由 Linux自动选择一个足够大的自由空间。新的vm_area_struct数据结构被放入由shmid_ds指向的vm_area_struct结构的双向链表中。这个双向链表由vm_area_struct结构中的vm_next_shared指针和vm_prev_shared指针链接在一起。在执行连接操作时，系统实际上还没有创建该共享存储区，只有在第一个进程要访问共享存储区时，系统才会执行实际的创建工作。

当某一进程第一次访问共享存储区的某一页时，系统会产生一个页失效。 Linux在处理页失效时，它会找到描述该页的vm_area_struct数据结构。在vm_area_struct结构中包含处理这种共享存储区页失效的例程的句柄。共享存储区页失效处理例程会为 shmid_ds结构查找页表项的列表，以确定共享存储区中的这个页是否存在。如果不存在， Linux分配一个物理页，并为该页创建页表项。这个新的页表项会被同时保存到当前进程的页表和 shmid_ds结构中。这种

处理方法使得在下一个进程访问这个页，产生页失效时，共享存储区页失效处理例程会再次使用被分配的物理页。因此，第一个访问共享存储区某个页的进程会导致系统创建该共享页，而其他访问该共享页的进程仅仅会把该页增加到它们的虚地址空间中。

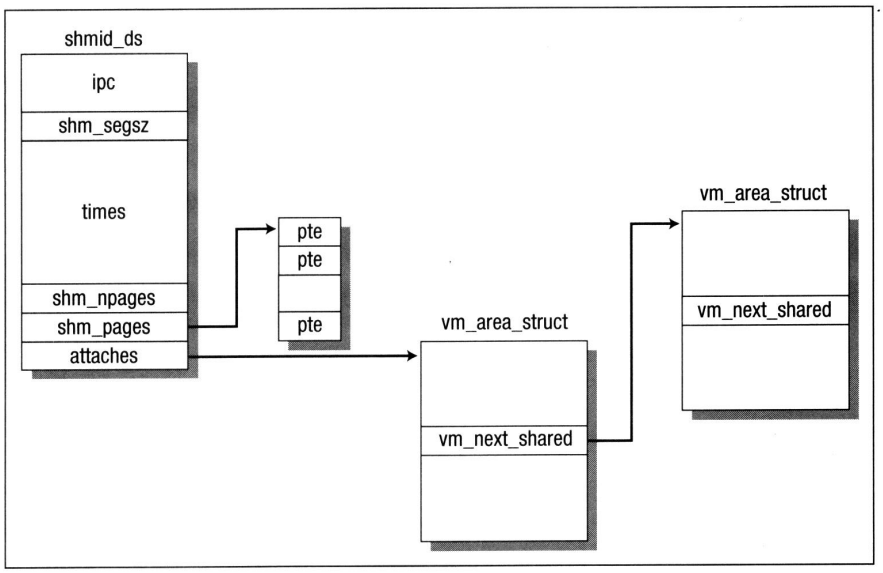


图1-4-4 System V IPC 共享存储区

当进程不再使用共享存储区时，进程会执行分离操作。只要还有其他的进程仍在使用这块存储区，分离操作就只会影响当前进程。进程的 vm_area_struct结构会被从shmid_ds结构中删除、释放掉，系统更新当前进程的页表以使原来被共享的虚地址区域无效。在最后一个使用共享存储区的进程执行分离操作时，处在物理存储器中的共享页面才会被释放掉，同时该共享存储区对应的shmid_ds数据结构也会被释放。

当共享存储区没有被锁定在物理存储区上时，会产生更复杂的情况。这时如果内存利用率比较高，共享存储区的页面会被交换到系统的磁盘交换区中。如何将共享存储区交换进和交换出物理存储器已在第2章中讨论过了，详细情况请参见第 2章。

第5章 PCI

PCI的英文全称为Periheral Component Interconnect。正如它的名称一样， PCI局部总线是微型计算机系统上处理器/存储器与外围控制部件、外围附加板之间的互连机构。“ PCI局部总线规范3 ”规定了互连机构的协议、电气、机械以及配置空间规范。本章主要介绍 Linux的内核如何初始化系统的PCI总线和设备。

图1-5-1是一个基于PCI局部总线的系统逻辑示意图。PCI局部总线和PCI-PCI桥是将系统的部件连接起来的连接器。 CPU连接到PCI局部总线0 ，这条总线主要用于连接视频设备。被称为PCI-PCI桥的特别PCI设备将PCI局部总线0与从PCI局部总线连接起来。这种“ 从 PCI总线 ”被称为 PCI局部总线1。按照PCI规范的术语来讲， PCI局部总线1被称作位于PCI-PCI桥的下游(downstream)，而PCI局部总线0被称为桥的上游(up-stream)。从PCI局部总线主要连接系统的 SCSI设备和以太网设备。桥、从PCI局部总线以及上面的两种设备都可以物理地集成在同一个 PCI集成卡中。系统中的 PCI-ISA桥支持老式、遗留下来的 ISA设备。在图1-5-1中画出了一个超级 I/O控制器芯片，它可以控制键盘、鼠标和软盘驱动器。

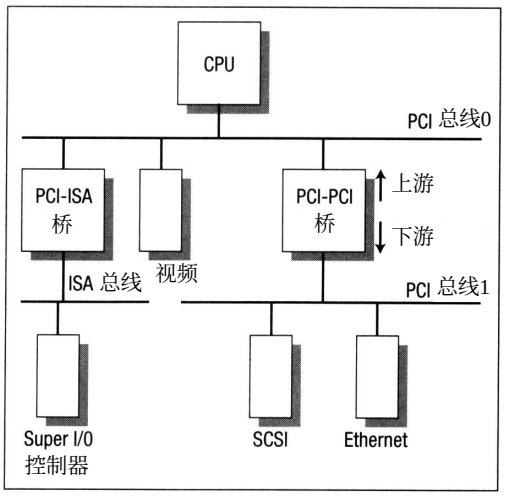


图1-5-1 基于PCI局部总线系统的示例

5.1 PCI的地址空间

CPU和PCI的设备需要访问在二者之间共享的存储空间。这部分存储器用于设备驱动程序控制PCI设备并在驱动程序和设备之间传递信息。典型的共享存储器包括设备的控制和状态寄存器，这些寄存器可以控制 PCI设备并读取设备的状态信息。例如： PCI的SCSI设备驱动程序从SCSI设备读取状态信息，以确定 SCSI设备是否可以向SCSI磁盘写入一整块的数据。 SCSI设备驱动程序还可以在启动后，向控制寄存器写入控制命令以启动 SCSI设备运行。

CPU的系统存储器可以用作共享存储器。但如果这样做的话，每次 PCI设备访问存储器时，CPU不得不暂停，等待PCI设备完成访存操作。这样访存操作就变成每次只有一个系统部件可以访存，导致整个系统的性能下降。让系统的外设以不加控制的方式来访问主存本身就不是一个好办法。这样做非常危险，因为一个劣质的外设会使系统非常不稳定。因此外设可以有自己的存储空间， CPU可以访问这些存储空间而外设对系统存储空间的访问用 DMA(Direct Memory Access，直接存储访问)通道的方式来严加控制。 ISA设备可以访问两类地址空间：ISA的I/O空间和ISA存储空间。 PCI设备可以访问三类地址空间： PCI的I/O空间、 PCI的存储空间和PCI的配置空间。所有的这些空间对 CPU来说都是可访问的，其中 PCI的I/O空间和PCI

存储空间被设备驱动程序使用， PCI的配置空间被Linux内核中的初始化程序使用。

Alpha AXP处理器不支持对除系统地址空间之外的其他地址空间的直接访问。它使用支持芯片组来访问像PCI配置空间这样的其他地址空间。实现上一般采用分析地址映射机制，即从整个虚拟地址空间中窃取一部分地址空间，并把它映射到 PCI的地址空间上。

5.2 PCI配置头

系统中的每个PCI设备(包括PCI-PCI桥设备)都在PCI的配置地址空间的某处有一个配置数据结构。PCI的配置头允许系统来标识、控制设备。而 PCI配置头在PCI配置地址空间的精确位置依赖于设备在PCI拓朴结构图中的位置。例如：一块 PCI视频卡插到PCI主板的一个PCI插槽中，它的配置头会出现在 PCI配置空间的一个地址上；如果把它换到另一个 PCI插槽中，那么它的配置头会出现在另一个地址上。但这种情况不会造成混淆，因为无论 PCI设备和桥在哪里，系统都会使用它们的配置头中的状态和配置寄存器找到并配置这些设备。

典型的系统一般都被设计成使每个PCI插槽的配置头的偏移量与插槽在主板上的位置相关。例如：主板上的第一个PCI插槽的PCI配置头的偏移量是0，第二个的偏移量是256(注：所有的配置头有相同的长度 —— 256个字节)，其他的以此类推。PCI总线规范定义了一种与系统相关的硬件机制，使得PCI的配置程序通过检验PCI总线上所有可能的PCI配置头中的一个域，就能区分哪些设备是连接在总线上的，那些是断开的。“ PCI局部总线规范3 ”定义了在读取空PCI插槽的厂商标识和设备标识域时，会返回值为0XFFFFFFFF的错误信息。因此PCI的配置程序一般读取配置头中的厂商标识域来判定PCI插槽的状态。如果返回错误信息0XFFFFFFFF，则说明插槽为空。

图1-5-2给出了256字节的PCI配置头的格式，它包含下列几个域：

- 厂商标识(Vendor Id) 是一个用于唯一标识 PCI设备生产厂商的数值。 Digital公司的 PCI厂商标识为0x1011，而Intel的是0x8086。
- 设备标识(Device Id) 用于唯一标识一个特定设备的数值。例如Digtal公司的21141快速以太网设备的设备标识为0x0009。
- 状态(Status) 根据 “ PCI局部总线规范3 ”定义的域中每个位的含义来给出设备的状态。
- 命令(Command) 系统通过向这个域中写入命令来控制设备。例如：可以写入允许设备访问 PCI I/O地址空间的命令。
- 分类码(Glass Code) 是设备类型的标识。规范对每种设备进行了标准的分类。如视频设备、 SCSI设备等等。SCSI的分类码为0x0100。
- 基地址寄存器(Base Address Register) 这些寄存器用于决定设备可以使用的 PCI I/O空间和存储空间的类型、大小，并指定它们的起始位置。

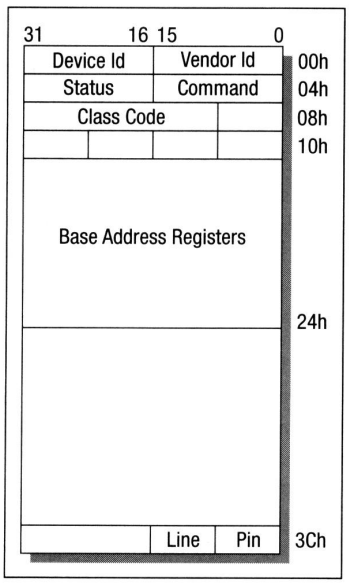


图1-5-2 PCI配置头

- 中断引脚(Pin) PCI卡有4个物理引脚，用于把中断从卡上发送到 PCI总线上。它们的标准标号为A、B、C、D。中断引脚(interrupt pin)域用于标识该PCI设备使用哪个引脚。一

般这种功能是由每个设备硬件实现的。也就是说，每次系统启动时，该设备都使用相同的中断引脚。这个信息的用处是让中断处理子系统可以处理来自该设备的中断。

- 中断线(Line) 设备的PCI配置头的中断线域用于在设备驱动程序的PCI初始化代码和Linux中断处理子系统间传递中断处理。写入中断线(interrupt line)域的数字对设备驱动程序来说是无意义的。但它允许中断处理器正确地把来自PCI设备的中断传送到操作系统中对应的设备驱动程序中断处理例程中。若想更进一步了解Linux如何处理中断，请看第6章。

5.3 PCI的I/O和存储地址空间

PCI设备共有两类地址空间可以用于与在核态运行的设备驱动程序进行通信。例如： DEC公司的21141快速以太网设备芯片把它的内部寄存器映射到 PCI的I/O地址空间中，这样它的设备驱动程序就可以通过读、写这些寄存器来控制快速以太网设备。典型的视频设备驱动程序通常会使用大量的PCI存储地址空间来获得视频信息。

在PCI系统建立之前和使PCI配置头中的命令域允许设备访问这些地址空间之前， PCI设备是不可设问的。在 Linux系统中，只有 PCI配置程序能读写 PCI的配置地址空间，而 Linux的设备驱动程序只能读写PCI的I/O和存贮器地址空间。

5.4 PCI-ISA桥

PCI-ISA桥通过把对PCI的I/O和存储地址空间的访问转换成对 ISA的I/O和存储器的访问，来实现对老的ISA设备的支持。现在售出的大量 PC系统都包括几个ISA总线插槽和几个PCI总线插槽。随着时间的推移，对这种向后兼容能力的需求会不断减少，最后会出现只有 PCI总线的PC系统。在ISA地址空间中， ISA设备有一些由早期基于 Intel 8080的PC定义的一些固定的寄存器。例如即使一台售价为 5000美元的Alpha AXP计算机系统，也会像第一台 IBM PC一样在ISA I/O地址空间的相同位置留有ISA软盘控制器。 PCI规范通过将PCI的I/O和存贮地址空间的低端保留给系统中的 ISA外设使用，用一个 PCI-ISA桥把对这部分 PCI存储空间的访问转换为对ISA地址空间访问的方式，解决了上述兼容问题。

5.5 PCI-PCI 桥

PCI-PCI桥是将系统中的所有PCI总线连接在一起的特殊的PCI设备。简单的系统可以只有一条PCI总线。但是一条总线可以支持的 PCI总线数受它的电气特性的限制。通过 PCI桥增加更多的PCI总线可以使系统支持更多的 PCI设备，这对一台高性能的服务器来说是至关重要的。当然， Linux完全支持使用PCI-PCI桥。

5.5.1 PCI-PCI桥：PCI I/O和存储地址空间的窗口

PCI-PCI桥只是把对部分PCI I/O和存储地址空间的读写请求传送到下游。例如：对图1-5-1，如果读写的是SCSI设备或以太网设备的 PCI I/O和存储地址空间， PCI-PCI桥会把这些读写从PCI总线0传送到PCI总线1，而忽略对其他PCI I/O和存储地址空间的访问。这种过滤功能防止了系统中不必要的地址传播。为了达到这个目标，必须对系统中的 PCI-PCI桥进行编程，为从主总线传送到从总线的PCI I/O和存储地址访问指定基址和上界。一旦系统中的 PCI-PCI桥配置成功了，那么只要Linux设备驱动程序，通过这些窗口访问 PCI的I/O和存贮器地址空间， PCI-

PCI桥对它们来说就是不可见的。这对 Linux PCI 设备驱动程序的编写者来说就是一个减轻工作量的重要特征。但它却使得 Linux配置PCI-PCI桥变得非常复杂。

5.5.2 PCI-PCI桥：PCI配置周期和PCI总线编号

由于CPU的PCI初始化代码可以查找出不在主 PCI总线上的设备。因此 PCI-PCI桥也一定有一种机制，使得它可以决定是否把配置周期从主接口传送到从接口上去。周期在 PCI总线上总以地址的形式出现。PCI规范定义了两种类型的 PCI配置地址：0型和1型，图1-5-3和图1-5-4给出了它们的格式。0型的PCI配置周期不包含总线号，由该总线上的设备像解释 PCI配置地址一样进行解释。0型配置周期的11至31位是设备选择域。一种设计系统的方式是由设备选择域的每一位选择一个设备，这时第 11位可以选择在插槽 0的PCI设备，第12位可以选择在插槽 1的PCI设备，并以此类推；另一种实现方式是把 PCI设备号直接写到第 11到31位中。一个系统使用哪种实现方式依赖于系统的 PCI存储控制器。

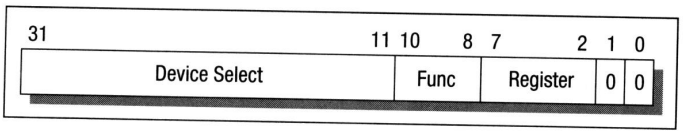


图1-5-3 0型 PCI配置周期

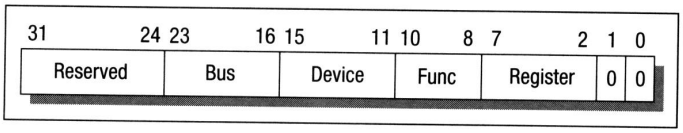


图1-5-4 1型PCI配置周期

1型PCI配置周期包含PCI总线号。除了PCI-PCI桥之外所有的PCI设备都忽略它。所有看到1型配置周期的PCI-PCI桥可以选择把它们传送到与自己相连的下游 PCI总线上。而PCI-PCI桥是忽略1型配置周期还是把它传送到下游的 PCI总线取决于PCI-PCI桥的配置。每个PCI-PCI桥都有一个主总线接口号和一个从总线接口号。其中主总线接口指离 CPU近的一端，而从总线接口指离CPU较远的一端。每个PCI-PCI桥还有一个下级总线号，它是从从总线接口桥接出去的所有PCI总线中最大的总线号。从另一角度来讲，下级总线号指位于 PCI-PCI桥下游的最大PCI总线号。在PCI-PCI桥收到1型PCI配置周期时，它可以做下列操作之一：

1) 如果指出的总线号不在桥的从总线号和下级总线号 (包括下级总线号)之间，桥简单地忽略它。

2) 如果指出的总线号等于桥的从总线号，桥将 1型配置周期转变为0型配置周期。

3) 如果指出的总线号大于桥的从总线号，而小于等于桥的下级总线号。桥将其无变化地传送到从总线接口上。所以要想在图 1-5-9的系统拓扑图中寻址总线 3上的设备1，CPU会在总线0上产生一个1型配置命令。桥1将此命令无变化地传到总线 1上，桥2会忽略它，而桥 3会把它转化为0型配置命令，发送到总线3上。在那儿设备1对其作出响应。

在PCI配置期间，由操作系统独立地分配总线号。但无论采用什么编号方式，系统中的PCI-PCI桥必须遵守如下规则：

所有在PCI-PCI桥下游的PCI总线号必须在该桥的从总线号和下级总线号 (包含它)之间。

如果这个规则被打破了，那么PCI-PCI桥就无法正确地传递和转换1型PCI配置周期，而操作

系统也无法发现、初始化系统中的所有PCI设备。为了遵循上述编号方式，Linux按一种特别的方式来配置这些特殊设备。5.6.2节根据一个工作示例详细讲述了Linux的PCI桥和总线编号机制。

5.6 Linux PCI初始化

Linux系统中的PCI初始化程序分成三个逻辑部分。

- PCI设备驱动程序　这个伪设备驱动程序从总线0开始搜索PCI系统，定位系统中所有的PCI设备和桥。它建立一个数据结构的链表来描述系统的拓扑结构。另外，它还为所有找到的桥分配编号。
- PCI BIOS　这一软件层提供了“PCI BIOS只读存储器规范 4”指出的所有服务。尽管Alpha AXP没有BIOS服务，但在Linux内核中有等价的代码来提供相同的功能服务。
- PCI修理部分　与系统相关的修理程序，用于整理与系统相关的PCI初始化过程中的故障点。

5.6.1 Linux内核PCI数据结构

Linux内核初始化PCI系统时，它将建立起能反映系统中实际的PCI拓扑结构的数据结构。图1-5-5给出了各数据结构之间的关系。它是从图1-5-1中的示例性PCI系统导出的。每个PCI设备由pci_bus数据结构来描述。最后产生的结果是一种树形结构的PCI总线，其中每个总线有若干个子PCI设备连接在上面。由于一个PCI总线只能通过PCI-PCI桥才能到达，所以除总线0以外，每个pci_bus数据结构都包括指向与其相连的位于上游的PCI-PCI桥的指针。PCI设备是所在PCI总线的父总线的子辈。在图1-5-5中给出了名为pci_devices的指向系统中所有PCI设备的指针，系统中每个PCI设备都将它们的pci_dev数据结构加入到这个队列中，该队列由Linux内核使用，以快速找到系统中所有的PCI设备。

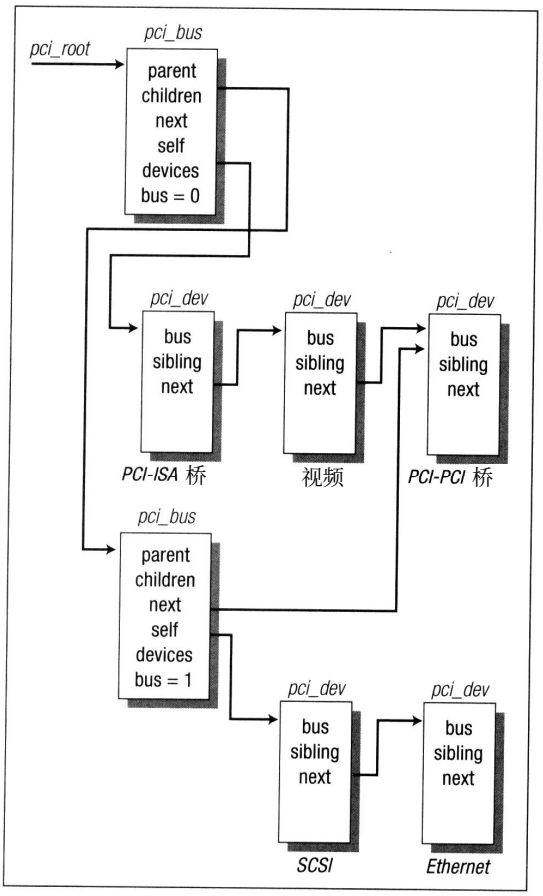


图1-5-5 Linux内核中PCI数据结构

5.6.2 PCI设备驱动程序

PCI设备驱动程序根本就不是一个真正的驱动程序，它只是在系统初始化时由操作系统调用的一个函数。PCI初始化程序必须先扫描系统中的所有PCI总线，查找系统中的所有PCI设备(包含PCI-PCI桥)。它使用PCI BIOS中的例程来确定当前正在扫描的PCI总线的每个PCI插槽是否是空的。如果PCI插槽被占用了，

它会建立一个描述该设备的pci_dev数据结构，并把它连入由pci_devices指针指向的已知PCI设备链表中。

PCI初始化程序从PCI总线0开始扫描，它对每个PCI插槽中的每个可能的PCI设备都读取厂商标识域和设备标识域。一旦找到了一个被占用的插槽，它会建立一个描述该设备的pci_dev数据结构，由PCI初始化程序建立的所有pci_dev数据结构(包括PCI-PCI桥)都被链入pci_devices指向的链表中。

如果PCI初始化程序发现某个PCI设备是桥，就会建立一个pic_bus数据结构，并把它链接到由pci_root指向的pci_bus和pci_dev数据结构的树中。由于PCI-PCI桥的分类码为0X060400，所以PCI初始化程序可以区分某个设备是否为PCI-PCI桥。Linux内核会立即配置在PCI-PCI桥下游的PCI总线；如果在那条总线上还有多个PCI-PCI桥，那么它们也会被配置。这个过程被称为深度优先算法(depthwise algorithm)。因此在广度搜索之前，系统的PCI拓扑结构先按深度优先的方式进行映射。图1-5-1中，Linux在配置PCI总线0上的视频设备之前，首先配置PCI总线1上的以太网和SCSI设备。

在Linux插索下游PCI总线时，它还要配置PCI-PCI桥的从总线号和下级总线号之间的间隔。这将在5.6.2小节详细介绍。

配置PCI-PCI桥——分配PCI总线号

要想让PCI-PCI桥传递对PCI I/O地址空间、存储地址空间和PCI配置地址空间的读写，需要知道下列的信息：

- 主总线(primary bus)号　与PCI-PCI桥直接相连的上游PCI总线的总线号。
- 从总线(secondary bus)号　与PCI-PCI桥直接相连的下游PCI总线的总线号。
- 下级总线(subordinate bus)号　由该桥的下游可以到达的所有PCI总线中最大的总线号。
- PCI I/O和存储器窗口　PCI-PCI桥所有下游地址的PCI I/O地址空间和PCI存储地址空间的窗口的基址和大小。

问题是当你想配置一个给定的PCI-PCI桥时，却无法知道该桥下级总线的数目。即使知道了，但仍不知道下游是否还有PCI-PCI桥以及给它们分配什么标号。解决的办法是使用反向的深度优先搜索算法，先搜索与PCI桥相连的每条总线，并为找到的总线分配标号。在找到每一个PCI-PCI桥和它的从总线号后，先为PCI-PCI桥分配一个0xFF的临时下级总线号，再去搜索它的下游的所有PCI-PCI桥并为其分配标号。上述算法看起来很复杂，但下面的例子有助于理解该算法。

1. PCI-PCI桥标号过程：第1步

使用图1-5-6的拓扑结构图，搜索过程会先发现桥1。与桥1相连的下游总线被标号为1，所以桥1的从总线标号为1而临时的下级总线标号为0XFF。这表示所有指出的PCI总线号大于等于1的1型配置地址都会穿过桥1，到达PCI总线1上。如果1型配置地址指出的总线号为1，则会被转换成0型配置周期，其他的总线号会被无变化的传送到总线1上。上面的过程正是Linux PCI初始化程序为了继续搜索PCI总线1所做的。

2. PCI-PCI桥标号过程：第2步

由于Linux使用深度优先算法，所以初始化程序会继续搜索总线1，在总线1上它找到了PCI-PCI桥2，由于在PCI-PCI桥2下面再没有PCI-PCI桥了，所以它为桥2分配了值为2的下级总线号。它与桥2从接口总线号相同。图1-5-7给出了在这一时刻初始化程序是如何为总线和PCI-PCI桥标号的。

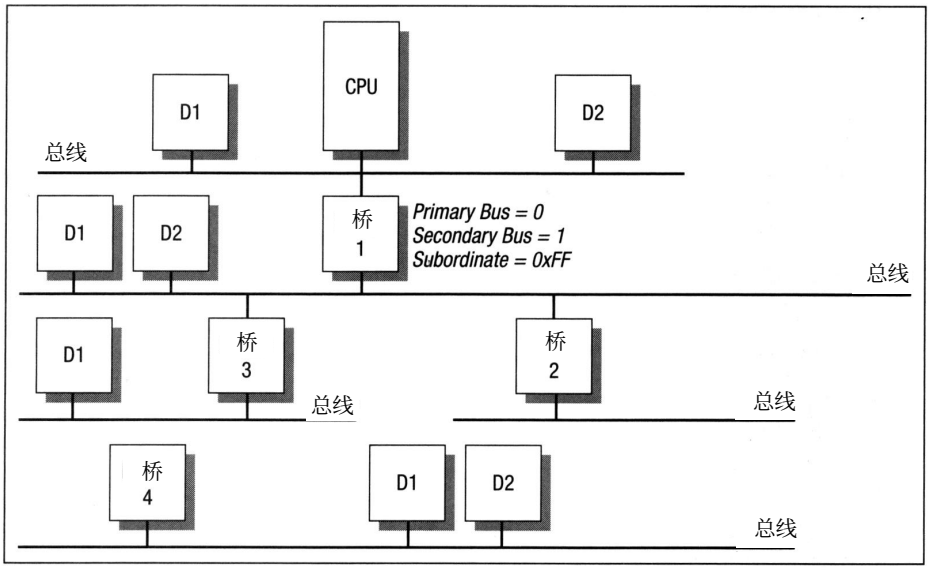


图1-5-6 配置一个PCI系统：第1步

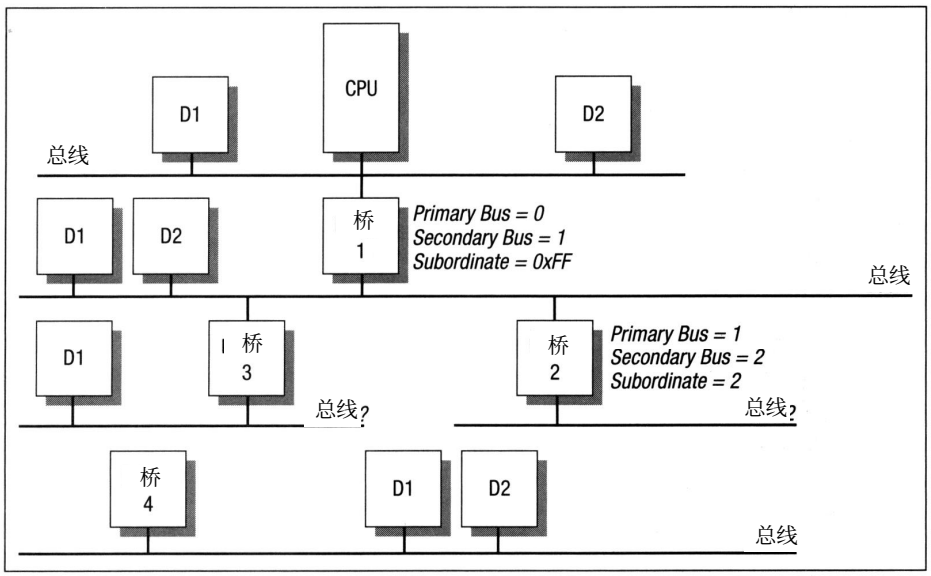


图1-5-7 配置一个PCI系统：第2步

3. PCI-PCI桥标号过程：第3步

PCI初始化程序回头继续搜索 PCI总线1，找到了另一个 PCI-PCI桥 —— 桥3。因此桥 3的主总线接口号为 1，从总线接口号为 3，临时的下级总线号为 0xFF。图1-5-8给出了目前系统的配置信息， 这时总线号为 1、2、3的1型PCI配置周期能被正确地传送到对应的 PCI总线上。

4. PCI-PCI桥标量过程：第4步

Linux开始搜索PCI-PCI桥4下游的PCI总线3。在PCI总线3上有另一个PCI桥4，因此桥4的

主总线号为3，从总线号为4，由于它是在这条分支上的最后一个桥，所以其下级总线号为 4。初始化回到PCI-PCI桥3时，给它分配4作为下级总线号。PCI初始化程序给桥1最后分配的下级总线号为4。图1-5-9给出了最终的总线和桥的标号情况。

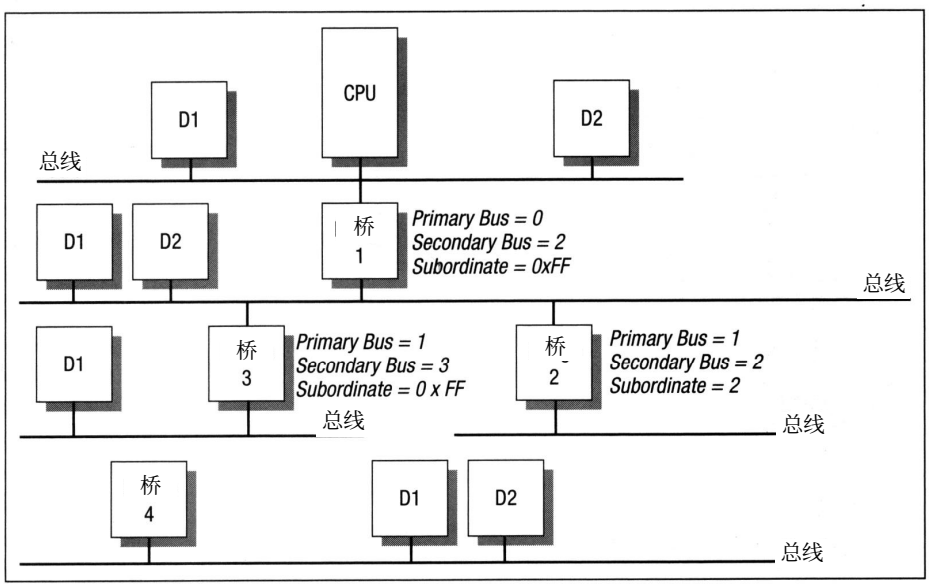


图1-5-8 配置一个PCI系统：第3步

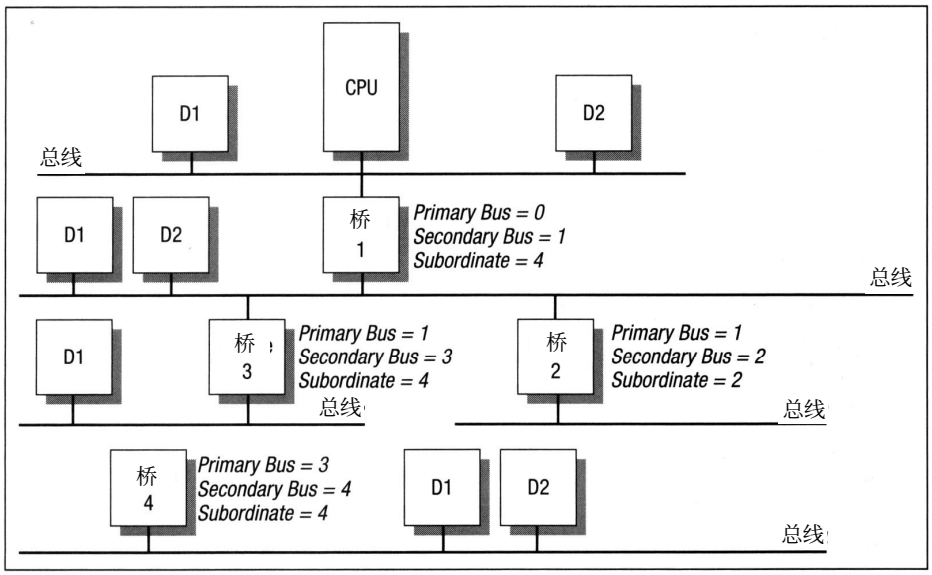


图1-5-9 配置一个PCI系统：第4步

5.6.3 PCI的BIOS函数

PCI的BIOS函数是一套跨越平台的通用标准例程。例如，在 Intel平台和Alpha AXP平台上它们都是相同的。BIOS函数允许CPU控制对所有PCI地址空间的访问，但只有 Linux内核和设

备驱动程序可以使用它们。

5.6.4 PCI修正过程

Alpha AXP 系统的PCI修正程序做的工作要比 Intel 多得多。由于Intel系统有系统BIOS，它在系统启动时运行，几乎完全配置好了 PCI系统。所以Intel系统的PCI修正程序除了对配配信息进行映射外，几乎没什么可做的。对于非 Intel的系统则需要做如下的进一步配置工作：

- 为每个设备分配PCI I/O空间和PCI 存贮器空间。
- 为系统中的每个PCI-PCI桥配置PCI I/O和存贮器地址窗口。
- 为设备设置中断线的值，通过这种方法控制设备的中断处理。

下面几小段讲述修正程序的工作过程。

1. 确定每个设备需要的PCI I/O空间和PCI存储空间的大小

修正程序对找到的每个PCI进行询问，以确定它需要的 PCI I/O空间和PCI存储地址空间的大小。为了达到这个目的，修正程序向每个设备的基址(Base Address)寄存器写入全1，然后读出。设备会在它不关注的地址返回 0，在其他位明确指出所需的地址空间大小。

系统中有两种基本类型的基址寄存器，用于指出设备寄存器是位于 PCI的I/O空间中，还是在PCI的存贮空间中。这是由寄存器的第 0位来指示的。图1-5-10给出了两种分别放在PCI的存贮空间和PCI的I/O空间的基址寄存器的形式。

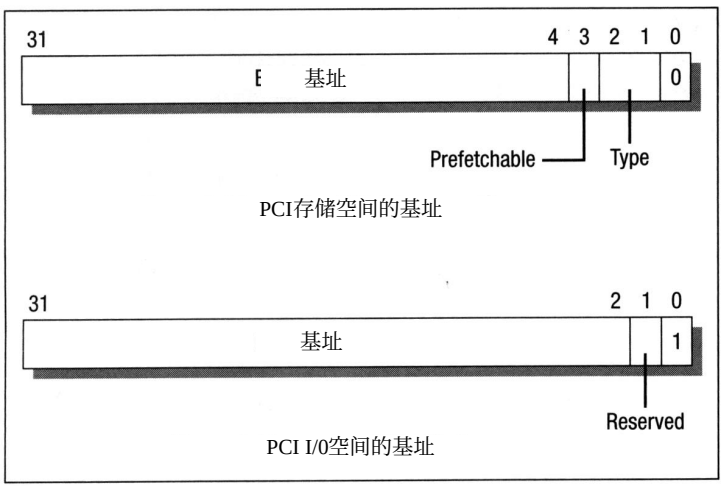


图1-5-10 PCI配置头：基址寄存器

为了确定一个基址寄存器需要多少地址空间，修正程序要先向寄存器中写入全 1，然后再从该寄存器中读取。设备会把不关注的地址位设为 0，在其余位明确给出所需的地址空间。这种方式表明使用的所有地址空间的大小都是 2的幂次，并且按自然边界对齐。例如：当你在初始化DEC片组的21142 PCI快速以太网设备时，该设备会通知你，它需要 0x100字节的PCI I/O地址空间或者 PCI存贮空间。初始化程序为它分配空间，一旦分配了空间以后， 21142的控制和状态寄存器在这些地址空间就是可见的了。

2. 为PCI-PCI桥和设备分配PCI I/O空间和PCI存贮空间

像所有的存贮器一样， PCI I/O和存贮空间是有限的，并且十分匮乏。非 Intel系统的PCI修正程序必须在考虑效率的原则下，为每个设备分配其所需数量的存储器。分配给每个设备的PCI I/O空间和存储空间，必须按自然边界对齐。例如：如果一个设备请求 0xB0大小的PCI I/O空间，那么它必须按照能被 0xB0整除的地址进行对齐。除此之外，任何一个桥的 PCI I/O空间和PCI存储空间的基址必须分别按照 4K和1M的边界进行对齐。由于任何一个下游设备的地址空间必须位于所有的上游 PCI-PCI桥存贮空间的地址范围内，所以有效的分配空间是比较困难的问题。

Linux使用的算法依赖于由 PCI设备驱动程序建立的总线 /设备树中的设备来分配 PCI的I/O地址空间，它是按照升幂的顺序来分配的。 Linux再次使用了一个反向算法来遍历由 PCI初始化程序建立的pci_bus和pci_dev数据结构。 BIOS的修正程序从由 pci_root指向的根PCI总线开始执行算法。

- 分别为当前的全局PCI I/O空间和存贮空间的基址按 4K和1M进行对齐。
- 对当前总线上的每个设备 (按所需PCI I/O空间大小的升幂顺序)执行下列操作：

—— 为其在PCI I/O空间和/或PCI存贮空间分配空间。

—— 按适合的数量调整全局 PCI I/O和存储空间的基址。

—— 允许设备使用PCI I/O和存储空间。

- 为当前总线的所有下游总线递归分配空间。注意这会改变全局 PCI I/O和存储空间的基址。
- 分别按4K和1M字节边界对当前的全局 PCI I/O和存储空间的基址执行对齐操作。在对齐操作的过程中，计算出当前 PCI-PCI桥所需的PCI I/O窗口和PCI存储窗口的大小和基址。
- 对PCI-PCI桥进行编程，将当前总线与它的 PCI I/O空间和PCI存储空间的基址、上界链接起来。
- 打开PCI-PCI桥对PCI I/O空间和PCI存储空间访问的桥接功能。这表示任何对桥的主 PCI总线的PCI I/O 空间和PCI存贮空间的访问，如果落在它的PCI I/O和PCI存储地址窗口内，都会被桥接到它的从PCI总线上。

以图1-6-1中的PCI系统作为例子， PCI修正程序将按如下的方式建立系统：

- 对齐PCI基址 PCI的I/O空间是按4K边界进行对齐的，而PCI存储空间是按1M边界对齐。这种方式允许PCI-ISA桥把到它下游的所有地址转换成ISA地址周期。

视频设备 视频设备需要2M的PCI存储空间，因此我们在当前的 PCI存储空间中为它分配基址为0x200000的2M空间。很明显它与设备要求的内存大小是自然对齐的。当前 PCI存贮空间的基址移到了0x400000，而PCI I/O空间的基址仍保持在0x4000处。

PCI-PCI桥 现在通过PCI-PCI桥为它下游的设备分配了 PCI存储空间。由于基址已经正确地对齐了，所以不必再去做基址对齐操作了。

a. 以太网设备 该设备要求0xB0字节的PCI I/O空间和PCI存储空间。因此修正程序为该设备的PCI I/O空间分配的基址为 0x4000，为PCI存储空间分配的基址为 0x400000。当前PCI存储空间的基址移到0x400B0处， PCI I/O空间的基址移到0x40B0处。

b. SCSI设备　该设备需要1K的PCI存储空间。因此在执行自然边界对齐操作后，为该设备分配的PCI存储空间的基址为0x401000。当前的PCI I/O空间的基址仍为0x40B0，而当前PCI存储空间的基址移到了0x402000。

PCI-PCI桥的PCI I/O空间和存储空间的窗口：

我们再回到上游的PCI-PCI桥，为它设置PCI I/O空间的窗口的范围为0x4000到0x40B0，它的PCI存储空间窗口的范围为0x400000到0x402000。上面的设置表示，如果是以太网设备和SCSI设备的话，PCI-PCI桥会忽略对视频设备的PCI存储空间的访问，并把它们转发到下游总线上。

第6章　中断处理与设备驱动程序

本章介绍Linux内核中中断的处理机制及 Linux内核是如何管理系统中的物理设备的。对于中断，尽管大多数的中断处理细节是与体系结构相关的，但内核中同时仍有一些通用的处理中断的机制和接口。操作系统正是通过设备驱动程序为用户隐藏下层硬件设备的细节（如虚文件系统为所有安装的文件系统向上层提供一个统一的视图，而与下层的物理设备无关），而设备驱动程序与中断处理息息相关，因此本章先介绍 Linux内核中中断的处理机制，并基于此对Linux中的设备驱动程序做进一步的介绍。

6.1　中断与中断处理

Linux使用大量不同的硬件去执行不同的任务，例如： Linux用视频设备驱动监视器(monitor)，用IDE设备驱动磁盘。你可以用同步的方式来驱动这些设备，这种方法也就是向设备发送某种操作的请求(如：请求把一块内存写入磁盘中)，然后等待设备完成操作。这种方法虽然能正确工作，但效率非常低，操作系统在等待每个操作完成期间会浪费大量时间。一个更高效的方法是先向设备发送请求，然后做其它更有用的工作。当设备完成操作系统的请求后，它会中断操作系统的运行。使用这种机制，同一时刻可以向系统中的设备发出许多未完成的请求。

现在有支持设备中断当前 CPU运行的硬件。大多数的通用处理器 (如：Alpha)都使用十分相似的中断方式。 CPU的某些引脚与电子线路相连，通过改变电子线路上的电压值 (如：从+5V变为-5V)就可以暂停CPU当前运行的程序，使它转去执行用于处理中断的特殊程序 —— 中断处理例程。这些引脚中有一个还可以与间隔计时器相连，每隔 1/1000秒就能收到一个中断。其它的引脚可以连接到系统中像 SCSI控制器等设备上。

在把中断信号传送到 CPU的一个中断引脚上时，系统经常用中断控制器对设备中断进行分组。这种办法节约了 CPU上的中断引脚，也提高了设计系统时的灵活性。中断控制器有控制中断的屏蔽和状态寄存器。设置屏蔽寄存器某一位可以允许或禁止对应的中断，而状态寄存器用于返回系统中当前活跃的中断。

系统中的某些中断是硬连线实现的，如：实时时钟的间隔定时器是永久连接到中断控制器的第3脚上的；而其它引脚所连接的中断内容却是由插在某个 PCI或ISA插槽上的某种控制卡来决定的。例如中断控制器的引脚 4连接到0号PCI插槽上，而0号PCI插槽可能某天插的是以太网网卡，而另一天是 SCSI控制器。根本问题是每个硬件系统有自己的中断路由机制，所以操作系统必须要能灵活地处理。

大多数现代的通用微处理器按相同的方式来处理中断。当发生硬件中断时， CPU会暂停正在执行的指令转移到内存中的某个地址，在这个地址处或者包含中断处理例程、或者是用于转移到中断处理例程的指令。这部分指令通常在 CPU的中断模式下执行，一般没有其它的中断可以在这个模式下发生。但对某些处理器仍然有特殊的情况，某些 CPU把中断按优先级分类，高级中断可以打断低级的中断。这说明第一级的中断处理代码必须得仔细编写，它经

常要有用于存储CPU处理中断前执行状态的栈(CPU的执行状态包括CPU的所有通用寄存器的值和上下文)。某些CPU有一组只在中断模式下才可见的寄存器，中断处理例程可以用这些寄存器去做大多数的上下文(context)保存工作。

在中断处理完成后， CPU的执行状态被恢复，中断被解除了。 CPU继续执行中断前的工作。把中断处理例程设计的尽可能高效对操作系统来说是非常重要的，而操作系统也不应太频繁或太长时间的阻塞中断的发生。

6.1.1 可编程中断控制器

若不是IBM的PC使用了Intel 82C59A-2 CMOS可编程中断控制器及其后继产品，系统设计者本可以自由选择他们想要的中断体系结构。这个控制器在 PC诞生时就出现了，它可以通过在ISA地址空间众所周知地址上的寄存器来进行编程。现在即使是一个十分先进的 CPU支持逻辑芯片组都会在 ISA存贮空间的相同位置保留有功能等价的寄存器组。而非 Intel的系统(如Alpha AXP系统)可以不受这些体系结构的限制，因而经常使用不同的中断控制器。

图1-6-1画出了两个链接在一起的 8位中断控制器。每个控制器都有一个名为 PIC1的屏蔽寄存器和一个名为PIC2的中断状态寄存器。两个中断屏蔽寄存器放在0X21和0XA1的地址处，两个状态寄存器分别在 0X20和0XA0处。向中断屏蔽寄存器的某一位写入 1允许中断，而写0就禁止中断。因此向中断屏蔽寄存器的第 3位写入1能够允许中断3，而写入0即禁止中断3。很令人苦恼的是中断屏蔽存器是只写的，你无法读取刚刚写入的值。这使得 Linux不得不保留一个中断屏蔽寄存器的本地副本，每次在中断允许和中断禁止例程中先改变副本的值，然后再把副本完全写入中断屏蔽寄存器中。

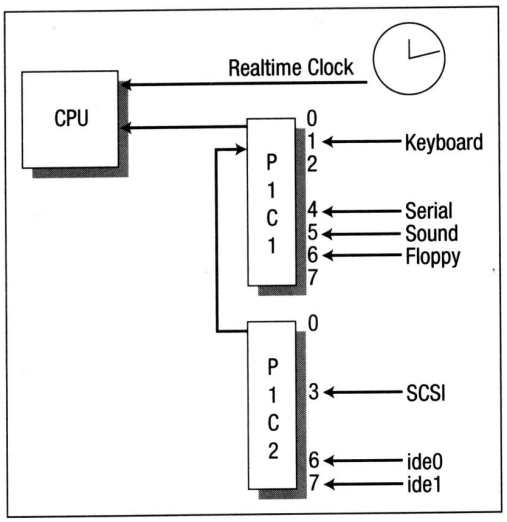


图1-6-1 中断路由逻辑图

中断产生时，中断处理例程读取两个中断状态寄存器的值。它把 0X20处的中断状态寄存器的值放在 16位中断寄存器的低 8位，而把0XA0处的中断状态寄存器的值放在高 8位。因此在0XA0处中断状态寄存器的第一位的中断会被当作系统的第 9位中断。 PIC1的第2位是不可用的，因为它用于连接来自 PIC2的中断的，而PIC2上的任何一个中断都会使 PIC1的第2位置1。

6.1.2 初始化中断处理数据结构

内核的中断处理数据结构是由设备驱动程序在请求系统中断控制时建立起来的。为了建立这些结构，设备驱动程序使用 Linux内核中的一组用于请求中断、允许中断、禁止中断的服务。每个设备驱动程序调用这些例程注册它们的中断处理例程的地址。

有些中断由于遵守 PC体系结构的习惯而被固定下来，因此这种类型的驱动程序在初始化时只是简单地请求它的中断，这正是软盘设备驱动程序所做的 —— 它总是请求IRQ 6。在某些

情况下设备驱动程序无法知道设备使用哪个中断；这对 PCI设备驱动程序是不成为问题的。因为驱动程序总是知道它们的中断号是多少。而让 ISA设备驱动程序找到它们的中断号就不是一个简单的问题。Linux通过允许设备驱动程序检测它们自己中断的方式解决了这个问题。

设备驱动程序先对设备进行某些操作，使设备产生中断。然后它把系统中所有未被分配的中断置成允许状态。这个操作表示该设备的待处理的中断会通过可编程中断控制器传送过来。Linux读中断状态寄存器并把它的值返回给设备驱动程序，一个非 0的值表示在检测期间有一个或多个中断发生了。驱动程序接下来关闭检测并把所有未分配的中断置成禁止状态。如果ISA设备驱动程序成功地找到了它的 IRQ号，那么就可以像正常的驱动程序一样请求中断控制。

基于PCI的系统比基于ISA的系统有更多的动态性。 ISA设备使用的中断引脚是由硬件设备上的跳线来设定的，并且在设备驱动程序中是固定的，而 PCI设备是在系统启动时 PCI初始化过程中由PCI BIOS或PCI子系统来分配中断号的。每个 PCI设备可以使用A、B、C、D四个中断引脚中的一个。在设备生产时，设备的中断引脚就是固定的了，而且大多数设备缺省使用引脚A上的中断。每个PCI插槽上的PCI中断线A、B、C、D都被路由到中断控制器上。所以PCI插槽4上的引脚A就可能路由到中断控制器的引脚 6，而该插槽的引脚 B被路由到中断控制器的引脚7，其它的引脚就以此类推。

PCI中断如何路由完全是由系统决定的。系统中有一些建立程序用于掌握 PCI中断路由的拓扑结构。在基于 Intel的PC上这些建立程序是系统启动时运行的 BIOS中的代码，而对像 Alpha AXP那样没有BIOS的系统，由Linux内核做这部分建立工作。 PCI建立程序把每个设备的中断控制器的引脚号写入到它的 PCI配置头中。它使用 PCI中断路由拓扑结构信息， PCI设备的插槽号以及PCI设备使用的中断引脚号来共同决定该设备的中断号。一个 PCI设备使用的中断引脚是固定的，并记录在该设备的 PCI配置头的一个域中。 PCI建立程序把中断号记录在专门为中断保留的中断线域中，当设备驱动程序运行时，它从配置头中读取中断号信息，并使用它向Linux内核请求中断控制。

在使用PCI-PCI桥时，系统中会有很多 PCI中断资源，因此中断源的数量可能会超过系统的可编程中断控制器的引脚数。在这种情况下， PCI设备要共享中断，即中断控制器的一个引脚要接受一个以上 PCI设备的中断请求。 Linux通过允许中断源的第一个请求者声明该中断是否是可以共享的来支持中断共享机制。共享中断使得 irq_action vector向量中的一项指向若干个irqaction数据结构。当一个共享中断发生时， Linux会调用该中断源的所有中断处理例程。因此，任何支持共享中断的设备驱动程序必须为其中断处理例程被调用而没有中断等着处理这种情况做好准备。

6.1.3 中断处理

Linux中断处理子系统的一个基本任务是把中断路由到正确的中断处理程序去。因此这部分代码必须懂得系统的中断拓扑结构。例如对中断控制器引脚 6上发生的软盘控制器中断，中断处理子系统必须识别出该中断来自于软盘控制器并把它传送给软盘设备驱动程序的中断处理例程。 Linux使用一组指针来指向包含处理系统中断的例程地址的数据结构。这些例程属于系统设备的设备驱动程序，并由设备驱动程序在初始化时请求这些例程使用的中断控制。在图1-6-2中，irq_action是一组指向irqaction数据结构的指针，每个 irqaction数据结构包含关于

该中断处理例程的信息，其中有中断处理例程的地址。由于中断的数量和处理方式随体系结构和系统的不同而不同，所以Linux中断处理程序是与体系结构相关的，这就意味着 irq_action vertor向量的大小取决于系统中中断源的数目。

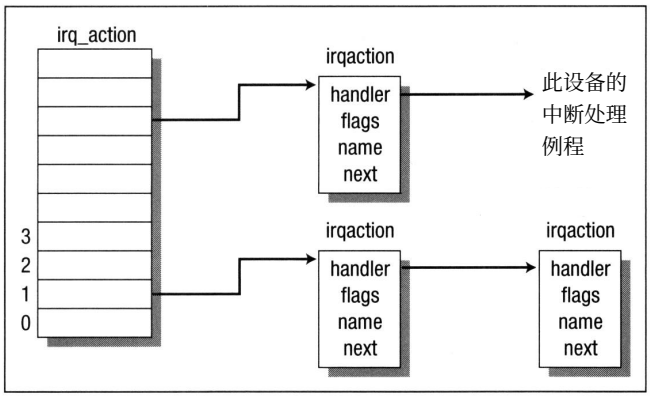


图1-6-2 Linux中断处理数据结构

当中断发生时，Linux通过读取系统的可编程中断控制器中中断状态寄存器的值来确定中断源。然后Linux把中断源转换成相对于irq_action vector向量的偏移。如对在中断控制器引脚6的来自软盘控制器的中断，Linux会把它转换为中断处理向量的第七个指针。如果系统对发生的中断没有对应的中断处理例程，那么 Linux内核会记录一个错误信息。否则的话它会调用该中断源对应的所有irqaction数据结构中的所有中断处理例程。

在设备驱动程序的中断处理例程被 Linux内核调用时，它必须立即确定中断原因并做出响应。为了查找中断原因，设备驱动程序会读取发出中断的设备的状态寄存器，而设备可能报告一条错误信息或是请求操作完成的信息。如软盘控制器可能会报告它已把软驱的磁头定位在软盘的正确扇区上了。一旦中断原因被查明了，设备驱动程序可能需要做进一步的处理工作。如果设备驱动程序要做进一步的工作，Linux内核提供一种允许驱动程序延缓处理工作的机制，它可以避免 CPU在中断模式下花费太多的时间。要想了解进一步的信息，请看 6.2节“设备驱动程序”。

6.2 设备驱动程序

CPU并不是系统中唯一的智能设备，每个物理设备都有它自己的控制器，键盘、鼠标、串口的控制器是SuperIO芯片，IDE磁盘的控制器是IDE控制器，SCSI磁盘的控制器是SCSI控制器……每个硬件控制器都有自己的控制和状态寄存器组 (CSR)并随设备的不同而不同。如Adaptec 2940 SCSI控制器的CSR就完全不同于NCR 810 SCSI控制器的CSR。CSR主要用于启停设备、初始化设备以及诊断设备的故障，Linux并不是把系统中的硬件控制器的管理程序放在应用程序中，而是把这些程序全放在内核里。设备驱动程序指用于处理、管理硬件控制器的软件。Linux内核中的设备驱动程序是一组长驻内存具有特权的共享库，也是一组低级的硬件处理例程。系统正是用Linux的设备驱动程序处理它所管理设备的特殊性问题。

UN*X的一个基本特征就是它抽象了设备的处理。所有的硬件设备都与常规的文件十分相似，它们可以通过与操纵文件完全一样的标准系统调用来打开、关闭、读和写。系统中的每个设备由一个特殊设备文件来表示，如系统中的第一个 IDE硬盘由/dev/hda文件表示。对于块

设备和字符设备，这些特殊设备文件可以由 mknod命令创建，并由主次设备号来描述对应的设备。网络设备也可以由特殊设备文件来表示，但它是在 Linux查找初始化网络控制器时建立的。所有由同一个设备驱动程序驱动的设备有相同的主设备号，次设备号用于把这些不同的设备及它们的控制器区别开。如主 IDE硬盘的每个分区有不同的次设备号。所以 /dev/hda2是主IDE硬盘的第二个分区，它的主设备号为 3，次设备号为2。Linux通过用主设备号和一组系统表格(如：字符设备表— chrdevs)，在系统调用中把特殊设备文件 (假定在块设备的装配文件系统中)映射到设备的设备驱动程序上。 Linux支持三种硬件设备类型：字符设备、块设备、网络设备。字符设备是支持无缓存读写的设备，如系统的串口 /dev/cuao和/dev/cual。块设备只能按多个块的大小进行读写，典型的块大小是 512字节或1024字节。块设备是通过缓冲区缓存来访问的，并支持随机地访问 —— 即无论该块在设备的何处，都能够直接读写。块设备能通过特殊设备文件来访问，但大多数情况下是通过文件系统来访问的。只有块设备才支持安装的文件系统。网络设备通过 BSD套接字接口来访问的，在第 8章网络中对网络子系统有详细的介绍。

Linux内核中有许多不同的设备驱动程序，但它们有一些共同的属性：

内核程序　设备驱动程序是内核的一部分，像其它内核中的程序一样，如果出错可能会严重地损害系统。一个编写得很差的驱动程序甚至会使系统崩溃，也可能损坏文件系统而造成数据丢失。

内核接口　设备驱动程序为Linux内核或内核的子系统提供了一套标准的接口。例如：终端驱动程序为Linux内核提供了文件I/O接口，而SCSI设备驱动程序为SCSI子系统提供了SCSI设备接口，又为内核提供了文件I/O和缓冲区缓存接口。

内核机制和服务　设备驱动程序可以使用像存储分配、中断转接、待操作队列这些标准内核服务。

可加载性　大多数的Linux设备驱动程序可以像内核的模块一样在需要时载入内存，在不使用时卸载，这种方式使得内核在处理系统资源方面适应性强、效率很高。

可配置性　Linux设备驱动程序可以安装到内核中，在内核被编译时，这些内核中的设备驱动程序是可配置的。

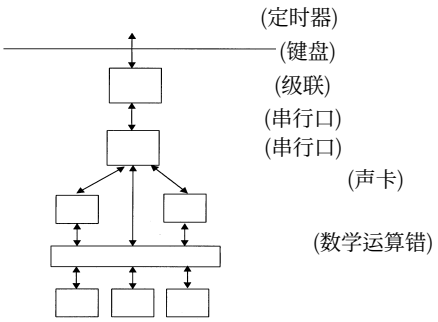
动态性　系统启动时每个设备驱动程序初始化，查找它所控制的硬件设备。如果某个设备驱动程序对应的设备不存在的话，那么该设备驱动程序就变成冗余的驱动程序，除了占用一点系统存储空间以外，不会造成任何损害。

6.2.1 测试与中断

每当设备接收一个类似于“ 将读磁头移到软盘的第 42个扇区 ”的命令时，设备驱动程序可以有两种方式来确定命令的完成情况：不断地测试设备或等待设备的中断。

测试设备表示驱动程序不断地读设备的状态寄存器，等待设备状态寄存器的状态变为操作完成状态。作为内核一部分的设备驱动程序，如果不断测试设备的话，会使得内核在设备完成请求之前无法运行其它的程序。使用测试方式工作的设备驱动程序还可以用系统定时器，让内核在一定时间间隔之后调用设备驱动程序中的一个例程，该定时器例程被调用时会检查设备的命令状态。这种方式是 Linux软盘驱动程序的工作机制。采用定时器的测试机制近乎于是一种最佳的工作机制。但使用中断仍是一种比它更有效的一种方法。

中断驱动的设备驱动程序是指该驱动程序控制的硬件设备在需要服务时，会向驱动程序发硬件中断。例如对一个以太网网卡设备，它在收到网络上的以太网报文时会向以太网驱动程序发中断。Linux内核要具有把来自硬件设备的中断转交给正确的设备驱动程序的能力。这是由设备驱动程序向内核登记它的中断使用情况来实现的。可以通过查看 /proc/interrupts来确定设备驱动程序使用中断的情况，以及当前系统中有多少种中断类型：



对中断资源的申请要在驱动程序初始化时完成。但系统中的一部分中断是固定的，它是IBM PC机体系结构中遗留下来的。如：软盘控制器永远使用中断 6。其它的中断与来自于PCI设备的中断一样是在启动时动态分配的。在这种情况下，设备驱动程序在请求中断拥有权之前必须先找到它所控制的设备的中断号。对 PCI设备的中断， Linux支持标准的PCI BIOS回调函数来检测包括中断号在内的系统中的设备信息。

中断传送到CPU的方式是与体系结构相关的，但在大多数体系结构中中断是通过先阻止系统产生其它中断，这种特别的方式来传送中断的。设备驱动程序应该在中断处理例程中做尽可能少的工作，以使得 Linux内核尽可能快地解除中断，继续执行中断前的任务。那些收到中断后需要做大量处理的驱动程序可以使用内核的 bottom_half处理例程或任务队列机制，将处理例程加入到队列中，使得它在不久后能够被调用。

6.2.2 直接存储器访问(DMA)

在数据量少的时候，使用中断驱动的设备驱动程序向硬件设备传送数据或从硬件设备读取数据工作得比较好。如一个 9600bps的调制解调器可以大约每 1/1000秒传送一个字符。如果中断延迟 —— 从硬件产生中断到设备驱动程序的中断处理例程被调用之间的时间开销，比较低的话(如2/1000秒)，那么数据传输占用的系统的开销就比较低。 9600bps 调制解调器的数据传输大约只花费CPU 0.002%的处理时间。但对像硬盘控制器或以太网设备这些数据传输率高的设备，一个SCSI设备就可以每秒传输40M字节的信息。

DMA(Direct Memory Access，直接存储器访问)就是用于解决这个问题的。 DMA控制器可以在不打扰 CPU的情况下,允许设备将数据传输到存储器中或从存储器中读取数据。一个ISA的DMA控制器有8个DMA通道，其中有7个对设备驱动程序是可用的。每个 DMA通道都有一个16位的地址寄存器和一个 16位的记数寄存器。在初始化数据传输时，设备驱动程序先设定DMA通道的地址寄存器、记数寄存器以及数据传输的方向 —— 读还是写；然后它通知设备可以在就绪后启动 DMA传输了。传输完成时，设备会中断 CPU。在整个传输过程中， CPU完全不受影响可以做任何其它工作。

设备驱动程序在使用 DMA时必须小心：第一点是所有的 DMA控制器根本不知道虚存，它

只能访问系统中的物理内存。而且 DMA传入或传出的数据只能是连续的物理内存块，这就表明你无法对进程的虚地址空间使用 DMA操作；但在DMA操作期间，可以锁定进程的物理页面，防止它被交换到交换设备上去。第二点是 DMA控制器无法访问整个物理内存， DMA通道的地址寄存器代表DMA地址的前16位，而后面8位取自于页面登记表。这说明 DMA请求被限制在下面16M的存储器中。

由于DMA通道只有7个，而且它们不能被设备驱动程序共享，所以这些通道是稀缺的系统资源。就像中断一样，设备驱动程序必须能确定出它们正使用的 DMA通道。某些设备使用固定的DMA通道，如软盘设备永远使用 DMA通道2。有时设备的DMA通道可以由跳线设定，一些以太网设备采用的就是这种技术。更加灵活的设备可以通过设定它们的 CSR来确定使用哪个DMA通道。这时设备驱动程序可以选择一个空闲的 DMA通道来使用。

Linux使用dma_chan数据结构(每个DMA通道对应一个)的向量来跟踪DMA通道的使用情况。dma_chan数据结构包括两个域：一个是指向描述 DMA通道使用者的字符串的指针 ,另一个描述DMA通道是否被占用的标志域。当你使用 cat/proc/dma命令时，屏幕上打印出来的正是dma_chan数据结构的向量。

6.2.3 存储器

设备驱动程序在使用存储器时必须要小心。作为 Linux内核的一部分，它们无法使用虚存。每次设备驱动程序运行时，可能由于收到了中断或者是由于 bottom_half例程或任务队列被调度运行了，因而当前的进程可能会改变。尽管设备驱动程序独立地执行，但它也不能依赖于正在运行的某些特定进程。像内核中的其它进程一样，设备驱动程序用数据结构来跟踪它所操纵的设备。这些数据结构作为设备驱动程序代码的一部分是静态分配的。但由于它使得内核比需要的要大，所以比较浪费。大多数设备驱动程序使用内核中未分页的存贮器来记录数据。

Linux提供了内核存储空间的分配和释放例程，以供设备驱动程序使用。尽管设备驱动程序需要的空间可能不是 2的幂次方，但内核的存储空间必须按 2的幂次方进行分配，如：它可能是128或512字节。设备驱动程序需要的存储空间字节数被扩大到最接近的一个 2的幂次方块，这种方法使内核在释放存贮空间时很容易把小的自由块组合成大的块。

在分配内核存储空间时， Linux需要做大量的额外工作。如果系统中自由空间的数量比较少，系统需要释放一些物理页，并把它写到交换设备上。一般来说， Linux会挂起请求者，把进程放在等待队列上，等待系统出现足够的物理内存。并不是所有的设备驱动程序都想按此处理，所以内核存储空间分配例程在无法立即分配内存时，可以返回失败信息。如果设备驱动程序要对分配的内存空间进行 DMA操作，就可以指出该存贮空间是可 DMA操作的。这正是Linux内核而不是设备驱动程序来获知系统中哪些内存是可 DMA操作的一种方式。

6.2.4 设备驱动程序与内核的接口

Linux内核可以按照一种标准的方式和驱动程序进行交互。每类设备驱动程序 —— 字符设备、块设备、网络设备，都为内核在使用它们的服务时提供相同的使用接口。这些相同的接口使得内核可以对完全不同的设备和它们的驱动程序按照完全相同的方式进行处理，如对SCSI硬盘和IDE硬盘这两个不同的设备来说， Linux内核对它们使用完全相同的接口。

Linux具有很高的动态性，每次 Linux内核启动时，会遇到不同的物理设备，需要不同的设备驱动程序。Linux允许在编译内核时通过配置脚本把设备驱动程序加入到内核中，而这些驱动程序在启动初始化时，允许找不到要控制的硬件。其它的驱动程序可以在需要时作为内核的模块被载入。为了实现设备驱动程序的动态性，设备驱动程序在初始化时要向内核注册。 Linux维护一个设备驱动程序的表，并把它作为与驱动程序接口的一部分。这些表包括支持该类设备接口的例程和其它信息。

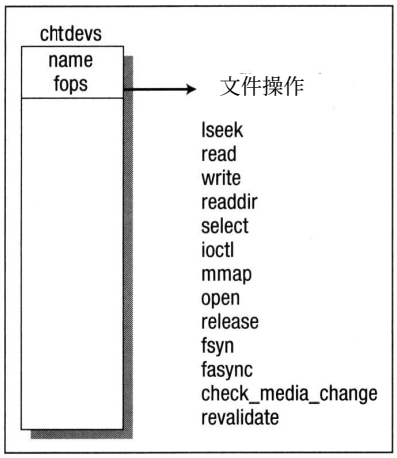


图1-6-3 字符设备

1. 字符设备

字符设备是一种可以像文件一样进行访问的简单的Linux设备(见图1-6-3)。应用程序可以像对待文件一样对待这类设备，用标准的系统调用打开、读、写、关闭它们；即使这类设备可能是 PPP守护程序使用的、用于把Linux系统连接到网络上的modem。但对该设备的这些操作仍然是完全可以的，字符设备的驱动程序通过在 device_struct数据结构的chrdevs向量中增加一项的方法来向Linux内核注册自己。在注册过程中，驱动程序初始化字符设备，该设备的主设备号 (对于tty设备为4)是chrdevs向量的索引值，因此每个设备的主版本号是固定的。在 chrdevs向量的每一项中，device_struct数据结构包括两个部分：一个是指向注册设备驱动程序名字的指针；另一个是指文件操作块的指针。文件操作块中有字符设备驱动程序的处理例程的地址，这些处理例程是用于处理像打开、读、写、关闭这类专门的文件操作。 /proc/devices中关于字符设备的内容是从chrdevs向量中获取的。

在打开一个代表字符设备的特殊设备文件时，内核要做一些建立工作，以使得系统能调用正确的字符设备驱动程序的文件操作例程。像普通的文件和目录一样，每个特殊设备文件由一个VFS inode节点来表示。每个特殊字符设备文件的 VFS inode节点，实际上对所有的特殊设备文件是通用的。它包含设备的主标识和次标识。该 VFS inode节点由下层文件系统创建，在查找特殊设备文件名时，系统会从真正的文件系统中读取关于该设备的信息。

每个VFS inode节点都是与一组文件操作相关联，但这些操作取决于该 inode节点代表的文件系统中的对象。一旦一个代表特殊字符设备文件的 VFS inode节点被建立起来，则它的文件操作被置成字符设备的缺省操作 —— 只有一个打开文件的操作。应用程序打开特殊字符设备文件时，通用的打开文件操作把设备的主标识作为 chrdevs向量的索引，取出该设备的文件操作块。同时它还会为该特殊设备文件建立文件数据结构，把文件数据结构的文件操作指针指向设备驱动程序的文件操作例程。至此所有的应用程序的文件操作都被映射到对字符设备文件操作例程的调用上去了。

2. 块设备

块设备也支持文件操作，为打开的特殊块设备文件提供的对应的文件操作机制与字符设备十分相似。 Linux在blkdevs向量中维护着注册的块设备集，而 blkdevs向量像chrdevs向量一样，由设备的主设备号进行索引。它的每个表项仍然是 device_struct结构，但这些结构是属于块设备的。 SCSI设备是块设备中的一类，而 IDE设备是另一类。正是这些类向 Linux内

核注册自己，并对内核提供文件操作。每一类块设备的驱动程序都为该类提供专门的接口。例如SCSI设备为SCSI子系统提供接口，而 SCSI子系统用该设备提供的文件操作为内核服务。

和正常的文件操作接口一样，每个块设备驱动程序要为缓冲区缓存机制提供接口。每个块设备要填写blk_dev_struct数据结构的blk_dev向量中的对应表项，对该向量的索引仍然是该设备的主设备号。blk_dev_struct数据结构包括请求例程的地址和指向 request数据结构表的指针。每个request数据结构对应于一个缓冲区缓存对该设备读 /写数据块的请求。

每当缓冲区缓存机制要从注册的设备读一块数据或写入一块数据时，它都会在blk_dev_struct中加入一个请求数据结构。图 1-6-4表明每个读写一块数据的请求都有一个指向一个或多个buffer_head数据结构的指针。buffer_head数据结构是由缓冲区缓存锁定的，系统中有等待该块操作完成的进程。每个请求结构是从名为 all_requests的静态表中分配出来的。如果有请求被加到一个空请求表中，该驱动程序的请求函数就会被调用，开始处理请求队列；否则的话，驱动程序会处理请求表中的每个请求。

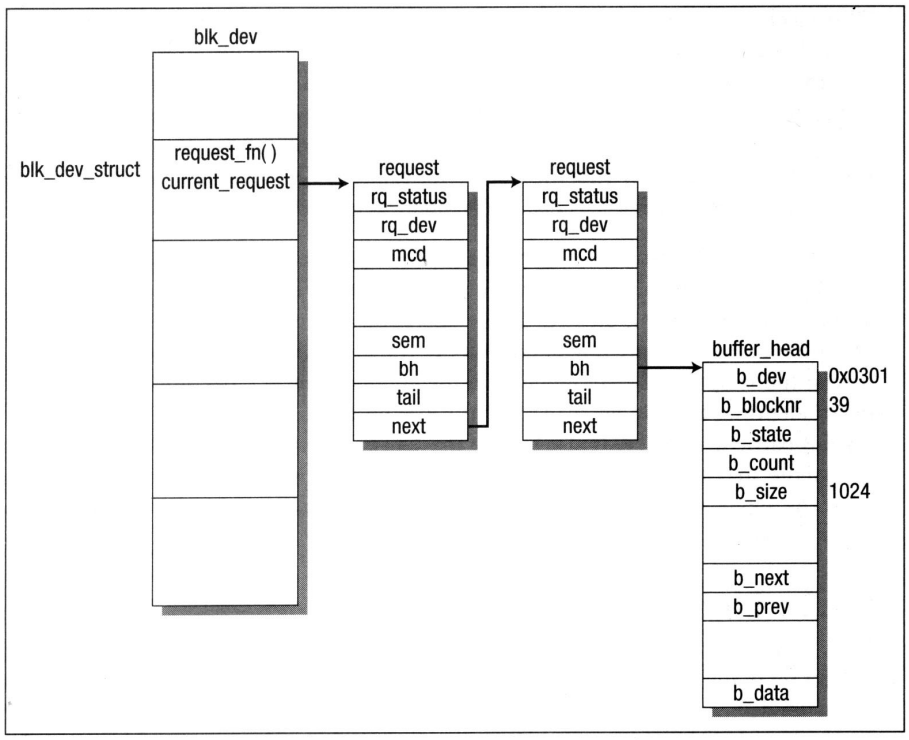


图1-6-4 缓冲区缓冲块设备的请求

一旦设备驱动程序完成了请求，它从该 request结构中删除所有的 buffer_head数据结构，把它们标志为更新状态并解锁。对 buffer_head的解锁会唤醒“睡眠”在等待块操作完成队列上的所有进程。上面过程的一个例子是文件名的解析过程， EXT2文件系统要从包含该文件系统的块设备上读取包含下一级 EXT2目录项的数据块，进程会“睡眠”在包含该目录项的buffer_head结构上，等待设备驱动程序将其唤醒。完成请求后， request被标志为空闲状态，以便于其它块请求可以使用它。

6.2.5 硬盘

硬盘通过把数据记录在盘片上，提供了一种更持久的信息保存方法。写数据时，一个小磁头磁化盘面上的微小质点，接着一个读磁头读出刚写入的数据，以确定某个微小质点是否被正确的磁化。

磁盘驱动器包括一个或多个盘片，每个盘片由光滑的玻璃或复合陶瓷组成，外表覆盖了一层氧化钢。盘片的中央都连接在一个轴上并按恒定的速度转动。转动速度根据硬盘类型的不同从每分钟3000转到每分钟10000转不等。而软盘驱动器的转速只有 360RPM(转/分钟)。硬盘的读写头用于读写数据，每个盘片有一对读写头，一面有一个头。读写头实际上并不接触盘面，它们漂浮在非常薄的 (大约14万分之一英寸)的气垫上。所有的读写头是连接在一起的，在马达的带动下，一起在盘片上移动。

每个盘片的表面被化分成狭窄的同心圆——磁道。 0磁道是最外面的磁道而编号最高的磁道最靠近中轴。柱面指具有相同磁道号的所有磁盘的集合。因此磁盘中所有盘片的所有盘面的第5磁道被称为第5柱面。由于柱面数与磁道数相同，因此你经常可以看到用柱面表示的磁盘结构。每个磁道被划分为扇区，扇区是硬盘读写的最小数据单元，它正好是磁盘块的大小，一般扇区的大小是512字节，扇区的大小通常是在硬盘生产过程中设定的。

硬盘通常由盘面号、头号和扇区号来表示。例如在启动时 Linux把一个IDE硬盘描述为：

```
hdb: Conner Peripherals 540MB - CFS540A, 516MB w/64kB Cache, CHS=1050/16/63
```

这表示它有1050个柱面，16个头，每道有63个扇区。如果扇区大小为512字节，整个盘的存贮容量为529200字节，与磁盘标明的516M容量不符，原因是有些扇区用于存放磁盘分区信息。部分硬盘可以自动地查找坏扇区，并重新建立硬盘索引以跳过这些部分。

硬盘可以进一步地被分成硬盘分区。一个分区指一大组用于某种特殊目的的扇区。对硬盘进行分区使得硬盘能够被多个操作系统使用。大多数的 Linux系统的一个硬盘，有三个分区：一个是DOS文件系统分区；一个是EXT2文件系统分区；最后一个是交换分区。硬盘的分区由分区表来描述，分区表中的每一项用头、扇区、柱面的形式标出分区的开始、终止位置。对DOS格式的硬盘来说，它可以有4个主磁盘分区，由fdisk命令来划分。但在分区表中并不是所有的四个项都是可用的。 fdisk只支持三种类型的分区 —— 主分区、扩展分区、逻辑分区。扩展分区不是真正的分区，它可以包含任意多个逻辑分区。扩展分区加逻辑分区是一种避免四个主分区限制的方式。接下来是 fdisk对一个包含两个主分区的硬盘的输出信息。

```
Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders
Units = cylinders of 2048 * 512 bytes

   Device Boot    Begin     Start       End    Blocks   Id  System
/dev/sda1             1         1       478    489456   83  Linux native
/dev/sda2           479       479       510     32768   82  Linux swap

Expert command (m for help): p

Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders

Nr AF  Hd Sec  Cyl  Hd Sec  Cyl    Start     Size ID
```

```
1 00    1    1     0  63  32  477      32  978912 83
2 00    0    1   478  63  32  509  978944   65536 82
3 00    0    0     0   0   0    0       0       0 00
4 00    0    0     0   0   0    0       0       0 00
```

这表明第一个分区从0柱面、1头、1扇区开始，一直延伸到包括477柱面、32扇区、63头在内的所有扇区。由于一个磁道有32个扇区，该硬盘有64个读/写头，所以这个分区是柱面大小的整数。fdisk缺省方式是按照柱面边界对齐分区的。所以该分区从最外面的0柱区向内扩展到478柱面，第二个交换分区从478柱面开始扩展到硬盘的最里面的柱面。

初始化过程中，Linux把硬盘的拓扑结构映射到系统中，它可以确定系统有多少个硬盘以及硬盘的类型。另外Linux还可以确定每个硬盘的分区数，这些信息是由gendisk_head表指针指向的gendisk数据结构表来表示的。在对像IDE这样的硬盘子系统初始化时，Linux为每个找到的硬盘产生一个代表该硬盘的gendisk数据结构。同时它会注册文件操作并在blk_dev数据结构中加入表项。每个gendisk数据结构有一个唯一的全设备号，并与该特殊块设备的主设备号一致。例如：SCSI硬盘子系统会建立一个gendisk表项(“sd”)，主版本号为8。这与所有SCSI硬盘设备的主版本号一致。图1-6-5给出了两个gendisk表项，第一个是SCSI硬盘子系统的，第二个是IDE硬盘子系统的，第二个表项的名称是“ide0”，指主IDE控制器。

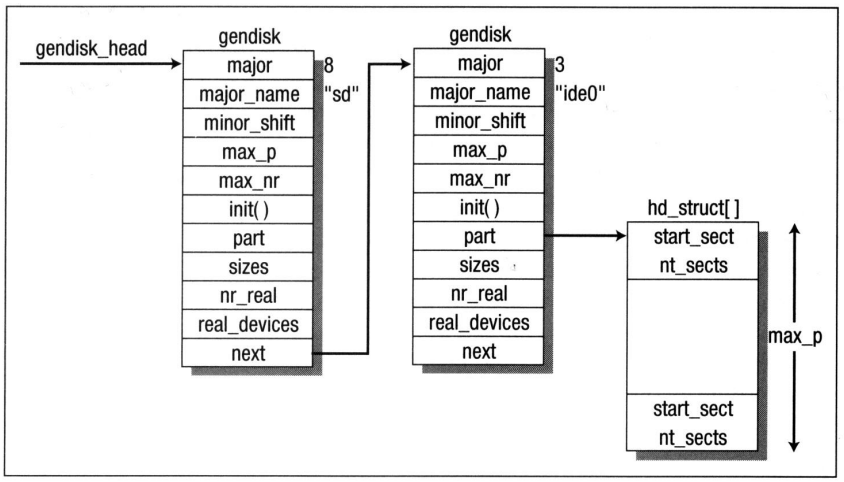


图1-6-5 硬盘表链

尽管硬盘子系统在初始化时建立gendisk表项，但它们只是在分区检查时才被Linux使用。每个硬盘子系统维护它自己的数据结构，从而允许其用主设备号和从设备号来映射到物理盘的某个分区上。无论块设备是通过缓冲区缓存机制还是文件操作进行读写，内核都会使用在特殊块设备文件中的主设备号把操作映射到相应的设备上。接着由每个设备驱动程序或内核子系统再把次设备号映射到实际的物理设备上。

1. IDE硬盘

现在Linux系统使用的大多数硬盘是IDE硬盘(Integrated Disk Eletronic)。IDE是一种磁盘接口，而不是像SCSI一样是一种I/O总线。每个IDE控制器可以最多支持2个硬盘：一个是主盘，另一个是从盘，硬盘的主从功能通常是由硬盘上的跳线来设定的。系统中的第一个IDE控制器被称为主IDE控制器，第二个被称为从IDE控制器。IDE可以管理大约3.3Mbps的数据传输量，

最大的IDE磁盘大小是538M字节，增强型IDE接口(EIDE)可以最大支持8.6G字节的硬盘，数据传输率增加到16.6Mbps，由于IDE和EIDE硬盘比SCSI硬盘便宜，所以大多数的现代PC都在主板上集成了一个或多个IDE控制器。

Linux按照硬盘所在的IDE控制器的次序来命名IDE硬盘，主控制器上主盘的名字是/dev/hda，从盘是/dev/hdb。/dev/hdc是从IDE控制器上的主盘。IDE子系统向Linux内核注册IDE控制器而不是硬盘。主IDE控制器的主标识是3而从IDE控制器的主标识是22。这表示如果一个系统有两个IDE控制器的话，IDE子系统会在blk_dev和blkdevs向量的索引值为3和为22的两个不同位置有两个不同的表项。IDE硬盘的特殊块设备文件能反映出这种编号方式，硬盘/dev/hda和/dev/hdb都连接在主IDE控制器上，因而它们的主标识为3。由于内核使用主标识作为索引值，所以任何对这些特殊块设备文件上的IDE子系统的文件或缓冲区缓存操作都会被定向到IDE子系统去。当有请求产生时，由IDE子系统计算出该请求是对哪个IDE磁盘的。为了计算出上述信息，IDE子系统使用特殊设备标识中的次设备号信息。在次设备号中包含能使IDE子系统把请求定向到正确的硬盘的正确分区的信息。/dev/hdb —— 主IDE控制器的从IDE硬盘的设备标识是(3, 64)，而该盘的第一个分区的设备标识是(3, 65)。

2. 初始化IDE子系统

IDE硬盘一直环绕着大部分IBM PC的历史。现在对这些设备的接口已经改变了，但这只是使得IDE子系统的初始化工作比以前更加复杂了。

Linux可以支持的最大IDE控制器的数目是4个，每个控制器由ide_hwifs向量中的一个ide_hwif_t数据结构来表示。每个ide_hwif_t数据结构包含两个ide_drive_t数据结构，分别用于支持主从IDE驱动器。在IDE子系统初始化时，Linux在系统CMOS存储器中查找当前硬盘的信息。CMOS存储器是由电池供电的，即使在PC关机时也不会丢失信息。CMOS存储器的位置是由系统的BIOS指定的，它可以通知Linux当前系统中找到的IDE控制器和驱动器。Linux从BIOS中读取硬盘的结构信息，并使用这些信息来建立该驱动器的ide_hwif_t数据结构。大多数的现代PC使用像Intel的82430 VX那样的芯片组，它们都包含PCI的EIDE控制器，IDE子系统使用PCI BIOS的回调功能来定位系统中的PCI(E)IDE控制器，然后它为这些芯片组调用PCI的特殊询问例程。

一旦发现了一个IDE接口或控制器，它的ide_hwif_t数据结构就被建立起来，以反映控制器及其相连的硬盘状态。在操作过程中，IDE驱动程序会向I/O存贮空间的IDE命令寄存器写入命令。主IDE控制器的控制和状态寄存器的缺省I/O地址是0xF0到0xF7。这些地址是遵照早期的IBM PC的习惯而设定的，IDE驱动程序会向Linux的缓冲区缓存和VFS文件系统注册所有的控制器，并把它们分别加到blk_dev和blkdevs中去。IDE驱动程序也会请求相应的中断控制，由于遵循IBM PC的习惯，主IDE控制器的中断号为14，从IDE控制器的中断号是15。但它们也像IDE的所有参数一样可以由内核的命令行选项来配置的，IDE驱动程序会在gendisk列表中为启动时找到的每个IDE控制器增加一个gendisk表项。这个表接着会被用于查找启动时发现的所有硬盘的分区表信息。分区表检查程序知道每个IDE控制器可以连接两个IDE硬盘。

3. SCSI硬盘

SCSI(小型计算机系统接口)总线是一种高效的对等数据总线，每条总线最多支撑8个设备(包括一个或多个主动设备)。总线上的每个设备都有一个通常由硬盘上的跳线设定的唯一标识。数据可以以同步或异步的方式在总线上的任何两个设备之间进行传送，在使用32位总线时数

据的最大传输率可以达到40M字节/秒。SCSI总线在设备间既可以传递数据又可以传递信息，总线上创始者与目标设备间的一个交易可以最多包括 8个不同的阶段。可以从总线的 5个信号中获得当前SCSI总线的阶段。这八个阶段是：

- 总线空闲 无设备控制总线，也没有交易正在进行。
- 仲裁 SCSI设备通过在地址引脚上设置它的 SCSI标识来申请获得SCSI总线的控制权。SCSI标识号最高的设备获得控制权。
- 选择 当一个设备成功地通过仲裁获得了SCSI总线的控制权后，会向这个SCSI请求的目标设备发信息，通知目标设备它要发送命令。创始者是通过在地址引脚上设置目标设备的SCSI标识来完成通知的。
- 再选择 SCSI设备可以在处理请求时断开连接，接着目标设备会重新选择创始者。但并不是所有的SCSI设备都支持这一阶段。
- 命令 创始者可能会向目标设备发送6字节、10字节或12字节的命令。
- 数据输入/输出：在这一阶段中，创始者与目标设备间正在交换数据。
- 状态 所有命令完成之后才进入这一阶段，在该阶段目标设备可以向创始者发送状态字节以显示命令成功与否。
- 消息输入/输出 这一阶段用于在创始者、目标设备间传送额外的消息。

Linux的SCSI子系统由两个基本部分组成，每一部分由一个数据结构来表示。

- 主动设备 一个SCSI主动设备是一部分物理硬件 —— SCSI控制器。NCR810 PCI SCSI控制器就是一种SCSI主动设备。如果一个Linux系统可以有同类型的一个以上的 SCSI控制器，那么每个实例都会由一个独立的 SCSI主动设备来表示。这意味着 SCSI设备驱动程序可以控制一个以上的控制器实例， SCSI设备几乎总是SCSI命令的创始者。
- 设备 最普通的SCSI设备是SCSI硬盘，但SCSI标准还支持几种设备类型：磁带、 CD-ROM以及通用SCSI设备。SCSI设备几乎总是SCSI命令的目标设备，这些设备必须按不同的方式来处理。如对 CD-ROM和磁带这种有可移动介质的设备， Linux必须检测介质是否被取出了。由于不同的磁盘类型有不同的主设备号，所以 Linux可以把不同的块设备请求定向到相应的SCSI类型的设备上去。

(1) 初始化SCSI子系统

初始化SCSI子系统十分复杂，反映出了 SCSI总线和SCSI设备的动态特征。 Linux在启动时初始化SCSI子系统，它先查找系统中的所有 SCSI控制器，然后再检测每条 SCSI控制器的SCSI总线，找出连接的所有设备。最后它初始化这些设备，使得它们通过普通文件操作和缓冲区缓存设备操作对Linux内核的其它部分是可用的。初始化过程分四个阶段：

第一阶段：Linux找出在编译内核时安装的那些用于控制硬件的 SCSI主动设备适配器或控制器，每个在内核中安装的 SCSI主动设备都在 builtsin_scsi_hosts向量中占据一个Scsi_Host_Template表项。Scsi_Host_Template数据结构包含指向执行特殊的 SCSI主动设备动作的例程的指针，这些动作包括检测有哪些 SCSI设备附着在这个SCSI主动设备上。这些例程在SCSI子系统，自我配置时被调用并且它们也是支持这种类型主动设备的 SCSI设备驱动程序的一部分。每一个被检测的 SCSI主动设备，为每个依附于它的真实的 SCSI设备，都在活动SCSI主动设备的SCSI_Host列表中增加一个scsi_Host_Template数据结构。每个被检测出来的主动类型设备的实例都由 scsi_hostlist列表中的一个Scsi_Host数据结构来表示。例如一个系统

如果有2个NCR810 PCI SCSI控制器，则它在列表中会有两个Scsi_Host表项，每个控制器一个。每个由Scsi_Host_Template指向的Scsi_Host结构都代表着它对应的设备驱动程序。

在找到了所有的SCSI主动设备之后，SCSI子系统要确定每个主设备的总线上连接着哪些SCSI设备。SCSI设备在0~7之间进行编号，每个SCSI设备在它所连接的总线上都有唯一的设备号或SCSI标识。SCSI标识通常是由设备上的跳连来设置的。SCSI初始化程序通过向SCSI总线发送TEST_UNIT_READY命令来查找该总线上的设备，一个设备作出响应时，SCSI初始程序通过向它发送ENQUIRY命令获得它的标识。从标识中Linux可以获得厂商名、设备模式和修正名。SCSI命令由Scsi_Cmnd数据结构表示，通过调用该SCSI主动设备的Scsi_Host_Template数据结构中的设备驱动程序，例程可以把标识传送给该主动设备的设备驱动程序。每个找到的SCSI设备由一个Scsi_Device数据结构表示，每个结构中都包含有指向父结点 Scsi _Host结构的指针，所有的Scsi_Device数据结构都被加入到scsi_devices表中。图1-6-6给出了主要的结构间的关系。

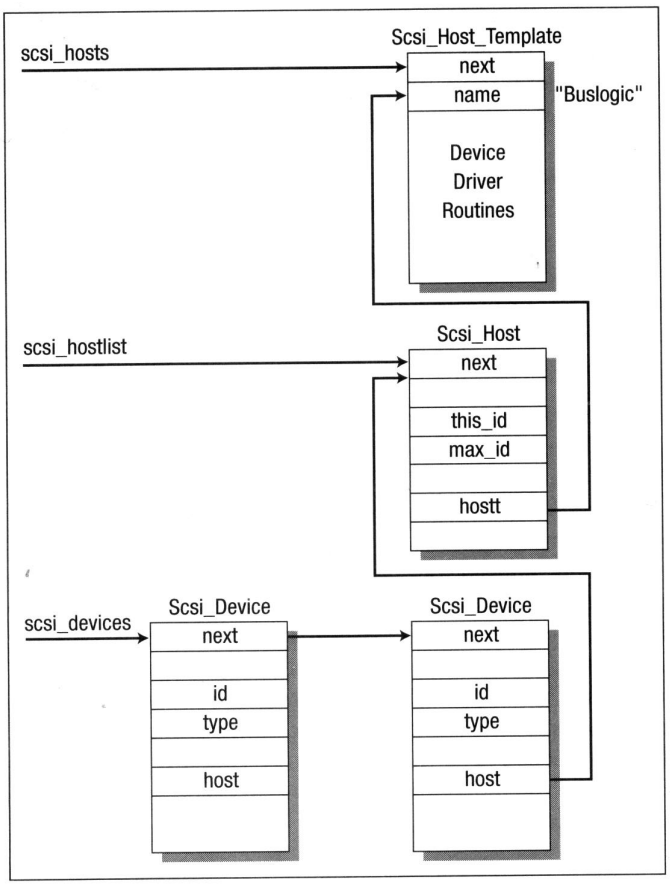


图1-6-6 SCSI数据结构

总共有4种SCSI设备类型：磁盘、磁带、CD-ROM和通用SCSI设备，每种SCSI类型都在内核中以不同的主块设备类型进行注册。然而如果系统中有一种或多种 SCSI设备类型，那么每种设备只会注册它们自己。每种 SCSI类型都维护它们自己的设备表，它使用这些表格把内核的块操作定向到正确的 SCSI设备驱动程序或 SCSI主动设备上。每种 SCSI设备类型由一个

Scsi_Device_Template数据结构来表示。Scsi_Device_Template数据结构包含对应的SCSI设备类型的信息以及用于执行各种操作的例程的地址。 SCSI子系统使用这些模板为每种类型的SCSI设备调用相应的例程，换句话说就是，如果SCSI子系统要连接一个SCSI硬盘设备，它会调用SCSI硬盘类型的连接例程。如果有一个或多个某种类型的 SCSI设备被检测出来了，那么该种类型对应的Scsi_Type_Template数据就会被加入到scsi_devicelist表中。

SCSI子系统初始化的最后阶段是为每个注册的 Scsi_Device_Template调用结束函数。对SCSI硬盘类型来说，它会使所有找到的 SCSI硬盘旋转起来，记录它们的磁盘参数；它还会把代表所有SCSI硬盘的gendisk数据结构加到图1-6-5中给出的硬盘链表中。

(2) 传送块设备请求

一旦Linux初始化了SCSI子系统，SCSI设备就可以使用了。每个活动的 SCSI设备类型都向Linux内核注册自己，以便Linux能够把块设备请求定向给它。这些请求包括通过 blk_dev结构的缓冲区缓存请求和通过 blkdevs的文件操作请求。用有一个以上 EXT2文件系统分区的SCSI硬盘的驱动程序作为例子，让我们看一下当安装了多个 EXT2分区时，内核缓冲区请求是如何被定向到正确的SCSI硬盘上的。

每个从SCSI硬盘分区中读一块数据或写入一块数据的请求都会使 blk_dev向量中SCSI硬盘的current_request列表增加一个新的请求数据结构。如果请求表正在被处理的话，缓冲区缓存机制不需要做其它的工作；否则，它要通知 SCSI硬盘子系统处理请求队列。系统中的每个SCSI硬盘由一个Scsi_Disk数据结构来表示，这些数据结构被保存在 rscsi_disks向量中，使用SCSI硬盘分区次设备号的一部分来进行索引。例如： /dev/sdb1的主设备号是8，次设备号为17；因此它对rscsi_disks向量的索引值为1。每个Scsi_Disk数据结构都有一个指向代表该设备的Scsi_Device数据结构的指针，而 Scsi_Device反过来又包含有指向它所在的主动设备的Scsi_Host数据结构的指针。从缓冲区缓存来的 request数据结构先被转化成要发送给 SCSI设备的SCSI命令的Scsi_Cmd数据结构，然后这些Scsi_Cmd结构被加入到代表该设备的Scsi_Host结构的队列中。每个SCSI驱动程序一旦读/写了适当数据块后就会处理这些命令。

6.2.6 网络设备

网络设备是与Linux网络子系统相关的，用于发送/接收数据报文的实体。它通常是像以太网卡那样的物理设备，但也可能是软件；如用于向自己发送报文的回送 (loopback)设备。每个网络设备由一个device数据结构来表示，在内核启动、网络初始化时，网络设备驱动程序向Linux注册它们所控制的设备， device数据结构中包括该设备的信息以及允许各种支持的网络协议使用设备服务的函数集合的地址。这些函数多半是与使用网络设备传输数据相关的。设备使用标准的网络支持机制，把接收到的数据上传到适宜的协议层中去。所有的网络数据 (报文)的发送和接收都是由sk_buff数据结构来表示，这些数据结构非常灵活，允许网络协议头可以很容易地被加入或删除。网络协议层如何使用网络服务以及它们是如何使用 sk_buff数据结构正向和反向传送数据的？这些都在第8章有详细的介绍。本部分主要考虑的是 device数据结构以及网络设备是如何被发现和初始化的。

device数据结构包含下列关于网络设备的信息：

- 名称 不像块设备和字符设备那样可以通过 mknod命令建立它们的特殊设备文件，网络设备的特殊设备文件是在系统的网络设备检测和初始化时建立起来的。网络设备的名称

是标准的，每类名称代表一种设备类型。同一类型的多个设备以 0开始向上编号，所以以太网设备可以被称为 /dev/eth0、/dev/eth1、/dev/eth2等等。一些通用的网络设备名如下所示：

/dev/ethN	以太网设备
/dev/slN	SLIP设备
/dev/pppN	PPP设备
/dev/lo	Loopback设备

- 总线信息　是设备驱动程序用于控制设备的信息。 irq号是该设备正在使用的中断号。而基地址(base address)是设备的控制和状态寄存器在 I/O空间的起始地址， DMA通道(channel)是网络设备正在使用的 DMA通道号。所有的这些信息都是在系统启动时设备初始化过程中设定的。
- 接口标志　它们描述了网络设备的特性和能力：

IFF_UP	接口正在运行
IFF_BROADCAST	在设备中的广播地址是有效的
IFF_DEBUG	打开设备调试
IFF_LOOPBACK	这是一个回送设备
IFF_POINTTOPOINT	这是点到点链接(SLIP和PPP)
IFF_NOTRAILERS	不允许网络追踪
IFF_RUNNING	分配资源
IFF_NONRP	不支持APP协议
IFF_PROMISC	设备处于混杂接收方式，它能接受所有的报文，而不考虑报文的目的地址
IFF_ALLMULTI	接收所有的IP多路广播帧
IFF_MULTICAST	可以接收IP的多路广播帧

- 协议信息　设备用于描述如何支持网络协议层的信息。
- mtu　不包括要加的所有链路层头时，网络可以传输的最大报文大小。该值由 IP等协议层使用，用于选择发送报文的大小。
- 家族　家族用于表示设备能够支持的协议家族。所有 Linux网络设备的家族是 AF_INET —— Intenet地址家族。
- 类型　硬件接口类型描述该网络设备连接的介质类型。 Linux网络设备支持很多种不同的介质类型，包括：以太网、 X.25、令牌环、 SLIP、PPP和Apple Localtalk。
- 地址　device数据结构包含很多与该网络设备有关的地址，如该设备的 IP地址等等。
- 报文队列　它是等待由该网络设备发送的由 sk_buff结构表示的报文的队列。
- 支持函数组　每个设备都有一组标准的例程，作为设备链路层接口的一部分可以由协议层来调用。它们包括建立数据帧、传送数据帧的例程以及增加标准帧头和收集统计信息等例程。统计信息可以由 ifconfig命令看到。

初始化网络设备

网络设备驱动程序可以像其它的 Linux设备驱动程序那样，安装在 Linux内核中。每种可能的网络设备都由 dev_bast表指针指向的网络设备表中的 device数据结构来表示。如果网络层

需要设备执行某些特殊工作的话，它可以调用记录在 device数据结构中的某一个网络设备服务例程。但是每个device数据结构最开始只包括初始化或探测例程的地址。

网络设备驱动程序需要解决两个问题：第一个问题是并不是安装在内核中的所有网络设备驱动程序都能找到要控制的设备。第二个问题是无论以太网的下层设备驱动程序是什么，它们在系统中的名称总是/dev/eth0、/dev/eth1……。网络设备“缺少”的问题很容易解决，在调用每个网络设备的初始化例程时，它会返回一个状态码显示它是否找到其驱动的控制器实例。如果驱动程序没找到任何设备，那么它在由 dev_base指向的device列表中的表项就被删除了。如果驱动程序找到了设备，它会用该设备的信息填充 device数据结构的其余部分，并在其中写入网络设备驱动程序的支持函数的地址。

对第二个动态地将以太网设备分配给标准的特殊设备文件 /dev/ethN的问题，Linux用一种更高明的办法解决了。在设备表中共有 8个标准表项，第一个对应 eth0，第二个对应eth1。并以此类推。8个表项中的初始化例程是完全相同的。 Linux轮询安装在内核中的每个以太网设备驱动程序，直到有一个驱动程序找到了它控制的设备。一旦一个驱动程序找到了它的以太网设备，就会填充它现在占用的 ethN device数据结构。并且在网络设备初始化时，驱动程序会初始化它控制的物理硬件，找出设备使用的中断号、 DMA通道号等信息。如果设备驱动程序找到了它控制的网络设备的多个实例的话，就会占用多个 /dev/ethN device数据结构。一旦8个标准/dev/ethN特殊设备文件都被分配了， Linux就不会再检测其它的以太网设备了。

第7章 文件系统

本章介绍Linux内核是如何维护它支持的文件系统中的文件的，我们先介绍 VFS(Virtual File System，虚拟文件系统)，再解释一下Linux内核的真实文件系统是如何得到支持的。

Linux的一个最重要特点就是它支持许多不同的文件系统。这使 Linux非常灵活，能够与许多其他的操作系统共存。在写这本书的时候， Linux共支持15种文件系统： ext、 ext2、xia、 minix、 umsdos、 msdos 、vfat、 proc、 smb、 ncp、 iso9660、 sysv、 hpfs、 affs 和ufs。无疑随着时间的推移， Linux支持的文件系统数还会增加。

Linux像UNIX一样，系统可用的独立文件系统不是通过设备标识来访问的，而是把它们链接到一个单独的树形层次结构中。该树形层次结构把文件系统表示成一个整个的独立实体。Linux以装配的形式把每个新的文件系统加入到这个单独的文件系统树中。无论什么类型的文件系统，都被装配到某个目录上，由被装配的文件系统的文件覆盖该目录原有的内容。该个目录被称为装配目录或装配点。在文件系统卸载时，装配目录中原有的文件才会显露出来。

在硬盘初始化时，硬盘上的分区结构把物理硬盘化分成若干个逻辑分区。每个分区可以包含一个像EXT2那样的一个独立的文件系统。文件系统用目录将文件按照逻辑层次结构组织起来。目录是记录在物理设备块中的软链接信息。包含文件系统的设备被称为块设备。

IDE硬盘分区/dev/hda1 —— 系统中第一个IDE硬盘驱动器的第一个分区，就是一个块设备。Linux文件系统把这些块设备当作简单的线性块的集合，它不知道也不在意下层物理硬盘的实际结构。每个块设备驱动程序的任务就是把系统读该设备上某一块的请求映射成对本设备有意义的术语，如该块所在的磁道、扇区或柱面号。无论文件系统存在在何种设备上，它都可以按相同的方式进行操作。而且在使用文件系统时，由不同的硬件控制器控制的不同物理介质上的不同文件系统对系统用户是透明的。文件系统既可能在本地系统上的，也可能是通过网络链接装配的远程文件系统。请看下面根文件系统位于 SCSI硬盘上的Linux系统示例：

```
A       E       boot    etc     lib     opt     tmp     usr
C       F       cdrom   fd      proc    root    var     sbin
D       bin     dev     home    mnt     lost+found
```

对文件进行操作的用户和程序都不需要知道 /C实际上是系统中第一个IDE硬盘上的一个装配了的VFAT文件系统，而/E是指从IDE控制器的主IDE硬盘。即使第一个IDE控制器是PCI控制器，而第二个是同时还控制 IDE 接口的CDROM的ISA控制器，这也不会给系统造成任何不便。我可以使用 PPP网络协议和 modem拨号到我所在公司的局域网上。这时，可以把 Alpha AXP Linux系统中的文件系统远程装配到 /mnt/remote目录上。

文件是数据的集合，如：本章的源文本文件就是一个叫做 filesystems.tex的ASCII码的文件。一个文件系统不仅包括该文件系统中所有文件的数据，还包括文件系统的结构信息。它记录下所有Linux用户和进程当作文件看待的信息、目录软链接信息以及文件保护信息等等。而且文件系统必需保证这些信息的安全性，因为操作系统的基本完整性就取决于它的文件系统。没有人会使用一个随机的丢失数据和文件的操作系统。

Linux支持的第一个文件系统是MINIX文件系统，它对用户有很多限制并且性能比较差。MINIX文件系统的文件名不能超过14个字符，最大文件长度是64M字节。初看起来64M字节好像足够大了，但现代的数据库系统需要更大的文件长度。第一个专门为Linux设计的文件系统是EXT(扩展文件系统)，在1992年4月设计完成并为Linux解决了大量的问题。但它在性能上仍有所欠缺。所以在1993年设计了第二个扩展文件系统——EXT2。本章主要介绍的就是这个文件系统。

在EXT文件系统加入到Linux中时，Linux系统发生了一个重大的发展。真实文件系统从操作系统中分离出来，而由一个接口层提供的真实文件系统的系统服务被称为虚拟文件系统(VFS)。VFS使得Linux可以支持许多种不同的文件系统，而这些文件系统都向VFS提供相同的软件接口。由于所有的Linux文件系统的细节都是由软件进行转换的，所有对Linux系统的其余部分和在系统中运行的程序来说这些文件系统是完全相同的。Linux的虚拟文件系统层使得你可以同时透明地装配很多不同的文件系统。

实现Linux虚拟文件系统要使得它对文件的访问要尽可能地快、尽可能地高效，而且一定要确保文件和数据的正确性。这两个要求彼此是不对称的。在每个文件系统被装配使用后，Linux的VFS会在内存中缓存来自于这些文件系统的信息。因此由于对文件或目录的创建、写、删除操作而改变了Linux缓存中的数据时，对文件系统的更新操作要格外小心。如果你能在运行的内核中看到文件系统的数据结构，就会看到那些文件系统读/写的数据块。代表被访问的文件和目录的数据结构可能被创建或删除，而设备驱动程序不停地工作，读取数据、保存数据。这些缓存中最重要的一个是缓冲区缓存，它把独立文件系统访问下层块设备的方法集成起来。每个被访问的块都被放到缓冲区缓存中，并根据它们的状态放在相应的队列中。缓冲区缓存不仅缓存数据缓冲区，它还有助于管理块设备驱动程序的异步接口。

7.1 第二个扩展文件系统EXT2

EXT2是为Linux设计的一个可扩展的高性能文件系统。它是目前为止Linux中最成功的文件系统，也是当前商业销售的Linux的基础。EXT2与其他的许多文件系统一样都是建立在如下前提的基础上：文件中所有数据都记录在数据块中而且这些数据块的长度都是相同的。但由于在创建EXT2文件系统时可以设定块的大小，所以不同的EXT2文件系统块的长度可能不同。文件长度与块长度的整数倍对齐。如果块的长度是1024字节，那么一个1025字节的文件会占用两个1024字节块。不幸的是这表示平均起来每个文件就要浪费半个块。在计算机领域，你通常要在CPU利用率和存储空间磁盘空间利用率之间作出折衷。这时Linux像大多数的操作系统一样，用降低磁盘利用率的办法来降低CPU的工作负荷。文件系统中并不是所有的块都有数据，有些块是用来记录描述文件系统结构信息的。EXT2文件系统通过用inode数据结构来表示系统中每个文件的方法来定义文件系统中的拓扑结构。inode结构包括文件占用了哪些数据块、文件访问权限、文件的更改时间、文件类型等信息。EXT2文件系统的每个文件都由一个inode来表示，而每个inode节点在系统中都有一个唯一标识号。文件系统的所有inode节点都被记录在inode表中。EXT2目录只是一种特殊的文件，它包含指向目录中所有对象的inode节点的指针。

图1-7-1给出了占用块设备一组连续块的EXT2文件系统的结构。从文件系统的角度来说，块设备只是一组可读写的块。文件系统不必关注块是在哪个物理介质上，因为这是设备驱动

程序的工作。

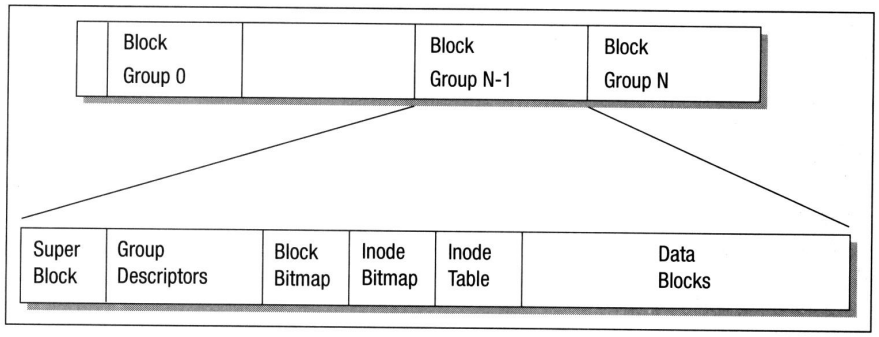


图1-7-1 EXT2文件系统的物理结构

文件系统无论是从块设备上读信息还是数据，它都会向设备驱动程序请求读取块长度的整数倍。EXT2文件系统把它所在的逻辑分区化分成块组。每组除了复制了对文件系统完整性至关重要的信息以外，还包括以数据块形式存放的物理文件、目录等信息。这种备份是必须的，因为一旦系统发生严重灾难，文件系统需要这些信息做恢复工作。下面对每个块组中的内容作详细介绍。

7.1.1 EXT2系统的inode节点

在EXT2文件系统中，inode节点是系统的基本单元。文件系统中的每个文件或目录都由一个inode节点来表示。每个块组中的所有 inode节点都被记录在 inode节点表中，系统使用位图来跟踪inode节点的分配情况。图1-7-2给出了EXT2系统的inode节点的格式。在inode所有域中，有下列几个域比较重要：

- 模式　它包含两部分信息：该inode节点代表哪种文件系统的对象、用户对它的访问许可。在EXT2系统中，一个 inode 节点可以代表文件、目录、符号链接、块设备、字符设备或FIFO管道。
- 拥有者信息　该文件或目录的拥有者的用户标识和组标识。文件系统可以根据它分配正确的访问权限。
- 长度　以字节为单位来表示的文件长度。
- 时戳　该inode节点的创建时间和它最后一次修改的时间。
- 数据块　该域包含指向本 inode节点所代表的文件或目录数据块的指针。前12个指针是直接指向物理块的指针。最后3个指针分别指向 1级到3级的索引。例如2级索引块指针指向一个指针块，而该指针块中的每个指针又指向

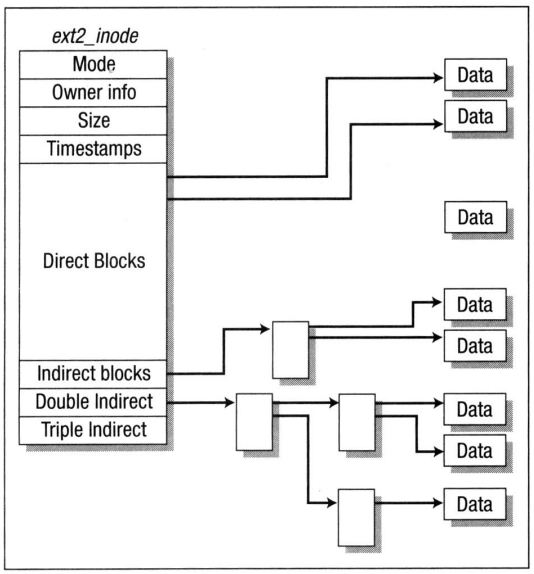


图1-7-2 EXT2的inode结点

一个直接指向数据块的指针块。这种方式使得对长度小于或等于 12个数据块文件的访问比大文件快。

EXT2系统的inode节点还可以表示特殊设备文件。它们不是实际的文件，只是程序用于访问对应的设备的句柄(handle)。在/dev目录下的所有设备文件允许程序通过它们访问 Linux的设备。例如：mount 程序用要装配的设备文件作为输入参数。

7.1.2 EXT2系统的超级块

超级块(Superblock)中包含有对本文件系统的块长度和结构的描述信息。文件系统的管理程序使用超级块中的信息来管理、维护整个文件系统。通常在装配文件系统时，系统只会读取块组0的超级块，但文件系统的每个块组中都有超级块的副本以防止文件系统崩溃。超级块中包含以下一些比较重要的域：

- MagicNumber(魔数) 该域是装配软件用于检查本超级块是否为 EXT2文件系统的超级块的。对当前版本的EXT2系统，该域的值是0xEF53。
- Revision Level 主、次Revision 域允许装配程序检测本文件系统是否支持某些只有在文件系统的某些修正版中才支持的特殊特征。超级块中还有兼容特征域。它可以协助装配程序检测哪些新特征可以在这个文件系统上安全使用。
- 装配记数和最大装配记数 这两个域一起帮助系统决定是否作完全的文件系统检查。每次装配文件系统时，装配记数值都会增 1。如果它等于最大装配记数了，系统会显示“达到最大装配记数，推荐运行 ezfsck”的警告信息。
- 块组号 存放本超级块副本的块组号。
- 块长度 以字节为单位表示的本文件系统块的长度，如 1024字节。
- 每组的块数 每组中块的数目。像块的长度一样，它是在文件系统建立时固定下来的。
- 自由块数 文件系统中空闲块的数目。
- 自由inode数 文件系统中空闲inode节点数。
- 第一个inode节点 本域中存放文件系统中第一个 inode节点的节点号。EXT2根文件系统的第一个inode节点是‘/’目录的目录项。

7.1.3 EXT2系统的组描述符

每个块组都有一个描述它的数据结构。像超级块一样，所有块组的所有组描述符在每个块组中都有副本，以防止文件系统崩溃。每个组描述符包括如下信息：

- 块位图 存放该块组(Block Group)块分配位图的块号，该域在系统分配、释放块时使用。
- Inode位图 本块组的inode分配位图的块号。该域在系统分配、释放 inode节点时使用。
- inode表 本块组的inode节点表的起始块块号。每个 inode节点由EXT2系统的inode数据结构来表示。

自由块记数、自由inode记数、使用的目录数

组描述符在块组中一个接一个存放，它们一起组成了组描述符表。每个块组在超级块的副本之后包含了整个组描述符表。 EXT2文件系统实际上只使用第一个副本 (在块组0中)。其他的副本像超级块的副本一样，用于防止主副本损坏。

7.1.4 EXT2系统的目录

在EXT2文件系统中，目录是一种特殊的文件。它用于创建、保持对文件系统中文件的访问路径。图1-7-3给出了内存中一个目录项的结构图。目录文件是一组目录项的列表，每个目录项包含下列信息：

- inode节点号　该目录项的inode节点号。它是块组的inode表中记录的inode数组的索引值。在图1-7-3中，一个叫“ file ”的文件的目录项中包含有值为i1的inode索引值。

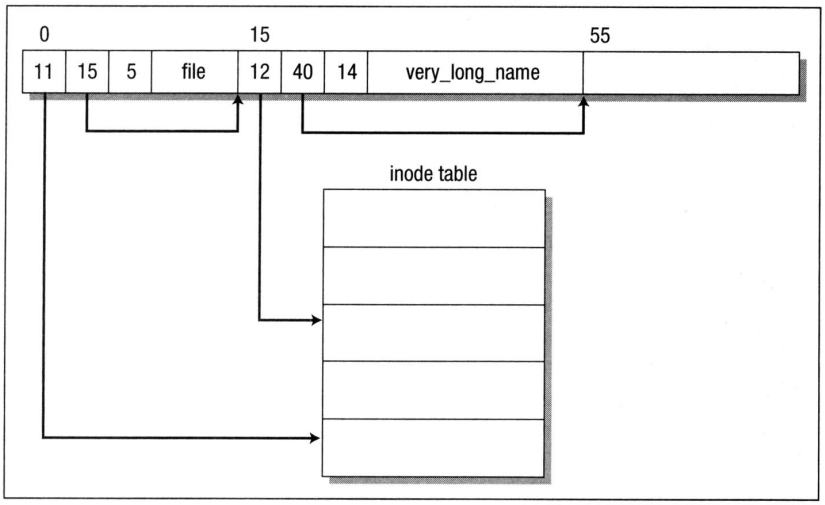


图1-7-3　EXT2系统的目录

- 名字长度　以字节为单位来表示本目录项的长度。
- 名字　本目录项的名字

每个目录的前两项总是标准的“ . ”和“ .. ”目录项，它们分别表示本目录和父目录。

7.1.5 在EXT2文件系统中查找文件

Linux的文件名与UNIX文件名有相同的格式，它是用“ / ”分隔的一组目录名，并且最后以文件名来结尾。文件名的例子是“ /home rusling/.cshrc ”，其中/home 和/rusling是目录名而文件名是.cshrc。像UNIX系统一样，Linux不注意文件名的格式问题。文件名可以任意长，可以包含任何可打印的字符，为了在 EXT2文件系统中找到代表该文件的 inode节点，系统按每次一个目录的方法解析文件名，直到最后到达要找的文件。

文件名解析时，我们需要的第一个 inode节点就是文件系统的根inode节点，在文件系统的超级块中有它的节点号。为了读 EXT2系统的inode节点，我们必须要在相应的块组的 inode表中查找该inode节点。例如：如果根inode节点号是42的话，我们需要块组0的inode表中第42个inode节点。根inode节点是EXT2系统的目录，换句话说就是，根 inode节点的模式把它描述成了一个目录，它的数据块中包含着 EXT2系统的目录项。

“ home ”只是众多目录项中的一个，但 home目录项给出了描述 /home目录的inode节点的节点号。为了查找rusling目录项，我们不得不读取 /home目录(通过先读home的inode节点，再读取由它的inode节点记录的所有包含目录项信息的数据块)。在rusling目录项中给出了描述

/home/rusling目录的inode节点的节点号。最后读出代表 /home/rusling目录的inode节点指向的目录项，在其中找出 .cshrc文件的inode节点号。至此我们获得了包含 file文件信息的数据块。

7.1.6 在EXT2文件系统中改变文件的大小

文件系统的一个共同问题是文件碎片。由于记录文件数据的块散布在文件系统的各处，数据块间离的越远对文件执行顺序的数据块访问的效率就越低。 EXT2文件系统试图通过给文件分配的新块要尽量与文件的当前数据块物理上接近或至少与它的当前数据块在同一个块组的办法来克服文件碎片问题。而只有在上面的条件都无法满足时， EXT2文件系统才会在另一个块组为文件分配数据块。

在进程向文件中写数据时，Linux文件系统要查看数据是否超过了文件的最后一个分配块。如果超过了，那么系统会为该文件再分配一个新的数据块。在分配操作完成之前，进程不能处于执行状态。在进程继续执行之前，它必须等待文件系统分配一个新的数据块，并把数据的剩余部分写入到新块中。 EXT2文件系统的块分配例程做的第一件事情就是锁定该文件系统的超级块。分配或释放块会改变超级块的某些域，而 Linux文件系统不允许一个以上进程同时操纵超级块。如果还有一个进程要分配数据块，它必须等待本进程完成。等待超级块的进程被系统挂起，在超级块的当前用户放弃控制权之前无法进入运行状态。对超级块的访问遵循先来先服务的原则，一旦一个进程取得了超级块的控制权，在完成操作之前它就一直保持对超级块的控制。进程锁定超级块之后，它先检查系统中是否有足够的空闲块。如果系统没有足够的空闲块，分配新块的操作失败，进程会放弃对文件系统超级块的控制。

系统中如果有足够的空闲块的话，系统就会为该进程分配一个。如果 EXT2文件系统支持预分配数据块，那么我们可以从预分配的数据块中直接拿一个。预分配的数据块并不是真的存在，它们只是被保存在分配块的位图中，需要分配新块的代表 file文件的VFS inode节点有两个特殊的EXT2文件系统域：prealloc_block和prealloc_count。其中prealloc_block指第一个预分配数据块的块号， prealloc_count指系统中共有多少个预分配数据块。如果系统中没有预分配的数据块或文件系统不允许块预分配机制，那么 EXT2文件系统必须要真正地分配一个新块。EXT2文件系统要先查看该文件最后一个数据块后面的数据块是否空闲。从逻辑上讲，由于这种分配方法使顺序访问更快，所以它是最高效的文件块分配方法。如果该块不是空闲的，则搜索继续进行，系统会在与理想块距离为 64个块的范围内查找一个空闲的数据块。这一块虽然不是最理想的，但至少非常接近，并与该文件的其他块处在同一块组中。

如果满足上面条件的数据块也找不到，那么进程会逐个地查找所有其他的块组，直到它找到了空闲块。块分配程序可以在块组中寻找一簇 8个连续的空闲块。若不能找到 8个连续的块，那么它也可以寻找少一些的连续空闲块。当块预分配进程被启动后，块预分配进程会相应地更新prealloc_block和prealloc_count两个域。

进程找到空闲块后，块分配程序会更新块组中的块位图，并在缓冲区缓存中为它分配一个数据缓冲区。这个数据缓冲区由文件所在文件系统的支持设备标识和新分配块的块号来唯一标识。缓存区中的数据先被清 0，然后文件系统把它标志为“ 脏 ”以表明该缓冲区中的数据还没有被回写到物理硬盘上。最后超级块解锁并被标记为“ 脏 ”来表明超级块被改变了。如果系统中有等待超级块的进程，那么等待队列中的第一个进程会进入运行状态，并获得对超级块的独占控制权。进程的数据被写入到新的数据块中，如果新块又被填满了，那么整个进

程会重复上面的块分配过程，分配下一个新块。

7.2 虚拟文件系统

图1-7-4给出了Linux内核的虚拟文件系统与它的真实文件系统之间的关系。虚拟文件系统要管理在任何时间装配的所有不同的文件系统，所以它要维护一些描述整个虚拟文件系统和真实的被装配的文件系统的数据结构。非常令人感到迷惑的是， VFS用超级块和inode节点的方式来表示系统的文件，这与 EXT2文件系统使用超级块和 inode节点的方式完全一样。与EXT2文件系统的inode节点一样， VFS的inode节点也用于描述系统中的文件、目录以及虚拟文件系统(VFS)的内容、拓扑结构。从现在开始为了避免混淆，我会使用 VFS inode节点和VFS超级块以区别于EXT2的inode节点、超级块。

在每个文件系统初始化时，它向 VFS进行注册。这个过程发生在系统启动操作系统自我初始化的过程中，真实的文件系统或者是安装在内核中的，或者是作为内核的可载入模块。文件系统模块只有在系统需要它们时才会被载入，例如 VFAT文件系统如果以内核模块的方式存在的话，它会在装配 VFAT文件系统时被载入。每当包含文件系统的块设备被装配时 (包括根文件系统)，VFS都会读入它的超级块。每种类型文件系统的超级块读例程必须要确定出整个文件系统的拓扑结构，并把这些信息映射到 VFS超级块数据结构中。VFS文件系统包含了系统中所有装配文件系统的列表和它们的 VFS超级块信息。每个VFS超级块包含执行某些特殊功能的例程的信息和指向它们的指针。例如代表装配的EXT2文件系统的超级块包含指向读 EXT2文件系统inode节点的例程。这个读例程像所有文件系统读inode节点的例程一样，会填充 VFS inode节点的对应域。每个 VFS超级块都有一个指向文件系统第一个 VFS inode节点的指针。对根文件系统来说，这个指针指向的 inode节点是用来表示“/”目录的。上面所讲的信息映射对 EXT2文件系统是非常高效的，但对其他的文件系统效率可能会降低。

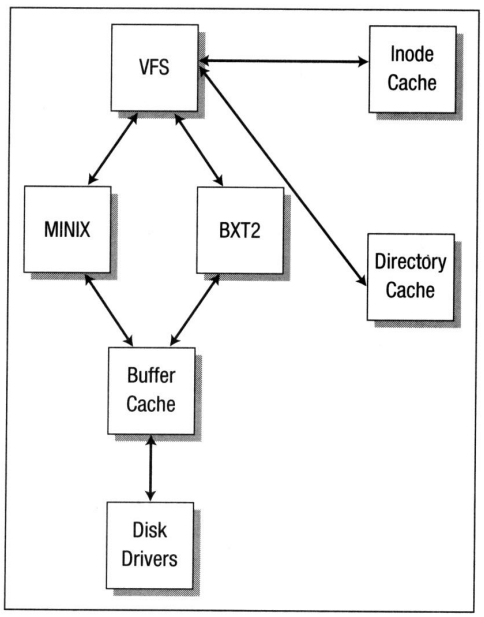


图1-7-4 VFS文件系统的逻辑结构图

在系统进程访问目录和文件时，系统会调用搜索系统中 VFS inode节点的进程。例如，对一个目录敲ls命令或对文件使用 cat命令都会使VFS文件系统搜索代表其所在文件系统的 VFS inode节点组。由于系统中的每个文件和目录都是由一个VFS inode节点表示的，所以有一些 inode节点会被经常访问。这些经常被访问的 inode节点被记录在缓存中以加速访问过程。如果要访问的 inode节点不在inode缓存中，那么系统会调用文件系统的专门例程来读入相应的 inode节点。读入inode节点的操作会使得它被加入到 inode缓存中，而对该inode节点的后续访问使得它被保存在 inode缓存中。那些访问频率较低的 VFS inode节点会被从inode缓存中删除掉。

所有的Linux文件系统使用共同的缓冲区缓存来缓存从下层物理设备来的数据，通过这种

方式来加速文件系统对它们对应的物理设备的访问。缓冲区缓区是独立于文件系统的，并被集成为Linux内核用于分配、读写数据缓冲区的一种机制。它的显著优势是使得 Linux文件系统从下层物理介质和支持物理介质的设备驱动程序中独立出来。所有的块设备向 Linux内核进行注册，并向上提供一个统一的、基于块操作的异步接口，即使像 SCSI这样复杂的块设备也支持相同的接口。在真实文件系统从下层物理硬盘读数据时，它会向控制该设备的设备驱动程序发出读物理块的请求。缓冲区缓存集成了这个块设备接口。所有由文件系统读出的块都被放入由所有的文件系统和 Linux内核共享的全局缓冲区缓存中，缓存中所有的缓冲区是由它的块号和所在设备的标识来标识的。所以，如果经常需要访问相同的数据的话，那么这些数据可以直接从缓冲区缓存中取出而不是从硬盘直接读出 (从硬盘读数据块花费的时间要长一些)。有些设备支持预读操作，这时系统推测可能使用的数据块会被设备读出，以防文件系统需要它们。

VFS文件系统还支持目录查找缓存机制，所以经常使用的目录的 inode节点可以更快地被找到。如果要做个实验的话，你可以对最近没有使用的目录作一下 ls操作。第一次ls操作时你可能会注意到一个短暂的停顿，而第二次它会立即返回结果，目录缓存存放的不是代表目录的inode节点(这些inode节点是在inode缓存中的)，而只有一些全目录名和对应索引节点的节点号的映射。

7.2.1 VFS文件系统的超级块

每个装配的文件系统由一个VFS超级块来表示， VFS超级块中主要包括下列几个域：

设备：该文件系统所在的块设备的设备标识。例如 /der/hda/——系统中第一个 IDE硬盘，它的设备标识是0X301.

inode指针： mounted inode节点指针指向文件系统中的第一个 inode节点。Covered inode节点指向代表该文件系统装配点目录的 inode节点。根文件系统的VFS超级块没有Covered指针项。

块长度：以字节为单位表示的该文件系统的块长度。如 1024字节。

超级块操作：指向该文件系统的一组超级块例程。这些例程主要由 VFS用于读写inode节点和超级块。

文件系统类型：指向装配的文件系统的 file_system_type数据结构的指针。

文件系统特征：指向该文件系统所需信息的指针。

7.2.2 VFS文件系统的inode节点

像EXT2文件系统一样， VFS文件系统的每个文件、目录等对象都是由一个 VFS inode节点表示的。每个VFS inode节点的信息都是由文件系统的专门例程从下层文件系统的信息中获得的。VFS inode节点只存在于内核的存贮空间中，只要它们对系统有用就一直被记录在 VFS inode节点的缓存中， VFS inode节点主要包括下列域：

- 设备 该VFS inode节点表示的文件系统对象所在物理设备的标识。
- inode号 inode节点的节点号，在本文件系统中是唯一的。设备和 inode号一起可以在VFS中唯一标识该VFS inode节点。
- 模式 像EXT2文件系统的这个域一样，它表示对 VFS inode节点的访问权限。

- 用户标识　拥有者标识。
- 时间　创建、更改和写的时间。
- 块长度　用字节为单位表示的本文件数据块的长度。
- inode操作　指向例程地址块的指针。这些例程是该文件系统专有的。用于操作该 inode 节点，如对该inode节点表示的文件作删减操作。
- 计数　当前使用的本VFS inode节点的系统进程数，值为0表示本inode节点可以被自由的丢弃或重用。
- 锁　本域用于锁定VFS inode节点。例如当它被文件系统读出时， VFS文件系统可以锁定它。
- 脏　表示该VFS inode节点是否被更改过。如果有改动，那么下层文件系统也要作相应的更改操作。

7.2.3 注册文件系统

在建立Linux内核时，可以选择支持的文件系统。在内核被建立后，文件系统的启动代码包含对所有安装在内核中的文件系统的初始化例程的调用。 Linux文件系统也可以作成模块的形式，它们可根据需要载入内存或用 insmod命令手工载入。

在文件系统模块被载入时，它向内核注册；在卸载时向内核注消。每个文件系统的初始化例程向VFS文件系统注册，并由一个 file_system_type数据结构来表示。该数据结构中包含文件系统的名字和指向它的 VFS超级块读例程的指针。图 1-7-5给出了在由file_systems指针指向的表中file_systm_type数据结构的格式。每个file_system_type数据结构包含下列信息：

- 超级块读例程　这个例程在文件系统的实例被装配时由 VFS文件系统调用。
- 文件系统名　该文件系统的名称：如ext2。
- 所需设备　这个文件系统需要支持设备吗？并不是所有的文件系统都要有保存它们的设备。如/proc文件系统不要求有支持它的块设备。

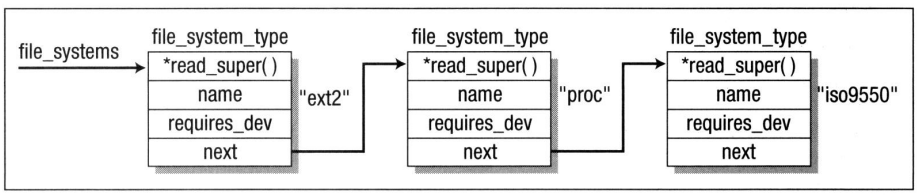


图1-7-5 注册的文件系统

通过查看/proc/tilesystems，可以看到当前注册的文件系统：

如：

```
        ext2
nodev proc
        iso9660
```

7.2.4 装配文件系统

在超级用户要装配文件系统时， Linux内核必须先验证传入系统调用中的参数的有效性。尽管mount确实做一些基本的检查，但它并不知道内核中安装了哪些文件系统以及指定的装配点是否实际存在。请看下面 mount命令的例子：

```
$ mount -t iso9660 -o ro /dev/cdrom /mnt/cdrom
```

这个mount命令会向内核传送三部分信息：文件系统名、包含该文件系统的块设备和新文件系统在当前文件系统拓扑结构中的装配位置。

VFS文件系统的第一个工作就是查找待装配的文件系统。它通过查看由 file_systems指向的表中的每个file_system_type数据结构，来搜索已知文件系统的列表。如果它找到了一个匹配名，就知道该文件系统是内核所支持的。从 file_system_type结构中，VFS文件系统可以获得读该文件系统超级块的文件系统专有例程的地址。如果它找不到匹配的文件系统，那么只要内核支持按需载入内核模块的话，一切还没有结束，这时内核在继续处理之前会要求内核守护进程载入适当的文件系统模块。

接下来如果由mount命令传送来的物理设备还没有被装配的话，VFS文件系统就必须要找到作为新文件系统的装配点的目录的 inode节点。该VFS inode节点可能在inode缓存中，也可能从装配点文件系统的物理块设备上读出。一旦 VFS找到了inode节点，就要检查该VFS inode节点是否代表着一个目录以及是否有其他的文件系统装配在这里。因为同一个目录不能作为一个以上文件系统的装配点。

这时，VFS文件系统的装配代码要分配一个 VFS超级块，并把装配信息传递给这个文件系统的超级块读例程。所有系统的 VFS超级块都被记录在super_block数据结构的super_blocks向量中，每次装配操作都必须分配一个超级块。超级块的读例程用从物理设备读取的信息填充VFS超级块的各域。对 EXT2文件系统来说，这种信息的映射或转换非常容易。它只是读入EXT2文件系统的超级块，并用它来填充 VFS文件系统的超级块。对象 MSDOS这样的文件系统就不是很容易实现了。无论什么类型的文件系统，填充 VFS超级块意味着文件系统必须从它所在的块设备中读取描述信息，如果块设备无法读取或块设备中没有这种类型的文件系统，mount命令就会失败。

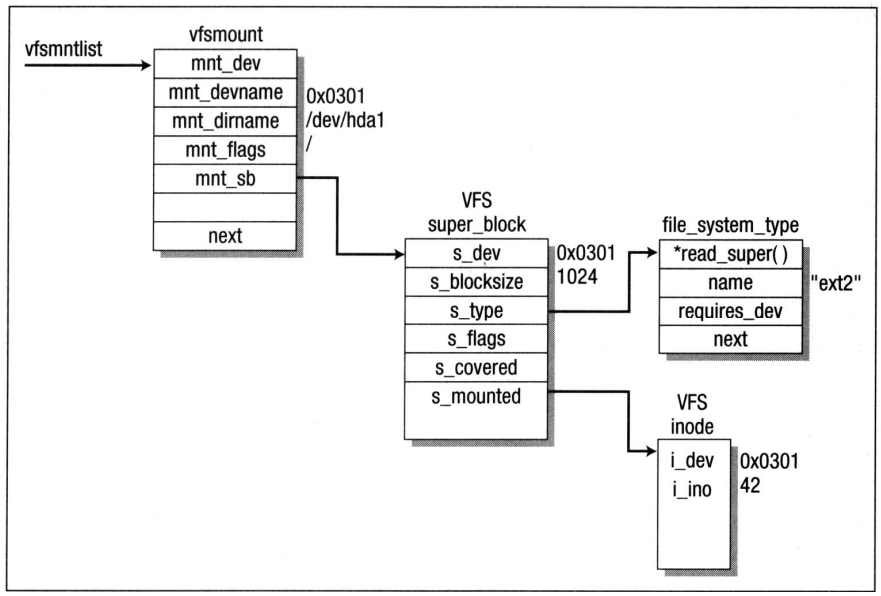


图1-7-6 安装的文件系统

每个装配的文件系统由 Vfsmount数据结构来表示。在图 1-7-6中，它们被放在由vfsmntlist

指向的队列链表中。另一个指针——vfsmntail指向表中的最后一项，mru_vfsmnt指针指向最近被使用的文件系统。每个vfsmount结构包括记录该文件系统的块设备的设备号、文件系统的装配目录和文件系统装配时分配的VFS超级块的指针。每个VFS超级块除了包含指向其对应文件系统的根inode节点的指针外，还指向它对应文件系统的file_system_type数据结构。每个文件系统的inode节点在本文件系统加载后一直驻留在VFS inode节点的缓存中。

7.2.5 在虚拟文件系统中查找文件

为了在VFS文件系统中查找某个文件的VFS inode节点，VFS文件系统必须以每次一个目录的方式来解析文件名，查找代表文件名中间目录的VFS inode节点。每个目录查找过程都包含对代表它父目录的VFS inode节点记录的文件系统专有的查找过程的调用。由于我们总是能通过该文件系统的超级块找到该文件系统的根VFS inode节点，所以上面的办法是可行的。每当真实文件系统查找一个inode节点时，系统都会先检查目录缓存。如果目录缓存中没有该目录项，那么真实文件系统可以通过下层的文件系统或者在inode缓存中找到要找的VFS inode节点。

7.2.6 卸载文件系统

如果系统的某些进程正在使用该文件系统的某个文件的话，该文件系统不能卸载。例如当系统的某个进程正在使用/mnt/cdrom目录或它的子目录的话，就无法卸载装配在/mnt/cdrom目录上的文件系统。如果有某些进程正在使用要卸载的文件系统的话，在VFS inode节点的缓存中就会有来自该文件系统的VFS inode节点。检查程序通过在inode节点表中查找来自于该文件系统所在设备的inode节点可以检测出这个问题。如果装配的文件系统的VFS超级块被标记为“脏”，这说明该超级块被改动过，所以文件系统要把它写回到硬盘上的文件系统中。一旦超级块被回写到硬盘上，由VFS超级块占用的存贮区就被归还给内核的自由存贮区池。最后为装配操作建立的vfsmount数据结构与vfsmntlist解除链接，被释放掉。

7.2.7 VFS文件系统的inode缓存

在访问装配的文件系统时，这些文件系统的inode节点被不断的读出、写入。VFS文件系统维护一个inode节点的缓存以加速对所有装配的文件系统的访问。每当有一个VFS inode节点被从inode缓存中读出时，系统就节约了一次对物理设备的访问。VFS inode节点缓存是用哈希(hash)表的形式实现的，表中的每一项是指向有相同哈希值的VFS inode节点的链表。VFS inode节点的哈希值是从它的inode节点号和包含它所在文件系统的下层物理设备标识计算出来的。当VFS文件系统要访问一个inode节点时，它先查找VFS inode节点缓存。为了找到在缓存中的inode节点，系统先计算它的哈希值，然后用这个哈希值作为inode哈希表的索引。这会返回给系统一个指向有相同哈希值的inode链表的指针。接着系统逐个读链表中的每个inode节点，直到找到了一个inode节点号和设备标识都与系统要寻找的inode节点完全相同的inode节点。

如果系统在缓存中找到了该inode节点，inode节点的引用计数加1，以表示又有另一个用户在使用该节点，然后文件系统的访问继续进行。否则的话，系统必须找到一个空闲的VFS inode节点，以便文件系统能够从硬盘上读出该inode节点。VFS文件系统在获得空闲inode节点

时可以有几种选择。如果系统可以分配多个 VFS inode节点的话，它会分配几个内核页，把它们折成新的空闲inode节点并加到inode表中。系统中所有 VFS inode节点除了在inode节点的哈希表中之外，还存在由 first_inode指向的一个链表中。如果系统已经分配了所有的可分配的inode节点，那么它必须要找到一个可重新使用的候选 inode节点。好的候选 inode节点指那些使用计数为0的节点，它表明当前系统没有在使用这些 inode节点。像文件系统的根 inode节点，这样真正重要的 VFS inode 节点，它的使用计数总是大于 0的。不可能被重新使用。一旦一个候选的重用inode节点被分配了，系统要先作清空操作。而 VFS inode节点可能被标记为“脏”，这时它需要被回写到文件系统中；或者它也有可能被锁定了，系统这时必须等到它被解锁后才能继续。候选的VFS inode节点在能重用之前必须先被清空。

在找到新的VFS inode节点之后，系统要调用文件系统的专有例程从下层真实文件系统读取信息来填充它。在该VFS inode节点被填充的同时，它的引用计数变为 1，并且被锁定以保证在它包含有效信息之前没有其他的进程可以访问它。

为了获取实际需要的VFS inode节点，文件系统需要访问几个其他的 inode节点。在你读一个目录时会发生这种现象；仅仅最后一个目录的 inode节点是我们需要的，但中间目录的 inode节点也必须被读取，随着对 VFS inode缓存的使用，它不断地被填满，最少使用的 inode节点被丢弃而使用率高的inode节点被保存在缓存中。

7.2.8 目录缓存

为了加速对经常使用的目录的访问， VFS文件系统维护着一个目录项的缓存。在真实文件系统查找目录时，它们的详细信息会被加入到目录缓存中，下次查找同一个目录时 (如列出目录中所有内容或打开该目录下的某个文件)，系统会在目录缓存中找到它。只有短目录项(最多15个字符)才会被缓存起来，但由于短目录名更经常被使用，所以这种方式是比较有道理的。如/usr/X11R6/bin在运行X服务器时经常会被访问到。

目录缓存包含一个哈希表，表中的每一项都指向有相同哈希值的目录缓存项的链表。哈希函数用该文件系统所在设备的设备号和目录名来计算对哈希表的索引值。它使得系统能较快地找到缓存的目录项。如果查找过程在缓存中花费很长时间才能确定是否会找到目录项的话，那么缓存不会带来任何好处。

为了保证缓存的更新和有效性， VFS文件系统保持最近被访问 (Least Recently Used,LRU)目录缓存项的列表。当目录项第一次被查找时，它被放在第一级 LRU表的尾端。在缓存满的情况下，它会替换掉该 LRU表前端的目录项。如果该目录项被再次访问的话，它会被提升到第二级LRU缓存表的尾端。如果缓存满的话，它会再次替换掉第二级 LRU缓存表前端的目录项。这种对第一级和第二级 LRU表前端的替换操作是符合 LRU规则的，因为位于表前端的目录项是最近没有被访问的。如果某些目录项被访问过，那么它们会更接近于表的尾部。在第二级LRU缓存表中的目录项比第一级 LRU缓存表的目录项要安全一些。它表明第二级的目录项不仅被查找过而且最近它们还被不断地访问过。

7.3 缓冲区缓存

在使用装配的文件系统时，它们会向块设备发出大量的读、写数据块的块请求。所有的读、写数据块的请求都通过标准内存例程调用以 buffer_head数据结构的形式发送给设备驱动

程序。在buffer_head数据结构中给出了块设备驱动程序所需的全部信息，有唯一标识设备的设备标识，通知驱动程序读哪一块的块号。所有的块设备都被当成相同大小块的线性集合。为了加速对物理设备的访问，Linux维护着一个块缓冲区的缓存。系统的所有块缓冲区都被放在这个缓冲区缓存中(包括新的未使用的缓存区)。这个缓存由系统的所有块设备共享，在某一时刻缓存中有很多分属于不同块设备、处于不同状态的块缓冲区。如果缓存中有系统所需的有效数据，那么它会减少一次对物理设备的访问。任何用于从块设备读出或写入数据的块缓冲区都被放入块缓冲区缓存中。随着时间的推移，一个缓存区可能因为为其他更有用的块腾出空间而被删除，也可能由于不断地被访问而一直保留在缓冲区缓存中。

缓存中的块缓冲区是由它所在设备的设备标识和对应块的块号唯一标识的，缓冲区缓存由两个功能部分组成。第一部分是自由块缓冲区的表。系统支持的不同长度的缓冲区都对应一个表。当自由块缓冲区被创建或被从缓冲区缓存中删除时，它们都会被放到这些自由块缓冲区的表队列中。系统当前支持的缓冲区大小可以是 512、1224、2048、4096、8192字节。第二个功能部分就是缓存本身，它是一个哈希表，其中每个项是指向有相同哈希索引值的缓冲区链表的指针。哈希索引值是通过该块所在设备的标识和数据块的块号来产生的。图 1-7-7给出了带有几个缓存项的哈希表，块缓冲区或者在某个自由表中或者在缓冲区缓存中。如果它们在缓冲区缓存中，就被放到 LRU表队列中。每种类型的缓冲区都对应一个 LRU表，系统用这些表执行像把缓冲区中数据回写到硬盘这样的工作。缓冲区缓存的类型反映出它的状态，Linux支持如下几种类型：

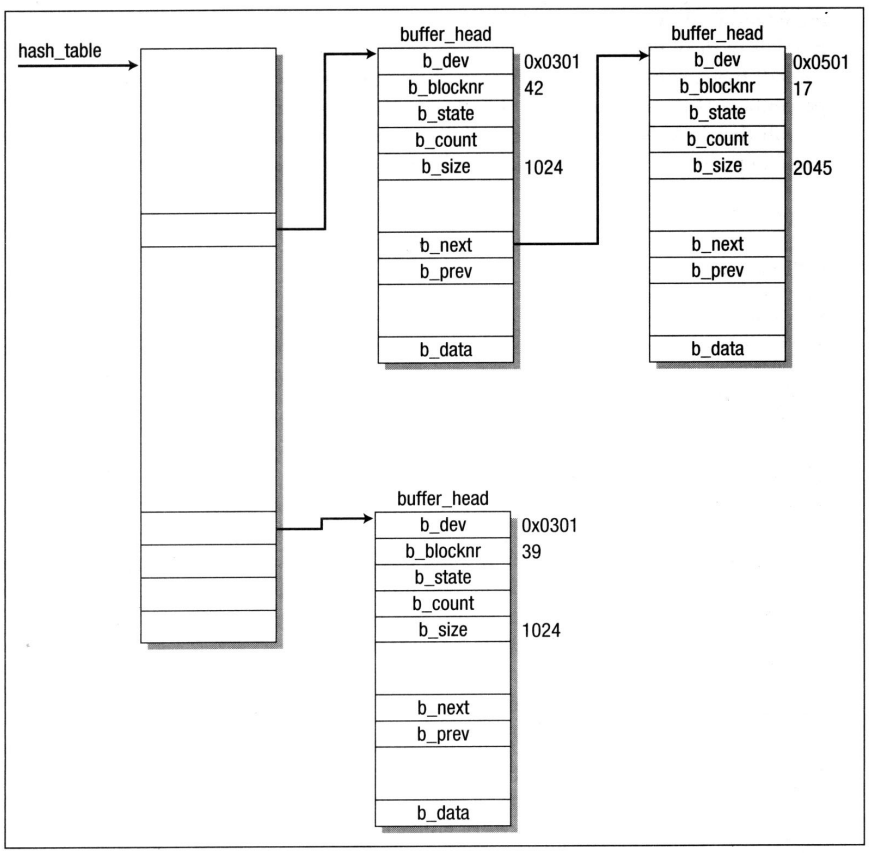


图1-7-7 缓冲区缓存

- 空白(clean) 未使用的新缓冲区。
- 锁定(locked) 缓冲区被锁定，等待写入。
- 脏(dirty) 脏缓冲区，它们包含新的将要被写回到硬盘上的有效数据，但系统还没有调度它们进行写操作。
- 共享(shared) 共享的缓冲区。
- 未共享(unshared) 缓冲区曾经被共享过，但现在没有处于共享状态。

文件系统需要从下层物理设备读缓冲区时，它会先试图在缓冲区缓存中找到该块。如果缓冲区缓存中没有，文件系统就会从相应大小的自由块表中找一个空白的缓冲区并把它加入到缓冲区缓存中，即使文件系统在缓冲区缓存中找到了这一块，该块也可能不是最新的。对新的块缓冲区和非最新的块缓冲区，文件系统要请求设备驱动程序从硬盘上读入相应的数据块。

像所有的缓存一样，系统必须维护缓冲区缓存以使得它是高效的并且对所有使用它的块设备来说能公平地分配缓存项。Linux使用bdflush内核守护进程对缓存区执行日常的维护工作而其他的工作是在使用缓存时自动进行的。

7.3.1 bdflush内核守护进程

bdflush内核守护进程是一个对有太多脏缓冲区的系统做出动态响应的简单的内核守护进程。脏缓冲区指包含还没有被回写到硬盘上的新数据的缓冲区。在系统启动时，它被作为内核的一个线程运行；但非常令人迷惑的是这时它的名字是 kflushed，也就是你使用ps命令查看系统进程时所看到的名字。这个进程大多数时间处于睡眠状态，等待系统中的脏缓冲区数目变得过大。每次分配释放缓冲区时，系统都要检查脏缓冲区的数目，如果它占系统总缓冲区数目的百分比太高的话，系统就会唤醒该进程。缺省的阀值是 60%，但如果系统急需缓冲区的话，也可以唤醒bdflush。阀值可以由update命令设置或查看：

```
# update -d

bdflush version 1.4
0:    60 Max fraction of LRU list to examine for dirty blocks
1:   500 Max number of dirty blocks to write each time bdflush activated
2:    64 Num of clean buffers to be loaded onto free list by refill_freelist
3:   256 Dirty block threshold for activating bdflush in refill_freelist
4:    15 Percentage of cache to scan for free clusters
5:  3000 Time for data buffers to age before flushing
6:   500 Time for non-data (dir, bitmap, etc) buffers to age before flushing
7:  1884 Time buffer cache load average constant
8:     2 LAV ratio (used to determine threshold for buffer fratricide).
```

当向缓冲区写入数据而把它们标志为“脏”时，所有的脏缓冲区都被链接到 BUF_DIRTY LRU表队列中；每次bdflush都会向脏缓冲区对应的硬盘中写入一定数缓冲区。这个数还可以由update命令查看和改变，它的缺省值是500。

7.3.2 update进程

update命令不仅是一个命令，也是一个守护进程。在系统初始化过程中它作为超级用户运

行，周期性地把旧的脏缓冲区写回到硬盘上。它通过调用一个与 bdflush的任务几乎相同的系统服务例程来完成上述工作。当系统使用完脏缓冲区时，它会被标记上系统时间，以便于确定何时应该被回写到硬盘上。每次update进程运行时，就会查找系统中的所有脏缓冲区，找出那些超期的脏缓冲区，把它们写回到磁盘上。

7.4 /proc文件系统

/proc文件系统才真正显示出了 Linux VFS文件系统的能量。/proc目录、子目录以及下面的文件都不是真正存在的。那么你怎么可以对 /proc/devices用cat命令呢？/proc文件系统像真实的文件系统一样向VFS文件系统注册自己。VFS文件系统在打开它的文件或目录，请求它的inode节点时，/proc文件系统利用来自内核的信息创建这些文件、目录。例如：内核的/proc/devices文件是从内核描述设备的数据结构创建来的。

/proc文件系统为用户提供了一个查看内核内部工作的只读窗口。像 Linux内核模式这样的Linux子系统，都在/proc文件系统中创建实体。

7.5 特殊设备文件

Linux像所有版本的 UNIX™一样，把它的硬件设备作为一个特殊文件看待。例如：/dev/null是空设备。设备文件不使用文件系统的任何数据空间，它只是设备驱动程序的访问点。EXT2文件系统和Linux VFS文件系统都把设备文件实现成特殊类型的 inode节点。系统有两种类型的设备文件：字符和块设备文件。在内核中设备驱动程序实现了文件的语义：你可以对它们执行打开、关闭等操作。字符设备允许为字符模式进行 I/O操作而块设备要求所有的I/O操作都经过缓冲区缓存。当设备文件收到的一个 I/O请求时，会把请求传送给系统中相应的设备驱动程序。这个设备驱动程序有时并不是真正的设备驱动程序而是像 SCSI设备驱动程序层那样的某些子系统的伪设备驱动程序。设备文件由标识设备类型的主设备号和标识主设备类型对应实例的次类型来访问。例如对系统第一个 IDE控制器的IDE硬盘，它的主设备号是 3，而IDE硬盘的第一个分区的类型号为 1。所以对/dev/hda1使用ls-l命令会产生如下结果：

```
$ brw-rw—   1 root    disk       3,    1  Nov 24  15:09 /dev/hda1
```

在内核中的每个设备都由一个两字节长的 kdev_t数据结构唯一地表示。kdev_t结构的第一字节包含次设备号，而第二字节包含主设备号。上面的 IDE设备在内核中被记为 0x0301。一个代表块或字符设备的 EXT2文件系统的inode节点，在它的第一个直接块指针中记录下设备的主设备号和次类型号。当 VFS文件系统读设备文件时，代表该设备文件的 VFS inode数据结构把自己的i_rdev域置成对应的设备标识符。

第8章 网 络

Linux几乎可以说是网络的一个同义词，事实上， Linux是一个Internet(因特网)或World Wide Web(万维网)的产品，其开发者和用户通过网络交换一些有用的思想和代码， Linux本身也经常用于网络的组织管理，本章介绍Linux是如何支持著名的TCP/IP协议族的。

TCP/IP，即传输控制协议/网际协议(Transmission Control Protocol/Internet Protocol)，实际上是一个由多种协议组成的协议族，它定义了计算机通过网络互相通信及协议族各层次之间通信的规范。

TCP/IP最初是在由美国政府资助的美国高等研究计划署的网络 ARPANET上发展起来的，该网络用于支持美国军事和计算机科学研究，正是由它提出了报文交换和网络分层概念。1988年以后， ARPANET由其继任者 —— 美国国家科学基金会的 NSFNET所取代，而NSFNET和全世纪数以万计的局域网和区域网共同连接成了一个巨大的联合体 —— 因特网(Internet)，举世闻名的万维网(World Wide Web)也是来自于ARPAnet并完全采用TCP/IP协议族。UNIX被广泛地应用于ARPANET，它的第一个网络版是4.3 BSD (Berkeley Software Distribution)，该版本支持BSD的套接字(略有扩充)和全部的TCP/IP协议，Linux的网络功能即是基于这个版本实现的。Linux之所以以该4.3 BSD版本为模型，是因为这个版本广为流行，并且它支持 Linux与其他UNIX平台之间应用程序的移植。

8.1 TCP/IP网络概述

本节将概述TCP/IP网络的主要原理。

在一个TCP/IP网络中，每台主机都分配有一个 32位的IP地址，该地址可以唯一地标识主机。IP地址通常用“ . ”隔开的四个十进制数表示，称为点分十进制表示，如 IP地址0x81124C15(16进制)通常写成129.18.76.21。

IP地址由两部分组成：网络 (network)地址和主机(host)地址。网络地址由IP地址的高位组成，主机地址由低位组成，这两部分的大小取决于网络的类型。如一个 B类地址(IP地址的第一个字节大小在128到191之间)，其IP地址的前两个字节是网络地址，后两个字节表示主机地址，这样一个B类地址可支持65536个网络，同时每个网络中可容纳65536台主机。

IP地址的主机部分可以分出多个子网，利用子网技术，大的网络 (即主机地址部分占较多字节)可以被分为若干小的子网 (subnetwork)，每一个子网均可独立维护。例如， IP地址16.42.0.9，可将其设置为子网地址16.42.0，其主机地址16.42.0.9，这种技术通常用来划分一个企业的网络，如将 16.42作为ACME计算机公司的网络地址，那么 16.42.0则为子网0，16.42.1则为子网1，这些子网或许位于相分离的建筑物中，它们通过租用的电话线甚至微波相连。通常由网络管理员为主机分配 IP地址，使用了子网技术将更有助于网络管理员的分派，并且管理员在其所辖子网内可以随意分配IP地址而不会有IP地址冲突。

一般说来，点分十进制表示的 IP地址不易记，而记名字则容易多了。因此，每台网络主机还有一个名字，由域名服务 (Damain Name Service, DNS)负责IP地址与网络主机名的互译，并在整个因特网上发布名字 —— IP地址数据库。使用网络主机名使得一台机器的 IP地址改变

(例如，这台机器被移到了另外一个网络上)时，不必担心别人在网络上找不到这台机器，这台机器的DNS记录只是更新了 IP地址，所有用网络主机名对这台机器的访问将继续有效。在Linux中，主机名字可静态地在 /etc/hosts文件中定义；也可请求分布式域名服务器 (Distributed Name Server, DNS)为其指定一个，这样主机必须应该知道一个或多个 DNS服务器的IP地址，这些信息定义在/etc/resolv.conf文件中。

当连接一台主机或访问一个Web主页时，都要通过本机的IP地址与被访问主机通信，用IP报文交换数据。IP报文分为两部分：IP报头与数据，在IP报头中，包含有源主机IP地址、目的主机IP地址、校验和和其他一些有用信息 (详见图1-8-1)，其中校验和是由源主机利用 IP报文数据计算得出的，目的主机据此判断报文在传输过程中是否被破坏。为了便于控制，应用程序传输的数据可能会被分片为更小的 IP报文，而IP报文的大小则由传输介质所决定：以太网报文通常要比点到点(Paint to Point Protocal, PPP)报文大一些，而目的主机在将数据交给应用程序之前，必须先重组报文。IP报头中的“ 标识 ”(IDENTIFICATION)域、“ 标志 ”(FLAG)域和“ 片偏移 ”(FRAGMENT OFFSET)域用来进行数据的分片与重组。如果通过较慢的网络传输图片，那么看一下图片的显示过程，即可感受到分片与重组的过程。

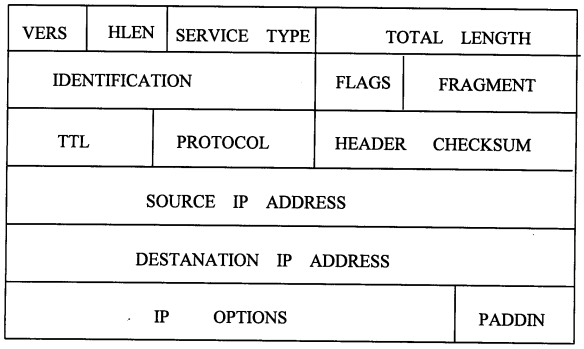


图1-8-1 IP报头数据格式

同一网络中的主机相互间可直接发送报文，但不同网络中的主机要通信，则必须通过一台特殊的主机：网关 (Gateway)。网关(或路由器)是一台同两个或更多个网络有直接连接的节点，网关可以在网络之间交换信息，把报文从一个网络传递到另一个网络。例如，网络16.42.1.0与16.42.0.0通过一个网关相连，则所有从网络 16.42.1.0发送到16.42.0.0的报文都须先发给网关，再由网关为其选择路由，转发报文。对应于每一个目的 IP地址，网关中的路由表都提供一个入口用以查询将该报文发送到哪台主机。这些路由表动态刷新，随应用程序使用网络的时机和网络技术的不同而改变。

我们可用netstat命令来查看某机器当前的路由表，其输出大致类似图 1-8-2。

```
dai %          netstat -rn
Kernel         routing        table
Destinaton     Gatevay address  Flags   Refcnt   Use      Iface
net/address
16.42.0.0      16.42.0.12       UN      0        1432     eth0
default        16.42.0.1        UGN     0        1432     eth0
17.0.0.1       17.0.0.1         UH      0        12       to
16.42.0.12     17.0.0.1         UH      0        12       to
```

图1-8-2 netstat命令输出

第一栏是路由表包括的目标网络或节点的地址。路由表第一条对应于网络 16.42.0，这是本机所在网络，任何本机发到这个网络的报文必须通过 16.42.0.12，即本机的IP地址。一般一个机器到自己网络的路由总是通过它自己。

Flags栏给出目标地址信息：U表示此路由“ up ”，N表示目标是一个网络，等等。 Refcnt和Use栏给出这条路由的使用统计。 Iface列出路由使用的网络设备。 eth0表示以太网接口， lo表示loopback设备。

路由表中第二条是缺省路由，适用于所有目的网络或带节点地址不在路由表中的报文。本例中16.42.0.1可看作通向外界的门户，所有 16.42.0子网的机器必须通过 16.42.0.1与其他网络通信。

路由表中第三条对应于地址 17.0.0.1，这是loopback地址，当机器想与自己建立 TCP/IP连接时适用这个地址，它使用 lo作为接口设备，避免了 loopback连接使用以太网接口 (eth0)，这样网络带宽不会因机器与自己对话而浪费。

路由表中最后一条是对地址 16.42.0.12，这是本机的IP地址。正如上述，它利用 17.0.0.1作为自己的网关。

连接不同网络的网关的路由表通常类似下面的例子 (图1-8-3)，假设这个网关在两个子网的地址分别为16.42.0.109和16.42.1.4。

Destination net/address	Gateway address	Flags	Refcnt	Use	Iface
16.42.0.0	16.42.0.109	UN	0	1432	eth0
16.42.1.0	16.42.1.4	UN	0	1432	eth1
default	16.42.1.43	UGN	0	1432	eth1
17.0.0.1	17.0.0.1	UH	0	12	to
16.42.0.109	17.0.0.1	UH	0	12	to

图1-8-3 netstat命令输出

本网关通过eth0设备与16.42.0网络相连，通过 eth1设备与16.42.1网络相连。如果16.42.0网络的机器想同外界通信，它将把报文发往它的网关 16.42.0.109，16.42.0.109将再把报文发往它的缺省路由，即网关 16.42.1.43。如此下去，报文从一个网关传递到下一个网关，直到到达目的网络。

从协议分层来看， IP是网络层协议， TCP是一个可靠的端到端传输层协议，它利用 IP层传接报文。TCP是面向连接的，它通过建立一条虚电路在不同的网络间传输报文，可以保证所传输报文的无丢失性和无重复性。用户数据报协议 (User Datagram Protocol, UDP)也要利用IP层传输报文，但它是一个非面向连接的传输层协议。利用 IP层传输报文时，当目的方 IP层收到IP报文后，必须能够识别出该报文所使用的上层协议 (即传输层协议)。因此，在IP报头(参见图1-8-1)中，设有一个“ 协议 ”域(PROTOCOL)。通过该域的值，即可判明其上层协议类型。传输层与网络层在功能上的最大区别是前者提供进程通信能力，而后者则不能。在进程通信的意义上，网络通信的最终地址就不仅仅是主机地址了，还包括可以描述进程的某种标识符。为此， TCP/UDP提出了协议端口(protocol port，简称端口)的概念，用于标识通信的进程。例如， Web服务器进程通常使用端口80，在/etc/services文件中有这些注册了的端口地址。

分层协议不只包括TCP、UDP与IP，IP层本身亦要用到许多不同的物理介质来传输 IP报文，这些介质也有自己的报头信息，例如以太网层。一个以太网允许多台主机同时连到一根缆线上，任一台主机发送的帧对其他所有主机都是可见的，因此每台主机都有一个唯一的以太网

地址，指明了目的以太网地址的帧将只被拥有该地址的主机接收。以太网地址是一个 48比特的整数，以机器可读的方式存入主机接口中，叫作硬件地址（ hardware address）或物理地址（physical address）。以太网地址的一个重要特性是每一地址都与一个特定主机接口联系在一起，接口与地址一一对应。以太网地址共 6字节长，为保证主机以太网地址的全球唯一性，以太网采用一种层次型地址分配方式：以太网地址管理机构 (IEEE)将以太网地址(48比特的不同组合)分成若干独立的连续地址组，生产以太网接口板的厂家从中购买一组，具体生产时，再从所购地址中逐个将唯一地址赋予接口板。以太网地址中有一些保留地址用于多目的通信，当某一以太网帧拥有这样一个目的地址时，以太网上将会有多台主机同时接收这一帧。像 IP报文一样，以太网帧同样支持多种上层协议，这样在以太网帧头中也有一个协议域，通过该域的值，以太网层即能将所收到的IP报文正确地传给IP层。

IP地址是一种简单的地址，在分配或改变 IP地址时，网络管理员可以随心所欲，但同时物理地址是固定的，而网络硬件只响应物理地址，这样就必须提供一种机制，来完成这两种地址的映射。在Linux中采用的是地址解析协议(Address Resolution Protocol, ARP)和逆向地址解析协议(Reverse Address Resolution Protocol, RARP)。ARP用于从IP地址到物理地址的映射，RARP用于从物理地址到IP地址的映射。当主机A欲解析主机B的IP地址I_B时，A首先广播一个ARP请求报文，请求IP地址为I_B的主机回答其物理地址P_B。网上所有主机(包括B)都将收到该ARP请求，但只有B识别出自己的I_B地址，并作出应答：向A发回一个ARP响应，回答自己的物理地址P_B。ARP并非只能服务于以太网，它也支持其他一些物理介质，例如 FDDI。RARP通常由网关所用，以响应对远程网络IP地址的ARP请求。

8.2 Linux中的TCP/IP网络层次结构

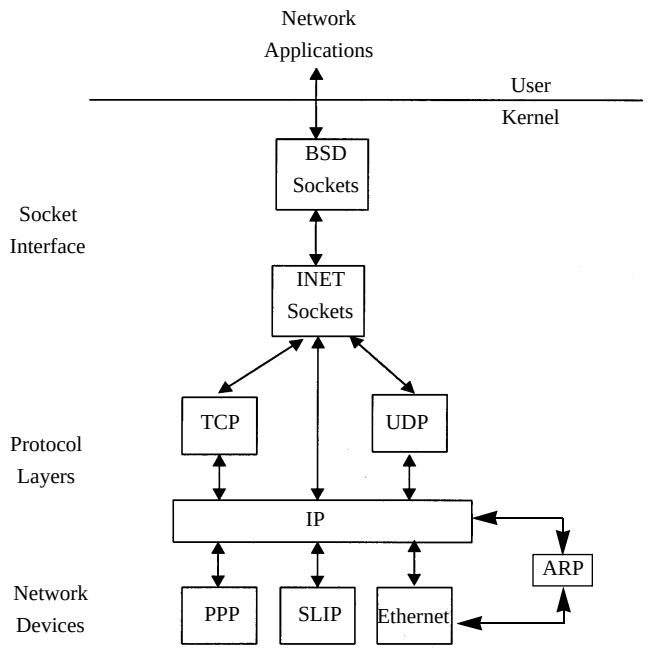


图1-8-4 Linux网络层次结构

图1-8-4描述了Linux对IP协议族的实现机制，如同网络协议自身一样， Linux也是通过视

其为一组相连的软件层来实现的。其中 BSD套接字（Socket）由通用的套接字管理软件所支持，该软件是INET套接字层，它来管理基于 IP的TCP与UDP的端到端互联问题。如前所述，TCP是一个面向连接协议，而 UDP则是一个非面向连接协议，当一个 UDP报文发送出去后，Linux并不知道也不去关心它是否成功地到达了目的主机。对于 TCP传输，传输节点间先要建立连接，然后通过该连接传输已排好序的报文，以保证传输的正确性。 IP层中代码用以实现网际协议，这些代码将IP头增加到传输数据中，同时也把收到的 IP报文正确地转送到TCP层或UDP层。IP层之下，是支持所有Linux网络应用的网络设备层，例如点到点协议 (Point to Point Protocol, PPP)和以太网层。网络设备并非总代表物理设备，其中有一些（例如回送设备）则是纯粹的软件设备，网络设备与标准的 Linux设备不同，它们不是通过 mknod命令创建的，必须是底层软件找到并进行了初始化之后，这些设备才被创建并可用。因此只有当启动了正确设置了以太网设备驱动程序的内核后，才会有 /dev/eth0文件。ARP协议位于IP层和支持地址解析的协议层之间。

8.3 BSD套接字接口

这是一个通用接口，它不仅支持不同的网络结构，同时也是一个内部进程间通信机制。一个套接字描述了一个连结的一个端点，因此两个互联的进程都要有一个描述它们之间连结的套接字，可以把套接字看作是一种特殊的管道，只是这种管道对于所包含的数据量没有限制。Linux支持套接字地址族中的多个，如下所列：

UNIX	Unix域套接字
INET	使用TCP/IP的因特网地址族
AX25	业余无线X25
IPX	IPX
APPLETALK	APPLETALK
X25	X25

Linux支持多种套接字类型。套接字类型，是指创建套接字的应用程序所希望的通信服务类型。同一协议族可能提供多种服务类型，比如 TCP/IP协议族提供的虚电路与数据报就是两种不同的通信服务类型， Linux BSD支持如下几种套接字类型：

- Stream 提供可靠的面向连接传输的数据流，保证数据传输过程中无丢失、无损坏和无冗余。INET地址族中的TCP协议支持该套接字。
- Datagram 提供数据的双向传输，但不保证消息 (message)的准确到达，即使消息能够到达，也无法保证其顺序性，并可能有冗余或损坏。 INET地址族中的UDP协议支持该套接字。
- Raw 是低于传输层的低级协议或物理网络提供的套接字类型，比如通过分析为以太网设备所创建的Raw套接字，可看到裸IP数据流。
- Reliable Delivered Messages 类似于Datagram套接字，但它可以保证数据的正确到达。
- Sequenced Packets 类似于Stream套接字，但它的报文大小是可变的。
- Packet 这是Linux对标准BSD套接字类型的扩展，它允许应用程序在设备层直接访问报文数据。

利用套接字通信的进程一般采用客户 —— 服务器模型：服务器提供服务，客户是这些服务

的使用者。例如Web服务器提供Web页，而Web用户访问这些Web页。服务器首先创建一个套接字，然后为其绑定一个名字，名字的格式取决于该套接字的地址族，由 sockaddr数据结构定义，但通常是该服务器的主机地址。一个 INET套接字还要绑定一个IP端口号，比如Web服务的端口号为80，在/etc/services文件中保存有注册过的端口号。地址绑定之后，服务器即开始在该地址上侦听连接请求，而客户端则为建立连接创建一个套接字，并指明目的地址 —— 服务器地址。对一个INET套接字而言，服务器地址就是该主机的 IP地址加上一个端口号。这些客户请求首先要能够通过多种协议层到达服务器端，然后进入等待队列，一旦服务器收到这些请求，首先要判断是否接受，若同意接受，则服务器必须再创建一个新的套接字用以接受该请求，这是因为一旦一个套接字用于侦听，那么它就不能再被用于支持连接。连接建立之后，双方即可进行数据传输；当连接不再需要时，须将其关闭。在传输过程中，要注意正确处理传输的数据报文。

一个BSD套接字操作的具体含意取决于低层的地址族，建立一个 TCP/IP连接就与建立一个业余无线X.25连接有很大不同。类似于虚文件系统， Linux利用与应用程序所用的BSD套接字接口相关的BSD套接字层将套接字接口抽象出来，同时这些应用程序所用的 BSD套接字接口又由各种独立的地址族软件所支持。初始化内核时，编入内核的各地址族将注册它们自己的BSD套接字接口；之后，当应用程序创建和使用 BSD套接字时，系统将把BSD套接字与支持该套接字的地址族联结起来，这种联结是通过交叉链接地址族例程的数据结构和表形成的。比如某一地址族定义了创建套接字例程，则当一应用程序创建一个新套接字时，将使用该例程。

当内核设置时，将会建立一些地址族和协议的协议向量 (protocol vector)，用它们的名字来代表每一个向量，例如“ INET ”和它的初始化例程地址。在系统启动时要初始化套接字接口，这时将调用每一个协议的初始化例程，这对套接字地址族而言意味着注册了一组协议操作，这些操作针对各自的地址族都做了些工作，它们被保存在 pops向量中。pops向量包括一些指向proto_ops数据结构的指针， proto_ops数据结构包括地址族类型和一组指针，这些指针指向特定地址族所定义的套接字操作例程，可利用地址族标志检索 pops向量，例如INET的标志为2。

8.4 INET的套接字层

INET套接字(socket)层支持包括TCP/IP协议在内的INET地址族(Address Family)，如前所述，这是一些分层协议，下层协议为上层协议提供服务。 Linux中实现TCP/IP协议的代码与数据结构充分体现了这种协议分层。 INET套接字层接口是通过一组INET地址族套接字操作实现的，这些操作在网络初始化时被注册到了 BSD套接字层，与其他注册的地址族一起保存在pops向量中。BSD套接字层通过调用注册在 INET proto_ops数据结构中的INET套接字层例程来完成上述操作。在进行每一项操作时， BSD套接字层都要把代表 BSD套接字的数据结构传给INET层，INET套接字层并非简单地抽取BSD套接字中的特定TCP/IP信息，而是使用自己的sock数据结构，该数据结构已被链接到 BSD socket数据结构上了，在图 1-8-5中给出了这种链接，这种链接通过BSD socket中的data(数据)指针将sock数据结构链到了BSD socket数据结构上。这样以来，随后的INET套接字调用将会很容易的得到套接字数据结构。在创建套接字时，也建立了指向套接字数据结构的协议操作的指针，这些指针与所使用的协议有关：当使用

TCP时，它们将指向与建立TCP连接有关的一组TCP协议的操作。

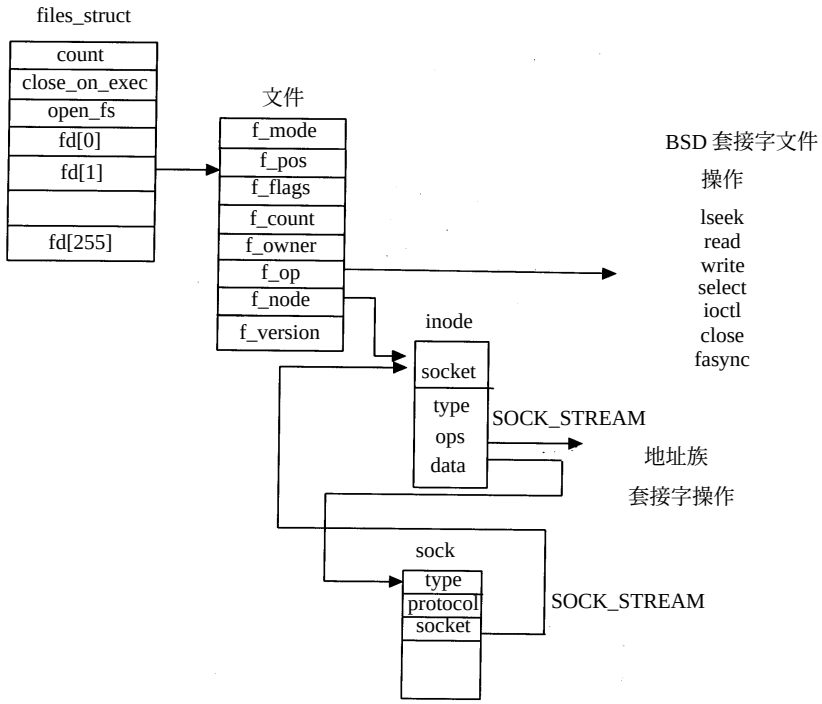


图1-8-5 Linux BSD套接字数据结构

8.4.1 创建BSD套接字

应用程序在使用套接字之前，首先必须拥有一个套接字，系统调用 socket( )向应用程提供创建套接字的手段，该系统调用必须给出所使用的地址族、套接字类型和协议的标志。

首先，系统利用地址族标志找到与之匹配的 pops向量，若地址族较为特殊，则应先由kerneld守护程序载入实现该地址族的功能模块，然后为这一 BSD套接字分配新的套接字数据结构。事实上，套接字数据结构从物理上讲是 VFS(Virtual File System)索引节点数据结构的一部分，分配一个套接字数据结构也就意味着分配一个 VFS索引节点数据结构。不必奇怪，只要想一想套接字也是一种文件就明白了。由于所有的文件都由 VFS索引节点所指代，为了支持文件操作，BSD套接字自然应有一个相应的VFS索引节点。

新创建的BSD套接字数据结构中包含了一个指针，它指向地址族所定义的套接字例程，在proto_ops数据结构中对比进行了设定。该套接字的类型按调用时的要求设定，诸如SOCK_STREAM或SOCK_DGRAM等等。对地址族所定义的创建例程的调用，是通过proto_ops数据结构中保存的地址来进行的。

还要从当前进程的fd(File descriptor)向量中分配一个空闲的文件描述符，并初始化该描述符所指向的file(文件)数据结构，这包括设置文件操作指针指向 BSD套接字接口所支持的一组BSD套接字文件操作。之后的任何文件操作都将直接调用套接字接口，而套接字接口程序将通过调用它的地址族操作例程将其转交给它相应的地址族。

8.4.2 为INET BSD Socket绑定地址

为了侦听从互联网上发来的连接请求，每一个服务器必须先创建一个套接字，然后为其绑定一个地址。通常绑定操作是在INET套接字层利用下层TCP或UDP协议的支持完成的。将要绑定的地址不能正用于其他的连接通信，这意味着套接字的状态必须为 TCP_CLOSE。待绑定地址包括一个IP地址及一个端口号(可选)，通常IP地址已分配给了一个网络设备，这个设备应支持INET地址族并且其接口是活跃而且可用的 (可通过ifconfig命令查看系统中当前活跃的网络接口进程)。IP地址可以为全“ 1 ”或全“ 0 ”的广播地址，这样通信数据将传给每一个网络设备，也可以任意指定一个 IP地址，只要本机是一个透明代理或防火墙，但只有具有超级用户权限的进程才可以这么做。在套接字数据结构中， recv_addr和saddr域保存了绑定的IP地址。若未指明可选的端口号，下层网络将会指定一个可用的端口号。按惯例，小于 1024的端口号对于无超级用户权限的进程是不可用的，因此下层网络通常分配一个大于 1024的端口号。

由于网络设备接收到报文后，还要将其正确地递交到 INET和BSD套接字，因此 TCP和UDP都维护有哈希(hash)地址表，其中保存有IP地址和BSD套接字的映射关系。通过用所收到报文的IP地址检索该表，即可找到相应的套接字，然后正确递交。因为 TCP是一种面向连接协议，所以处理TCP报文要比处理UDP报文使用更多的信息。

UDP哈希表中包括了 sock数据结构指针，通过以端口号作为参数的哈希函数进行检索。由于UDP的哈希表小于实际可用的端口号，所以表中指针通常指向一个 sock数据结构链，它们通过sock中的next指针链接起来。

TCP维护了多个哈希表，因为它相对复杂多了。但注意在操作过程中， TCP并非在进行绑定(bind)操作时将sock数据结构加入到哈希表中，而是在进行侦听 (listen)操作时完成这一加入过程的。绑定操作时， TCP仅仅审查一下所申请端口号是否可用。

8.4.3 建立INET BSD Socket连接

一旦一个套接字创建之后，若其未用于侦听本地进程间连接请求，即可将其用于侦听远程进程间连接请求。对于非面向连接的 UDP，不需作太多的工作，但对于 TCP，由于它是面向连接的，因此需要在两通信进程间建立一条虚电路。

用于建立连接的INET BSD套接字，其状态必须是SS_UNCONNECTED。UDP协议不需建立虚连接，因此它所传输的消息不一定能正确到达，但它支持 BSD套接字的连接操作。一个UDP INET BSD套接字连接操作只是简单地设定好远端进程地址 —— IP地址和端口号，并设置一个路由表入口缓存 (cache)，这样基于本BSD套接字的报文无需再次查询路由数据库 (除非该路由故障)。INET sock数据结构中的ip_route_cache指针指向缓存路由信息，如果这个 BSD套接字未指明地址信息，则使用该路由信息传送消息，并且由 UDP将sock状态置为TCP_ESTABLISHED。

对于TCP BSD套接字连接操作， TCP必须建立一个包含连接信息的消息，并发送到指定的IP地址。这些连接信息包括一个唯一的起始消息序号、初始化主机所能处理的最大消息长度以及接收窗口大小等等。 TCP中所有消息均被编号，首编号用于第一个消息， Linux选用了一种有效的随机方式取到首编号值，以避免恶意的网络攻击。传输过程中，接受方须向发送方进行确认，指示其所收到的正确消息编号，发送方将重传未被确认的消息。传输窗口大小表示在确认了字节之后还可以发送多少个消息。最大消息长度取决于发出初始连接请求的网

络设备，但如果接收方网络设备所支持的最大消息长度更小，则连接将取用两者中较小的。指明传输窗口大小的发送方要等待接收方的接受或拒绝响应，也就是说要等待接收消息，这样就需要将sock加入到tcp_listening_hash中，当所等待消息发来后，即可被正确递交给该 sock 数据结构，同时TCP启动定时器，确定传输是否超时。

8.4.4 INET BSD Socket侦听

套接字绑定了地址后，可能要侦听发给自己的绑定地址的连接请求 (某些应用程序可以不经绑定而直接侦听，这种情况下， INET套接字层会自动为其绑定一个可用的端口号)。侦听套接字程序就将套接字状态置为 TCP_LISTEN，并做好其他一些允许接收连接请求的工作。

对UDP而言，只需设置套接字状态即可；但对于 TCP，还要把套接字的sock数据结构放入两张哈希表中： tcp_bound_hash表和tcp_listonry_hash表，这两张表也都是通过以端口号为参数的哈希函数进行检索的。

一旦一个TCP连接请求到达后， TCP将建立一个新的sock数据结构代表该请求，若连接请求最终被接受，则新的 sock数据结构将成为半个TCP连接，同时复制包含连接请求的 sk_buff，并放入侦听 sock数据结构中的receive_queue队列，复制的sk_buff包含有指向新建 sock数据结构的指数。

8.4.5 接受连接请求

UDP不支持连接概念，因此接受 INET套接字连接请求是对 TCP而言。它引发由原侦听套接字复制新套接字数据结构。接受操作由支持套接字的协议层来完成，也就是由 INET接受发来的连接请求，若下层协议不支持连接 (如UDP)，则INET协议层接收失败；否则，将接受操作递交给TCP协议。接收操作有两种：有阻塞的和无阻塞。无阻塞操作中，若无连接请求到来，接受操作失败，释放新建的套接字数据结构；有阻塞操作中，实现接收操作的网络应用程序会在队列中等待并挂起，直到收到一 TCP连接请求。一旦接收到连接请求，则丢弃包含该请求的sk_buff，将sock数据结构回交给INET套接字层，并在该层将其链接到早先新建的套接字数据结构上。新套接字的文件描述符 (fd)回交给网络应用程序，然后应用程序就可以用文件描述符在新建的INET BSD套接字上进行套接字操作。

8.5 IP层

8.5.1 套接字缓冲区

协议分层为网络传输带来了一些问题，其中之一就是在利用各层发送数据时，每层都要加上自己的头部和尾部信息，而接收时又要由每层将这些信息去掉，这样就使得数据缓冲区的分配变得更为困难，因为每层都要能在缓冲区中找到特定的头部和尾部。一种解决方案是在每一层都对缓冲区进行全拷贝，但这样做效率太低。在 Linux中，各层协议和网络设备驱动程序间只传递套接字缓冲区或 sk_buff，sk_buff中有指针和长度域，这样各层协议即可通过标准函数或方法使用数据。

图1-8-6给出了sk_buff的数据结构，每一个 sk_buff有一个相关联的数据块。 sk_buff中有4个数据指针，用于使用和管理套接字缓冲区中的数据：

- head 指向内存中数据的起始区，一旦分配了 sk_buff及其相关数据块，即可确定该指针。
- data 指向当前协议数据的开始，该指针取决于当前拥有 sk_buff的协议层。
- tail 指向当前协议数据尾，与data一样，它也依赖于当前拥有sk_buff的协议层。
- end 指向内存中数据的结束区，分配了sk_buff后，该值确定。

其中有两个长度域：

a. len 当前协议报文的长度。

b. truesize 整个数据缓冲区长。

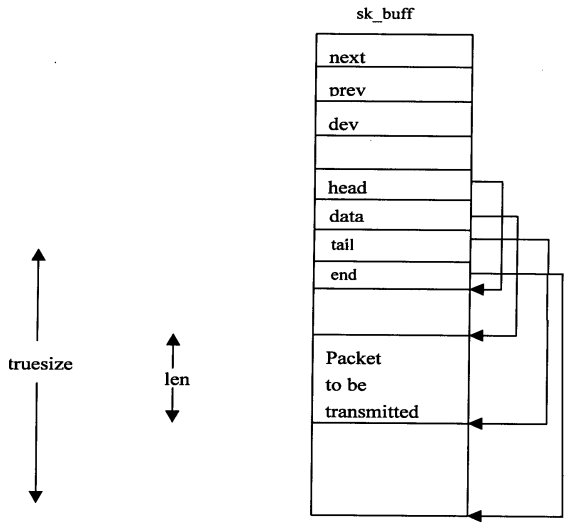


图1-8-6 套接字缓冲区

sk_buff的控制处理代码对添加和删除协议头和尾提供了标准操作，通过这些操作可安全地使用sk_buff中的data、tail和len域：

- push 将data指针指向数据开始区，并相应增加 len域，用于向待传输数据添加数据或协议头。
- pull 将data指针指向数据区的end，并相应减小len域，用于从接收到的数据中删除数据或协议头。
- put 将tail指针指向数据区的end，并相应增加len域，用于向发送的数据end处添加数据或协议信息。
- trim 将tail指针指向数据区起始处，并相应减小 len域，用于从接收到的报文中删除数据或协议尾。

sk_buff数据结构中还有 sk_buff的双向链表指针，并有通用的 sk_buff例程用这些指针从sk_buff链头或链尾对sk_buff数据结构进行添加或删除。

8.5.2 接收IP报文

第6章曾讲述了 Linux是如何将网络驱动程序建立到内核中并进行初始化的，这样在dev_base链中相应就有了一系列的设备数据结构，每个设备数据结构描述了其设备并提供了一组回调例程。当网络各协议层需要网络驱动程序为其工作时，就要调用这些例程，它们通

常和使用网络设备地址的数据传输有关。当网络设备从网上接收到报文后，必须先将接收到的数据转换到sk_buff数据结构中，然后把sk_buff添加到backlog队列中。当backlog队列太长时，新接收到报文的sk_buff将被抛弃，一旦队列中有待处理的sk_buff，网络bottom half即被置为可用状态。

网络bottom half控制处理进程被调度器激活后，先处理待发送报文，然后处理sk_buff的backlog队列，以决定向哪一层递交接收到的报文。Linux初始化网络各层时，每种协议都要注册，它们分别把各自的packet_type数据结构加入到ptype_all或ptype_base表中。packet-type数据结构中包括有协议类型、一个网络设备指针、一个协议接收数据处理例程指针和一个指向下一个packet_type数据结构的指针(用于维护表或哈希链)。ptype_all链用于检测从网络设备接收到的非常用协议报文。ptype_base哈希表以协议标志为索引，用以判别将接收到的网络报文递交给哪个协议。网络bottom half进程对协议类型进行匹配，以免在上述两类表中找到多个入口，但当检测所有网络传输时，的确可能匹配到不止一个入口，这种情况下，将复制sk_buff，所有sk_buff都将递交给匹配到的协议处理例程。

8.5.3 发送IP报文

应用程序相互间交换数据时，要传输报文；网络协议支持建立连接或当连接已经建立好后，也要进行报文传输。无论哪种情况下发送报文，都要首先建立包含数据的sk_buff，并且各层都要对即将发送的数据添加各自的协议头。

sk_buff必须先传送到网络设备然后才能发送。那么首先它要经过协议层，例如IP，由其决定使用哪个网络设备，这取决于发送该报文的最佳路径。如果计算机是通过modem连入网络的，即使用PPP协议，那么路径选择就很简单：通过回送设备发给本地主机或发给PPP modem连接末端的网关；但对于连接到以太网上的计算机而言，则将较为困难，因为网络中有许多台计算机。

对每一个IP报文的发送，IP用路由表解决寻径问题。通过查询路由表返回的rtable数据结构，即可找到到达目的IP地址的正确路径。这种查询要用到源IP地址、网络设备数据结构地址，有时还要用到预编译硬件头，这种硬件头由网络设备定义，内有源和目的物理地址以及其他一些特定的媒体信息。若网络设备为以太网设备，那么它的硬件头如图1-8-7所示，其中的源和目的地址就是以太网物理地址。因为使用任一个路由的IP发送报文都将硬件头加到IP报文之前，而硬件头的构造又需时间，所以将硬件头与IP路由缓存在一起，这样可以提高效率。对硬件头中物理地址的解析可能要用到ARP协议，这时要先把报文存起来，直到地址解析完成后再发送。要缓存解析后的地址信息缓存，之后用到同样接口的IP报文就不用再次进行ARP解析了。

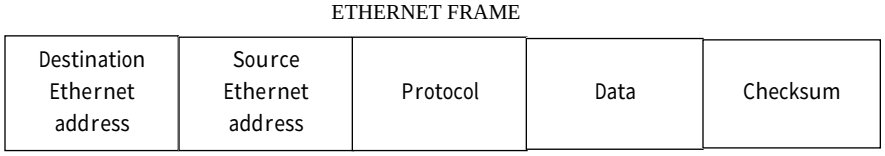


图1-8-7 以太网硬件头

8.5.4 数据分片

每种网络设备都有一个最大报文长度限制，对于大于该长度的报文，它既不能接收，也

无法发送。鉴于此，IP协议能将大的数据分片为几个，以支持网络设备的处理能力。如前所述，IP协议头中的标识域、标志域和片偏移域用于分片与重组工作。

IP报文发送前，先要从IP路由表中找出所用的网络设备。每个设备都有一个最大传输单元(Maximum Transfer Unit, MTU)域，指明该设备所能支持的最大报文长度。若设备的 MTU小于待发送的IP报文长度，则须将IP报文分片，由一个sk_buff代表每一片，片的IP头中的标志域指出其是否分片，片偏移域指出本片在整个 IP报文中的字节偏移量，若在分片过程中无法分配可用的sk_buff，则传输失败。

分片的方法及格式如图1-8-8，该图说明报头长24个字节、数据区长1600个字节的数据报在MTU为700字节的物理网络中分片的情况。

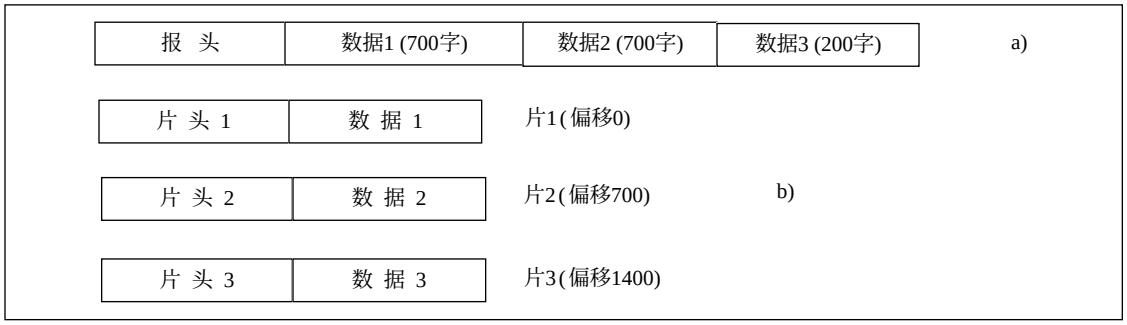


图1-8-8 IP报文的分片与重组

a) 数据区大小为1600字节的初始数据报 b) 在MTU=700字节的网络上的三个分片

接收IP分片会更复杂一些，因为分片可能以各种顺序到达，并且在接收到所有的分片之后才能对其重组。每当接收方收到IP报文后，先检测其是否被分片，当接收到第一个分片时，IP将创建一个新的ipq数据结构，并把它放入等待重组的ipqueue链中。更多的分片到达后，先找到相应的ipq数据结构，并创建一个新的ipfrag数据结构以描述各片。每个ipq数据结构都有唯一的源和目的IP地址、上层协议标识以及IP片标识，所有的分片都到达后，将被组合到一个sk_buff中，然后递交到上一层协议进行处理。每个ipq都有一个定时器，每收到一个正确的分片就重置定时器，若定时器超时，则释放 ipq数据结构以及它的ipfrag数据结构，然后生成一个“丢失”消息，最后将该消息交由上层协议处理。

8.6 地址解析协议

地址解析协议(Address Resolution Protocol, ARP)的任务就是实现IP地址和物理硬件地址(如以太网地址)间的互译，在把数据交由设备驱动程序进行发送之前，IP需要先得到解析结果。对于判别设备是否需要硬件头以及若需要则是否应重建硬件头， IP有多种检测方法，Linux通过缓存硬件头来避免对它们的多次重建。若硬件头确需重建，则调用设备特定的硬件头重建例程，所有的以太网设备使用同一个重建例程，该例程利用 ARP服务将目的IP地址转化为相应的物理地址。

ARP协议本身很简单，仅包括两种消息类型： ARP请求与ARP应答。ARP请求中包含有欲解析的IP地址，应答报文中则包含了对相应IP地址的解析结果 —— 硬件地址。

Linux中的ARP协议层是基于arp_table数据结构表实现的。arp_table数据结构记录了对应于某一IP地址的物理地址解析，当IP地址需要解析时则创建；记录已因过时而失效时则删除，每个arp_table数据结构都包括如下的域：

last used	ARP入口上次访问时间
last updated	ARP入口上次刷新时间
flags	描述接口状态，例如是否完成等
IP address	入口IP地址
hardware address	解析得到的硬件地址
hardware header	缓存的硬件头指针
timer	一个timer_list入口，用于检测ARP是否超时
retries	ARP请求的重试次数
sk_buff queue	sk_buff入口表，表中入口均等待本次ARP结果

ARP表中包含有一个指针表，这些指针指向arp_tables入口链，而这些链均已被缓存，以提高访问速度。以入口的最后两字节的IP地址为索引，对ARP表进行检索，然后在检索到的入口链中顺次查找所要的IP地址，Linux还缓存了hh_cache数据结构，该数据结构中有从arp_table入口中取出的预编译硬件头。

当发出一个IP地址解析请求而又无相应的arp_table入口时，ARP必须发出一个ARP请求消息。ARP先在arp_table入口表中创建一个新的arp_table，并将需要该地址解析结果的sk_buff放入新入口的sk_buff队列中，然后发送ARP请求报文并启动定时器。若超时无响应，将重发一定次数的ARP请求，多次请求仍无响应，则删除新建的arp_table入口，并通知等待的sk_buff数据结构队列，队列中的各sk_buff再请求上层协议处理本次失败操作。对此UDP并不关心是否丢失报文，但TCP将尝试在一条已建立的连接上重发报文，若目的主机响应请求并发回硬件地址，则置该arp_table入口为完成，并从sk_buff队列中移出各项，让它们继续完成发送工作，这时所需硬件地址已写入这些sk_buff的硬件头中。

ARP协议层必须响应指定了IP地址的ARP请求，它先注册协议类型(ETH_P_ARP)，并生成一个packet_type数据结构，这意味着所有由网络设备接收到的ARP报文都将交由它处理，包括ARP请求与响应，它利用保存在接收设备的device数据结构中的硬件地址完成响应。

由于网络拓朴结构可以随时改变，因此IP地址也可以重新分配给不同的硬件地址，例如，一些拨号服务每当建立起一个连接就分配一次IP地址，为了保持ARP表中入口的实时可用，ARP用一个周期性定时器，该定时器将定期检测ARP表以确定超时的入口，但它并不删除含有一个或多个缓存硬件头的入口，这样做会很危险，因为有其他数据结构有赖于这些入口。某些arp_table入口是永久性的，对它们也作有标记以免其被释放。由于每一个arp_table入口都要消耗核态内存，所以ARP表不能太大，一旦要分配一个新入口时发现ARP表已达最大，则通过查找并删除最旧的入口来压缩ARP表。

8.7 IP路由

进行IP寻径时，首先检测路由缓存，若无匹配的路由信息，再搜索转发信息数据库(Forwarding Information Database)，如果仍不成功，则本次IP报文发送失败，并通知应用程序；如果找到了路由信息，就生成一个包含该信息的新入口，并将其加入到路由缓存中。路

由缓存是一张表(ip_rt_hash_table)，表中包含了rtable数据结构链指针，对路由表的检索是通过哈希函数完成的，这个哈希函数以IP地址中最不重要的两个字节为参数，这两个字节应以如下规则选定：对于不同的目的地址应尽可能不相同，这样做可以提供更好的哈希值。每个rtable入口包含了路由信息：目的IP地址、到达目的IP地址所要用到的网络设备、最大消息长度等等。它还有一个引用计数、一个使用计数和一个上次被使用的时间戳。每次用到一个路由，就将其引用记数增1，以标明使用该路由的网络连接数量；应用程序不再使用该路由时，就将引用记数减1。使用计数是在每次对其路由进行了查找时增 1，以此在哈希入口链中对路由入口排序。上次被使用的时间戳用于对路由表周期性地检测，以便找出最旧的信息入口，若某个路由最近未被使用过，则将其删除。路由缓存中的入口以使用次数排序是为了使利用率最高的路由放在哈希队列头，这样可提高路由查找效率。

转发信息数据库

转发信息数据库(见图1-8-9)包含了相对IP当前本系统可用的路由信息，它是一个相当复杂的数据结构。尽管对其已采取了合理而有效的编排处理，但对它的访问仍然很慢，这也正是建立路由缓存的原因：利用已知的可用路由来提高路由查询速度。路由缓存中的信息，均取自转发信息数据库，并且是其中经常被使用的那一部分。

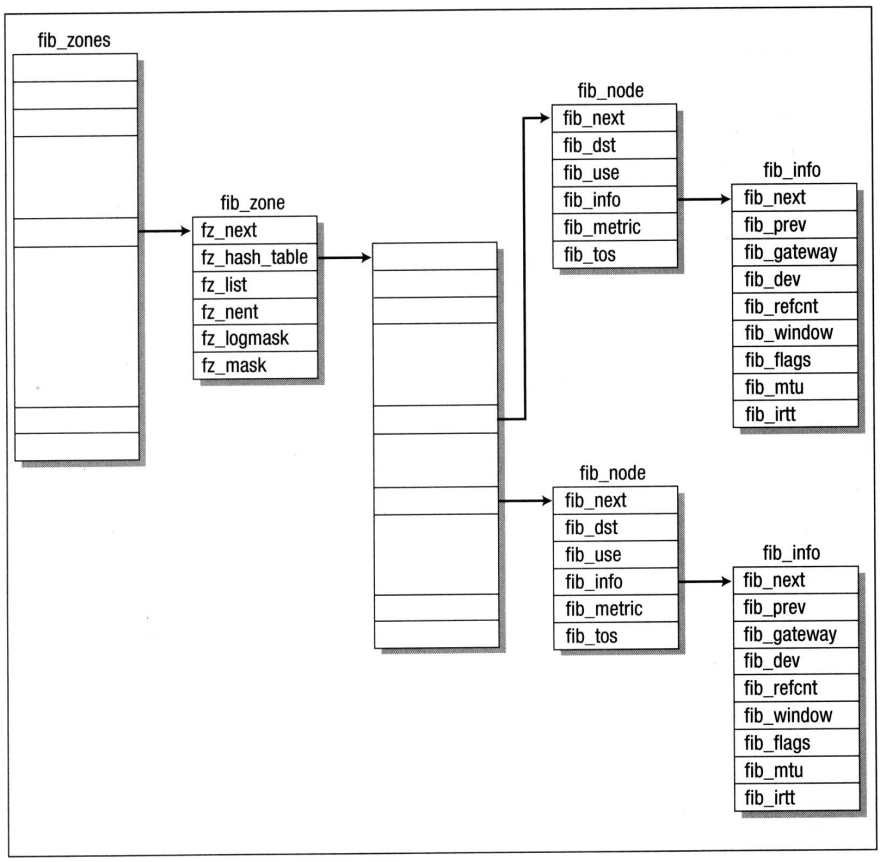


图1-8-9 转发信息数据库

由一个fib_zone数据结构来代表每一个IP子网，fib_zones哈希表中有所有这些fib_zone指针，其索引值取自IP子网掩码。每个fib_zone数据结构中都有fib_list项，所有发往同一子网的路由均由fib_list队列中的fib_node和fib_info数据结构所说明，若一个子网中的路由数过多，系统就会生成一张哈希表，以简化对fib_node数据结构的查找。

对于一个IP子网可能会有多个路由，这些路由将通过不同的几个网关。 IP路由层不允许通往同一子网的多个路由使用同一网关，换句话说，若有通往同一子网的多个路由，对每一个都使用不同的网关进行转发。有一个度量与每个路由相关联，用以指明路由的的优劣度，很重要的一种度量是“跳步(hop)”，它是IP报文到达目的子网前所经过的子网个数，度量值越高，该路由越差。

第9章　内核机制与模块

9.1　内核机制

本节讲述Linux内核提供的几种通用任务和机制，正是在它们的支持下，内核的其他部分才得以协调一致地工作。

9.1.1　Bottom Half控制

核态(kernel)下有很多时候系统为了进行一项工作而无法再干其他的事情，中断处理就是很好的例子，当有中断时，处理机停止当前的工作，由操作系统将中断转交给相应的设备驱动程序，然后等待，因此设备驱动程序应快速完成中断的处理，提高系统效率。然而还有另外一些工作，系统不必为它们停止当前的处理，因为可以稍后再进行这些工作，　Linux的Bottom Half控制程序正是为了处理这些工作而设计的。图 1-9-1给出了Bottom Half控制程序所用的内核数据结构，由图中可看出，一共有 32个不同的Bottom Half控制程序，同时有一个包含32个指针的bh_base向量，每个指针指向一个 Bottom Half控制例程。bh_active和bh_mask通过其位值分别指出安装了哪些控制程序和哪些控制程序是活跃的：如果 bh_mask第N位被置1，则表明bh_base中的第N个指针指向了一个 Bottom Half例程；如果bh_active第N位被置1，则表明一旦调度进程许可，立即调用第 N个Bottom Half例程。这些索引值都是静态设定的，如定时器Bottom Half控制器具有最高优先级 (索引值0)，控制台Bottom Half控制器优先级稍低(索引值 1)等等。典型的 Bottom Half控制例程总与一个任务表相关联，比如　Immediate Bottom Half控制程序负责immediate任务队列(tq-immediate)的处理，该队列中包含了需立即执行的任务。

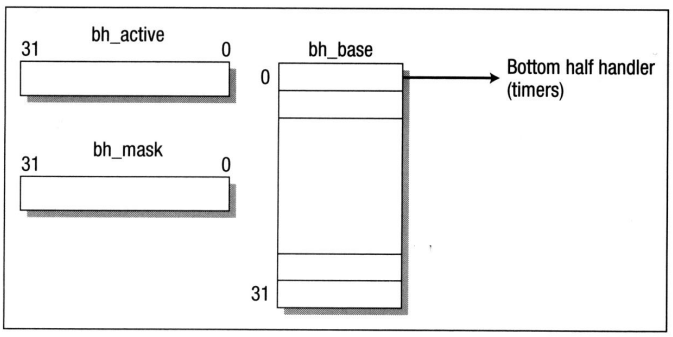


图1-9-1　Bottom Half控制数据结构

有一些内核的Bottom Half控制程序是由硬件设备所定义的，但另外一些则是通用的，如下所示：

TIMER	每次系统周期性定时器中断均被调用，以驱动内核定时器队列机制
CONSOLE	处理控制台消息
TQUEUE	处理tty消息
NET	处理通用网络运行
IMMEDIATE	为一些设备驱动程序设计的通用控制，用于将稍后进行的工作排队

一旦设备驱动程序或内核其他部分需要调度待执行工作，首先要加入工作到相应的系统队列中(例如定时器队列)，然后通知内核去执行某些 Bottom Half控制程序，这是通过设置bh_active中的相应位来实现的。如果驱动程序将某些工作加到了 immediate队列中，并希望执行Immediate Bottom Half以处理这些工作，则其会把 bh_active的第8位置1。在每次系统调用之后并尚未将控制器交给调用进程之前，都要检测 bh_active的各个位置，若发现某些位被置1，则调用相应的Bottom Half控制程序，检测顺序由第0位到第31位，调用完成之后bh_active中相应位清零。bh_active是暂时的，仅在两次调度进程调用之间有意义，对它的使用可避免无任何工作要做时盲目调Bottom Half控制程序。

9.1.2 任务队列

任务队列是内核用于延迟某些工作的一种方法。 Linux对于将工作排队稍后处理有一种通用机制。任务队列通常用于 Bottom Half控制程序的连接，当运行定时器队列 Bottom Half控制程序时则处理定时器队列。如图 1-9-2所示，任务队列是一种简单的数据结构，它包含了一个tq_struct数据结构的单向链表，每个 tq_struct数据结构中有一个例程地址和一个指向一些数据的指针，当处理任务队列中的各项时，将调用这个例程，并同时传递给它数据指针。

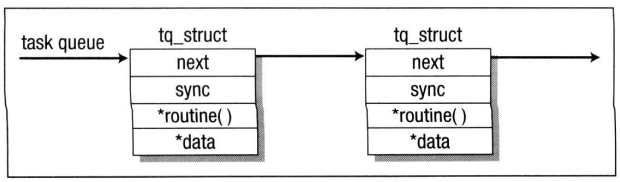


图1-9-2 任务队列

内核中的任何进程，例如设备驱动程序，都可以创建并使用任务队列，但其中有三种任务队列只能由内核进行创建和管理：

- timer 希望在下一个时钟节拍之后尽可能快地进行的工作使用本队列。每到一个时钟节拍，就检测本队列，看看其是否为空，若不空，则置定时器队列 Bottom Half控制程序为活跃的，当下一次运行调度进程时，则与其他 Bottom Half控制进程一起运行定时器队列 Bottom Half控制进程。不要把系统定时器与本队列混淆了，那是一个更为复杂的机制。
- immediate 调度进程运行活跃的Bottom Half控制程序时将同时处理本队列，从优先级上来看，Immediate Bottom Half控制程序低于定时器队列Bottom Half控制程序，因此它将在定时器任务处理之后执行。
- schedule 由调度进程直接运行，用于支持系统中的其他任务队列，从这点来讲，本队列的各个任务就是一个处理任务队列的例程。

每当处理完任务队列中的一项，就从队列中删除相应指针 (将其值置为NULL)。事实上，这种删除操作是一种原子(atomic)操作，不会被中断，这样队列中的每一项都依次调用其控制

例程。队列中的项通常是静态分配的数据，但对这些数据的删除并无由内存继承而来的回收机制。任务队列处理例程仅仅是把下一项加入表中，而正确释放掉所分配核心内存的工作是由这些任务自己来完成的。

9.1.3 定时器

操作系统有时需要安排将来的运行活动，因此需要提供一种机制，在该机制下，这些运行活动能够在相对准确的时间点上开始运行。任一个能支持操作系统的微处理器都必须支持可编程的内部定时器，该定时器将周期地中断处理器，这个定时器就是众所周知的系统时钟，它像一种节拍一样安排系统的各种活动。 Linux对时间有一种简单的观点：它从系统启动时开始以时钟节拍计时，所有的系统时间都基于这种计时方式，它的值可以通过全局变量 jiffies取到。

Linux有两种系统定时器，在某一系统时间同时被调用，但它们在实现上略有不同，图 1-9-3给出这两种机制。第一种，即老的定时器机制，有一个包含 32个指针的静态数据组和一个活跃定时器屏蔽码 (timer_active)，这些指针指向timer_struct数据结构，定时器程序与定时器表的连接是静态定义的，大多数定时器程序入口是在系统初始化时加入到定时器表中的；第二种，即新的定时器机制，使用了一个链表，表中的 timer_list数据结构以递增的超时数排序。

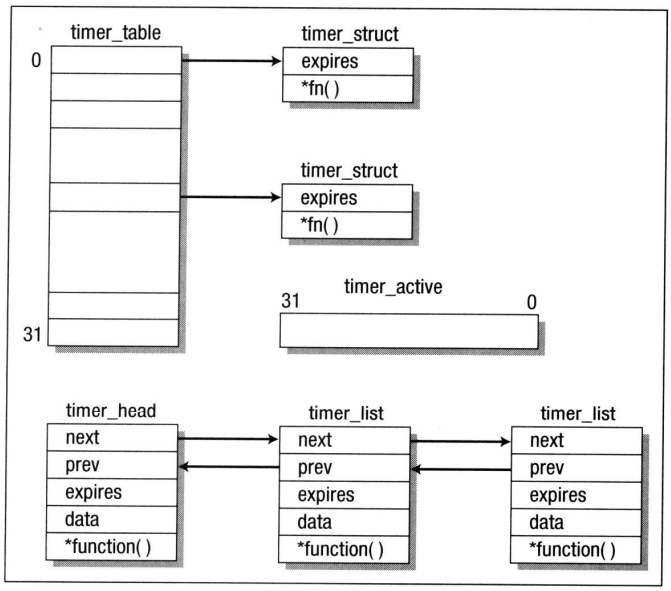


图1-9-3 系统定时器

两种机制都是用 jiffies值判断是否超时，因此如果一个定时器希望定时 5秒，那么它先要把5秒转化为jiffies的单位，然后把转换后的值加上当前系统时间以得到定时器超时的 jiffies值。由于每一系统时钟节拍都要把定时器 Bottom Half控制程序置为活跃，因此每当下次运行调度进程，都要处理定时器队列。定时器 Bottom Half控制程序可处理这两种系统定时器，对于老的系统定时器，它先检测 timer_active屏蔽码得到当前活跃的定时器，然后检测当前活跃的定时器是否超时，若是，则调用该定时器例程并把相应 timer_active中的位清零；对于新的系统

定时器，先检测timer_list链表数据中的入口，然后调用超时的定时器例程，并从链表中删除该项。新的定时器机制有一项优势：它允许向定时器例程传送一个参数。

9.1.4 等待队列

很多情况下处理器因等待某种系统资源而无法继续运行，例如：处理器需要一个描述目录的VFS索引节点，但该索引节点当前不在内存缓冲区中，这样处理器就必须先等到索引节点从磁盘中读到内存之后，才能继续运行。

对于这种等待的处理，Linux内核使用了一种简单的数据结构——等待队列(见图1-9-4)，其中包括一个指向task_struct的指针和一个指向队列中下一元素的指针。

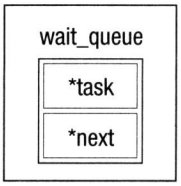


图1-9-4 等待队列

加入到等待队列中的进程可以是可中断，也可以是不可中断的。可中断进程在有定时器超时或等待信号的进程接收到了信号等事件发生时，可以被中断。通过进程状态参数值 (INTERRUPTIBLE或UNINTERRUPTIBLE)可判明该进程类型，由于调度进程中断了可中断进程的执行，而运行了另一进程，这样可中断进程将被挂起。

处理等待队列时，队列中每一个进程的状态都被置为 RUNNING，这时从运行队列中被移出的进程将重新进入运行队列，下次调度进程运行时，由于等待队列中的进程不需再等待了，因此它们将可以被调度运行。当处于等待队列中的一个进程准备运行时，首先要把自己从等待队列中移出。由于可以用等待队列实现对系统资源的并发访问，因此 Linux中用它来实现了信号量机制。

9.1.5 自旋锁

这是用于保护一个数据结构或一段代码的比较原始的方法，它限制了在一段时间内只允许有一个进程访问一块临界区。Linux中以一个整数域作为锁，来限制对数据结构的访问。每个要访问这些资源的进程必须首先将锁值由 0改为1，若锁的当前值已经是 1，则进程利用一个循环反复尝试对该锁的访问修改，直到成功。对内存区中锁的访问必须是原子操作：读锁值、检测锁值以及修改锁值的操作是不能被中断的。大多数 CPU体系结构是通过一条特殊的指令实现自旋锁(buzz lock)的，但也可以利用非缓存(uncached)主存来实现。

当占有临界区资源的进程使用完该资源之后，必须将锁值重置为 0，此时其他所有循环等待该锁的进程均可检测到其值为 0，但只有第一个检测到的进程有机会将锁值改为 1并访问临界区资源。

9.1.6 信号量

使用信号量(semaphore)是用来保护代码或数据结构的临界区资源。请不要忘记，只有核态下运行的进程才可以访问诸如描述文件目录的 VFS索引节点这样的临界区资源。如果允许一个进程去修改另一个进程正在使用的临界区资源，将是十分危险的，自旋锁可作为解决这种问题的一种方法，但自旋锁过于简单，使用这种方法会影响系统性能。在 Linux中，是用信号量来实现对临界区资源的并发访问的，未得到该资源的进程等待该资源被释放，并且这些等待进程将被挂起，其他进程照常运行。

Linux信号量数据结构包含如下一些信息：

count	跟踪记录要使用该资源的进程个数。其值大于零时，表示资源可用；值小于或等于零时，要使用该资源的进程必须等待。它的初始值为 1，这意味着同一时刻只有一个进程可以使用资源。希望使用该资源的进程将其值减1，用完后将其值加1
waking	正在等待该资源的进程数
wait queue	等待该资源的进程均放入此队列
lock	访问waking域所使用的自旋锁

假定一个信号量的count初始值为1，第一个访问该资源的进程将发现 count值大于0，于是将其减1，此时count值为0，这个进程现在拥有了受信号量保护的临界区资源；当进程访问完了，再把count值加1，此时最优的情况是没有其他进程要取得该资源的拥有权， Linux对信号量的实现在这种最优但也是最常见的情况下可以高效工作。

如果有另一个进程要访问已被占用的临界区资源，它也要把该信号量的 count值减1，这样count值已小于0(－1)，它已无法访问，只能等待资源被释放。Linux把等待进程置为睡眠状态，直到占用资源的进程释放掉资源再将其唤醒。等待进程会把自己放入信号量的等待队列中，然后循环检测waking域的值并调用调度进程，直到waking域变为非零为止。

而拥有临界资源的进程释放资源时，先检测 count值以得知是否有进程在等待该资源，然后把count值加1。在最优的情况下，此时 count值又回复到了初始值 1，资源拥有者进程把waking域值加1并唤醒信号量等待队列中的进程；被唤醒的进程由 waking值为1得知现在该资源可用，这样它将把 waking值减1，然后顺次执行。 Linux通过信号量中的lock域利用自旋锁机制实现了对waking域的访问保护。

9.2 模块

本节讲述Linux内核在需要的时候，是如何动态载入像文件系统这样的函数的。

Linux是一个单内核操作系统，也就是说它是一个独立的大程序，其所有的内核功能构件均可访问任一个内部数据结构和例程。对于这样的操作系统，一种选择是采用微内核结构，内核被分为若干独立的单位，各单位之间通过严格的通信机制相互访问。这样的话，若要向内核中加入新的构件，就必须利用设置进程，例如想加入一个未建立到内核中的 NCR 810 SCSI的设备驱动程序，就必须用设置进程重建一个新的内核；而在 Linux中可针对用户需要，动态地载入和卸载操作系统构件。 Linux模块是一些代码的集成，可以在启动系统后动态链接到内核的任一部分，当不再需要这些模块时，又可随时断开链接并将其删除。 Linux内核模块通常是一些设备驱动程序、伪设备驱动程序 (如网络驱动程序)或文件系统。

对于Linux的内核模块，可以用insmod或rmmod命令显式地载入或卸载，或是由内核在需要时调用内核守护程序(kerneld)进行载入和卸载。进行动态载入工作的代码非常有效，它将最小化内核大小并增加内核灵活性。当调试一个新内核时，模块也非常有用，通过对它的动态载入即可省去每次的重建和重启内核工作。当然，有利必有弊，使用模块将降低一些系统性能并消耗一部分内存空间，因为载入模块额外多出一些代码和数据结构，并会间接地降低访问内核资源的效率。

一旦Linux模块载入后，就与内核其他部分没什么区别了，它会拥有同样的权利和义务，换句话说，它也能像核心代码或设备驱动程序一样使内核崩溃。

为了使用内核资源，模块必须要先找到资源。假如一个模块希望调用内核内存分配例程kmalloc( )，由于模块创建时并不知道kmalloc( )在内存中的何处，所以在模块载入时，内核必须先调整该模块对kmalloc( )的引用，否则该模块无法正常工作。内核中维护了一张所有内核资源的符号表，因此它可以在模块载入时解决载入模块对内核资源的引用的问题。 Linux允许模块的栈操作，由此一个模块即可以使用其他模块所提供的服务。例如， VFAT文件系统模块就需要使用FAT文件系统提供的服务，因为 VFAT文件系统从某种意义来讲是 FAT文件系统的扩展。一个模块对另一个模块的服务或资源的使用与其对内核服务或资源的使用非常相似，不同的只是这些服务和资源从属于另一个模块而已。每载入一个模块，内核就会修改符号表，将该模块所有的服务和资源加入进去，这样当下一个模块载入后，即可访问已载入模块的服务。

当要卸载一个模块时，内核需要知道当前该模块是否被使用，并且还要能够通知该模块它将卸载，这样它就能释放掉所申请的所有系统资源。模块卸载后，内核从符号表中删除所有该模块所提供的资源和服务。

除了崩溃内核的可能性，模块还会带来另外一种危险：载入了一个与当前系统版本不同的模块会如何？若该模块以一个错误参数调用了系统例程，就会出问题，为防止这种情况发生，内核在载入模块前，会对该模块的版本号进行严格的检测。

9.2.1 模块载入

有两种载入模块的方法：一种是用 insmod命令手工载入，另一种方法更为灵活，是在需要时自动载入，这种方法也称为需求载入，当内核发现需要载入某个模块时，它会要求内核守护程序去载入相应的模块。

内核守护程序是一个拥有超级用户权限的一般用户进程，当它启动后 (系统启动时)会打开一个指向内核的内部进程间通信 (Inter-Process Communication, IPC)通道，内核用该通道通知内核守护程序进行各种操作。内核守护程序的主要工作是载入和卸载模块，它也做其他一些任务，如打开和关闭使用电话线的 PPP连接。内核守护程序并非亲自做这些工作，而是调用相应的程序(如insmod)来完成，它只是一个内核代理，自动地安排调度各项工作。

insmod工具在加载之前，先要打开欲加载的内核模块，需要时才被加载的模块保存在/lib/modules/kernel-version中，内核模块其实是一些链接的对象文件，与系统中其他的程序是一样的，只不过它们是作为重分配的映象被链接的，也就是说，它们并非从一特定地址开始运行。内核模块可以是 a.out或elf格式的对象文件， insmod要进行一次系统调用来查找内核导出符号，这些符号保存在符号名值对中。内核维护了一张模块表，模块表中第一个模块的数据结构内有内核导出符号表， module_list指针指向该符号表。只有特定的符号被加入到符号表中，并在编译和链接内核时创建，并非所有内核中的符号都导出到其模块上。“ request_irq ”就是一个符号，当一个驱动程序希望控制特定的系统中断时，就要调用该内核例程，通过查看/proc/ksyms文件或使用ksyms工具可以很方便地看到导出内核符号的名和值， ksyms工具能显示出所有的导出内核符号或仅仅被已载入模块所导出的符号。 insmod将模块读入虚存中，然后利用内核中的导出符号解决模块对内核进程的引用问题，解决方法是在内存中对模块映像进行修补：fnsmod把符号地址物理地写入模块的相应位置中。

解决了模块对导出内核符号的引用的问题之后， insmod通过系统调用为新的内核申请足

够的空间，内核即为该模块分配一个新的模块数据结构和足够的核心内存空间，并将其加到内核模块表尾。图1-9-5示出了载入两个模块(VFAT和FAT)后的内核模块表，图中未给出模块表中的第一个模块，该模块是一个伪模块，只用于保存内核导出符号表。可以用 lsmod命令列出所有已载入的模块及其相互的依赖关系，lsmod只是简单地重新组织一下/proc/modules文件，该文件是通过内核模块数据结构表创建的，它在内存中的地址被映射到了 insmod进程的地址空间中，便于该进程对它的访问。insmod把模块复制到为其分配的空间中，并重定位该模块，这样它就可以从内核地址上开始运行了。若一个模块在一个系统中需要载入两次时，为避免其两次载入拥有同一个地址，这种处理是必须的，由于这种重定位，则须用相应的地址对模块映像进行修补。

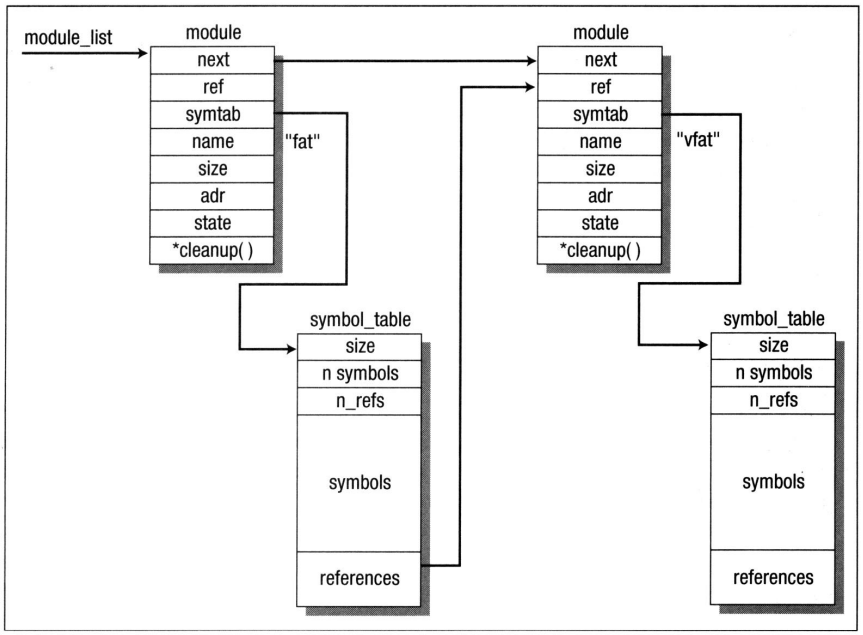


图1-9-5 内核模块表

新模块也要向内核导出符号，insmod将会为这些模块映像建立一张表，每一个内核模块必须包括模块初始化和清除例程，对于未导出符号的模块，insmod必须知道其地址以便告知内核。若一切正常，insmod即开始初始化模块，并通过系统调用将模块的初始化和清除例程地址交给内核。

新模块加入内核后，必须刷新内核字符并修改正在被其使用的模块。被其他模块引用的模块应维护一张引用表，该表在它们的符号表尾部，并可通过 module数据结构中的指针访问该表。从图1-9-5中可看出VFAT文件系统模块需要使用FAT文件系统模块，因此FAT模块中包含有一个对VFAT模块的引用，该引用是在载入 VFAT模块载入后加上的。内核会调用模块的初始化例程，若调用成功，将继续安装该模块，模块的清除例程地址保存在其 module数据结构中，核心卸载模块时将调用该例程，最后，置模块状态为 RUNNING。

9.2.2 模块卸载

可以用rmmod命令卸载模块，但对于需要时载入类型的模块，当其不再需要时，会由

kerneld自动将其从系统中删除。每次空闲定时器超时， kerneld都会利用系统调用来要求将所有当前未被使用的需要时载入类型的模块删除，定时器的值是在启动 kerneld时设定的。例如，如果对一个已mount的iso9660 CD ROM进行了unmount操作，则iso 9660模块不久即会被删除。

当一个模块正被其他内核构件所使用，则不能将其卸载。例如，当 mount了一个或多个VFAT文件系统之后，是无法卸载VFAT模块的，看一下lsmod的输入，会发现每一个模块有一与其相关的计数，例如：

```
Module:         #pages:  Used by:
msdos               5                 1
vfat                4                 1 (autoclean)
fat                 6    [vfat msdos] 2 (autoclean)
```

该计数是针对使用该模块的内核实体的。上例中， vfat和msdos模块都要使用fat模块，因此fat模块的计数是2；由于各有一个mount的文件系统使用vfat和msdos，由此它们的计数是1，若再载入一个VFAT文件系统， vfat模块的计数就变成2。模块在其映像的第1个longword中保存其计数值。

计数域是轻度重载的，因为该域也用以保存 AUTOCLEAN和VISITED标志，这两种标志都是指需要时载入类型的模块。有其他系统组件使用某个模块时，其标志就为 VISITED，每当kerneld通知系统删除不再使用的模块时，系统都要搜索所有模块以找到可能的删除模块，这样的查找结果是那状态为 RUNNING并且标志为 AUTOCLEAN的模块，这些模块中VISITED标志已清除的模块被删除，其他的模块则清除它们的 VISITED标志。

假定一个模块可被卸载，则会调用它的清除例程以释放其所占用的内核资源，它的module数据结构标记为DELETED并从内核模块链中将其删除，其他所有被该模块使用的模块都要修改它们的引用表，表明不再被它使用，还要释放掉为它分配的内存。

第10章 处 理 器

Linux 运行在若干处理器上，本章简要描述它们。

10.1 X86

TBD

10.2 ARM

ARM处理器实现了一种低功耗、高性能的 32位RISC体系结构，它被广泛使用于嵌入式设备如手机和 PDA(个人式数字化助手)中，它具有 31个32位寄存器，其中 16个在任何模式中均为可见的。它的指令是简单的装载存储指令 (从存储器中装载一个值、执行一个操作、把结果存储进存储器)。它的一个有趣特征是每个指令都是条件式的，例如：你可以测试一个寄存器的值，在此过程中，可以执行你想执行的任何指令，直到在同样条件下测试下一个值为止。另一个有趣的特征是当你装载值时，可以对值执行算术和移位操作，这可以在几种模式下进行，包括系统模式(可从用户模式通过SWI，即软中断进入到系统模式)。

它是一个可合成的核， ARM公司本身不生产处理器，相反 ARM的合伙人(例如Intel公司或LSI公司)在硅片上实现 ARM体系结构，通过一个公共处理器接口，它允许其他处理器紧紧耦合在一起，它有几个存储管理单元的变种，范围从简单的存储器保护格式到复杂的分页层次。

10.3 Alpha AXP处理器

Alpha AXP体系结构是以速度为考虑因素而设计的 64位存取RISC体系结构，所有的寄存器都是64位的，包括32个整数寄存器和 32个浮点寄存器。整数寄存器 31和浮点寄存器31都表示空操作，从中将读出一个 0值，向它们写则不会产生任何影响。所有的指令都是 32位定长，内存操作只有读或写，这种体系结构允许不同的实现方法，只要这种实现方法符合这种体系结构。

指令不能对内存中的数据直接进行操作，所有的数据操作都是在寄存器中进行的。因此，若想增加一个在内存中的计数值，首先把此计数值读入寄存器，然后在寄存器中对其进行修改，最后再写回内存，通过对寄存器或内存的读写指令，各指令间进行数据交互。 Alpha AXP的一个有趣的特征是这些指令能产生标志，例如检验两个寄存器的值是否相等，结果不存入处理器的静态寄存器，而放在第 3个寄存器中。一开始这看起来有些奇怪，但这样做消除了对静态寄存器的依赖性，意味着可以更加容易地建立一个在每一个周期内执行多条指令的CPU，彼此没有联系的寄存器指令并不需要互相等待执行，这与只有一个静态寄存器是不同的。不允许对内存的直接操作和数量庞大的寄存器对建立多指令的执行也是很有帮

助的。

Alpha AXP体系结构使用一组称为特权体系结构库的代码 (Privileged Architecture Library code，PALcode)。PALcode对操作系统、对 Alpha AXP体系结构的CPU实现方法和系统硬件，都是特定的，这些子例程提供了上下文切换、中断、异常和内存管理的操作系统原语，这些子例程能被硬件或被CALL_PAL指令调用。PALcode用标准的Alpha AXP汇编程序编写，包括了对一些实现方法的特殊扩充来提供对下层的硬件函数的直接访问，例如内部处理器寄存器。PALcode运行于PAL模式：一种能制止一些系统事件发生和允许 PALcode完全控制物理系统硬件的特权模式。

第11章　Linux内核源代码

本章讲述在Linux内核源码中，应该从何处开始查找特定的内核函数。

本书并不要求读者具有C语言编程能力，也不要求读者有一份可参阅的Linux内核源码，事实上，通过查看内核源码可以在一定深度上理解Linux操作系统，同时这也是一个很好的实践机会。本章给出了对内核源码的概览：它们是如何编排的以及从何处开始查找特定代码。

11.1　怎样得到Linux内核源码

所有主要的Linux系统(Craftworks、Debian、Slackware、Red Hat等等)都包含有内核源码，通常所安装的Linux系统都是通过这些源码创建的。由于Linux总是不断更新，因此用户所安装的Linux可能已过时，不过从附录A所列的站点上可得到最新的源码，所有这些站点地址都可在ftp://ftp.cs.helsinki.fi上查到。

Linux内核源码的版本号表示方法非常简单：所有偶数版(如2.0.30)都是已发行的稳定版；所有奇数版(如2.1.42)都是测试版，本书是基于2.0.30版撰写的。测试版包含所有的新特征，并支持所有的新设备，虽然测试版并不稳定，并且可能提供了一些用户不想要的东西，但对于Linux与用户沟通而言，测试新的内核是很重要的。不过请注意，在尝试非产品型的测试版之前，最好先完全备份系统。

对内核源码的修改是作为patch文件出现的，patch工具提供了一组对源码文件的编辑。例如，若想把2.0.29源码升级为2.0.30版，则要使用patch文件来完成对源码的编辑，操作如下：

```
$ cd /usr/src/linux
$ patch -p1 < patch-2.0.30
```

这样做可以避免对所有源码文件的拷贝。在http://www.linuxhq.com站点上可找到很好的内核源码的patch。

11.2　内核源码的编排

在源码目录树的最顶端(/usr/src/linux)可看到如下一些目录：

- arch　arch子目录包含所有的特定体系结构的内核源码，它的子目录分别对应着一种Linux所支持的体系结构，例如i386和alpha。
- include　include子目录包含大部分的编译内核源码所需文件。
- init　此目录下包含了内核的初始化代码，由此可以很好地开始了解内核是如何工作的。
- mm　此目录下包含了所有内存管理代码，特定体系结构的内存管理代码在arch/*/mm目录下。
- drivers　此目录下包含了系统所有的设备驱动程序，其下子目录各针对不同的设备驱动程序类。
- ipc　此目录下包含了内核的内部进程通信代码。
- modules　此目录只是用来保存创建的模块。

- fs 所有文件系统代码，其下子目录各针对不同的系统所支持的文件系统。
- kernel 内核主代码，特定体系结构内核代码保存在 arch/*/kernel中。
- net 内核的网络代码。
- lib 此目录包含内核库代码，特定体系结构的库代码保存在 arch/*/lib目录下。
- scripts 此目录包含了内核设置时用到的脚本。

11.3 从何处看起

像Linux这样复杂的大程序，探究起来使人迷茫，这就像一个找不出头绪的大线团。要查看内核的某一部分通常会被引向许多其他的相关文件，最后甚至忘记了最初的动机。下面给出了一些提示，根据这些提示，对于给定的内容即可找到最好的开始阅读代码部分。

1. 系统启动和初使化

在基于Intel的系统中，通常先运行loadlin.exe或LILO，由这两个程序将内核载入内存并启动内核，之后便由内核控制系统。在 arch/i386/kernel/head.s中可找到这一部分， head.s先进行一些特定体系结构的安装，然后跳转到 init/main.c中的main( )例程。

2. 内存管理

有关内存管理的代码大部分都在 mm中，但与特定体系结构相关的部分则保存在 arch/*/mm中，内存缺页处理代码在 mm/memory.c中，内存映射和页缓冲代码在 mm/filemap.c中，实现缓冲区缓存部分代码在 mm/buffer. c中，页交换代码在 mm/swap_state. c和mm/swapfile. c中。

3. 内核

大部分通用内核代码在 kernel中，与特定体系结构相关的代码在 arch/*/kernel中，调度进程代码在kernel/sched.c中，创建子进程代码在 kernel/fork.c中，Bottom Half控制程序代码在include/linux/interrupt.h中，task_struct数据结构的定义在include/linux/sched.h中。

4. PCI

PCI伪驱动程序在drivers/pci/pci.c中，系统全局定义在include/linux/pci.h中。每种体系结构都有其特定的PCI BIOS代码，如Alpha Axp的代码在arch/alpha/kernel/bios32.c中。

5. 内部进程间通信

所有相关代码都在 ipc中，所有的 System V IPC对象都有一个 ipc_perm数据结构，在include/linux/ipc.h中有该数据结构的定义。 System V的消息机制代码在ipc/msg.c中，共享内存代码在ipc/shm.c中，信号量代码在ipc/sem.c中，管道代码在ipc/pipe.c中。

6. 中断处理

内核的中断处理代码几乎都与特定微处理器相关。Intel的中断处理代码在arch/i386/kernel/irq.c中，并且定义在include/asm-i386/irq.h中。

7. 设备驱动程序

大部分的Linux内核源码行在设备驱动程序中，所有设备驱动程序代码在 drivers中，并分为如下几类：

- /block 块设备驱动程序(如ide.c)。若要了解其初始化过程，参看 drivers/block/genhd.c中的device_setup( )函数，该函数不仅能初始化硬盘，也可以初始化网络。块设备包括 IDE和SCSI设备。

- /char 字符设备驱动程序，例如ttys、串口和鼠标。
- /cdrom Linux的所有CDROM代码。在此可找到特定的CDROM设备(如Soundblaster CD ROM)，请注意，ide CD驱动程序在drivers/block下的ide-cd.c中，而SCSI CD驱动程序在drivers/scsi中的scsi.c中。
- /pci PCI伪驱动程序代码，由此可了解 PCI子系统是如何映射和初始化的， arch/alpha /kernel/bios32.c中的Alpha PCI固化代码也值得一看。
- /scsi 所有的SCSI代码，以及Linux所支持的所有scsi设备驱动程序代码。
- /net 所有网络设备驱动程序代码。
- /sound 所有声卡驱动程序代码。

8. 文件系统

EXT2文件系统的代码都在 fs/ext2/目录下，其数据结构定义在 include/linux/ex2_fs.h、ext2_fs_i.h和ext2_fs_sb.h中，虚文件系统(Virtual File System)数据结构在include/linux/fs.h中，代码在fs/*中，缓冲区缓存代码在fs/buffer.c中。

9. 网络

网络部分代码在 net中，其大部分的 include文件在include/net中， BSD套接字代码在net/socket.c中， IP版本4 INET套接字代码在net/ipv4/af_inet.c中，常用的协议支持代码 (包括sk_buff控制例程)在net/core中， TCP/IP网络代码在 net/ipv4中，而网络设备驱动程序在drivers/net中。

10. 模块

内核模块代码一部分在kernel中，另一部分在模块包中，内核代码都在kernel/mcdules.c中，其数据结构和内核守护程序 kerneld消息分别在include/linux/module.h和include/ linux/kerneld.h中， ELF对象文件的结构定义在include/linux/elf.h中。

第12章　Linux 数据结构

本章列出了Linux中用到的且在本书中出现过的主要数据结构。

block_dev_struct

用该数据结构注册对缓冲区缓存可用的块设备，它们统一保存在 blk_dev向量中。

```
struct blk_dev_struct {
    void (*request_fn)(void);
    struct request * current_request;
    struct request   plug;
    struct tq_struct plug_tq;
};
```

buffer_head

本数据结构包含了缓冲区缓存中一块缓冲区的信息

```
/* bh state bits */
#define BH_Uptodate  0   /* 1 if the buffer contains
valid data      */
#define BH_Dirty     1   /* 1 if the buffer is dirty
*/
#define BH_Lock      2   /* 1 if the buffer is locked
*/
#define BH_Req       3   /* 0 if the buffer has been invalidated     */
#define BH_Touched   4   /* 1 if the buffer has been touched (aging) */
#define BH_Has_aged  5   /* 1 if the buffer has been aged (aging)    */
#define BH_Protected 6   /* 1 if the buffer is protected             */
#define BH_FreeOnIO  7   /* 1 to discard the buffer_head after IO    */

struct buffer_head {
  /* First cache line: */
  unsigned long      b_blocknr;    /* block number                    */
  kdev_t             b_dev;        /* device (B_FREE = free)          */
  kdev_t             b_rdev;       /* Real device                     */
  unsigned long      b_rsector;    /* Real buffer location on disk    */
  struct buffer_head *b_next;      /* Hash queue list                 */
  struct buffer_head *b_this_page; /* circular list of buffers in one
                                      page                            */

  /* Second cache line: */
  unsigned long      b_state;      /* buffer state bitmap (above)     */
  struct buffer_head *b_next_free;
  unsigned int       b_count;      /* users using this block          */
  unsigned long      b_size;       /* block size                      */

  /* Non-performance-critical data follows. */
  char               *b_data;      /* pointer to data block           */
  unsigned int       b_list;       /* List that this buffer appears   */
```

```
    unsigned long      b_flushtime;  /* Time when this (dirty) buffer
                                      * should be written             */
    unsigned long      b_lru_time;   /* Time when this buffer was
                                      * last used.                     */
    struct wait_queue  *b_wait;
    struct buffer_head *b_prev;      /* doubly linked hash list       */
    struct buffer_head *b_prev_free; /* doubly linked list of buffers */
    struct buffer_head *b_reqnext;   /* request queue                 */
};
```

device

每一个系统中的网络设备均由一个device数据结构所代表。

```
struct device
{

  /*
   * This is the first field of the "visible" part of this structure
   * (i.e. as seen by users in the "Space.c" file).  It is the name
   * the interface.
   */
  char                  *name;

  /* I/O specific fields                                          */
  unsigned long         rmem_end;       /* shmem "recv" end     */
  unsigned long         rmem_start;     /* shmem "recv" start   */
  unsigned long         mem_end;        /* shared mem end       */
  unsigned long         mem_start;      /* shared mem start     */
  unsigned long         base_addr;      /* device I/O address   */
  unsigned char         irq;            /* device IRQ number    */

  /* Low-level status flags. */
  volatile unsigned char  start,        /* start an operation   */
                        interrupt;      /* interrupt arrived    */
  unsigned long         tbusy;          /* transmitter busy     */
  struct device         *next;

  /* The device initialization function. Called only once.        */
  int                   (*init)(struct device *dev);

  /* Some hardware also needs these fields, but they are not part of
     the usual set specified in Space.c. */
  unsigned char         if_port;        /* Selectable AUI,TP,   */
  unsigned char         dma;            /* DMA channel          */

  struct enet_statistics* (*get_stats)(struct device *dev);

  /*
   * This marks the end of the "visible" part of the structure. All
   * fields hereafter are internal to the system, and may change at
   * will (read: may be cleaned up at will).
```

```
 */

/* These may be needed for future network-power-down code.       */
unsigned long           trans_start;    /* Time (jiffies) of
                                           last transmit        */
unsigned long           last_rx;        /* Time of last Rx      */
unsigned short          flags;          /* interface flags (BSD)*/
unsigned short          family;         /* address family ID    */
unsigned short          metric;         /* routing metric       */
unsigned short          mtu;            /* MTU value            */
unsigned short          type;           /* hardware type        */
unsigned short          hard_header_len; /* hardware hdr len    */
void                    *priv;          /* private data         */
 /* Interface address info. */
 unsigned char          broadcast[MAX_ADDR_LEN];
 unsigned char          pad;
 unsigned char          dev_addr[MAX_ADDR_LEN];
 unsigned char          addr_len;       /* hardware addr len    */
 unsigned long          pa_addr;        /* protocol address     */
 unsigned long          pa_brdaddr;     /* protocol broadcast addr*/
 unsigned long          pa_dstaddr;     /* protocol P-P other addr*/
 unsigned long          pa_mask;        /* protocol netmask     */
 unsigned short         pa_alen;        /* protocol address len */

 struct dev_mc_list     *mc_list;       /* M'cast mac addrs     */
 int                    mc_count;       /* No installed mcasts  */

 struct ip_mc_list      *ip_mc_list;    /* IP m'cast filter chain */
 __u32                  tx_queue_len;   /* Max frames per queue   */

 /* For load balancing driver pair support */
 unsigned long          pkt_queue;      /* Packets queued       */
 struct device          *slave;         /* Slave device         */
 struct net_alias_info  *alias_info;    /* main dev alias info  */
 struct net_alias       *my_alias;      /* alias devs           */

 /* Pointer to the interface buffers. */
 struct sk_buff_head    buffs[DEV_NUMBUFFS];

 /* Pointers to interface service routines. */
 int                    (*open)(struct device *dev);
 int                    (*stop)(struct device *dev);
 int                    (*hard_start_xmit) (struct sk_buff *skb,
                                            struct device *dev);
 int                    (*hard_header) (struct sk_buff *skb,
                                        struct device *dev,
                                        unsigned short type,
                                        void *daddr,
                                        void *saddr,
                                        unsigned len);
```

```
    int                     (*rebuild_header)(void *eth,
                                              struct device *dev,
                                              unsigned long raddr,
                                              struct sk_buff *skb);
    void                    (*set_multicast_list)(struct device *dev);
    int                     (*set_mac_address)(struct device *dev,
                                              void *addr);
    int                     (*do_ioctl)(struct device *dev,
                                              struct ifreq *ifr,
                                              int cmd);
    int                     (*set_config)(struct device *dev,
                                              struct ifmap *map);
    void                    (*header_cache_bind)(struct hh_cache **hhp,
                                              struct device *dev,
                                              unsigned short htype,
                                              __u32 daddr);
    void                    (*header_cache_update)(struct hh_cache *hh,
                                              struct device *dev,
                                              unsigned char *  haddr);
    int                     (*change_mtu)(struct device *dev,
                                              int new_mtu);
    struct iw_statistics*   (*get_wireless_stats)(struct device *dev);
};
```

device_struct

该数据结构用于注册字符设备和块设备（包含了设备名和该设备所支持的文件操作），chrdevs和blkdevs向量中的每一个正确成员均指代一个字符设备或块设备。

```
struct device_struct {
    const char * name;
    struct file_operations * fops;
};
```

file

对应于每一个打开的文件。

```
struct file {
  mode_t f_mode;
  loff_t f_pos;
  unsigned short f_flags;
  unsigned short f_count;
  unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
  struct file *f_next, *f_prev;
  int f_owner;          /* pid or -pgrp where SIGIO should be sent */
  struct inode * f_inode;
  struct file_operations * f_op;
  unsigned long f_version;
  void *private_data;  /* needed for tty driver, and maybe others */
};
```

files_struct

对应于每一个进程所打开的文件。

```
struct files_struct {
  int count;
  fd_set close_on_exec;
  fd_set open_fds;
  struct file * fd[NR_OPEN];
};
```

fs_struct

```
struct fs_struct {
  int count;
  unsigned short umask;
  struct inode * root, * pwd;
};
```

gendisk

该数据结构保存有一个硬盘的信息。

```
struct hd_struct {
    long start_sect;
    long nr_sects;
};

struct gendisk {
    int major;                  /* major number of driver */
    const char *major_name;     /* name of major driver */
    int minor_shift;            /* number of times minor is shifted to
                                   get real minor */
    int max_p;                  /* maximum partitions per device */
    int max_nr;                 /* maximum number of real devices */

    void (*init)(struct gendisk *);
                                /* Initialization called before we
                                   do our thing */
    struct hd_struct *part;     /* partition table */
    int *sizes;                 /* device size in blocks, copied to
                                   blk_size[] */
    int nr_real;                /* number of real devices */

    void *real_devices;         /* internal use */
    struct gendisk *next;
};
```

inode

该数据结构保存有硬盘上的一个文件或目录信息。

```
struct inode {
    kdev_t                      i_dev;
    unsigned long               i_ino;
    umode_t                     i_mode;
    nlink_t                     i_nlink;
    uid_t                       i_uid;
```

```
    gid_t                       i_gid;
    kdev_t                      i_rdev;
    off_t                       i_size;
    time_t                      i_atime;
    time_t                      i_mtime;
    time_t                      i_ctime;
    unsigned long               i_blksize;
    unsigned long               i_blocks;
    unsigned long               i_version;
    unsigned long               i_nrpages;
    struct semaphore            i_sem;
    struct inode_operations     *i_op;
    struct super_block          *i_sb;
    struct wait_queue           *i_wait;
    struct file_lock            *i_flock;
    struct vm_area_struct       *i_mmap;
    struct page                 *i_pages;
    struct dquot                *i_dquot[MAXQUOTAS];
    struct inode                *i_next, *i_prev;
    struct inode                *i_hash_next, *i_hash_prev;
    struct inode                *i_bound_to, *i_bound_by;
    struct inode                *i_mount;
    unsigned short              i_count;
    unsigned short              i_flags;
    unsigned char               i_lock;
    unsigned char               i_dirt;
    unsigned char               i_pipe;
    unsigned char               i_sock;
    unsigned char               i_seek;
    unsigned char               i_update;
    unsigned short              i_writecount;
    union {
        struct pipe_inode_info  pipe_i;
        struct minix_inode_info minix_i;
        struct ext_inode_info   ext_i;
        struct ext2_inode_info  ext2_i;
        struct hpfs_inode_info  hpfs_i;
        struct msdos_inode_info msdos_i;
        struct umsdos_inode_info umsdos_i;
        struct iso_inode_info   isofs_i;
        struct nfs_inode_info   nfs_i;
        struct xiafs_inode_info xiafs_i;
        struct sysv_inode_info  sysv_i;
        struct affs_inode_info  affs_i;
        struct ufs_inode_info   ufs_i;
        struct socket           socket_i;
        void                    *generic_ip;
    } u;
};
```

ipc_perm

该数据结构描述了一个system V IPC对象的访问许可。

```
struct ipc_perm
{
  key_t  key;
  ushort uid;   /* owner euid and egid */
  ushort gid;
  ushort cuid;  /* creator euid and egid */
  ushort cgid;
  ushort mode;  /* access modes see mode flags below */
  ushort seq;   /* sequence number */
};
```

irqaction

该数据结构描述系统中断处理器。

```
struct irqaction {
  void (*handler)(int, void *, struct pt_regs *);
  unsigned long flags;
  unsigned long mask;
  const char *name;
  void *dev_id;
  struct irqaction *next;
};
```

linux_binfmt

用于指代每一种Linux所能理解的二进制文件格式。

```
struct linux_binfmt {
  struct linux_binfmt * next;
  long *use_count;
  int (*load_binary)(struct linux_binprm *, struct  pt_regs * regs);
  int (*load_shlib)(int fd);
  int (*core_dump)(long signr, struct pt_regs * regs);
};
```

mem_map_t

该数据结构包含有内存中物理页信息。

```
typedef struct page {
  /* these must be first (free area handling) */
  struct page        *next;
  struct page        *prev;
  struct inode       *inode;
  unsigned long      offset;
  struct page        *next_hash;
  atomic_t           count;
  unsigned           flags;     /* atomic flags, some possibly
                                   updated asynchronously */
  unsigned           dirty:16,
                     age:8;
```

```
    struct wait_queue  *wait;
    struct page        *prev_hash;
    struct buffer_head *buffers;
    unsigned long      swap_unlock_entry;
    unsigned long      map_nr;     /* page->map_nr == page - mem_map */
  } mem_map_t;
```

mm_struct

该数据结构描述了一个任务或进程的虚存。

```
  struct mm_struct {
    int count;
    pgd_t * pgd;
    unsigned long context;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack, start_mmap;
    unsigned long arg_start, arg_end, env_start, env_end;
    unsigned long rss, total_vm, locked_vm;
    unsigned long def_flags;
    struct vm_area_struct * mmap;
    struct vm_area_struct * mmap_avl;
    struct semaphore mmap_sem;
  };
```

pci_bus

每个pci_bus数据结构指代一个系统的PCI总线。

```
  struct pci_bus {
    struct pci_bus  *parent;     /* parent bus this bridge is on */
    struct pci_bus  *children;   /* chain of P2P bridges on this bus */
    struct pci_bus  *next;       /* chain of all PCI buses */

    struct pci_dev  *self;       /* bridge device as seen by parent */
    struct pci_dev  *devices;    /* devices behind this bridge */

    void    *sysdata;            /* hook for sys-specific extension */

    unsigned char  number;       /* bus number */
    unsigned char  primary;      /* number of primary bridge */
    unsigned char  secondary;    /* number of secondary bridge */
    unsigned char  subordinate;  /* max number of subordinate buses */
  };
```

pci_dev

每一个pci_dev数据结构指代一个系统PCI设备（包括PCI-PCI桥和PCI-ISA桥）。

```
  /*
   * There is one pci_dev structure for each slot-number/function-number
   * combination:
   */
  struct pci_dev {
    struct pci_bus  *bus;       /* bus this device is on */
```

```
    struct pci_dev  *sibling;  /* next device on this bus */
    struct pci_dev  *next;     /* chain of all devices */

    void    *sysdata;          /* hook for sys-specific extension */

    unsigned int  devfn;       /* encoded device & function index */
    unsigned short  vendor;
    unsigned short  device;
    unsigned int  class;       /* 3 bytes: (base,sub,prog-if) */
    unsigned int  master : 1;  /* set if device is master capable */
    /*
     * In theory, the irq level can be read from configuration
     * space and all would be fine.  However, old PCI chips don't
     * support these registers and return 0 instead.  For example,
     * the Vision864-P rev 0 chip can uses INTA, but returns 0 in
     * the interrupt line and pin registers.  pci_init()
     * initializes this field with the value at PCI_INTERRUPT_LINE
     * and it is the job of pcibios_fixup() to change it if
     * necessary.  The field must not be 0 unless the device
     * cannot generate interrupts at all.
     */
    unsigned char  irq;        /* irq generated by this device */
};
```

request

该数据结构用于向系统中的块设备发申请。

```
struct request {
    volatile int rq_status;
#define RQ_INACTIVE             (-1)
#define RQ_ACTIVE               1
#define RQ_SCSI_BUSY            0xffff
#define RQ_SCSI_DONE            0xfffe
#define RQ_SCSI_DISCONNECTING   0xffe0
    kdev_t rq_dev;
    int cmd;        /* READ or WRITE */
    int errors;
    unsigned long sector;
    unsigned long nr_sectors;
    unsigned long current_nr_sectors;
    char * buffer;
    struct semaphore * sem;
    struct buffer_head * bh;
    struct buffer_head * bhtail;
    struct request * next;
};
```

rtable

每个rtable数据结构包含了将报文发往一个IP主机的信息，在IP路由缓存中用到该数据。

```
struct rtable
{
```

```
    struct rtable     *rt_next;
    __u32             rt_dst;
    __u32             rt_src;
    __u32             rt_gateway;
    atomic_t          rt_refcnt;
    atomic_t          rt_use;
    unsigned long     rt_window;
    atomic_t          rt_lastuse;
    struct hh_cache   *rt_hh;
    struct device     *rt_dev;
    unsigned short    rt_flags;
    unsigned short    rt_mtu;
    unsigned short    rt_irtt;
    unsigned char     rt_tos;
};
```

semaphore

该数据结构用于保护对临界区数据结构和代码的访问。

```
struct semaphore {
    int count;
    int waking;
    int lock ;                  /* to make waking testing atomic */
    struct wait_queue *wait;
};
```

sk_buff

当在协议层间传输数据时，用该数据结构描述数据信息。

```
struct sk_buff
{
  struct sk_buff      *next;       /* Next buffer in list                  */
  struct sk_buff      *prev;       /* Previous buffer in list              */
  struct sk_buff_head *list;       /* List we are on                       */
  int                 magic_debug_cookie;
  struct sk_buff      *link3;      /* Link for IP protocol level buffer chains
*/
  struct sock         *sk;         /* Socket we are owned by               */
  unsigned long       when;        /* used to compute rtt's                */
  struct timeval      stamp;       /* Time we arrived                      */
  struct device       *dev;        /* Device we arrived on/are leaving by  */
  union
  {
      struct tcphdr   *th;
      struct ethhdr   *eth;
      struct iphdr    *iph;
      struct udphdr   *uh;
      unsigned char   *raw;
      /* for passing file handles in a unix domain socket */
      void            *filp;
  } h;
```

```
    union
    {
        /* As yet incomplete physical layer views */
        unsigned char   *raw;
        struct ethhdr   *ethernet;
    } mac;

    struct iphdr        *ip_hdr;     /* For IPPROTO_RAW                        */
    unsigned long       len;         /* Length of actual data                  */
    unsigned long       csum;        /* Checksum                               */
    __u32               saddr;       /* IP source address                      */
    __u32               daddr;       /* IP target address                      */
    __u32               raddr;       /* IP next hop address                    */
    __u32               seq;         /* TCP sequence number                    */
    __u32               end_seq;     /* seq [+ fin] [+ syn] + datalen          */
    __u32               ack_seq;     /* TCP ack sequence number                */
    unsigned char       proto_priv[16];
    volatile char       acked,       /* Are we acked ?                         */
                        used,        /* Are we in use ?                        */
                        free,        /* How to free this buffer                */
                        arp;         /* Has IP/ARP resolution finished         */
    unsigned char       tries,       /* Times tried                            */
                        lock,        /* Are we locked ?                        */
                        localroute,  /* Local routing asserted for this frame */
                        pkt_type,    /* Packet class                           */
                        pkt_bridged, /* Tracker for bridging                   */
                        ip_summed;   /* Driver fed us an IP checksum           */
#define PACKET_HOST         0        /* To us
*/
#define PACKET_BROADCAST    1        /* To all
*/
#define PACKET_MULTICAST    2        /* To group
*/
#define PACKET_OTHERHOST    3        /* To someone else
*/
    unsigned short      users;       /* User count - see datagram.c,tcp.c      */
    unsigned short      protocol;    /* Packet protocol from driver.           */
    unsigned int        truesize;    /* Buffer size                            */
    atomic_t            count;       /* reference count                        */
    struct sk_buff      *data_skb;   /* Link to the actual data skb            */
    unsigned char       *head;       /* Head of buffer                         */
    unsigned char       *data;       /* Data head pointer                      */
    unsigned char       *tail;       /* Tail pointer                           */
    unsigned char       *end;        /* End pointer                            */
    void                (*destructor)(struct sk_buff *); /* Destruct function */
    __u16               redirport;   /* Redirect port                          */
};
```

sock

每个sock数据结构保存有关于BSD套接字的特定协议信息。

```
struct sock
{
  /* This must be first. */
  struct sock             *sklist_next;
  struct sock             *sklist_prev;

  struct options          *opt;
  atomic_t                wmem_alloc;
  atomic_t                rmem_alloc;
  unsigned long           allocation;       /* Allocation mode */
  __u32                   write_seq;
  __u32                   sent_seq;
  __u32                   acked_seq;
  __u32                   copied_seq;
  __u32                   rcv_ack_seq;
  unsigned short          rcv_ack_cnt;      /* count of same ack */
  __u32                   window_seq;
  __u32                   fin_seq;
  __u32                   urg_seq;
  __u32                   urg_data;
  __u32                   syn_seq;
  int                     users;            /* user count */
/*
 *    Not all are volatile, but some are, so we
 *     might as well say they all are.
 */
  volatile char           dead,
                          urginline,
                          intr,
                          blog,
                          done,
                          reuse,
                          keepopen,
                          linger,
                          delay_acks,
                          destroy,
                          ack_timed,
                          no_check,
                          zapped,
                          broadcast,
                          nonagle,
                          bsdism;
  unsigned long           lingertime;
  int                     proc;

  struct sock             *next;
  struct sock             **pprev;
  struct sock             *bind_next;
  struct sock             **bind_pprev;
  struct sock             *pair;
```

```
    int                     hashent;
    struct sock             *prev;
    struct sk_buff          *volatile send_head;
    struct sk_buff          *volatile send_next;
    struct sk_buff          *volatile send_tail;
    struct sk_buff_head     back_log;
    struct sk_buff          *partial;
    struct timer_list       partial_timer;
    long                    retransmits;
    struct sk_buff_head     write_queue,
                            receive_queue;
    struct proto            *prot;
    struct wait_queue       **sleep;
    __u32                   daddr;
    __u32                   saddr;          /* Sending source */
    __u32                   rcv_saddr;      /* Bound address */
    unsigned short          max_unacked;
    unsigned short          window;
    __u32                   lastwin_seq;    /* sequence number when we last
                                               updated the window we offer */
    __u32                   high_seq;       /* sequence number when we did
                                               current fast retransmit */
    volatile unsigned long  ato;            /* ack timeout */
    volatile unsigned long  lrcvtime;       /* jiffies at last data rcv */
    volatile unsigned long  idletime;       /* jiffies at last rcv */
    unsigned int            bytes_rcv;
/*
 *    mss is min(mtu, max_window)
 */
    unsigned short          mtu;            /* mss negotiated in the syn's */
    volatile unsigned short mss;            /* current eff. mss - can change
*/
    volatile unsigned short user_mss;       /* mss requested by user in ioctl
*/
    volatile unsigned short max_window;
    unsigned long           window_clamp;
    unsigned int            ssthresh;
    unsigned short          num;
    volatile unsigned short cong_window;
    volatile unsigned short cong_count;
    volatile unsigned short packets_out;
    volatile unsigned short shutdown;
    volatile unsigned long  rtt;
    volatile unsigned long  mdev;
    volatile unsigned long  rto;
    volatile unsigned short backoff;
    int                     err, err_soft;  /* Soft holds errors that don't
                                               cause failure but are the
cause
                                               of a persistent failure not
                                               just 'timed out' */
```

```
    unsigned char           protocol;
    volatile unsigned char  state;
    unsigned char           ack_backlog;
    unsigned char           max_ack_backlog;
    unsigned char           priority;
    unsigned char           debug;
    int                     rcvbuf;
    int                     sndbuf;
    unsigned short          type;
    unsigned char           localroute;       /* Route locally only */
/*
 *    This is where all the private (optional) areas that don't
 *    overlap will eventually live.
 */
    union
    {
          struct unix_opt   af_unix;
#if defined(CONFIG_ATALK) || defined(CONFIG_ATALK_MODULE)
        struct atalk_sock   af_at;
#endif
#if defined(CONFIG_IPX) || defined(CONFIG_IPX_MODULE)
        struct ipx_opt      af_ipx;
#endif
#ifdef CONFIG_INET
        struct inet_packet_opt  af_packet;
#ifdef CONFIG_NUTCP
        struct tcp_opt      af_tcp;
#endif
#endif
    } protinfo;
/*
 *    IP 'private area'
 */
    int                     ip_ttl;           /* TTL setting */
    int                     ip_tos;           /* TOS */
    struct tcphdr           dummy_th;
    struct timer_list       keepalive_timer;  /* TCP keepalive hack */
    struct timer_list       retransmit_timer; /* TCP retransmit timer */
    struct timer_list       delack_timer;     /* TCP delayed ack timer */
    int                     ip_xmit_timeout;  /* Why the timeout is running */
    struct rtable           *ip_route_cache;  /* Cached output route */
    unsigned char           ip_hdrincl;       /* Include headers ? */
#ifdef CONFIG_IP_MULTICAST
    int                     ip_mc_ttl;        /* Multicasting TTL */
    int                     ip_mc_loop;       /* Loopback */
    char                    ip_mc_name[MAX_ADDR_LEN]; /* Multicast device name
*/
    struct ip_mc_socklist   *ip_mc_list;      /* Group array */
#endif

/*
```

```
 *    This part is used for the timeout functions (timer.c).
 */
    int                     timeout;        /* What are we waiting for? */
    struct timer_list       timer;          /* This is the TIME_WAIT/receive
                                             * timer when we are doing IP
                                             */
    struct timeval          stamp;
 /*
  *    Identd
  */
    struct socket           *socket;
 /*
  *    Callbacks
  */
    void                    (*state_change)(struct sock *sk);
    void                    (*data_ready)(struct sock *sk,int bytes);
    void                    (*write_space)(struct sock *sk);
    void                    (*error_report)(struct sock *sk);

};
```

socket

每个socket数据结构保存一个BSD套接字的信息，但它不是独立存在的，而是 VFS inode 数据结构的一个部分。

```
struct socket {
  short                 type;         /* SOCK_STREAM, ...             */
  socket_state          state;
  long                  flags;
  struct proto_ops      *ops;         /* protocols do most everything */
  void                  *data;        /* protocol data                */
  struct socket         *conn;        /* server socket connected to   */
  struct socket         *iconn;       /* incomplete client conn.s     */
  struct socket         *next;
  struct wait_queue     **wait;       /* ptr to place to wait on      */
  struct inode          *inode;
  struct fasync_struct *fasync_list; /* Asynchronous wake up list    */
  struct file           *file;        /* File back pointer for gc     */
};
```

task_struct

每个task_struct数据结构描述了一个系统中的进程或任务。

```
struct task_struct {
/* these are hardcoded - don't touch */
  volatile long         state;          /* -1 unrunnable, 0 runnable, >0 stopped
*/
  long                  counter;
  long                  priority;
  unsigned              long signal;
  unsigned              long blocked;   /* bitmap of masked signals */
```

```
    unsigned              long flags;     /* per process flags, defined below */
    int errno;
    long                  debugreg[8];    /* Hardware debugging registers */
    struct exec_domain    *exec_domain;
  /* various fields */
    struct linux_binfmt   *binfmt;
    struct task_struct    *next_task, *prev_task;
    struct task_struct    *next_run,  *prev_run;
    unsigned long         saved_kernel_stack;
    unsigned long         kernel_stack_page;
    int                   exit_code, exit_signal;
    /* ??? */
    unsigned long         personality;
    int                   dumpable:1;
    int                   did_exec:1;
    int                   pid;
    int                   pgrp;
    int                   tty_old_pgrp;
    int                   session;
    /* boolean value for session group leader */
    int                   leader;
    int                   groups[NGROUPS];
    /*
     * pointers to (original) parent process, youngest child, younger sibling,
     * older sibling, respectively.  (p->father can be replaced with
     * p->p_pptr->pid)
     */
    struct task_struct    *p_opptr, *p_pptr, *p_cptr,
                          *p_ysptr, *p_osptr;
    struct wait_queue     *wait_chldexit;
    unsigned short        uid,euid,suid,fsuid;
    unsigned short        gid,egid,sgid,fsgid;
    unsigned long         timeout, policy, rt_priority;
    unsigned long         it_real_value, it_prof_value, it_virt_value;
    unsigned long         it_real_incr, it_prof_incr, it_virt_incr;
    struct timer_list     real_timer;
    long                  utime, stime, cutime, cstime, start_time;
  /* mm fault and swap info: this can arguably be seen as either
     mm-specific or thread-specific */
    unsigned long         min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
    int swappable:1;
    unsigned long         swap_address;
    unsigned long         old_maj_flt;    /* old value of maj_flt */
    unsigned long         dec_flt;        /* page fault count of the last time */
    unsigned long         swap_cnt;       /* number of pages to swap on next pass
  */
  /* limits */
    struct rlimit         rlim[RLIM_NLIMITS];
    unsigned short        used_math;
    char                  comm[16];
  /* file system info */
```

```
  int                   link_count;
  struct tty_struct     *tty;           /* NULL if no tty */
/* ipc stuff */
  struct sem_undo       *semundo;
  struct sem_queue      *semsleeping;
/* ldt for this task - used by Wine.  If NULL, default_ldt is used */
  struct desc_struct *ldt;
/* tss for this task */
  struct thread_struct tss;
/* filesystem information */
  struct fs_struct      *fs;
/* open file information */
  struct files_struct  *files;
/* memory management info */
  struct mm_struct      *mm;
/* signal handlers */
  struct signal_struct *sig;
#ifdef __SMP__
  int                   processor;
  int                   last_processor;
  int                   lock_depth;     /* Lock depth.
                                           We can context switch in and out
                                           of holding a syscall kernel lock...
*/
#endif
};
```

timer_list

该数据结构用于实现进程的真实时间定时器。

```
struct timer_list {
  struct timer_list *next;
  struct timer_list *prev;
  unsigned long expires;
  unsigned long data;
  void (*function)(unsigned long);
};
```

tq_struct

每个tq_struct数据结构包含有队列列中一项工作的信息。

```
struct tq_struct {
    struct tq_struct *next;   /* linked list of active bh's */
    int sync;                 /* must be initialized to zero */
    void (*routine)(void *);  /* function to call */
    void *data;               /* argument to function */
};
```

vm_area_struct

每个vm_area_struct数据结构描述一个进程的虚内存空间。

```
struct vm_area_struct {
  struct mm_struct * vm_mm;  /* VM area parameters */
  unsigned long vm_start;
  unsigned long vm_end;
  pgprot_t vm_page_prot;
  unsigned short vm_flags;
/* AVL tree of VM areas per task, sorted by address */
  short vm_avl_height;
  struct vm_area_struct * vm_avl_left;
  struct vm_area_struct * vm_avl_right;
/* linked list of VM areas per task, sorted by address */
  struct vm_area_struct * vm_next;
/* for areas with inode, the circular list inode->i_mmap */
/* for shm areas, the circular list of attaches */
/* otherwise unused */
  struct vm_area_struct * vm_next_share;
  struct vm_area_struct * vm_prev_share;
/* more */
  struct vm_operations_struct * vm_ops;
  unsigned long vm_offset;
  struct inode * vm_inode;
  unsigned long vm_pte;      /* shared mem */
};
```

附录A　有用的Web和FTP站点

下列是一些有用的World Wide Web和ftp站点。

http://www.azstarnet.com/～axplinux：这是David Mosberger-Tang的Alpha AXP linux网络站点，所有的Alpha AXP How To均可在此找到，并且还包括大量的Linux的指示器和Alpha AXP的特殊信息，如CPU数据表格。

http://www.redhat.com/Red Hat的网络站点，包括大量的有用的指示器。

ftp://sunsite.unc.edv：大量免费软件的主要汇集点。pub/linux目录下有linux专用软件。

http://www.intel.com：Intel公司的网络站点，是查寻Intel芯片信息的好地方。

http://www.ssc.com/lj/index.html：The Linux Journal是一本非常好的Linux杂志，值得每年摘录其中优秀的文章。

http://www.blackdown.org/java/linux.html：关于Java和Linux的基本站点。

ftp://tsx-11.mit.edu/ftp/pub/Linux：MIT的Linuxftp站点。

ftp://ftp.cs.helsinki.fi/pub/software/linux/kernel：Linux的核心资源。

http://www.linux.org.uk：UF Linux Use Group。

http://sunsite.unc.edu/mdw/linux.html：Linux Documentation Project主页。

http://www.digital.com：Digital Equipment Corporation的主要网页。

http://altavista.digital.com：DIGITAL公司的 Altavista的搜索引擎，是在网络和新闻群中搜寻信息的非常好的站点。

http://www.Linuxhq.com：The Linux HQ网络站点，存储最新的官方和非官方的patch，即帮助读者得到系统所需的最好的内核资源的建议和网络指示器。

http://www.amd.com：The AMD网络站点。

http://www.cyrix.com：Cyrix的网络站点。

附录B 词 汇 表

Argument：函数和例程的参数。

ARP：地址解析协议，用于把IP地址解析成物理硬件地址。

ASCII：美国标准信息交换码，字母表中每一个字母用一个 8位代码表示。

Bit：数据的一个位，表示0或1(开或关)。

Bottom Half Handler：在内核中的工作队列处理器。

Byte：8位数据。

C：一种高级程序语言，大多数的Linux内核用C语言编写。

CPU：中央处理单元，计算机的主要部件，包括微处理器和处理器。

Data Structure：内存中一组域数据的集合。

Device Driver：设备驱动程序，控制特定设备的软件。

DMA：直接内存访问。

ELF：可执行和可链接文件格式，现已由 UNIX系统实验室指定为Linux中最通用的对象文件格式。

EIDE：扩展IDE。

Executable Image：包含机器指令和数据的结构化文件，可将该文件载入处理器虚内存并执行。

Function：完成一项操作的一块软件。

IDE：集成硬盘电路。

Image：见Executable image。

IP：网际协议。

IPC：内部进程通信。

Interface：标准的例程调用和数据传输方式。

IRQ：中断请求队列。

ISA：工业标准体系结构，这是一种较为过时的标准数据总线接口。

Kernel Module：动态载入的内核函数。

Kilobytes：一千个数据字节，通常记为KB。

Megabytes：一百万个数据字节，通常记为MB。

Microprocessor：一种高度集成的CPU。

Module：一种包含CPU指令的文件，这些指令以汇编或C语言编成。

Object File：一种文件，包含已被编译但未链接的机器指令和数据。

Page：页，物理内存被均分为若干页。

Pointer：指向其他内存位置的内存数据。

Process：一种能够执行程序的实体。

Processor：Microprocessor的简写，等价于CPU。

PCI：外围构件互连，一种标准，定义计算机系统外围构件如何连接。

Peripheral：一种代替系统CPU工作的智能处理器，如IDE控制芯片。

Protocol：一种网络语言，用于两远程进程间传输应用数据。

Register：芯片中的一处区域，用于存储信息或指令。

Routine：类似于Function，只是不需要返回值。

SCSI：小型计算机系统接口。

shell：一个程序，起到操作系统和用户之间接口的作用，也称为 command shell，Linux中最通用的是bash shell。

SMP：对称多处理机，系统中有多个处理器，平均分工。

Socket：一个套接字代表网络连接中的一端， Linux支持BSD Socket接口。

System V：1983生产的一种变型UNIX，包含了System V IPC机

TCP：传输控制协议。

Task Queue：Linux内核用于延迟工作的一种机制。

UDP：用户数据报协议。

Virtual memory：一种硬件和软件机制，用于支持大于实际值的系统物理内存。

第1章 Hello, World

如果第一个程序员是一个山顶洞人，它在山洞壁 (第一台计算机)上凿出的第一个程序应该是用羚羊图案构成的一个字符串“ Hello, World ”。罗马的编程教科书也应该是以程序“ Salut, Mundi ”开始的。我不知道如果打破这个传统会带来什么后果，至少我还没有勇气去做第一个吃螃蟹的人。

内核模块至少必须有两个函数：init_module和cleanup_module。第一个函数是在把模块插入内核时调用的；第二个函数则在删除该模块时调用。一般来说， init_module可以为内核的某些东西注册一个处理程序，或者也可以用自身的代码来取代某个内核函数 (通常是先干点别的什么事，然后再调用原来的函数)。函数cleanup_module的任务是清除掉init_module所做的一切，这样，这个模块就可以安全地卸载了。

```
ex hello.c

/* hello.c
 * Copyright (C) 1998 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version.
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work
*/
#include <linux/module.h>   /* Specifically, a module */
/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif



/* Initialize the module */
int init_module()
{
  printk("Hello, world - this is the kernel speaking\n");

  /* If we return a non zero value, it means that
   * init_module failed and the kernel module
   * can't be loaded */
  return 0;
}
```

```
/* Cleanup - undid whatever init_module did */
void cleanup_module()
{
  printk("Short is the life of a kernel module\n");
}
```

1.1 内核模块的Makefiles文件

内核模块并不是一个独立的可执行文件，而是一个对象文件，在运行时内核模块被链接到内核中。因此，应该使用 -c命令参数来编译它们。还有一点需要注意，在编译所有内核模块时，都将需要定义好某些特定的符号。

- _ _KERNEL_ _—— 这个符号告诉头文件：这个程序代码将在内核模式下运行，而不要作为用户进程的一部分来执行。
- MODULE —— 这个符号告诉头文件向内核模块提供正确的定义。
- LINUX —— 从技术的角度讲，这个符号不是必需的。然而，如果程序员想要编写一个重要的内核模块，而且这个内核模块需要在多个操作系统上编译，在这种情况下，程序员将会很高兴自己定义了LINUX这个符号。这样一来，在那些依赖于操作系统的部分，这个符号就可以提供条件编译了。

还有其它的一些符号，是否包含它们要取决于在编译内核时使用了哪些命令参数。如果用户不太清楚内核是怎样编译的，可以查看文件 /usr/include/linux/config.h。

- _ _SMP_ _—— 对称多处理。如果编译内核的目的是为了支持对称多处理，在编译时就需要定义这个符号 (即使内核只是在一个 CPU上运行也需要定义它)。当然，如果用户使用对称多处理，那么还需要完成其它一些任务 (参见第12章)。
- CONFIG_MODVERSIONS —— 如果CONFIG_MODVERSIONS可用，那么在编译内核模块时就需要定义它，并且包含头文件 /usr/include/linux/modversions.h。还可以用代码自身来完成这个任务。

```
ex Makefile

# Makefile for a basic kernel module

CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o: hello.c /usr/include/linux/version.h
        $(CC) $(MODCFLAGS) -c hello.c
        echo insmod hello.o to turn it on
        echo rmmod hello to turn if off
        echo
        echo X and kernel programming do not mix.
        echo Do the insmod and rmmod from outside X.
```

完成了以上这些任务以后，剩下唯一要做的事就是切换到根用户下 (你不是以root身份编译内核模块的吧？别玩什么惊险动作哟！)，然后根据自己的需要插入或删除 hello模块。在执行完insmod命令以后，可以看到新的内核模块在 /proc/modules中。

顺便提一下，Makefile建议用户不要从X执行insmod命令的原因在于，当内核有个消息需要使用printk命令打印出来时，内核会把该消息发送给控制台。当用户没有使用 X时，该消息

将发送到用户正在使用的虚拟终端 (用户可以用Alt-F<n>来选择当前终端)，然后用户就可以看到这个消息了。而另一方面，当用户使用 X时，存在两种可能性。一种情况是用户用命令xterm -C打开了一个控制台，这时输出将被发送到那个控制台；另一种情况是用户没有打开控制台，这时输出将送往虚拟终端7 —— 被X所“覆盖”的一个虚拟终端。

当用户的内核不太稳定时，没有使用 X的用户更有可能取得调试信息。如果没有使用 X，printk将直接从内核把调试消息发送到控制台。而另一方面，在X中printk的消息将被送给一个用户模式的进程(xterm -C)。当那个进程获得CPU时间时，它将把该消息传送给X服务器进程。然后，当X服务器获得CPU时间时，它将显示该消息 —— 但是一个不稳定的内核通常意味着系统将要崩溃或者重新启动，所以用户不希望推迟错误信息显示的时间，因为该信息可能会向用户解释什么地方出了问题，如果显示的时刻晚于系统崩溃或重启的时刻，用户将会错过这个重要的信息。

1.2 多重文件内核模块

有时候在多个源文件间划分内核模块是很有意义的。这时用户需要完成下面三件任务：

1) 除了一个源文件以外，在其它所有源文件中加入一行 #define _ _ NO_VERSION_ _。这点很重要，因为module.h中通常会包含有kernel_version的定义(kernel_version是一个全局变量，它表明该模块是为哪个内核版本所编译的)。如果用户需要version.h文件，那么用户必须自己把它包含在源文件中，因为在定义了 _ _NO_VERSION_ _的情况下， module.h是不会为用户完成这个任务的。

2) 像平常一样编译所有的源文件。

3) 把所有的对象文件组合进一个文件中。在 x86下，可以使用命令：

ld -m elf_i386 -r -o〈模块名称〉.o (第一个源文件).o (第二个源文件).o来完成这个任务。

下面是这种内核模块的一个例子。

ex start.c

```
/* start.c
 * Copyright (C) 1999 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version.
 * This file includes just the start routine
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */


/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


/* Initialize the module */
int init_module()
{
  printk("Hello, world - this is the kernel speaking\n");
```

```
  /* If we return a non zero value, it means that
   * init_module failed and the kernel module
   * can't be loaded */
  return 0;
}
ex stop.c
/* stop.c
 * Copyright (C) 1999 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version. This
 * file includes just the stop routine.
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */

#define __NO_VERSION__      /* This isn't "the" file
                             * of the kernel module */
#include <linux/module.h>   /* Specifically, a module */

#include <linux/version.h>   /* Not included by
                              * module.h because
                              * of the __NO_VERSION__ */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Cleanup - undid whatever init_module did */
void cleanup_module()
{
  printk("Short is the life of a kernel module\n");
}
ex Makefile

# Makefile for a multifile kernel module
CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o:    start.o stop.o
            ld -m elf_i386 -r -o hello.o start.o stop.o

start.o:    start.c /usr/include/linux/version.h
            $(CC) $(MODCFLAGS) -c start.c

stop.o:             stop.c /usr/include/linux/version.h
            $(CC) $(MODCFLAGS) -c stop.c
```

第2章　字符设备文件

我们现在就可以吹牛说自己是内核程序员了。虽然我们所写的内核模块还什么也干不了，但我们仍然为自己感到骄傲，简直可以称得上趾高气扬。但是，有时候在某种程度上我们也会感到缺少点什么，简单的模块并不是太有趣。

内核模块主要通过两种方法与进程打交道。一种方法是通过设备文件（例如在目录/dev中的文件），另一种方法是使用proc文件系统。既然编写内核模块的主要原因之一就是支持某些类型的硬件设备，那么就让我们从设备文件开始吧。

设备文件最初的用途是使进程与内核中的设备驱动程序通信，并且通过设备驱动程序再与物理设备(调制解调器、终端等等)通信。下面我们要讲述实现这一任务的方法。

每个设备驱动程序都被赋予一个主编号，主要用于负责某几种类型的硬件。可以在/proc/devices中找到驱动程序以及它们对应的主编号的列表。由设备驱动程序管理的每个物理设备都被赋予一个从编号。这些设备中的每一个，不管是否真正安装在计算机系统上，都将对应一个特殊的文件，该文件称为设备文件，所有的设备文件都包含在目录 /dev中。

例如，如果执行命令 ls -l /dev/hd[ab]*，用户将可以看到与一个计算机相连接的所有的IDE硬件分区。注意，所有的这些硬盘分区都使用同一个主编号： 3，但是从编号却各不相同。需要强调的是，这里假设用户使用的是 PC体系结构。我并不知道在其它体系结构上运行的Linux的设备是怎么样的。

在安装了系统以后，所有的设备文件都由命令 mknod创建出来。从技术的角度上讲，并没有什么特别的原因一定要把这些设备文件放在目录 /dev中，这只不过是一个有用的传统习惯而已。如果读者创建设备文件的目的只不过是为了试试看，就像本章的练习一样，那么把该设备文件放置在编译内核模块的目录中可能会更有意义一些。

设备一般分为两种类型：字符设备和块设备。它们的区别在于块设备具有一个请求缓冲区，所以块设备可以选择按照何种顺序来响应这些请求。这对于存储设备来说是很重要的。在存储设备中，读或写相邻的扇区速度要快一些，而读写相互之间离得较远的扇区则要慢得多。另一个区别在于块设备只能以成块的形式接收输入和返回输出（块的大小根据设备类型的变化而有所不同），而字符设备则可以随心所欲地使用任意数目的字节。当前大多数设备都是字符设备，因为它们既不需要某种形式的缓冲，也不需要按照固定的块大小来进行操作。如果想知道某个设备文件对应的是块设备还是字符设备，用户可以执行命令 ls -l，查看一下该命令的输出中的第一个字符，如果第一个字符是“ b ”，则对应的是块设备；如果是“ c ”，则对应的是字符设备。

模块分为两个独立的部分：模块部分和设备驱动程序部分。前者用于注册设备。函数init_module调用module_register_chrdev，把该设备驱动程序加入到内核的字符设备驱动程序表中，它还会返回供驱动程序所使用的主编号。函数 cleanup_module则取消该设备的注册。

注册某设备和取消它的注册是这两个函数最基本的功能。内核中的东西并不是按照它们自己的意愿主动开始运行的，就像进程一样，而是由进程通过系统调用来调用，或者由硬件

设备通过中断来调用，或者由内核的其它部分调用 (只需调用特定的函数)，它们才会执行。因此，如果用户往内核中加入了代码，就必须把它作为某种特定类型事件的处理程序进行注册；而在删除这些代码时，用户必须取消它的注册。

设备驱动程序一般是由四个 device_<action>函数所组成的，如果用户需要处理具有对应主编号的设备文件，就可以调用这四个函数。通过 file_operations结构Fops内核可以知道调用哪些函数。因为该结构的值是在注册设备时给定的，它包含了指向这四个函数的指针。

在这里我们还需要记住的一点是：无论如何不能乱删内核模块。原因在于如果设备文件是由进程打开的，而我们删去了该内核模块，那么使用该文件就将导致对正确的函数 (读/写) 原来所处的存储位置的调用。如果我们走运，那里没有装入什么其它的代码，那我们至多得到一些难看的错误信息，而如果我们不走运，在原来的同一位置已经装入了另一个内核模块，这就意味着跳转到了内核中另一个函数的中间，这样做的后果是不堪设想的，起码不会是令人愉快的。

一般来说，如果用户不愿意让某件事发生，可以让执行这件事的函数返回一个错误代码 (一个负数)。而对cleanup_module来说这是不可能的，因为它是一个 void函数。一旦调用了 cleanup_module，这个模块就死了。然而，还有一个计数器记录了有多少个其它的内核模块正在使用该内核模块，这个计数器称为引用计数器 (就是位于文件/proc/modules信息行中的最后那个数值)。如果这个数值不为零， rmmod将失败。模块的引用计数值可以从变量 mod_use_count_中得到。因为有些宏是专门为处理这个变量而定义的(如MOD_INC_USE_COUNT和MOD_DEC_USE_COUNT)，我们宁愿使用这些宏，也不愿直接对mod_use_count_进行操作，这样一来，如果将来实现方法有所变化，我们也会很安全。

them, rather than mod_use_count_ directly, so we'll be safe if the implementation c
in the future.

ex chardev.c

```
/* chardev.c
 * Copyright (C) 1998-1999 by Ori Pomerantz
 *
 * Create a character device (read only)
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* For character devices */
```

```
#include <linux/fs.h>       /* The character device
                             * definitions are here */
#include <linux/wrapper.h>  /* A wrapper which does
                             * next to nothing at
                             * at present, but may
                             * help for compatibility
                             * with future versions
                             * of Linux */


/* In 2.2.3 /usr/include/linux/version.h includes
 * a macro for this, but 2.0.35 doesn't - so I add
 * it here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif
/* Conditional compilation. LINUX_VERSION_CODE is
 * the code (as per KERNEL_VERSION) of this version. */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for put_user */
#endif



#define SUCCESS 0


/* Device Declarations **************************** */

/* The name for our device, as it will appear
 * in /proc/devices */
#define DEVICE_NAME "char_dev"


/* The maximum length of the message from the device */
#define BUF_LEN 80

/* Is the device open right now? Used to prevent
 * concurent access into the same device */
static int Device_Open = 0;

/* The message the device will give when asked */
static char Message[BUF_LEN];

/* How far did the process reading the message
 * get? Useful if the message is larger than the size
 * of the buffer we get to fill in device_read. */
static char *Message_Ptr;


/* This function is called whenever a process
```

```
 * attempts to open the device file */
static int device_open(struct inode *inode,
                       struct file *file)
{
  static int counter = 0;

#ifdef DEBUG
  printk ("device_open(%p,%p)\n", inode, file);
#endif

  /* This is how you get the minor device number in
 * case you have more than one physical device using
 * the driver. */
printk("Device: %d.%d\n",
   inode->i_rdev >> 8, inode->i_rdev & 0xFF);

/* We don't want to talk to two processes at the
 * same time */
if (Device_Open)
  return -EBUSY;

/* If this was a process, we would have had to be
 * more careful here.
 *
 * In the case of processes, the danger would be
 * that one process might have check Device_Open
 * and then be replaced by the schedualer by another
 * process which runs this function. Then, when the
 * first process was back on the CPU, it would assume
 * the device is still not open.
 *
 * However, Linux guarantees that a process won't be
 * replaced while it is running in kernel context.
 *
 * In the case of SMP, one CPU might increment
 * Device_Open while another CPU is here, right after
 * the check. However, in version 2.0 of the
 * kernel this is not a problem because there's a lock
 * to guarantee only one CPU will be kernel module at
 * the same time. This is bad in  terms of
 * performance, so version 2.2 changed it.
 * Unfortunately, I don't have access to an SMP box
 * to check how it works with SMP.
 */

Device_Open++;

/* Initialize the message. */
sprintf(Message,
  "If I told you once, I told you %d times - %s",
```

```
    counter++,
    "Hello, world\n");
  /* The only reason we're allowed to do this sprintf
   * is because the maximum length of the message
   * (assuming 32 bit integers - up to 10 digits
   * with the minus sign) is less than BUF_LEN, which
   * is 80. BE CAREFUL NOT TO OVERFLOW BUFFERS,
     * ESPECIALLY IN THE KERNEL!!!
     */

    Message_Ptr = Message;

    /* Make sure that the module isn't removed while
     * the file is open by incrementing the usage count
     * (the number of opened references to the module, if
     * it's not zero rmmod will fail)
     */
    MOD_INC_USE_COUNT;

    return SUCCESS;
  }


  /* This function is called when a process closes the
   * device file. It doesn't have a return value in
   * version 2.0.x because it can't fail (you must ALWAYS
   * be able to close a device). In version 2.2.x it is
   * allowed to fail - but we won't let it.
   */
  #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  static int device_release(struct inode *inode,
                            struct file *file)
  #else
  static void device_release(struct inode *inode,
                             struct file *file)
  #endif
  {
  #ifdef DEBUG
    printk ("device_release(%p,%p)\n", inode, file);
  #endif

    /* We're now ready for our next caller */
    Device_Open --;

    /* Decrement the usage count, otherwise once you
     * opened the file you'll never get rid of the module.
     */
    MOD_DEC_USE_COUNT;

  #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    return 0;
```

```
#endif
}
/* This function is called whenever a process which
 * have already opened the device file attempts to
 * read from it. */


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_read(struct file *file,
    char *buffer,    /* The buffer to fill with data */
    size_t length,   /* The length of the buffer */
    loff_t *offset)  /* Our offset in the file */
#else
static int device_read(struct inode *inode,
                       struct file *file,
    char *buffer,   /* The buffer to fill with
                     * the data */
    int length)     /* The length of the buffer
                     * (mustn't write beyond that!) */
#endif
{
  /* Number of bytes actually written to the buffer */
  int bytes_read = 0;

  /* If we're at the end of the message, return 0
   * (which signifies end of file) */
  if (*Message_Ptr == 0)
    return 0;

  /* Actually put the data into the buffer */
  while (length && *Message_Ptr)  {

    /* Because the buffer is in the user data segment,
     * not the kernel data segment, assignment wouldn't
     * work. Instead, we have to use put_user which
     * copies data from the kernel data segment to the
     * user data segment. */
    put_user(*(Message_Ptr++), buffer++);


    length --;
    bytes_read ++;
  }

#ifdef DEBUG
   printk ("Read %d bytes, %d left\n",
     bytes_read, length);
#endif
 /* Read functions are supposed to return the number
  * of bytes actually inserted into the buffer */
```

```
  return bytes_read;
}



/* This function is called when somebody tries to write
 * into our device file - unsupported in this example. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_write(struct file *file,
    const char *buffer,    /* The buffer */
    size_t length,   /* The length of the buffer */
    loff_t *offset)  /* Our offset in the file */
#else
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
#endif
{
  return -EINVAL;
}



/* Module Declarations ***************************** */

/* The major device number for the device. This is
 * global (well, static, which in this context is global
 * within this file) because it has to be accessible
 * both for registration and for release. */
static int Major;

/* This structure will hold the functions to be
 * called when a process does something to the device
 * we created. Since a pointer to this structure is
 * kept in the devices table, it can't be local to
 * init_module. NULL is for unimplemented functions. */


struct file_operations Fops = {
  NULL,   /* seek */
  device_read,
  device_write,
  NULL,   /* readdir */
  NULL,   /* select */
  NULL,   /* ioctl */
  NULL,   /* mmap */
  device_open,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
```

```
  NULL,   /* flush */
#endif
  device_release  /* a.k.a. close */
};


/* Initialize the module - Register the character device */
int init_module()
{
  /* Register the character device (atleast try) */
  Major = module_register_chrdev(0,
                                 DEVICE_NAME,
                                 &Fops);

  /* Negative values signify an error */
  if (Major < 0) {
    printk ("%s device failed with %d\n",
        "Sorry, registering the character",
        Major);
    return Major;
  }

  printk ("%s The major device number is %d.\n",
          "Registeration is a success.",
          Major);
  printk ("If you want to talk to the device driver,\n");
  printk ("you'll have to create a device file. \n");
  printk ("We suggest you use:\n");
  printk ("mknod <name> c %d <minor>\n", Major);
  printk ("You can try different minor numbers %s",
          "and see what happens.\n");

  return 0;
}


/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{
  int ret;
  /* Unregister the device */
  ret = module_unregister_chrdev(Major, DEVICE_NAME);

  /* If there's an error, report it */
  if (ret < 0)
    printk("Error in unregister_chrdev: %d\n", ret);
}
```

多内核版本源文件

系统调用是内核提供给进程的主要接口，它并不随着内核版本的变化而有所改变。当然

可能会加入新的系统调用，但是老的系统调用是永远不会改变的。这主要是为了提供向后兼容性的需要 —— 新的内核版本不应该使原来工作正常的进程出现问题。在大多数情况下，设备文件也应该保持不变。另一方面，内核里面的内部接口则可以变化，并且也确实随着内核版本的变化而改变了。

Linux内核的版本可以划分为稳定版本(n.<偶数>.m)和开发版本(n.<奇数>.m)两种。开发版本中包括所有最新最酷的思想，当然其中也可能有一些将来会被认为是馊主意的错误，有些将会在下一个版本中重新实现。因此，用户不能认为在这些版本之间接口也会保持一致 (这就是我为什么懒得在本书中介绍它们的原因，这需要做大量的工作，而且很快就会过时被淘汰)。另一方面，在稳定的版本中，我们可以无视错误修正版本 (带数字m的版本)而认为接口是保持不变的。

这个MPG版本包含对Linux内核版本2.0.x和版本2.2.x的支持。因为这两个版本之间存在差异，这就要求用户根据内核版本号来进行条件编译。为了做到这一点，可以使用宏LINUX_VERSION_CODE。在内核版本a.b.c中，该宏的值将会是$2^{16}a+2^{8}b+c$。为了获取特定内核版本的值，我们可以使用宏 KERNEL_VERSION。在2.0.35中该宏没有定义，如果需要的话我们可以自己定义它。

第3章　/proc文件系统

在Linux中，内核和内核模块还可以通过另一种方法把信息发送给进程，这种方法就是/proc 文件系统。最初/proc 文件系统是为了可以轻松访问有关进程的信息而设计的 (这就是它的名称的由来)，现在每一个内核部分只要有些信息需要报告，都可以使用 /proc 文件系统，例如/proc /modules包含一个模块的列表，/proc /meminfo包含有关内存使用的统计信息。

使用/proc 文件系统的方法其实与使用设备驱动程序的方法是非常类似的 —— 用户需要创建一个结构，该结构包含了 /proc文件所需要的所有信息，包括指向任意处理程序函数的指针 (在我们本章的例子中只有一个处理程序函数，当有人试图读 /proc文件时将调用这个函数)。然后，init_module将向内核注册这个结构，而 cleanup_module将取消它的注册。

在程序中之所以需要使用 proc_register_dynamic，是因为我们不想事先判断文件所使用的索引节点编号，而是让内核去决定，这样可以避免编号冲突。一般文件系统都是位于磁盘上的，而不是仅仅存在于内存中 (/proc是存在于内存中的)，在那种情况下，索引节点编号是一个指向某个磁盘位置的指针，在那个位置上存放了该文件的索引节点 (简写为inode)。索引节点包含该文件的有关信息，例如文件的访问权限，以及指向某个或者某些磁盘位置的指针，在这个或者这些磁盘位置中，存放着文件的数据。

因为在文件打开或者关闭时该模块不会被调用，所以我们无需在该模块中使用MOD_INC_USE_COUNT和MOD_DEC_USE_COUNT。如果文件打开以后模块被删除了，没有任何措施可以避免这一后果。在下一章中，我们将学习到一种比较难于实现，但却相对方便的方法，可以用于处理/proc文件，我们也可以使用那个方法来防止这个问题。

```
ex procfs.c

/* procfs.c -  create a "file" in /proc
 * Copyright (C) 1998-1999 by Ori Pomerantz
 */


/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


/* Necessary because we use the proc fs */
```

```
#include <linux/proc_fs.h>



/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* Put data into the proc fs file.

   Arguments
   =========
   1. The buffer where the data is to be inserted, if
      you decide to use it.
   2. A pointer to a pointer to characters. This is
      useful if you don't want to use the buffer
      allocated by the kernel.
   3. The current position in the file.
   4. The size of the buffer in the first argument.
   5. Zero (for future use?).


   Usage and Return Value
   ======================
   If you use your own buffer, like I do, put its
   location in the second argument and return the
   number of bytes used in the buffer.

   A return value of zero means you have no further
   information at this time (end of file). A negative
   return value is an error condition.


   For More Information
   ====================
   The way I discovered what to do with this function
   wasn't by reading documentation, but by reading the
   code which used it. I just looked to see what uses
   the get_info field of proc_dir_entry struct (I used a
   combination of find and grep, if you're interested),
   and I saw that  it is used in <kernel source
   directory>/fs/proc/array.c.

   If something is unknown about the kernel, this is
   usually the way to go. In Linux we have the great
   advantage of having the kernel source code for
```

```
     free - use it.
   */
  int procfile_read(char *buffer,
                    char **buffer_location,
                    off_t offset,
                    int buffer_length,
                    int zero)
  {
    int len;  /* The number of bytes actually used */

    /* This is static so it will still be in memory
     * when we leave this function */
    static char my_buffer[80];

    static int count = 1;
  /* We give all of our information in one go, so if the
   * user asks us if we have more information the
   * answer should always be no.
   *
   * This is important because the standard read
   * function from the library would continue to issue
   * the read system call until the kernel replies
   * that it has no more information, or until its
   * buffer is filled.
   */
  if (offset > 0)
    return 0;

  /* Fill the buffer and get its length */
  len = sprintf(my_buffer,
    "For the %d%s time, go away!\n", count,
    (count % 100 > 10 && count % 100 < 14) ? "th" :
      (count % 10 == 1) ? "st" :
        (count % 10 == 2) ? "nd" :
          (count % 10 == 3) ? "rd" : "th" );
  count++;

  /* Tell the function which called us where the
   * buffer is */
  *buffer_location = my_buffer;

  /* Return the length */
  return len;
}


struct proc_dir_entry Our_Proc_File =
  {
    0, /* Inode number - ignore, it will be filled by
        * proc_register[_dynamic] */
    4, /* Length of the file name */
```

```
"test", /* The file name */
S_IFREG | S_IRUGO, /* File mode - this is a regular
                    * file which can be read by its
                    * owner, its group, and everybody
                    * else */
1,      /* Number of links (directories where the
     * file is referenced) */
0, 0,  /* The uid and gid for the file - we give it
        * to root */
80, /* The size of the file reported by ls. */
    NULL, /* functions which can be done on the inode
           * (linking, removing, etc.) - we don't
           * support any. */
    procfile_read, /* The read function for this file,
                    * the function called when somebody
                    * tries to read something from it. */
    NULL /* We could have here a function to fill the
          * file's inode, to enable us to play with
          * permissions, ownership, etc. */
  };


/* Initialize the module - register the proc file */
int init_module()
{
  /* Success if proc_register[_dynamic] is a success,
   * failure otherwise. */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  /* In version 2.2, proc_register assign a dynamic
   * inode number automatically if it is zero in the
   * structure, so there's no more need for
   * proc_register_dynamic
   */
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif

  /* proc_root is the root directory for the proc
   * fs (/proc). This is where we want our file to be
   * located.
   */
}


/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

第4章　把/proc用于输入

到目前为止，可以通过两种方法从内核模块产生输出：我们可以注册一个设备驱动程序，并使用mknod命令创建一个设备文件；我们还可以创建一个 /proc文件。这样内核模块就可以告诉我们各种各样的信息。现在唯一的问题是我们没有办法来回答它。如果要把输入信息发送给内核模块，第一个方法就是把这些信息写回到 /proc文件中。

因为编写proc文件系统的主要目的是为了让内核可以把它的状态报告给进程，对于输入并没有提供相应的特别措施。结构 proc_dir_entry中并不包含指向输入函数的指针，它只包含指向输出函数的指针。如果需要输入，为了把信息写到 /proc文件中，用户需要使用标准的文件系统机制。

Linux为文件系统注册提供了一个标准的机制。因为每个文件系统都必须具有自己的函数专门用于处于索引节点和文件操作，所以 Linux提供了一个特殊的结构inode_operations，该结构存放指向所有这些函数的指针，其中包含一个指向结构 file_operations的指针。在/proc中，无论何时注册一个新文件，用户都可以指定使用哪个 inode_operations结构来访问它。这就是我们所使用的机制。结构 inode_operations包含指向结构 file_operations的指针，而结构file_operations又包含指向module_input和module_output函数的指针。

注意　在内核中读和写的标准角色是互换的，读函数用于输出，而写函数则用于输入，记住这点很重要。之所以会这样，是因为读和写实际上是站在用户的观点来说的—如果一个进程从内核读信息，内核需要做的是输出这些信息，而如果进程向内核写信息，内核当然会把它当作输入来接收。

这里另外一个有趣的地方是module_permission函数。只要进程试图对/proc文件干点什么，这个函数就将被调用，它可以判断是允许对文件进行访问，还是拒绝这次访问。目前这种判断还只是基于操作本身以及当前所使用的 uid来作出(当前所使用的 uid可以从current得到，current是一个指针，指向包含有关当前运行进程的信息结构)，但是函数module_permission还可以基于用户所选择的任意条件来作出允许或是拒绝访问的判断，例如其它还有什么进程正在使用这个文件、日期和时间、或者我们最近接收到的输入。

在程序中我们之所以使用put_user和get_user，主要是因为Linux内存是分段的(在Intel体系结构下；有些其它的处理器可能会有所不同)。这就意味着一个指针并不能指向内存中的某个唯一的位置，而只能指向一个内存段。为了能够使用指针，用户必须知道它指向的是哪个内存段。内核只对应一个内存段，且每个进程都对应一个内存段。

进程所能访问的唯一的内存段就是它自己的内存段，所以在写将要当作进程来运行的常规程序时，程序员不需要考虑有关分段的问题，而当用户编写内核模块时，通常用户需要访问内核的内存段，该内存段是由系统自动处理的。然而，如果内存缓冲区中的内容需要在当前运行的进程和内核之间传送，内核函数将会接收到一个指针，该指针指向进程段中的内存缓冲区。宏put_user和get_user使用户可以访问那块内存。

ex procfs.c

```
/* procfs.c -  create a "file" in /proc, which allows
 * both input and output. */

/* Copyright (C) 1998-1999 by Ori Pomerantz */


/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif
/* Necessary because we use proc fs */
#include <linux/proc_fs.h>


/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for get_user and put_user */
#endif

/* The module's file functions ********************** */


/* Here we keep the last message received, to prove
 * that we can process our input */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];


/* Since we use the file operations struct, we can't
 * use the special proc output provisions - we have to
 * use a standard read function, which is this function */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_output(
    struct file *file,    /* The file read */
```

```
    char *buf, /* The buffer to put data to (in the
                * user segment) */
    size_t len,  /* The length of the buffer */
    loff_t *offset) /* Offset in the file - ignore */
#else
static int module_output(
    struct inode *inode, /* The inode read */
    struct file *file,   /* The file read */
    char *buf, /* The buffer to put data to (in the
                * user segment) */
    int len)  /* The length of the buffer */
#endif
{
 static int finished = 0;
 int i;
 char message[MESSAGE_LENGTH+30];

 /* We return 0 to indicate end of file, that we have
  * no more information. Otherwise, processes will
  * continue to read from us in an endless loop. */
 if (finished) {
   finished = 0;
   return 0;
 }

 /* We use put_user to copy the string from the kernel's
  * memory segment to the memory segment of the process
  * that called us. get_user, btw, is
  * used for the reverse. */
 sprintf(message, "Last input:%s", Message);
 for(i=0; i<len && message[i]; i++)
   put_user(message[i], buf+i);


 /* Notice, we assume here that the size of the message
  * is below len, or it will be received cut. In a real
  * life situation, if the size of the message is less
  * than len, then we'd return len and on the second call
  * start filling the buffer with the len+1'th byte of
  * the message. */
 finished = 1;

 return i;  /* Return the number of bytes "read" */
}


/* This function receives input from the user when the
 * user writes to the /proc file. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_input(
```

```
    struct file *file,    /* The file itself */
    const char *buf,      /* The buffer with input */
    size_t length,        /* The buffer's length */
    loff_t *offset)       /* offset to file - ignore */
#else
static int module_input(
    struct inode *inode, /* The file's inode */
    struct file *file,    /* The file itself */
    const char *buf,      /* The buffer with the input */
    int length)           /* The buffer's length */
#endif
{
  int i;

  /* Put the input into Message, where module_output
   * will later be able to use it */
  for(i=0; i<MESSAGE_LENGTH-1 && i<length; i++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    get_user(Message[i], buf+i);
  /* In version 2.2 the semantics of get_user changed,
   * it not longer returns a character, but expects a
   * variable to fill up as its first argument and a
   * user segment pointer to fill it from as the its
   * second.
   *
   * The reason for this change is that the version 2.2
   * get_user can also read an short or an int. The way
   * it knows the type of the variable it should read
   * is by using sizeof, and for that it needs the
   * variable itself.
   */
#else
    Message[i] = get_user(buf+i);
#endif
  Message[i] = '\0';  /* we want a standard, zero
                       * terminated string */

  /* We need to return the number of input characters
   * used */
  return i;
}



/* This function decides whether to allow an operation
 * (return zero) or not allow it (return a non-zero
 * which indicates why it is not allowed).
 *
 * The operation can be one of the following values:
 * 0 - Execute (run the "file" - meaningless in our case)
 * 2 - Write (input to the kernel module)
```

```
 * 4 - Read (output from the kernel module)
 *
 * This is the real function that checks file
 * permissions. The permissions returned by ls -l are
 * for referece only, and can be overridden here.
 */
static int module_permission(struct inode *inode, int op)
{
  /* We allow everybody to read from our module, but
   * only root (uid 0) may write to it */
  if (op == 4 || (op == 2 && current->euid == 0))
    return 0;

  /* If it's anything else, access is denied */
  return -EACCES;
}



/* The file is opened - we don't really care about
 * that, but it does mean we need to increment the
 * module's reference count. */
int module_open(struct inode *inode, struct file *file)
{
  MOD_INC_USE_COUNT;

  return 0;
}


/* The file is closed - again, interesting only because
 * of the reference count. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
int module_close(struct inode *inode, struct file *file)
#else
void module_close(struct inode *inode, struct file *file)
#endif
{
  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;  /* success */
#endif
}


/* Structures to register as the /proc file, with
 * pointers to all the relevant functions. ********** */
/* File operations for our proc file. This is where we
```

```
 * place pointers to all the functions called when
 * somebody tries to do something to our file. NULL
 * means we don't want to deal with something. */
static struct file_operations File_Ops_4_Our_Proc_File =
  {
    NULL,  /* lseek */
    module_output,  /* "read" from the file */
    module_input,   /* "write" to the file */
    NULL,  /* readdir */
    NULL,  /* select */
    NULL,  /* ioctl */
    NULL,  /* mmap */
    module_open,    /* Somebody opened the file */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    NULL,   /* flush, added here in version 2.2 */
#endif
    module_close,    /* Somebody closed the file */
    /* etc. etc. etc. (they are all given in
     * /usr/include/linux/fs.h). Since we don't put
     * anything here, the system will keep the default
     * data, which in Unix is zeros (NULLs when taken as
     * pointers). */
  };



/* Inode operations for our proc file. We need it so
 * we'll have some place to specify the file operations
 * structure we want to use, and the function we use for
 * permissions. It's also possible to specify functions
 * to be called for anything else which could be done to
 * an inode (although we don't bother, we just put
 * NULL). */
static struct inode_operations Inode_Ops_4_Our_Proc_File =
  {
    &File_Ops_4_Our_Proc_File,
    NULL, /* create */
    NULL, /* lookup */
    NULL, /* link */
    NULL, /* unlink */
    NULL, /* symlink */
    NULL, /* mkdir */
    NULL, /* rmdir */
    NULL, /* mknod */
    NULL, /* rename */
    NULL, /* readlink */
    NULL, /* follow_link */
    NULL, /* readpage */
    NULL, /* writepage */
    NULL, /* bmap */
    NULL, /* truncate */
```

```
    module_permission /* check for permissions */
  };


/* Directory entry */
static struct proc_dir_entry Our_Proc_File =
  {
    0, /* Inode number - ignore, it will be filled by
        * proc_register[_dynamic] */
    7, /* Length of the file name */
    "rw_test", /* The file name */
    S_IFREG | S_IRUGO | S_IWUSR,
    /* File mode - this is a regular file which
     * can be read by its owner, its group, and everybody
     * else. Also, its owner can write to it.
     *
     * Actually, this field is just for reference, it's
     * module_permission that does the actual check. It
     * could use this field, but in our implementation it
     * doesn't, for simplicity. */
    1,  /* Number of links (directories where the
         * file is referenced) */
    0, 0,  /* The uid and gid for the file -
            * we give it to root */
    80, /* The size of the file reported by ls. */
    &Inode_Ops_4_Our_Proc_File,
    /* A pointer to the inode structure for
     * the file, if we need it. In our case we
     * do, because we need a write function. */
    NULL
    /* The read function for the file. Irrelevant,
     * because we put it in the inode structure above */
  };


/* Module initialization and cleanup ****************** */

/* Initialize the module - register the proc file */
int init_module()

{
  /* Success if proc_register[_dynamic] is a success,
   * failure otherwise */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  /* In version 2.2, proc_register assign a dynamic
   * inode number automatically if it is zero in the
   * structure, so there's no more need for
   * proc_register_dynamic
   */
  return proc_register(&proc_root, &Our_Proc_File);
```

```
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif
}


/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

第5章　把设备文件用于输入

设备文件一般代表物理设备，而大多数物理设备既可用于输出，也可用于输入，所以Linux必须提供一些机制，以便内核中的设备驱动程序可以从进程获得输出信息，并把它发送到设备。要做到这一点，可以为输出打开设备文件，并且向它写信息，就像写普通的文件一样。在下面的例子中，这些任务是由 device_write完成的。

当然仅有上面这种方法是不够的。假设用户有一个与调制解调器相连接的串行口 (即使用户拥有的是一个内置式的调制解调器，从 CPU角度来看，它还是由一个与调制解调器相连的串行口来实现的，所以这样的用户也无需过多地苛求自己的想象力)。用户将会做的最自然的事情就是使用设备文件来把信息写到调制解调器 (调制解调器命令或者数据将会通过电话线来传送)，并且利用设备文件从调制解调器读信息 (命令的响应或者数据也是通过电话线接收的)。然而，这会带来一个很明显的问题：如果用户需要与串行口本身交换信息的话，用户该怎么办？例如用户可能发送有关数据发送和接收的速率的值。

在Unix中，可以使用一个称为 ioctl的特殊函数来解决这个问题 (ioctl是输入输出控制的英文缩写)。每个设备都有属于自己的 ioctl命令，可以是读 ioctl(从进程把信息发送到内核)、写 ioctl(把信息返回到进程)、都有或者都没有。调用 ioctl函数必须带上三个参数：适当的设备文件的文件描述符， ioctl编号以及另外一个长整型的参数，用户可以使用这个长整型参数来传送任何信息。

ioctl编号是由主设备编号、 ioctl类型、命令以及参数的类型这几者编码而成。这个 ioctl编号通常是由一个头文件中的宏调用 (取决于类型的不同，可以是 _IO、_IOR、_IOW或者 _IOWR)来创建的。然后，将要使用 ioctl的程序以及内核模块都必须通过 #include命令包含这个头文件。前者包含这个头文件是为了生成适当的 ioctl，而后者是为了能理解它。在下面的例子中，头文件的名称是 chardev.h，而使用它的程序是 ioctl.c。

如果用户希望在自己的内核模块中使用 ioctl，最好是接受正式的 ioctl的约定，这样如果偶尔得到别人的 ioctl，或者如果别人获得了你的 ioctl，你可以知道那些地方出现了错误。如果读者想知道更多的信息，可以查询 Documentation/ioctl -number.txt下的内核源代码树。

```
ex chardev.c

/* chardev.c
 *
 * Create an input/output character device
 */


/* Copyright (C) 1998-99 by Ori Pomerantz */


/* The necessary header files */
```

```
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* For character devices */

/* The character device definitions are here */
#include <linux/fs.h>

/* A wrapper which does next to nothing at
 * at present, but may help for compatibility
 * with future versions of Linux */
#include <linux/wrapper.h>
/* Our own ioctl numbers */
#include "chardev.h"


/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for get_user and put_user */
#endif



#define SUCCESS 0


/* Device Declarations ******************************** */


/* The name for our device, as it will appear in
 * /proc/devices */
#define DEVICE_NAME "char_dev"


/* The maximum length of the message for the device */
#define BUF_LEN 80
```

```
/* Is the device open right now? Used to prevent
 * concurent access into the same device */
static int Device_Open = 0;

/* The message the device will give when asked */
static char Message[BUF_LEN];

/* How far did the process reading the message get?
 * Useful if the message is larger than the size of the
 * buffer we get to fill in device_read. */
static char *Message_Ptr;
/* This function is called whenever a process attempts
 * to open the device file */
static int device_open(struct inode *inode,
                       struct file *file)
{
#ifdef DEBUG
  printk ("device_open(%p)\n", file);
#endif

  /* We don't want to talk to two processes at the
   * same time */
  if (Device_Open)
    return -EBUSY;

  /* If this was a process, we would have had to be
   * more careful here, because one process might have
   * checked Device_Open right before the other one
   * tried to increment it. However, we're in the
   * kernel, so we're protected against context switches.
   *
   * This is NOT the right attitude to take, because we
   * might be running on an SMP box, but we'll deal with
   * SMP in a later chapter.
   */

  Device_Open++;

  /* Initialize the message */
  Message_Ptr = Message;

  MOD_INC_USE_COUNT;

  return SUCCESS;
}


/* This function is called when a process closes the
 * device file. It doesn't have a return value because
 * it cannot fail. Regardless of what else happens, you
 * should always be able to close a device (in 2.0, a 2.2
```

```
 * device file could be impossible to close). */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static int device_release(struct inode *inode,
                          struct file *file)
#else
static void device_release(struct inode *inode,
                          struct file *file)
#endif
{
#ifdef DEBUG
  printk ("device_release(%p,%p)\n", inode, file);
#endif

  /* We're now ready for our next caller */
  Device_Open --;

  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;
#endif
}



/* This function is called whenever a process which
 * has already opened the device file attempts to
 * read from it. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_read(
    struct file *file,
    char *buffer, /* The buffer to fill with the data */
    size_t length,     /* The length of the buffer */
    loff_t *offset) /* offset to the file */
#else
static int device_read(
    struct inode *inode,
    struct file *file,
    char *buffer,   /* The buffer to fill with the data */
    int length)     /* The length of the buffer
                     * (mustn't write beyond that!) */
#endif
{
  /* Number of bytes actually written to the buffer */
  int bytes_read = 0;

#ifdef DEBUG
  printk("device_read(%p,%p,%d)\n",
    file, buffer, length);
#endif
```

```
  /* If we're at the end of the message, return 0
   * (which signifies end of file) */
  if (*Message_Ptr == 0)
    return 0;

  /* Actually put the data into the buffer */
  while (length && *Message_Ptr)  {

    /* Because the buffer is in the user data segment,
     * not the kernel data segment, assignment wouldn't
     * work. Instead, we have to use put_user which
     * copies data from the kernel data segment to the
     * user data segment. */
    put_user(*(Message_Ptr++), buffer++);
    length --;
    bytes_read ++;
  }

#ifdef DEBUG
   printk ("Read %d bytes, %d left\n",
     bytes_read, length);
#endif

   /* Read functions are supposed to return the number
    * of bytes actually inserted into the buffer */
  return bytes_read;
}


/* This function is called when somebody tries to
 * write into our device file. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_write(struct file *file,
                            const char *buffer,
                            size_t length,
                            loff_t *offset)
#else
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
#endif
{
  int i;

#ifdef DEBUG
  printk ("device_write(%p,%s,%d)",
    file, buffer, length);
#endif

  for(i=0; i<length && i<BUF_LEN; i++)
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    get_user(Message[i], buffer+i);
#else
    Message[i] = get_user(buffer+i);
#endif

  Message_Ptr = Message;

  /* Again, return the number of input characters used */
  return i;
}


/* This function is called whenever a process tries to
 * do an ioctl on our device file. We get two extra
 * parameters (additional to the inode and file
 * structures, which all device functions get): the number
 * of the ioctl called and the parameter given to the
 * ioctl function.
 *
 * If the ioctl is write or read/write (meaning output
 * is returned to the calling process), the ioctl call
 * returns the output of this function.
 */
int device_ioctl(
    struct inode *inode,
    struct file *file,
    unsigned int ioctl_num,/* The number of the ioctl */
    unsigned long ioctl_param) /* The parameter to it */
{
  int i;
  char *temp;
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  char ch;
#endif

  /* Switch according to the ioctl called */
  switch (ioctl_num) {
    case IOCTL_SET_MSG:
      /* Receive a pointer to a message (in user space)
       * and set that to be the device's message. */

      /* Get the parameter given to ioctl by the process */
      temp = (char *) ioctl_param;

      /* Find the length of the message */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      get_user(ch, temp);
      for (i=0; ch && i<BUF_LEN; i++, temp++)
        get_user(ch, temp);
```

```
#else
      for (i=0; get_user(temp) && i<BUF_LEN; i++, temp++)
    ;
#endif

      /* Don't reinvent the wheel - call device_write */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      device_write(file, (char *) ioctl_param, i, 0);
#else
      device_write(inode, file, (char *) ioctl_param, i);
#endif
      break;

    case IOCTL_GET_MSG:
      /* Give the current message to the calling
       * process - the parameter we got is a pointer,
       * fill it. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      i = device_read(file, (char *) ioctl_param, 99, 0);
#else
      i = device_read(inode, file, (char *) ioctl_param,
                      99);
#endif
      /* Warning - we assume here the buffer length is
       * 100. If it's less than that we might overflow
       * the buffer, causing the process to core dump.
       *
       * The reason we only allow up to 99 characters is
       * that the NULL which terminates the string also
       * needs room. */

      /* Put a zero at the end of the buffer, so it
       * will be properly terminated */
      put_user('\0', (char *) ioctl_param+i);
      break;
    case IOCTL_GET_NTH_BYTE:
      /* This ioctl is both input (ioctl_param) and
       * output (the return value of this function) */
      return Message[ioctl_param];
      break;
  }

  return SUCCESS;
}


/* Module Declarations ************************** */


/* This structure will hold the functions to be called
```

```
 * when a process does something to the device we
 * created. Since a pointer to this structure is kept in
 * the devices table, it can't be local to
 * init_module. NULL is for unimplemented functions. */
struct file_operations Fops = {
  NULL,   /* seek */
  device_read,
  device_write,
  NULL,   /* readdir */
  NULL,   /* select */
  device_ioctl,   /* ioctl */
  NULL,   /* mmap */
  device_open,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  NULL,  /* flush */
#endif
  device_release  /* a.k.a. close */
};


/* Initialize the module - Register the character device */
int init_module()
{
  int ret_val;

  /* Register the character device (atleast try) */
  ret_val = module_register_chrdev(MAJOR_NUM,
                                   DEVICE_NAME,
                                   &Fops);

  /* Negative values signify an error */
  if (ret_val < 0) {
    printk ("%s failed with %d\n",
            "Sorry, registering the character device ",
            ret_val);
    return ret_val;
  }

  printk ("%s The major device number is %d.\n",
          "Registeration is a success",
          MAJOR_NUM);
  printk ("If you want to talk to the device driver,\n");
  printk ("you'll have to create a device file. \n");
  printk ("We suggest you use:\n");
  printk ("mknod %s c %d 0\n", DEVICE_FILE_NAME,
          MAJOR_NUM);
  printk ("The device file name is important, because\n");
  printk ("the ioctl program assumes that's the\n");
  printk ("file you'll use.\n");

  return 0;
```

```
}


/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{
  int ret;

  /* Unregister the device */
  ret = module_unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

  /* If there's an error, report it */
  if (ret < 0)
    printk("Error in module_unregister_chrdev: %d\n", ret);
}
ex chardev.h

/* chardev.h - the header file with the ioctl definitions.
 *
 * The declarations here have to be in a header file,
 * because they need to be known both to the kernel
 * module (in chardev.c) and the process calling ioctl
 * (ioctl.c)
 */
#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>



/* The major device number. We can't rely on dynamic
 * registration any more, because ioctls need to know
 * it. */
#define MAJOR_NUM 100


/* Set the message of the device driver */
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)
/* _IOR means that we're creating an ioctl command
 * number for passing information from a user process
 * to the kernel module.
 *
 * The first arguments, MAJOR_NUM, is the major device
 * number we're using.
 *
 * The second argument is the number of the command
 * (there could be several with different meanings).
 *
 * The third argument is the type we want to get from
```

```
 * the process to the kernel.
 */

/* Get the message of the device driver */
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
 /* This IOCTL is used for output, to get the message
  * of the device driver. However, we still need the
  * buffer to place the message in to be input,
  * as it is allocated by the process.
  */


/* Get the n'th byte of the message */
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
 /* The IOCTL is used for both input and output. It
  * receives from the user a number, n, and returns
  * Message[n]. */
/* The name of the device file */
#define DEVICE_FILE_NAME "char_dev"


#endif
ex ioctl.c

/* ioctl.c - the process to use ioctl's to control the
 * kernel module
 *
 * Until now we could have used cat for input and
 * output. But now we need to do ioctl's, which require
 * writing our own process.
 */

/* Copyright (C) 1998 by Ori Pomerantz */


/* device specifics, such as ioctl numbers and the
 * major device file. */
#include "chardev.h"


#include <fcntl.h>      /* open */
#include <unistd.h>     /* exit */
#include <sys/ioctl.h>  /* ioctl */


/* Functions for the ioctl calls */

ioctl_set_msg(int file_desc, char *message)
{
  int ret_val;
```

```
  ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);

  if (ret_val < 0) {
    printf ("ioctl_set_msg failed:%d\n", ret_val);
    exit(-1);
  }
}
ioctl_get_msg(int file_desc)
{
  int ret_val;
  char message[100];

  /* Warning - this is dangerous because we don't tell
   * the kernel how far it's allowed to write, so it
   * might overflow the buffer. In a real production
   * program, we would have used two ioctls - one to tell
   * the kernel the buffer length and another to give
   * it the buffer to fill
   */
  ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);

  if (ret_val < 0) {
    printf ("ioctl_get_msg failed:%d\n", ret_val);
    exit(-1);
  }

  printf("get_msg message:%s\n", message);
}



ioctl_get_nth_byte(int file_desc)
{
  int i;
  char c;

  printf("get_nth_byte message:");

  i = 0;
  while (c != 0) {
    c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);

    if (c < 0) {
      printf(
      "ioctl_get_nth_byte failed at the %d'th byte:\n", i);
      exit(-1);
    }

    putchar(c);
  }
  putchar('\n');
}
```

```
/* Main - Call the ioctl functions */
main()
{
  int file_desc, ret_val;
  char *msg = "Message passed by ioctl\n";

  file_desc = open(DEVICE_FILE_NAME, 0);
  if (file_desc < 0) {
    printf ("Can't open device file: %s\n",
            DEVICE_FILE_NAME);
    exit(-1);
  }

  ioctl_get_nth_byte(file_desc);
  ioctl_get_msg(file_desc);
  ioctl_set_msg(file_desc, msg);

  close(file_desc);
}
```

第6章 启动参数

在前面所给出的许多例子中，我们不得不把一些东西硬塞进内核模块中，如 /proc文件的文件名或者设备的主设备编号，这样我们就可以使用该设备的 ioctl命令。但这是与 Unix和Linux的宗旨背道而驰的，Unix和Linux提倡编写用户所习惯的易于使用的程序。

在程序或者内核模块开始工作之前，如果希望告诉它一些它所需要的信息，可以使用命令行参数。如果是内核模块，我们不需要使用 argc和argv —— 相反，我们还有更好的选择。我们可以在内核模块中定义一些全局变量，然后使用 insmod命令，它将替我们给这些变量赋值。

在下面这个内核模块中，我们定义了两个全局变量： str1和str2。用户所需做的全部工作就是编译该内核模块，然后运行 insmod str1=xxx str2=yyy。当调用init_module时，str1将指向字符串“ xxx ”，而str2将指向字符串“ yyy ”。

在版本2中，对这些参数不进行类型检查。如果 str1或者str2的第一个字符是一个数字，则内核将用该整数的值填充这个变量，而不会用指向字符串的指针去填充它。如果用户对此不太确定，那么就必须亲自去检查一下。

而另一方面，在版本2.2中，用户使用宏MACRO_PARM告诉insmod自己希望参数、它的名称以及类型是什么样的。这就解决了类型问题，并且允许内核模块接受那些以数字开头的字符串。

```
ex param.c
/* param.c
 *
 * Receive command line parameters at module installation
 */

/* Copyright (C) 1998-99 by Ori Pomerantz */




/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


#include <stdio.h>  /* I need NULL */
```

```
/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* Emmanuel Papirakis:
 *
 * Parameter names are now (2.2) handled in a macro.
 * The kernel doesn't resolve the symbol names
 * like it seems to have once did.
 *
 * To pass parameters to a module, you have to use a macro
 * defined in include/linux/modules.h (line 176).
 * The macro takes two parameters. The parameter's name and
 * it's type. The type is a letter in double quotes.
 * be a string.
 */


char *str1, *str2;


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
MODULE_PARM(str1, "s");
MODULE_PARM(str2, "s");
#endif


/* Initialize the module - show the parameters */
int init_module()
{
  if (str1 == NULL || str2 == NULL) {
    printk("Next time, do insmod param str1=<something>");
    printk("str2=<something>\n");
  } else
    printk("Strings:%s and %s\n", str1, str2);

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  printk("If you try to insmod this module twice,");
  printk("(without rmmod'ing\n");
  printk("it first), you might get the wrong");
  printk("error message:\n");
  printk("'symbol for parameters str1 not found'.\n");
#endif

  return 0;
}
```

```
}


/* Cleanup */
void cleanup_module()
{
}
```

第7章 系统调用

到现在为止，我们所做过的唯一的工作就是使用一个定义好的内核机制来注册 /proc文件和设备处理程序。如果用户仅仅希望做一个内核程序员份内的工作，例如编写设备驱动程序，那么以前我们所学的知识已经足够了。但是如果用户想做一些不平凡的事，比如在某些方面，在某种程度上改变一下系统的行为，那应该怎么办呢？答案是，几乎全部要靠自己。

这就是内核编程之所以危险的原因。在编写下面的例题时，我关掉了系统调用 open。这将意味着我不能打开任何文件，不能运行任何程序，甚至不能关闭计算机。我只能把电源开关拔掉。幸运的是，我没有删除掉任何文件。为了保证自己也不丢失任何文件，请读者在执行insmod和rmmod之前先运行sync。

现在让我们忘记/proc文件，忘记设备文件，他们只不过是无关大雅的细节问题。实现内核通信机制的“真正”进程是系统调用，它是被所有进程所使用的进程。当某进程向内核请求服务时(例如打开文件、产生一个新进程、或者请求更多的内存)，它所使用的机制就是系统调用。如果用户希望以一种有趣的方式改变内核的行为，也需要依靠系统调用。顺便说一下，如果用户想知道程序使用的是哪个系统调用，可以运行命令 strace <command> <arguments>。

一般来说，进程是不能访问内核的。它不能访问内核存储，它也不能调用内核函数。CPU的硬件保证了这一点(那就是为什么称之为“保护模式”的原因)。系统调用是这条通用规则的一个特例。在进行系统调用时，进程以适当的值填充注册程序，然后调用一条特殊的指令，而这条指令是跳转到以前定义好的内核中的某个位置 (当然，用户进程可以读那个位置，但却不能对它进行写的操作)。在Intel CPU下，以上任务是通过中断0x80来完成的。硬件知道一旦跳转到这个位置，用户的进程就不再是在受限制的用户模式下运行了，而是作为操作系统内核来运行——于是用户就被允许干所有他想干的事。

进程可以跳转到的那个内核中的位置称为 system_call。那个位置上的过程检查系统调用编号(系统调用编号可以告诉内核进程所请求的是什么服务)。然后，该过程查看系统调用表(sys_call_table)，找出想要调用的内核函数的地址，然后调用那个函数。在函数返回之后，该过程还要做一些系统检查工作，然后再返回到进程 (或者如果进程时间已用完，则返回到另一个进程)。如果读者想读这段代码，可以查看源文件 arch/<architecture>/kernel/entry.S，它就在ENTRY(system_call)那一行的后面。

这样看来，如果我们想要改变某个系统调用的工作方式，我们需要编写自己的函数来实现它(通常是加入一点自己的代码，然后再调用原来的函数)，然后改变sys_call_table表中的指针，使它指向我们的函数。因为我们的函数将来可能会被删除掉，而我们不想使系统处于一个不稳定的状态，所以必须用 cleanup_module使sys_call_table表恢复到它的原始状态，这是很重要的。

本章的源代码就是这样一个内核模块的例子。我们想要“侦听”某个特定的用户。无论何时，只要那个用户一打开某个文件，程序就会用 printk打印出一个消息。为了做到这一点，我们用自己的函数代替了用来打开文件的那个系统调用。我们的函数称为 our_sys_open。该函

数查看当前进程的uid(用户的ID)，如果它就是我们想要侦听的 uid，它就调用printk，显示出将要打开的文件的名称。接下来，不管当前进程的 uid是不是想要侦听的uid，该函数都使用同样的参数调用原来的open函数，真正地打开那个文件。

init_module函数替换sys_call_table表中相应的位置，并把原来的指针存放在一个变量中；而cleanup_module函数则使用那个变量把一切都恢复成原来正常的状态。这种方法是具有一定的危险性的，因为可能有两个内核模块都修改同一个系统调用。现在假设有两个内核模块A和B。A模块打开文件的系统调用是 A_open，而B的系统调用是B_open。当把A插入到内核中时，系统调用被换成了A_open，它在被调用时将会调用原来的sys_open。接下来，当B被插入到内核中时，系统调用将被替换成 B_open，它在被调用时，将会调用它自以为是原始系统调用的那个系统调用，即A_open。

现在假设B先被删除，那么一切都将正常 —— 系统调用将被恢复成A_open，而A_open会调用原始的系统调用。然而，如果 A先被删除然后B被删除，系统将会崩溃。删除 A时系统调用被恢复成原始的sys_open, B_open将被忽略。接着，当删除 B时，B将把系统调用恢复成它自认为是原始的系统调用，即 A_open，而A_open已经不再位于内存中了。乍一看上去，好象用户可以检查系统调用是不是打开文件函数，如果是就不做任何修改 (这样在删除B时它就不会改变系统调用)，似乎这样可以避免问题的发生。但这样做将会导致一个更为严重的问题。当A被删除时，它看到系统调用已经被改变为 B_open，而不再指向A_open，所以A在从内存中删除前不会将系统调用恢复为 sys_open。不幸的是， B_open仍将试图调用 A_open，而A_open已不在内存中了，这样，甚至还不到删除 B时系统就将崩溃。

我认为有两种方法可以解决这个问题。第一个方法是把系统调用恢复为原始的值：sys_open。不幸的是，sys_open不是/proc/ksyms中内核系统表的一部分，所以我们不能访问它。另一个方法是一旦装入了模块，马上设立一个引用计数器，以防止根用户把它 rmmod掉。对于产品模块来说，这样做是很好的，但却不适合于作为教学的例子 —— 这就是我没有在这里实现它的原因。

```
ex syscall.c

/* syscall.c
 *
 * System call "stealing" sample
 */


/* Copyright (C) 1998-99 by Ori Pomerantz */


/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
```

```
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <sys/syscall.h>  /* The list of system calls */

/* For the current (process) structure, we need
 * this to know who the current user is. */
#include <linux/sched.h>



/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>
#endif



/* The system call table (a table of functions). We
 * just define this as external, and the kernel will
 * fill it up for us when we are insmod'ed
 */
extern void *sys_call_table[];


/* UID we want to spy on - will be filled from the
 * command line */
int uid;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
MODULE_PARM(uid, "i");
#endif

/* A pointer to the original system call. The reason
 * we keep this, rather than call the original function
 * (sys_open), is because somebody else might have
 * replaced the system call before us. Note that this
 * is not 100% safe, because if another module
 * replaced sys_open before us, then when we're inserted
 * we'll call the function in that module--and it
 * might be removed before we are.
 *
 * Another reason for this is that we can't get sys_open.
 * It's a static variable, so it is not exported. */
```

```
asmlinkage int (*original_call)(const char *, int, int);


/* For some reason, in 2.2.3 current->uid gave me
 * zero, not the real user ID. I tried to find what went
 * wrong, but I couldn't do it in a short time, and
 * I'm lazy - so I'll just use the system call to get the
 * uid, the way a process would.
 *
 * For some reason, after I recompiled the kernel this
 * problem went away.
 */
asmlinkage int (*getuid_call)();
/* The function we'll replace sys_open (the function
 * called when you call the open system call) with. To
 * find the exact prototype, with the number and type
 * of arguments, we find the original function first
 * (it's at fs/open.c).
 *
 * In theory, this means that we're tied to the
 * current version of the kernel. In practice, the
 * system calls almost never change (it would wreck havoc
 * and require programs to be recompiled, since the system
 * calls are the interface between the kernel and the
 * processes).
 */
asmlinkage int our_sys_open(const char *filename,
                            int flags,
                            int mode)
{
  int i = 0;
  char ch;

  /* Check if this is the user we're spying on */
  if (uid == getuid_call()) {
   /* getuid_call is the getuid system call,
    * which gives the uid of the user who
    * ran the process which called the system
    * call we got */

    /* Report the file, if relevant */
    printk("Opened file by %d: ", uid);
    do {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      get_user(ch, filename+i);
#else
      ch = get_user(filename+i);
#endif
      i++;
      printk("%c", ch);
    } while (ch != 0);
```

```
      printk("\n");
    }

    /* Call the original sys_open - otherwise, we lose
     * the ability to open files */
    return original_call(filename, flags, mode);
}
/* Initialize the module - replace the system call */
int init_module()
{
    /* Warning - too late for it now, but maybe for
     * next time... */
    printk("I'm dangerous. I hope you did a ");
    printk("sync before you insmod'ed me.\n");
    printk("My counterpart, cleanup_module(), is even");
    printk("more dangerous. If\n");
    printk("you value your file system, it will ");
    printk("be \"sync; rmmod\" \n");
    printk("when you remove this module.\n");

    /* Keep a pointer to the original function in
     * original_call, and then replace the system call
     * in the system call table with our_sys_open */
    original_call = sys_call_table[__NR_open];
    sys_call_table[__NR_open] = our_sys_open;

    /* To get the address of the function for system
     * call foo, go to sys_call_table[__NR_foo]. */

    printk("Spying on UID:%d\n", uid);

    /* Get the system call for getuid */
    getuid_call = sys_call_table[__NR_getuid];

    return 0;
}


/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{
    /* Return the system call back to normal */
    if (sys_call_table[__NR_open] != our_sys_open) {
        printk("Somebody else also played with the ");
        printk("open system call\n");
        printk("The system may be left in ");
        printk("an unstable state.\n");
    }

    sys_call_table[__NR_open] = original_call;
}
```

第8章 阻塞处理

如果有人想让你做一些目前无法做到的事，你会怎么处理呢？如果打扰你的是某个人的话，你可以说的唯一的一句话是：“现在不行，我很忙，你走吧！”但是如果是进程让内核模块做一些目前它无法处理的事，内核模块却可以有另一种处理方法。内核可以让进程睡眠，直到能够为它服务为止。毕竟，随时都会有进程被内核置为睡眠状态或者唤醒 (这就是多个进程看上去好象同时在一个CPU上运行的道理)。

本章的内核模块就是这样的一个例子。该文件 (名为/proc/sleep)每次只能被一个进程所打开。如果文件早已经被打开，内核模块将调用 module_interruptible_sleep_on。这个函数把任务的状态改为TASK_INTERRUPTIBLE(任务是内核数据结构，它包含着有关它所处的进程和系统调用的信息，如果存在系统调用的话)，把它加入到WaitQ当中，这就意味着任务在被唤醒之前将不会运行。WaitQ是等待访问文件的任务队列。然后，函数调用上下文调度程度，切换到另一个拥有CPU时间的进程。

在进程结束了文件操作以后，进程将关闭文件，并调用 module_close。该函数将唤醒队列中的所有进程(没有只唤醒一个进程的机制)，然后函数返回，刚刚关闭文件的那个进程就可以继续运行了。如果那个进程的时间片用完了，则调度程度将会及时判断出这一点，并将 CPU的控制权交给另一个进程。最后，等待队列中的某个进程将会被调度程序授予 CPU的控制权。该进程将从紧接着调用 module_interruptible_sleep_on后面的那个点开始执行。然后它会设置一个全局变量，告诉其它所有进程文件依然打开着，然后该进程将继续执行。当其它进程获得CPU时间片时，它们将看到那个全局变量，于是继续睡眠。

更为有趣的是，并不是只有 module_close才能唤醒那些等待访问文件的进程。一个信号，例如Ctrl+C(SIGINT)也可以唤醒进程。在那种情况下，我们希望立刻用 EINTR返回。这是很重要的，只有这样用户才能在进程接收到文件之前杀死那个进程。

还需要记住一点，有时候进程并不想睡眠；它们希望或者立刻拿到它们想要的东西，或者直接告诉它们这是不可能的。这样的进程在打开文件时使用标志 O_NONBLOCK。内核在进行该操作时，将会返回错误代码 EAGAIN来作响应，否则该操作就将阻塞，就像下面例子中的打开文件操作一样。本章的源目录中有一个程序 cat_noblock，可以用于带O_NONBLOCK标志打开一个文件。

```
ex sleep.c

/* sleep.c - create a /proc file, and if several
 * processes try to open it at the same time, put all
 * but one to sleep */


/* Copyright (C) 1998-99 by Ori Pomerantz */


/* The necessary header files */
```

```
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necessary because we use proc fs */
#include <linux/proc_fs.h>

/* For putting processes to sleep and waking them up */
#include <linux/sched.h>
#include <linux/wrapper.h>
/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for get_user and put_user */
#endif


/* The module's file functions ********************** */


/* Here we keep the last message received, to prove
 * that we can process our input */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];


/* Since we use the file operations struct, we can't use
 * the special proc output provisions - we have to use
 * a standard read function, which is this function */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_output(
    struct file *file,    /* The file read */
    char *buf, /* The buffer to put data to (in the
                * user segment) */
    size_t len,  /* The length of the buffer */
    loff_t *offset) /* Offset in the file - ignore */
#else
static int module_output(
    struct inode *inode, /* The inode read */
```

```
      struct file *file,   /* The file read */
      char *buf, /* The buffer to put data to (in the
                  * user segment) */
      int len)  /* The length of the buffer */
  #endif
  {
    static int finished = 0;
    int i;
    char message[MESSAGE_LENGTH+30];
  /* Return 0 to signify end of file - that we have
   * nothing more to say at this point. */
   if (finished) {
     finished = 0;
     return 0;
   }

   /* If you don't understand this by now, you're
    * hopeless as a kernel  programmer. */
   sprintf(message, "Last input:%s\n", Message);
   for(i=0; i<len && message[i]; i++)
     put_user(message[i], buf+i);

   finished = 1;
   return i;  /* Return the number of bytes "read" */
}


/* This function receives input from the user when
 * the user writes to the /proc file. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_input(
    struct file *file,    /* The file itself */
    const char *buf,      /* The buffer with input */
    size_t length,        /* The buffer's length */
    loff_t *offset)       /* offset to file - ignore */
#else
static int module_input(
    struct inode *inode, /* The file's inode */
    struct file *file,   /* The file itself */
    const char *buf,     /* The buffer with the input */
    int length)          /* The buffer's length */
#endif
{
  int i;

  /* Put the input into Message, where module_output
   * will later be able to use it */
  for(i=0; i<MESSAGE_LENGTH-1 && i<length; i++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    get_user(Message[i], buf+i);
#else
```

```
    Message[i] = get_user(buf+i);
#endif
/* we want a standard, zero terminated string */
    Message[i] = '\0';
  /* We need to return the number of input
   * characters used */
  return i;
}

/* 1 if the file is currently open by somebody */
int Already_Open = 0;

/* Queue of processes who want our file */
static struct wait_queue *WaitQ = NULL;


/* Called when the /proc file is opened */
static int module_open(struct inode *inode,
                       struct file *file)
{
  /* If the file's flags include O_NONBLOCK, it means
   * the process doesn't want to wait for the file.
   * In this case, if the file is already open, we
   * should fail with -EAGAIN, meaning "you'll have to
   * try again", instead of blocking a process which
   * would rather stay awake. */
  if ((file->f_flags & O_NONBLOCK) && Already_Open)
    return -EAGAIN;

  /* This is the correct place for MOD_INC_USE_COUNT
   * because if a process is in the loop, which is
   * within the kernel module, the kernel module must
   * not be removed. */
  MOD_INC_USE_COUNT;

  /* If the file is already open, wait until it isn't */
  while (Already_Open)
  {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    int i, is_sig=0;
#endif

    /* This function puts the current process,
     * including any system calls, such as us, to sleep.
     * Execution will be resumed right after the function
     * call, either because somebody called
     * wake_up(&WaitQ) (only module_close does that,
     * when the file is closed) or when a signal, such
     * as Ctrl-C, is sent to the process */
    module_interruptible_sleep_on(&WaitQ);
```

```
    /* If we woke up because we got a signal we're not
     * blocking, return  -EINTR (fail the system call).
     * This allows processes to be killed or stopped. */


/*
 * Emmanuel Papirakis:
 *
 * This is a little update to work with 2.2.*. Signals
 * now are contained in two words (64 bits) and are
 * stored in a structure that contains an array of two
 * unsigned longs. We now have to make 2 checks in our if.
 *
 * Ori Pomerantz:
 *
 * Nobody promised me they'll never use more than 64
 * bits, or that this book won't be used for a version
 * of Linux with a word size of 16 bits. This code
 * would work in any case.
 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)

    for(i=0; i<_NSIG_WORDS && !is_sig; i++)
      is_sig = current->signal.sig[i] &
        ~current->blocked.sig[i];
    if (is_sig) {
#else
    if (current->signal & ~current->blocked) {
#endif
      /* It's important to put MOD_DEC_USE_COUNT here,
       * because for processes where the open is
       * interrupted there will never be a corresponding
       * close. If we don't decrement the usage count
       * here, we will be left with a positive usage
       * count which we'll have no way to bring down to
       * zero, giving us an immortal module, which can
       * only be killed by rebooting the machine. */
      MOD_DEC_USE_COUNT;
      return -EINTR;
    }
  }

  /* If we got here, Already_Open must be zero */
  /* Open the file */
  Already_Open = 1;
  return 0;  /* Allow the access */
}

/* Called when the /proc file is closed */
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
int module_close(struct inode *inode, struct file *file)
#else
void module_close(struct inode *inode, struct file *file)
#endif
{
  /* Set Already_Open to zero, so one of the processes
   * in the WaitQ will be able to set Already_Open back
   * to one and to open the file. All the other processes
   * will be called when Already_Open is back to one, so
   * they'll go back to sleep. */
  Already_Open = 0;

  /* Wake up all the processes in WaitQ, so if anybody
   * is waiting for the file, they can have it. */
  module_wake_up(&WaitQ);

  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;  /* success */
#endif
}



/* This function decides whether to allow an operation
 * (return zero) or not allow it (return a non-zero
 * which indicates why it is not allowed).
 *
 * The operation can be one of the following values:
 * 0 - Execute (run the "file" - meaningless in our case)
 * 2 - Write (input to the kernel module)
 * 4 - Read (output from the kernel module)
 *
 * This is the real function that checks file
 * permissions. The permissions returned by ls -l are
 * for reference only, and can be overridden here.
 */
static int module_permission(struct inode *inode, int op)
{
  /* We allow everybody to read from our module, but
   * only root (uid 0) may write to it */
  if (op == 4 || (op == 2 && current->euid == 0))
    return 0;

  /* If it's anything else, access is denied */
  return -EACCES;
}
```

```
/* Structures to register as the /proc file, with
 * pointers to all the relevant functions. ********** */


/* File operations for our proc file. This is where
 * we place pointers to all the functions called when
 * somebody tries to do something to our file. NULL
 * means we don't want to deal with something. */
static struct file_operations File_Ops_4_Our_Proc_File =
  {
    NULL,  /* lseek */
    module_output,  /* "read" from the file */
    module_input,   /* "write" to the file */
    NULL,  /* readdir */
    NULL,  /* select */
    NULL,  /* ioctl */
    NULL,  /* mmap */
    module_open,/* called when the /proc file is opened */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    NULL,   /* flush */
#endif
    module_close      /* called when it's classed */
  };



/* Inode operations for our proc file. We need it so
 * we'll have somewhere to specify the file operations
 * structure we want to use, and the function we use for
 * permissions. It's also possible to specify functions
 * to be called for anything else which could be done to an
 * inode (although we don't bother, we just put NULL). */
static struct inode_operations Inode_Ops_4_Our_Proc_File =
  {
    &File_Ops_4_Our_Proc_File,
    NULL, /* create */
    NULL, /* lookup */
    NULL, /* link */
    NULL, /* unlink */
    NULL, /* symlink */
    NULL, /* mkdir */
    NULL, /* rmdir */
    NULL, /* mknod */
    NULL, /* rename */
    NULL, /* readlink */
    NULL, /* follow_link */
    NULL, /* readpage */
    NULL, /* writepage */
    NULL, /* bmap */
    NULL, /* truncate */
```

```
    module_permission /* check for permissions */
  };

/* Directory entry */
static struct proc_dir_entry Our_Proc_File =
  {
    0, /* Inode number - ignore, it will be filled by
        * proc_register[_dynamic] */
    5, /* Length of the file name */
    "sleep", /* The file name */
    S_IFREG | S_IRUGO | S_IWUSR,
    /* File mode - this is a regular file which
     * can be read by its owner, its group, and everybody
     * else. Also, its owner can write to it.
     *
     * Actually, this field is just for reference, it's
     * module_permission that does the actual check. It
     * could use this field, but in our implementation it
     * doesn't, for simplicity. */
    1,  /* Number of links (directories where the
         * file is referenced) */
    0, 0,  /* The uid and gid for the file - we give
            * it to root */
    80, /* The size of the file reported by ls. */
    &Inode_Ops_4_Our_Proc_File,
    /* A pointer to the inode structure for
     * the file, if we need it. In our case we
     * do, because we need a write function. */
    NULL  /* The read function for the file.
           * Irrelevant, because we put it
           * in the inode structure above */
  };

/* Module initialization and cleanup **************** */


/* Initialize the module - register the proc file */
int init_module()
{
  /* Success if proc_register_dynamic is a success,
   * failure otherwise */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif

  /* proc_root is the root directory for the proc
   * fs (/proc). This is where we want our file to be
   * located.
   */
```

```
}


/* Cleanup - unregister our file from /proc. This could
 * get dangerous if there are still processes waiting in
 * WaitQ, because they are inside our open function,
 * which will get unloaded. I'll explain how to avoid
 * removal of a kernel module in such a case in
 * chapter 10. */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

第9章　替换printk

在第1章，我曾经提到过X编程和内核模块编程不能混为一谈。这句话在开发内核模块时是正确的。但是在实际应用中，用户可能希望向运行 tty命令的那个模块发送消息。在释放内核模块以后，这对于识别错误是非常重要的。因为该内核模块将会被所有使用 tty命令的模块所使用。

通过使用current可以做到这一点， current是一个指向当前正在运行的任务的指针，通过使用它可以获得当前任务的tty结构。然后查看tty结构，可以找到指向写字符串的函数的指针，我们就是用这个指针向tty写字符串的。

```
ex printk.c

/* printk.c - send textual output to the tty you're
 * running on, regardless of whether it's passed
 * through X11, telnet, etc. */


/* Copyright (C) 1998 by Ori Pomerantz */


/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work
*/
#include <linux/module.h>   /* Specifically, a module */
/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necessary here */
#include <linux/sched.h>    /* For current */
#include <linux/tty.h>      /* For the tty declarations */


/* Print the string to the appropriate tty, the one
 * the current task uses */
void print_string(char *str)
{
  struct tty_struct *my_tty;

  /* The tty for the current task */
  my_tty = current->tty;
```

```
/* If my_tty is NULL, it means that the current task
 * has no tty you can print to (this is possible, for
 * example, if it's a daemon). In this case, there's
 * nothing we can do. */
if (my_tty != NULL) {

  /* my_tty->driver is a struct which holds the tty's
   * functions, one of which (write) is used to
   * write strings to the tty. It can be used to take
   * a string either from the user's memory segment
   * or the kernel's memory segment.
   *
   * The function's first parameter is the tty to
   * write to, because the  same function would
   * normally be used for all tty's of a certain type.
   * The second parameter controls whether the
   * function receives a string from kernel memory
   * (false, 0) or from user memory (true, non zero).
   * The third parameter is a pointer to a string,
   * and the fourth parameter is the length of
   * the string.
   */
  (*(my_tty->driver).write)(
      my_tty, /* The tty itself */
      0, /* We don't take the string from user space */
  str, /* String */
  strlen(str));  /* Length */

  /* ttys were originally hardware devices, which
   * (usually) adhered strictly to the ASCII standard.
   * According to ASCII, to move to a new line you
   * need two characters, a carriage return and a
   * line feed. In Unix, on the other hand, the
   * ASCII line feed is used for both purposes - so
   * we can't just use \n, because it wouldn't have
   * a carriage return and the next line will
   * start at the column right
   *                          after the line feed.
   *
   * BTW, this is the reason why the text file
   *  is different between Unix and Windows.
   * In CP/M and its derivatives, such as MS-DOS and
   * Windows, the ASCII standard was strictly
   * adhered to, and therefore a new line requires
   * both a line feed and a carriage return.
   */
  (*(my_tty->driver).write)(
    my_tty,
    0,
    "\015\012",
    2);
```

```
  }
}


/* Module initialization and cleanup ****************** */


/* Initialize the module - register the proc file */
int init_module()
{
  print_string("Module Inserted");

  return 0;
}


/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  print_string("Module Removed");
}
```

第10章 任务调度

常常有一些“内务处理”任务需要我们定时或者经常去做。如果任务是由进程去完成的，我们可以把它放在crontab文件中。如果任务要由内核模块来完成，那么我们将面临两种选择。第一种方案是把一个进程放在 crontab文件中，该进程在需要的时候通过系统调用唤醒模块。例如，通过打开一个文件来做到这一点。第三种方案需要在 crontab中运行一个新进程，把一个新的可执行程序读入内存，而这一切都只是为了唤醒一个早已经位于内存中的内核模块这样做效率是非常低的。

除了以上这两种方法以外，我们还可以创建一个函数，在每次时钟中断时调用一次那个函数。为了做到这一点，需要创建一个任务，存放在结构 tq_struct中，而该结构将存放指向函数的指针。接着，我们使用 queue_task把那个任务放置到一个称为 tq_timer的任务列表中，该任务列表中的任务都将在下次时钟中断时执行。由于我们希望该函数能继续执行下去，无论该函数何时被调用，我们一定要把它放回到 tq_timer中，这样在下一次时钟中断时它还可以执行。

在这里还需要记住一点。当使用 rmmod命令删除一个模块时，首先需要检查它的引用计数器值。如果计数器的值为 0，则调用module_cleanup。然后，该模块以及它的所有函数就被从内存中删除掉了。没有人会记得检查一下看看时钟的任务列表中是否碰巧包含了一个指向这些函数中某个函数的指针，而该函数已经不再是可用的了。很长一段时间以后 (这是从计算机的角度来说的，从人的角度来看这段时间不值一提，有可能比百分之一秒还短)，内核发生了一次时钟中断，并试图调用任务列表中的函数。不幸的是，该函数早已经不在那里了。在大多数情况下，该函数原来所在的内存页还没有被使用，这时用户得到的将是一段难看的错误信息。但如果那个内存位置已经存放了一些新的其它的代码，事情就变得非常糟糕了。不幸的是，我们还没有一种简便的方法可以从任务列表中取消一个任务的注册。

由于cleanup_module不能返回错误代码(它是一个void函数)，解决的方法是根本不让它返回，相反，它调用 sleep_on或者module_sleep_on，把rmmod进程置为睡眠状态。而在此之前，它会设置一个全局变量，通知那些将要在时钟中断时调用的函数停止连接。然后，在下一次时钟中断发生时， rmmod进程将被唤醒，我们的函数已经不在队列中了，这时就可以安全地删除模块。

```
ex sched.c

/* sched.c - scheduale a function to be called on
 * every timer interrupt. */


/* Copyright (C) 1998 by Ori Pomerantz */


/* The necessary header files */

/* Standard in kernel modules */
```

```
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necessary because we use the proc fs */
#include <linux/proc_fs.h>

/* We scheduale tasks here */
#include <linux/tqueue.h>

/* We also need the ability to put ourselves to sleep
 * and wake up later */
#include <linux/sched.h>

/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif
/* The number of times the timer interrupt has been
 * called so far */
static int TimerIntrpt = 0;


/* This is used by cleanup, to prevent the module from
 * being unloaded while intrpt_routine is still in
 * the task queue */
static struct wait_queue *WaitQ = NULL;

static void intrpt_routine(void *);


/* The task queue structure for this task, from tqueue.h */
static struct tq_struct Task = {
  NULL,   /* Next item in list - queue_task will do
           * this for us */
  0,      /* A flag meaning we haven't been inserted
           * into a task queue yet */
  intrpt_routine, /* The function to run */
  NULL    /* The void* parameter for that function */
};

/* This function will be called on every timer
 * interrupt. Notice the void* pointer - task functions
```

```
 * interrupt. Notice the void* pointer - task functions
 * can be used for more than one purpose, each time
 * getting a different parameter. */
static void intrpt_routine(void *irrelevant)
{
  /* Increment the counter */
  TimerIntrpt++;

  /* If cleanup wants us to die */
  if (WaitQ != NULL)
    wake_up(&WaitQ);   /* Now cleanup_module can return */
  else
    /* Put ourselves back in the task queue */
    queue_task(&Task, &tq_timer);
}
/* Put data into the proc fs file. */
int procfile_read(char *buffer,
                  char **buffer_location, off_t offset,
                  int buffer_length, int zero)
{
  int len;  /* The number of bytes actually used */

  /* This is static so it will still be in memory
   * when we leave this function */
  static char my_buffer[80];

  static int count = 1;

  /* We give all of our information in one go, so if
   * the anybody asks us if we have more information
   * the answer should always be no.
   */
  if (offset > 0)
    return 0;

  /* Fill the buffer and get its length */
  len = sprintf(my_buffer,
                "Timer was called %d times so far\n",
                TimerIntrpt);
  count++;

  /* Tell the function which called us where the
   * buffer is */
  *buffer_location = my_buffer;

  /* Return the length */
  return len;
}


struct proc_dir_entry Our_Proc_File =
```

```
{
  0, /* Inode number - ignore, it will be filled by
      * proc_register_dynamic */
  5, /* Length of the file name */
  "sched", /* The file name */
  S_IFREG | S_IRUGO,
  /* File mode - this is a regular file which can
   * be read by its owner, its group, and everybody
   * else */
  1,  /* Number of links (directories where
         * the file is referenced) */
    0, 0,  /* The uid and gid for the file - we give
            * it to root */
    80, /* The size of the file reported by ls. */
    NULL, /* functions which can be done on the
           * inode (linking, removing, etc.) - we don't
           * support any. */
    procfile_read,
    /* The read function for this file, the function called
     * when somebody tries to read something from it. */
    NULL
    /* We could have here a function to fill the
     * file's inode, to enable us to play with
     * permissions, ownership, etc. */
  };


/* Initialize the module - register the proc file */
int init_module()
{
  /* Put the task in the tq_timer task queue, so it
   * will be executed at next timer interrupt */
  queue_task(&Task, &tq_timer);

  /* Success if proc_register_dynamic is a success,
   * failure otherwise */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif
}


/* Cleanup */
void cleanup_module()
{
  /* Unregister our /proc file */
  proc_unregister(&proc_root, Our_Proc_File.low_ino);

  /* Sleep until intrpt_routine is called one last
```

```
 * time. This is necessary, because otherwise we'll
 * deallocate the memory holding intrpt_routine and
 * Task while tq_timer still references them.
 * Notice that here we don't allow signals to
 * interrupt us.
    *
    * Since WaitQ is now not NULL, this automatically
    * tells the interrupt routine it's time to die. */
  sleep_on(&WaitQ);
}
```

第11章　中断处理程序

除了第10章以外，到目前为止我们在内核中所做的所有工作都是为了响应进程的请求，或者是处理一个特殊的文件，发送 ioctl，或者是发出一个系统调用。但是内核的任务并不仅仅是为了响应进程请求。另一个任务也是同样重要的，那就是内核还需要和与机器相连接的硬件进行通信。

在CPU和计算机的其它硬件之间有两种相互交互的方式。第一种是 CPU向硬件发出命令，另一种是硬件需要告诉 CPU一些事情。第二种方式称之为中断，相比较而言，中断要难实现得多，这是因为它需要在硬件方便的时候进行处理，而不是在 CPU方便的时候去处理。一般来说，每个硬件设备都会有一个数量相当少的 RAM，如果在可以读它们的信息的时候用户没有去读，则这些信息将丢失。

在Linux下，硬件中断称为IRQ(即Interrupt Request的简称，中断请求)。IRQ分为两种类型：短的和长的，短IRQ是指需要的时间周期非常短，在这段时间内，机器的其它部分将被阻塞，并且不再处理其它的中断。而长 IRQ是指需要的时间相对长一些，在这段时间内也可能会发生别的中断(但是同一个设备上不会再产生中断)。如果有可能的话，中断处理程序还是处理长IRQ要好一些。

当CPU接收到一个中断时，它将停下手中所有的工作 (除非它正在处理一个更为重要的中断，在那种情况下，只有处理完那个更重要的中断以后， CPU才会去处理这个中断)，在栈中保存某些特定的参数，并调用中断处理程序。这就意味着在中断处理程序内部有些特定的事情是不允许的，因为系统处于一个未知的状态下。为了解决这个问题，中断处理程序应该做那些需要立刻去做的事情，通常是从硬件读一些信息或者向硬件发送一些信息，在稍后的某时刻再调度去做新信息的处理工作 (这称为“ 底半处理 ”)并返回。内核必须保证尽快调用底半处理 —— 在进行底半处理工作时，内核模块中允许做的所有工作都可以做。

要实现这一点，可以调用 request_irq，这样一来，在接收到相关 IRQ时，就可以调用用户的中断处理程序 (在Intel平台上共有16种IRQ)。request_irq接收IRQ编号、函数名称、标志、/proc/interrupts的名称以及传送给中断处理函数的一个参数。这里提到的标志包括 SA_SHIRQ和SA_INTERRUPT，前者表明用户愿意与其它中断处理程序分享 IRQ(通常是因为同一个 IRQ上有多个硬件设备)；而后者表明这是个高速中断。只有当这个 IRQ上还没有处理程序，或者两者都愿意共享这个IRQ时，request_irq函数才会成功。

接下来，我们从中断处理程序内部与硬件进行通信，并使用 tq_immediate和mark_bh(BH_IMMEDIATE)调用queue_task_irq，以调度底半处理。在版本2中我们不能使用标准的queue_task，这是因为中断有可能正好发生在其它的 queue_task中间。我们之所以需要使用mark_bh，是因为以前版本的Linux只有一个由32个底半处理所组成的队列，而现在它们中的一个(BH_IMMEDIATE)已经被用作底半处理的链接列表，这是为那些没有得到分配给它们的底半处理入口的驱动程序所准备的。

Intel体系结构的键盘

警告 本章余下的内容完全是与Intel相关的。如果读者不是在Intel平台上运行，这些内容将不能正常工作。读者甚至不要去编译这些代码。

在写本章的示例代码时我遇到了一个问题。一方面，为了让所给出的例子能有用，它必须可以在所有人的计算机上运行，且能得到有意义的结果。但是另一方面，内核中早已经为所有的通用设备准备了设备驱动程序，而那些设备驱动程序与我将要编写的驱动程序是不能共存的。最后我找到了解决的办法，就是为键盘中断写驱动程序，并且首先关闭常规的键盘中断驱动程序。因为它被定义成内核源文件 (特别是指drivers/char/keyboard.c)中的静态符号，所以没有办法恢复它。如果用户珍惜自己的文件系统的话，在执行 insmod命令插入这个代码以前，请先在另一个终端上执行 sleep 120;reboot。

该代码把它自己绑定在IRQ 1上，在Intel体系结构下，IRQ1是控制键盘的IRQ。这样，当它接收到键盘中断时，它读键盘的状态 (在程序中使用 inb (0x64)来实现)，并查看代码，该代码是由键盘所返回的值。然后在内核认为适合的时候，它立刻运行 got_char，该函数将给出所使用的键的代码 (即查看到的代码的前七位)，并给出该键是被按下 (如果第八位为 0)还是被松开(如果第八位为1)。

```
ex intrpt.c

/* intrpt.c - An interrupt handler. */


/* Copyright (C) 1998 by Ori Pomerantz */



/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/sched.h>
#include <linux/tqueue.h>

/* We want an interrupt */
#include <linux/interrupt.h>

#include <asm/io.h>


/* In 2.2.3 /usr/include/linux/version.h includes a
```

```
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* Bottom Half - this will get called by the kernel
 * as soon as it's safe to do everything normally
 * allowed by kernel modules. */
static void got_char(void *scancode)
{
  printk("Scan Code %x %s.\n",
    (int) *((char *) scancode) & 0x7F,
    *((char *) scancode) & 0x80 ? "Released" : "Pressed");
}
/* This function services keyboard interrupts. It reads
 * the relevant information from the keyboard and then
 * schedules the bottom half to run when the kernel
 * considers it safe. */
void irq_handler(int irq,
                 void *dev_id,
                 struct pt_regs *regs)
{
  /* This variables are static because they need to be
   * accessible (through pointers) to the bottom
   * half routine. */
  static unsigned char scancode;
  static struct tq_struct task =
        {NULL, 0, got_char, &scancode};
  unsigned char status;

  /* Read keyboard status */
  status = inb(0x64);
  scancode = inb(0x60);

  /* Scheduale bottom half to run */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  queue_task(&task, &tq_immediate);
#else
  queue_task_irq(&task, &tq_immediate);
#endif
  mark_bh(IMMEDIATE_BH);
}



/* Initialize the module - register the IRQ handler */
int init_module()
{
```

```
    /* Since the keyboard handler won't co-exist with
     * another handler, such as us, we have to disable
     * it (free its IRQ) before we do anything. Since we
     * don't know where it is, there's no way to
     * reinstate it later - so the computer will have to
     * be rebooted when we're done.
     */
    free_irq(1, NULL);

    /* Request IRQ 1, the keyboard IRQ, to go to our
     * irq_handler. */
    return request_irq(
    1,  /* The number of the keyboard IRQ on PCs */
    irq_handler,  /* our handler */
    SA_SHIRQ,
    /* SA_SHIRQ means we're willing to have other
     * handlers on this IRQ.
     *
     * SA_INTERRUPT can be used to make the
     * handler into a fast interrupt.
     */
    "test_keyboard_irq_handler", NULL);
}


/* Cleanup */
void cleanup_module()
{
  /* This is only here for completeness. It's totally
   * irrelevant, since we don't have a way to restore
   * the normal keyboard interrupt so the computer
   * is completely useless and has to be rebooted. */
  free_irq(1, NULL);
}
```

第12章　对称多处理

要提高硬件的性能，最简单的(同时也是最便宜的)方法是把多个CPU放到一个主板上。实现这一点，可以有两种方法。一是使不同的CPU执行不同的任务(即非对称多处理)，或者使这些CPU并行运行，执行同样的任务(即对称多处理，简写为SMP)。进行非对称多处理非常需要对计算机将要执行的任务有特别的了解，而在像Linux这样的操作系统中，这是不可能的。另一方面，对称多处理相对要容易实现一些。这里所说的“相对容易”就是相对的意思——并不是指它真的容易实现。在对称多处理环境中，CPU共享同一个内存，这样做的后果是一个CPU中运行的代码可能会影响另一个CPU所使用的内存。用户不再可以肯定自己在前面一行中设置了值的那个变量仍然保持着那个值——另一个CPU可能在用户不注意的时候对那个变量进行了处理。很显然，要编出这样的程序是不可能的。

在进程编程时，一般来说上面这个问题就不是什么问题了，因为进程在某个时刻一般是只在一个CPU上运行的。而另一方面，内核则可以被运行在不同CPU上的不同进程所调用。

在版本2.0.x中，这个也不是什么问题，因为整个内核就是一个大的自旋锁(spinlock)。这意味着如果某个CPU在内核中而另一个CPU试图进入内核(假设是由于系统调用)，则后到的那个CPU必须等待，直到第一个CPU处理完，这使得Linux SMP比较安全，但是效率却相当低。

在版本2.2.x中，几个CPU可以同时位于内核中，这是模块编程人员需要留意的地方。我已经就SMP的问题向其它高手求助了，希望在本书的下一个版本中将会包含更多的信息。

第13章　常 见 错 误

在读者踌躇满志地准备动手编写内核模块以前，我还要提醒大家注意一些事情。如果是因为我没有警告过你而发生了不愉快的情况，请把你遇到的问题告诉我。我将把你买此书所付的钱全部还给你。

1) 使用标准库　不能这样做，在内核模块中用户只能使用内核函数，也就是可以在/proc/ksyms中找到的那些函数。

2) 关闭中断　用户可能需要在短时间内暂时关闭中断，那没什么问题，但是事后如果忘了打开它们，系统将会瘫痪，用户将不得不把电源拔掉。

3) 无视危险的存在　可能不需要提醒读者，但是最后我还是要说，只是为了以防万一。

附录A　2.0和2.2之间的差异

实际上我对整个内核了解得并不是很透彻，没有透彻到能够列出所有变化的地步。在转换本书例子的过程中(或者更确切地说是对Emmanuel Papirakis所做的转换进行修改)，我遇到了下面的差异，我把它们全部列举了出来，以便帮助模块编程人员 (特别是那些曾经学习过本书的前一版本，并熟悉我所使用的技术的读者)转换到新的版本。

希望进行转换工作的用户还可以访问如下网址：

http://www.atnf.csiro.au/~rgooch/linux/docs/porting_to_2.2.html.

1) asm/uaccess.h　如果需要使用put_user或者get_user，用户必须用#include包含这个文件。

2) get_user　在版本2.2中，get_user既可以接收指向用户内存的指针，也可以接收内核内存中的变量，以便填充用户信息。之所以这样，是因为在版本 2.2中get_user可以一次读两个或四个字节，如果我们所读的变量是两个字节或四个字节长的话， get_user可以读入它。

3) file_operations　该结构已经在open函数和close函数之间加入了一个刷新函数。

4) file_operations中的close函数　在版本2.2中，close函数返回一个整数，所以它可以失败。

5) file_operations中的read和write函数　这两个函数的头部已经改变了。在版本 2.2中，它们不再返回整数，而是返回一个 ssize_t类型的值，它们的参数列表也不同了。索引节点不再作为一个参数，但同时却加入了文件中的偏移量作为参数。

6) proc_register_dynamic　该函数已经不再存在了。取而代之的是，用户可以调用proc_register并把结构的索引结点字段置为0。

7) 信号　在任务结构中的信号不再是32位长的整数值，而是变成了由 _NSIG_WORDS整数组成的一个数组。

8) queue_task_irq　即使用户希望从中断处理程序内部调度任务来执行，他也应该使用queue_task，而不要使用queue_task_irq。

9) 模块参数　用户不再是只把模块参数定义为全局变量了。在 2.2中，用户必须同时使用MODULE_PARM来定义它们的类型。这是一个很大的改进，因为它允许模块可以接收以数字开头的字符串参数，而不会搞混淆。

10) 对称多处理　内核不再是一个大的自旋锁了，这就意味着内核模块在使用 SMP时必须小心。

附录B 其他资源

如果笔者愿意的话，可以轻易地在本书中再加入几章。可以加入一章介绍如何创建新的文件系统，或者介绍如何增加新的协议栈 (但这是不必要的 —— 因为读者很难找到Linux所不支持的协议栈)。当然还可以加入一些内核机制的解释，而这些机制我们并没有接触过，例如引导机制或者磁盘接口。

然而，笔者没有这么做，因为笔者写这本书的目的是为了探索内核模块编程的奥秘，希望教给用户一些内核模块编程的通用技术。对于那些对内核编程有着强烈兴趣的读者，笔者建议他们阅读如下网址的内核资源列表： http://jungla.dit.upm.es/~jmseyas /linux/kernel /hackers-docs.html。正如Linus所说的那样，学习内核的最佳方法就是自己阅读代码。

如果读者希望得到更多的短内核模块的例子，我建议阅读 Phrack杂志，即使用户对安全性不是太感兴趣，作为一个程序员也应该培养这方面的兴趣。 Phrack杂志上的内核模块都是一些很好的例子，介绍了用户可以在内核中所做的有关工作。而且这些例子都很短，读者不用费太大的劲就可以弄懂它们。

希望我能帮助读者成为一个更好的程序员，或者至少培养了在这方面的兴趣。如果读者写出了有用的内核模块，希望也能按照 GPL把它们出版出来，这样我也可以使用它们。

附录C　给出你的评价

这是一本“自由”的书。在GNU公共许可证所规定的内容以外，读者不受任何限制。如果读者想为这本书做点什么，我有以下几点建议：

- 给我寄一张明信片，地址是：

  Ori Pomerantz

  Apt. #1032

  2355 N Hwy 360

  Grand Prairie, TX 75050

  USA

  如果希望收到我的致谢回函，请在明信片上写清你的电子邮箱地址。
- 给自由软件组织捐献一些钱，或者一些时间 (更佳)。编写一个程序或者写作一本书，并按照GPL的条款出版，教别人怎样使用自由软件，例如 Linux或者Perl。
- 向别人解释一下自私与社会生活是格格不入的，与帮助其它人是格格不入的。我很高兴我写了这本书，并且我相信出版这本书将来一定会获得回报。同时，我写这本书是为了帮助读者(如果我做到了的话)。记住，快乐的人常常比不快乐的人对自己更有用，而有才华的人常常比低能儿要有帮助的多。
- 高兴起来。如果我将遇见你，而且你是个愉快的人，这次会面将会给我留下美好的印象，而且愉快的个性也会使你变得对我更有用。

第1章　Linux操作系统

1991年3月，Linus Benedict Torvalds为他的AT386计算机买了一个多任务操作系统：Minix。他使用这个操作系统来开发自己的多任务系统，并称之为 Linux。1991年9月，他向Internet网上的其他一些Minix用户发电子邮件，发布了第一个系统原型，这样就揭开了 Linux工程的序幕。从那时起，有许多程序员都开始支持 Linux。他们增加设备驱动程序，开发应用程序，他们的目标是符合 POSIX标准。现在的Linux功能已经非常强大了，但是 Linux更吸引人的地方在于，它是免费的 (当然并不像免费啤酒那样，不是完全免费)。现在人们正在把Linux移植到其他平台上。

第2章　Linux内核

Linux的基础就是它的内核。用户可以替换某个库，或者将所有库都进行替换，但是只要Linux内核存在，它就还是Linux。内核包括设备驱动程序、内核管理、进程管理以及通信管理。内核高手总是遵循POSIX规则，该规则有时会使编程变得简单，有时会使它变得比较复杂。如果用户的程序在一个新的Linux内核版本上行为发生了变化，可能是因为实现了一个新的POSIX规则。如果读者想了解更多的关于Linux内核编程的信息，可以阅读《Linux Kernel Hacker ’s Guide》。

第3章　Linux libc包

Libc：ISO 8859.1；位于<linux/param.h>中；包括YP函数、加密函数、一些基本的影子过程(默认情况不包含)，……在libcompt中有一些为了保持兼容性而提供的老过程 (默认情况下不激活)；提供英文、法文，或者德文的错误信息；在 libcurses中有一些具有bsd 4.4lite兼容性的屏幕处理过程；在libbsd中有一些bsd兼容的过程；在libtermcap中有一些屏幕处理过程；在libdbm中有用于数据库管理的过程；在 libm中有数学过程；在crtO.o???中有执行程序的入口，在libieee???有一些字节信息(请别再笑话我了，能不能给我提供一些信息？)，在libgmon中是用户空间的配置信息。

我希望由某位Linux libc开发人员来编写本章。现在我能说的唯一的一句话是 a.out可执行格式将会变化成elf(可执行并可链接)格式(出版者注：这个变化已经发生了)，而后者又意味着在创建共享库方面的一个变化。当前这两种格式 (a.out和elf)都支持。

Linux libc包的绝大部分都是遵守库 GNU公共许可证的，尽管有些文件是遵守特殊的版权规定的，例如 crtO.o。对商业版本来说，这就意味着一个限制 ,即禁止静态链接可执行程序。在这里动态链接可执行程序又是一个特殊的例外。 FSF(自由软件基金会)的Richard Stallman说过：

在我看来，我们应该明确地允许发行不带伴随库的动态链接可执行程序，只要组成该可执行程序的对象文件按照第 5节的规定是不受限制的。… …，所以我决定现在就允许这样做。实际上，要更新LGPL将需要等到我有时间的时候，并且需要检查一下新版本。

第4章 系统调用

系统调用是向操作系统(内核)所作出的一次申请，请求操作系统做一次硬件/系统相关的操作，或者是作一次只有系统才能做的操作。以Linux 1.2为例，它总共定义了140个系统调用。有些系统调用(如close ( ))是在Linux libc中实现的。这种实现常常需要调用一个宏，而该宏最后会调用syscall ( )。传送给syscall ( )的参数是系统调用的编号，在编号后面的参数是其他一些必需的变元。如果通过实现一个新的libc库而更新了<sys/syscall.h>，则真正的系统调用编号可以在<linux/unistd.h>中找到。如果新的系统调用在libc中还没有代理程序，用户可以使用syscall ( )。下面给出一个例子，可以像下面一样使用syscall ( )来关闭一个文件(不提倡)：

```
#include <syscall.h>

extern int syscall(int, ...);

int my_close(int filedescriptor)
{
   return syscall(SYS_close, filedescriptor);
}
```

在i386体系结构下，除了系统调用编号以外，系统调用只能带5个以下的变元，这是由于受到了硬件寄存器数目的限制。如果读者是在另一个体系结构下运行Linux的，可以检查一下<asm/unistd.h>中的_syscall宏，看看硬件支持多少个变元，或者说开发人员选择支持多少个变元。这些_syscall可以用于取代syscall ( )，但是并不提倡用户这么做，这是因为由这些宏扩展而成的完整的函数有可能在库中早已存在了。

所以，只有内核高手才能去使用_syscall宏。为了说明这一点，下面给出一个使用_syscall宏的close ( )的例子：

```
#include <linux/unistd.h>

_syscall1(int, close, int, filedescriptor);
```

在_syscall1宏扩展以后，得到函数close ( )。这样，我们就有两个close ( )了，一个在libc中，另一个在我们的程序中。如果系统调用失败，syscall ( )或者_syscall宏的返回值是－1；而如果系统调用成功，则返回值将是0或者更大的数值，如果系统调用失败，我们可以查看一下全局变量errno，看看到底发生了什么。

下面这些系统调用在BSD和Sys V中是可用的，但Linux 1.2不支持：

audit ( )、audition ( )、auditsvc ( )、fchroot ( )、getauid ( )、getdents ( )、getmsg ( )、mincore ( )、poll ( )、putmsg ( )、setaudit ( )、setauid ( )。

第5章 “瑞士军刀”：ioctl

ioctl代表输入/输出控制，它用于通过文件描述符来操作字符设备。 ioctl的格式如下所示：

ioctl (unsigned int fd, unsigned int request, unsigned long argument)

如果出错则返回值为－ 1，如果请求成功则返回值将大于或者等于 0，这就像其他系统调用一样。内核能区分特殊文件和普通文件。特殊文件一般可以在 /dev和/proc中找到。它们与普通文件的区别在于，它们隐藏了驱动程序的接口，并不是一个包含着文本或二进制数据的真正的(常规的)文件。这是Unix的特点，它允许用户对每一个文件都可以使用普通的读 /写操作。但是，如果用户想要对特殊文件或者普通文件进行更多的处理，用户可以使用……对了，就是ioctl。用户把ioctl用于特殊文件的机会比用于普通文件的机会要多得多，但是在普通文件上也可以使用ioctl。

第6章　Linux进程间通信

下面我们详细地介绍在Linux操作系统中实现的IPC(进程间通信)机制。

6.1　介绍

Linux IPC(进程间通信)机制为多个进程之间相互通信提供了一种方法。对 Linux C程序员来说，有许多种IPC的方法：

- 半双工Unix管道
- FIFO(命名管道)
- SYSV形式的消息队列
- SYSV形式的信号量集合
- SYSV形式的共享内存段
- 网络套接字(Berkeley形式)(本书不作介绍)
- 全双工管道(STREAMS管道)(本书不作介绍)

如果使用得当的话，这些机制将为任意 Unix系统(包括Linux)上的客户机/服务器开发提供一个坚强的框架。

6.2　半双工Unix管道

6.2.1　基本概念

简单地说，管道就是一种把一个进程的标准输出与另一个进程的标准输入相连接的方法。管道是最古老的IPC工具，自从Unix操作系统诞生以来，管道就已经存在了。它们提供了一种进程之间单向通信的方法(这就是术语“半双工”的由来)。

管道这一特征已经被广泛地使用了，甚至在 Unix命令行中(在Shell)中也用到了管道。

图3-6-1显示了建立一个管道，把 ls的输出当作sort的输入，以及把sort的输出当作lp输入进行处理的一系列过程。数据是通过一个半双工管道传输的，从管道的左边传输到管道的右边。

尽管我们中的大多数人都曾经非常频繁地在 shell脚本编程时使用管道，但在我们这么做的时候常常懒得想一下在内核一级到底发生了一些什么事。

在进程创建管道时，内核创建两个文件描述符，以供管道使用。其中一个描述符用于允许输入管道的路径(写)；而另一个用于从管道获得数据 (读)。如果仅仅是这样，管道将没有什么实用性，创建管道的进程只能用该管道与它自己通信。下图显示了在创建管道以后进程和内核的状态。

注意　出版者注：作者尚未提供3-6-1图。

从上图可以很容易地看出这两个文件描述符是如何连接在一起的。如果进程通过管道

(fd0)发送数据，它可以从fd1获得(读)那个信息。然而，相对上面这个简单的草图来说，用户希望实现的目标要大得多。当一个管道把进程与它自己相连接时，在管道中传输的数据需要经过内核。特别是在Linux下，管道实际上是用一个合法索引节点在内部表示的。当然，这个索引节点自己也驻留在内核中，并且不在任何物理文件系统的范围之内，这个特点将为用户提供一些方便的I/O方法，稍后读者就能了解到这一点。

到目前为止，管道显得毫无用处。然而，如果我们只想和自己通信，我们为什么要不辞辛苦地去创建管道呢？就目前来说，创建进程一般都是产生一个子进程。因为子进程将从父母那里继承所有打开的文件描述符，我们现在已经得到了一个多进程通信的框架(在父与子之间通信)。我们上面这个简单的草图经过升级以后如图3-6-2所示。

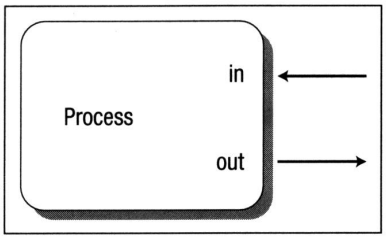


图 3-6-2

现在读者可以看到，两个进程都能访问组成管道的那两个文件描述符。就在这个时候需要作出一个关键的决定，即希望数据向哪个方向传输？是子进程向父进程发送信息，还是反过来？一旦决定以后，两个进程彼此都同意这个决定，并把它们各自所不关心的管道的某一端关掉。为了方便进行介绍，这里假设子进程进行某些处理操作，并通过管道把消息发送回父进程。经过修改以后，图3-6-2变成了图3-6-3。

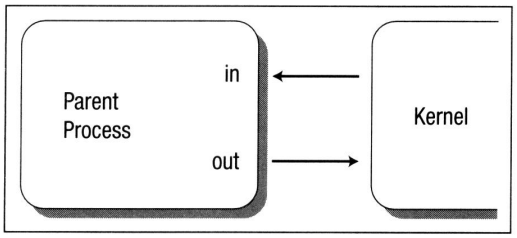


图 3-6-3

管道的创建工作到此全部完成！唯一剩下要做的事就是使用管道了。为了直接访问管道，可以使用那些用来完成低级文件I/O的系统调用(记住，管道实际上在内部是表示成合法的索引节点的)。

为了把数据发送到管道，我们使用 write ( )系统调用；而为了从管道接收数据，我们可以使用 read ( )系统调用。记住，低级文件I/O系统调用是使用文件描述符的！然而，读者需要注意的是，有些特定的系统调用，例如 lseek ( ), 对管道的文件描述符是无效的。

6.2.2 用C语言创建管道

与简单的shell例子比较起来，用高级编程语言 C创建管道要常见一些。为了用 C语言创建一个简单的管道，用户可以使用 pipe ( )系统调用。这个系统调用只需要一个变元，它是由两个整数组成的一个数组。如果调用成功，该数组将会包含管道所使用的两个新的文件描述符。在创建管道以后，进程一般会产生一个新进程 (记住子进程将继承打开的文件描述符)。

```
SYSTEM CALL: pipe();

  PROTOTYPE: int pipe( int fd[2] );
    RETURNS: 0 on success
             -1 on error: errno = EMFILE (no free descriptors)
                                  EMFILE (system file table is full)
                                  EFAULT (fd array is not valid)
```

```
NOTES: fd[0] is set up for reading, fd[1] is set up for writing
```

数组中的第一个函数(元素0)是为了读操作而创建和打开的，而第二个函数 (元素1)是为了写操作而创建和打开的。直观地说， fd1的输出变成了fd0的输入。再说一遍，通过管道传输的所有数据都将经过内核。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

main()
{
        int     fd[2];

        pipe(fd);
        .
        .
}
```

记住，在C语言中，数组的名称实际上变成了指向它的第一个成员的指针，在上面的程序中，fd相当于&fd[0]。一旦创建了管道，就可以产生子进程了：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
        .
        .

}
```

如果父进程希望从子进程接收到数据，它应该关闭 fd1，而子进程应该关闭fd0。如果父进程希望把数据发送到子进程，它应该关闭 fd0，而子进程应该关闭fd1。因为描述符是在父进程和子进程之间共享的，所以一定要确定我们所关闭的管道的那一端是我们所不关心的。从技术的角度来看，如果管道中不需要的那一端没有显式地关闭的话，将永远不会返回 EOF。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
```

```
main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);
        }
        .
        .
}
```

正如以前所提到的那样，管道一旦创建好以后，它的文件描述符就可以像普通文件的文件描述符一样处理了。

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: pipe.c
 *****************************************************************************/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
```

```
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, strlen(string));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

常常会有这样的情况，用户把子进程的文件描述符复制到标准输入或输出。这样，子进程可以用exec ( )来执行另一个程序，而那个程序将得到标准的输入或输出流。请看下面的 dup ( )系统调用：

```
SYSTEM CALL: dup();
  PROTOTYPE: int dup( int oldfd );
    RETURNS: new descriptor on success
             -1 on error: errno = EBADF (oldfd is not a valid descriptor)
                                  EBADF (newfd is out of range)
                                  EMFILE (too many descriptors for the process)
Notes: The old descriptor is not closed! Both may be used interchangeably
```

尽管老的文件描述符和新创建的文件描述符可以互换使用，但一般首先关闭一个标准的输入/输出流。系统调用dup ( )使用编号最低且未被使用的描述符作为新的描述符。

Consider:

```
        .
        .
        childpid = fork();

        if(childpid == 0)
        {
                /* Close up standard input of the child */
                close(0);

                /* Duplicate the input side of pipe to stdin */
```

```
        dup(fd[0]);
        execlp("sort", "sort", NULL);
        .
}
```

尽管文件描述符0(stdin)被关闭了，对dup ( )的调用会把管道的输入描述符 (fd0)复制到它的标准输入上，然后再调用 execlp ( )，以便让排序程序的文本段 (代码)覆盖子进程的文本段 (代码)。因为新执行的程序从它们的创建者那里继承了标准的输入 /输出流，它实际上就继承了管道的输入端作为自己的标准输入！这样一来，原来的父进程发送到管道的任何数据都将传送到排序工具那里。

用户还可以使用另一个系统调用 dup2 ( )。这个特殊的系统调用最早是由第 7版的Unix提供的，在BSD的发行版本中一直沿续了下来，现在已被 POSIX标准列入规范要求。

```
SYSTEM CALL: dup2();

PROTOTYPE: int dup2( int oldfd, int newfd );
  RETURNS: new descriptor on success
           -1 on error: errno = EBADF (oldfd is not a valid descriptor)
                                EBADF (newfd is out of range)
                                EMFILE (too many descriptors for the process)
```

注意 旧的文件描述符是用dup2 ( )关闭的！

使用这个特殊的调用，用户可以在一次系统调用中完成关闭操作以及文件描述符的复制工作。另外，该系统调用保证是原子性的，这就意味着它永远不会被一个突如其来的信号所中断。在把控制权还给内核进行信号调度以前，整个操作将会全部完成。如果用户使用原来的dup ( )系统调用，则程序员在调用它之前需要执行一次 close ( )操作。这就需要进行两次系统调用，在两次系统调用之间的短暂时间里，存在一点点危险的可能性。如果在那个稍纵即逝的瞬间到达了一个信号，则文件描述符的复制将会失败。当然， dup2 ( )会为用户解决这个问题。

Consider:

```
        .
        .
        childpid = fork();

        if(childpid == 0)
        {
                /* Close stdin, duplicate the input side of pipe to stdin */
                dup2(0, fd[0]);
                execlp("sort", "sort", NULL);
                .
                .
        }
```

6.2.3 简便方法

如果读者认为以上这种方法是一种创建和利用管道的间接方法，觉得它不够直观和简便

的话，可以选择使用下面这种方法：

```
LIBRARY FUNCTION: popen();

PROTOTYPE: FILE *popen ( char *command, char *type);
  RETURNS: new file stream on success
           NULL on unsuccessful fork() or pipe() call
Notes: creates a pipe, and performs fork/exec operations using "command."
```

以上这个标准库函数通过在内部调用 pipe ( )创建了一个半双工管道，然后它生成一个子进程，执行Bourne shell，并在shell中执行“ command ”变元。数据流动的方向由第二个变元“ type ”所决定。它的值可以是“ r ”或者“ w ”。“ r ”代表“ 读 ”，而“ w ”代表“ 写 ”，但不能既为读又为写！在Linux下，管道打开的方式是由“ type ”变元的第一个字符决定的。这样一来，如果用户把“ type ”的值置为“ rw ”，则管道只能以“ 读 ”方式打开。

可能读者会认为这个库函数非常实用，但是读者要知道，这么做所付出的代价也不小。在使用pipe ( )系统调用并自己处理fork/exec操作时，用户对管道有着很好的控制能力，但是使用这个库函数使用户失去了控制权。然而，由于这里直接使用 Bourne shell，所以允许在“ command ”变元中使用shell元字符扩展(包括通配符)。

使用popen ( )创建的管道必须用 pclose ( ) 关闭。到现在为止，读者可能已经意识到popen/pclose与标准文件流I/O函数fopen ( )和fclose ( )有着惊人的相似性。

```
LIBRARY FUNCTION: pclose();

PROTOTYPE: int pclose( FILE *stream );
  RETURNS: exit status of wait4() call
           -1 if "stream" is not valid, or if wait4() fails
Notes: waits on the pipe process to terminate, then closes the stream.
```

在由popen ( )所生成的进程上， pclose ( )函数执行一次wait4 ( )命令。当它返回时，它将摧毁管道和文件流。在这里读者可以再次体会到， pclose ( )和普通的基于流的文件I/O fclose ( )函数非常相似。

请看下面的例子。该例为排序命令打开了一个管道，接着开始处理一个字符串数组的排序：

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: popen1.c
 *****************************************************************************/

#include <stdio.h>

#define MAXSTRS 5

int main(void)
{
        int  cntr;
        FILE *pipe_fp;
        char *strings[MAXSTRS] = { "echo", "bravo", "alpha",
                                   "charlie", "delta"};
```

```
/* Create one way pipe line with call to popen() */
if (( pipe_fp = popen("sort", "w")) == NULL)
{
        perror("popen");
                exit(1);
        }

        /* Processing loop */
        for(cntr=0; cntr<MAXSTRS; cntr++) {
                fputs(strings[cntr], pipe_fp);
                fputc('\n', pipe_fp);
        }

        /* Close the pipe */
        pclose(pipe_fp);

        return(0);
}
```

因为popen ( )使用了shell来完成它的任务，所以可以使用所有的 shell扩展字符和元字符！此外，popen ( )还可以利用一些更高级的技术，例如重定向，甚至还可以利用输出管道技术。请看下面的实例调用：

```
popen("ls ~scottb", "r");
popen("sort > /tmp/foo", "w");
popen("sort | uniq | more", "w");
```

下面的这个小程序是popen ( )的另一个例子，它打开了两个管道 (一个连接到ls命令，另一个连接到sort命令)：

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: popen2.c
 *************************************************************************/

#include <stdio.h>

int main(void)
{
        FILE *pipein_fp, *pipeout_fp;
        char readbuf[80];

        /* Create one way pipe line with call to popen() */
        if (( pipein_fp = popen("ls", "r")) == NULL)
        {
                perror("popen");
                exit(1);
        }
```

```
    /* Create one way pipe line with call to popen() */
    if (( pipeout_fp = popen("sort", "w")) == NULL)
    {
            perror("popen");
            exit(1);
    }

    /* Processing loop */
    while(fgets(readbuf, 80, pipein_fp))
            fputs(readbuf, pipeout_fp);

    /* Close the pipes */
    pclose(pipein_fp);
    pclose(pipeout_fp);

    return(0);
}
```

为了对popen ( )进行最后的说明。我们创建了一个普通的程序。该程序在执行的命令和文件名称之间打开一个管道：

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: popen3.c
 *************************************************************************/

#include <stdio.h>

int main(int argc, char *argv[])
{
        FILE *pipe_fp, *infile;
        char readbuf[80];

        if( argc != 3) {
                fprintf(stderr, "USAGE:  popen3 [command] [filename]\n");
                exit(1);
        }

        /* Open up input file */
        if (( infile = fopen(argv[2], "rt")) == NULL)
        {
                perror("fopen");
                exit(1);
        }
/* Create one way pipe line with call to popen() */
if (( pipe_fp = popen(argv[1], "w")) == NULL)
{
        perror("popen");
```

```
        exit(1);
    }

    /* Processing loop */
    do {
        fgets(readbuf, 80, infile);
        if(feof(infile)) break;

        fputs(readbuf, pipe_fp);
    } while(!feof(infile));

    fclose(infile);
    pclose(pipe_fp);

    return(0);
}
```

使用以下的命令试着运行这个程序：

```
popen3 sort popen3.c
popen3 cat popen3.c
popen3 more popen3.c
popen3 cat popen3.c | grep main
```

6.2.4 管道的原子操作

如果说一个操作是“原子性”的，那么就不能以任何理由来中断该操作。整个操作都是一次性完成的。在/usr/include/posix1_lim.h中，POSIX标准对管道上的原子操作的最大缓冲区大小是这样规定的：

```
#define _POSIX_PIPE_BUF         512
```

在一次原子操作中，最多可以把512个字节写到管道，或者最多从管道读取512个字节。如果超过了这个范围，操作就将被分开，不能称之为原子操作了。然而，在Linux下，原子操作的限制在“linux/limits.h”中是这样定义的：

```
#define PIPE_BUF        4096
```

读者不难看出，Linux调整了POSIX规定的很少数目的字节数，甚至可以说是大大地调整了这个数目。当包含多个进程时，管道的原子操作变得非常重要。例如，如果写到管道的字节数目超过了一个操作所能达到的原子性限制，并且有多个进程都在写管道，则数据将会被“交叉”或者“分块”。换句话说，一个进程可能会在另一个进程进行写的时候把数据插入到管道中。

6.2.5 关于半双工管道需要注意的几个问题

- 通过创建两个管道，并在子进程中正确地重新分配好文件描述符，用户就可以创建双向管道。
- pipe ( )调用必须在调用fork ( )以前进行，否则子进程将无法继承文件描述符 (对popen ( )来说也是如此)。

• 使用半双工管道互相连接的任意进程必须位于一个相关的进程家族里。因为管道必须受到内核的限制，所以如果进程没有在管道创建者的家族里面，则该进程将无法访问管道。这一点是与命名管道(FIFO)有区别的。

6.3 命名管道

6.3.1 基本概念

命名管道的工作方式与普通的管道非常相似，但也有一些明显的区别。

• 在文件系统中命名管道是以设备特殊文件的形式存在的。
• 不同家族的进程可以通过命名管道共享数据。
• 因为所有的I/O工作都是由共享进程处理的，文件系统中保留命名管道是为了将来使用。

6.3.2 创建FIFO

有许多种方法可以创建命名管道。其中前两者可以直接用 shell来完成。

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

上面这两个命令执行同样的操作，只有一个地方存在差异。 mkinfo命令提供了一个挂钩，可以在创建 FIFO文件之后直接改变它的许可。而如果用户使用 mknod，则需要立刻调用chmod命令。

在物理文件系统中，用户可以很快找到 FIFO文件，因为在长目录列表中， FIFO文件有一个“ p ”指示符。

```
$ ls -l MYFIFO
prw-r--r--   1 root     root            0 Dec 14 22:15 MYFIFO|
```

同时读者还可以注意到，在文件名的后面有一条竖线 (“ 管道符号 ”)。由此不难体会到运行Linux无所不在的好处。

为了用C语言创建FIFO，用户可以使用mknod ( )系统调用：

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

the file “/tmp/MYFIFO” is created as a FIFO file. The request
ough they are affected by the umask setting as follows:

```
final_umask = requested_permissions & ~original_umask
```

rick is to use the umask() system call to temporarily zap the

```
umask(0);
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

关于mknod ( )更详细的介绍，用户可以参考 man的帮助信息，现在请看用C语言创建FIFO的一个简单的例子：

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

在上面这个例子中，文件 /tmp/MYFIFO是当作一个FIFO文件而创建的。所申请的访问许

可权限是“ 0666 ”，尽管该权限会受到如下定义的掩码的影响：

```
final_umask = requested_permissions & ~original_umask
```

一个常见的技巧是使用umask ( )系统调用暂时屏蔽掉掩码的值：

```
umask(0);
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

此外， mknod ( )的第三个参数可以忽略，除非用户创建的是设备文件，在那种情况下，mknod ( )应该指明设备文件的主编号和从编号。

6.3.3 FIFO操作

对FIFO来说， I/O操作与普通的管道I/O操作基本上是一样的，但也存在着一个主要的区别。在FIFO中，必须使用一个“ open ”系统调用或者库函数来物理地建立联接到管道的通道。而对于半双工管道而言，这是不必要的，因为管道是驻留在内核中的，而不是驻留在物理文件系统上的。在下面的例子中，我们将把管道当作一个流来看待，使用 fopen ( )来打开它，并使用fclose ( )来关闭它。

请看下面这个简单的服务器进程：

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: fifoserver.c
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#include <linux/stat.h>

#define FIFO_FILE       "MYFIFO"

int main(void)
{
        FILE *fp;
        char readbuf[80];

        /* Create the FIFO if it does not exist */
        umask(0);
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1)
        {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);
                fclose(fp);
```

```
        }

        return(0);
}
```

因为FIFO是默认阻塞的，所以在编译了这个服务器进程之后，可以在后台运行它：

```
$ fifoserver&
```

我们稍后就将讨论FIFO的阻塞动作。首先，请看上面这个服务器进程的简单的客户前端，如下所示：

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: fifoclient.c
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define FIFO_FILE       "MYFIFO"

int main(int argc, char *argv[])
{
        FILE *fp;

        if ( argc != 2 ) {
                printf("USAGE: fifoclient [string]\n");
                exit(1);
        }

        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }

        fputs(argv[1], fp);

        fclose(fp);
        return(0);
}
```

6.3.4 FIFO上的阻塞动作

一般来说FIFO总是处于阻塞状态。换句话说，如果 FIFO是为了读操作而打开的，则进程将“阻塞”，直到某些其他进程打开该FIFO并且写数据。这个阻塞动作反过来也是成立的。如果用户不希望发生阻塞，可以在 open ( )调用中使用O_NONBLOCK标志，以关闭默认的阻塞动作。

在上面那个简单的服务器进程的例子中，我们只是把它放在后台，让它在那里进行它的

阻塞动作。用户还可以选择跳转到另一个虚拟控制台，并运行客户前端，来回切换以查看所导致的动作。

6.3.5 SIGPIPE信号

最后需要注意的一点是，管道必须有人读并有人写。如果某个进程企图往管道中写数据，而没有进程去读该管道，则内核将向该进程发送 SIGPIPE信号。当在管道操作中涉及到两个以上的进程时，这个信号是非常必要的。

6.4 系统V IPC

6.4.1 基本概念

在Unix系统V中，AT&T引入了三种新的IPC方法(消息队列、信号量和共享内存)。POSIX委员会还没有完全制定好这些工具的标准，但大多数实现已经支持这三种 IPC方法。此外，Berkeley (BSD)使用套接字来作为其IPC的主要方法，而不是采用系统V的方法。Linux可以同时使用这两种IPC方法(BSD和系统V)。在稍后的某章中将会介绍套接字。

在Linux中，系统V IPC的实现是由Krishna Balasubramanian完成的，他的电子邮箱地址是balasub@cis.ohio_state.edu。

1. IPC标识符

每一个IPC对象都有一个独一无二的IPC标识符与之联系。这里所说的“ IPC对象 ”是指单个的消息队列、信号量集合或者共享内存段。在内核中这个标识符可以用于唯一地标识一个IPC对象。例如，为了访问某个特定的共享内存段，用户唯一需要的是赋给那个内存段的独一无二的ID值。

标识符的唯一性是与对象的类型相关的。为了说明这一点，假设有一个数字标识符“ 12345 ”。当然不可能会有两个消息队列都对应到这个标识符。但还是存在一种明显的可能性，即某个消息队列和某个共享内存段都具有这个数字标识符。

2. IPC关键字

为了获得一个唯一的ID号。必须使用关键字，客户进程和服务器进程双方都必须同意这个关键字，这是构造应用程序的客户/服务器框架的第一步。

当你给某人打电话时，你必须知道他的电话号码。此外，电话公司必须知道如何把你发出的呼叫转达到它最终的目的地。一旦对方提起电话，开始作出响应，联接就建立起来了。

在系统V IPC方法中，“ 电话 ”直接对应到使用的对象的类型，而“ 电话公司 ”，或者说寻找路由的方法，则与IPC关键字有着直接的联系。

关键字对不同的对象可以具有相同的值，它是通过把关键字的值硬编码进应用程序来实现的。这样做有一个缺点，就是关键字可能早已经在使用了。用户通常可以使用 ftok ( )函数来为客户进程和服务器进程生成关键字的值。

```
LIBRARY FUNCTION: ftok();
PROTOTYPE: key_t ftok ( char *pathname, char proj );
  RETURNS: new IPC key value if successful
           -1 if unsuccessful, errno set to return of stat() call
```

ftok ( )返回的关键字值是这样生成的：把索引节点号、第一个变元中文件的次设备编号，以及第二个变元中的工程标识符 (一个字符)组合起来，这就是返回值。这样的返回值不能保证是唯一的，但应用程序可以查找到是否存在冲突，如果有的话则重新试着生成关键字。

```
key_t   mykey;
mykey = ftok("/tmp/myapp", 'a');
```

在上面这个短的代码片段中，目录 /tmp/myapp与 ’ a ’ 这个标识符组合在一起。另一个通用的例子是使用当前目录：

```
key_t   mykey;
mykey = ftok(".", 'a');
```

采取什么样的关键字生成算法完全是由应用程序编程人员决定的。只要可以防止竞争条件、死锁等等，无论采取什么方法都可以。为了方便进行介绍，这里将使用 ftok ( )方法。如果用户能确定每个客户进程都是从独一无二的“ home ”目录运行的，则所产生的关键字应该能够满足需要。

无论关键字的值是如何产生的，它将用于后续的 IPC系统调用中，用来创建 IPC对象或者获取对它的访问权限。

3. ipcs命令

ipcs命令可以用来获取所有系统 V IPC对象的状态。这个工具的 Linux版本也是由 Krishna Balasubramanian编写的。

```
ipcs      -q:    Show only message queues
ipcs      -s:    Show only semaphores
ipcs      -m:    Show only shared memory
ipcs --help:     Additional arguments
```

在默认的情况下，所有三种类型的 IPC对象都将显示出来。请看下面这个 ipcs命令的输出示例：

```
------ Shared Memory Segments --------
shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
semid      owner      perms      nsems      status
 ------ Message Queues --------
 msqid      owner      perms      used-bytes  messages
 0          root       660        5           1
```

从上面这个输出中读者可以找到一个具有标识符“ 0 ”的消息队列，它是由 root用户所拥有的，具有八进制形式的访问权限 660，即-rw-rw--。在此队列中有一条消息，而那个消息它的大小总共为5个字节。

ipcs命令是一个功能非常强大的工具，它使用户可以研究内核中 IPC对象的存储机制。读者应该学好它，用好它。

4. ipcrm命令

ipcrm命令可以用来从内核删除一个 IPC对象。IPC对象可以通过在用户代码中调用系统调用来删除(稍后将介绍这方面的内容)。然而，用户常常会需要手工删除 IPC对象，尤其是在开发环境下，ipcrm命令的用法非常简单：

```
ipcrm <msg | sem | shm>  <IPC ID>
```

只需指定想要删除的对象是消息队列(msg)、信号量集合(sem)还是共享内存段(shm)即可。IPC ID号可以使用ipcs命令来获得。用户必须指明对象的类型，因为在相同类型的对象之中，标识符是各不相同的(参见前面的讨论)。

6.4.2 消息队列

1. 基本概念

消息队列的最佳定义是：内核地址空间中的内部链表。消息可以顺序地发送到队列中，并以几种不同的方式从队列中获取。当然，每个消息队列都是由 IPC标识符所唯一标识的。

2. 内部和用户数据结构

要完成理解象系统 V IPC这样复杂的问题，关键是要彻底熟悉内核的几个内部数据结构。甚至对那些最基本的操作来说，直接访问这些结构中的某几个结构也是必要的，而其他的结构则停留在一个更低的级别上。

3. 消息缓冲区

我们要介绍的第一个结构是 msgbuf结构。这个特殊的数据结构可以认为是消息数据的模板。虽然定义这种类型的数据结构是程序员的职责，但是读者绝对必须知道实际上存在msgbuf类型的结构。在linux/msg.h中，它的定义如下所示：

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
    long mtype;         /* type of message */
    char mtext[1];      /* message text */
};
```

在msgbuf结构中有两个成员：

• mtype —— 它是消息类型，以正数来表示。这个数必须为一个正数！

• mtext —— 它就是消息数据。

因为用户可以给特定的消息赋予一个类型，这样用户就能够在一个消息队列上进行消息的多路传送。例如，用户必须赋予客户进程一个幻数，这个幻数可以用作服务器进程所发送的消息的消息类型。而服务器本身也可以使用其他的一些数，客户可以使用这些数向它发送消息。在另一种情况下，应用程序可以把错误消息标记为具有消息类型号 1的消息，而请求消息的类型为2，等等。这种可能性是不胜枚举的。

读者需要注意的另一点是，不要被被消息数据元素 (mtext)的描述性太强的名称所误导，这个域并不是只能存放字符数据，它还能存放任意形式的任意数据。因为应用程序编程人员可以重新定义msgbuf结构，所以mtext域实际上是随机性很强的。请看下面的例子：

```
struct my_msgbuf {
        long    mtype;          /* Message type */
        long    request_id;     /* Request identifier */
        struct  client info;    /* Client information structure */
};
```

在这个定义中也有消息类型，这点与以前的定义一样，但是结构的剩余部分则被替换成两个其他的元素，其中有一个是另一种结构！这就是消息队列的可爱之处。这样一来，不论是哪种类型的数据，内核均无需作转换工作，任意信息均可以发送。

然而，对于给定消息的最大的大小，确实存在一个内部的限制。在 Linux中，它在

linux/msg.h中是这样定义的：

```
#define MSGMAX  4056   /* <= 4056 */   /* max size of message (bytes) */
```

消息总的大小不能超过4,056个字节，这其中包括mtype成员，它的长度是4个字节(long类型)。

4. 内核msg结构

内核把消息队列中的每个消息都存放在msg结构的框架中。该结构是在linux/msg.h中定义的，如下所示：

```
/* one msg structure for each message */
struct msg {
    struct msg *msg_next;   /* next message on queue */
    long  msg_type;
    char *msg_spot;         /* message text address */
    short msg_ts;           /* message text size */
};
```

- msg_next —— 这是一个指针，指向消息队列中的下一个消息。在内核寻址空间中，它们是当作一个链表存储的。
- msg_type —— 这是消息类型，它的值是在用户结构msgbuf中赋予的。
- msg_spot —— 这是一个指针，指向消息体的开始处。
- msg_ts —— 这是消息文本(消息体)的长度。
- 内核msgid_ds结构 —— IPC对象分为三类，每一类都有一个内部数据结构，该数据结构是由内核维护的。对于消息队列而言，它的内部数据结构是 msqid_ds结构。对于系统上创建的每个消息队列，内核均为其创建、存储和维护该结构的一个实例。该结构在linux/msg.h中定义，如下所示：

```
/* one msqid structure for each queue on the system */
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;  /* first message on queue */
    struct msg *msg_last;   /* last message in queue */
    time_t msg_stime;       /* last msgsnd time */
    time_t msg_rtime;       /* last msgrcv time */
    time_t msg_ctime;       /* last change time */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    ushort msg_cbytes;
    ushort msg_qnum;
    ushort msg_qbytes;      /* max number of bytes on queue */
    ushort msg_lspid;       /* pid of last msgsnd */
    ushort msg_lrpid;       /* last receive pid */
};
```

虽然读者可能很少会关心这个结构的大部分成员，为了叙述的完整性，下面还是对每个成员都给出一个简短的介绍：

- msg_perm —— 它是ipc_perm结构的一个实例， ipc_perm结构是在linux/ipc.h中定义的。该成员存放的是消息队列的许可权限信息，其中包括访问许可信息，以及队列的创建者

的有关信息(如uid等等)。

- msg_first —— 链接到队列中的第一个消息(列表头部)。
- msg_last —— 链接到队列中的最后一个消息(列表尾部)。
- msg_stime —— 发送到队列的最后一个消息的时间戳(time_t)。
- msg_rtime —— 从队列中获取的最后一个消息的时间戳。
- msg_ctime —— 对队列进行最后一次变动的时间戳(稍后将作详细介绍)。
- wwait和rwait —— 是两个指针，指向内核的等待队列。如果消息队列上的某次操作使进程进入睡眠状态，则需要使用这两个成员！(类似的操作包括：队列已满，进程等待一次打开操作)。
- msg_cbytes —— 在队列上所驻留的字节的总数(即所有消息的大小的总和)。
- msg_qnum —— 当前处于队列中的消息数目。
- msg_qbytes —— 队列中能容纳的字节的最大数目。
- msg_lspid —— 发送最后一个消息的进程的PID。
- msg_lrpid —— 接收最后一个消息的进程的PID。

5. 内核ipc_perm结构

内核把IPC对象的许可权限信息存放在ipc_perm类型的结构中。例如在前面描述的某个消息队列的内部结构中，msg_perm成员就是ipc_perm类型的，它的定义是在文件linux/ipc.h中，如下所示：

```
struct ipc_perm
{
  key_t  key;
  ushort uid;   /* owner euid and egid */
  ushort gid;
  ushort cuid;  /* creator euid and egid */
  ushort cgid;
  ushort mode;  /* access modes see mode flags below */
  ushort seq;   /* slot usage sequence number */
};
```

以上所有的成员都具有相当的自扩展性。对象的创建者以及所有者(它们可能会有不同)的有关信息，以及对象的IPC关键字都是存放在该结构中的。八进制形式的访问模式也是存放在这里的，它是以一种无符号短整型的形式存储的。最后，时间片使用序列编号存放在最后面，每次通过系统调用关闭IPC对象(摧毁)时，这个值将被增加一，至多可以增加到能驻留在系统中的IPC对象的最大数目。用户需要关心这个值吗？答案是“不”。

有关这个问题，在Richard Stevens所著的《Unix Network Programming》一书的第125页中作了精辟的讨论。该书还介绍了ipc_perm结构的存在和行为在安全性方面的原因。

6. 系统调用：msgget ( )

为了创建一个新的消息队列，或者访问一个现有的队列，可以使用系统调用 msgget ( )。

```
SYSTEM CALL: msgget();

PROTOTYPE: int msgget ( key_t key, int msgflg );
  RETURNS: message queue identifier on success
           -1 on error: errno = EACCESS (permission denied)
```

```
EEXIST (Queue exists, cannot create)
EIDRM (Queue is marked for deletion)
ENOENT (Queue does not exist)
ENOMEM (Not enough memory to create queue)
ENOSPC (Maximum queue limit exceeded)
```

msgget ( )的第一个变元是关键字的值(在我们的例子中该值是调用 ftok ( )的返回值)。这个关键字的值将被拿来与内核中其他消息队列的现有关键字值相比较。比较之后，打开或者访问操作依赖于msgflg变元的内容。

• IPC_CREAT —— 如果在内核中不存在该队列，则创建它。

• IPC_EXCL —— 当与IPC_CREAT一起使用时，如果队列早已存在则将出错。

如果只使用了IPC_CREAT, msgget ( )或者返回新创建消息队列的消息队列标识符，或者会返回现有的具有同一个关键字值的队列的标识符。如果同时使用了 IPC_EXCL和IPC_CREAT，那么将可能会有两个结果。或者创建一个新的队列，或者如果该队列存在，则调用将出错，并返回－1。IPC_EXCL本身是没有什么用处的，但在与IPC_CREAT组合使用时，它可以用于保证没有一个现存的队列为了访问而被打开。

有个可选的八进制许可模式，它是与掩码进行 OR操作以后得到的。这是因为从功能上讲，每个IPC对象的访问许可权限与Unix文件系统的文件许可权限是相似的！

下面我们创建一个包装程序，它可用于打开或者创建消息队列：

```
int open_queue( key_t keyval )
{
        int     qid;

        if((qid = msgget( keyval, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(qid);
}
```

注意程序中使用了显式的许可权限 0660。这个小函数或者返回消息队列标识符 (int类型)，或者出错返回－1。必须出错返回－1。必须把关键字的值传给它，这是它唯一的变元。

7. 系统调用：msgsnd ( )

一旦获得了队列标识符，用户就可以开始在该消息队列上执行相关操作了。为了向队列传递消息，用户可以使用 msgsnd系统调用：

```
  SYSTEM CALL: msgsnd();

  PROTOTYPE: int msgsnd ( int msqid, struct msgbuf *msgp, int msgsz, int msgflg
);
    RETURNS: 0 on success
             -1 on error: errno = EAGAIN (queue is full, and IPC_NOWAIT was
asserted)
                                  EACCES (permission denied, no write permis-
sion)
```

```
                                        EFAULT (msgp address isn't accessable -
invalid)

                                        EIDRM  (The message queue has been removed)
                                        EINTR  (Received a signal while waiting to
write)

                                        EINVAL (Invalid message queue identifier,
nonpositive

                                                message type, or invalid message
size)

                                        ENOMEM (Not enough memory to copy message
buffer)
```

msgsnd的第一个变元是队列标识符，它是前面调用 msgget获得的返回值。第二个变元是msgp，它是一个指针，指向我们重新定义和载入的消息缓冲区。 msgsz变元则包含着消息的大小，它是以字节为单位的，其中不包括消息类型的长度 (四个字节长)。

msgflg变元可以设置为0(忽略)，也可以设置为IPC_NOWAIT。如果消息队列已满，则消息将不会被写入到队列中，控制权将被还给调用进程。如果没有指定 IPC_NOWAIT，则调用进程将被中断(阻塞)，直到可以写消息为止。

下面创建另一个发送消息的包装程序：

```
int send_message( int qid, struct mymsgbuf *qbuf )
{
        int     result, length;

        /* The length is essentially the size of the structure minus
sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);
        if((result = msgsnd( qid, qbuf, length, 0)) == -1)
        {
                return(-1);
        }

        return(result);
}
```

这个小函数接收到一个地址 (qbuf)，并试着把驻留在那个地址中的消息发送到消息队列标志符(qid)所指定的消息队列中， qid也是作为一个参数传送给该函数的。下面这个示例代码片段将用到前面创建的那两个包装函数：

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
#include <linux/msg.h>

main()
{
        int    qid;
        key_t  msgkey;
        struct mymsgbuf {
                long    mtype;          /* Message type */
                int     request;        /* Work request number */
```

```
        double  salary;         /* Employee's salary */
    } msg;

    /* Generate our IPC key value */
    msgkey = ftok(".", 'm');

    /* Open/create the queue */
    if(( qid = open_queue( msgkey)) == -1) {
            perror("open_queue");
            exit(1);
    }

    /* Load up the message with arbitrary test data */
    msg.mtype   = 1;        /* Message type must be a positive number! */
    msg.request = 1;        /* Data element #1 */
    msg.salary  = 1000.00;  /* Data element #2 (my yearly salary!) */

    /* Bombs away! */
    if((send_message( qid, &msg )) == -1) {
            perror("send_message");
            exit(1);
    }
}
```

在创建或者打开了消息队列以后，接下去用测试数据装填消息缓冲区 (注意这里没有使用字符数据，这是为了说明我们有关传送二进制信息的观点)，调用send_message，它会把消息传送到消息队列中。

现在既然消息队列有一个消息，用户可以使用 ipcs命令来查看队列的状态。下面将介绍如何真正从队列中获取消息。为了做到这点，可以使用系统调用 msgrcv ( )：

```
  SYSTEM CALL: msgrcv();
  PROTOTYPE: int msgrcv ( int msqid, struct msgbuf *msgp, int msgsz, long
mtype, int msgflg );
    RETURNS: Number of bytes copied into message buffer
             -1 on error: errno = E2BIG  (Message length is greater than msgsz,
no MSG_NOERROR)
                                  EACCES (No read permission)
                                  EFAULT (Address pointed to by msgp is in-
valid)
                                  EIDRM  (Queue was removed during retrieval)
                                  EINTR  (Interrupted by arriving signal)
                                  EINVAL (msgqid invalid, or msgsz less than 0)
                                  ENOMSG (IPC_NOWAIT asserted, and no message
exists
                                          in the queue to satisfy the request)
```

显然，第一个变元是用来指定在消息获取过程中所使用的队列的 (该值是由前面调用msgget得到的返回值)。第二个变元(msgp)代表消息缓冲区变量的地址，获取的消息将存放在这里。第三个变元(msgsz)代表消息缓冲区结构的大小，不包括 mtype成员的长度。再说一遍，可以使用下面的公式来计算大小：

```
msgsz = sizeof(struct mymsgbuf) - sizeof(long);
```

第四个变元(mtype)指定要从队列中获取的消息的类型。内核将查找队列中具有匹配类型的最老的消息，并把它的一个拷贝返回到由 msgp变元所指定的地址中。这里存在一种特殊的情况：如果mtype变元传送一个为零的值，则将返回队列中最老的消息，不管该消息的类型是什么。

如果把IPC_NOWAIT作为一个标志传送给该系统调用，而队列中没有任何消息。则该次调用将会向调用进程返回ENOMSG。否则，调用进程将阻塞，直到满足 msgrcv ( )参数的消息到达队列为止。如果在客户等待消息的时候队列被删除了，则返回 EIDRM。如果在进程阻塞并等待消息的到来时捕获到一个信号，则返回 EINTR。

请看下面这个从队列中获取消息的包装函数：

```
int read_message( int qid, long type, struct mymsgbuf *qbuf )
{
        int     result, length;

        /* The length is essentially the size of the structure minus
sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);

        if((result = msgrcv( qid, qbuf, length, type,  0)) == -1)
        {
                return(-1);
        }

        return(result);
}
```

在成功地从队列中获取了一个消息之后，队列中的消息项目将被摧毁。

msgflg变元中的MSG_NOERROR位提供了一些其他的功能。如果物理消息数据的大小比msgsz要大，同时又声明了MSG_NOERROR位，则消息将被截断，并只返回 msgsz个字节。一般来说，msgrcv ( )系统调用将返回－1(E2BIG)，并且该消息将保留在队列中，以供以后获取。可以利用这个特点来创建另一个包装程序，该程序使用户能够窥探队列的内部，看看是否有满足请求的消息到达：

```
int peek_message( int qid, long type )
{
        int     result, length;

        if((result = msgrcv( qid, NULL, 0, type,  IPC_NOWAIT)) == -1)
        {
                if(errno == E2BIG)
                        return(TRUE);
        }

        return(FALSE);
}
```

在上面这个程序中，细心的读者可以发现缺少缓冲区地址和长度。在这种特殊的情况下，

我们希望调用出错。然而，我们可以检测 E2BIG的返回，它显示了确实存在匹配所申请的类型的消息。如果成功，则包装程序将返回 TRUE，否则将返回 FALSE。同时请读者注意IPC_NOWAIT标志的使用，它防止了前面介绍过的阻塞行为的发生。

系统调用：msgctl ( )

通过前面介绍的那些包装程序的开发，读者现在应该知道怎样在应用程序中简单地，但同时也是聪明地创建和利用消息队列。下面再回头介绍一下如何直接地对那些与特定的消息队列相联系的内部结构进行操作。

为了在一个消息队列上执行控制操作，用户可以使用 msgctl ( )系统调用。

```
  SYSTEM CALL: msgctl();
  PROTOTYPE: int msgctl ( int msgqid, int cmd, struct msqid_ds *buf );
    RETURNS: 0 on success
             -1 on error: errno = EACCES (No read permission and cmd is
IPC_STAT)
                                  EFAULT (Address pointed to by buf is invalid
with IPC_SET and
                                          IPC_STAT commands)
                                  EIDRM  (Queue was removed during retrieval)
                                  EINVAL (msgqid invalid, or msgsz less than 0)
                                  EPERM  (IPC_SET or IPC_RMID command was
issued, but
                                          calling process does not have write
(alter)
                                          access to the queue)
  NOTES:
```

现在有一种普遍的观点，认为直接对内部内核数据结构进行操作会给人带来一种满足感和成就感。不幸的是，由此而给程序员方面带来的责任却不是那么轻松的，除非用户喜欢破坏自己的IPC子系统。通过使用具有一个命令可选集合的 msgctl ( )，用户可以操纵那些不太容易造成破坏的项目。请看这些命令：

IPC_STAT

获取队列的msqid_ds结构，并把它存放在buf变元所指定的地址中。

IPC_SET

设置队列的msqid_ds结构的ipc_perm成员的值。它是从buf变元中取得该值的。

IPC_RMID

从内核删除队列。

前面介绍过消息队列的内部数据结构 (msqid_ds)。对于系统中存在的每一个队列，内核为它们都维护该结构的一个实例。通过使用 IPC_STAT命令，用户可以获取该结构的一个拷贝以供检查。请看下面这个包装程序函数，它将获取内部结构，并将其拷贝到作为参数传送给它的一个地址中：

```
int get_queue_ds( int qid, struct msgqid_ds *qbuf )
{
        if( msgctl( qid, IPC_STAT, qbuf) == -1)
        {
                return(-1);
        }
```

```
        return(0);
}
```

如果不能拷贝这个内部缓冲区，函数将向调用函数返回－ 1。如果一切工作顺利，则将返回一个为零的值，并且缓冲区中将会包含队列标识符 (qid)所代表的消息队列的内部数据结构的一个拷贝。缓冲区和qid都是作为参数传送给该函数的。

既然已经拥有了队列的内部数据结构的一个拷贝，那么，用户可以操作什么属性？用户应该怎样改变它们呢？在那个数据结构中，唯一可以修改的项目就是 ipc_perm成员。它包含该队列的许可权限，以及关于拥有者和创建者的信息。然而， ipc_perm结构中唯一可以修改的成员是mode、uid和gid。用户可以改变拥有者的用户 ID，拥有者的组ID以及队列的访问许可权限。

下面创建一个包装程序函数，用于改变队列的模式。该模式必须以字符数组的形式传送(如“ 660 ”)。

```
int change_queue_mode( int qid, char *mode )
{
        struct msqid_ds tmpbuf;

        /* Retrieve a current copy of the internal data structure */
        get_queue_ds( qid, &tmpbuf);

        /* Change the permissions using an old trick */
        sscanf(mode, "%ho", &tmpbuf.msg_perm.mode);

        /* Update the internal data structure */
        if( msgctl( qid, IPC_SET, &tmpbuf) == -1)
        {
                return(-1);
        }

        return(0);
}
```

通过调用get_queue_ds包装程序函数，可以获取内部数据结构的当前拷贝。然后再调用sscanf ( )，改变相联系的msg_perm结构的方式成员。然而，在用新拷贝更新当前内部版本之前，没有发生任何变化。这是通过使用 IPC_SET命令调用msgctl ( )来实现的。

注意！用户可以修改队列的许可权限。然而在实施过程中稍微的粗心大意会导致难以预料的后果。记住，这些IPC对象不会自动消失，除非它们被正确地删除，或者系统被重启。所以，即使用ipcs命令无法看到队列，也不意味着它不存在了。

为了说明这一点，有一则在某种程度上可以称得上幽默的轶事比较说明问题。在南佛罗里达大学教一门Unix内部结构课程时，我曾经遇到过一个相对尴尬的问题。为了编译和测试一个实验例题以便在我一个星期长的教学中使用，我在上课前的晚上进入实验服务程序。在测试过程中，我发现在用于改变某消息队列的许可权限的代码中，出现了一个打字错误。我创建了一个简单的消息队列，在测试发送和接收能力时并没有什么问题，然而当我试图把队列的访问许可方式从“ 660 ”变为“ 600 ”时，所带来的后果居然是我无法访问自己的队列了。因此，在源目录的同一个目录下，我不能测试消息队列的实验例题。因为我使用 ftok ( )函数

来创建IPC关键字，所以实际上我是企图访问自己没有正确访问权限的队列。最后，在上课前的那个早晨，我联系到了本地系统管理员，花了整整一个小时向他解释这是个什么样的消息队列，而我为什么需要他执行一次ipcrm命令帮我删除该队列。

在成功地从队列获取了一个消息以后，该消息将被删除。然而，正如前面所介绍的，除非用户显式地删除它，或者系统被重启，否则 IPC对象将保留在系统中。因此，消息队列还存在于内核中，在单个消息消失以后很长时间内仍然可用。为了结束一个消息队列的生命周期，用户需要使用IPC_RMID命令来调用msgctl ( )，以便显式地删除它：

```
int remove_queue( int qid )
{
        if( msgctl( qid, IPC_RMID, 0) == -1)
        {
                return(-1);
        }

        return(0);
}
```

如果队列被删除且无错误发生，则上面这个包装程序函数将返回 0，否则就将返回－ 1。队列的删除动作是原子性的，并且以后不论用什么理由访问该队列都将失败。

8. msgtool：交互式消息队列操作程序

如果随时可以得到正确的技术方面的信息，那将会给用户带来很直接的利益，没有人能否认这一点。对于学习和研究新的技术领域，信息和资料会提供一个强有力的帮助。同理可知，如果伴随着这些技术信息还能提供一些实际应用的例子，这无疑会加速学习进程，增强学习效果。

到目前为止，本书所提供的几个有用的例子是操作消息队列的包装程序函数。虽然它们是非常有用的，但它们的表达方式还不足以保证下一步的学习和实验。为了弥补这一点，我们给出msgtool，这是一个操作IPC消息队列的交互式命令行工具。虽然它实际上经常作为一个工具用于教学目的，但是，通过利用标准 shell脚本提供消息队列功能，该工具也可以直接应用在实际的赋值操作中。

9. 背景知识

msgtool依赖于命令行变元来确定自己的行为。当从 shell脚本调用时，这个特征使它显得尤其有用。该工具提供了所有的功能，从创建、发送和接收，到改变许可权限以及最后删除队列。当前，它使用字符数组来作数据，允许用户发送文本消息。当然也可以改变这一点，使它可以发送其他类型数据的工作。我们当作一个练习把它留给读者去完成。

10. 命令行语法

发送消息

```
msgtool s (type) "text"
```

获取消息

```
msgtool r (type)
```

改变许可权限(模式)

```
msgtool m (mode)
```

删除队列

```
msgtool d
```

11. 实例

```
msgtool  s   1 test
msgtool  s   5 test
msgtool  s   1 "This is a test"
msgtool  r   1
msgtool  d
msgtool  m   660
```

12. 源代码

下面就是msgtool工具的源代码。应该在支持系统V IPC的最新内核版本上编译这段代码。在作重建时一定要在内核中使能系统V IPC。

顺便提一句，不论请求执行哪种类型的动作，这个工具在消息队列不存在时都会把它创建出来。

因为这个工具使用ftok ( )函数来生成IPC关键字的值。用户可能会遇到目录冲突的问题。如果在脚本的任意位置改变目录，它可能不会工作。另一种解决方案是把一个更复杂的路径(如“ /tmp/msgtool ”)硬编码进msgtool中，或者甚至把路径和其他可操作的变元一起通过命令行传送。

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: msgtool.c
 *************************************************************************
 A command line tool for tinkering with SysV style Message Queues
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_SEND_SIZE 80

struct mymsgbuf {
        long mtype;
        char mtext[MAX_SEND_SIZE];


void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);
    int main(int argc, char *argv[])
    {
```

```
        key_t key;
        int   msgqueue_id;
        struct mymsgbuf qbuf;

        if(argc == 1)
                usage();

        /* Create unique key via call to ftok() */
        key = ftok(".", 'm');

        /* Open the queue - create if necessary */
        if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1) {
                perror("msgget");
                exit(1);
        }

        switch(tolower(argv[1][0]))
        {
                case 's': send_message(msgqueue_id, (struct mymsgbuf *)&qbuf,
                                        atol(argv[2]), argv[3]);
                          break;
                case 'r': read_message(msgqueue_id, &qbuf, atol(argv[2]));
                          break;
                case 'd': remove_queue(msgqueue_id);
                          break;
                case 'm': change_queue_mode(msgqueue_id, argv[2]);
                          break;

                 default: usage();

        }

        return(0);
}

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
        /* Send a message to the queue */
        printf("Sending a message ...\n");
        qbuf->mtype = type;
        strcpy(qbuf->mtext, text);

        if((msgsnd(qid, (struct msgbuf *)qbuf,
                strlen(qbuf->mtext)+1, 0)) ==-1)
        {
                perror("msgsnd");
                exit(1);
        }
}

void read_message(int qid, struct mymsgbuf *qbuf, long type)
```

```
{
        /* Read a message from the queue */
        printf("Reading a message ...\n");
        qbuf->mtype = type;
        msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);

        printf("Type: %ld Text: %s\n", qbuf->mtype, qbuf->mtext);
}

void remove_queue(int qid)
{
        /* Remove the queue */
        msgctl(qid, IPC_RMID, 0);
}

void change_queue_mode(int qid, char *mode)
{
        struct msqid_ds myqueue_ds;

        /* Get current info */
        msgctl(qid, IPC_STAT, &myqueue_ds);

        /* Convert and load the mode */
        sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);

        /* Update the mode */
        msgctl(qid, IPC_SET, &myqueue_ds);
}

void usage(void)
{
        fprintf(stderr, "msgtool - A utility for tinkering with msg queues\n");
        fprintf(stderr, "\nUSAGE: msgtool (s)end <type> <messagetext>\n");
        fprintf(stderr, "               (r)ecv <type>\n");
        fprintf(stderr, "               (d)elete\n");
        fprintf(stderr, "               (m)ode <octal mode>\n");
        exit(1);
}
```

6.4.3 信号量

1. 基本概念

信号量的最佳定义是：它是一种计数器，用来控制对多个进程共享的资源所进行的访问。它们常常被用作一个锁机制，在某个进程正在对特定资源进行操作时，信号量可以防止另一个进程去访问它。信号量常常被认为是系统 V的这三类IPC对象中最难掌握的一种。为了充分地理解信号量，在介绍任何系统调用和操作原理以前，将会先简要地讨论一下。

信号量这个名称实际上是个古老的铁路方面的术语，指的是在交叉路口的“ 长臂 ”，它把汽车挡住，防止它们在有火车经过时横穿铁轨。这实际上也可以说是一个简单的信号量集合。如果信号量打开(长臂向上举着)，那么资源是可以使用的(汽车可以横穿铁轨)。然而，如果信

号量关闭(长臂已经放下来了)，那么资源就不能使用(汽车必须等待)。

虽然可以用这个简单的例子来引入信号量的概念，但读者应该认识到信号量实际上是以集合的形式实现的，而不是作为单个实体实现的，认识这点很重要。当然，某个特定的信号量集合中可能只有一个信号量，就象在铁路的例子中一样。

要理解信号量这个概念，另一个方法可能就是把它们当作资源计数器。让我们把这个概念应用到另一种实际情况上。假设有一个打印假脱机程序，它可以处理多个打印机，而每个打印机需要处理多个打印请求。假脱机打印管理员将会使用信号量集合来监测对每个打印机的访问。

假设在我们团体的打印房间中有五个打印机在线。假脱机打印管理员分配一个信号量集合，它包括五个信号量，每个信号量对应系统上的一个打印机。因为在物理上每个打印机每次只能打印一个任务，在集合中的这五个信号量每个都将初始化为值等于 1，这意味着它们都在线，并且可以接受请求。

John向假脱机程序发送一个打印请求。打印管理员查看信号量集合，找到第一个值为 1的信号量。在把John的打印请求发送给物理设备之前，打印管理员使用一个为－ 1的值，使对应打印机的信号量减一。这样一来，那个信号量的值现在为 0。在系统V的信号量中，值为0就代表在那个信号量上的资源被百分之百地利用了。在我们的例子中，不能再把其他的请求发送给那个打印机了，直到信号量的值不再等于 0。

在John的打印任务完成之后，打印管理员将使对应打印机的信号量的值增加一。它的值现在又回到了1。这意味着那个打印机又可以使用了。一般来说，如果所有的 5个信号量值都为0，则表示它们都在忙着完成打印请求，即当前没有打印机可用。

尽管这是个简单的例子，但是请不要被赋予给集合中每个信号量的初值 1所迷惑了。信号量尽管被当成是资源计数器，它还可以被初始化为任何一个正的整数值，并不局限于或者为 1或者为0。如果这5个打印机每个可以同时处理 10个打印任务，用户就可以把每个信号量都初始化为10，每到达一个新任务则减去一，每完成一个打印任务则增加一。在下一章中读者将发现，信号量与共享内存段的工作关系非常紧密。共享内存段的任务就像一条看门狗，它可以防止多个进程同时写同一个内存段。

在介绍相关的系统调用之前，我们简单地了解一下在信号量操作中用到的各种内部数据结构。

2. 内部数据结构

让我们简单地了解一下内核为信号量集合所维护的数据结构。

3. 内核semid_ds结构

和消息队列一样，内核也为存在于它的寻址空间中的每个信号量集合维护一个特殊的内部数据结构。这个结构的类型是semid_ds，该类型的定义在linux/sem.h中，如下所示：

```
        /* One semid data structure for each set of semaphores in the system.
*/
        struct semid_ds {
                struct ipc_perm sem_perm;       /* permissions .. see ipc.h */
                time_t          sem_otime;      /* last semop time */
                time_t          sem_ctime;      /* last change time */
                struct sem      *sem_base;      /* ptr to first semaphore in
array */
```

```
                struct wait_queue *eventn;
                struct wait_queue *eventz;
                struct sem_undo  *undo;          /* undo requests on this array
*/
                ushort           sem_nsems;      /* no. of semaphores in array
*/
        };
```

与消息队列一样，这个结构上的操作是由某个特殊的系统调用执行的，并且不能直接对其作任何操作。下面是最重要的字段的描述：

sem_perm

这是ipc_perm结构的一个实例。它是在 linux/ipc.h中定义的，它里面存放的是信号量集合的许可权限信息，包括访问许可权限，以及关于集合的创建者的有关信息 (uid等等)。

sem_otime

最后一个semop ( )操作的时间(稍后再作详细介绍)。

sem_ctime

最后一次修改这个结构(改变方式等等)的时间。

sem_base

这是一个指针，指向数组中第一个信号量 (参见下一个结构)。

sem_undo

在这个数组中的undo请求的次数(稍后再作详细介绍)。

sem_nsems

在该信号量集合(数组)中信号量的数目。

4. 内核sem结构

在semid_ds结构中，存在一个指针，指向信号量数组的基址。每个数组成员都是 sem类型的。Sem结构也是在linux/sem.h中定义的，如下所示：

```
        /* One semaphore structure for each semaphore in the system. */
        struct sem {
                short   sempid;         /* pid of last operation */
                ushort  semval;         /* current value */
                ushort  semncnt;        /* num procs awaiting increase in
semval */
                ushort  semzcnt;        /* num procs awaiting semval = 0 */
        };
```

sem_pid

执行最后一次操作的PID(进程ID号)。

sem_semval

信号量的当前值。

sem_semncnt

等待资源变为可用状态的进程数目。

sem_semzcnt

等待资源利用率达到百分之百的进程的数目。

5. 系统调用：semget ( )

为了创建一个新的信号量集合，或者访问现有的集合，可以使用系统调用 semget ( )。

```
SYSTEM CALL: semget();

PROTOTYPE: int semget ( key_t key, int nsems, int semflg );
  RETURNS: semaphore set IPC identifier on success
           -1 on error: errno = EACCESS (permission denied)
                                EEXIST (set exists, cannot create (IPC_EXCL))
                                EIDRM (set is marked for deletion)
                                ENOENT (set does not exist, no IPC_CREAT was
used)
                                ENOMEM (Not enough memory to create new set)
                                ENOSPC (Maximum set limit exceeded)

NOTES:
```

semget ( )系统调用的第一个变元是关键字的值 (在我们的例子中，该值是调用 ftok ( )的返回值)。然后再把该关键字值与内核中现有的其他信号量集合的关键字值都比较。在比较之后，打开或者访问操作依赖于semflg变元的内容。

IPC_CREAT

如果内核中不存在这样的信号量集合，则把它创建出来。

IPC_EXCL

当与IPC_CREAT一起使用时，如果信号量集合早已存在，则操作将失败。

如果单独使用IPC_CREAT，semget ( )或者返回新创建的信号量集合的信号量集合标识符，或者返回早已存在的具有同一个关键字值的集合的标识符。如果同时使用 IPC_EXCL和IPC_CREAT，那么将有两种可能的结果：如果集合不存在，则创建一个新的集合；如果集合早已存在，则调用失败，并返回－1。IPC_EXCL本身是没有什么用处的，但当与 IPC_CREAT组合使用时，它可以用于保证没有为了访问而打开现有的信号量集合。

与系统V IPC的其他种类一样，可以把信号量集合的许可权限与掩码进行 OR操作，得到一个可选的八进制形式的许可模式。

nsems变元可以指定在新的集合中应该创建的信号量的数目。在前面所介绍的虚构的打印房间的例子中，这个值代表打印机的数目。在集合中最多能容纳的信号量的数目是在linux/sem.h中定义的，如下所示：

```
#define SEMMSL  32      /* <=512 max num of semaphores per id */
```

注意，如果用户显式地打开某个现有的集合，则 nsems变元将被忽略。

下面创建一个包装程序函数，用来打开或创建信号量集合：

```
int open_semaphore_set( key_t keyval, int numsems )
{
        int     sid;

        if ( ! numsems )
                return(-1);

        if((sid = semget( mykey, numsems, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }
```

```
        return(sid);
}
```

请注意函数中使用了显式的许可权限 660，这个小函数要么返回一个信号量集合标识符(整型)，要么出错并返回－1，必须把关键字值传送给它。如果是创建新集合的话，还需要传递信号量的数目，以便给它们分配空间。在本章最后所给出的例子中，读者可以注意到使用了IPC_EXCL标志，这是为了判断该信号量集合是否存在。

6. 系统调用：semop ( )

```
  SYSTEM CALL: semop();
  PROTOTYPE: int semop ( int semid, struct sembuf *sops, unsigned nsops);
    RETURNS: 0 on success (all operations performed)
             -1 on error: errno = E2BIG (nsops greater than max number of ops
allowed atomically)
                                  EACCESS (permission denied)
                                  EAGAIN (IPC_NOWAIT asserted, operation could
not go through)
                                  EFAULT (invalid address pointed to by sops
argument)
                                  EIDRM (semaphore set was removed)
                                  EINTR (Signal received while sleeping)
                                  EINVAL (set doesn't exist, or semid is
invalid)
                                  ENOMEM (SEM_UNDO asserted, not enough memory
to create the
                                          undo structure necessary)
                                  ERANGE (semaphore value out of range)
    NOTES:
```

semop ( )的第一个变元是关键字的值(在我们的例子中，它是调用semget所获得的返回值)。第二个变元(sops)是一个指针，指向将要在信号量集合上执行的操作的一个数组，而第三个变元(nsops)则是该数组中操作的个数。

sops变元指向的是类型为sembuf的结构的一个数组。sembuf结构是在linux/sem.h中定义的，如下所示：

```
        /* semop system call takes an array of these */
        struct sembuf {
                ushort  sem_num;        /* semaphore index in array */
                short   sem_op;         /* semaphore operation */
                short   sem_flg;        /* operation flags */
        };
```

sem_num

用户希望处理的信号量的编号。

sem_op

将要执行的操作(正、负，或者零)。

sem_flg

可操作的标志。

如果sem_op为负，则它的值需要从信号量中减掉。这对应着获得信号量所控制的或者监

视器所访问的资源。如果没有指定 IPC_NOWAIT，那么调用进程将睡眠，直到在信号量中已经有了能够满足需求的数量的资源(另一个进程释放了一些资源)。

如果sem_op为正，则它的值将被加到信号量上。这对应着把资源归还给应用程序的信号量集合。当不再需要资源时，应该记得把它们归还给信号量集合！

最后，如果sem_op为零(0)，则调用进程将睡眠，直到信号量的值为零为止。这对应于等待信号量的利用率达到百分之百。关于这点有一个很好的例子。如一个以超级用户权限运行的守护进程，当信号量集合达到完全的利用率时，它可以动态地调整信号量集合的大小。

为了解释semop调用，读者可以回想一下以前介绍过的有关打印房间的例子。假设我们只有一台打印机，每次只能处理一个任务。在创建信号量集合时，应该让它只含一个信号量 (只有一台打印机)，并且把那个信号量初始化为值等于 1(每次只能处理一个任务)。

每次用户需要向这台打印机发送一个任务时，首先需要确定资源是可用的。为了做到这一点，用户可以尝试着从信号量获得一个单元的资源。可以通过装入一个 sembuf数组来执行这个操作：

```
struct sembuf sem_lock = { 0, -1, IPC_NOWAIT };
```

我们解释一下上面的这个初始化结构的含义，它意味着将在信号量集合的 0号信号量上加上一个值－1。换句话说，将从集合中唯一的信号量 (第0号)获得一个单元的资源。在这里指定了IPC_NOWAIT，所以这次调用或者将立刻被满足，或者如果另一个打印任务正在打印，则调用出错。下面是在semop系统调用中使用初始化sembuf结构的一个例子：

```
if((semop(sid, &sem_lock, 1) == -1)
        perror("semop");
```

第三个变元(nsops)显示我们只执行一个操作 (在操作数组中只有一个 sembuf结构)。变元sid是信号量集合的IPC标识符。

在完成了打印任务之后，我们必须把资源归还给信号量集合，这样其他人就可以使用打印机了。

```
struct sembuf sem_unlock = { 0, 1, IPC_NOWAIT };
```

上述这个初始化结构的意义是：信号量集合中的第 0号信号量将被增加一个为 1的值。换句话说，有一个单元的资源被归还给集合。

7. 系统调用：semctl ( )

```
SYSTEM CALL: semctl();
PROTOTYPE: int semctl ( int semid, int semnum, int cmd, union semun arg );
  RETURNS: positive integer on success
           -1 on error: errno = EACCESS (permission denied)
                                EFAULT (invalid address pointed to by arg
argument)
                                EIDRM (semaphore set was removed)
                                EINVAL (set doesn't exist, or semid is
invalid)
                                EPERM (EUID has no privileges for cmd in arg)
                                ERANGE (semaphore value out of range)
NOTES: Performs control operations on a semaphore set
```

系统调用 semctl用于在信号量集合上执行控制操作。这个调用类似于系统调用 msgctl，msgctl是用于消息队列上的操作。如果读者比较一下两个系统操作的变元列表，将会发现

semctl和msgctl的列表之间差别非常小。我们知道信号量实际上是以集合的形式实现的，而不是以单个实体的形式实现的。对于信号量操作而言，不光需要传递 IPC关键字，还需要传递集合中的目标信号量。

这两个系统调用都使用到一个 cmd变元，用于指定在IPC对象上执行的命令。剩下的区别是两个调用的最后一个变元。在msgctl中，最后一个变元代表的是内核所使用的内部数据结构的一个拷贝。我们使用该结构来获取有关消息队列的内部信息，以及设置和改变队列的许可权限和拥有权。对信号量而言，它还支持其他的可操作命令，这样就需要一个更复杂的数据类型来作最后一个变元。对许多初学信号量的编程人员来说，联合体的使用是一个相当困惑的难题。我们将认真地对这个结构进行详细研究，以尽可能避免读者的困惑。

semctl ( )的第一个变元是关键字的值(在我们的例子中它是调用semget所返回的值)。第二个变元(semun)是将要执行操作的信号量的编号。大致而言，它可以看成是信号量集合的一个索引值，对于集合中的第一个信号量 (有可能只有这一个信号量)来说，它的索引值将是一个为零的值。

cmd变元代表将要在集合上执行的命令。正如读者将会看到的，这里包括我们熟悉的IPC_STAT/IPC_SET命令，以及其他一些信号量集合所特有的命令：

IPC_STAT

获取某个集合的semid_ds结构，并把它存储在semun联合体的buf变元所指定的地址中。

IPC_SET

设置某个集合的semid_ds结构的ipc_perm成员的值。该命令所取的值是从 semun联合体的buf变元中取到的。

IPC_RMID

从内核删除该集合。

GETALL

用于获取集合中所有信号量的值。整数值存放在无符号短整数的一个数组中，该数组由联合体的array成员所指定。

GETNCNT

返回当前正在等待资源的进程的数目。

GETPID

返回最后一次执行semop调用的进程的PID。

GETVAL

返回集合中某个信号量的值。

GETZCNT

返回正在等待资源利用率达到百分之百的进程的数目。

SETALL

把集合中所有信号量的值设置为联合体的 array成员所包含的对应值。

SETVAL

把集合中单个信号量的值设置为联合体的 val成员的值。

变元arg代表类型semun的一个实例。这个特殊的联合体是在 linux/sem.h中定义的，如下所示：

```
/* arg for semctl system calls. */
union semun {
        int val;                /* value for SETVAL */
        struct semid_ds *buf;   /* buffer for IPC_STAT & IPC_SET */
        ushort *array;          /* array for GETALL & SETALL */
        struct seminfo *__buf;  /* buffer for IPC_INFO */
        void *__pad;
};
```

val

当执行SETVAL命令时将用到这个成员，它用于指定要把信号量设置成什么值。

buf

在命令IPC_STAT/IPC_SET中使用。它代表内核中所使用的内部信号量数据结构的一个拷贝。

array

用在GETALL/SETALL命令中的一个指针。它应当指向整数值的一个数组。在设置或获取集合中所有信号量的值的过程中，将会用到该数组。

剩下的变元_buf和_pad将在内核中的信号量代码中内部使用，对于应用程序开发人员来说，它们用处很少，或者说没有用处。一个明显的事实是，这两个变元是 Linux操作系统所特有的，在其他的Unix实现中没有。

因为这个特殊的系统调用被认为是所有系统 V IPC调用中最难掌握的，我们将列出它的多个使用实例。

下面的这个代码片段返回作为参数传递给它的信号量的值。在使用 GETVAL命令时，最后一个变元(联合体)将被忽略：

```
int get_sem_val( int sid, int semnum )
{
        return( semctl(sid, semnum, GETVAL, 0));
}
```

我们再看一下打印机的那个例子，假设所有五个打印机的状态都需要了解：

```
#define MAX_PRINTERS 5

printer_usage()
{
        int x;

        for(x=0; x<MAX_PRINTERS; x++)
                printf("Printer %d: %d\n\r", x, get_sem_val( sid, x ));
}
```

请再看下面这个函数，它可以用来初始化一个新的信号量的值：

```
void init_semaphore( int sid, int semnum, int initval)
{
        union semun semopts;

        semopts.val = initval;
```

```
        semctl( sid, semnum, SETVAL, semopts);
}
```

注意，semctl的最后一个变元是联合体的一个拷贝，而不是指向它的一个指针。虽然目前正在介绍的内容是把这个联合体当作变元使用的，但还是请允许我介绍一个在使用这个系统调用的时候相当普遍的一个错误。

在msgtool的实例中，我们用IPC_STAT和IPC_SET命令来改变队列的许可权限。虽然信号量也支持这些命令，它们的用法稍有不同。这是因为内部数据结构是从联合体的一个成员获取和拷贝的，而不是从单个实体获得的。读者能找出下面这段程序中的错误吗？

```
/* Required permissions should be passed in as text (ex: "660") */

void changemode(int sid, char *mode)
{
        int rc;
        struct semid_ds mysemds;

        /* Get current values for internal data structure */
        if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
        {
                perror("semctl");
                exit(1);
        }
        printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);

        /* Change the permissions on the semaphore */
        sscanf(mode, "%o", &semopts.buf->sem_perm.mode);

        /* Update the internal data structure */
        semctl(sid, 0, IPC_SET, semopts);

        printf("Updated...\n");
```

该段代码试图为集合所使用的内部数据结构制作一个拷贝，修改许可权限，并用IPC_SET命令把它们重置回内核。然而，第一次调用 semctl将立刻返回EFAULT。或者返回最后一个变元(联合体！)的地址。此外，如果我们不去查看那次调用的错误，我们将会导致内存出错。读者知道为什么吗？

我们知道 IPC_SET/IPC_STAT命令使用的是联合体的 buf成员，它是一个指向类型semid_ds的指针。指针就是指针，不是别的什么！ buf成员必须指向一些合法的存储位置，这样我们的函数才能正常工作。请看下面这个修改过的版本：

```
void changemode(int sid, char *mode)
{
        int rc;
        struct semid_ds mysemds;

        /* Get current values for internal data structure */
```

```
/* Point to our local copy first! */
semopts.buf = &mysemds;

/* Let's try this again! */
if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
{
        perror("semctl");
        exit(1);
}

printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);

/* Change the permissions on the semaphore */
sscanf(mode, "%o", &semopts.buf->sem_perm.mode);

/* Update the internal data structure */
semctl(sid, 0, IPC_SET, semopts);
        printf("Updated...\n");
}
```

6.4.4 semtool：交互式信号量操作程序

1. 背景知识

semtool程序依赖于命令行变元来决定自己的行为。当从 shell脚本调用它时，这个特征使它显得尤其有用。 semtool提供了所有的功能，从创建和操作，到修改许可权限以及最后删除信号量集合，它可以用于通过标准 shell脚本来控制共享资源。

2. 命令行语法

创建信号量集合

semtool c(集合中信号量的个数)

锁住某个信号量

semtool l(要锁住的信号量的编号)

解锁某个信号量

semtool u(要解锁的信号量的编号)

改变许可权限(模式)

semtool m (mode)

删除信号量集合

semtool d

3. 实例

```
semtool  c 5
semtool  l
semtool  u
semtool  m 660
semtool  d
```

4. 源代码

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"

 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: semtool.c
 *************************************************************************
 A command line tool for tinkering with SysV style Semaphore Sets

 ************************************************************************/

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEM_RESOURCE_MAX        1       /* Initial value of all semaphores */

void opensem(int *sid, key_t key);
void createsem(int *sid, key_t key, int members);
void locksem(int sid, int member);
void unlocksem(int sid, int member);
void removesem(int sid);
unsigned short get_member_count(int sid);
int getval(int sid, int member);
void dispval(int sid, int member);
void changemode(int sid, char *mode);
void usage(void);

int main(int argc, char *argv[])
{
        key_t key;
        int   semset_id;

        if(argc == 1)
                usage();

        /* Create unique key via call to ftok() */
        key = ftok(".", 's');

        switch(tolower(argv[1][0]))
        {
                case 'c': if(argc != 3)
                                usage();
                          createsem(&semset_id, key,  atoi(argv[2]));
                          break;
                case 'l': if(argc != 3)
                                usage();
                          opensem(&semset_id, key);
```

```
                        locksem(semset_id, atoi(argv[2]));
                        break;
                case 'u': if(argc != 3)
                                usage();
                        opensem(&semset_id, key);
                        unlocksem(semset_id, atoi(argv[2]));
                        break;
                case 'd': opensem(&semset_id, key);
                        removesem(semset_id);
                        break;
                case 'm': opensem(&semset_id, key);
                        changemode(semset_id, argv[2]);
                        break;
                 default: usage();

        }

        return(0);
}

void opensem(int *sid, key_t key)
{
        /* Open the semaphore set - do not create! */

        if((*sid = semget(key, 0, 0666)) == -1)
        {
                printf("Semaphore set does not exist!\n");
                exit(1);
        }

}

void createsem(int *sid, key_t key, int members)
{
        int cntr;
        union semun semopts;

        if(members > SEMMSL) {
                printf("Sorry, max number of semaphores in a set is %d\n",
                        SEMMSL);
                exit(1);
        }

        printf("Attempting to create new semaphore set with %d members\n",
                                members);
        if((*sid = semget(key, members, IPC_CREAT|IPC_EXCL|0666))
                        == -1)
        {
                fprintf(stderr, "Semaphore set already exists!\n");
                exit(1);
```

```
        }

        semopts.val = SEM_RESOURCE_MAX;

        /* Initialize all members (could be done with SETALL) */
        for(cntr=0; cntr<members; cntr++)
                semctl(*sid, cntr, SETVAL, semopts);
}

void locksem(int sid, int member)
{
        struct sembuf sem_lock={ 0, -1, IPC_NOWAIT};

        if( member<0 || member>(get_member_count(sid)-1))
        {
                fprintf(stderr, "semaphore member %d out of range\n", member);
                return;
        }

        /* Attempt to lock the semaphore set */
        if(!getval(sid, member))
        {
                fprintf(stderr, "Semaphore resources exhausted (no lock)!\n");
                exit(1);
        }

        sem_lock.sem_num = member;

        if((semop(sid, &sem_lock, 1)) == -1)
        {
                fprintf(stderr, "Lock failed\n");
                exit(1);
        }
        else
                printf("Semaphore resources decremented by one (locked)\n");

        dispval(sid, member);
}

void unlocksem(int sid, int member)
{
 struct sembuf sem_unlock={ member, 1, IPC_NOWAIT};
 int semval;

 if( member<0 || member>(get_member_count(sid)-1))
 {
        fprintf(stderr, "semaphore member %d out of range\n", member);
        return;
 }

 /* Is the semaphore set locked? */
```

```
        semval = getval(sid, member);
        if(semval == SEM_RESOURCE_MAX) {
                fprintf(stderr, "Semaphore not locked!\n");
                exit(1);
        }

        sem_unlock.sem_num = member;

        /* Attempt to lock the semaphore set */
        if((semop(sid, &sem_unlock, 1)) == -1)
        {
                fprintf(stderr, "Unlock failed\n");
                exit(1);
        }
        else
                printf("Semaphore resources incremented by one (unlocked)\n");

        dispval(sid, member);
}

void removesem(int sid)
{
        semctl(sid, 0, IPC_RMID, 0);
        printf("Semaphore removed\n");
}

unsigned short get_member_count(int sid)
{
        union semun semopts;
        struct semid_ds mysemds;

        semopts.buf = &mysemds;

        /* Return number of members in the semaphore set */
        return(semopts.buf->sem_nsems);
}
int getval(int sid, int member)
{
        int semval;

        semval = semctl(sid, member, GETVAL, 0);
        return(semval);
}

void changemode(int sid, char *mode)
{
        int rc;
        union semun semopts;
        struct semid_ds mysemds;

        /* Get current values for internal data structure */
```

```
        semopts.buf = &mysemds;

        rc = semctl(sid, 0, IPC_STAT, semopts);

        if (rc == -1) {
                perror("semctl");
                exit(1);
        }

        printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);

        /* Change the permissions on the semaphore */
        sscanf(mode, "%ho", &semopts.buf->sem_perm.mode);

        /* Update the internal data structure */
        semctl(sid, 0, IPC_SET, semopts);

        printf("Updated...\n");

}

void dispval(int sid, int member)
{
        int semval;

        semval = semctl(sid, member, GETVAL, 0);
        printf("semval for member %d is %d\n", member, semval);
}

void usage(void)
{
        fprintf(stderr, "semtool - A utility for tinkering with semaphores\n");
        fprintf(stderr, "\nUSAGE:  semtool4 (c)reate <semcount>\n");
        fprintf(stderr, "                (l)ock <sem #>\n");
        fprintf(stderr, "                (u)nlock <sem #>\n");
        fprintf(stderr, "                (d)elete\n");
        fprintf(stderr, "                (m)ode <mode>\n");
        exit(1);
}
```

5. semstat：semtool的伴随程序

作为一个附赠品，下面给出一个称为 semstat的伴随程序的源代码。 semstat程序显示出集合中每个信号量的值，该集合是由 semtool所创建的。

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: semstat.c
 *************************************************************************
 A companion command line tool for the semtool package.  semstat displays
```

```
 the current value of all semaphores in the set created by semtool.
 ****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int get_sem_count(int sid);
void show_sem_usage(int sid);
int get_sem_count(int sid);
void dispval(int sid);

int main(int argc, char *argv[])
{
        key_t key;
        int   semset_id;

        /* Create unique key via call to ftok() */
        key = ftok(".", 's');

        /* Open the semaphore set - do not create! */
        if((semset_id = semget(key, 1, 0666)) == -1)
        {
                printf("Semaphore set does not exist\n");
                exit(1);
        }

        show_sem_usage(semset_id);
        return(0);
}

void show_sem_usage(int sid)
{
        int cntr=0, maxsems, semval;

        maxsems = get_sem_count(sid);

        while(cntr < maxsems) {
                semval = semctl(sid, cntr, GETVAL, 0);
                printf("Semaphore #%d:  --> %d\n", cntr, semval);
                cntr++;
        }
}

int get_sem_count(int sid)
{
        int rc;
        struct semid_ds mysemds;
```

```
        union semun semopts;

        /* Get current values for internal data structure */
        semopts.buf = &mysemds;

        if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1) {
                perror("semctl");
                exit(1);
        }

        /* return number of semaphores in set */
        return(semopts.buf->sem_nsems);
}

void dispval(int sid)
{
        int semval;

        semval = semctl(sid, 0, GETVAL, 0);
        printf("semval is %d\n", semval);
}
```

6.4.5 共享内存

1. 基本概念

共享内存的最佳定义是：它是对将要在多个进程间映射和共享的内存区域 (段)所作的映射。这是IPC最快捷的方式，因为这里没有什么中介 (例如管道、消息队列等等)。相反，消息直接从某内存段映射，并映射到调用进程的寻址空间。内存段可被某进程创建，并接着写到任意数量的进程中去，或者是从任意数量的进程中读数据。

2. 内部和用户数据结构

让我们简单地了解一下内核为共享内存段所维护的数据结构。

3. 内核shmid_ds结构

和消息队列以及信号量集合一样，对于当前存在于内核的寻址空间中的每个共享内存段，内核均为其维护着一个特殊的内部数据结构。这个结构的类型是 shmid_ds，它是在linux/shm.h中定义的，如下所示：

```
        /* One shmid data structure for each shared memory segment in the
system. */
        struct shmid_ds {
                struct ipc_perm shm_perm;       /* operation perms */
                int     shm_segsz;              /* size of segment (bytes) */
                time_t  shm_atime;              /* last attach time */
                time_t  shm_dtime;              /* last detach time */
                time_t  shm_ctime;              /* last change time */
                unsigned short  shm_cpid;       /* pid of creator */
                unsigned short  shm_lpid;       /* pid of last operator */
                short   shm_nattch;             /* no. of current attaches */
```

```
                                                    /* the following are private
    */

                    unsigned short   shm_npages;     /* size of segment (pages) */
                    unsigned long   *shm_pages;      /* array of ptrs to frames ->
    SHMMAX */
                    struct vm_area_struct *attaches; /* descriptors for attaches */
            };
```

在这个结构上所进行的操作是由一个特殊的系统调用执行的，并且还不能直接对它进行操作，下面是对一些更重要的域的描述：

shm_perm

它是ipc_perm结构的一个实例，它是在 linux/ipc.h中定义的，该域存放着该内存段的许可权限信息，包括访问许可权限，以及有关内存段的创建者的信息 (uid等等)。

shm_segsz

内存段的大小(以字节为单位测量)。

shm_atime

最后一次进程连接到该内存段的时间。

shm_dtime

最后一个进程与该内存段断开连接的时间。

shm_ctime

对这个内存段进行最后一次修改 (改变模式等等)的时间。

shm_cpid

创建进程的PID。

shm_lpid

对该内存段进行最后一次操作的进程的 PID。

shm_nattch

当前与该内存段相连接的进程的数量。

4. 系统调用：shmget ( )

为了创建一个新的共享内存段，或者访问一个现有的共享内存段，可以使用 shmget ( )系统调用。

```
  SYSTEM CALL: shmget();

  PROTOTYPE: int shmget ( key_t key, int size, int shmflg );
    RETURNS: shared memory segment identifier on success
             -1 on error: errno = EINVAL (Invalid segment size specified)
                                  EEXIST (Segment exists, cannot create)
                                  EIDRM (Segment is marked for deletion, or was
removed)
                                  ENOENT (Segment does not exist)
                                  EACCES (Permission denied)
                                  ENOMEM (Not enough memory to create segment)

  NOTES:
```

这个系统调用对我们来说早已不陌生了，它与消息队列以及信号量集合对应的调用惊人

的相似。

shmget的第一个变元是关键字的值(在我们的例子中它是由调用 ftok ( )所返回的值)。然后，这个值将与内核中现有的其他共享内存段的关键字值相比较。在比较之后，打开和访问操作都将依赖于shmflg变元的内容。

IPC_CREAT

如果在内核中不存在该内存段，则创建它。

IPC_EXCL

当与IPC_CREAT一起使用时，如果该内存段早已存在，则此次调用将失败。

如果只使用IPC_CREAT，shmget( )或者将返回新创建的内存段的段标识符，或者返回早已存在于内核中的具有相同关键字值的内存段的标识符。如果同时使用 IPC_CREAT和IPC_EXCL，则可能会有两种结果，如果该内存段不存在，则将创建一个新的内存段；如果内存段早已存在，则此次调用失败，并将返回－ 1。IPC_EXCL本身是没有什么用处的，但在与IPC_CREAT组合使用时，它可用于保证没有一个现有的内存段为了访问而打开着。

再说一遍，用户可以把许可方式与掩码进行 OR操作，得到一个可选的八进制形式的许可模式。

下面我们创建一个包装程序函数，用于查找或者创建一个共享内存段：

```
int open_segment( key_t keyval, int segsize )
{
        int     shmid;

        if((shmid = shmget( keyval, segsize, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(shmid);
}
```

注意，程序中使用了显式的许可权限 0660。这个小函数或者将返回一个共享内存段标识符(整型)，或者在出错时返回－ 1。关键字的值和所申请的段的大小 (以字节为单位)都是作为变元而传递给该函数的。

一旦进程获得了给定内存段的合法 IPC标识符，它的下一步操作就是连接该内存段，或者把该内存段映射到自己的寻址空间中。

5. 系统调用：shmat ( )

```
  SYSTEM CALL: shmat();

  PROTOTYPE: int shmat ( int shmid, char *shmaddr, int shmflg);
    RETURNS: address at which segment was attached to the process, or
             -1 on error: errno = EINVAL (Invalid IPC ID value or attach
address passed)
                                  ENOMEM (Not enough memory to attach segment)
                                EACCES (Permission denied)
 NOTES:
```

如果addr变元值等于零(0)，则内核将试着查找一个未映射的区域。这是我们推荐使用的方法。用户可以指定一个地址，但通常该地址只用于访问所拥有的硬件，或者解决与其他应用程序的冲突。SHM_RND标志可以与标志变元进行OR操作，结果再置为标志变元，这样可以让传送的地址页对齐(舍入到最相近的页面大小)。

此外，如果把SHM_RDONLY标志与标志变元进行OR操作，结果再置为标志变元，这样映射的共享内存段只能标记为只读方式。

这个调用使用起来可能是最简单的了。请看下面这个包装程序函数，它的一个参数是某内存段的合法IPC标识符，它将返回那个内存段所连接的地址：

```
char *attach_segment( int shmid )
{
        return(shmat(shmid, 0, 0));
}
```

一旦正确地连接上了一个内存段，并且进程拥有一个指向该内存段始地址的指针，则对那个内存段进行读和写就象引用和间接引用指针一样简单！请注意不要丢了原始指针的值！如果把它丢了，则用户将没有办法访问该内存段的基址 (开始)。

6. 系统调用：shmctl( )

```
  SYSTEM CALL: shmctl();
  PROTOTYPE: int shmctl ( int shmqid, int cmd, struct shmid_ds *buf );
    RETURNS: 0 on success
             -1 on error: errno = EACCES (No read permission and cmd is
IPC_STAT)
                                  EFAULT (Address pointed to by buf is invalid
with IPC_SET and
                                          IPC_STAT commands)
                                  EIDRM  (Segment was removed during retrieval)
                                  EINVAL (shmqid invalid)
                                  EPERM  (IPC_SET or IPC_RMID command was
issued, but
                                          calling process does not have write
(alter)
                                          access to the segment)
       NOTES:
```

这个调用与消息队列的msgctl调用是完全类似的。正因为如此，我们将不会过于详细地去介绍它，合法的命令值是：

IPC_STAT

获取内存段的shmid_ds结构，并把它存储在buf变元所指定的地址中。

IPC_SET

设置内存段shmid_ds结构的ipc_perm成员的值，此命令是从buf变元中获得该值的。

IPC_RMID

标记某内存段，以备删除。

IPC_RMID命令并不真正地把内存段从内存中删除。相反，它只是标记上该内存段，以备将来删除。只有当前连接到该内存段的最后一个进程正确地断开了与它的连接，实际的删除

操作才会发生。当然，如果当前没有进程与该内存段相连接，则删除将立刻发生。

为了正确地断开与其共享内存段的连接，进程需要调用 shmdt系统调用。

7. 系统调用：shmdt ( )

```
SYSTEM CALL: shmdt();

PROTOTYPE: int shmdt ( char *shmaddr );
  RETURNS: -1 on error: errno = EINVAL (Invalid attach address passed)
```

当某进程不再需要一个共享内存段时，它必须调用这个系统调用来断开与该内存段的连接。正如前面所介绍的那样，这与从内核删除内存段是两回事！在成功完成了断开连接操作以后，相关的shmid_ds结构的shm_nattch成员的值将减去一。如果这个值减到零 (0)，则内核将真正删除该内存段。

8. shmtool：交互式共享内存操作程序

背景知识

我们最后一个关于系统V IPC对象的例子将是shmtool，它是一个命令行工具，用于创建、读、写和删除共享内存段。再说一遍，与前面的例子一样，在任何操作中，如果内存段事先不存在，则它将被创建出来。

9. 命令行语法

向内存段写字符串

```
shmtool w "text"
```

从内存段获取字符串

```
shmtool r
```

改变许可权限(模式)

```
shmtool m (mode)
```

删除内存段

```
shmtool d
```

10. 实例

```
shmtool  w   test
shmtool  w   "This is a test"
shmtool  r
shmtool  d
shmtool  m   660
```

11. 源代码

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SEGSIZE 100

main(int argc, char *argv[])
{
        key_t key;
```

```
    int   shmid, cntr;
    char  *segptr;

    if(argc == 1)
            usage();

    /* Create unique key via call to ftok() */
    key = ftok(".", 'S');

    /* Open the shared memory segment - create if necessary */
    if((shmid = shmget(key, SEGSIZE, IPC_CREAT|IPC_EXCL|0666)) == -1)
        {
                printf("Shared memory segment exists - opening as client\n");

                /* Segment probably already exists - try as a client */
                if((shmid = shmget(key, SEGSIZE, 0)) == -1)
                {
                        perror("shmget");
                        exit(1);
                }
        }
        else
        {
                printf("Creating new shared memory segment\n");
        }

        /* Attach (map) the shared memory segment into the current process */
        if((segptr = shmat(shmid, 0, 0)) == -1)
        {
                perror("shmat");
                exit(1);
        }

        switch(tolower(argv[1][0]))
        {
                case 'w': writeshm(shmid, segptr, argv[2]);
                          break;
                case 'r': readshm(shmid, segptr);
                          break;
                case 'd': removeshm(shmid);
                          break;
                case 'm': changemode(shmid, argv[2]);
                          break;
                 default: usage();

        }
}

writeshm(int shmid, char *segptr, char *text)
{
```

```
        strcpy(segptr, text);
        printf("Done...\n");
}

readshm(int shmid, char *segptr)
{
        printf("segptr: %s\n", segptr);
}

removeshm(int shmid)
{
        shmctl(shmid, IPC_RMID, 0);
        printf("Shared memory segment marked for deletion\n");
}

changemode(int shmid, char *mode)
{
        struct shmid_ds myshmds;

        /* Get current values for internal data structure */
        shmctl(shmid, IPC_STAT, &myshmds);

        /* Display old permissions */
        printf("Old permissions were: %o\n", myshmds.shm_perm.mode);

        /* Convert and load the mode */
        sscanf(mode, "%o", &myshmds.shm_perm.mode);

        /* Update the mode */
        shmctl(shmid, IPC_SET, &myshmds);

        printf("New permissions are : %o\n", myshmds.shm_perm.mode);
}

usage()
{
        fprintf(stderr, "shmtool - A utility for tinkering with shared
memory\n");
        fprintf(stderr, "\nUSAGE:  shmtool (w)rite <text>\n");
        fprintf(stderr, "                (r)ead\n");
        fprintf(stderr, "                (d)elete\n");
        fprintf(stderr, "                (m)ode change <octal mode>\n");
        exit(1);
}
```

第7章 声音编程

一台PC机至少有一个声音设备：内部扬声器。但是，用户还可以买一块声卡插入自己的计算机，以提供更复杂的声音设备。读者可以查看《 Linux Sound User ’s Guide》或者《Sound HOWTO》来阅读有关声卡的内容。

7.1 内部扬声器编程

不管读者相信还是不相信， PC机上的扬声器是Linux控制台的一部分，因此它是个字符设备。所以，可以使用ioctl ( )请求来操作它。对内部扬声器而言，存在下面两个请求：

1. KDMKTONE

使用内核定时器产生一段特定时间长的哔哔声。

例子：ioctl (fd, KDMKTONE, (long) argument).

2. KIOCSOUND

产生一个持续的哔哔声，或者中止当前正在发出的哔哔声。

例子：ioctl (fd, KIOCSOUND, (int) tone).

变元由两部分组成，低位字是声调 (tone)的值，而高位字是声音持续的时间周期。 tone的值并不是指声音频率。 PC主板定时器8254的时钟频率是1.19MHz，所以它是1190000/秒。周期长度是以时钟滴答的次数为单位测量的。上面这两个 ioctl调用将立即返回，所以用户可以使用这种方法产生哔哔声，而无需阻塞程序。

KDMKTONE应该用来产生警告信息，因为用户无需考虑将这种声音停下来。

KIOCSOUND可以用来演奏悦耳的音乐，就像在例子程序 splay中一样。为了停下它的声音，用户可以使用为0的tone值来调用KIOCSOUND。

7.2 声卡编程

作为一个程序员，了解当前 Linux系统中是否插了一块声卡是很重要的。有一个检查的方法是查看/dev/sndstat。如果打开/dev/sndstat失败，并且errno等于ENODEV，那么说明没有激活声音驱动程序，这意味着用户无法得到内核声音驱动程序的帮助。除此之外，用户还可以试着去打开 /dev/dsp，也可以达到查看声卡的目的。如果它没有连接到 pcsnd驱动程序，则open ( )操作将不会失败。

如果用户希望在硬件级别上使用声卡，则通过使用 outb ( )和inb ( )调用的一些组合，将可以检测到用户正在查找的声卡。

用户在应用程序中使用声音驱动程序，可能会造成这些应用程序在其他 i386系统上也能正常运行。这是因为聪明的程序员们已经决定在 Linux、isc、FreeBSD以及大部分其他基于i386的系统上使用相同的驱动程序。如果 Linux是在提供同一个声音设备接口的其他体系结构下运行，则这个特征在移植程序时会大有帮助。声卡不是 Linux控制台的一部分，它是一个特殊的设备。声卡主要提供三个重要的特征：

- 数字取样输入/输出
- 频率调制输出
- MIDI接口

这三个特征都有它们自己的设备驱动程序接口。数字取样的接口是 /dev/dsp。频率调制的接口/dev/sequencer，而MIDI接口是/dev/midi。声音设备(如音量、平衡或者贝斯)可以通过/dev/mixer接口来控制。为了兼容性的需要，还提供了一个 /dev/audio设备，该设备可用于读SUN_law的声音数据，但它是映射到数字取样设备的。

如果读者猜想自己可以使用ioctl ( )来操作这些设备，那么读者猜对了。 ioctl ( )请求是在<linux/soundcard.h>中定义的，它们以SNDCTL_开头。

因为我自己并没有声卡，所以希望有志之士接着完善本章。

第8章　字符单元图形

本章讨论的是基于字符的屏幕输入和输出，而不是基于像素的屏幕输入和输出。这里所提到的字符，是指像素的一种组合，它可以根据字符集而变化。用户的图形卡早已提供了一种或多种的字符集，默认是以文本(字符集)模式进行操作的，这是因为文本的处理速度比像素图形的处理速度要快得多。对终端来说，用户除了可以把它用作简单的 (沉默的)并且是烦人的文本显示器以外，还可以用它们来完成很多任务。下面将介绍怎样利用　Linux终端尤其是Linux控制台所提供的特殊功能。

- printf、sprintf、fprintf、scanf、sscanf、fscanf —— 使用Linux的这些函数，用户可以把格式化的字符串输出到标准输出和标准错误中，或者输出到其他定义为 FILE *stream形式的流中(例如文件)。Scanf(……)提供了一个类似的方法，可以从 Linux读取格式化的输入。
- termcap —— 在ASCII文件/etc/termcap中，有一个终端描述项目的集合，它就是终端功能数据库(TERMinal CAPabilities database)。在这里用户可以找到一些信息，介绍如何显示特殊字符，如何执行操作(删除、插入字符或者行，等等)，以及怎样初始化一个终端。这个数据库可以通过编辑器vi来使用。系统包含一些视图库函数，可以读和使用终端功能(termcap (3x))。使用这个数据库，程序就可以用相同的代码处理各种终端了。termcap数据库和库函数只提供了对终端的低级访问。程序员必须亲自完成诸如改变属性或颜色、参数化输出以及优化等工作。
- terminfo数据库 —— 终端信息数据库(TERMinal INFOrmation database)是基于termcap数据库的，它也介绍了终端的功能。但是所处的级别要比 termcap高。使用terminfo数据库，程序可以轻易地改变屏幕的属性，并且可以使用特殊的键，如功能键，等等。该数据库位于/usr/lib/terminfo/[A-z, 0-9]*中。每个文件描述一个终端。
- curses —— Terminfo是个不错的数据库，非常适合在程序中用来进行终端处理。而(BSD_)CURSES库则使用户可以对终端进行高级访问，它是基于 terminfo数据库的。Curses允许用户在屏幕上打开并操作一个窗口，它提供了一个完整的输入和输出函数的集合，并可以在150多个终端上用不依赖于终端的方式改变视频的属性。 curses库可以在/usr/lib/libcurses.a中找到。这是一个BSD版本的curses库。
- ncurses —— Ncurses是一个更高级的版本。在版本1.8.6中，它应该与定义在SYSVR4中的AT&T的curses库兼容。它具有一些扩展的功能，如颜色操作、输出的特殊优化、终端特定的优化，等等。该库已经在大量的系统上测试通过，如 Sun OS、HP和Linux。笔者建议用户使用ncurses，而不要使用其他库。在SYSV Unix系统上(如Sun的Solaris)，应该存在一个curses库，它的功能与ncurses相同(实际上solaris的curses提供的函数更多，并提供鼠标支持)。

下面几节将介绍怎样使用不同的服务包来访问终端。在　Linux下，用户拥有 termcap的GNU版本，用户可以使用ncurses，而不是curses。

8.1 libc中的I/O函数

8.1.1 格式化输出

printf(……)函数的功能是提供格式化的输出，并允许变元的转换。

```
int fprintf(FILE *stream, const char *format, ...),
```

上面这个函数将对输出 (在……中填写的变元)转换，并把它写到 stream中，同时还将把format中定义的格式也写到stream中。该函数将会返回实际写的字符的个数；如果出错则返回一个负数。

format包含两种类型的对象：

1. 需要输出的普通字符。
2. 关于如何转换或者格式化变元的信息。

格式信息必须以%打开，后面是该格式的值，再后面是需要转换的字符 (如果要打印%自身，需要使用%%)。格式的可能值如下：

标志

－

格式化的变元在域内向左靠输出 (默认的方式是在变元域中向右靠输出)。

＋

每个数打印时都要带上符号，例如 +12或者－2.32

Blank

当第一个字符不是符号时，需要插入一个空格

0

对于数字转化而言，如果域的宽度定义得超过数字的位数，则在数字的左边用 0来填充。

#

根据变元转换的不同，输出也将不同

- 0 第一个数字为0。
- x或X 在变元的前面会打印0x或0X。
- e、E、f或F 输出将带有小数点。
- g或G 变元尾部的0会打印出来。

```
A number for the minimal field width.
```

转换以后的变元打印的宽度至少应该与变元本身一样大。通过在格式中指定一个数值，用户就可以让字段宽度稍微大一点。如果格式化的变元比较小，则字段的宽度将用 0或者空格来填充。

```
A point to separate the field width and the precision.
A number for the precision.
```

这个转换的可能值如表8-1所示。

```
int printf(const char *format, ...)
```

这个函数与fprintf (stdout, ……)相同。

```
int sprintf(char *s, const char *format, ...)
```

这个函数与printf (…)相同，只有一点区别，即输出将被写到字符指针 S中(带一个后缀\0)。

用户必须为S分配足够的内存空间。

```
vprintf(const char *format, va_list arg)
vfprintf(FILE *stream, const char *format, va_list arg)
vsprintf(char *s, const char *format, va_list arg)
```

这三个函数与前面的函数相同，不过变元列表被设置为 arg。

表8-1 libc-printf转换

注意 出版者注：作者尚未提供此表。

8.1.2 格式化输入

就像在格式化输出中使用printf (…)一样，用户可以把scanf (…)用于格式化输入。

```
int fscanf(FILE *stream, const char *format, ...)
```

fscanf (…)从stream中读数据，并按照在format中定义的规则对其进行转换。转换结果将放置在变元……中(**注意**：所有变元**必须**是指针)。当format中的转换规则已经用完时，读操作即告结束。如果第一个转换遇到文件尾，或者发生了某些错误，则 fscanf (…)将返回EOF，否则它将返回转换以后变元的数目。

format可以存放有关如何格式化输入变元的规则 (见下表8-2)。

它还可以包括：

- 空格或者制表符，这两者均被忽略。
- 任意的普通字符(除了%以外)，这些字符必须出现在输入的对应位置上。
- 转换规则，它的构成如下：一个 %号，可选的符号*(这个符号使fscanf (…)可以赋予一个变元)、一个可选的数字、一个可选的符号 h、l或L(它规定了输入目标的长度)以及转换字符。

```
int scanf(const char *format, ...)
```

该函数与fscanf (stdin, …)类似。

```
int sscanf(char *str, const char *format, ...)
```

该函数与scanf(…)相同，除了它的输出是来自 str以外。

表8-2 libc-scanf转换

注意 出版者注：作者尚未提供此表。

8.2 termcap库

8.2.1 前言

termcap库是termcap数据库所提供的API，该库可以在/etc/termcap/中找到。库函数允许进行如下动作：

- 获得当前终端的描述：tgetent (…)。
- 在描述中查找信息：tgetnum (…)、tgetflag (…)、tgetstr (…)。
- 以终端特定的格式编码数字参数：tparam (…)、tgoto (…)。
- 计算并执行填充：tputs (…)。

如果程序中用到termcap库，则它必须包含.h头文件，并与这些头文件相链接。

注意 出版者注：作者尚未提供表

termcap函数是一些独立于终端的过程，但它们只允许程序员对终端进行低级访问。如果需要稍高级别的服务包，应该使用curses或者ncurses。

8.2.2 获得终端描述

```
int tgetent(void *buffer, const char *termtype)
```

在Linux操作系统上，当前终端的名称包含在环境变量 TERM中。所以termtype是调用(3)的返回结果。

对buffer而言，在使用termcap的GNU版本时无需分配任何内存，这一点实际上根据我们对Linux的认识也应该想到了。否则，如果使用的不是 GNU版本的termcap，用户就必须分配2,048个字节。(以前缓冲区只需要1,024个字节，但是现在这个大小已经翻倍了。)

tgetent (…)在成功时将返回1；而如果找到数据库却没有对应TERM的项目，则返回0；如果出错则返回其他不同的值。

下面的例子说明了tgetent (…)的用法：

注意 出版者注：作者尚未提供任何例子。

默认情况下，termcap使用/etc/termcap/作数据库。如果环境变量TERMCAP被设置成其他的值，例如 $HOME/mytermcap，则所有的函数将不再使用 /etc/termcap，而使用$HOME/mytermcap。TERMCAP前面没有斜线，这个定义好的值将用作终端的名称。

8.2.3 查看终端描述

每一段信息都被称为一个权能(capability)，每个权能都是两个字母的代码，这两个字母代码的后面是该权能的值。权能的类型包括：

- 数值型：例如co —— 列的数目。
- 布尔型或标志：例如hc —— 硬拷贝终端。
- 字符串：例如st —— 设置制表符停止。

每个权能都只有一种值的类型(co永远是数值型、hc永远是标志，而st永远是字符串)。因为存在三种不同类型的值，所以相应地需要用三个函数来查询它们。 char *name是权能的两个字母的代码。

```
int tgetnum(char *name)
```

它获得类型为数值型的权能的值，例如 co的值。如果权能是可用的， tgetnum (…)将返回该数值，否则返回1。(注意，返回值非负。)

```
int tgetflag(char *name)
```

它获得一个布尔型的权能值(或者标志型)。如果该标志存在，则返回1，否则返回0。

```
char *tgetstr(char *name, char **area)
```

它获得一个字符串型的权能值。如果该值存在，则返回一个指向该字符串的指针；否则，如果不存在则返回 NULL。在GNU版本中，如果 area为NULL，则termcap将自己分配内存。termcap将不会再引用该指针。所以在离开程序之前别忘了释放 name。我们提倡使用这个方法，因为用户不知道指针将需要多少空间，所以应该让 termcap帮用户完成分配工作。

8.2.4 termcap权能

布尔型权能	
5i	打印机将不回送到屏幕上
am	自动对齐，这意味着自动换行
bs	Control+H (8 dec.)执行一个后退操作
bw	在最左一列作后退操作将转换到前一行的最右边
da	显示保留在上面的屏幕
db	显示保留在下面的屏幕
eo	按空格将会删除光标位置的所有字符
es	在状态行中可以使用Esc组合键和特殊字符
gn	普通设备
hc	这是个硬拷贝终端
HC	当光标不在最底下一行时将难以看到它
hs	有一个状态行
hz	Hazeltine错误，终端不能打印发音符号
in	终端在空白区域插入null，而不是插入空格
km	终端有一个元键
mi	光标移动是以插入方式工作的
ms	光标移动是以standout/underline方式工作的
NP	没有填充字符
NR	ti不会返转te
nx	不有填充，但必须使用XON/XOFF
os	终端可能会叠印
ul	尽管终端不能叠印，但它可以划下划线
xb	Beehive信号，F1发送ESCAPE，F2发送^c
xn	新行/环绕信号
xo	终端使用XON/XOFF协议
xs	在标准输出文本上打印的文本将显示在标准输出中
xt	Teleray信号，破坏性制表符，以及奇数standout方式

数值型权能

字符串权能

8.3 Ncurses简介

本章将使用下述的术语：

- 窗口：这是一个内部表述，它包含屏幕某部分的映像。 WINDOW是在curses.h中定义的。
- 屏幕：它是一个窗口，它的大小就是整个屏幕的大小 (从左上角到右下角)。

• 终端：它是一个特殊的屏幕，它包含着当前屏幕是什么样的有关信息。
• 变量：在curses.h中定义了如下一些变量和常量：

WINDDW *curscr：	当前屏幕
WINDOW *stdscr：	标准屏幕
int LINES：	终端上的行数
int COLS：	终端上的列数
bool TRUE：	真标志，1
bool FALSE：	假标志，0
int ERR：	错误标志，－1
int OK：	OK标志，0

• 函数：在函数的描述中，变元一般具有如下类型：

win：	WINDOW*
bf：	bool
ch：	chtype
str：	char*
chstr：	chtype*
fmt：	char*
否则：	int

一般来说，使用ncurses库的程序看起来像下面一样：

注意 出版者注：作者尚未提供例子程序。

ncurses.h中定义了ncurses的变量和类型，例如WINDOW和函数原型，所以应该包含这个头文件。它将会自动包含stdio.h、ncurses/unctrl.h、stdarg.h和stddef.h。

initscr ( )可以用来初始化ncurses数据结构，并可用于读入正确的terminfo文件。内存也是在这个函数中分配的。如果发生了错误，则 initscr将返回ERR，否则将返回一个指针。此外，屏幕将被刷新并被初始化。

endwin ( )将清除所有已经分配的ncurses资源，并把tty方式恢复到在调用initstr ( )以前的状态。在调用ncurses库的任意其他函数之前，必须先调用 initscr ( )；而在用户退出程序之前，必须调用endwin ( )。如果用户希望在多个终端上执行输出，则用户可以使用 newterm (…)，而不要使用initscr ( )。

可以用下面的方法编译该程序：

注意 出版者注：原书缺少正文。

用户可以加入自己愿意加入的任何标志 (gcc (1))，因为ncurses.h的路径已经改变了，所以用户必须加入下面一行：

注意 出版者注：原书缺少正文。

否则将无法找到ncurses.h、nterm.h、termcap.h以及unctrl.h。Linux可能的其他标志为：
• 2：让gcc做一些优化工作。
• _ansi：用于ansi一致性C代码。
• _Wall：将打印出所有的警告信息。
• _m486：将对Intel 486使用优化的代码(该二进制代码也可用在Intel 386上)。

ncurses库可在/usr/lib/中找到。ncurses库总共有三种版本：

• libncurses.a —— 普通的ncurses库。

• libdcurses.a —— 用于调试的ncurses库。

• libpcurses.a —— 用于配置的ncurses库(不知是否因为1.8.6 libpcurses.a不再存在的原因)。

• libcurses.a —— 实际上不是ncurses库的第四个版本，它是原始的 BSD curses(在笔者的Slackware 2.1.0中，它是bsd服务包)。

屏幕的数据结构称为窗口，它是在 ncurses.h中定义的。在某种程度上，窗口就像内存中的一个字符数组，程序员可以对其进行操作，无需输出到终端。默认的窗口大小就是终端的大小，用户可以用newwin (…)创建其他的窗口。

为了可以更好地更新物理终端， ncurses声明了另一个窗口。这是一个关于终端实际是什么样子的映像，并且它还是一个关于终端应该是什么样子的映像。输出必须在用户调用 refresh ( )时进行。然后Ncurses就可以使用Linux中的信息来更新物理终端了。库函数在更新进程中将使用内部优化，这样用户就可以改变不同的窗口，并立即以最优的方式更新屏幕。

通过使用ncurses函数，用户可以对数据结构窗口进行操作。以 w开头的函数将允许用户指定窗口；而其他函数一般用于影响 Linux。以mv开头的函数将首先把光标移动到位置 y, x上。

有一个字符类型是 chtype，这是一种无符号长整型，它可用于存放一些附加的信息 (属性等等)。

Ncurses使用数据库。一般来说该数据库位于 /lib/terminfo/，ncurses将在那里查找本地终端定义。如果用户希望在不改变原始的 terminfo的情况下测试终端的一些定义，用户可以设置环境变量。ncurses将检查这个变量，并使用存放在那里的定义，而不使用 /usr/lib/terminfo/中的定义。

当前ncurses版本是1.8.6 ( )。

注意 出版者注：当前版本是4.2, www.gnu.ai.mit.edu/software/ncurses/ncurses.html。

在本章的末尾，读者可以读到一个表，它总结了 BSD_Curses、ncurses以及Sun Os 5.4的curses。如果读者希望查找某个特定的函数以及它是在哪里定义的，则可以查找那个表。

8.4 初始化

• WINDOW *initscr ( ) —— 在使用ncurses的程序中，一般首先应该调用这个函数。在有些情况下，在调用initscr ( )以前，可能会需要调用slk_init (int)、filter ( )、ripoffline (…)或者use_env (bf)。当使用多个终端时 (或者可能测试权能时)，用户可以使用 newterm (…)来取代initscr ( )。

initscr ( )将读入正确的terminfo文件，初始化ncurses数据结构，并为其分配内存，将其设置为终端所具有的值。该函数将返回一个指针；而如果出错则返回 ERR。用户无需初始化该指针。

initscr ( )将为用户完成初始化工作。如果它的返回值是 ERR，则用户的程序将退出。这是因为没有ncurses函数可以正常工作。

• SCREEN *newterm (char *type, FILE *outfd, FILE *intd)

如果具有多个终端输出，则对用户访问的每个终端都应该调用 newterm (…)，而不用initscr ( )。type是终端的名称，包括在$TERM中(如ansi, xterm, vt100等等)。outfd是输出指针，

而infd是输入指针。对使用newterm (…)打开的每个终端都应该调用endwin ( )来退出。

• SCREEN *set_term (SCREEN *new)

通过使用 set_term(SCREEN)，用户可以切换当前终端。所有的函数都将在set_term(SCREEN)设置的当前终端上起作用。

• int endwin ( )

endwin ( )将执行清理工作，把终端模式恢复到调用 initscr ( )以前的状态，并把光标移动到屏幕的左上角。在调用endwin ( )退出程序之前，不要忘了关闭所有打开的窗口。

在调用endwin ( )之后还可以调用refresh ( )，它将把终端恢复到调用initscr ( )以前的状态(可视模式)，否则屏幕将被清除(不可视模式)。

• int isendwin ( )

如果调用endwin ( )之后还调用了refresh ( )，则该函数返回TRUE，否则返回FALSE。

• void delscreen (SCREEN *sp)

当SCREEN不再需要时，在调用完endwin ( )之后，调用delscreen (SCREEN)将可以释放所有占用的资源。(注，此函数尚未实现。)

8.5 窗口

窗口可以创建、删除、移动、拷贝、按掀、复制等等。

• WINDOW *newwin (nlines, ncols, begy, begx)

begy和begx是窗口左上角的坐标。nlines是一个整数，它存放着行的数目，而ncols也是一个整数，它存放着列的数目。

图3-8-1 Ncurses-newwin方案

注意 出版者注：原书缺图。

该窗口的左上角位于第10行，第10列，该窗口有10行，60列。如果nlines为零，则窗口将有LINES_begy行。同理可知，如果ncols为零，则窗口将有COLS_begx列。

如果用户调用newwin且把所有的参数设置为零，则打开的窗口的大小将与屏幕的大小相同。

使用……(原书缺)我们可以在屏幕的中间打开一个窗口，不管它的尺寸有多大：

(原书缺)

这命令将在屏幕的中间打开一个 22行和70列的窗口。在打开窗口以前应该查看屏幕的大小。在Linux控制台中，我们可以有 25行以上，80列以上，但是在xterm中情况却并不一定如此(它们是可以缩放的)。

此外，用户还可以使用(……)把两个窗口调整为屏幕大小：

在例子目录中有一些C程序，读者可以阅读这些程序，找到更多的解释。

• int delwin (win)

它删除窗口win。如果存在子窗口，则在删除 win以前先要删除这些子窗口。这个函数将释放win所占据的所有资源。在调用 endwin ( )之前用户应该删除所有的窗口。

• int mvwin (win, by, bx)

它将把窗口移到坐标(by，bx)处。如果这个坐标将把窗口移出屏幕边界的范围，则此函数

什么也不做，并返回ERR。

• WINDOW *subwin (origwin, nlines, ncols, begy, begx)

它返回一个位于origwin窗口中间的子窗口。如果用户改变这两个窗口 (origwin或者那个新窗口)中的一个，则这种改变将会同时反映到这两个窗口上。在下一次调用 refresh ( )之前，先要调用touchwin (origwin)。

begx和begy是相对于屏幕的，而不是相对于 origwin的。

• WINDOW *derwin (origwin, nlines, ncols, begy, begx)

此函数与subwin (…)相同，只不过这里的 begx和begy是相对于窗口origwin的，而不是相对于屏幕的。

• int mvderwin (win, y, x)

此函数将把win移到父窗口内。(注意：此函数尚未实现)。

• WINDOW *dupwin (win)

此函数复制窗口win。

• Duplicate the window win.
• `int syncok(win, bf)`
• `void wsyncup(win)`
• `void wcursyncup(win)`
• `void wsyncdown(win)` (注：尚未实现)
• `int overlay(win1, win2)`
• `int overwrite(win1, win2)`
• overlay (…)将把win1中的所有文本拷贝到win2中，但是不拷贝空格。 overwrite (…)也是做文本拷贝工作的函数，但它拷贝空格。
• int copywin (win1, win2, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)
• 它所做的工作与overlay (…)和overwrite (…)相似，但是该函数还可以让程序员选择拷贝窗口的哪个区域。

8.6 输出

• `int addch(ch)`
• `int waddch(win, ch)`
• `int mvaddch(y, x, ch)`
• `int mvwaddch(win, y, x, ch)`

这些函数可以用于把字符输出到窗口。它们将操作窗口，为了把字符写到屏幕上，用户必须调用函数 refresh ( )。addch (…)和waddch (…)把字符 ch输出到屏幕上或者 win中，mvaddch (…)和mvwaddch (…)执行相同的工作，但它们首先会把光标移动到位置 y, x上。

• `int addstr(str)`
• `int addnstr(str, n)`

- int waddstr(win, str)
- int waddnstr(win, str, n)
- int mvaddstr(y, x, str)
- int mvaddnstr(y, x, str, n)
- int mvwaddstr(win, y, x, str)
- int mvwaddnstr(win, y, x, str, n)

这些函数把字符串输出到某个窗口，它们的作用等价于对 addch (…)进行一系列的调用。str是一个以null结束的字符串(“ blafoo\0 ”)。名称中有字母w的函数把字符串str写到窗口win中，而其他函数则把字符串输出到屏幕上。名称中有字母 n的函数将输出字符串中的n个字符。如果n的值为－1，则输出整个字符串str。

- int addchstr(chstr)
- int addchnstr(chstr, n)
- int waddchstr(win, chstr)
- int waddchnstr(win, chstr, n)
- int mvaddchstr(y, x, chstr)
- int mvaddchnstr(y, x, chstr, n)
- int mvwaddchstr(win, y, x, chstr)
- int mvwaddchnstr(win, y, x, chstr, n)

这些函数把chstr拷贝到窗口映射中(或者win中)。起始位置是当前光标所处的位置。名称中有字母n的函数将输出chstr中的n个字符。如果n的值为－1，则输出整个字符串 chstr。光标的位置不移动，并且不检查控制字符。这些函数比 addstr (…)那些过程的运行速度要快。 chstr是指向chtype数组的一个指针。

- int echochar(ch)
- int wechochar(win, ch)

这两个函数等价于先调用 addch (…) (waddch (…))，再接着调用 refresh ( ) (wrefresh (win))。

8.6.1 格式化输出

```
int printw(fmt, ...)

int wprintw(win, fmt, ...)

int mvprintw(y, x, fmt, ...)

int mvwprintw(win, y, x, fmt, ...)

int vwprintw(win, fmt, va_list)
```

这些函数对应于printf (…)以及Linux中的其他函数。

在服务包中printf (…)用于格式化输出。用户可以定义一个输出字符串，在其中包含不同类型的变量。

如果需要使用vwprintw (…)，用户也必须包含.h头文件。

8.6.2 插入字符/行

```
int insch(c)

int winsch(win, c)

int mvinsch(y,x,c)

int mvwinsch(win,y,x,c)
```

这些函数把字符ch插入到光标的左边，光标后面的所有字符则向右移动一个位置。在这一行最右端的字符可能会丢失。

```
int insertln()

int winsertln(win)
```

这两个函数在当前行的上方插入一个空行 (最底下的一行将被丢失)。

```
int insdelln(n)

int winsdelln(win, n)
```

如果n为正数，则这些函数将在适当的窗口的当前光标上方插入 n行(这样一来最底下的n行将丢失)；如果n为负数，则光标下面的n行将被删除，余下的行将上升，顶替它们的位置。

```
int insstr(str)

int insnstr(str, n)

int winsstr(win, str)

int winsnstr(win, str, n)

int mvinsstr(y, x, str)

int mvinsnstr(y, x, str, n)
int mvwinsstr(win, y, x, str)
int mvwinsnstr(win, y, x, str, n)
```

这些函数将在当前光标的左边插入str(字符的个数不能超过一行的限度)。在光标右边的字符将右移，如果到达行尾，则字符将丢失，光标位置不变。

y和x是指在插入str以前先要把光标移动到的坐标， n是要插入的字符的数目 (如果n为0则插入整个字符串)。

8.6.3 删除字符/行

```
int delch()
```

```
int wdelch(win)

int mvdelch(y, x)

int mvwdelch(win, y, x)
```

删除光标左边的字符，并把光标右边余下的字符向左移动一个位置。
y和x是在进行删除操作以前要把光标移动到的坐标。

```
int deleteln()

int wdeleteln(win)
```

删除光标下面的一行，并把下面所有的其他行都向上移动一个位置。此外，屏幕最底下的一行将被清除。

8.6.4 方框和直线

```
int border(ls, rs, ts, bs, tl, tr, bl, br)

int wborder(win, ls, rs, ts, bs, tl, tr, bl, br)

int box(win, vert, hor)
```

这些函数在窗口的边界 (或者win的边界)画上方框。在下面的表格中，读者将可以看到字符，以及它们的默认值。当用零去调用 box (…)时将会用到这些默认值。在下面的图中读者可以看到方框中字符的位置。

表8-3 Ncurses：边框字符

图3-8-2 Ncurses：方框字符

注意 出版者注：表和图原书缺。

```
int vline(ch, n)

int wvline(win, ch, n)

int hline(ch, n)

int whline(win, ch, n)
```

这些函数将从当前光标位置开始画一条水平线或者垂直线。 ch是画线所使用的字符， n是要画的字符的个数，光标位置并不移动。

8.6.5 背景字符

```
void bkgdset(ch)

void wbkgdset(win, ch)
```

这两个函数设置窗口或者屏幕的背景字符和属性。 ch的属性将和窗口中所有非空格的字

符的属性进行OR操作。背景是窗口的一部分，将不会随着滚动、输入或输出而改变。

```
int bkgd(ch)
int wbkgd(win, ch)
```

它们将把背景字符改为ch，把属性改为ch的属性。

8.7 输入

```
int getch()

int wgetch(win)

int mvgetch(y, x)

int mvwgetch(win, y, x)
```

getch ( )将从终端读取输入，读取的方式取决于是否设置了延迟模式。如果设置了延迟模式，则getch ( )将一直等待，直到用户按下一个键为止；如果没有设置延迟模式，则它将返回输入缓冲区中的数据，如果输入缓冲区为空，则它将返回 ERR。mvgetch (…)和mvwgetch (…)首先把光标移动到位置 (y,x)上。名称中有 w字母的函数将从与窗口 win相关的终端读取输入，getch ( )和mvgetch (…)则从屏幕相关的终端读取。

如果使能了keypad (…)，在用户按下某个功能键时， getch ( )将返回一个代码，该代码在.h头文件中被定义为 KEY_*宏。如果用户按下 Esc键(它可能会是某个组合功能键的第一个键)，则ncurses将启动一个一秒钟的定时器，如果在这一秒钟时间内没有按完其余的键，则返回Esc。否则就返回功能键的值。 (如果需要的话，可以使用 notimeout ( )来关闭第二个定时器。)

```
int ungetch(ch)
```

这个函数将把字符ch送回输入缓冲区。

```
int getstr(str)
int wgetstr(win, str)
int mvgetstr(y, x, str)
int mvwgetstr(win, y, x, str)
int wgetnstr(win, str, n)
```

这些函数的作用相当于对getch ( )进行一系列的调用，直到接收到一个新行。行中的字符存放在str中(所以，在调用getstr (…)之前，不要忘记给字符指针分配内存)。如果打开了回送，则字符串将被显示出来 (使用noecho ( )可以关闭回送)，而用户的删除字符以及其他特殊字符也会被解释出来。

```
chtype inch()

chtype winch(win)
chtype mvinch(y, x)
chtype mvwinch(win, y, x)
```

这些函数从屏幕或窗口返回一个字符，因为返回值的类型是 chtype，所以还包括了属性信息。这一信息可以使用常量 A_*从字符中扩展得到。

```
int instr(str)
int innstr(str, n)
int winstr(win, str)
int winnstr(win, str, n)
int mvinstr(y, x, str)
int mvinnstr(y, x, str, n)
int mvwinstr(win, y, x, str)
int mvwinnstr(win, y, x, str, n)
```

这些函数从屏幕或窗口返回一个字符串。(注：这些函数尚未实现)

```
int inchstr(chstr)
int inchnstr(chstr, n)
int winchstr(win, chstr)
int winchnstr(win, chstr, n)
int mvinchstr(y, x, chstr)
int mvinchnstr(y, x, chstr, n)
int mvwinchstr(win, y, x, chstr)
int mvwinchnstr(win, y, x, chstr, n)
```

这些函数将从屏幕或窗口返回一个 chtype字符串。该字符串包含了每个字符的属性信息(注：尚未实现，在ncurses库中没有包含lib_inchstr)。

格式化输入

```
int scanw(fmt, ...)
int wscanw(win, fmt, ...)
int mvscanw(y, x, fmt, ...)
int mvwscanw(win, y, x, fmt, ...)
int vwscanw(win, fmt, va_list)
```

这些函数类似于scanf (…)。如果在调用这些函数之前调用了 wgetstr (…)，则它的结果将会成为这些函数的输入。

8.8 选项

8.8.1 输出选项

```
int idlok(win, bf)
void idcok(win, bf)
```

这两个函数为窗口使能或者关闭终端的 insert/delete特征(idlok (…)针对一行，而idcok (…)则针对字符)。(注：idcok (…)尚未实现)

```
void immedok(win, bf)
```

如果bf设置为TRUE，则对窗口win的每一次改变都将导致物理屏幕的一次刷新。这将使程序的性能降低，所以默认的值是 FALSE。(注：此函数尚未实现)

```
int clearok(win, bf)
```

如果bf值为TRUE，则下一次调用wrefresh (win)时将会清除屏幕，并完全地把它重新画一遍(就像用户在编辑器vi中按下Ctrl+L一样)。

```
int leaveok(win, bf)
```

默认的行为是，ncurses让物理光标停留在上次刷新窗口时的同一个位置上。不使用光标的程序可以把leaveok (…)设置为TRUE，这样一般可以节省光标移动所需要的时间。此外，ncurses将试图使终端光标不可见。

```
int nl()
int nonl()
```

这两个函数控制新行的平移。使用 nl ( )可以打开平移，这样在回车时就会平移到新的一行，在输出时就会走行。而 nonl ( )可以把平移关上。关上平移之后，ncurses做光标移动操作时速度就会快一些。

8.8.2 输入选项

```
int keypad(win, bf)
```

如果bf为TRUE，该函数在等待输入时会使能用户终端的键盘上的小键盘。ncurses将返回一个键代码，该代码在 .h头文件中被定义为 KEY_*宏，它是针对小键盘上的功能键和方向键的。对于PC键盘来说，这一点是非常有帮助的，因为这样用户就可以使能数字键和光标键。

```
int meta(win, bf)
```

如果bf为TRUE，从getch ( )返回的键代码将是完整的8位(最高位将不会被去掉)。

```
int cbreak()
int nocbreak()
int crmode()
int nocrmode()
```

cbreak ( )和nocbreak ( )将把终端的CBREAK模式打开或关闭。如果CBREAK打开，则程序就可以立刻使用读取的输入信息。如果 CBREAK关闭，则输入将被缓存起来，直到产生新的一行(注意：crmode ( )和nocrmode ( )只是为了提供向上兼容性，不要使用它们)。

```
int raw()
int noraw()
```

这两个函数将把RAW模式打开或关闭。RAW与CBREAK相同，它们的区别在于 RAW模式不处理特殊字符。

```
int echo()
int noecho()
```

如果把echo ( )设置为TRUE，则用户所敲的输入将会回送并显示出来，而 noecho ( )则对此保持沉默。

```
int halfdelay(t)
```

此函数与cbreak()相似，但它要延迟t秒钟。

```
int nodelay(win, bf)
```

终端将被设置为非阻塞模式。如果没有任何输入则 getch ( )将返回ERR，否则如果设置为FALSE，则getch ( )将等待，直到用户按下某个键为止。

```
int timeout(t)
int wtimeout(win, t)
```

笔者提倡大家使用这两个函数，而不要使用 halfdelay (t)和nodelay (win, bf)。getch ( )的结

果取决于t的值。如果t是正数，则读操作将被阻塞t毫秒；如果t为零，则不发生任何阻塞；如果t是负数，则程序将阻塞，直到有输入为止。

```
int notimeout(win, bf)
```

如果bf为TRUE，则getch ( )将使用一个特殊的定时器 (一秒钟长)。到时间以后，再对以Esc等键打头的输入序列进行解释。

```
int typeahead(fd)
```

如果fd是-1，则不检查超前键击，否则ncurses将使用文件描述符fd来进行这些检查。

```
int intrflush(win, bf)
```

当bf为TRUE时使能该函数。在终端上按下任意中断键 (quit、break…)时，所有的输出将会刷新到tty驱动程序队列中。

```
void noqiflush()
void qiflush()
```

(注，这两个函数尚未实现)

8.8.3 终端属性

```
int baudrate()
```

此函数返回终端的速度，以bps为单位。

```
char erasechar()
```

此函数返回当前删除的字符。

```
char killchar()
```

此函数返回当前杀死的字符。

```
int has_ic()
int has_il()
```

如果终端具有插入/删除字符的能力，则has_rc()将返回TRUE。如果终端具有插入/删除行的能力，则has_il ( )将返回TRUE，否则这两个函数将返回ERR。(注：尚未实现)

```
char *longname()
```

此函数所返回的指针允许用户访问当前终端的描述符。

```
chtype termattrs()
```

(注：此函数尚未实现)

```
char *termname()
```

这个函数从用户环境中返回TERM的内容。(注：尚未实现)

8.8.4 使用选项

读者现在已经了解了窗口选项和终端方式，下面将介绍它们的用法。

首先，在Linux上用户必须使用小键盘。这将使用户可以使用 PC键盘上的光标键和数字键。

目前有两种类型的输入：

1. 程序希望用户输入一个键，然后根据这个键再调用某个函数。 (例如，等待用户按“ q ”，按“ q ”代表quit)。

2. 程序希望用户在屏幕上的某个位置输入一个字符串。例如，数据库中的某个目录或者地址。

首先我们使用下面的选项和方式，这样while循环将可以正确地工作。

在用户按下某个键之前，程序将一直挂起。如果用户所按的是“ q”，则调用退出函数，否则需要一直等待其他输入。

程序需要依次判断switch语句的条件，直到找到符合条件的输入函数。使用KEY_*宏可以检查特殊的键，例如小键盘上的光标键。在文件浏览器中循环看起来如下所示：

(译者注：原文缺)

在这一秒钟时间内，我们只需要设置echo ( )，用户输入的字符将会显示到屏幕上。如果想把字符显示在用户所希望的位置，可以使用函数move (…)或者wmove (…)。

或者用户也可以打开一个窗口，在窗口中有一个掩模 (与窗口不同的一些其他颜色)，然后要求用户输入一个字符串：

注意 出版者注：作者尚未提供实例。

在实例目录中有一些C语言程序，读者可以在这些程序中了解到更多的情况。

8.9 更新终端

正如以前所介绍的那样， ncurses窗口是内存中的一个映像。这意味着在进行刷新以前，对窗口的任意改变将不会显示在屏幕上，这将优化屏幕的输出，因为这样一来用户就可以进行大量的操作，然后再一次性地把它们刷新到屏幕上。否则，每一次改变窗口都将刷新到终端，这将会降低程序的性能。

```
int refresh()
int wrefresh(win)
```

refersh ( )将把窗口映像拷贝到终端，而wrefresh (win)将把窗口映像拷贝到win，并使它看起来象原来的样子。

```
int wnoutrefresh(win)
int doupdate()
```

wnoutrefresh (win)将会只拷贝到窗口win，这意味着在终端上将不进行任何输出，但是虚拟屏幕实际上看起来象程序员所希望的那样。 doupdate ( )将输出到终端上。程序可以改变许多窗口，对每个窗口都调用一次wnoutrefresh (win)，然后再调用一次doupdate ( )来更新物理屏幕。

例如下面的程序，该程序中用到两个窗口，通过修改一些文本行可以改变这两个窗口。用户可以使用wrefresh (win)来编写函数changewin (win)。

(译者注：原文未提供例题)

这样做会让ncurses更新终端两次，会降低程序的执行速度。我们使用 doupdate ( )来改写函数changewin (win)，这样主函数性能会好一些。

```
int redrawwin(win)
int wredrawln(win, bline, nlines)
```

如果在往屏幕上输出新内容时需要清除一些行或者整个屏幕，可以使用这两个函数。 (可能这些行已经被破坏了或者由于其他的原因。)

```
int touchwin(win)
int touchline(win, start, count)
int wtouchln(win, y, n, changed)
int untouchwin(win)
```

这些函数通知ncurses整个win窗口已经被改动过了，或者从start直到start+count的这些行已经被改动过了。例如，如果用户有一些重叠的窗口（正如在example.c中一样），对某个窗口的改动不会影响其他窗口的映像。

wtouchln (…)将按掀从y开始的n行。如果change的值是TRUE，则这些行被按掀过了，否则就还未被按掀过(改变或未改变)。

untouchwin (win)将把窗口win标记为自上次调用refresh ( )以来还未被按掀。

```
int is_linetouched(win, line)
int is_wintouched(win)
```

通过使用这两个函数，用户可以检查自从上次调用 refresh ( )以来，第line行或者窗口win是否已被按掀过。

8.10 视频属性与颜色

属性是特殊的终端权能，当往屏幕上输出字符时需要用到它。字符可以打印成粗体、带下划线的、闪烁的等等。在 ncurses中，用户可以打开或关闭属性，以获得更佳的输出效果。下面列出了一些可能的属性。

表8-4 ncurses：属性

注意 出版者注：作者尚未提供此表。

ncurses定义了八种颜色，在带有彩色支持的终端上用户可以使用这些颜色。首先，调用start_color ( )初始化颜色数据结构，然后使用 has_colors ( )检查终端权能。start_color ( )将初始化COLORS和COLOR_PAIR。前者是终端所支持的最多的颜色数目，而后者是用户可以定义的色彩对的最大数目。

表8-5 ncurses：颜色

注意 出版者注：作者尚未提供此表。

这两个属性可以使用OR操作组合起来。“ COLORPAIRS_1 COLORS_1 ”

注意 出版者注：作者尚未提供代码。

```
int color_content(color, r, g, b)
```

此函数获取color的颜色成份r, g和b。

那么，如何把属性和颜色组合起来呢？有些终端，例如 Linux中的控制台，具有一些颜色，而其他终端却没有(xterm、VS100等等)。下面的代码可以解决这个问题：

注意 出版者说明：作者尚未提供源代码。

首先，函数CheckColor调用start_color ( )初始化颜色，如果当前终端有彩色的话，则函数has_colors ( )将返回TRUE。我们检查了这一点以后，调用 init_pair (…)把前景色和背景色组

合起来，再调用wattrset (…)为特定的窗口设置这些颜色对。此外，如果我们使用的是黑白终端，还可以单独使用wattrset (…)来设置属性。

如果要在xterm中获取颜色，我认为最佳方法是使用 ansi_xterm，以及来自Midnight Commander的terminfo项目。用户可以获取 ansi_xterm和Midnight Commander的源代码(mc_x.x.tar.gz)，然后编译ansi_xterm，并对mc_x.x.tar.gz文档中的xterm.ti和vt100.ti使用tic命令。执行ansi_xterm，把它试验出来。

8.11 光标和屏幕坐标

```
int move(y, x)
int wmove(win, y, x)
```

move ( )将移动光标，而wmove (win)则从窗口win中移动光标。对输入/输出函数来说，还定义了其他的一些宏，在调用特定函数之前，这些宏可以移动光标。

```
int curs_set(bf)
```

这个函数将把光标置为可见或者不可见，如果终端具有这个功能的话。

```
void getyx(win, y, x)
```

getyx (…)将返回当前光标位置。(注意：这是一个宏)

```
void getparyx(win, y, x)
```

如果win是个子窗口，getparyx (…)将把该窗口对应父窗口的坐标存储在y和x中，否则y和x都将为－1。(注：此函数尚未实现)

```
void getbegyx(win, y, x)
void getmaxyx(win, y, x)
int getmaxx(win)
int getmaxy(win)
```

这些函数把窗口win的开始坐标和大小坐标存放在y和x中。

```
int getsyx(int y, int x)
int setsyx(int y, int x)
```

getsyx (…)把虚拟屏幕光标存放在y和x中，而setsyx (…)则设置这个坐标。如果y和x是－1，用户调用getsyx (…)将会设置leaveok。

8.12 滚动

```
int scrollok(win, bf)
```

当光标在屏幕的右下角并且输入了一个字符 (或者新的一行)时，如果bf为TRUE，则窗口win中的文本将上滚一行。如果bf为FALSE，则鼠标留在原来的位置上。

当滚动特征打开时，使用下面的函数可以滚动窗口中的内容。 (注意：当用户在窗口的最后一行输入一个新行时，也应该发生相应的滚动操作，所以在使用 scrollok (…)时要十分小心，否则可能会得到出乎意料的结果。)

```
int scroll(win)
```

此函数将使窗口向上滚动一行(数据结构中的行也向上滚动)。

```
int scrl(n)
int wscrl(win, n)
```

这两个函数将使屏幕或者窗口 win向上向下滚动，滚动方向取决于整数 n的值。如果n是正数，则窗口向上滚动n行，否则如果n是负数，则窗口向下滚动 n行。

```
int setscrreg(t, b)
int wsetscrreg(win, t, b)
```

这两个函数设置一个软滚动区。

下面的代码将向读者说明怎样在屏幕上获得滚动文本的效果，也可以参见实例目录中的 .c文件。

注意 出版者注：作者尚未提供代码。

在程序中，我们有一个18行66列的窗口，我们希望在其中滚动文本。 S[ ]是存放文本的字符数组。 Max_s是s[ ]中最后一行的编号。 Clear_line将使用窗口的当前属性打印空白字符，从当前光标位置一直打印到本行的结束， (属性不同于clrtoeol所使用的A_NORMAL)。Beg是当前显示在屏幕上的 s[ ]文本的最后一行。 Scroll是个枚举类型，它告诉函数应该做些什么，并把文本的NEXT行或者PREV行显示出来。

8.13 小键盘

```
WINDOW *newpad(nlines, ncols)
WINDOW *subpad(orig, nlines, ncols, begy, begx)
int prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
int pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
int pechochar(pad, ch)
```

8.14 软标签

```
int slk_init(int fmt)
int slk_set(int labnum, char *label, int fmt)
int slk_refresh()
int slk_noutrefresh()
char *slk_label(int labnum)
int slk_clear()
int slk_restore()
int slk_touch()
int slk_attron(chtype attr)
int slk_attrset(chtype attr)
int slk_attroff(chtype attr)
```

这些函数是与attron (attr)、attrset (attr)和attroft (attr)相对应的，但它们尚未实现。

8.15 杂项

```
int beep()
int flash()
char *unctrl(chtype c)
char *keyname(int c)
int filter()
```

(注：以上函数尚未实现。)

```
void use_env(bf)
int putwin(WINDOW *win, FILE *filep)
```

(注：以上函数尚未实现。)

```
WINDOW *getwin(FILE *filep)
```

(注：此函数尚未实现。)

```
int delay_output(int ms)
int flushinp()
```

8.16 低级访问

```
int def_prog_mode()
int def_shell_mode()
int reset_prog_mode()
int reset_shell_mode()
int resetty()
int savetty()
int ripoffline(int line, int (*init)(WINDOW *, int))
int napms(int ms)
```

8.17 屏幕转储

```
int scr_dump(char *filename)
```

(注：此函数尚未实现。)

```
int scr_restore(char *filename)
```

(注：此函数尚未实现。)

```
int scr_init(char *filename)
```

(注：此函数尚未实现。)

```
int scr_set(char *filename)
```

(注：此函数尚未实现。)

8.18 Termcap模拟

```
int tgetent(char *bp, char *name)
int tgetflag(char id[2])
int tgetnum(char id[2])
char *tgetstr(char id[2], char **area)
char *tgoto(char *cap, int col, int row)
int tputs(char *str, int affcnt, int (*putc)())
```

8.19 Terminfo函数

```
int setupterm(char *term, int fildes, int *errret)
int setterm(char *term)
int set_curterm(TERMINAL *nterm)
int del_curterm(TERMINAL *oterm)
int restartterm(char *term, int fildes, int *errret)
```

(注：以上函数尚未实现。)

```
char *tparm(char *str, p1, p2, p3, p4, p5, p6, p7, p8, p9)
p1 - p9 long int.
int tputs(char *str, int affcnt, int (*putc)(char))
int putp(char *str)
int vidputs(chtype attr, int (*putc)(char))
int vidattr(chtype attr)
int mvcur(int oldrow, int oldcol, int newrow, int newcol)
int tigetflag(char *capname)
int tigetnum(char *capname)
int tigetstr(char *capname)
```

8.20 调试函数

- void _init_trace()
- void _tracef(char *, ...)
- char *_traceattr(mode)
- void traceon()
- void traceoff()

8.21 Terminfo权能

8.21.1 布尔型权能

变 量	权能名称	初始化	描 述
auto_left_margin	bw	bw	cub1从最后一列换行到第0列
auto_right_margin	am	am	终端的边界自动对齐
back_color_erase	bce	ut	屏幕以背景色清除
can_change	ccc	cc	终端可以重新定义现有的颜色
ceol_standout_glitch	xhp	xs	标准输出不会被覆盖所清除(hp)
col_addr_glitch	xhpa	YA	对hpa/mhpa大写字符而言只作正向移动
cpi_changes_res	cpix	YF	改变字符间距将会影响解析度
cr_cancels_micro_mode	crxm	YB	使用cr关闭宏模式
eat_newline_glitch	xenl	xn	在80列之后将忽略新行(Concept)
erase_overtrike	eo	eo	可以用空格来删除叠印
generic_type	gn	gn	通用行类型(如dialup, switch)
hard_copy	hc	hc	硬拷贝终端
hard_cursor	chts	HC	光标很难看到
has_meta_key	km	km	有一个元键(shift, 设置奇偶校验位)
has_print_wheel	daisy	YC	打印机需要操作员来改变字符集
has_status_line	hs	hs	有一个额外的“状态行”
hue_lightness_saturation	hls	hl	终端只使用HLS颜色表示法(Tektronix)
insert_null_glitch	in	in	插入模式, 能识别空行
lpi_changes_res	lpix	YG	改变行距将影响解析度
memory_above	da	da	显示可以保留在屏幕上方
memory_below	db	db	显示可以保留在屏幕下方

(续)

变 量	权能名称	初始化	描 述
move_insert_mode	mir	mi	在插入模式下可以安全地移动
move_standout_mode	msgr	ms	在标准输出模式下可以安全地移动
needs_xon_xoff	nxon	nx	不能填充，需要nxon/xoff
no_esc_ctl_c	xsb	xb	Beehive信号（F1=Escape，F2=Ctrl C）
non_rev_rmcup	nrrmc	NR	smcup不能反转rmcup
no_pad_char	npc	NP	填充字符不存在
non_dest_scroll_region	ndscr	ND	滚动区不可摧毁
over_strike	os	os	终端可以叠印
prtr_silent	mc5i	5i	打印机不向屏幕回送
row_addr_glitch	xvpa	YD	vhp/mvpa大写字母只能作正向移动
semi_auto_right_margin	sam	YE	打印在最后一列将导致cr
status_line_esc_ok	eslok	es	在状态行上可以使用Esc键
dest_tabs_magic_smso	xt	xt	制表符不可用(Teleray 1061)
tilde_glitch	hz	hz	Hazel_tine；不能打印 ' s
transparent_underline	ul	ul	下划线字符叠印
xon_coff	xon	xo	终端使用xon/xoff握手机制

8.21.2 数值型权能

变 量	权能名称	初始值	描 述
bit_image_entwining	bitwin	Yo	在SYSV中未作描写
buffer_capacity	bufsz	Ya	在打印前缓存的字节的数目
columns	cols	co	在一行中列的数目
dot_vert_spacing	spinv	Yb	在水平方向上点与点的距离，以每英寸多少点为单位
dot_horz_spacing	spinh	Yc	在垂直方向上针之间的距离，以每英寸多少针为单位
init_tabs	it	it	每#个空格算一个制表符的位置
label_height	lh	lh	每个标签多少行
label_width	lw	lw	每个标签多少列
lines	lines	li	屏幕或页面上行的数目
lines_of_memory	lm	lm	如果>lines则表示内存中的行数，0意味着可变
magic_cookie_glitch	xmc	sg	smso或rmso所剩下的空白字符的数目
max_colors	colors	Co	在屏幕上颜色的最大数目
max_micro_address	maddr	Yd	在micro_…_address中的最大值
max_micro_jump	mjump	Ye	在parm_…_micro中的最大值
max_pairs	pairs	pa	在屏幕上颜色对的最大数目
micro_col_size	mcs	Yf	在宏模式中字符间距的大小
micro_line_size	mls	Yg	在宏模式中行距的大小
no_color_video	ncv	NC	不能使用彩色的视频属性
number_of_pins	npins	Yh	在打印头中针的数目
num_labels	nlab	Nl	屏幕上标签的数量
output_res_char	orc	Yi	水平解析度，以每行单元数为单位
output_res_line	orl	Yj	垂直解析度，以每行单元数为单位
output_res_horz_inch	orhi	Yk	水平解度，以每英寸单元数为单位
output_res_vert_inch	orvi	Yl	垂直解析度，以每英寸单元数为单位
padding_baud_rate	pb	pb	在需要cr/nl填充时最低的波特率
virtual_terminal	vt	vt	虚拟终端号(Unix系统)
width_status_line	wsl	ws	状态行的第n列

(下面的数值型权能是在 SYSV term结构中定义的，但在 man帮助中还没有提供对它们的描述。我们的解释来自term结构的头文件。)

(续)

变 量	权能名称	初始值	描 述
bit_image_type	bitype	Yp	位映像设备的类型
buttons	btns	BT	鼠标按键的数目
max_attributes	ma	ma	终端能够处理的最多的组合属性
maximum_windows	wnum	MW	可定义窗口的最大数目
print_rate	cps	Ym	打印速率，以每秒字符数为单位
wide_char_size	widcs	Yn	在双宽度模式中字符间距的大小

8.21.3 字符串型权能

变 量	权能名称	初始值	描 述
acs_chars	acsc	ac	图形字符集对 一def = vt100
alt_scancode_esc	scesa	S8	扫描码模拟的另一种换码(默认值是VT100)
back_tab	cbt	bt	向后tab (p)
bell	bel	bl	声音信号(响铃)(p)
bit_image_repeat	birep	Xy	把位映像单元重复＃1 ＃2次(使用tparm)
bit_image_newline	binel	Zz	移动到位映像的下一行(使用tparm)
bit_image_carriage_return	bicr	Yv	移动到同一行的开头(使用tparm)
carriage_return	cr	cr	回车(p*)
change_char_pitch	cpi	ZA	改变为每英寸＃个字符
change_line_pitch	lpi	ZB	改变为每英寸＃行
change_res_horz	chr	ZC	改变水平解度
change_res_vert	cvr	ZD	改变垂直解析度
change_scroll_region	csr	cs	把滚动区改变为从＃1行到＃2行(VT100)(PG)
char_padding	rmp	rP	与ip相似，但它用在插入模式中
char_set_names	csnm	Zy	字符集名称的列表
clear_all_tabs	tbc	ct	清除所有的制表符停止(p)
clear_margins	mgc	MC	清除所有的页边
clear_screen	clear	cl	清除屏幕和home光标(p*)
clr_bol	el1	cb	清除到行首
clr_eol	el	ce	清除到行尾(P)
clr_eos	ed	cd	清除到显示的末尾(p*)
code_set_init	csin	ci	多个代码集合的初始化序列
color_names	colornm	Yw	＃1号颜色的名称
column_address	hpa	ch	设置光标列(PG)
command_character	cmdch	CC	在原型中终端可以设置的cmd字符
cursor_address	cup	cm	屏幕光标移动到＃1行＃2列(PG)
cursor_down	cud1	do	下移一行
cursor_home	home	ho	Home光标(如果没有环的话)
cursor_invisible	civis	vi	使光标不可见
cursor_left	cub1	le	把光标向左移一个空格
cursor_mem_address	mrcup	CM	内存相对的光标寻址
cursor_normal	cnorm	ve	使光标以最普通的外形显示(undo vs/vi)
cursor_right	cuf1	nd	不具有破坏性的空白(光标向右移)
cursor_to_ll	ll	ll	最后一行，第一列(如果没有环的话)
cursor_up	cuu1	up	Upline(光标向上移)

(续)

变量	权能名称	初始值	描述
cursor_visible	cvvis	vs	使光标可见
define_bit_image_region	defbi	Yx	定义方形的位映像区(使用tparm)
define_char	defc	ZE	定义字符集中的某个字符
delete_character	dch1	dc	删除字符(p*)
delete_line	dl1	dl	删除行(p*)
device_type	devt	dv	显示语言/代码集支持
dis_status_line	dsl	ds	关闭状态行
display_pc_char	dispc	S1	显示PC字符
down_half_line	hd	hd	向下移动半行(向前换1/2行)
ena_acs	enacs	eA	使能另一个字符集合
end_bit_image_region	endbi	Yy	结束位映像区(使用tparm)
enter_alt_charset_mode	smacs	as	开始另一个字符集(p)
enter_am_mode	smam	SA	打开自动对齐特征
enter_blink_modeblink	mb		打开字符闪烁效果
enter_bold_mode	bold	md	打开粗体(特别亮)模式
enter_ca_mode	smcup	ti	启动使用环的程序的字符串
enter_delete_mode	smdc	dm	删除模式(输入)
enter_dim_mode	dim	mh	打开半亮模式
enter_doublewide_mode	swidm	ZF	使能双倍宽度模式
enter_draft_quality	sdrfq	ZG	设置草图效果的打印方式
enter_insert_mode	smir	im	插入模式(输入)
enter_italics_mode	sitm	ZH	使能斜体字模式
enter_leftward_mode	slm	ZI	使能向左回车移动
enter_micro_mode	smicm	ZJ	使能宏移动功能
enter_near_letter_quality	snlq	ZK	设置NLQ打印
enter_normal_quality	snrmq	ZL	设置一般质量的打印方式
enter_pc_charset_mode	smpch	S2	输入PC字符显示模式
enter_protected_mode	prot	mp	打开保护模式
enter_reverse_mode	rev	mr	打开反转视频模式
enter_scancode_mode	smsc	S4	输入PC扫描码
enter_secure_mode	invis	mk	打开空白模式(字符不可见)
enter_shadow_mode	sshm	ZM	使能阴影打印模式
enter_standout_mode	smso	so	开始标准输出模式
enter_subscript_mode	ssubm	ZN	使能下标打印
enter_superscript_mode	ssupm	ZO	使能上标打印
enter_underline_mode	smul	us	开始下划线模式
enter_upward_mode	sum	ZP	使能向上回车移动
enter_xon_mode	smxon	SX	打开xon/xoff握手机制
erase_chars	ech	ec	删除#1个字符(PG)
exit_alt_charset_mode	rmacs	ae	终止可选的字符集(P)
exit_am_mode	rmam	RA	关闭自动对齐方式
exit_attribute_mode	sgr0	me	关闭所有属性
exit_ca_mode	rmcup	te	终止使用环的程序的字符串
exit_delete_mode	rmdc	ed	终止删除模式
exit_doublewide_mode	rwidm	ZQ	关闭双倍宽度打印方式
exit_insert_mode	rmir	ei	结束插入模式
exit_italics_mode	ritm	ZR	关闭斜体打印模式

(续)

变 量	权能名称	初始值	描 述
exit_leftward_mode	rlm	ZS	使能右向(普通的)回车移动
exit_micro_mode	rmicm	ZT	关闭宏移动能力
exit_pc_charset_mode	rmpch	S3	关闭PC字符显示
exit_scancode_mode	rmsc	S5	关闭PC扫描码模式
exit_shadow_mode	rshm	ZU	关闭阴影打印模式
exit_standout_mode	rmso	se	结束标准输出模式
exit_subscript_mode	rsubm	ZV	关掉下标打印方式
exit_superscript_mode	rsupm	ZW	关掉上标打印方式
exit_underline_mode	rmul	ue	结束下划线模式
exit_upward_mode	rum	ZX	打开向下(普通的)回车移动
exit_xon_mode	rmxon	RX	关掉xon/xoff握手机制
flash_screen	flash	vb	可视响铃(不能移动光标)
form_feed	ff	ff	硬拷贝终端页面的换页(p*)
from_status_line	fsl	fs	从状态行返回
init_1string	is1	i1	终端初始化字符串
init_2string	is2	i2	终端初始化字符串
init_3string	is3	i3	终端初始化字符串
init_file	if	if	所包含的文件名称
init_prog	iprog	iP	初始化程序的路径名
initialize_color	initc	Ic	初始化颜色的定义
initialize_pair	initp	Ip	初始化颜色对
insert_character	ich1	ic	插入字符(P)
insert_line	il1	al	加入一个新的空白行(p*)
insert_padding	ip	ip	在插入的字符之后再插入填充字符(p*)
key_a1	ka1	K1	小键盘左上方的键
key_a3	ka3	K3	小键盘右上方的键
key_b2	kb2	K2	小键盘中央的键
key_backspace	kbs	kb	由回退键所发送
key_beg	kbeg	1	开始键
key_btab	kcbt	kB	向右一tab键
key_c1	kc1	K4	小键盘左下角的键
key_c3	kc3	K5	小键盘右下角的键
key_cancel	kcan	2	取消键
key_catab	ktbc	ka	由clear_all_tabs键发送
key_clear	kclr	kC	由清除屏幕或者删除键发送
key_close	kclo	3	关闭键
key_command	kcmd	4	命令键
key_copy	kcpy	5	拷贝键
key_create	kcrt	6	创建键
key_ctab	kctab	kt	由clear_tab键发送
key_dc	kdch1	kD	由删除字符键发送
key_dl	kdl1	kL	由删除行键发送
key_down	kcud1	kd	由终端向下光标键发送
key_eic	krmir	kM	在插入模式中由rmir或smir发送
key_end	kend	7	结束键
key_enter	kent	8	输入/发送键
key_eol	kel	kE	由clear_to_end_of_line键发送

(续)

变 量	权能名称	初始值	描 述
key_eos	ked	kS	由clear_to_end_of_screen键发送
key_exit	kext	9	退出键
key_find	kfnd	0	查找键
key_help	khlp	%1	帮助键
key_home	khome	kh	由home键发送
key_ic	kich1	kl	由ins char/enter ins mode键发送
key_il	kil1	kA	由插入行发送
key_left	kcub1	kl	由终端向左键发送
key_ll	kll	kH	由home_down键发送
key_mark	kmrk	%2	标记键
key_message	kmsg	%3	消息键
key_move	kmov	%4	移动键
key_next	knxt	%5	下一个键
key_npage	knp	kN	由下页键发送
key_open	kopn	%6	打开键
key_options	kopt	%7	选项键
key_ppage	kpp	kP	由前页键发送
key_previous	kprv	%8	前一键
key_print	kprt	%9	打印键
key_redo	krdo	%0	redo键
key_refrence	kref	&1	引用键
key_refresh	krfr	&2	刷新键
key_replace	krpl	&3	替换键
key_restart	krst	&4	重启键
key_resume	kres	&5	恢复键
key_right	kcuf1	kr	由终端向右键发送
key_save	ksav	&6	保存键
key_sbeg	kBEG	&9	按下开始键的同时按下shift键
key_scancel	kCAN	&0	按下取消键的同时按下shift键
key_scommand	kCMD	*1	按下命令键的同时按下shift键
key_scopy	kCPY	*2	按下拷贝键的同时按下shift键
key_screate	kCRT	*3	按下创建键的同时按下shift键
key_sdc	kDC	*4	按下删除字符键的同时按下shift键
key_sdl	kDL	*5	按下删除行键的同时按下shift键
key_select	kslt	*6	选择键
key_send	kEND	*7	按下结束键的同时按下shift键
key_seol	kEOL	*8	按下行尾键的同时按下shift键
key_sexit	kEXT	*9	按下退出键的同时按下shift键
key_sf	kind	kF	由前滚/下滚键发送
key_sfind	kFND	*0	按下查找键的同时按下shift键
key_shelp	kHLP	#1	按下帮助键的同时按下shift键
key_shome	kHOM	#2	按下Home键的同时按下shift键
key_sic	kIC	#3	按下插入字符键的同时按下shift键
key_sleft	kLFT	#4	按下向左键的同时按下shift键
key_smessage	kMSG	%a	按下消息键的同时按下shift键
key_smove	kMOV	%b	按下移动键的同时按下shift键
key_snext	kNXT	%c	按下向后键的同时按下shift键

(续)

变 量	权能名称	初始值	描 述
key_soptions	kOPT	%d	按下选项键的同时按下shift键
key_sprevious	kPRV	%e	按下向前键的同时按下shift键
key_sprint	kPRT	%f	按下打印键的同时按下shift键
key_sr	kri	kR	由后滚/下滚键发送
key_sredo	kRDO	%g	按下redo键的同时按下shift键
key_sreplace	kRPL	%h	按下替换键的同时按下shift键
key_sright	kRIT	%l	按下向右键的同时按下shift键
key_srsume	kRES	%j	按下恢复键的同时按下shift键
key_ssave	kSAV	!1	按下保存键的同时按下shift键
key_ssuspend	kSPD	!2	按下中断键的同时按下shift键
key_sundo	kUND	!3	按下取消键的同时按下shift键
key_stab	khts	kT	由set_tab键发送
key_suspend	kspd	&7	中断键
key_undo	kund	&8	取消键
key_up	kcuul	ku	由终端的向上键发送
keypad_local	rmkx	ke	不处于“小键盘发送”方式之中
keypad_xmit	smkx	ks	把终端置为“小键盘发送”方式
lab_f0	lf0	l0	如果不是f0的话，则为功能键f0的标签
lab_f1	lf1	l1	如果不是f1的话，则为功能键f1的标签
lab_f2	lf2	l2	如果不是f2的话，则为功能键f2的标签
lab_f3	lf3	l3	如果不是f3的话，则为功能键f3的标签
lab_f4	lf4	l4	如果不是f4的话，则为功能键f4的标签
lab_f5	lf5	l5	如果不是f5的话，则为功能键f5的标签
lab_f6	lf6	l6	如果不是f5的话，则为功能键f6的标签
lab_f7	lf7	l7	如果不是f7的话，则为功能键f7的标签
lab_f8	lf8	l8	如果不是f8的话，则为功能键f8的标签
lab_f9	lf9	l9	如果不是f9的话，则为功能键f9的标签
lab_f10	lf10	la	如果不是f10的话，则为功能键f10的标签
label_on	smln	LO	打开软标签
label_off	rmln	LF	关闭软标签
meta_off	rmm	mo	关闭“元模式”
meta_on	smm	mm	打开“元模式”(8位)
micro_column_address	mhpa	ZY	近似宏调整的列—地址,
micro_down	mcud1	ZZ	近似宏调整的光标—向下
micro_left	mcutb1	Za	近似宏调整的光标—向左
micro_right	mcuf1	Zb	近似宏调整的光标—向右
micro_row_address	mvpa	Zc	近似宏调整的行—地址
micro_up	mcuu1	Zd	近似宏调整的光标—向上
newline	nel	nw	新行(行为近似于cr后跟lf)
order_of_pins	porder	Ze	匹配软件以及打印头中的针
orig_colors	oc	oc	重置所有的颜色对
orig_pair	op	op	把默认的颜色对设置为原始的那个
pad_char	pad	pc	填充字符(非空)
parm_dch	dch	DC	删除#1字符(PG*)
parm_delete_line	dl	DL	删除#1行(PG*)
parm_down_cursor	cud	DO	把光标向下移#1行(PG*)
parm_down_micro	mcud	Zf	近似宏调用的cub

(续)

变 量	权能名称	初始值	描 述
parm_ich	ich	IC	插入＃1个空白符号(PG*)
parm_index	indn	SF	向上滚动＃1行(PG)
parm_insert_line	il	AL	加入＃1个新的空白行(PG*)
parm_left_cursor	cub	LE	把光标向左移＃1个空格(PG)
parm_left_micro	mcub	Zg	近似宏调整中的cub
parm_right_cursor	cuf	Rl	把光标向右移＃1个空格(PG*)
parm_right_micro	mcuf	Zh	近似宏调整中的cuf
parm_rindex	rin	SR	回滚＃1行(PG)
parm_up_cursor	cuu	UP	把光标上移＃1行(PG*)
parm_up_micro	mcuu	Zi	近似宏调整中的cuu
pkey_key	pfkey	pk	把功能键＃1定义为字符＃2的类型
pkey_local	pfloc	pl	把功能键＃1定义为执行字符串＃2
pkey_xmit	pfx	px	把功能键＃1定义为发送字符串＃2
pkey_plab	pfxl	xl	把功能键＃1定义为发送＃2，并显示＃3
plab_norm	pln	pn	编程标签＃1，以显示字符串＃2
print_screen	mc0	ps	打印屏幕内容
prtr_non	mc5p	pO	打开打印机，打印＃1个字节
prtr_off	mc4	pf	关闭打印机
prtr_on	mc5	po	打开打印机
repeat_char	rep	rp	把字符＃1重复＃2次(PG*)
req_for_input	rfi	RF	输入请求
reset_1string	rs1	r1	把终端完全置为sane方式
reset_2string	rs2	r2	把终端完全置为sane方式
reset_3string	rs3	r3	把终端完全置为sane方式
reset_file	rf	rf	包含重置字符串的文件名称
restore_cursor	rc	rc	把光标置为上一个屏幕上的位置
row_address	vpa	cv	垂直绝对位置(设置行)(PG)
save_cursor	sc	sc	保存光标位置(P)
scancode_escape	scesc	S7	为了扫描码模拟按下Esc键
scroll_forward	ind	sf	把文本向上滚动(P)
scroll_reverse	ri	sr	把文本向下滚动(P)
select_char_set	scs	Zj	选择字符集
set0_des_seq	s0ds	s0	切换到代码集0(EUC集0，ASCII)
set1_des_seq	s1ds	s1	切换到代码集1
set2_des_seq	s2ds	s2	切换到代码集2
set3_des_seq	s3ds	s3	切换到代码集3
set_a_background	setab	AB	使用ANSI设置背景颜色
set_a_foreground	setaf	AF	使用ANSI设置前景颜色
set_attributes	sgr	sa	定义视频属性(PG9)
set_background	setb	Sb	设置当前背景颜色
set_bottom_margin	smgb	Zk	设置当前行的底部边界
set_bottom_margin_parm	smgbp	ZI	从bottomset_color_band的＃1行或＃2行设置底行
setcolor	Yz		改变＃1号色带颜色
set_color_pair	scp	sp	设置当前颜色对
set_foreground	setf	Sf	设置当前前景色
set_left_margin	smgl	ML	设置当前行的左边界
set_left_margin_parm	smglp	Zm	在＃1行(＃2行)设置左(右)边界

(续)

变 量	权能名称	初始值	描 述
set_lr_margin	smglr	ML	设置左右边界
set_page_length	slines	YZ	把页的长度设置为#1行(使用tparm)
set_right_margin	smgr	MR	把右边界设置为当前列
set_right_margin_parm	smgrp	Zn	把右边界设置为#1列
set_tab	hts	st	在当前列的所有行设置制表符
set_tb_margin	smgtb	MT	设置上下边界
set_top_margin	smgt	Zo	把上边界设置为当前行
set_top_margin_parm	smgtp	Zp	把上边界设置为#1行
set_window	wind	wi	当前窗口是从#1行到#2行，从#3列到#4列
start_bit_image	sbim	Zq	开始打印位映像图形
start_char_set_def	scsd	Zr	开始定义字符集
stop_bit_image	rbim	Zs	结束打印位映像图形
stop_char_set_def	rcsd	Zt	结束定义字符集
subscript_characters	subcs	Zu	下标字符的列表
superscript_characters	supcs	Zv	上标字符的列表
tab	ht	ta	跳转到下面8个空格硬件的制表符位置
these_cause_cr	docr	Zw	这些字符导致CR
to_status_line	tsl	ts	跳到状态行，第1列
underline_char	uc	uc	给某字符划下划线，并移过它
up_half_line	hu	hu	上移半行(反转1/2行)
xoff_character	coffc	XF	XON字符
xon_character	xonc	XN	XOFF字符

(下面的字符串权能是在SYSVr终端结构中定义的，但在man帮助信息中还未作描述，对它们的解释是从终端结构头文件中得到的。)

label_format	fln	Lf	??
set_clock	sclk	SC	设置时钟
display_clock	dclk	DK	显示时钟
remove_clock	rmclk	RC	删除时钟
create_window	cwin	CW	把窗口#1定义为从#2行,#3列到#4行，#5列
goto_window	wingo	WG	跳到窗口#1
hangup	hup	HU	挂起电话
dial_phone	dial	DI	拨电话号码#1
quick_dial	qdial	QD	拨电话号码#1，但不做进度检查
tone	tone	TO	选择接触声调拨叫
pulse	pulse	PU	选择脉冲拨叫
flash_hook	hook	fh	闪光切换分支
fixed_pause	pause	PA	暂停2～3秒
wait_tone	wait	WA	等待拨叫声音
user0	u0	u0	用户字符串#0
user1	u1	u1	用户字符串#1
user2	u2	u2	用户字符串#2
user3	u3	u3	用户字符串#3
user4	u4	u4	用户字符串#4
user5	u5	u5	用户字符串#5
user6	u6	u6	用户字符串#6

(续)

变 量	权能名称	初始值	描 述
user7	u7	u7	用户字符串 #7
user8	u8	u8	用户字符串 #8
user9	u9	u9	用户字符串 #9
get_mouse	getm	Gm	surses应获得按钮事件
key_mouse	kmous	Km	??
mouse_info	minfo	Mi	鼠标状态信息
pc_term_options	pctrm	S6	PC终端选项
req_mouse_pos	reqmp	RQ	请求鼠标位置报告
zero_motion	zerom	Zx	后继字符没有移动

8.22 [N]Curses函数概述

下面介绍不同的(n)curses服务包。第一列是bsd_curses(如Slackware 2.1.0中，与Sun OS 4.x中一样)，第二列是sysv_curses(在Sun OS 5.4/Solaris 2中)，第三列是ncurses(版本1.8.6)。第四列的描述是摘自描述该函数的文本(如果有对该函数的描述的话)。

注意 出版者注：作者未提供此表。

x
服务包中有这个函数。
n
此函数尚未实现。
(尚待完成)

第9章　I/O端口编程

通常一台pc机至少有两个串行接口和一个并行接口。这些接口是特殊的设备，它们以如下方式映射：

- 它们是RS232串行设备0～n，这里n的大小取决于用户的硬件。
- 它们是并行设备0～n，这里n的值取决于用户的硬件。
- 它们是游戏杆设备0～n。

/dev/ttys*和/dev/cua*设备之间的差异之处在于怎样处理 open( )函数的调用。/dev/cua*设备常用作调用设备，通过调用 open( )可以获得其他的默认设置；而 /dev/ttys*设备则将为到达的调用以及发出的调用而初始化。

注意　出版者注：/dev/cua*设备现在正逐渐淘汰。

在默认情况下，设备通常由那些打开该设备的进程来控制。通常 ioctl()请求可以处理所有这些特殊的设备，但POSIX更喜欢定义新函数，以便根据结构 termios的不同来处理异步终端。这两种方法都需要包含〈termios.h〉。

1) ioctl 方法：TCSBRK、TCSBRKP、TCGETA(获得属性)、TCSETA(设置属性)、终端I/O控制(TIOC)请求：TIOCGSOFTCAR(设置软回车)、TIOCSSOFTCAR(获得软回车)、TIOCSCTTY(设置控制 tty)、TIOCMGET(获得modemline)、TIOCMSET(设置modemline)、TIOCGSERIAL、TIOCSSERIAL、TIOCSERCONFIG、TIOCSERGWILD、TIOCSERSWILD、TIOCSERGSTRUCT、TIOCMBIS、TIOCMBIC、……

2) POSIX方法：tcgetattr( )、tcsetattr( )、tcsendbreak( )、tcdrain( )、tcflush( )、tcflow( )、tcgetpgrp( )、tcsetpgrp( )、cfsetispeed( )、cfgetispeed( )、cfsetospeed( )、cfgetospeed( )

3) 其他方法：对硬件使用 outb和inb，就好像在编程时在没在打印机的情况下使用找印机端口。

9.1　鼠标编程

鼠标或者与串行口相连接，或者直接连接到 AT总线上。不同的鼠标所发送的数据也不同，这使得鼠标编程比较难实现。但是，　Andrew Haylett是个好人，他提供了自己程序集的版权，这意味着用户可以在自己的过程中使用他的鼠标过程。在这部分的内容中，读者可以读到带有版权声明的程序集1.8的预发行版本。x11早已提供了一个优秀的鼠标 API，所以Andrew的过程只运用于非x11的应用程序。

在该程序集服务包中，用户可以使用 mouse.c和mouse.h这两个模块。为了获取鼠标事件，用户只需调用ms_init( )和get_ms_event( )这两个函数。ms_init需要下面这10个变元：

1) int acceleration　它是一个加速因子。如果用户移动鼠标的距离超过 delta个像素，则鼠标移动的速度将取决于这个值的大小。

2) int baud　它是用户鼠标所使用的bps速率(通常为1,200)。

3) int delta 它是像素的数量。鼠标的移距离必须超过这个值，才能启运加速因子进行加速。

4) char *device 它是鼠标设备的名称 (例如，/dev/mouse)。

5) int toggle 在初始化时用于切换DTR或RTS，或者同时切换这两种鼠标modem线。

6) int sample 鼠标的灵敏度(dpi)(通常为100)。

7) mouse_type mouse 鼠标的标识符，笔者的鼠标的标识符是 P_MSC(Mouse Syctems公司)。

8) int slack 它是边框处slack的数量，如果slack为-1，则当鼠标光标在屏幕边框处时，用户再移动鼠标，鼠标将停在边框上；如果 slack的值大于等于0，则当鼠标光标在边框处，用户再把它移动slack个像素位置以后，鼠标光标将跳转到另一端。

9) int maxx 用户当前终端在x方向上的解析度。在使用默认字体的情况下，字符宽度为10个像素，所以整个屏幕x方向的解析度是10*80-1。

10) int maxy 用户终端在y方向上的解析度，使用默认的字体，一个字符是 12个像素高。所以整个屏幕y方向的解析度是12*25-1。

get_ms_event( )只需要一个参数，即指向结构 ms_event的指针。如果get_ms_event( )返回－1，则函数将出错。在成功时它将返回 0，而结构ms_event中将包含实际的鼠标状态。

9.2 调制解调器编程

请参见实例miniterm.c。

使用termios来控制rs232端口。

使用Hayes命令来控制调制解调器。

9.3 打印机编程

请参见实例checklp.c。

不要使用termios来控制打印机端口。如果需要时要使用 ioctl和inb/outb。

使用Epson、Postscript、PCL等命令来控制打印机。

〈linux/lp.h〉

ioctl调用：LPCHAR、LPTIME 、LPABORT、LPSETIRQ、LPGETIRQ、LPWAIT

用于状态和控制端口的inb/out

9.4 游戏杆编程

请参见游戏杆可装入内核模块服务器包中的实例 js.c。

〈linux/joystick.h〉

ioctl调用：JS_SET_CAL、JS_GET_CAL、JS_SET_TIMEOUT、JS_GET_TIMEPOUT、JS_SET_TIMELIMIT、JS_GET_TIMELIMIT、JS_GET_ALL、JS_SET_ALL。在/dev/jsn上执行一次读操作将返回结构JS_DATA_TYPE。

第10章　把应用程序移植到Linux上

10.1　介绍

把Unix应用程序移植到Linux上是相当容易的。Linux以及它所使用的GNU C库，在设计时就已经充分地考虑到了应用程序的可移植性。这就意味着许多应用程都可以简单地使用make进行编译，只要它们没有利用特定实现的一些不知名特征，并且没有紧密地依赖于未定义或者未写入文档的行为(例如某个特定的系统调用)。

Linux基本上服从IEEE Std 1003.1-1988(POSIX.1)标准，但也并不保证一定都是这样。与此相似，Linux也支持和实现了许多SVIP和BSD Unix中的特征，但是并不是在所有情况下都支持这些特征，一般来说，Linux在设计时已经被设计成与其他Unix实现相兼容，可以让移植应用程序更容易，在许多情况下它改进或纠正了这些实现中的某些行为。

作为一个例子，在poll操作中，Linux会真正减少传送给select系统调用的timeout变元。而其他某些实现可能根本不修改该值。而如果不按照这个规定，应用程序在Linux下编译时可能会崩溃。BSD和SunOS select系统调用提供的man帮助信息说：在“将来的实现中”该系统调用会修改timeout指针。不幸的是，许多应用程序仍然假设那个值不会改变。

本章的目标就是概述有关把应用程序移植到Linux上的问题，重点是介绍Linux、POSIX.1、SVID和BSD在以下方面存在的差异：信号处理、终端I/O、进程控制和消息集成，以及可移植条件编译。

10.2　信号处理

近年来，Unix的不同实现所给出的信号的定义也不相同，信号的语义也有所不同。当前主要有两类信号：可靠的和不可靠的。不可靠信号是指它们的信号处理程序调用以后并不保留安装。如果程序希望此信号保留安装，则这种“单发”信号必须在信号处理程序内部重新安装信号处理程序。正因为如此，我们假设在处理程序重要新安装以前该信号再次到达，这时就构成了一个竞争条件。它的后果是要么丢失了信号，要么激活该信号的原始行为(例如杀死进程)。所以，这些信号是“不可靠的”，因为信号的获取操作和重新安装操作不是原子操作。

在不可靠信号语义下，当被信号中断时，系统调用并不自动重启。所以，为了让程序能处理所有情况，在每一次系统调用以后，程序都需要检查errno的值，如果值为EINTR则重新发出系统调用。

同理可知，不可靠信号语义并不提供获得原子中断操作的简单方法(中断操作是指把进程置为睡眠状态，直到某信号到达为止)。因为重新安装的信号处理程序是不可靠的，所以有些时候信号到达时程序却无动于衷。

另一方面，在可靠信号语义下，信号处理程序在调用时就保留安装，这时重新安装所带

来的竞争条件就避免了。而且，特定的系统调用可以重启，通过 POSIX的sigsuspend函数还可以提供原子中断操作。

10.2.1 SVR4、BSD和POSIX.1下的信号

SCV4所实现的信号提供以下函数：signal、sigset、sighold、sigrelse、sgignore和sigpause。SVR4下的signal函数与典型的Unix V7下的信号相同，只提供不可靠信号。另外的函数确实提供了信号处理程序和自动重新安装，但不支持系统调用的重新启动。

BSD支持函数signal、sigvec、sigblock、sigsetmask和sigpause。所有这些函数均提供可靠信号，系统调用是默认重启的。但如果程序员愿意，他可以关闭这一特征。

POSIX.1提供函数sigaction、sigprocmask、sigpending和sigsuspend。注意这里没有signal函数，因为根据POSIX.1标准，它的价值不大。这些函数提供了可靠信号，但 POSIX没有定义系统调用的重启行为。如果在 SVR4和BSD下使用sigaction，则系统调用的重启在默认情况下是关闭的。但如果指定了信号标志 SA_RESTART，则该特征也可以打开。

所以，在程序中使用信号的最佳方法是使用函数 sigaction，它将允许程序员显式地指定信号处理程序的行为。然而，在许多应用程序中仍然使用 signal函数，而读者可以看到，上面所列出的signal在SVR4和BSD下语义是不同的。

10.2.2 Linux信号选项

在 Linux下，sigaction结构的sa_flags成员定义了如下一些值：

- SA_NOCLDSTOP：当子进程停止执行时，无需发送 SIGCHLD信号。
- SA_RESTART：当被某信号处理程序中断时，强迫特定的系统调用重新启动。
- SA_NOMASK：关闭信号掩码(在信号处理程序执行时它会阻塞信号)。
- SA_ONESHOT：在信号处理程序执行以后清除该处理程序。注意 SVR4使用SA_RESETHAND表示同一个操作。
- SA_INTERRUPT：在Linux下定义，但未使用。在 SunOS下，系统调用是自动重启的，而该标志可以关闭那个行为。
- SA_STACK：当前它被用于信号栈操作。

注意POSIX.1只定义了SA_NOCLDSTOP，而SVR4下定义的许多其他选项在 Linux下均不可用。在移植使用 sigaction的应用程序时，用户可能需要修改 sa_flags的值，以便得到正确的行为。

10.2.3 Linux下的信号

在Linux下，signal函数等价于使用SA_ONESHOT和SA_NOMASK选项来调用sigaction。也就是说，它对应着SVR4所使用的典型的不可靠信号语义。

如果用户想让信号使用 BSD语义，大部分Linux系统都提供了一个与 BSD兼容的库，可以与它链接。为了使用该库，用户需要在编译命令行中加入如下选项：

```
-I/usr/include/bsd -lbsd
```

当移植的应用程序使用信号时，请密切关注程序对于信号处理程序的使用作出什么假设，并修改代码(以正确的定义进行编译)，以获得正确的行为。

10.2.4 Linux支持的信号

Linux几乎支持SVR4、BSD和POSIX所提供的所有信号，但以下几个信号Linux不支持：

- SIGEMT不被支持，在SVR4和BSD下它对应着一个硬件错误。
- SIGINFO不被支持，SVR4下它可以用来处理键盘信息请求。
- SIGSYS不被支持。在SVR4和BSD中它指的是非法系统调用。如果用户与libbsd相链接，该信号被重新定义为SIGUNUSED。
- SIGABRT和SIGIOT相同。
- SIGIO、SIGPOLL和SIGURG相同。
- SIGBUS被定义为SIGUNUSED。从技术上讲，在Linux环境中不存在“总线错误”。

10.3 终端I/O

与信号一样，终端I/O控制在SVR4、BSD和POSIX.1下的实现各不相同。

SVR4使用termio结构，以及终端设备上的几个ioctl调用(如TCSETA、TCGETA等等)，以获取和设置termio结构的成员值。该结构定义如下：

```
struct termio {
  unsigned short c_iflag;  /* Input modes */
  unsigned short c_oflag;  /* Output modes */
  unsigned short c_cflag;  /* Control modes */
  unsigned short c_lflag;  /* Line discipline modes */
  char c_line;             /* Line discipline */
  unsigned char c_cc[NCC]; /* Control characters */
};
```

在BSD下，sgtty结构可用于几个ioctl调用中，如TIOCGETP、TIOCSETP等等。

在POSIX下，termios结构是与POSIX.1所定义的几个函数一起使用的。(如tcsetattr和tcgetattr等)。termios结构与SVR4所使用的结构termios相同，但是它们的类型不同(POSIX使用的类型为tcflag_t，而不再是unsigned short)，而NCCS则用于c_cc数组的大小。

在Linux下，POSIX.1 termios和SVR4 termio都直接受内核支持。这意味着如果用户使用这两种方法之一来访问终端I/O，它应该直接在Linux下编译。如果读者有什么疑问，可以很容易地把使用termio的代码修改为使用termios，这种修改只需对这两种方法稍有了解即可。然而用户永远不会需要作这种修改。但是，如果程序要在termio结构中使用c_line，用户一定要小心。对于几乎所有的应用程序来说，它应该是N_TTY，如果程序假定其他一些行规则也可用，则程序员将陷入困境。

如果用户的程序使用BSD sgtty实现，用户可以像上面介绍的那样与libbsd.a进行链接。这将提供ioctl的一个替代品，它将以内核使用的POSIX termios调用重新提交终端I/O请求。在编译这样的程序时，如果没有定义TIOCGETP等符号，则用户需要链接libbsd。

10.4 进程信息和控制

系统必须向程序(如ps、top和free)提供一些方法，以便它们从内核获取有关进程和系统资源的信息。类似地，调试器和其他类似的工具也需要有控制和监察运行进程的能力。不同版

本的Uinx通过许多接口提供了这些特征，几乎所有这些特征都是与计算机相关的，或者紧密地和特定内核设计联系在一起的。到目前为止，对于这种进程与内核之间的交互，还没有一种广泛接受的接口。

10.4.1 kvm过程

许多系统使用kvm_open、kvm_nlist和kvm_read等过程来直接访问内核数据结构，这些访问是通过/dev/ kmem设备进行的。一般来说，这些程序将打开/dev/ kmem，读入内核的符号表，使用该表查找运行的内核中的数据，并使用过程读取内核地址空间中适当的地址。因为这将需要用户程序和内核同意以这种方式读入的数据结构的大小和格式，所以这样的程序在移植时，对每次内核变动以及CPU类型变化，它都需要重新建立。

10.4.2 ptrace和/proc文件系统

在4.3BSD和SVID中，使用ptreace系统调用可以控制进程，并从该进程读取数据。该系统调用通常由调试器使用，以捕获运行进程的执行，或测试其状态。在 SVR4下，ptrace被/proc文件系统所取代，/proc文件系统看上去就象一个目录，每个运行进程在它里面都有一个对应的文件项目，该项目的名称就是运行进程的 ID号。用户程序可以打开感兴趣的进程所对应的文件，并在它上面发出几个 ioctl调用，以控制它的执行，或者从内核获取进程的信息。相同的道理 ，程序可以通过文件描述符从 /proc文件系统读取进程地址空间中的数据，或者把进程地址空间的数据写入到/proc文件系统中。

10.4.3 Linux下的进程控制

在 Linux下，ptrace系统调用可用于进程控制，它的工作过程与 4.3BSD中类似。为了获取进程和系统信息， Linux还提供了/proc文件系统，但它的语义有很大的差别，在 Linux下，/proc包括许多文件，这些文件可以提供通用系统信息，如内存使用、负载平均、装入模块统计，以及网络统计等。这些文件一般是通过 read和write来访问的，它们的内容可以使用 scanf来过滤。 Linux的/proc文件系统还为每个运行进程都提供了一个目录项，它的名称就是进程ID，它包含的文件项目存放如下信息：命令行、到当前工作目录和可执行文件的链接，以及打开的文件描述符等等。内核随时提供所有这些信息，以响应 read请求。这一实现与Plan 9中的/proc文件系统相似，但它确实也有缺点，例如，对于工具 ps(它的作用是列出所有运行进程的信息列表)来说，需要使用许多目录，打开并且读许多文件。相比较而言，在其他 Unix系统上，kvm过程只需要几个系统调用就能直接读内核数据。

很明显，因为每个实现的差别如此之大，移植那些使用它们的应用程序将会是个很困难的工作。必须指出的是， SVR4的/proc文件系统与Linux中的/proc差别是非常大的，它们不能用在同一个上下文中。一般来说，使用 kvm过程或者SVR4/proc文件系统的任何程序都不是真正可移植的，这些代码段对每个操作系统都需要重写。

Linux ptrace调用与BSD中的ptrace几乎相同，它们存在以下几点差异：

- BSD下的请求PTRACE_PEEKUSER在Linux下的名称是PTRACE_PEEKUSR，而BSD下的请求PTRACE_POKEUSER在Linux下的名称则是PTRACE_POKEUSR。
- 进程注册程序可以调用 PTRACE_POKEUSR请求，带上/usr/include/linux/ptrace.h中的偏

移量来进行设置。

- 不支持SunOS请求PTRACE_{READ，WRITE}{TEXT，DATA}，同时也不支持PTRACE_SETACBKPT，PTRACE-SETWRBKPT，PTRACE_CLRBKPT，或者PTRACE_DUMPCORE。这些请求只会对少量现有程序造成影响。

Linux并不提供kvm过程来从用户程序中读取内核地址空间，但有些程序 (最重要的是kmem_ps)实现了这些过程。一般来说，它们不是可移植的。使用 kvm过程的任意代码可能需要依赖于内核中特定符号或数据结构—但作出这样的假设是不可靠和不安全的。使用 kvm过程应被视为是系统结构相关的。

10.5 可移植条件编译

如果用户需要对现有代码进行修改，以便把它移植到 Linux下，用户可以使用ifdef…endif对，用它们包含与Linux相关的代码部分—或者对应到其他实现上的代码。关于如何选择基于操作系统进行编译的代码部分，目前还没有实际的标准，但大多数程序都使用这样一个约定，为系统V的代码定义SVR4，为BSD代码定义BSD，并为与Linux相关的代码定义 linux。

Linux所使用的GNU C库允许用户在编译时定义各种宏，以便打开该库的各种特征。这些宏如下所示：

- _ _STRICT_ANSI_ _：只对ANSI C特征。
- _POSIX_SOURCE：对POSIX.1特征。
- _POSIX_C_SOURCE：如果定义为 1，则对应 POSIX.1特征；如果定义为 2，则对应POSIX.2特征。
- _BSD_SOURCE：ANSI、POSIX和BSD特征。
- _SVID_SOURCE：ANSI、POSIX和系统V特征。
- _GNU_SOURCE：ANSI、POSIX、BSD、SVID和GNU扩展。如果以上这些宏均来定义，则它是默认值。

如果用户自己定义了 _BSD_SOURCE，则还需为库定义一个 _FAVOR_BSD定义。这将导致某些代码会选择BSD行为，而不选择POSIX或SVR4。例如，如果定义了 _FAVOR_BSD，则setjmp和longjmp将保存和恢复信号掩码，而 getpgrp将接收一个PID变元。注意，用户必须链接libbsd，以便使本章前面曾经介绍过的特征可以采取 BSD式的行为。

在Linux下，gcc自动定义了大量的宏，用户可以在程序中使用这些宏。它们是：

- _ _GNUC_ _(GNU C的主版本号，例如2)
- _ _GNUC_MINOR_ _(GNU C的次版本号，例如5)
- unix
- i386
- linux
- _ _ unix_ _
- _ _i386_ _
- _ _linux_ _
- _ _unix
- _ _i386

• _ _linux

许多程序使用#ifdef linux来包围与Linux相关的代码。通过使用这些编译时的宏，用户可以轻易地调整现有代码，加入或者剔除所需的修改，以便把程序移植到 Linux下。注意，因为Linux一般支持比较多的系统 V形式的特征。所以在为系统 V和为BSD所写的程序中，最好是使用系统V版本的代码进行移植，另外，用户也可以从基于 BSD的代码移植，并链接到libbsd。

10.6 补充说明

本章介绍了大部分移植过程中会遇到的问题，但没有介绍系统调用的丢失和流的丢失。前者在系统调用一章中介绍过；有人认为可装入的流模块应该位于 pub/systems/linux/isdn中的ftp.uni_stuttgart.de。

附录　以字母顺序排列的系统调用

表A-1　以字母顺排列的系统调用

系统调用	描　述
_exit	与exit相似，但动作较少(m+c)
accept	接受套接字上的联接(m+c!)
access	检查用户对某文件的许可权限(m+c)
acct	尚未实现(mc)
adjtimex	设置/获取内核时间变量(-c)
afs_syscall	保留的andrew文件系统调用(-)
alarm	在某特定时刻发送SIGALRM(m+c)
bdflush	把某个污染缓冲区刷新到磁盘上(-c)
bind	为进程间通信命名一个套接字(m!c)
break	尚未实现(一)
brk	改变数据段的大小(mc)
chdir	改变工作目录(m+c)
chmod	改变文件属性(m+c)
chown	改变文件所有权(m+c)
chroot	设置新的根目录(mc)
clone	参见fork(m-)
close	通过调用关闭一个文件(m+c)
connect	连接两个套接字(m!c)
creat	创建文件(m+c)
create_module	为可装入内核模块分配空间(-)
delete_module	卸载一个内核模块(-)
dup	创建文件描述符复制(m+c)
dup2	复制文件描述符(m+c)
execl、execlp、execle, …	参见execve(m+!c)
execve	执行某文件(m+c)
exit	终止一个程序(m+c)
fchdir	通过调用改变工作目录
fchmod	参见chmod(mc)
fchown	改变文件的所有权(mc)
fclose	通过调用关闭文件(m+!c)
fcntl	文件/文件描述符控制
flock	改变文件锁定(m!c)
fork	创建字进程(m+c)
fpathconf	通过调用获取文件的有关信息(m+!c)
fread	从流中读取二进制数据的数组(m+!c)
fstat	获取文件状态(m+c)
fstatfs	通过调用获取文件系统状态(mc)
fsync	把文件高速缓存写到磁盘上(mc)
ftime	获取从1970年1月1日以来的时区十秒数信息
ftruncate	改变文件大小(mc)

(续)

系统调用	描 述
fwrite	把二进制数据的数组写入流中(m+!c)
get_kernel_syms	获取内核符号表或它的大小
getdomainname	获取系统的域名(m!c)
getdtablesize	获取文件描述符表的大小(m!c)
getegid	获取有效的组id(m+c)
geteuid	获取有效的用id(m+c)
getgid	获取真正的组id(m+c)
getgroups	获取补充组(m+c)
gethostid	获取唯一的主机标识符(m!c)
gethostname	获取系统主机名(m!c)
getitimer	获取间隔定时器的值(mc)
getpagesize	获取系统页的大小(m-!c)
getpeername	获取相连接的同等套接字的地址(m!c)
getpgid	获取某进程的父进程的组id(+c)
getpgrp	获取当前进程的父进程的组id(m+c)
getpid	获取当前进程的进程id(m+c)
getppid	获取父进程的进程id(m+c)
getpriority	获取进程/组/用户的伏先级(mc)
getrlimit	获取资源限制(mc)
getrusage	获取资源的利用率(m)
getsockname	获取套接字的地址(m!c)
getsockopt	获取套接字的选项设置(m!c)
gettimeofday	获取1970年1月1日以来的时区十秒数信息(mc)
getuid	获取真正的uid(m+c)
getty	尚未实现()
idle	使进程成为可以交换的候选进程(mc)
init_module	插入一个可装入的内核模块(-)
ioctl	操作字符设备(mc)
ioperm	设置一些I/O端口的许可权限(m-c)
iopl	设置所有I/O端口的许可权限(m-c)
ipc	进程间通信(-c)
kill	向进程发送信号(m+c)
killpg	向进程组发送信号(mc!)
klog	参见syslog(-!)
link	为现有的文件创建硬连接(m+c)
listen	监听套接字连接(m!c)
llseek	大型文件所使用的lseek(-)
lock	尚未实现( )
lseek	改变某文件描述符的指针的位置(m+c)
lstat	获取文件状态(mc)
mkdir	创建目录(m+c)
mknod	创建设备(mc)
mmap	把文件映射到内存(mc)
modify_ldt	读或写本地描述符表(-)
mount	挂装一个文件系统(mc)
mprotect	读、写或执行保护内存(-)
mpx	尚未实现( )

(续)

系统调用	描 述
msgctl	ipc消息控制(m!c)
msgget	获取一个ipc消息队列的id(m!c)
msgrcv	接收一个ipc消息(m!c)
msgsnd	发送ipc消息(m!c)
munmap	从内存取某消息的文件映射(mc)
nice	改变进程优先级(mc)
oldfstat	不再使用
oldlstat	不再使用
oldolduname	不再使用
oldstat	不再使用
olduname	不再使用
open	打开文件(m+c)
pathconf	获取文件的有关信息(m+!c)
pause	睡眠，直到信号到达为止(m+c)
personality	改变lbcs当前执行域(-)
phys	尚未实现(m)
pipe	创建管道(m+c)
prof	尚未实现( )
profil	执行时间配置(m!c)
ptrace	跟踪子进程(mc)
quotactl	尚未实现
read	从文件中读数据(m+c)
readv	从文件读数据块(m!c)
readdir	读目录(m+c)
readlink	获取符号连接的内容(mc)
reboot	重启(-mc)
recv	从相连接的套接字接收消息(m!c)
recvfrom	从套接字接收消息(m!c)
rename	删除或者重命名一个文件(m+c)
rmdir	删除一个空目录(m+c)
sbrk	参见brk(mc!)
select	睡眠，直到在文件描述符上执行一个动作(mc)
semctl	ipc信号量控制(m!c)
semget	ipc获取信号量集合标识符(m!c)
semop	在信号量集合成员上执行的ipc操作(m!c)
send	把消息发送到相连接的套接字(m!c)
sendto	把消息发送到套接字(m!c)
setdomainname	设置系统的域名(mc)
setfsgid	设置文件系统组id( )
setfsuid	设置文件系统用户id( )
setgid	设置真正的组id(m+c)
setgroups	设置补充组(mc)
sethostid	设置唯一的主机标识符(mc)
sethostname	设置系统的主机名(mc)
setitimer	设置间隔定时器(mc)
setpgid	设置进程的组id(m+c)
setpgrp	没有效果(mc!)

(续)

系统调用	描 述
setpriority	设置进程/组/用户的优先级(mc)
setregid	设置真正的和有效的组id(mc)
setreuid	设置真正的和有效的用户id(mc)
setrlimit	设置资源限制(mc)
setsid	创建会话(+c)
setsockopt	改变进程的选项(mc)
settimeofday	设置自1970年1月1日以来的时区十秒数信息
setuid	设置真正的用户id(m+c)
setup	初始化设备并挂装根目录(-)
sgetmask	参见siggetmask(m)
shmat	把共享内存连接到数据段上(m!c)
shmctl	ipc操作共享内存(m!c)
shmdt	从数据段上断开共享内存的连接(m!c)
shmget	获取/创建共享内存段(m!c)
shutdown	关闭套接字(m!c)
sigaction	设置/获取信号处理程序(m+c)
sigblock	阻塞信号(m!c)
siggetmask	获取当前进程的信号阻塞(!c)
signal	设置信号处理程序(mc)
sigpause	使用新的信号掩码；直到到达一个信号(mc)
sigpending	获取追加的并且是阻塞的信号(m+c)
sigprocmask	设置/获取当前进程的信号阻塞(+c)
sigreturn	尚未使用( )
sigsetmask	设置当前进程的信号阻塞(c!)
sigsuspend	取代sigpause(m+c)
sigvec	参见sigaction(m!)
socket	创建套接字通信端点(m!c)
socketcall	套接字调用的组合(-)
socketpair	创建两个相互连接的套接字(m!c)
ssetmask	参见sigsetmask(m)
stat	获取文件状态(m+c)
statfs	获取文件系统状态(mc)
stime	设置1970年1月1日以来的秒数(mc)
stty	尚未实现
swapoff	停止交换到文件/设备中(m-c)
swapon	开始交换到文件/设备中(m-c)
symlink	创建到某文件的符号链接(m+c)
sync	同步内存和磁盘缓冲区(mc)
syscall	按编号执行系统调用(-!c)
sysconf	获取某系统变量的值(m+!c)
sysfs	获取配置的文件系统的有关信息( )
sysinfo	获得Linux系统的信息(m-)
syslog	操作系统登录(m-c)
system	执行shell命令(m!c)
time	获取自1970年1月1日以来的秒数(m+c)
times	获取进程时间(m+c)
truncate	改变文件大小(mc)

(续)

系统调用	描 述
ulimit	获取/设置文件限制(c!)
umask	设置文件创建掩码(m+c)
umount	挂装文件系统(mc)
uname	获取系统信息(m+c)
unlink	在不忙时删除某文件(m+c)
uselib	使用共享库(m-c)
ustat	尚未实现(c)
utime	修改索引节点时间项(m+c)
utimes	参见utime(m!c)
vfork	参见fork(m!c)
vhangup	可视地挂起当前tty(m-c)
vm86	进入虚拟的8086模式(m-c)
wait	等待进程中止(m+!c)
wait3	bsd下等待特定进程(m+!c)
wait4	bsd下等待特定进程(mc)
waitpid	等待特定进程(m+c)
write	把数据写到文件中(m+c)
wretev	把数据块写到文件中(m!c)

说明

- (m) 存在对应的man帮助页。
- (+) 服从POSIX的规定。
- (-) Linux相关。
- (c) 在libc中。
- (!) 不是一个单独的系统调用，需要使用另一个系统调用。

第1章 系统结构

1.1 系统概述

Linux内核本身作为一个独立的东西是没有什么用的；它只是参与了一个更大的系统，成为那个系统的一部分，而该系统从整体上看是非常有用的。因此，在整个系统的上下文中介绍内核的作用就显得很有意义了。图 4-1-1显示了整个Linux操作系统的结构分析。

Linux操作系统由4个主要的子系统所组成：

- 用户应用程序 —— 在某个特定的Linux系统上运行的应用程序集合，它将随着该计算机系统的用途不同而有所变化，但一般会包括文字处理应用程序和 Web浏览器。
- O/S服务 —— 这些服务一般认为是操作系统的一部分（开窗系统，命令外壳程序，等等）；此外，内核的编程接口（编译工 具和库）也属于这个子系统。
- Linux内核 —— 这是本文的关注焦点；包括内核抽象和对硬件资源（如CPU）的间接访问。
- 硬件控制器 —— 这个子系统包含在Linux实现中所有可能的物理设备，例如，CPU、内存硬件、硬盘以及网络硬件等都是这个系统的成员。

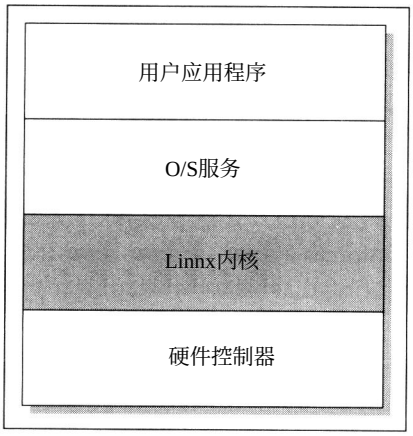


图4-1-1 Linux系统可分为4个子系统

这个系统划分方法是照搬 Garlan和Shaw在[Garlan 1994]中介绍的分层类型。每个子系统层都只能与跟它相邻的层通信。此外，子系统之间的依赖关系是从上到下的：靠上的层依赖于靠下的层，但靠下的层并不依赖于靠上的层。

因为本文的重点是 Linux内核，所以本文将完全不介绍用户应用程序子系统，对硬件和O/S服务也只考虑它们与Linux内核子系统的接口。

1.2 内核的目标

Linux内核向用户进程提供了一个虚拟机器接口。编写进程的时候并不需要知道计算机上安装了哪些物理硬件 —— Linux内核会把所有的硬件抽象成统一的虚拟接口。此外， Linux以对用户透明的方式支持多任务：每个进程工作时就象它是计算机上唯一的进程，好象是独占使用了主存和其他硬件资源一样。内核实际上同时运行许多个进程，并负责对硬件资源的间接访问，这样可以保证各个进程访问的公平性，并保证进程间的安全性。

1.3 内核结构的概述

Linux内核由5个主要的子系统构成：

1. 进程调度程序（SCHED）负责控制进程访问CPU。调度程序所使用的策略可以保证进程能够公平地访问CPU，同时保证内核可以准时执行一些必需的硬件操作。

2. 内核管理程序（MM）使多个进程可以安全地共享机器的主存系统。此外，内核管理程序支持虚拟内存。虚拟内存使得Linux可以支持进程使用超过系统中的内存数量的内存。暂时用不着的存储信息可以交换出内存，存放到使用文件系统的永久性存储器上，然后在需要它们的时候再交换回来。

3. 虚拟文件系统（VFS）。通过提供一个所有设备的公共文件接口， VFS抽象了不同硬件设备的细节。此外， VFS支持与其他操作系统兼容的不同的文件系统格式。

4. 网络接口（NET）提供了对许多建网标准和网络硬件的访问。

5. 进程间通信（IPC）子系统为单个Linux系统上进程与进程之间的通信提供了一些机制。

图4-1-2是Linux内核的高级分解，图中线段的箭头是从依赖别人的子系统指向被依赖的子系统。

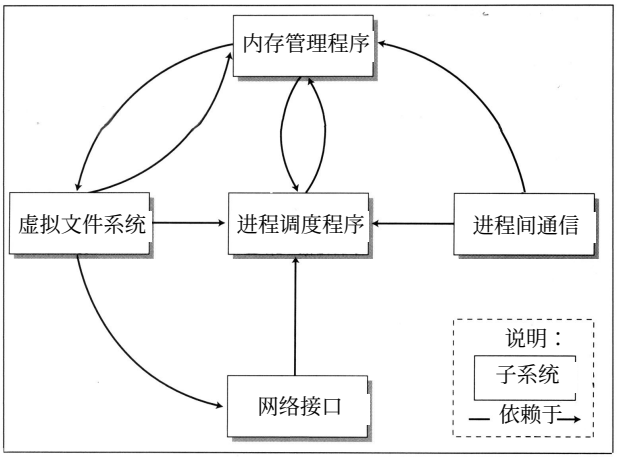


图4-1-2 核心子系统

从图中可以看出，最中心最重要的子系统是进程调度程序：其他所有的子系统都依赖于进程调度程序，这是因为所有的子系统都需要中断和恢复进程的执行。一般来说，子系统会中断那些等待硬件操作完成的进程，同时恢复那些操作已经完成了的进程。例如，当某进程试图通过网络发送消息时，网络接口可能需要中断该进程，直到硬件成功完成了消息的发送。当消息发送完以后（或者硬件返回出错信号），则网络接口将用返回码来恢复该进程。返回码显示了操作是成功完成还是失败。别的子系统（内存管理程序、虚拟文件系统、以及进程间通信）都是由于相似的原因而必须依赖于进程调度程序。

相比之下，其他的依赖关系就不那样明显了，但是它们也同样重要：

- 在进程恢复执行时，进程调度程序将使用内程管理程序来调整硬件内存映射。
- 进程间通信子系统依赖于内存管理程序来支持共享内存通信机制。进程除了可以访问它们通常的私有内存外，共享内存通信机制将使它们可以访问一个公共的内存区。
- 虚拟文件系统使用网络接口来支持网络文件系统（ NFS），它还使用内存管理程序来提供ramdisk设备。
- 内存管理程序使用虚拟文件系统来支持交换。这是内存管理程序之所以依赖于进程调度程序的唯一原因。当某进程访问的内存当前已经被交换出内存时，内存管理程序请求文件系统从永久性存储设备中去取该内存，并中断该进程。

除了显式的依赖关系以外，内核中所有的子系统还依赖于一些在任何子系统中都没有显示出来的公共资源。它们包括：所有内核子系统用于分配和释放内核所使用内存的过程，打

印警告或错误信息的过程，以及系统调试过程等等。这些资源之所以称之为非显式的，是因为它们只能在图1-1所示的内核层中使用。

这一级别的系统结构类型类似于Garlan和Shaw在[Garlan 1994]中所讨论的数据抽象类型。这里描述的每一个系统都包含状态信息，这些状态信息可以使用过程接口来访问，而每个子系统都需要维护它们所管理的资源的完整性。

1.4 支持多个开发人员

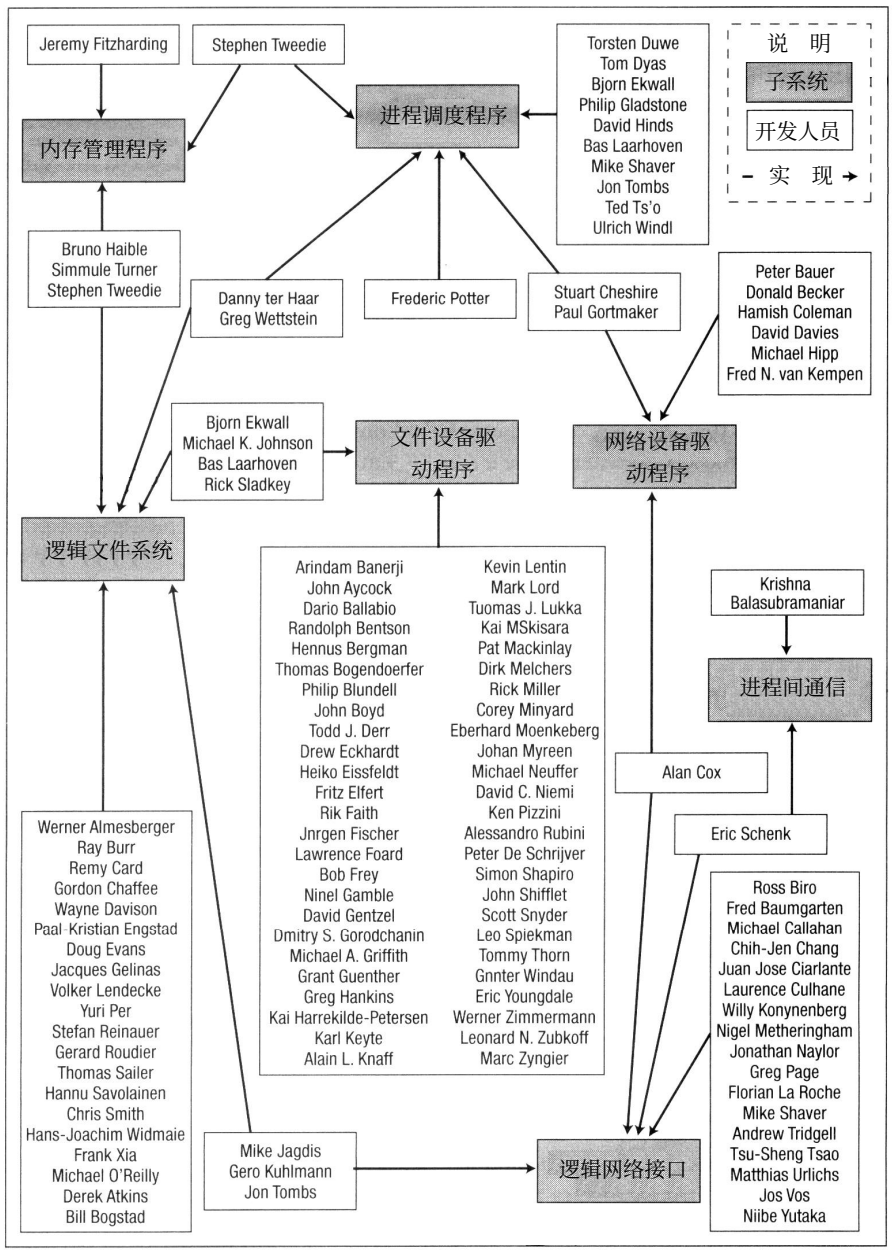


图4-1-3 开发人员职责划分图

Linux系统是由大量的志愿者所开发的（当前 CREDITS文件列举出了196个曾经进行过Linux系统开发的开发人员）。开发人员非常多，而且他们都是志愿者，这势必会影响到系统的构造。因为开发人员在地域上分布上很广泛，所以很难开发出一个紧耦合的系统来 —— 开发人员常常需要阅读别人的代码。因此， Linux系统系统结构的原则是：把那些可能需要作很大修改的子系统 —— 文件系统、硬件接口和网络系统 —— 设计成高度模块化的系统。例如，一个 Linux实现可能需要支持具有不同接口的许多硬件设备，一种朴实的系统结构是把所有硬件设备的实现做在一个子系统中。而一种支持多个开发人员的更好的方法是：把每个硬件设备的代码分离成一个设备驱动程序，这些驱动程序是文件系统中独立的模块。

图4-1-3列出了曾经在Linux内核开发方面作出了贡献的大部分开发人员，以及他们已经实现了的区域。一些开发人员修改了内核的很多部分。为了描述方便，我们没有列出这些开发人员。例如， Linus Torvalds是大部分内核子系统的实现者，尽管后续的开发是由其他人完成的。该图不能做到十分精确，因为在内核开发过程中，开发人员的组成经常变动，但从图中基本可以看出哪些系统是开发人员花费了大量精力去实现的。

该图重新确认了前面描述过的内核的大规模结构。很有趣的是，很少有开发人员曾经在多个系统上工作过，但这样的人的确存在，它主要发生在那些存在子系统依赖性的多个子系统上。这种组织方式符合 Melvin Conway所提出的拇指规则（参见 [Raymond 1993]），即系统的组织应该反映开发人员的组织。大部分开发人员都在开发硬件设备驱动程序、逻辑文件系统模块、网络设备驱动程序以及网络协议模块。内核的这4个区域的系统结构是最具有扩展性的，这一点也不奇怪。

1.5 系统数据结构

1.5.1 任务列表

进程调度程序为每个活跃的进程维护一块数据。这些数据块是存储在链表中的，该链表称为任务列表。进程调度程序常常维护一个当前指针，该指针显示当前的活跃进程。

1.5.2 内存映射

内存管理程序为每个进程都存储了一个从虚拟地址到物理地址的映射表，它还存储了一些有关如何取和替换特定页面的其他信息。这一信息存放在内存映射数据结构中，该结构存放在进程调度程序的任务列表中。

1.5.3 索引节点

在逻辑文件系统上，虚拟文件系统使用索引节点来表示文件。索引节点数据结构存放着从文件块编号到物理设备地址的映射。如果两个进程打开了同一个文件，则这两个进程可以共享索引节点数据结构。这一共享是通过指向同一个索引节点的任务数据块来完成的。

1.5.4 数据连接

所有的数据结构都保存在进程调度程序的任务列表中。系统的每个进程都会有一个数据结构，该数据结构中包含着指向内存映射信息的指针，以及指向代表所有打开文件的索引节点的指针。最后，任务数据结构还包含着指向一些数据结构的指针，这些数据结构代表了与每个任务相联系的打开的网络连接。

第2章　子系统的系统结构

2.1　进程调度程序系统结构

2.1.1　目标

进程调度程序是Linux内核中最重要的子系统。它的目标是控制对计算机 CPU的访问，这里不仅包括用户进程的访问，还包括其他内核子系统的访问。

2.1.2　模块

调度程序可以划分为三个主要的模块：

1) 调度策略模块负责判定哪个进程对 CPU有访问权；策略的设计应该使进程可以公平地访问CPU。

2) 系统结构相关的模块一般使用公共接口设计而成，这样可以抽象出任意特定的计算机系统结构的细节。这些模块负责与 CPU的通信，以中断或者恢复进程的执行。这些操作包括需要为每个进程保留哪些寄存器和状态信息，以及执行汇编代码来影响中断或者恢复操作。

3) 独立于系统结构的模块与策略模块互相通信，来确定接下来执行哪个进程，然后调用系统结构相关的模块来恢复适当的进程。此外，这个模块调用内存管理程序，来确保已经为被恢复的进程准备好了内存硬件，如图 4-2-1所示。

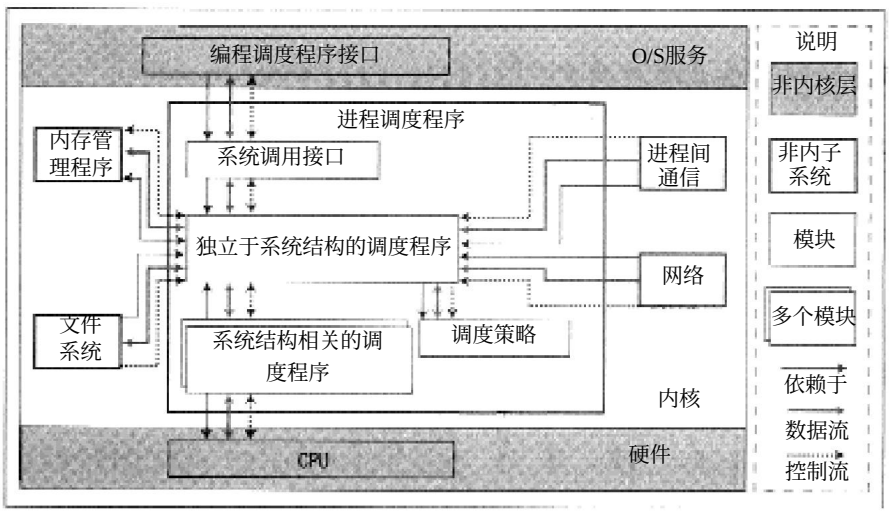


图4-2-1　上下文中的进程调度程度子系统

系统调用接口模块只允许用户进程访问由内核显式打开的那些资源。这就决定了用户进程必须依赖于内核能提供一个很少变动的且定义良好的用户接口，而不管其他的内核模块在

实现上怎样变化。

2.1.3 数据表达

调度程序维护一个数据结构 —— 任务列表。其中每个项目对应一个活跃进程，这个数据结构包含足够的信息，可以中断和恢复进程的执行，但同时也包含了一些记帐和状态信息。在整个内核层中都可以用到这个数据结构。

2.1.4 依赖性、数据流和控制流

正如前面所说的那样，进程调度程序调用内存管理程序子系统。因此，进程调度程序依赖于内存管理子系统。此外，其他所有的内核子系统也都依赖于进程调度程序，在等待硬件请求完成时，可以中断和恢复进程。这些依赖性是通过函数调用以及对共享任务列表数据结构的访问来体现的。所有的内核子系统都会读写代表当前任务的数据结构，这导致了整个系统中数据的单向流动。

除了内核层中的数据流和控制流以外， O/S服务层为用户进程提供了一个接口，以登录定时器通知。这对应于 [Garlan 1994]中所描述的隐式执行系统结构类型。它会引起从调度程序到用户进程的控制流。类似恢复静止进程这样普通的例子不应看作是普通意义上的控制流，因为用户进程不能检测出这个操作。最后，调度程序与 CPU通信，以便中断和恢复进程，这会导致数据流和控制流。CPU负责中断当前执行的进程，并允许内核调度其他的进程。

2.2 内存管理程序系统结构

2.2.1 目标

内存管理程序子系统负责控制进程对硬件内存资源的访问。这是通过硬件内存管理系统来完成的，该系统提供进程内存引用与计算机的物理内存之间的映射。内存管理程序子系统为每个进程都维护一个这样的映射关系，这样，两个进程就可以访问同一个虚拟内存地址，而实际使用的是不同的物理内存位置。此外，内存管理程序子系统支持交换，它把暂时不使用的内存页面移出内存，存放到永久性存储器中，这样计算机就可以支持比物理内存要多的虚拟内存。

2.2.2 模块

内存管理程序子系统由三个模块组成：

1) 系统结构相关的模块提供了一个内存管理硬件的虚拟接口。

2) 系统结构无关的管理程序执行每个进程的映射工作和虚拟内存的交换工作。当出现页面错时，该模块负责判定应该把哪个内存页面移出内存 —— 这里不存在独立的策略模块，这是因为这个策略不太可能会有变化。

3) 内存管理程序子系统还提供了一个系统调用接口，以提供对用户进程的约束访问。该接口允许用户进程分配和释放存储，并可以执行内存映射文件 I/O。

2.2.3 数据表示

内存管理程序为每个进程存储一个从物理地址到虚拟地址的映射。在进程调度程度的任

务列表数据结构中，这一映射关系是以引用的方式存储的。除了这一映射关系以外，数据块中的其他细节也将告诉内存管理程序如何取页面及存储页面。例如，可执行代码可以使用可执行映象作为后援存储，但动态分配的数据必须备份到系统分页文件中。最后，内存管理程序在这个数据结构中存储了许可信息和计帐信息，以便保证系统安全性，如图 4-2-2所示。

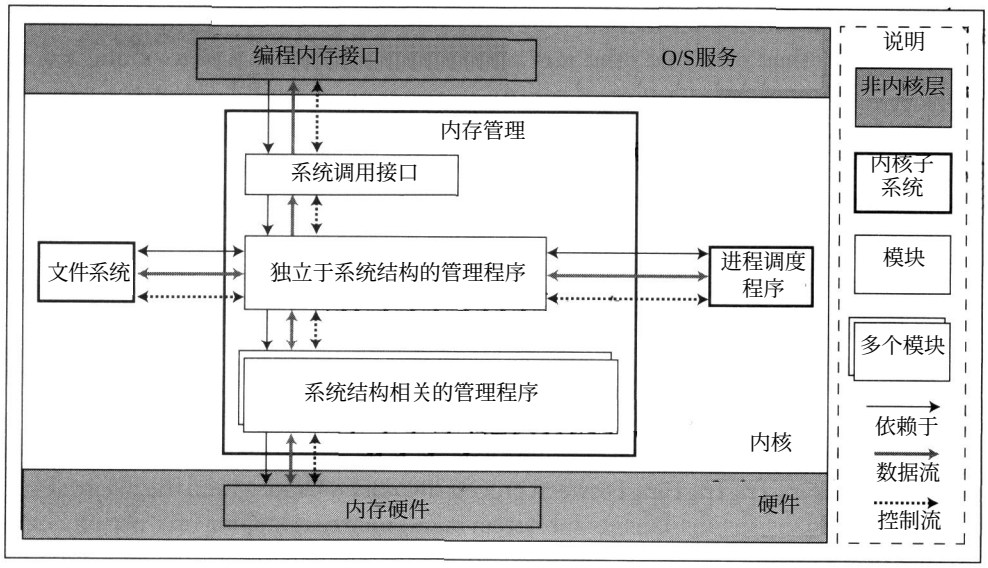


图4-2-2 上下文中的内存管理程序子系统

2.2.4 数据流、控制流和依赖性

上下文中的虚拟文件系统如图 4-2-3所示。

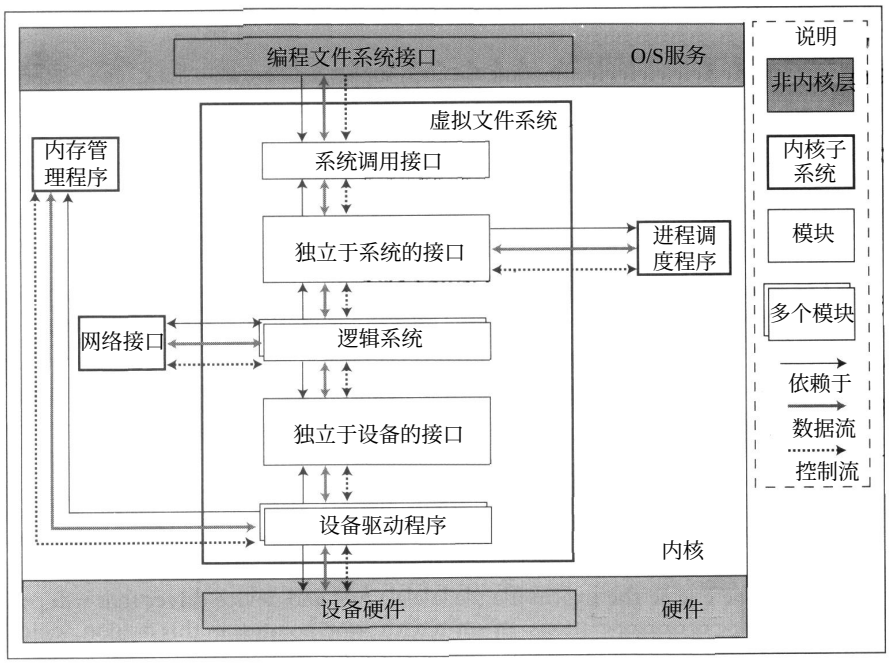


图4-2-3 上下文中的虚拟文件系统

内存管理程序控制内存硬件，在发生页面错时，它从硬件接收到一个通知 —— 这意味着在内存管理程序模块和内存管理程序硬件之间存在单向的数据流和控制流。此外，内存管理程序使用文件系统来支持交换和内存映射 I/O。这就意味着内存管理程序需要向文件系统作过程调用，以便把页面存储到永久性存储器中，或者从永久性存储器中读取页面。因为文件系统请求不能立即完成，所以内存管理程序需要中断某个进程的执行，直到该内存页面被交换回来。这就导致了内存管理程序必须向进程调度程序作过程调用。此外，因为每个进程的内存映射信息是存储在进程调度程序的数据结构中的，所以在内存管理程序和进程调度程序之间存在单向的数据流。用户进程可以在进程的地址空间中建立起新的内存映射关系，并在新映射的区域登记它们，以便发出页错误通知。这又导致了一个从内存管理程序通过系统调用接口模块到用户进程的控制流。用户进程不会发出传统意义上的数据流，但是用户进程使用系统调用接口模块中的系统调用 select，可以从内存管理程序获取一些信息。

2.3 虚拟文件系统系统结构

2.3.1 目标

虚拟文件系统设计的目的是为了提供一个存储在硬件设备上的数据的统一视图。几乎计算机中的所有设备都是使用通用设备驱动程序接口来表示的。而虚拟文件系统则更进一步，允许系统管理员在任意物理设备上挂装任意一个逻辑文件系统的集合，逻辑文件系统提高了与其他操作系统标准的兼容性，并允许开发人员用不同的策略来实现文件系统。这个虚拟文件系统抽象了物理设备和逻辑文件系统的细节，允许用户进程使用公共接口访问文件，而无需知道文件实际驻留在哪个物理或逻辑系统上。

除了传统的文件系统目标以外，虚拟文件系统还负责装入新的可执行程序。这个任务是由逻辑文件系统模块完成的，这就使得 Linux可以支持许多种可执行文件格式。

2.3.2 模块

1) 每个硬件控制器都对应一个设备驱动程序模块。因为存在大量不兼容的硬件设备，所以设备驱动程序的数量也很多。 Linux系统的大多数通用扩展都是增加了新的设备驱动程序。

2) 独立于设备的接口模块提供了所有设备的统一视图。

3) 每个受支持的文件系统都有一个逻辑文件系统模块。

4) 独立于系统的接口提供了硬件资源的视图，该视图与硬件和逻辑文件系统无关。这一模块使用面向块的或面向字符的文件接口来表示所有资源。

5) 最后，系统调用接口为用户进程提供了对文件系统的受控制的访问。这个虚拟文件系统只把特定的功能导出给用户进程。

2.3.3 数据表示

所有的文件都是使用索引节点来表示的，每个索引结点结构都包含着位置信息，以指定文件块在物理设备上的位置。此外，索引节点存储了指向逻辑文件系统中过程的指针，还存储了指向将要执行所需读写操作的设备驱动程序的指针。通过以这种方式存储函数指针，逻辑文件系统和设备驱动程序在内核中注册自己时，就不需要内核依赖于任何特定的模块。

2.3.4 数据流、控制流和依赖性

有个特定的设备驱动程序，它就是 ramdisk，这个设备分配主存的一个区域，并把它当作永久性存储设备来对待。这个设备驱动程序使用内存管理程序来完成它的任务，所以在文件系统设备驱动程序和内存管理程序之间存在依赖性，控制流和数据流。

网络文件系统是受支持的一个特定的逻辑文件系统（仅作为一个客户），该文件系统可以访问另一台计算机的文件，就好象它们是本地计算机的一部分。为了完成这个任务，逻辑文件系统模块需要使用网络子系统来完成它的任务。这就导致在两个子系统之间存在依赖性、数据流和控制流。

正如前面在2.2节中所说的那样，内存管理程序使用虚拟文件系统来完成内存交换和内存映射I/O。此外，在进程等待硬件请求完成时，虚拟文件系统使用进程调度程序来中断进程，而一旦请求完成则立刻恢复它们的执行。最后，系统调用接口允许用户进程调用虚拟文件系统，以便存储或者获取数据。与前一个子系统不同的是，这里没有为用户提供隐式调用的注册机制，所以在虚拟文件系统到用户进程之间没有控制流（恢复进程并不认为是控制流）。

2.4 网络接口系统结构

2.4.1 目标

网络子系统允许Linux系统通过网络与其他系统相连接。因为它支持大量的硬件设备，所以相应地也需要使用大量的网络协议。网络子系统抽象了这两者在实现上的细节，这样用户进程和其他内核子系统在访问网络时就无需知道使用的是什么物理设备和协议了。

2.4.2 模块

上下文中的网络接口子系统如图 4-2-4所示。

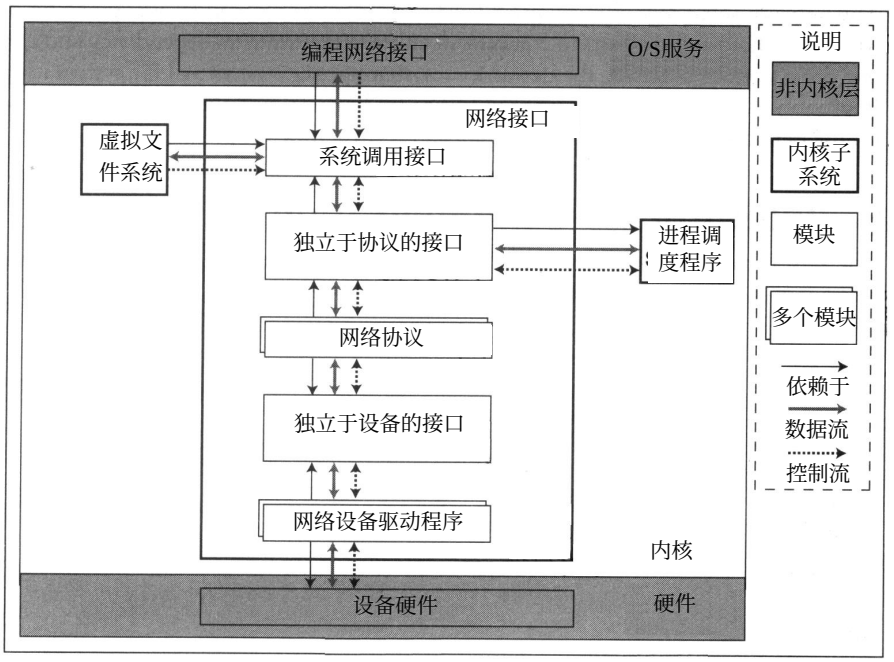


图4-2-4 上下文中的网络接口子系统

1) 网络设备驱动程序与硬件设备通信。每个可能的硬件设备都对应一个设备驱动程序模块。

2) 独立于设备的接口模块为所有的硬件设备提供了一个统一的视图，这样在子系统的高级别上就不需要关于所使用的硬件的特定知识了。

3) 网络协议模块负责实现每个可能的网络传输协议。

4) 独立于协议的接口模块提供了一个独立于硬件设备和网络协议的接口。其他内核子系统可以通过该接口模块来访问网络，而无需依赖于特定的协议或硬件。

最后，系统调用接口限制了用户进程可以访问的导出过程。

2.4.3 数据表示

每个网络对象都表示成一个套接字。套接字与进程的关系类似于索引节点与进程的关系。通过使两个任务数据结构指向同一个套接字数据结构，进程之间可以共享套接字。

2.4.4 数据流、控制流和依赖性

在等待硬件请求完成时，网络子系统使用进程调度程序中断和恢复进程的执行（这导致了依赖性、控制流和数据流）。此外，网络子系统为虚拟文件系统提供了逻辑文件系统（ NFS）的实现，这导致了虚拟文件系统依赖于网络接口，并与它之间存在数据流和控制流。

2.5 进程间通信系统结构

因为这个子系统不如其他子系统有趣，所以本文不介绍进程间通信子系统的系统结构。

第3章 结 论

Linux内核是整个Linux系统的系统结构中的一层。在概念上内核是由5个主要的子系统组成的：进程调度程度、内存管理程序、虚拟文件系统、网络接口和进程间通信接口。这些子系统之间是通过函数调用和共享数据结构相互通信的。

在最高级别上，Linux内核的系统结构类型与Garlan和Shaw在[Garlan 1994]中所描述的数据抽象类型十分接近。内核是由子系统所组成的，而子系统通过使用特定的过程接口来保持内部表示的一致性。通过对每个子系统进行认真的解剖分析，我们可以看出它的系统结构类型与Garlan和Shaw的分层类型十分相似。每一个子系统都是由模块组成的，这些模块只能与相邻层相通信。

Linux内核的概念系统结构已经被证明是非常成功的；而它之所以成功，关键在于它规定了开发人员的组织和系统扩展性。Linux内核系统结构必须支持大量独立的志愿开发人员。这就要求投入工作量最大的系统分区（如硬件设备驱动程序和文件以及网络协议）以一种可扩展的方式来实现。Linux系统结构者为了使这些系统可以扩展，使用了一种数据抽象技术：每个硬件设备驱动程序都是作为一个独立的模块来实现的，而这些模块又都支持一个通用接口。通过这种方式，单个的开发人员可以加入一个新的设备驱动程序，并且不需要与 Linux内核的其他开发人员进行很多交互。内核由大量的志愿开发人员实现成功，这个事实已经证明了这个策略的正确性。

Linux内核的另一个重要扩展是加入了更多受支持的硬件平台。通过把与硬件相关的所有代码分离到每个子系统各自的模块中，系统的系统结构支持了这种扩展性。通过这种方式，一小群开发人员通过重新实现内核的机器相关部分，就可以把 Linux内核移植到新的硬件系统结构下。

附录A 术语定义

设备驱动程序（Device Driver）

设备驱动程序是与特定的硬件设备交互所需要的所有代码。设备驱动程序实际上是内核的一部分，但是Linux内核提供了一种机制，它允许动态装入设备驱动程序。

索引节点（I-Node）

索引节点即index node，可以被文件系统用来跟踪文件系统数据所存储的硬件地址。每个索引节点存储了一个文件块到物理块的映射关系，以及安全性方面以及计帐方面的其他一些信息。

网络文件系统（Network File System, NFS）

网络文件系统是一个文件系统接口，它把远程计算机上存储的文件表示成本地计算机上的文件系统。

进程（Process）

进程（也称为任务）就是执行中的程序，它由可执行代码和动态数据所组成。

内核为每个进程都保留了足够多的信息，以便停止或恢复它的执行。

Ramdisk

Ramdisk是一个设备驱动程序，它把主存的一块区域用作文件系统设备。它允许把经常访问的文件存储在能随时提供有效访问的区域中，在使用 Linux支持硬实时的需求时，这个特征非常有用。对于一般的情况而言，普通的文件系统高速缓存机制将高效地使用内存，以提供对文件的高效访问。

交换

Linux支持进程使用超过计算机上物理内存数量的内存。要做到这一点，内存管理程序必须把暂时不使用的内存页面交换到永久性存储设备中。当以后访问该内存时，再把它交换回主存（这时可能会导致其他页面被交换出去）。

任务（Task）

参见进程。

附录B 参考文献

Garlan 1994

David Garlan和Mary Shaw, An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, 第1卷, World Scientific Publishing Company, 1993。

Monroe 1997

Robert T.Monroe, Andrew Kompanek, Ralph Melton和David Garlan Architectural Styles, Design Patterns and Objects, IEEE Software January 1997, pp43-52。

Paker 1997

Slackware Linux Unleased, Timothy Parker等人所著, Sams Publishing, 201 West 103rd Street, Indianapolis。

Perry 1992

Dewayne E.Perry和Alexander L.Wolf, Foundations for the Study of Software Architecture, ACM SIGSOFT Software Engineering Notes, 17:4, October 1992 pp.40-52。

Raymond 1993

The New hackers Dictionary, 第2版, 由Eric S.Raymond编辑, The MIT Press, Cambridge Massachusetts, 1993。

Rusling 1997

The Linux Kernel David A.Rusling所著, 初稿, 版本 0.1-13(19), ftp://sunsite. unc.edu /pub/Linux/docs/linux-doc-project/linux-kernel/或者 http: //www. linuxhq. com/guides /TLK/index.html。

Soni 1995

Soni,D.; Nord, R.L.; Hofmeister, C., Software Architecture in Industrial Applications, IEEE ICSE 1995, pp.196-210。

Tanenbaum 1992

Modern Operating Systems, Andrew S.Tanenbaum所著, Prentice Hall, 1992。

Wirzenius 1997

Lars Wirzenius所著Linux System Administrators ' Guide 0.6, http://www.iki.fi/liw /linux/sag/ 或者 http://www.linuxhq.com/LDP/LDP/sag/index.html。

第1章 前 言

1.1 目标

本部分的目标是描述Linux内核的具体系统结构。具体系统结构指的是系统创建之后的实际系统结构。我们希望开发具体的系统结构，以提供现有 Linux内核的高级文档资料。

1.2 Linux介绍

Linus B.Torvalds于1991年编写出第一个Linux内核。由于它一直是作为自由软件发布的，所以Linux变得很流行。因为源代码随手可得。用户可以随意改变内核，以使它适应自己的需求。然而，在编写新的系统程序之前，了解 Linux内核的发展过程以及当前它的工作原理是很重要的。

基于Linux内核源代码的具体系统结构可以为 Linux内核高手和开发人员提供一个可靠的和及时的参考。自从 1991年以来，大量的志愿者多次修改过 Linux，他们在因特网上通过Usenet新闻组相通信。过去， Torvalds是主要的内核开发人员。现在 Linux Torvalds已经不再是Linux内核工程小组的成员了。如果能提供准确的和及时的具体系统结构，我们有理由相信Linux会得到进一步修改和进一步发展。

Linux是一个Unix兼容的系统。大部分通用的 Unix工具和程序现在都可以在 Linux下运行。最初设计Linux时是为了它能在Intel 80386微处理器上执行。最初的版本因为使用了 Intel所特有的中断处理例程，所以不能移植到其他的平台上。当把 Linux移植到其他硬件平台（如 DEC Alpha和Sun SPARC）上时，大部分依赖于平台的代码被移入平台相关的模块中，这些模块支持通用接口。

Linux的用户群是巨大的。在1994年，Ed Chi估计Linux大约有40,000个用户（[Chi 1994]）。Linux文档工程小组（LDP）正在开发有用的和可靠的 Linux内核文档，既提供给 Linux用户，也提供给Linux开发人员使用。就我们所知，LDP并不利用逆推机制得到最新的具体系统结构。目前有大量的书和文档资料介绍 Linux内核方面的知识[CS746G Bibliography]。然而，还没有什么文档资料很详细地介绍 Linux的概念和具体系统结构。有些出版物（如 [Beck 1996]和[Rusling 1997]）介绍了Linux内核的工作原理，然而，这些书并没有透彻地分析子系统以及子系统之间的相互依赖性。

1.3 软件系统结构的背景知识

最近以来，在工业和学术团体中，对软件系统结构的研究非常流行。软件系统结构的研究带动了大型软件系统的研究。最近的研究表明软件系统结构是很重要的，因为它增强了系统支持者之间的通信。软件系统结构可以用于帮助开发人员作出一些早期的设计决定。此外，它还可以用作系统的一个可传送的抽象表示（ [Bass 1998]）。

软件系统结构与软件可维护性的研究有关。维护现有的（或者传统的）系统常常是非常

麻烦的。这些现有系统的状态既可能是设计非常好的，文档编制得非常好的；也可能是设计非常差的，文档编制得非常不理想的。在许多情况下，原来的一部分或者全部系统结构者和开发人员不会再参加现有系统的开发工作，而缺乏系统结构实践知识将大大地增加软件维护任务的复杂性和困难程度。为了对现有系统的功能进行变动、扩展、修改或者删除，就必须理解系统的实现原理。这个问题就需要研究从现有系统中抽取系统结构信息和设计信息的相关技术。从源代码抽取高级模型的过程常常称为逆推工程。

逆推工程的方法主要分为两种[Bass 1998]：

1. 技术方法：抽取方法是基于现有的产品抽取有关系统的信息。具体来说，抽取的对象包括源代码、注释、用户文档、可执行模块以及系统描述。

2. 人类知识和推理：这些方法的焦点是人怎样理解软件。一般来说，工作人员常使用下列策略：

- 自顶向下策略：从最高级别的抽象开始，依次进行各个子部分的理解。
- 自底向上策略：先理解最低级别的部件，并理解这些部件是怎样在一起工作来完成系统的目标的。
- 基于模型的策略：理解系统工作的概念模型，并试着深入理解所选定的区域。
- 随机应变策略：综合使用以上的这些方法。

在本部分中，我们同时使用技术方法和人类知识方法来描述 Linux内核的具体系统结构。随机应变的策略与[Tzerpos 1996]中所描述的混合方法实际上是同一个策略。我们没有使用Linux内核开发人员的实践知识，而是使用了现代操作系统的领域相关知识（例如，来自任务1的概念系统结构），用来反复精化Linux内核的具体系统结构。

1.4 方法与途径

在本部分中，我们使用随机应变策略来开发 Linux内核的具体系统结构。我们修改了在[Tzerpos 1996]中所描述的方法，并用它来判断 Linux内核的结构。所采取的步骤如下（但不一定按照这种顺序）：

- 定义概念系统结构。因为我们无法直接获得开发人员的实践知识，所以使用自己的现代操作系统的领域知识来创建 Linux内核的概念系统结构。这个工作是在任务 1中完成的（[Bowman 1998], [Siddiqi 1998]和[Tanuan 1998]）。
- 从源代码中提取事实。我们使用 Portable Bookshelf公司的C Fact Extractor (cfx)和Fact Base Generator (fbgen)（在[holt 1997]中介绍过），从源代码中提取出依赖性事实。
- 集成到子系统。我们使用 Fact Manipulator（例如，grok和grok脚本），把得到的事实集成到子系统中。集成工作有一部分是由工具完成的（使用文件名和目录），另外有一部分是使用Linux内核的概念模型完成的。
- 概览所生成的软件结构。我们使用 Landscape Layouter、Adjuster、Editor和Viewer（[Holt 1997]）来对抽取的设计信息进行可视化。基于这些图表，用户可以清晰地看出子系统之间的依赖性。 Landscape图表可以肯定用户对具体系统结构的理解。有些时候抽取的系统结构与概念系统结构并不吻合，这时我们需要手工检查源代码和文档。
- 使用概念系统结构优化集成工作。我们使用 Linux内核的概念系统结构来检查部件的集成，以及这些部件之间的依赖关系。

- 使用概念系统结构优化布局。延续前一个步骤，我们使用 Visio画图工具手工画出Linux内核结构的布局图。

根据目的和视角的不同， Linux内核有许多视图。在本部分中，我们使用软件结构（[Mancoridis Slides]）来描述具体的系统结构。使用软件结构可以完成下面的工作：

- 指定Linux内核划分为5个主要的子系统。
- 描述了部件（如子系统和模块）的接口。
- 描述了部件之间的依赖关系。

我们描述资源之间的依赖关系，这里所指的资源可以是子系统、模块、过程或者变量。依赖关系是个非常广泛的概念，通常我们不区分函数调用、变量引用和类型使用。

软件结构与Linux内核的运行时结构关系不大。然而，我们相信如果能够把软件结构和详细的规范结合起来，将可以给潜在的 Linux开发人员提供足够的信息，使他们无需读完所有的源代码就能修改或扩展内核。我们主要关心的不是 Linux内核的进程视图[Kruchten 1995]，这是因为我们把Linux内核当作是一个执行进程。

1.5 适用本书的读者

我们假设本书读者在计算机科学和操作系统方面具有足够的背景知识，可以透彻地理解本部分中关于Linux内核的主要部件以及部件之间的交互的讨论。我们并不要求读者对 Linux操作系统有太多的了解。

1.6 本部分的章节安排

本部分的剩下章节是这样安排的：

- 第2章讨论了整个系统的系统结构。它描述了系统的系统结构，展示了它的五个主要子系统，以及子系统之间的相互依赖关系。
- 第3章介绍了主要子系统（进程调度程序、内存管理程序、虚拟文件系统、进程间通信以及网络接口）的系统结构。我们将使用图表来帮助介绍子系统，以展示上下文中的子系统，并用线段来展示依赖关系。第 3章还解释了系统的抽象，以及从 Linux内核源代码抽取出来的设计信息。
- 第4章阐述了在本部分中我们所遇到的问题，并给出我们的发现，得出我们的结论。

第2章 系统结构

Linux内核本身并没有什么用，它只是整个系统中的一个层（[Bowman 1998]）。

在内核层中，Linux是由5个主要的子系统所组成的：进程调度程序（sched）、内存管理程序（mm）、虚拟文件系统（vfs）、网络接口（net）以及进程间通信（ipc）。对于一个创建好的系统结构，它的划分方法与概念系统结构的划分方法类似 [Siddiqi 1998]、[Tanuan 1998]和[Bowman 1998]。如果读者知道概念系统结构是从创建以后的系统结构抽取出来的，那么读者这不难理解上述的对应关系了。这里所采用的分解方法并不完全按照源代码的目录结构，因为我们相信目录结构并不完全匹配子系统的分组。然而，我们的集成是与该目录结构非常相近的。

在把抽取的设计细节可视化以后，读者可以认清一个区别，即子系统的依赖性和概念系统结构的依赖性是有很大不同的。概念系统结构显示出很少的系统间依赖关系，如图 5-2-1a所示（摘自[Bowman 1998]）。

尽管概念系统结构的依赖性相当少，具体系统结构显示出 Linux内核的5个主要的子系统之间存在着很强的依赖性。图 5-2-1-b显示子系统之间的连接，它与一个完全图的区别是只缺两条边（[PBS:kernel.html]详细介绍了哪个模块是跨子系统交互的）。这种相互依赖关系与概念系统结构有着很大的不同。这表明，任何基于互联属性的逆推技术（如在 [Müller 1993]中描述的Rigi系统）在从这样的系统中抽取任何相关结构时，都将会遭到失败。这就证明了Tzerpos（[Tzerpos 1996]）所介绍的混合方法的有效性。

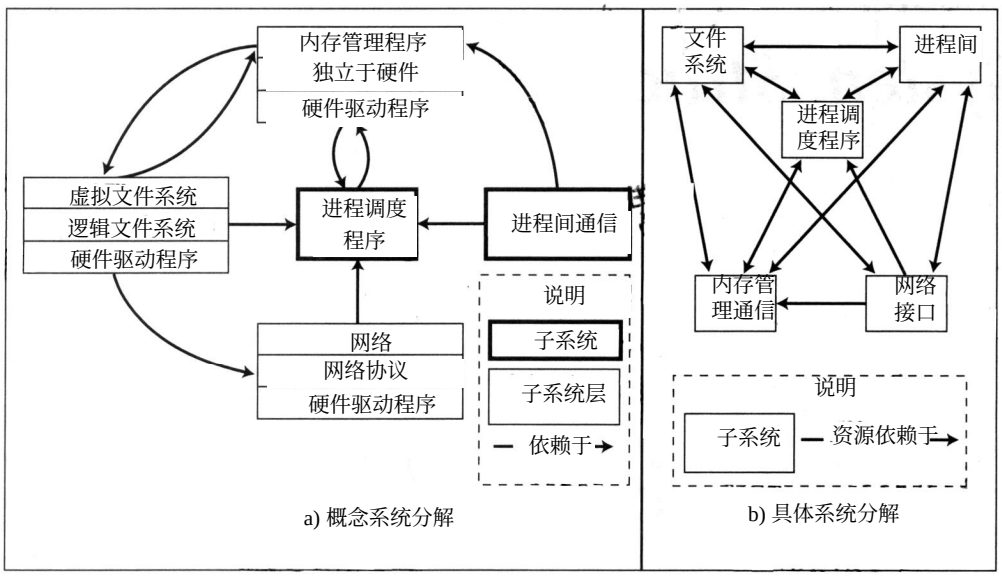


图5-2-1 概念系统的分解和具体系统的分解

系统级别上的差异是子系统级别上差异的延伸。子系统结构大体上对应着概念结构。然而，我们发现具体系统结构中的许多依赖性在概念系统结构中并不存在（在 [Murphy 1995]中称之为散度）。下面一节将会讨论这些附加的依赖关系的原因，在下一节我们将会详细介绍每个主要的子系统的设计。

第3章　子系统结构

3.1　进程调度程序

3.1.1　目标

进程调度程序是Linux操作系统的核心。进程调度程序需要完成以下任务：

- 允许进程创建它们自己的新拷贝。
- 判定哪个进程可以访问CPU，并影响运行进程之间的传输。
- 接收中断，并把它们转交给适当的内核子系统。
- 向用户进程发送信号。
- 管理定时器硬件。
- 当进程结束执行时清除进程资源。

进程调度程序还为动态装入的模块提供支持；动态装入模块是指在内核开始执行之后才能装入的内核功能。虚拟文件系统和网络接口将会用到这个可装入的模块功能。

3.1.2　外部接口

进程调度程序提供了两个接口：首先，它提供了一个可供用户进程调用的受限系统调用接口；其次，它为内核系统的其他部分提供了一个丰富的接口。

进程只能通过拷贝现有进程的方法创建其他的进程。在系统引导的时候， Linux系统只有一个运行进程：init。然后该进程再产生其他的进程，而其他进程又可以利用自身的拷贝产生新进程，方法是使用fork ( )系统调用。fork ( )调用会产生新的子进程，而子进程实际上是父进程的拷贝。在系统关闭的时候，某个用户进程（隐式地或者显式地）调用 _exit系统调用。

系统提供了几个过程来处理可装入模块。 create_module ( )系统调用将分配足够的内存,以便装入模块，该调用将初始化 module结构（下面将会介绍该结构），填入分配的模块的名称、大小、始地址以及初始状态。系统调用 init_module ( )将从磁盘装入模块并激活它。最后delete_module ( )将会卸装一个运行中的模块。

可以用系统调用setitimer ( )和getitimer ( )来完成定时器的管理，前者设置一个定时器，而后者则获取定时器的值。

在重要的信号函数之中，最重要的函数要算是 signal ( )了。这个函数允许用户进程把函数处理程序和特定的信号联系起来。

3.1.3　子系统描述

进程调度程序子系统主要负责用户进程的装入、执行以及正确的结束。在执行用户进程时，一般在两个不同的地方调用这个调度算法。首先，有一些系统调用可以直接调用调度程

序，比如sleep ( )。其次，在每一个系统调用之后，以及在每一个慢的系统中断之后（稍后再作介绍），可以调用调度算法。

信号可以认为是一种IPC机制，所以将在介绍进程间通信的一节中进行讨论。

中断允许硬件与操作系统进行通信。 Linux下的中断分为快速中断和慢中断。慢中断是典型的中断，当系统正在处理慢中断时，其他的中断是合法的，一旦处理完了慢中断， Linux就可以正常工作了，例如可以调用调度算法。定时器中断是慢中断的典型例子。相比之下，快中断所完成的任务要简单一些，例如处理键盘输入等等。在处理快速中断时，其他的中断将被屏蔽，除非快速中断处理程序显式地使能了其他的中断。

Linux OS使用的定时器每10ms发生一次定时器中断，这样，根据前面所讲的调度程序描述，应该至少每10ms发生一次任务调度。

3.1.4 数据结构

结构task_struct代表Linux任务，其中有一个域表示的是进程的状态，它的值可以是如下一些：

- 运行。
- 从系统调用返回。
- 处理中断过程。
- 处理系统调用
- 就绪
- 等待

此外，该结构中还有一个域，它表示进程优先级，另外还有一个域存放着时钟滴答（以10ms为周期）的数量，它表示进程可以持续执行多长时间而不用重新调度。该结构中还有一个域存放着最近一次出错系统调用的错误编号。

为了跟踪所有的执行进程，系统维护一个双链表（通过两个指向 task_struct的域）。因为每一个进程与其他一些进程相关，所以需要提供以下域来描述一个进程：原始父母、父母、最小子进程、弟弟进程，最后是兄长进程。

mm_struct是一个嵌套结构，它包含着进程的内存管理信息（例如代码段的始地址和结束地址）。

进程ID信息也存放在task_struct中。该结构中存放的信息包括进程 ID和组ID。同时还提供了组ID的一个数组，所以进程可以与多个组联系在一起。

文件相关的进程数据存放在 fs_struct子结构中。该结构中存放着一个指针，指向对应着处理器的根目录以及当前工作目录的索引节点。

由进程所打开的所有文件将通过 task_struct的子结构files_struct来跟踪。

最后，还有一些域提供计时信息；例如，进程在用户模式下花费的时间总量。

所有的执行进程在进程表中都有对应的一项。进程表被实现为指向任务结构的指针的一个数组。进程表中的第一个项目是特殊的 init进程，它是Linux系统所执行的第一个进程。

最后，还实现了一个 module结构来表示已经装入的模块。该结构中包含着一些域，可用来实现一个模块结构的列表：一个域指向模块符号表，另一个域存放着模块的名称。 Module结构中还存放着模块的大小（以页面为单位），以及指向模块始地址的指针。

3.1.5 子系统结构

图5-3-1显示了进程调度程序子系统。它以集成的方式来表示进程的调度和管理（例如：装入和卸装），以及定时器管理和模块管理功能。

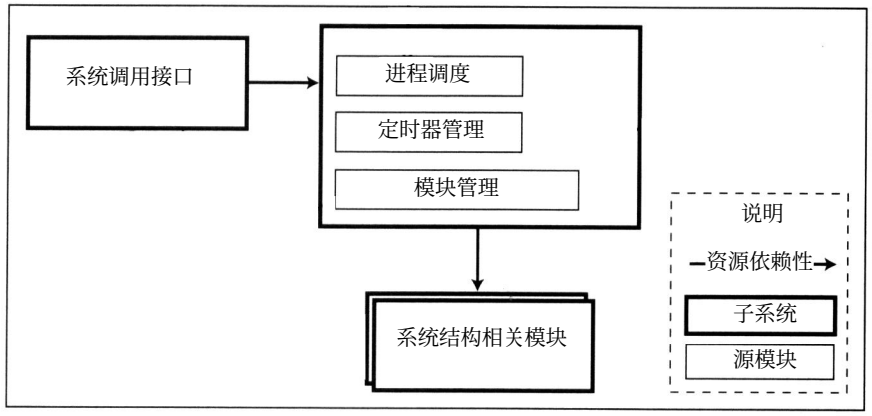


图5-3-1 进程调度程序结构

3.1.6 子系统依赖性

图5-3-2显示出进程调度程序是怎样依赖于其他内核子系统的。在调度进程时，进程调度程序需要内存管理程序来设置内存映射。此外，因为底半操作中（在 3.3 节中介绍）需要用到等待队列，所以进程调度程序依赖于IPC子系统。最后，进程调度程序依赖于文件系统从永久性设备中装入可装入的模块。所有的子系统都依赖于进程调度程序，因为它们需要在完成硬件操作的时候中断用户进程。如果读者希望了解更多关于子系统模块之间特定依赖性的知识，请参见 [PBS:kernel.html]。

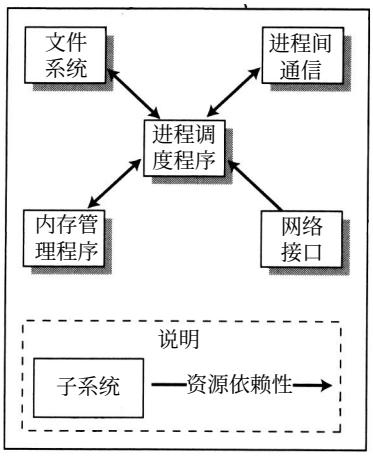


图5-3-2 进程调度程序依赖性

3.2 内存管理程序

3.2.1 目标

正如在[Rusling 1997]第13页到第30页中所介绍的那样，内存管理程序向它的客户提供以下一些功能：

- 大地址空间 —— 用户程序可以使用超过物理上实际所有的内存数量。
- 保护 —— 进程的内存是私有的，不能被其他进程所读取和修改；而且，内存管理程序可以防止进程覆盖代码和只读数据。
- 内存映射 —— 客户可以把一个文件映射到虚拟内存区域，并把该文件当作内存来访问。
- 对物理内存的公平访问 —— 内存管理程序确保所有的进程都能公平地访问计算机的内存资源，这样就可以确保理想的系统性能。

- 共享内存 —— 内存管理程序允许进程共享它们内存的一部分。例如，可执行代码通常可以在进程间共享。

3.2.2 外部接口

内存管理程序向它的功能提供了两个接口：用户进程所使用的系统调用接口，以及其他内核子系统用来完成任务的接口。

- 系统调用接口
- malloc ( )/free ( ) —— 分配或者释放一块供进程使用的内存区域。
- mmap ( )/munmap ( )/msync ( )/mremap ( ) —— 把文件映射到虚拟内存区域。
- mprotect —— 改变对虚拟内存区域的保护。
- mlock ( )/mlockall ( )/munlock ( )/munlockall ( ) —— 超级用户例程，防止内存被交换。
- swapon ( )/swapoff ( ) —— 超级用户例程，可以为系统增加或删除交换文件。
- 内核内部的接口
- kmalloc ( )/kfree ( ) —— 分配或者释放由内核的数据结构所使用的内存。
- verify_area ( ) —— 检查一下用户内存的某块区域是否以所需的许可权限映射。
- get_free_page ( )/free_page ( ) —— 分配和释放物理内存页面。

除了以上这些接口以外，内存管理程序允许在内核中使用它所有的数据结构以及大部分的过程。许多内核模块访问这些数据结构以及子系统的实现细节，它们通过这样的方式与内存管理程序相交互。

3.2.3 子系统描述

因为Linux支持许多硬件平台，所以内存管理程序中有一个与平台相关的部分，它把所有硬件平台的细节抽象成一个通用的接口。对硬件内存管理程序进行的所有访问都是通过这个抽象接口实现的。

内存管理程序使用硬件内存管理程序把虚拟地址（由用户进程所使用）映射到物理内存地址。当用户进程访问内存地址时，硬件内存管理程序把这个虚拟内存地址翻译成物理地址，然后使用物理地址来执行这次访问，因为存在这种映射关系，用户进程无需关心特定的虚拟内存地址是与哪个物理地址相联系的。这就允许内存管理程序子系统把进程的内存映射到物理内存的各个地方。此外，如果两个进程的虚拟内存地址空间区域映射到同一个物理地址空间，实际上这种映射方式允许这两个进程共享该物理内存。

此外，当某进程内存未使用时，内存管理程序会把它交换到分页文件中。这就允许系统可以执行那些需要使用较多内存的进程，即使它们所需的内存超过系统上实际所有的内存。内存管理程序包括一个守护进程（kswapd）。Linux使用的术语“守护进程”指的是内核线程。守护进程也是由进程调度程序来调度的，方法与调度用户进程的方法相同，但守护进程可以直接访问内核数据结构。这样，相对进程而言，守护进程的概念更接近于线程。

kswapd守护进程周期性地检查是否有任何物理内存页面最近没有使用到。如果有，则这些页面将被移出物理内存，如果需要则把它们存储在磁盘上。内存管理子系统特别注意减少所需的磁盘活动的数量。如果可以用另一种方法来达到目的，内存管理程序将避免把页面写以磁盘上。

如果用户进程所访问的内存地址当前没有映射到物理内存位置，则硬件内存管理程序将会检测到这一问题。它将把这次页面错误通知给 Linux内核，而解决这个问题是内存管理程序子系统的责任。这里存在两种可能性：或者该页面当前已被移出内存，这时需要把它再交换进来，或者用户进程是在对映射内存之外的内存地址作非法访问。硬件内存管理程序也能检测到对内存地址的非法调用，例如写可执行代码或者执行中的数据。这些调用也会招致页面错误，并被报告给内存管理程序子系统。如果内存管理程序检测到非法的内存访问，它用一个信号通知用户进程，如果进程不处理这个信号，它将被终止执行。

3.2.4 数据结构

下面的数据结构在系统结构上是相关的：

1) vm_area —— 内存管理程序为每个进程存储一个对应的数据结构，里面记录着哪块虚拟内存区域被映射到哪个物理页面上。这个数据结构中还存放着一个函数指针的集合，允许它在进程的某块虚拟内存区域上执行操作。例如，进程的可执行代码区域并不需要交换到系统分页文件中，因为它可以使用可执行文件来作后援存储。当进程的虚拟内存区域被映射时（例如在装入可执行文件时），虚拟地址空间中每一块连续的区域都将设置一个 vm_area_struct。因为在查看vm_area_struct以检查页面错误时，速度是非常关键的，所以该结构存放在 AVL树中。

2) mem_map —— 内存管理程序为系统上物理内存的每个页面都维护一个数据结构。该数据结构中存放着表示页面状态（例如，该页面当前是否正在使用）的标志。所有的页面数据结构都可以在一个向量（mem_map）中找到，该向量是在内核引导的时候进行初始化的，当页面状态变化时，该数据结构中的属性也相应地更新。

3) free_area —— free_area向量用于存储未分配的物理内存页面，当页面被分配时，需要从free_area中删除该页面，而当页面被释放时，它又返回到 free_area中。当从free_area分配页面时，需要使用Buddy系统[knowlton 1965; Tanenbaum 1992]。

3.2.5 子系统结构

内存管理程序子系统是由许多源代码模块组成的，按照它们的职责可以划分成如下的几个组（如图5-3-3所示）：

- 系统调用接口 —— 这组模块通过一个定义良好的接口（在前面介绍过），把内存管理程序的服务提供给用户进程。
- 内存映射文件（mmap） —— 这组模块负责受支持的内存映射文件 I/O。
- swapfile访问（swap） —— 这组模块控制内存交换。这些模块引起页调入和页调出的操作。
- 核心内存管理（core） —— 这些模块负责核心内存管理程序功能，其他内核子系统将需要使用该功能。
- 系统结构相关的模块 —— 这些模块为所有支持的硬件平台提供一个通用的接口，这些模块执行命令来改变硬件MMU的虚拟内存映射，并在发生页面错误时提供一种通用方法，用来通知内存管理程序子系统的其他部分。

内存管理程序结构的一个有趣之处在于使用了 kswapd，它是一个用于判定应该把哪个内

存页面交换出去的守护进程。 kswapd可以作为一个内核线程来执行，它周期性地检查使用中的物理页面，看看是否可以把哪个页面交换出去。这个守护进程是与内存管理程序子系统的其他部分并发执行的。

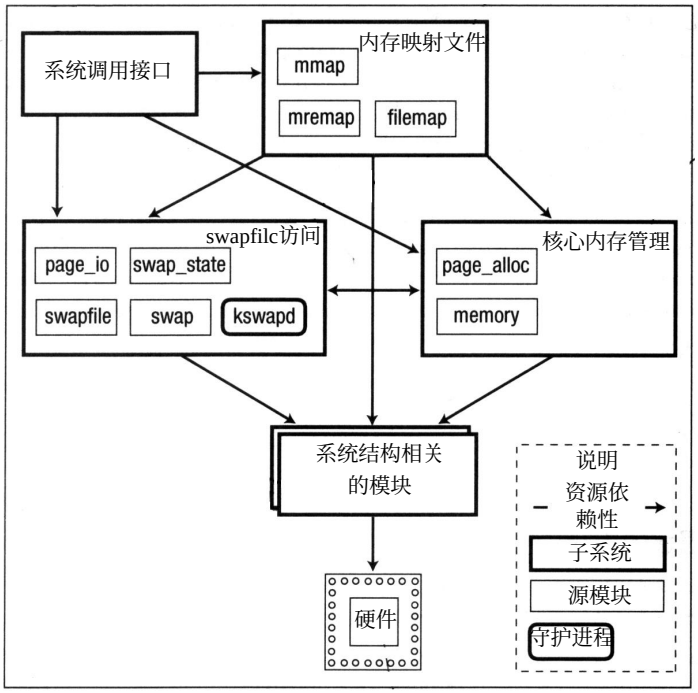


图5-3-3 内存管理程序结构

3.2.6 子系统依赖性

内存管理程序直接被每个 sched、fs、ipc和net所使用（通过数据结构和实现函数）。这种依赖性是很难用简洁的语言来描述的。如果读者希望了解更多有关子系统依赖性的详细知识，可以参考[PBS:mm.html]。图5-3-4展示了内存管理程序和其他子系统之间的高级依赖关系。为了简洁明了起见，图中没有画出内部依赖关系。

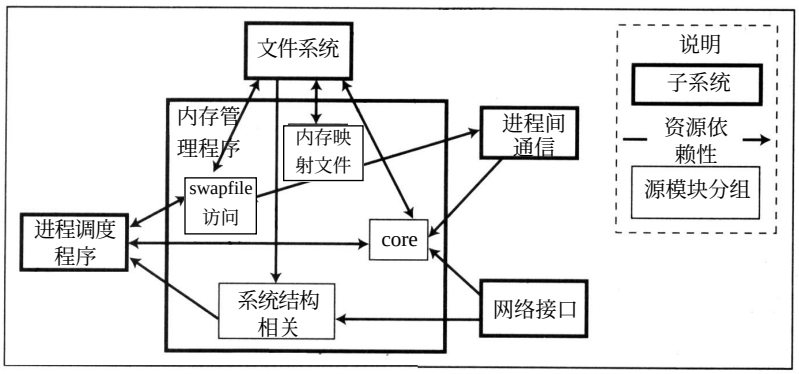


图5-3-4 内存管理程序依赖性

3.3 虚拟文件系统

3.3.1 目标

Linux在设计时就考虑到支持许多不同的物理设备。甚至就一种特定类型的设备而言，例如硬盘驱动器，在不同的硬件厂商之间也会存在许多接口上的差异。除了 Linux所支持的物理设备以外， Linux还支持大量的逻辑文件系统。正因为它能支持许多逻辑文件系统，所以Linux可以轻松地与其他操作系统进行互操作。 Linux文件系统支持下列目标：

- 多个硬件设备 —— 提供对许多不同的硬件设备的访问。
- 多个逻辑文件系统 —— 支持许多不同的逻辑文件系统。
- 多个可执行格式 —— 支持许多不同的可执行文件格式（例如 a.out、ELF、java）。
- 均一性 —— 为所有的逻辑文件系统以及所有的硬件设备提供了一个通用接口。
- 性能 —— 提供对文件的高速访问。
- 安全 —— 不会丢失或毁坏数据。
- 保密性 —— 限制用户访问文件的许可权限，限制分配给用户的总的文件大小。

3.3.2 外部接口

文件系统提供了两个级别的接口：用户进程所使用的系统调用接口，以及其他内核子系统所使用的内部接口。系统调用接口主要负责处理文件和目录。文件操作主要包括由 POSIX兼容系统所提供的一般的 open/close/read/write/seek/tell等操作，目录操作包括 POSIX系统的readdir/creat/unlink/chmod/stat等操作。

文件子系统用于支持其他内核子系统的接口要丰富一些。文件系统提供了一些数据结构和实现函数，以供其他内核子系统进行直接操作。特别是，它为内核的其他部分提供了两个接口 —— 索引节点和文件。其他内核子系统也会使用文件子系统的另一些实现细节，但用得不太普通。

索引节点接口

- create ( )：在目录中创建一个文件。
- lookup ( )：使用文件名称在目录中查找文件。
- link ( )/symlink ( )/unlink ( )/readlink ( )/follow_link ( )：管理文件系统连接。
- mkdir ( )/rmdir ( )：创建或删除子目录。
- mknod ( )：创建目录、特殊文件或者常规文件。
- readpage ( )/writepage ( )：从后援存储中读物理内存页面，或者把物理内存页面写入到后援存储中。
- truncate ( )：把文件的长度设置为0。
- permission ( )：检查一下用户进程是否具有执行某操作的权限。
- smap ( )：把逻辑文件块映射到物理设备扇区。
- bmap ( )：把逻辑文件块映射到物理设备块。
- rename ( )：重新命名文件或者目录。

除了以上这些用户调用索引节点的方法以外，还提供了 namei ( )函数，它允许其他内核子

系统可以找到与文件或目录相联系的索引节点。

文件接口

- open ( )/release ( )：打开或关闭文件。
- read ( )/write ( )：读或写文件。
- select ( )：一直等待，直到文件处于一个特殊的状态（可读或者可写）。
- lseek ( )：如果此功能受支持的话，则移动到文件中特定的偏移量处。
- mmap ( )：把一个文件区域映射到用户进程的虚拟内存。
- fsync ( )/fasync ( )：把任意内存缓冲区与物理设备同步。
- readdir：读某个目录文件所指向的文件。
- ioctl：设置文件属性。
- check_media_change：检查一下可移动媒体（如软盘）是否已被拿走。
- revalidate：确认所有高速缓冲的信息都是合法的。

3.3.3 子系统描述

文件子系统需要支持许多不同的逻辑文件系统和许多不同的硬件设备。它是通过拥有两个概念层而做到这一点的，这两个概念层很容易扩展。设备驱动程序层用一个通用的接口来表示所有的物理设备。虚拟文件系统层（ VFS）则用通用接口来表示所有的逻辑文件系统。Linux内核的概念系统结构（[Bowman 1998]、[Siddiqi 1998]展示了这种分解在概念上是怎样安排的。

3.3.4 设备驱动程序

设备驱动程序层负责向所有的物理设备提供一个通用接口。 Linux内核有三种类型的设备驱动程序：字符设备驱动程序、块设备驱动程序和网络设备驱动程序。其中与文件子系统相关的两种类型是字符设备和块设备。字符设备必须以串行的顺序依次访问，它的典型例子是磁带驱动器、调制解调器和鼠标。块设备可以用任意顺序进行访问，但是只能从多个块大小中读或者写入到多个块大小中。

所有的设备驱动程序都支持前面所介绍的文件操作接口。因此，每个设备都可以当作是文件系统中的一个文件来进行访问（这里所说的文件是指设备特殊文件）。因为内核的大部分都是通过这个文件接口来处理设备的，通过实现设备相关代码来支持这个抽象的文件系统，则加入新设备驱动程序的操作会变得简单一些。因为存在大量不同的硬件设备，所以越容易编写新的设备驱动程序越好，这一点是很重要的。

Linux内核在访问块设备时，使用一个缓冲区高速缓存来提高性能。所有对块设备的访问都是通过缓冲区高速缓存子系统发生的。因为缓冲区高速缓存可以减少对硬件设备的读写操作次数，所以它大大提高了系统的性能。每个硬件设备都有一个请求队列；当缓冲区高速缓存不能在内存缓冲区中满足用户请求时，它把请求加入到该设备的请求队列中，并开始睡眠，直到该请求被满足为止。缓冲区高速缓存使用一个独立的内核线程 kflushd，把缓冲区页面写到设备上，并把它们从高速缓存中删除。

当设备驱动程序需要满足请求时，它首先通过对设备的控制与状态寄存器（ CSR）进行操作，完成该硬件设备操作的初始化工作。有三种通用机制可以把数据从主计算机上移到分

层设备上：轮询、直接内存访问（DMA）以及中断。在使用轮询的情况下，设备驱动程序周期性地检查分层设备的CSR，看看当前请求是否已经完成了。如果已经完成，则驱动程序初始化下一个请求，再继续进行上述操作。对低速硬件设备如软盘驱动器和调制解调器而言，轮询是一种适合的机制，传输的另一种机制是DMA。在这种情况下，设备驱动程序将初始化计算机的主内存和分层设备之间的DMA传输。该传输和主CPU并发工作，在操作继续进行的过程中，允许CPU处理其他任务。当DMA操作完成以后，CPU接收到一个中断。在Linux内核中，中断处理是相当普遍的，它比其他两种方法都要复杂得多。

当硬件设备想要报告某些条件的变化时（例如鼠标键被按下，键盘键被按下），或者报告某操作的完成时，它向CPU发送一个中断，如果使能了中断，则CPU停止执行当前指令，开始执行Linux内核的中断处理代码。内核找到应该调用的合适的中断处理程序（每个设备驱动程序都会注册一些处理程序，用于处理该设备所产生的中断）。当正在处理某中断时，CPU在特殊的上下文中执行，其他的中断将被推迟，直到该中断处理完毕。因为这一限制，中断处理程序的效率需要相当高，这样才不会丢失其他的中断。有时候在时间限度内中断处理程序不能完成所需的全部工作，在这种情况下，中断处理程序在底半处理程序中调度余下的工作。底半处理程序是一段代码，它是下一次完成系统调用时调度程度将要执行的代码。通过把不太重要的工作交给底半处理程序去完成，设备驱动程序可以减少中断延迟，提高并发性。

总而言之，设备驱动程序隐藏了对分层设备的CSR进行操作的细节，以及每个设备的数据传输机制。对块设备而言，缓冲区高速缓存试图在内存缓存区中满足文件系统请求，这样就提高了系统的性能。

3.3.5 逻辑文件系统

尽管可以通过设备特殊文件来访问物理设备，通过逻辑文件系统来访问块设备要更普遍一些。逻辑文件系统可以挂装在虚拟文件系统的挂装点上。这意味着相联系的块设备包含着文件和结构信息，这些信息允许逻辑文件系统访问该设备。在任意的时刻，一个物理设备只能支持一个逻辑文件系统。然而，该设备可以被重新格式化，以便支持另一个逻辑文件系统。在进行写操作时，Linux支持15个逻辑文件系统，这提高了Linux与其他操作系统的互操作性。

当文件系统被作为子目录挂装时，在该设备上的所有目录和文件都将被视为挂装点的子目录。虚拟文件系统的用户不需要考虑哪种逻辑文件系统实现了目录树的哪个部分，也无需考虑哪个物理设备包含着这些逻辑文件系统。这种抽象性在选择物理设备和逻辑文件系统时，提供了很大的方便，而这种方便性是Linux操作系统成功的重要因素之一。

为了支持虚拟文件系统，Linux使用了索引节点的概念。Linux使用索引节点来表示块设备上的文件。索引节点中包含了一些操作，根据文件所驻留的逻辑系统和物理系统的不同，这些操作的实现方法也会有所变化。从这层意义上讲，索引节点是虚拟的。索引节点接口使所有的文件在其他Linux子系统看起来都是一样的。索引节点可以用作一个存储位置，存储磁盘上与某个打开的文件相关的所有信息。索引节点存储相关缓冲，以块为单位的文件总长度，以及文件偏移量和设备块之间的映射。

3.3.6 模块

虚拟文件系统的大部分功能都是以动态装入模块的形式（参见3.1节）提供的。这种动态

配置的特征使得Linux用户可以编译一个尽可能小的内核，如果在单个操作过程中需要的话，还可以允许内核装入所需的设备驱动程序和文件系统模块。例如， Linux系统可能会有一个打印机连接到它的并行口。如果打印机驱动程序永远链接在内核中，则当没有打印机时内存将会浪费。通过使打印机驱动程序成为一个可装入的模块， Linux允许用户在需要使用打印机时装入驱动程序。

3.3.7 数据结构

下面这些数据结构在系统结构上是与文件子系统相关的：

- super_block：每个逻辑文件系统都有一个相联系的超级块，用于把它提供给 Linux内核的其他部分，该超级块包含着整个挂装文件系统的有关信息 —— 正在使用哪些块、块的大小有多大等等。超级块与索引节点的相似点在于：它们均可作为逻辑文件系统的虚拟接口。
- inode：索引节点是一个内存中的数据结构，它表示内核需要知道的有关磁盘上某文件的所有信息。单个索引节点可以被打开该文件的所有进程使用。索引节点存储着内核需要与某个文件相关联的所有信息。计帐、缓冲以及内存映射信息都存储在索引节点中。一些逻辑文件系统在磁盘上也会有一个索引节点结构，可以永久性地保存这一信息，但这与内核的其他部分所使用的索引节点数据结构是不同的。
- fite：文件结构代表由特定进程打开的文件。所有打开的文件都存储在一个双链表中（由first_file所指向），在POSIX形式的过程（open、read、write）中所使用的文件描述符是这个链表中某个打开文件的索引。

3.3.8 子系统结构

（原文缺）

3.3.9 子系统依赖性

图5-3-5、图5-3-6显示了文件系统是怎样依赖于其他内核子系统的。再说一遍，文件系统依赖于所有其他的内核子系统，而所有其他的内核子系统也依赖于文件子系统。特别是，网络子系统之所以依赖于文件系统，是因为网络套接字是作为文件描述符提供给用户的。内存管理程序依赖于文件系统以支持交换。 IPC子系统依赖于文件系统以实现管道和 FIFO。进装调度程序依赖于文件系统来装入可装入的模块。

文件系统使用网络接口来支持 NFS；它使用内存管理程序来实现缓冲区高速缓存以及ramdisk设备；它使用IPC子系统来帮助支持可装入模块；当正在进行硬件请求处理时，它还使用进程调度程序把用户进程置为睡眠状态。如果想要了解依赖性方面的更详细的知识，请参阅[PBS:fs.html]。

3.4 进程间通信

3.4.1 目标

之所以提供Linux IPC机制，是为了给并发执行的进程提供一种方法，是它们可以共享资

源，与其他进程同步并且交换数据。在相同系统上执行的进程之间， Linux通过共享资源、内核数据结构以及等待队列实现了各种形式的 IPC机制。

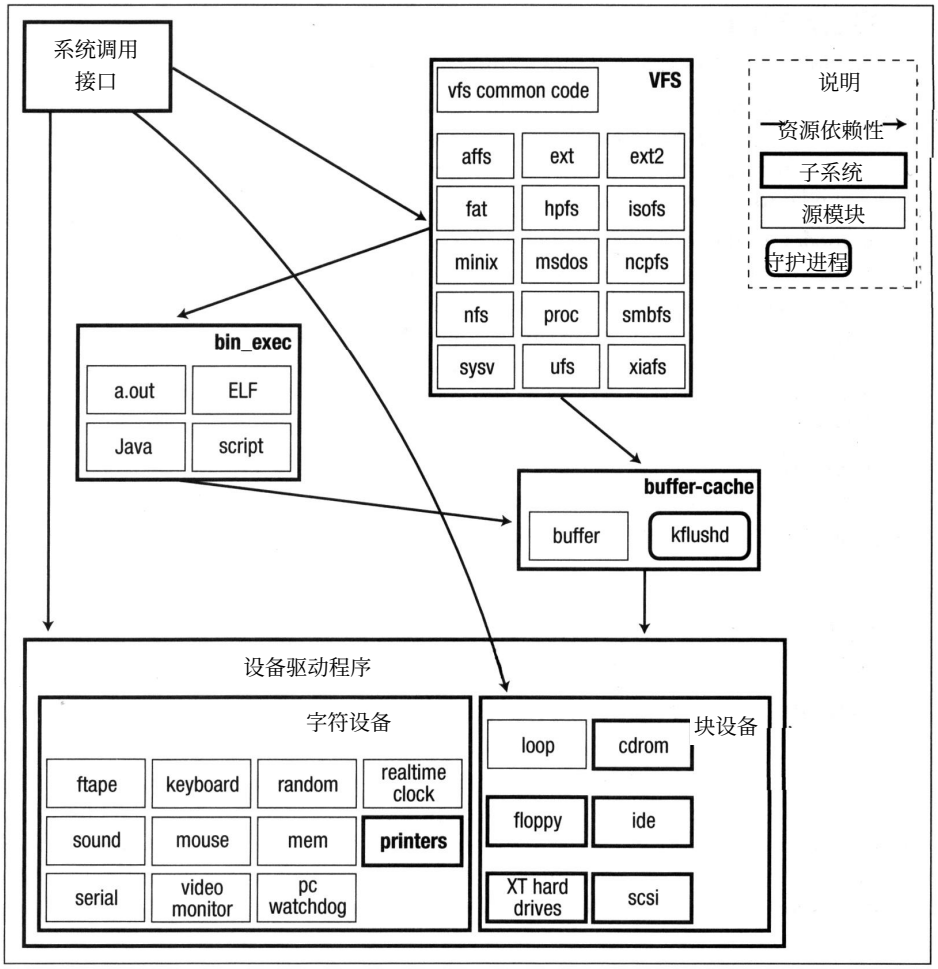


图5-3-5 文件子系统结构

Linux提供了下列形式的IPC机制：

- 信号 —— 可能是最古老的 Unix IPC形式。信号是发往某进程的异步消息。
- 等待队列 —— 当进程正在等待某操作的完成时，等待队列提供了一种机制，可以把进程置为睡眠状态。进程调度程序使用这一机制来实现 3.3.3节所介绍的底半处理操作。
- 文件锁 —— 它提供了一种机制，允许进程声明文件的一个区域，或者整个文件本身，把它们声明为对所有进程都是只读的，除了拥有文件锁的进程以外。

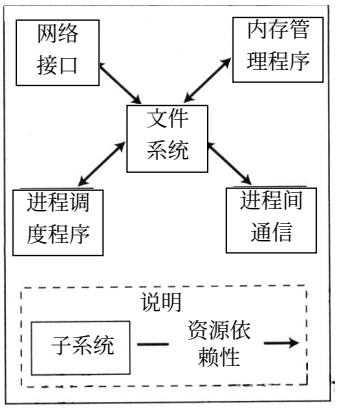


图5-3-6 文件子系统依赖性

- 管道和命名管道 —— 允许在两个进程之间进行面向联接的单向的数据传输，方法可以是显式地建立管道连接，或者通过驻留在文件系统中的命名管道进行通讯。
- 系统V IPC
- 信号量 —— 传统信号量模型的一种实现，该模型还允许创建信号量数组。
- 消息队列 —— 一种无联接的数据传输模型。消息是字节的序列，并带有相应的类型。消息可以写入到消息队列中，并且可以通过从消息队列中读取来获得消息，当然也可以限制读入哪种类型的消息。
- 共享内存 —— 通过使用这种机制，几个进程可以访问物理内存的同一块区域。
- Unix域套接字 —— 另一种面向连接的数据传输机制，它提供了与 INET套接字相同的通信模型。INET套接字将在下一章进行介绍。

3.4.2 外部接口

信号是内核或者另一个进程发送给某个进程的通知。信号是使用 send_sig ( )函数发送的。信号编号以及目标进程都必须作为参数提供给该函数。进程可以通过使用 signal ( )函数来注册处理信号。

文件锁直接由Linux文件系统支持。为了锁定整个文件，可以使用系统调用 open ( )，或者使用系统调用sys_fcntl ( )。锁定文件中的几个区域也可以通过 sys_fcntl ( )系统调用来完成。

管道可以使用pipe ( )系统调用来创建。然后可以使用文件系统的 read ( )和write ( )调用在管道上传输数据。命名管道使用 open ( )系统调用来打开。

系统V IPC机制有一个通用接口，它就是 ipc ( )系统调用。使用该系统调用的参数可以指定各种IPC操作。

Unix域套接字功能也封装在单个系统调用 socketcall ( )中。

上面所提到的这些系统调用的文档都编制得很好，提倡读者去查询相应的 man帮助页面。

IPC子系统向其他内核子系统提供等待调用。因为用户进程不能使用等待队列，所以它们没有系统调用接口。等待队列可以用于实现信号量、管道以及底半操作处理程序（参见 3.3.3节）。过程add_wait_queue（）把一个任务插入到等待队列中。过程 remove_wait_queue（）则把任务从等待队列中删除。

3.4.3 子系统描述

下面简要描述一个3.4.1节中所列出的每个IPC机制的低级功能。

信号用于把事件通知给进程，根据特定信号的语义不同，信号可以改变接收进程的状态。内核可以向任意执行中的进程发送信号。只有在拥有相应的 PID或GID的情况下，用户进程才能向进程或者进程组发送信号。对静止的进程来说，信号并不立即处理。相反，在下一次调度程序设置进程使其在用户模式中运行以前，它检查是否有个信号被发送给该进程，如果是，则调度程序调用do_signal ( )函数，该函数可以正确地处理信号。

等待队列只是指向任务结构的指针的链表，该链表对应着等待某内核事件的进程。内核事件的例子有DMA传输的结束等。进程可以把自己加入到等待队列中，方法是调用 sleep_on ( )或者interruptable_sleep_on ( )函数。与之相对应，函数wake_up ( )和wake_up_interruptable ( )可以把进程从等待队列中删除。中断例程也使用等待队列来避免条件竞争。

Linux使用户进程可以防止其他进程访问某文件。这种排它性可以基于整个文件，也可以基于文件的一块区域。文件锁可以用于实现这种排它性。文件系统实现包括在数据结构中适当的数据域，该域允许内核去判断该锁锁定的是文件，还是文件中的区域。如果是前者，则试图去锁一个已经锁定的文件将会出错；而如果是后者，则试图去锁一个已经锁定的区域也将出错。在两种情况下，请求进程均不允许访问文件，这是因为内核还没有确认该锁。

管道和命名管道具有相似的实现，这是因为它们的功能也几乎是相同的。虽然它们的创建过程是不同的。然而，无论是创建管道还是命名管道，都将会返回一个文件描述符，该文件描述符就指向该管道。在创建以后，把一个内存页面与打开的管道联系在一起。这个内存被当作环形缓存区来处理，在该缓冲区中，写操作是以原子操作的方式来完成的，当缓冲区已满后，写进程将阻塞。如果读请求所要读的数据量超出了现有的数据量，则读进程将阻塞。这样一来，每个管道相应该有一个等待队列与之相连。在读和写的过程中，进程将被增加到队列中，或者从队列中删除。

信号量是使用等待队列来实现的，它遵循的是典型的信号量模型。每个信号量都有一个与之相联系的值。在信号量上实现了两个操作 up ( )和down ( )。当信号量的值为零时，在信号量上执行减一操作的进程将在等待队列上阻塞。信号量数组只是信号量的一个连续的集合。每个进程还保留着它执行过的信号量操作的列表，这样一来，如果进程过早地退出，这些操作可以撤消。

消息队列是一个线性列表，进程可以从消息队列读字节的一个序列，或者把字节序列写到消息队列中。消息的接收顺序与当时写入它们的顺序是一致的。有两个等待队列与消息队列相联系，一个用于存放想要把消息写入到一个已满的消息队列的进程，而另一个则用于串行化消息的写操作。消息的实际大小在创建消息队列时设置。

共享内存是IPC最快捷的方式。这个机制允许进程共享它们内存的一个区域。共享内存区域的创建工作是由内存管理系统来处理的。通过调用系统调用 sys_shmat ( )，可以把共享页面与用户进程的虚拟内存空间连接在一起。通用调用 sys_shmdt ( )调用，可以把共享页面从进程的用户段删除掉。

Unix域套接字的实现方式与管道相似，因为它们都是基于环形缓冲区来实现的，而环形缓冲区又是基于内存页面的。然而，套接字为每个通信方向都提供了一个独立的缓冲区。

3.4.4 数据结构

本节将介绍用于实现上述IPC机制所需的一些重要的数据结构。

信号是通过task_struct结构中的signal域来实现的。每个信号都由这个域中的一个位来表示。这样一来，Linux的某个版本所能支持的信号的数目就不能超过一个字中位的数目。在域block中，存放着进程所阻塞的信号。

等待队列只有一个数据结构与之相联系，即 wait_queue结构。这些结构包含着一个指针，指向相联系的task_struct，并被链接到一个列表中。

文件锁有一个相联系的 file_lock结构。这个结构中存放的信息有：一个指向拥有进程的task_struct结构的指针、被锁定文件的文件描述符、等待取消文件锁的进程的等待队列，以及被锁定的文件区域。每个打开的文件的 file_lock结构均被链接到一个列表中。

管道都是由文件系统索引节点来表示的，不管它是无名管道还是命名管道。该索引节点

在pipe_inode_info结构中存放着附加的一些管道相关的信息。这个结构中存放着一个等待队列，用于存储在读或写操作上阻塞的进程，它还存放着指向内存页面的指针（该页面被用作管道的环形缓冲区）、在管道中数据的数量，以及当前正在读写管道的进程的数量。

所有的系统V IPC对象都是在内核中创建的，每个对象都有与之相联系的访问许可权限。这些访问许可权限存放在ipc_perm结构中。信号量是用sem结构来表示的，该结构中存放着信号量的值，以及最后一次在信号量上执行操作的进程的 pid。信号量数组是用semid_ds结构来表示的，它存放着访问许可权限、最近一次执行信号量操作的时间、指向数组中第一个信号量的指针，以及执行信号量操作时阻塞的进程的队列。结构 sem_undo可用于创建某进程所执行的信号量操作的一个列表，这样在该进程被杀掉时，可以撤消这些操作。

消息队列是基于结构msquid_ds的，该结构中存放着管理和控制信息。这个结构中存储着下列域：

- 访问许可权限。
- 用于实现消息队列的链接字段（例如，指向 msquid_ds的指针）。
- 最后一次发送、接收和修改的时间。
- 进程阻塞的队列，和前一节所介绍的那样。
- 当前在队列中的字节数量。
- 消息的数量。
- 队列的大小（以字节为单位）。
- 最后一个发送者的进程编号。
- 最后一个接收者的进程编号。

消息本身是用msg结构存储在内核中的。该结构存放一个链接域，用来实现消息的一个链表，它还存放着消息的类型、消息数据的地址，以及消息的长度。

共享内存的实现是基于shmid_ds结构的。该结构与msquid_ds结构相似，也是存放一些管理和控制信息。 shmid_ds结构中存放着访问控制许可权限、最后一次连接、断开连接和修改的时间、创建者的pid、最后一次对该共享段执行操作的进程的 pid、共享内存段所连接的进程的数量、组成共享内存段的页面数量，以及页面表项目的一个域。

Unix域套接字是基于socket数据结构的，该结构将在网络接口那一节（ 3.5）中介绍。

3.4.5 子系统结构

图5-3-7显示了IPC子系统资源的依赖性，控制从系统调用层向下流入每个模块中。系统 V IPC工具都是在内核源代码的ipc目录中实现的。内核IPC模块调用在kernel目录中实现的IPC工具。文件工具和网络IPC工具的实现与此类似。

系统V IPC模块依赖于内核IPC机制。特别是，信号量是用等待队列实现的。其他所有的IPC工具在实现上都是相互独立的。

3.4.6 子系统依赖性

图5-3-8显示了IPC子系统与其他内核子系统之间的资源依赖性。

IPC子系统是因为套接字而依赖于文件系统的。套接字使用文件描述符，并且一旦打开它们，它们都会被赋予一个索引节点。内存管理之所以依赖于 IPC，是因为页面交换过程调用

IPC子系统来实现共享内存的交换。 IPC之所以依赖于内存管理，主要是因为缓冲区的分配以及共享内存的实现。

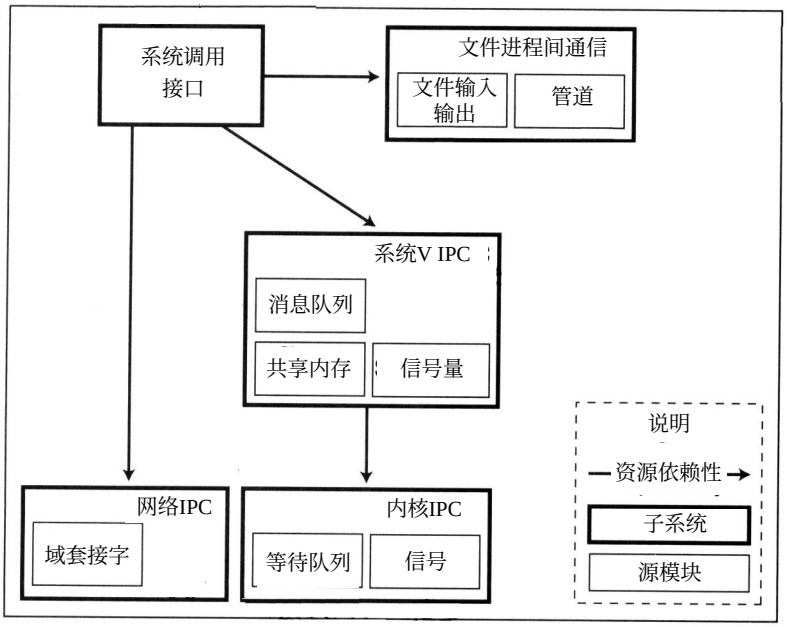


图5-3-7 IPC子系统的结构

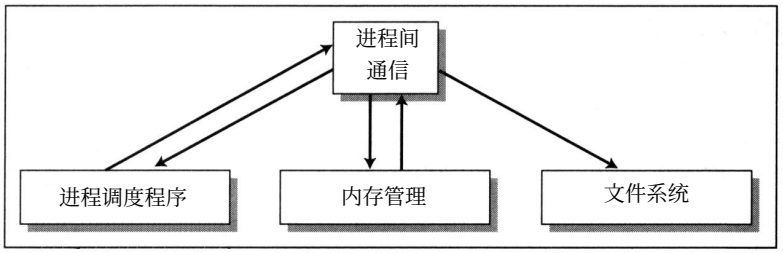


图5-3-8 IPC子系统的依赖性

一些IPC机制赖使用定时器。定时器是在进程调度程序子系统中实现的。进程调度主要依靠信号。因为这两个原因， IPC和进程调度程度模块必须相互依赖。如果读者想要了解 IPC子系统模块和其他内核子系统之间依赖性的更多知识，可以参考 [PBS:ipc.html]。

3.5 网络接口

3.5.1 目标

Linux网络系统提供了计算机之间的网络连接，以及套接字通信模型。 Linux总共提供了两种类型的套接字实现：BSD套接字和INET套接字。BSD套接字是使用INET套接字实现的。

Linux网络系统提供了两种传输协议，它们的通信模型和服务质量是不同的。这两种传输协议是不可靠的，基于消息的 UDP协议；以及可靠的流式 TCP协议。它们都是在 IP网络协议的基础上实现的。 INET套接字则同时基于传输协议和 IP协议来实现。

最后，IP协议是位于基本设备驱动程序的上一层。设备驱动程序可以为三种不同类型的连接而提供：串行联接（SLIP）、并行联接（PLIP）以及以太网联接。在IP和以太网驱动程序之间，存在一个地址解析协议。这个地址解析协议的任务是在逻辑 IP地址和物理以太网地址之间进行翻译。

3.5.2 外部接口

其他子系统和用户是通过套接字接口来使用网络服务的。套接字可以通过socketcall ( )系统调用来创建和操作，通过在套接字文件描述符上使用read ( )和write ( )调用，可以发送和接收数据。

网络子系统不提供其他的网络机制和功能。

3.5.3 子系统描述

BSD套接字模型是提供给用户进程使用的。该模型是面向联接的、流式的，并且是缓存的通信服务。BSD套接字是在INET套接字模型上实现的。

BSD套接字模型处理的任务与 VFS相似，它还为套接字联接管理一个通用的数据结构。BSD套接字模型的目标是：通过把通信细节抽象成通用接口来提供更大的可移植性。 BSD接口在现代操作系统（如Unix和Microsoft Windows）中广泛使用。INET套接字模型为基于IP的UDP和TCP协议管理实际通信端点。

网络I/O是以读写套接字开始的。它将调用一个 read/write系统调用，而该调用由虚拟文件系统的部件进行处理（对网络子系统层的 read/write调用是对称的，所以从现在起，我们只考虑写操作）。那个部件判定BSD套接字sock_write ( )是用于实现实际的文件系统写调用的，所以它将调用sock_write ( )。该过程处理管理细节，并把控制传送给 inet_write ( )函数。而该函数反过来又调用传输层写调用（如 tcp_write ( )）。

传输层写协议负责把到达的数据分成传输报文。这些过程把控制传送给ip_build_header ( )过程，它创建一个IP协议报头，以便插入到欲发送的报文中，然后再调用tcp_build_header ( )来创建TCP协议报头。一旦完成了以上工作，就可以使用底层的设备驱动程序来进行实际数据发送了。

网络子系统提供两种不同的传输服务，每一种的通信模型和服务质量都是不同的。 UDP提供了一个无联接的、不可靠的数据传送服务。它负责从 IP层接收报文，并找到应该把报文数据发送到哪个目标套接字。如果没有提供目标套接字，则将会报错。否则，如果有足够的缓冲区内存，报文数据将加入到套接字接收的报文列表中。在读操作上睡眠的所有套接字均获得通知，并被唤醒。

TCP传输协议提供了一个更为复杂的方案。除了处理发送进程和接收进程之间的数据传输以外，TCP协议还执行复杂的联接管理。TCP把数据以流的形式向上传送，而不是以报文序列的形式发送，发送目标是套接字层，并保证提供一个可靠的传输服务。

IP协议提供报文传输服务。给定一个报文，以及报文的目标， IP通信层将负责把该报文传送到正确的主机上。对于一个向外流出的数据流， IP负责以下工作：

- 把该流分成IP报文。
- 把IP报文传送到目标地址。
- 产生一个报文头，以供硬件设备驱动程序使用。
- 选择发送到合适的网络设备上。

对于一个到达的报文流，IP必须执行以下操作：

- 检查报文头，以确认其合法性。
- 把目标地址和本地地址相比较，如果报文没有到达正确的目标，则把它向前移动。
- 解析IP报文。
- 把报文向上发送到TCP或者UDP层，以便进一步处理。

ARP（地址解析协议）负责转换 IP地址和实际硬件地址。 ARP支持各种硬件设备，如以太网、 FDDI等等。因为套接字是处理 IP地址的，而底层硬件设备不能直接使用 IP地址，所以ARP的功能是非常必要的。因为使用的是一个中性的编址方案，所以同一个通信协议可以在各种硬件设备间实现。

网络子系统为串行联接、并行联接以及以太网联接提供了自己的设备驱动程序。为了隐藏通信媒体之间差异，不让网络子系统的上层知道，这里提供了一个各种硬件设备的抽象接口。

3.5.4 数据结构

图5-3-9显示了网络子系统的结构。

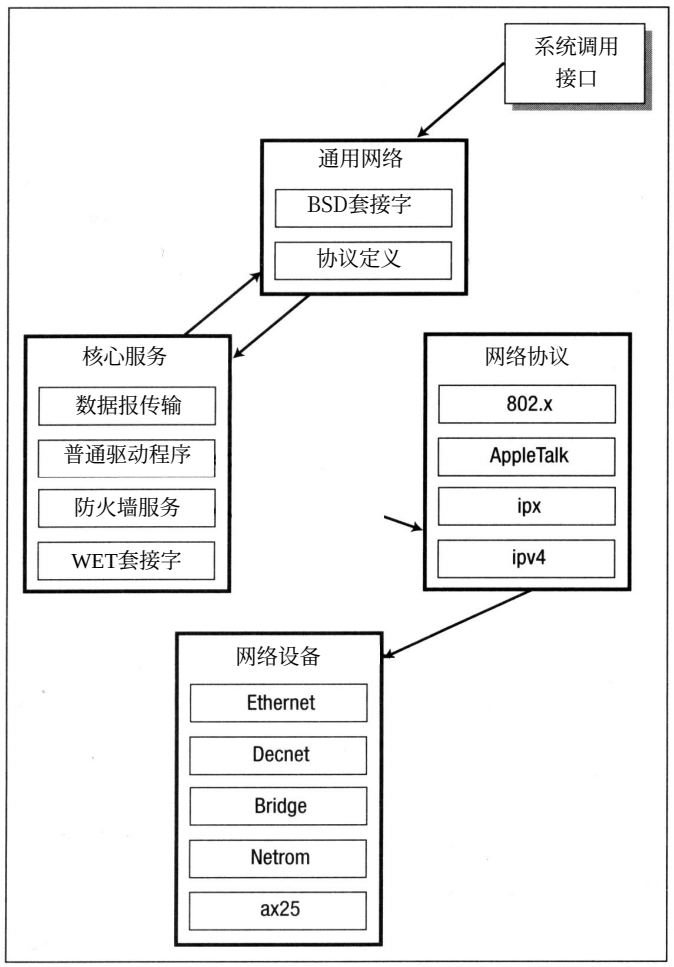


图5-3-9 网络子系统的结构

BSD套接字实现是用 socket结构来表示的。它包含着一个域，该域标识了套接字的类型

(流式的或者数据报的)，该域还标识套接字的状态（联接还是未联接)，该结构中还包含一个存放着标志的域，该标志修改了套接字的操作。除此之外，它还包含一个指针，指向一个结构，该结构用于描述可以在该套接字上执行的操作。另外 socket结构还包含一个指针，指向相联系的INET套接字实现，以及对索引节点的引用。每个 BSD套接字都是与一个索引节点相联系的。

结构sk_buff可以用于管理单个通信报文。该缓冲区指向它所属的套接字，其中包含上一次它被传输的时间，以及一个链接域，这样与给定套接字相联系的所有报文都可以链接在一起（在一个双链表中)。在该缓冲区中存放着源地址和目的地址、头部信息以及报文数据。该结构封装了建网系统所使用的所有报文（例如 TCP报文、UDP报文、IP报文等等)。

sock结构调用INET套接字相关的信息。该结构的成员包括套接字申请读写内存的次数，TCP协议所需的序列号、可以设置用来改变套接字行为的标志、缓冲区管理域（例如为了保存给定套接字所接收到的所有报文的列表)，以及阻塞读和阻塞写的等待队列。除此以外，该结构还提供了一个指针，指向保存一个函数指针列表的结构，而这些函数指针又是用来处理协议相关过程的。最后， sock结构中还包含proto结构。proto结构更大更复杂，但是最重要的是，它向TCP和UDP协议提供了一个抽象接口。还提供了源地址和目的地址，以及更多的 TCP相关数据字段。TCP使用定时器是为了处理时间消耗完这一事件。这样， sock结构中包含着与定时器操作相适应的数据域，以及一些函数指针，它们可以用作定时器响铃的回调处理。

最后，device设备被用于定义网络设备驱动程序。这与用来表示文件系统设备驱动程序的结构是同一个结构。

3.5.5 子系统结构

通用网络包含着那些向用户进程提供最高级接口的模块。这些主要是 BSD套接字接口，另外通用网络还包括网络层所支持的协议的定义。这里所包含的协议包括 MAC协议802.x、ip、ipx以及Apple Talk。

核心服务与高级实现工具相对应，这些工具包括 INET套接字、支持防火墙、通用网络设备驱动程序接口以及数据报和 TCP传输服务的工具。

系统调用接口与 BSD套接字接口相交互。 BSD套接字层为套接字通信提供了一个通用的抽象，这种抽象是使用 INET套接字来实现的。这就是通用网络模块与核心服务模块之间存在依赖性的原因。

该协议模块包含着一些代码，用于取得用户数据，并用特定协议所需的格式来格式化这些数据。协议层最终会把数据发送到合适的硬件设备，因此导致网络协议模块和网络设备模块之间存在依赖性。

网络设备模块包含设备类型相关的一些高级过程。实际的设备驱动程序与常规设备驱动程序一起驻留在目录drivers/net中。

3.5.6 子系统依赖性

图5-3-10显示了网络子系统与其他子系统间的依赖性。

网络子系统因为缓冲区而依赖于内存管理系统。文件系统用于给套接字提供一个索引节点接口。文件系统使用网络系统来实现 NFS。网络子系统需要使用 kerneld守护进程，所以它

依赖于IPC。网络子系统使用进程调度程序来完成计时器功能，并在执行网络传输时中断进程。如果读者想要了解更多有关网络子系统模块与其他内核子系统之间依赖性的知识，可以参考[PBS:net.html]。

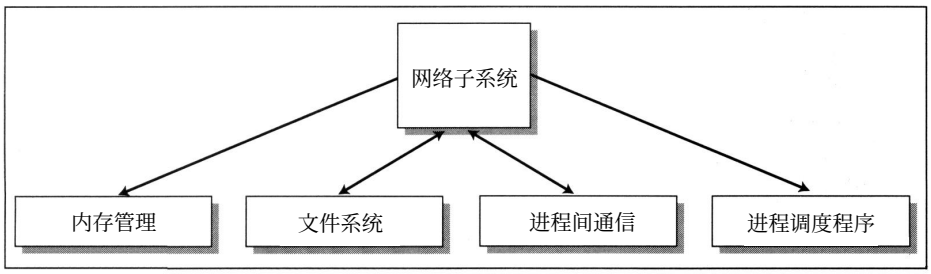


图5-3-10 网络子系统依赖性

第4章 结 论

总而言之，我们概览了一遍现有的文档，抽取出源文件事实，把模块集成到子系统中，并使用现代操作系统的领域知识优化了这种集成工作。我们发现，具体系统结构与我们以前所形成的概念系统结构并不匹配。最初抽取的结构显示，由于抽取工具的限制，我们遗失了依赖性。在人工确认了这些依赖关系以后，我们发现所有的失误都是由于这些工具的限制所造成的。另一方面，最后得到的具体系统结构展示了几个在概念系统结构中我们没有考虑到的依赖性。这些依赖性是由于一些设计决定所造成的。例如让 IPC子系统执行一些内存管理交换函数。这些预想之外的依赖性在 [Murphy 1995]中称为散度，它显示出概念系统结构的表示方式还不太准确，或者具体系统结构没有使用到合适的集成机制。然而，我们相信这些散度是Linux开发人员忽略系统结构方面的考虑而造成的。

我们浏览了Linux内核的可用文档。当前有着大量的系统文档，如 [Rulsing 1997]，甚至还有一些随源代码一起提供的受限文档。然而，这种文档因为层次太高，不适合作为详细的具体系统结构。

我们使用cfx工具从Linux内核代码中抽取详细的设计信息。cfx抽取工具产生了100,000个事实，这些事实使用grok可以组合成1,600个源模块中的15,000个依赖关系。在考察了事实抽取工具的输出以后，我们注意到结果存在两个问题。第一，有些依赖关系没有被检测出来。特别是在隐式调用的情况下更容易出现这个问题。在隐式调用的情况下，子系统使用另一个子系统注册它的功能，因为对注册子系统的访问是通过函数指针完成的，所以cfx工具不能检查出这种依赖关系。第二，cfx工具会报告一些不存在的依赖性。这种情况的一个例子发生在IPC子系统中，在IPC子系统中有三个源模块具有相同名称的函数（findkey)。因为cfx工具使用的是名称相等假设，它会报告说这三个模块相互依赖，而人工检查的结果却显示出这三个模块并不互相依赖。

除了由cfx工具所引入的一些假象以外，我们遇到另外一些问题，它们是由于 grok脚本假设每个源模块（*.c）都有一个相关的头文件（ *.h）而引起的。在Linux内核中这个假设一般是对的，但是有些头文件在许多源模块中都实现了。例如， mm.h在内存管理程序子系统的多个源模块中都有实现。因为对函数所作的调用都被抽象成对声明函数的头文件的调用，在把源模块分到子系统中时，我们就会遇到困难。所有的源模块都好象依赖于 mm.h，而不是依赖于实现所使用资源的特定源模块。可以用修改 grok脚本的方法来纠正这个问题，也许某些能够识别出头文件（如mm.h）的特殊本质的综合方法可能会更有用。

因为存在这些问题（丢失依赖性、伪造依赖性或者依赖性定位不准），我们需要人工检查抽取事实中与我们的期望不同的地方。不幸的是，因为源代码数量很大，而我们自己 Linux的经验又相对有限，所以我们所抽取的事实不能认为是完全正确的。

将来的工作

应该调整PBS工具，以便处理Linux源结构，我们所提供的概念系统结构和具体系统结构应该优化，方法是向Linux开发人员讨教，在优化之后，可以使用反射模型 [Murphy 1995]把这两个模型进行比较。

附录A 术语定义

BSD

Berkeley系统发布（Berkeley System Distribution）的缩写，这个Unix版本是基于AT&T系统V Unix，它是由加利福利亚大学Berkeley分校的计算机系统研究小组开发的。

设备特殊文件（device special file）

在该文件系统中的每个文件都表示一个物理设备。对此文件进行读写操作将导致对该物理设备的直接访问。这些文件很少直接使用，除了字符模式的设备以外。

fs

Linux内核的文件系统子系统。

中断延迟（interrupt latency）

中断延迟是在把中断报告给 CPU到中断被处理的那一段时间。如果中断延迟太大，则高速分层设备将会出错，因为在发生下一次中断之前无法处理它们的中断（通过从 CSR读数据）。后续的中断将覆盖CSR中的数据。

进程间通信（IPC）

IPC是进程间通信（Interprocess communication）的缩写。Linux支持信号、管道以及系统V IPC机制。这些机制使用的名称是沿用它们第一次出现在 Unix版本中的名称。

内核线程（守护进程，daemon）

内核线程是在内核模式中运行的进程。这些进程准确地说是内核的一部分，因为它们可以访问内核的所有数据结构和函数。但是它们被当作独立的进程来对待，可以中断和恢复执行。内核线程没有虚拟内存，但是可以访问其他内核所访问的相同的物理内存。

逻辑文件系统（logical file system）

该系统把数据块表示成文件和目录，这些文件和目录存储逻辑文件系统所特有的属性，但可以包括许可权限、访问和修改时间，以及其他一些计帐信息。

mm

Linux内核的内存管理程序子系统

挂装点（mount point）

它是虚拟文件系统中的目录，是挂装逻辑文件系统所在的地方。在挂装的逻辑文件系统上的所有文件都好象是挂装点的子目录。根文件系统是在 ’ / ’ 下挂装的。

Net

Linux内核的网络接口子系统

逆推工程（reverse engineering）

从源代码中抽取高级模型的过程。

sched

Linux内核的进程调度程序子系统。

软件系统结构（software architecture）

它是系统的结构，由软件部件、这些部件的外部可视属性以及它们之间的关系所组成。

系统V（system V）

一个Unix版本，由AT&T的贝尔实验室开发。

Unix

一种操作系统，最早是在1970年由贝尔实验室开发的。

附录B 参考文献

Bass 1998

Bass、Len、Clements、Paul、Kazman及Rick所著：Software Architecture in Practice, ISBN 0-201-19930-0, Addison Wesley, 1998

Beck 1996

Beck等所著：Linux Kernel Internals, Addison Wesley, 1996

Bowman 1998

Bowman, I.:“Conceptual Architecture of the Linux Kernel”,
http://www.grad.math.uwaterloo.ca/~itbowman/CS 746G/al/,1998

Chi 1994

Chi, E.:“Introduction and History of Linux”,
http://lenti.med.umn.edu/~chi/technolog.html,1994

CS746G Bibliography

http://plg.uwaterloo.ca/~holt/cs/746/98/linuxbib.html

Holt 1997

Holt, R.:“Portable Bookshelf Tools”,
http://www.turing.toronto.edu/homes/holt/pbs/tools.html

Knowlton 1965

Knowlton.K.C.:“A Fast Storage Allocator”, Communications of the ACM, vol.8, 第 623页～第625页, Oct.1965

Kruchten 1995

Kruchten, P.B.:The 4+1 Views Model of Architecture, IEEE Software, Nov 95, 第42页～第50页

LDP

Linux Documentation Project:http://sunsite.unc.edu/mdw/linux.html

Mancoridis Slides

Mancoridis, S.:MCS680 Slides, Drexel University

Muller 1993

Muller、Hausi A.、Mehmet、O.A.、Tilley、S.R.以及Uhl、J.S.所著：“A Reverse Engineering Approach to Subsystem Identification”, Software Maintenance and Practice, vol.5, 181~204, 1993

Murphy 1995

Murphy, G.C、Notkin、D.以及Sullivan K.所著：“Software Reflexion Models:Bridging the Gap between Source and High-Level Models”, Proceedings of the Third ACM Symposium on the Foundations of Software Engineering (FSE’95)

PBS
我们所抽取的Linux描述：http://plg.uwaterloo.ca/~itbowman/pbs/.
Rusling 1997
Rusling,D.A:The Linux Kernel, ftp://sunsite.unc.edu/pub/Linux/docs/linux-doc-project/linux-kernel/,1997
Siddiqi 1998
Siddiqi, S.:“A Conceptual Architecture for the Linux Kernel”, http://se.math.uwaterloo.ca/~s4siddiq/CS746G/LA.html, 1998
Tanenbaum 1992
Tanenbaum, A.S.:Modern Operating Systems, Englewood Cliffs, NJ:Prentice-Hall, 1992
Tanuan 1998
Tanuan, M.:“An Introduction to the Linux Operating System Architecture”, http://www.grad.math.uwaterloo.ca/~mctanuan/cs746g/LinuxCA.html, 1998
Tzerpos 1996
Tzerpos, V.和Holt,R.所著：“A Hybrid Process for Recovering Software Architecture”, http://www.turing.toronto.edu/homes/vtzer/papers/hybrid.ps, 1996

第六部分　附　　录

附录A　Linux文档工程拷贝许可证

最后一次修改时间：1997年1月6日

下面的版权许可证适用于Linux文档工程小组所写的所有文档。

请认真阅读该许可证 —— 它在某种程度上类似于GNU通用公共许可证，但是在该许可证中有些条件可能与您熟悉的不一样。如果您有任何问题，请给 LDP负责人发电子邮件，邮箱地址是mdw@metalab.unc.edu。

Linux文档工程手册可以作为一个整体进行复制和发行，也可以只复制和发行它的一个部分，只需遵循以下条件即可。

所有的Linux文档工程手册都是由它们各自的作者版权所有的。它们并不是公共财产。

- 如果要完成或部分地拷贝文档，必须完整地保留上述版权声明以及这个许可声明。
- 如果对《Linux Installation And Getting Started》作任何翻译工作或者摘抄其中的内容，则在发行之前必须通过作者同意。
- 如果发行《Linux Installation And Getting Started》的一部分，则必须加入获得这本手册的完整版本的指导，并提供获得完整版本的方法。
- 如果想要在其他文档中引用本手册的内容，但不想在书中带上许可声明，则可以复制其中的一小部分作为概述或引言的说明。
- 下面所引用的GNU通用公共许可证在满足它所给的条件的情况下可以进行复制。

这些规则并不适用于科研目的：可以向作者写信并询问有关情况。这些限制出现在这里是为了保护我们作者的，而不是为了限制教育工作者和学者。《Linux Installation And Getting Started》中所有的源代码都受到GNU通用公共许可证的保护，读者可以通过GNU文档站点的匿名FTP服务器得到该许可证，网址是ftp://prep.ai.mit.edu:/pub/gnu/COPYING。

出版LDP手册

如果你是出版公司，并对出版任意一本LDP手册感兴趣，请阅读以下的文字。

因为有前一节所给的许可证，任何人都可以出版和发行 Linux文档工程小组手册的完整拷贝。如果您想这样做的话，并不需要我们明确地表示同意。然而，如果您想发行任意一本LDP手册的译本或者摘抄版本。在这样做以前，必须从作者申请书面形式的许可证。

对LDP手册所做的所有翻译工作和摘抄工作都必须符合前面一节所给出的 Linux文档许可证的规定。那就是说，如果您计划出版一本手册的译本，按照以上规定它也必须是“自由”发行的。

当然，也可以通过卖LDP手册来获利。我们鼓励您这么做。然而请记住，因为 LDP手册是自由发行的，任何人都可以不负任何责任地来影印或发行这些书的打印版本，如果他们想这样做的话。

我们并不要求从您销售LDP手册所获的利润中得到版税。然而，如果您确实从销售 LDP手册中获得了利益的话，我们建议您或者付给作者版税、或者把一部分利润捐献给作者，或者整个LDP，或者捐献给Linux开发社团。我们还希望您能给作者提供一本或几本正在发行的

LDP手册的免费拷贝。我们将会非常欣赏您支持 LDP和Linux社团的行为。

如果您把计划出版和发行 LDP手册的打算通知我们，我们将会非常感谢，因为这样一来我们可以知道这些手册正在发挥作用。如果您正在出版或者计划出版任意的 LDP手册，请给Matt Welsh发电子邮件、邮箱地址是mdw@metalab.unc.edu。

我们鼓励Linux软件发行人员随软件一起发行 LDP手册(例如《Installation And Getting Started Guide》)。LDP手册可以用作“正规的”Linux文档。如果我们看到邮件订购发行人员把LDP手册和软件捆绑在一起，我们将会很高兴。随着 LDP日臻成熟和完善，我们希望它们能更好地完成这一目标。

附录B　GNU通用公共许可证

我们加入GNU通用公共许可证(GPL)的目的是为了方便读者查询，因为它适用于本书所介绍的软件。然而，它并不适用于本书的内容。

第二版，1991年6月

序　言

大多数软件的许可证都被用来剥夺您享有和改变它的自由，但和它们不同，　GNU通用公共许可证是用来保证您分享和改变免费软件的权利 —— 保证软件对所有的用户都是免费的。这个通用公共许可证，适用于免费软件联盟的大部分软件和其他经过作者允许使用的程序(有些其他的免费软件联盟的软件由 GNU库通用公共许可证所包含)，您也可以为您的软件申请一份。

当我们说到免费软件，我们指的是自由，而不是价格，我们设计通用公共许可证的目的是为了保证您有发行免费软件拷贝的权利(而且如果您愿意，还可以根据需要做一些改变)。保证当您需要时能得到源代码，保证您能够改变软件或在其他的新的免费程序中使用其中的部分，而且您知道能够做这些事情。

为了保护您的权利，我们需要制定限制，使任何人无权剥夺您的这些权利或让您放弃这种权利。当您发行软件的拷贝或当您改变它时，这些限制实际上就是一定的责任。

例如，当您发行一个免费或收费的程序时，您必须把您所有的权利都给程序的接受者。您必需使他们也能收到或得到源代码，而且也必须给他们看这个条款，以使他们也知道他们的权利。

我们通过两个步骤来保证您的权利： (1)给软件加版权；(2)给您提供许可证，以使您合法地拷贝、发行和修改软件。

同时，为了保护我们自己以及作者的权利，我们力图使每一个人知道这些免费软件并没有任何担保。当软件已经被别人修改并被传递下去时，我们希望接受者知道他们手中的并不是最初的版本，因此由其他人所引入的问题不会影响到原作者的声誉。

最后，所有免费软件都常常受到软件专利的威胁，我们希望能防止使用者会将免费软件申请专利，使得程序变为私有财产。为了防止这种事的发生，我们明确规定专利申请必须以所有人能免费使用为前提，否则就不能申请。

准确的关于拷贝、发行和修改的规定如下。

软件拷贝，发行和修改的规定和条件

这个许可证适用于任何有如下条款的程序或其他任何作品，即版权上者同意根据通用公共许可证的条款发行其软件。以下所说的“ 程序 ”指的都是这样的程序和作品。一个“ 基于程序的作品 ”指的既可以是原程序或依据版权法规定由原程序派生出来的作品。也就是说包

括原程序的程序或者原程序的一部分的作品。无论是完全相同或是有所改动或被翻译成为另一种语言(在下文中，翻译被认为是一种“改动”行为，而“您”指的是同一个持有许可证的人)。

本许可证不涉及拷贝、发行和修改的行为，它们超出了本书的范围，运行程序的行为不受限制，对于程序的输出也只涉及当它的内容包括了有关程序本身内容时(而不是由运行程序后产生的所有结果)。以上是否适用应取决程序的作用。

1. 您可能会使用各种介质把收到的程序源代码原封不动地拷贝和发行。但必须在每一份拷贝上准确而明显地印上著作权条例和放弃授权的权利；并保持所有关于这个许可证和不提供任何担保的条款的完整性。而且在把程序给辊人的同时加上一份这个许可证的拷贝。可以收取邮寄拷贝的邮资，而且也可以收取费用以提供授权保护。

2. 您可以对拷贝的程序或它的一部分进行修改，形成一个基于原程序的作品。并在第一条款的约束下拷贝和发行它。同时您还必须满足下面的条件。

a) 必须在修改的文件上加上明显的标记说明您已经对文件进行了修改以及您所　做任何修改的日期。

b) 必须把发行或出版的作品，无论是整体包含或只包含了一部分或是从程序及　它的部分产生出来，都应一起无偿地根据这个许可证对第三方厂商授权。

c) 如果这个改动的程序在运行时是命令交互的，在开始进行正常地交互使用之前，必须使之打印或显示一个包括恰当的版权说明和注意事项的申明，该注意事项应包括：本程序没有被授权(或者说明您们提供授权)，使用者可以在一定条件下重新发行它们，以及告诉用户如何阅读这份许可证(除非这个程序本身交互但无法打印出这样的声明，或您基于该程序的工作不需要打印一个声明)。

这些要求统统适用于修改过的文件，如果这种文件中一些独立部分不是从源程序中产生，而且有理由认为它们本身就是独立的作品，那么在您把它们作为独立作品发行时，这个许可证及其条款就不再适用了。但当您把同样部分作为一个基于原程序的作品的一部分发行时，整个程序的发行都必须要符合这个许可证的条款。本许可证对于其他许可证持有人的许可适用于整个软件，也同样适用于其中的每一部分，无论作者是谁。

因此，这一条的目的并不是要获得权利或是争夺您对您所创造的作品所拥有的权利；相反，其目的只是实现对基于原程序的任何派生的或集成的程序发行进行控制。

另外，当您把该程序和不是基于该程序的作品简单地集成到一起存入内存或发行介质中时，本许可证并不适用于另外的作品。

3. 您有权根据条款1和条款2拷贝和发行程序(及基于它的作品，见条款 2)的源代码或可执行代码形式，但您必须做到以下条款中之一。

a) 与之一起发行相应的完整的机器可读源代码，但必须根据条款 1和条款2的规定把它装在常用于软件发行的介质上发行。

b) 与之一起提出一个至少在3年内有效的，为第三方厂商提供相应的完整的机器可读的源代码的拷贝的提议，但所收取的费用不能超过您邮寄的费用，并且要根据条款　1和条款2的规定把它装在常用于软件发行的介质上发行。

c) 与之一起发行有关如何提出这个提议的信息(这种选择只有当软件发行是非商业性的，并且您是以目标码和可执行文件的形式得到的才有效，与以上 b条款相适应)。

一个作品的源代码是该作品容易被修改的形式，对于可执行文件，完整的源代码等于它所包括的所有模块的源代码，加上任何相关的界面定义文件及控制编译和安装的脚本。然而，作为一个特别发行的源代码不需要包括那些经常被发行的软件(既可能是源代码又可能是二进制代码的形式)和运行可执行文件的操作系统的主要成分(编译器，内核等等)，除非可执行文件本身就带有这些成分。

如果目标代码和可执行文件的发行是通过指定的地点来提供拷贝服务，那么尽管第三方厂商可以在拷贝目标代码的同时拷贝源代码，但这种行为被视为发行源代码。

4. 您无权拷贝、修改、授予证书或发行程序，除非本许可证已经明确规定了。除此之外任何试图拷贝、修改、授予证书或发行程序的企图都是无效的，并自动中止您根据本许可证所享有的权利。

然而，根据本许可证从您这得到拷贝和授权的社团的许可证不会被终止，只要它们完全遵守本许可证。

5. 在没有签署许可证之前，您不需要接受这个许可证。然而没有别的东西能授予您修改和发行程序及其派生作品的权利，任何上述行为都被禁止。因此，当修改和发行程序(或基于该程序的任何作品)时，已经表明接受了这个许可证，及其他所有对于拷贝，发行或修改程序及基于它的作品的规定和条件。

6. 每一次重新发行程序(或任何基于这个程序的作品)，接受者自动从最初的许可证执有人接受拷贝、发行或修改程序的许可证，您无权在接受者执行其由这个许可证所赋予的权利之上加上任何其他的限制，也无需强制第三方厂商遵守这个许可证。

7. 如果由于侵犯他人的专利权或因为其他的原因(不仅限于专利问题)而被指控并遭受判决，对您强制执行一些规定(无论是根据法庭的判决，书面协议或其他的途径)，而这些规定与这个许可证相矛盾时，不能因为这些规定而违反本许可证。如果发行软件时不能同时遵守本许可证的规定和其他相关的义务，那么就不能发行这个程序。例如：专利许可证不允许在不交版权费的情况下由任何直接或间接从您这里得到程序的人来发行它。因此如果既要满足它的限制又能符合本许可证的规定就只能不发行该程序。

如果在任何特殊环境下这一条款的任何部分无法实行或无效时，那么本条款的宽容性就适用了，并且在其他的情况下，本条款可以完全适用。

本条款的目的并不是让您违反任何专利权或其他所有权，以及辩驳任何权利的有效性。本条款的唯一目的是为了保护软件发行系统的一致性。而这个系统是由公共许可证来实现的。很多人通过这个一致性系统为广泛发行软件做出了无私的贡献。应该由作者或赠予者来决定他或她是否愿意通过其他系统来发行软件，而不能由许可证执有人来作这个决定。

本条款的目的是要清楚地说明违反这个许可证的其余部分的后果。

8. 如果在某些国家发行和/或使用程序的权利受到专利法或版权法的限制，原先把作品置于本许可证保护下的版权所有人可以增加一条地理发行的限制，以把这些协家排除在外，也就是说在其他的国家内才能发行软件。在这种情况下许可证与这些附加条款结合成一个整体。

9. 自由软件联盟可以随时发表修订过的或全新的通用公共协议。这样的新版本和现在的版本应没有本质的不同，但可能在细节上有所区别以处理一些新问题和相关的事情。

10. 如果您想要把程序的某些部分加入一些发行条件完全不同的其他免费软件中，应写信给作者获得允许。对于版权属于自由软件联盟的软件，则应首先写信给自由软件联盟以获得

它的允许，有时我们也能通融一下。影响我们最后作出决定的有两个因素，要保护从我们的免费软件中派生出来的作品的免费状态及推动软件的共享和重用。

不承担任何责任

11. 由于这里的程序免费地颁发许可证，因此对软件所导致的任何问题在相应的法律所允许的范围内不承担责任。版权所有者和 /或其他第三方厂商在提供程序时并不提供任何书面的或隐含的担保，但包括但不限于对于可销售性和适用于某一特定用途的隐含的担保。您要承担由程序的质量和性能所带来的全部风险。假如程序中有缺陷，您应承担所有必须的服务、修复和更正的费用。

12. 版权所有者或其他在以上条款规定下任何可能修改和 /或发行程序的人，除非在适用的法律或书面的协定规定下，在其他任何条件下对所导致的破坏都不负责任。这些破坏包括任何使用和不能使用程序所导致的普通的，特殊的和偶然的错误(包括但不限于数据丢失或使数据不准确，由您或第三方厂商所导致的损失，以及程序和其他程序兼容性的问题)，即使您和其他方已经知道这种损坏的可能性。

如何在您的程序中使用这些条款

如果您编写了一个新程序并希望它能够最大限度地为公众所用，最好的实现方法是把它变成一个免费软件，那么无论谁都能在这些条款的许可下发行和改变它。

要这么做，应在程序中加入以下附注。最稳妥的方法是把它们加在每个文件的开头以最有效地说明不做如何担保：每一个文件至少都应包含有关版权的行和一个指向完整条款的指针。

<用一行说明程序的标题和关于它的功能的简要说明 >

Copyright © 19yy(作者的名字)

这个程序是免费软件；您可以根据由自由软件联盟所出版的通用公共许可证的条款重新发行和/或改变它，既可以根据它的第二版也可以自由选择更新的版本。

发行这个程序是希望它能所作用，但并不做任何的担保；甚至没有保证其商业性或适用于某一特定的目的，参见通用公共许可证(GNU)中的详细条款。

您应该在得到这个程序的同时得到通用公共许可证(GNU)的拷贝，如果没有的话，请写信向以下地址索取。

Free Software Foundation,Inc.,59 Temple Place,Suite 330,Boston,MA 02111-1307 USA.

请同时附上您的电子邮件地址和您的信件的地址。

如果程序是人机交互的，让它在开始进入交互状态之前输出一个简短的消息：

Gnomovision version 69, Copyright © 19yy name of author

Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type

‘ show w ’

This is free software, and you are welcome to redistribute it under

certain conditions; type ‘ show c ’ for details.

假想的命令show w和show c应能够显示出通用公共许可证的恰当的部分，当然可以使用和show w和show c不同的名字作为命令的名字；甚至可以用鼠标点击或使用菜单项 —— 只要适合您的程序。

如果您是一个程序员，您也可以让您的雇主或学校(假如有的话)，在必要的情况下为您的程序签署一个放弃版权的文件。下面有一个例子，您可以在作了适当的改动后使用它。

Yoyodyne, Inc., hereby disclaims all copyright interest in the program

“Gnomovision”(which makes passes at compilers) written by James

Hacker.

(signature of Ty Coon), l April 1989

Ty Coon, President of Vice

这个通用公共许可证不允许把您的程序加入别人的程序中。如果您的程序是一个子例程库，您认为把您的库和别人的应用程序连接起来更有用。如果您想要这样做，您不应使用这个许可证而应该使用GNU通用公共许可证库。