

# 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。

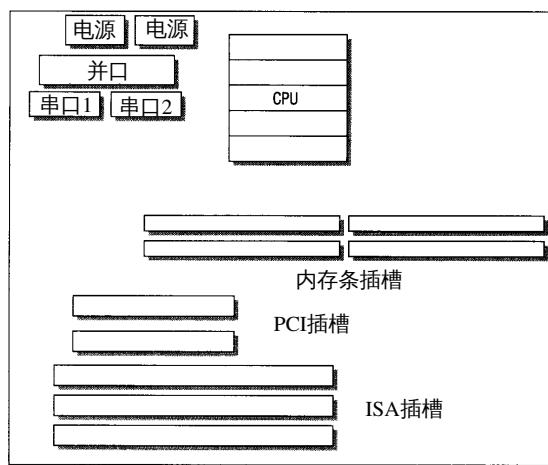


图1-1-1 典型的PC主板

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 $10 \times 10$ ，10的3次幂( $10^3$ )是 $10 \times 10 \times 10$ ，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。

[下载](#)

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 Programming 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 ;
```

**下载**

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          tc1
$
```

\$ 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IDE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

[下载](#)

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。

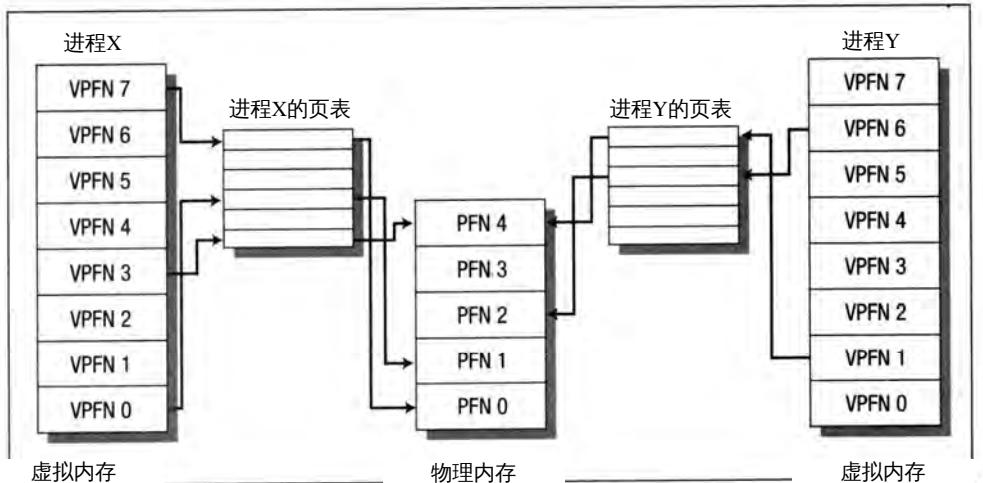


图1-2-1 虚拟地址到物理地址映射的抽象模型

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 ( $4 \times 0x2000$ )。加上 0x194 字节的偏移，最后得到物理地址 0x8194。

**下载**

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收回内存。

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一件非常痛苦的事情。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0xfffffc0000000000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0xfffffc0000310000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：

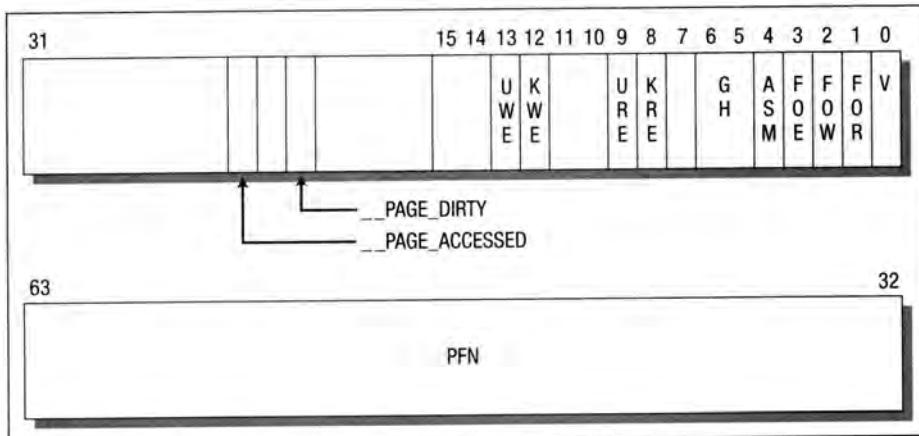


图1-2-2 Alpha AXP页表项

- 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，把它转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。

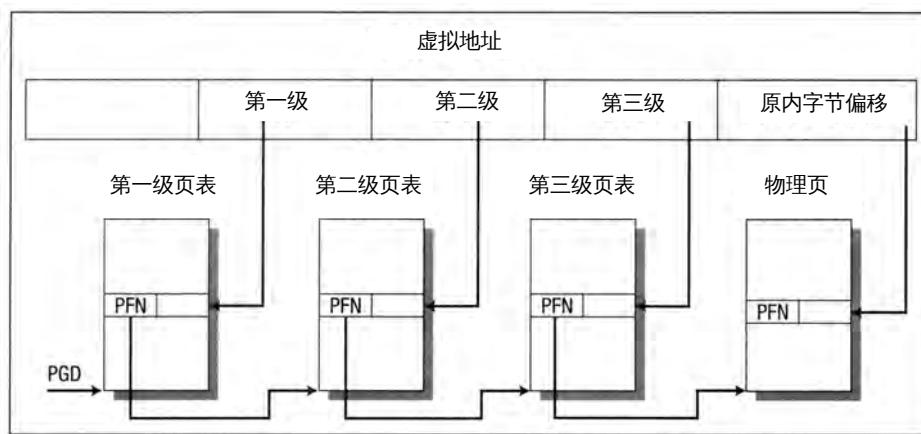


图1-2-3 三级页表

下载

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候选。
- `map_nr` `mem_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。

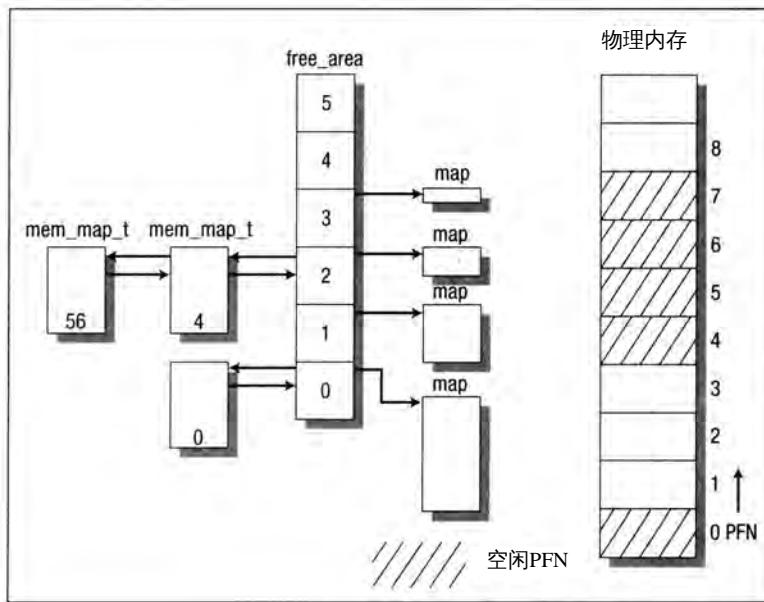


图1-2-4 free-area数据结构

### 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来满足请求 (`nr_free_pages > min_free_pages`)，分配代码将搜索`free_area`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 (memory 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_struct`数据结构将被产生。

[下载](#)

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。

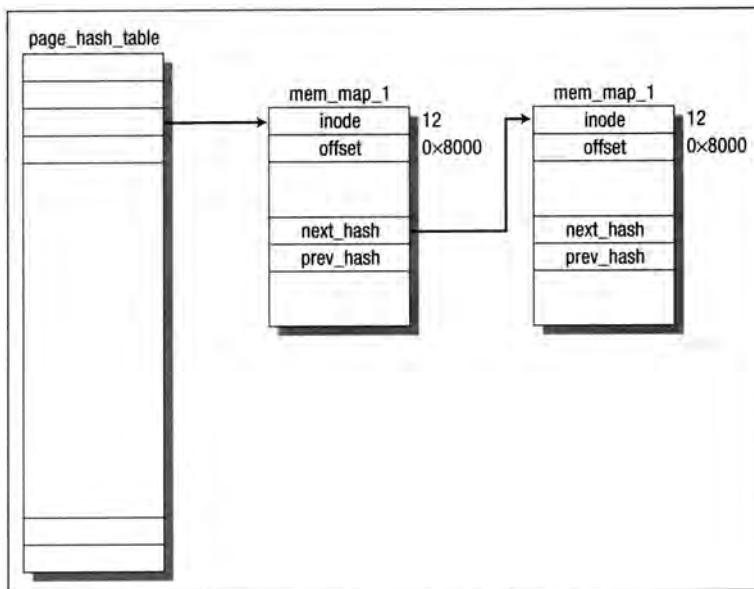


图1-2-5 虚拟内存区

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。

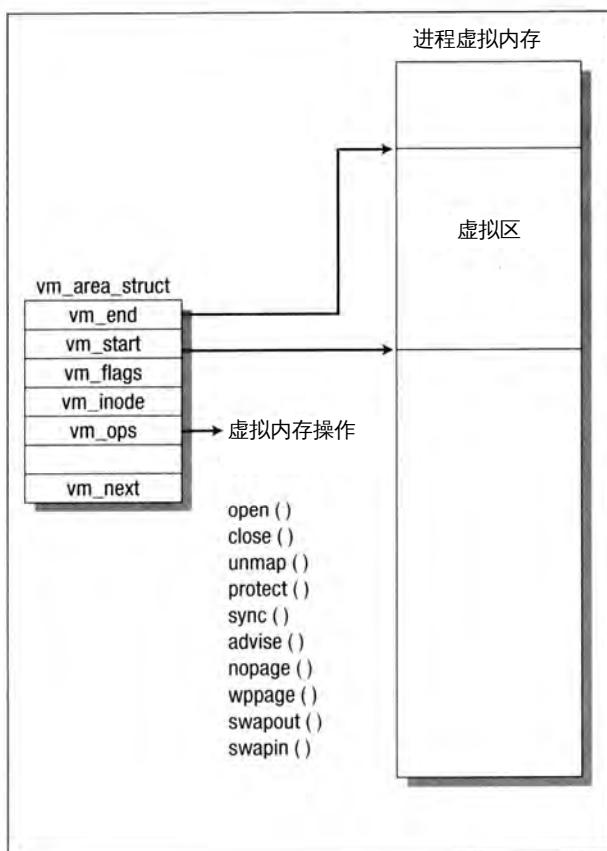


图1-2-6 Linux页缓存

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

[下载](#)

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(或者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钟算法。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 `swapon` 操作。如果有此虚拟内存区的 `swapon` 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 `swapon` 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)
|-kswappd(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

**下载**

“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 UNIX™ 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调度策略：“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 jiffies 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

**下载**

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 (last\_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自由改变。

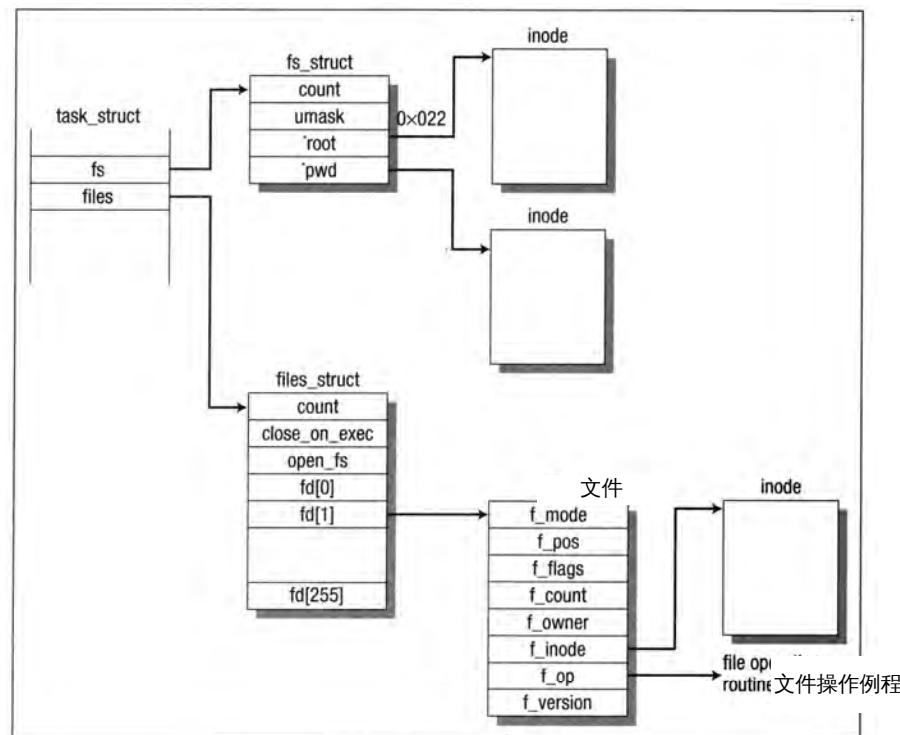


图 1-3-1 进程的文件

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

[下载](#)

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。

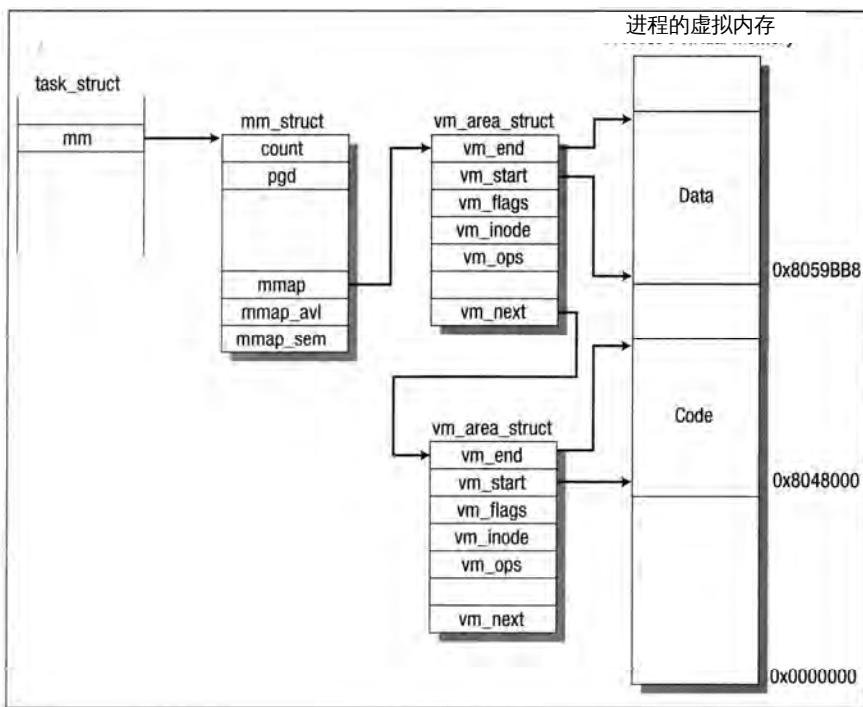


图1-3-2 进程的虚拟内存

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

**下载**

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，或者说等待子进程退出。你可以通过把子进程推到后面来使 shell 再次运动。敲入 control\_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。

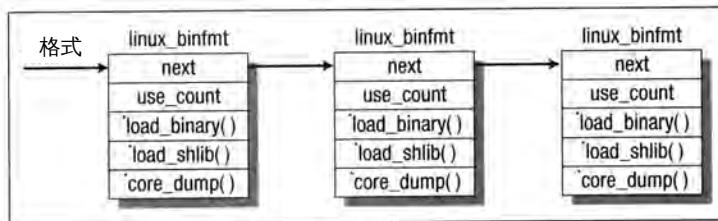


图1-3-3 注册的二进制格式

下载

## 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。

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

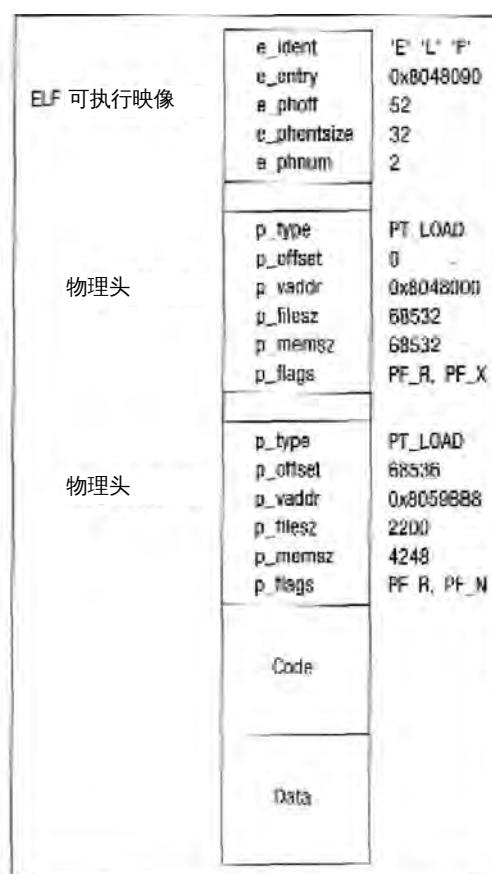


图1-3-4 ELF可执行文件格式

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储，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 结构中存储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：当前进程正在核心态中运行，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帧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，

**下载**

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。

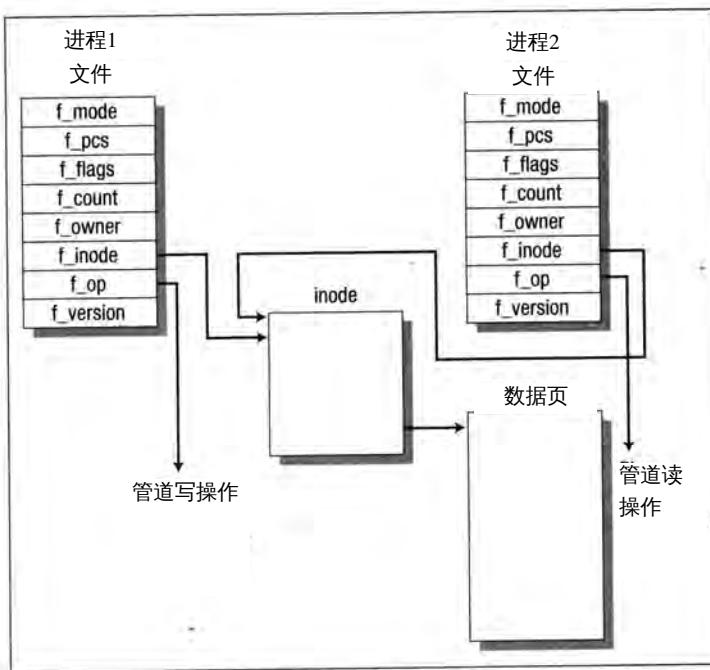


图1-4-1 管道

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，都使用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

[下载](#)

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。

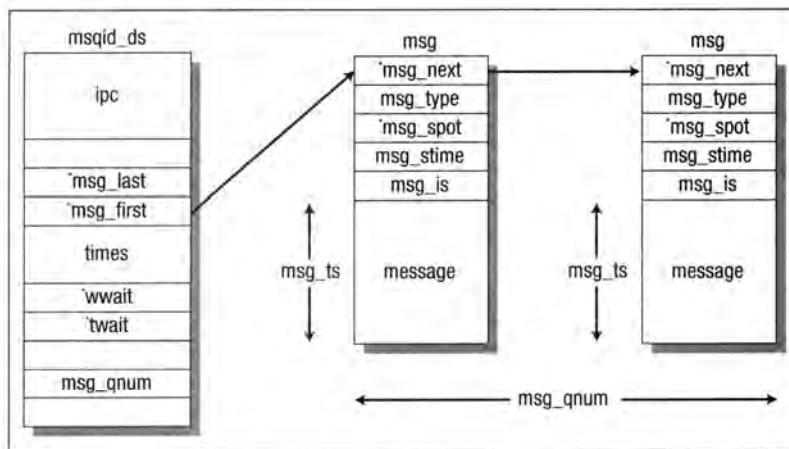


图1-4-2 System V IPC 消息队列

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\_pending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。

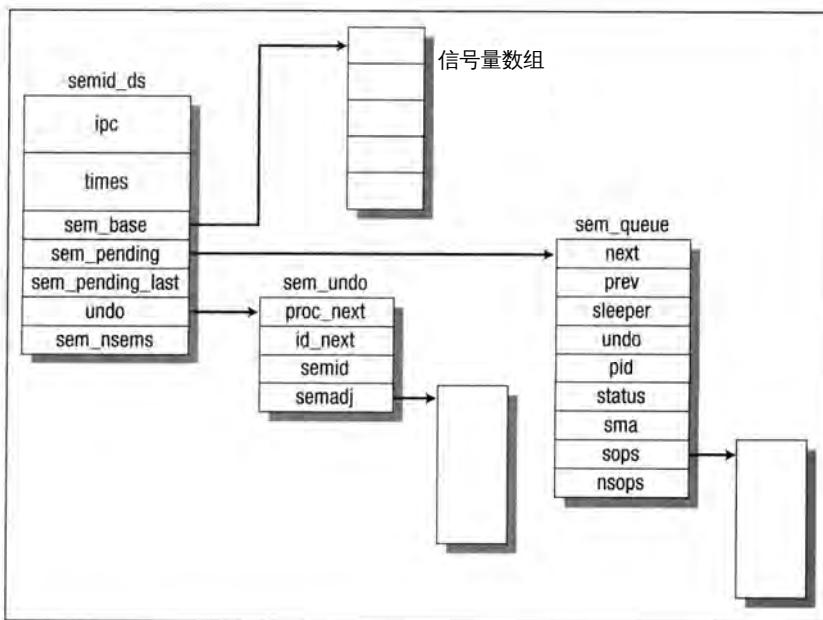


图1-4-3 System V 信号量

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，

[下载](#)

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。

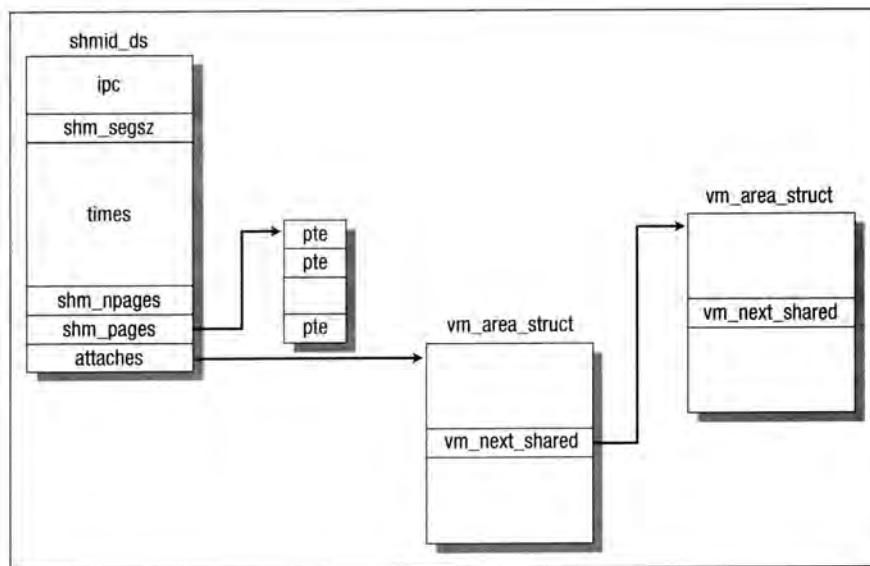


图1-4-4 System V IPC 共享存储区

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。

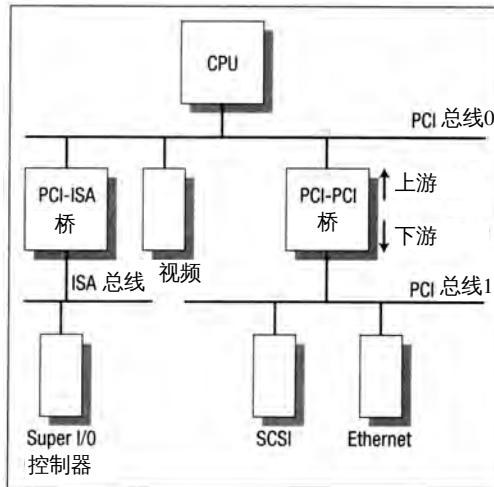


图1-5-1 基于PCI局部总线系统的示例

### 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拓扑结构图中的位置。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Digital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(Class 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。
- 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

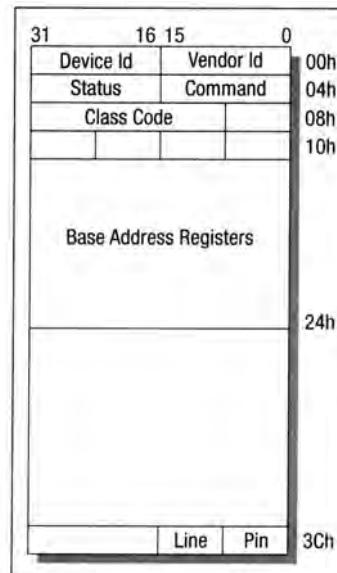


图1-5-2 PCI配置头

[下载](#)

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设备是不可访问的。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和存储器地址空间。

## 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和存储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和存储器地址空间，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。

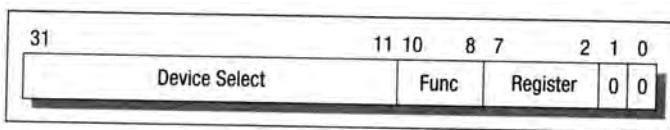


图1-5-3 0型PCI配置周期

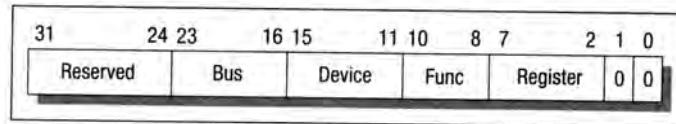


图1-5-4 1型PCI配置周期

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

下载

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。

### 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，

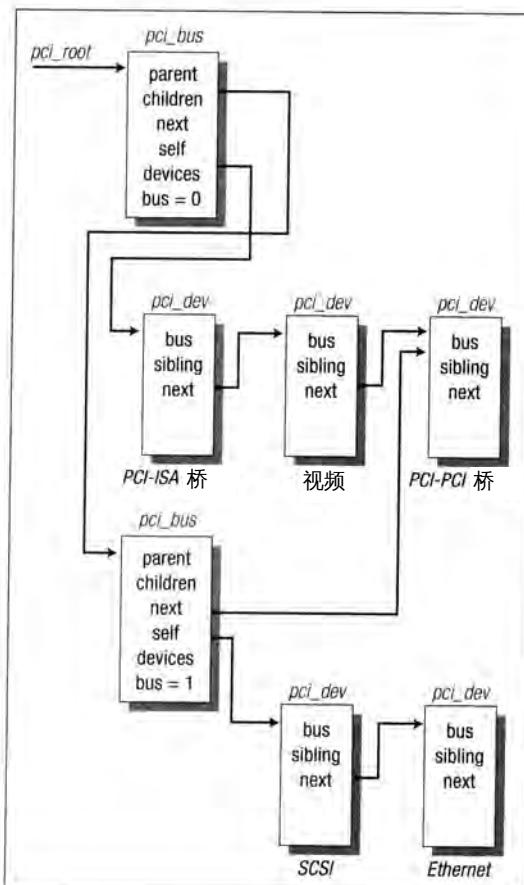


图 1-5-5 Linux 内核中 PCI 数据结构

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 pic\_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。

下载

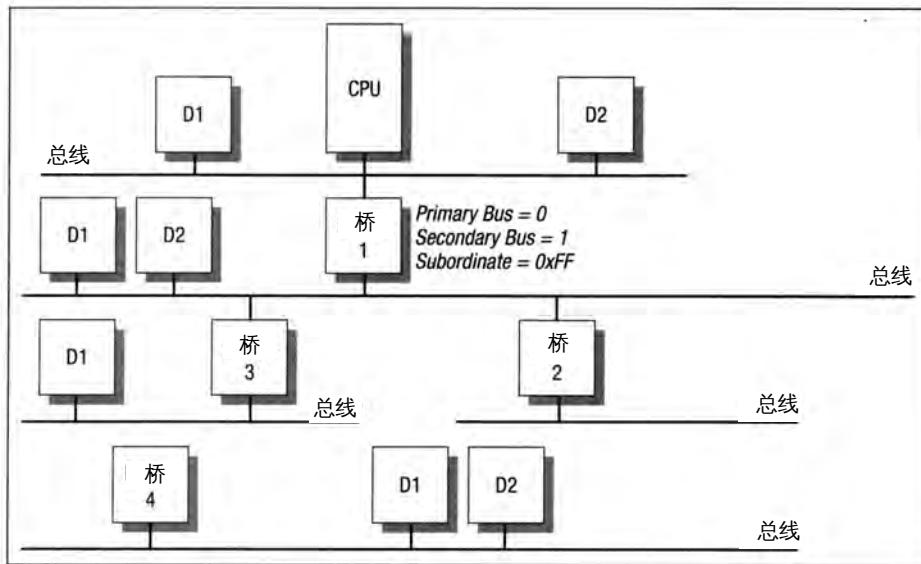


图1-5-6 配置一个PCI系统：第1步

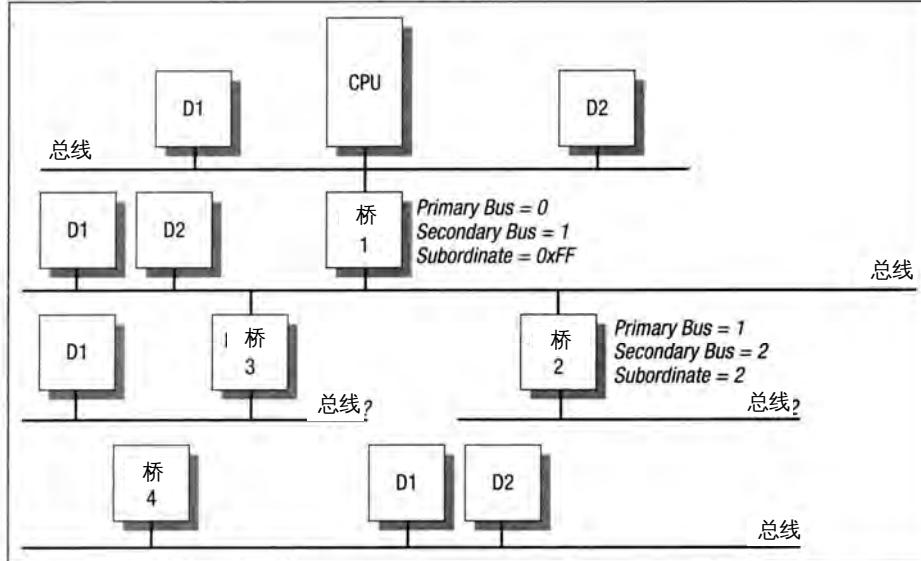


图1-5-7 配置一个PCI系统：第2步

### 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。

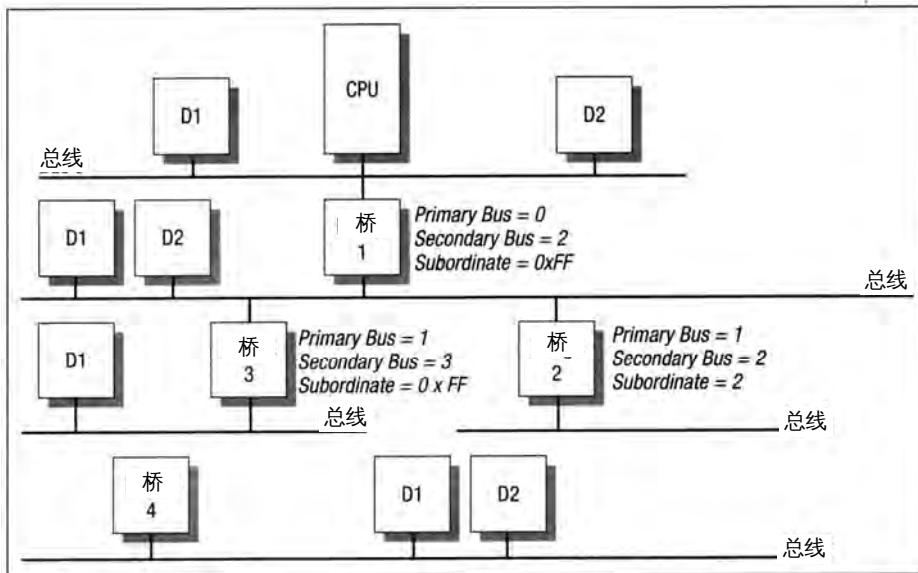


图1-5-8 配置一个PCI系统：第3步

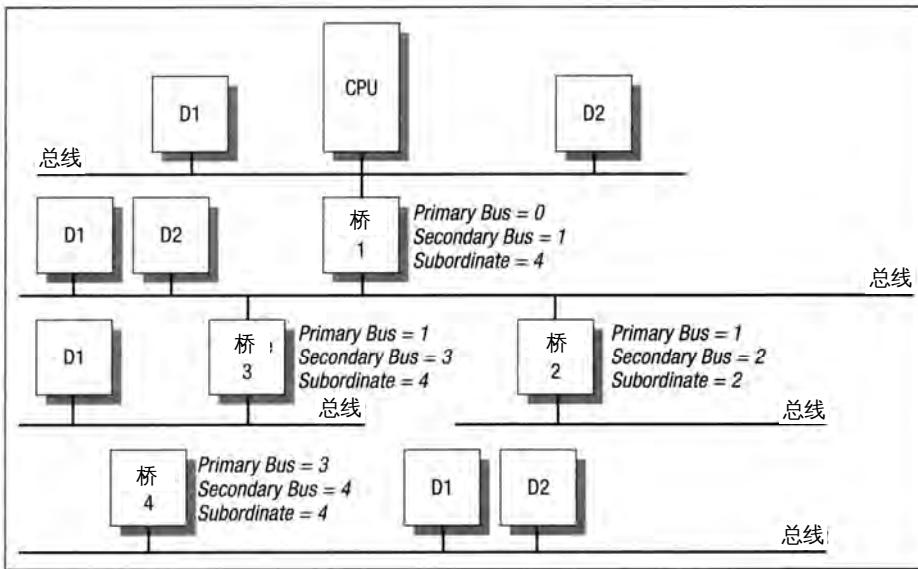


图1-5-9 配置一个PCI系统：第4步

### 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。

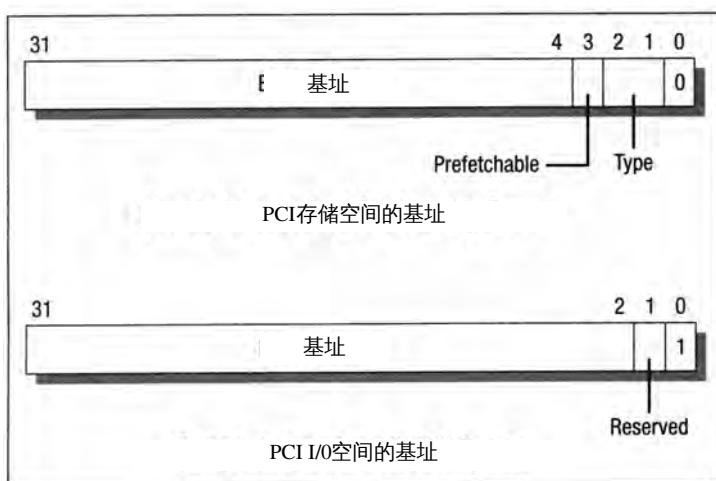


图1-5-10 PCI配置头：基址寄存器

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

[下载](#)

常要有用于存储CPU处理中断前执行状态的栈(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存储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寄存器是只写的，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。

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

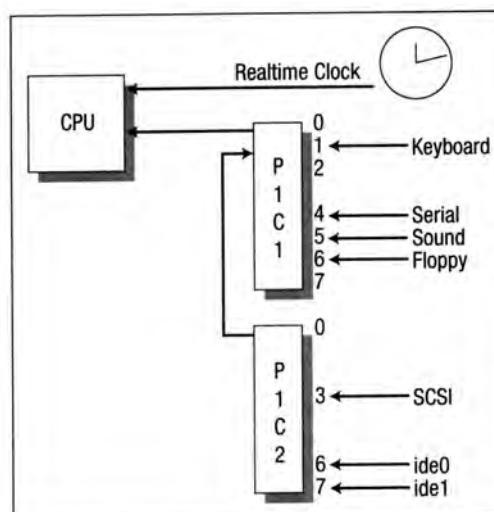


图1-6-1 中断路由逻辑图

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

下载

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 vector向量的大小取决于系统中中断源的数目。

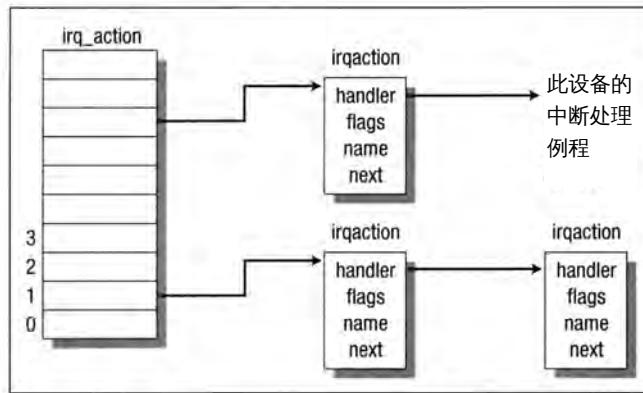


图1-6-2 Linux中断处理数据结构

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/cua0和/dev/cua1。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。

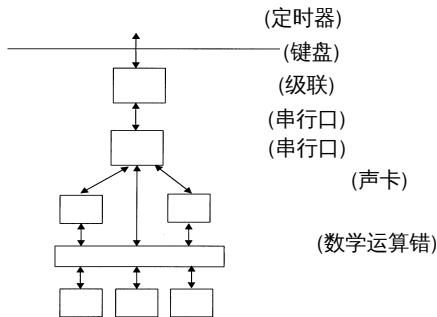
### 6.2.1 测试与中断

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，如果不间断测试设备的话，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。

[下载](#)

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。

下载

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。

### 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

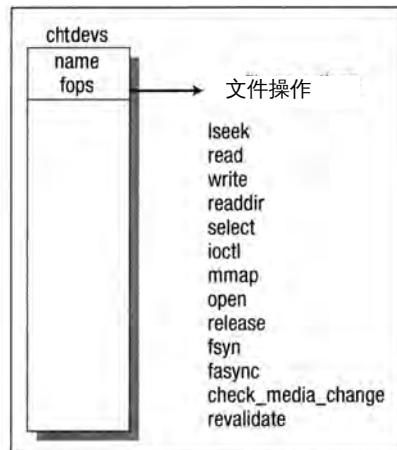


图1-6-3 字符设备

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。

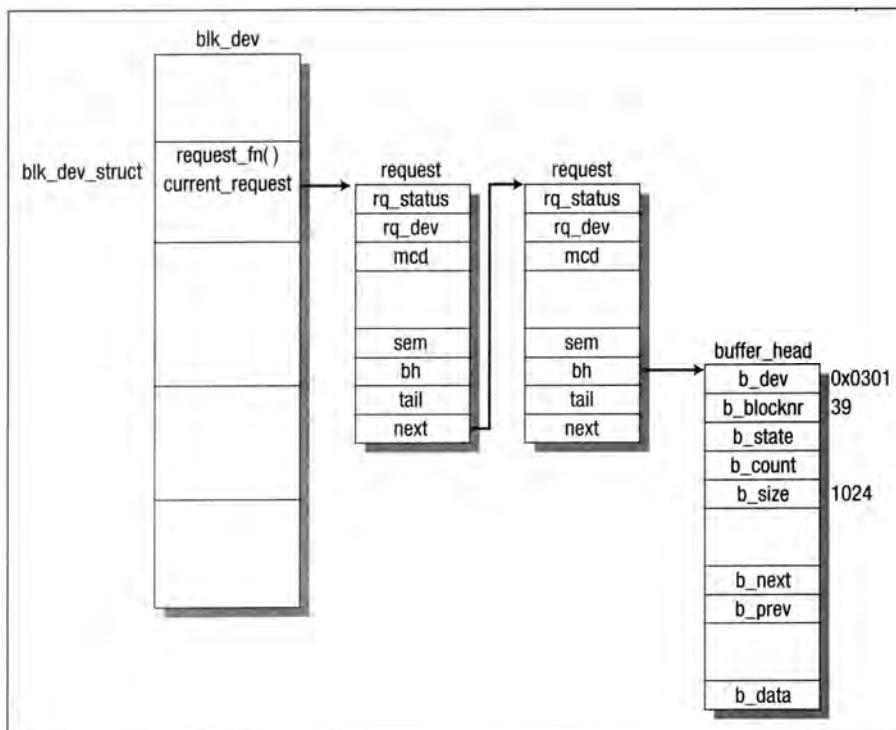


图1-6-4 缓冲区缓冲块设备的请求

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。

[下载](#)

## 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。

每个盘片的表面被划分为同心圆——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，整个盘的存储容量为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。

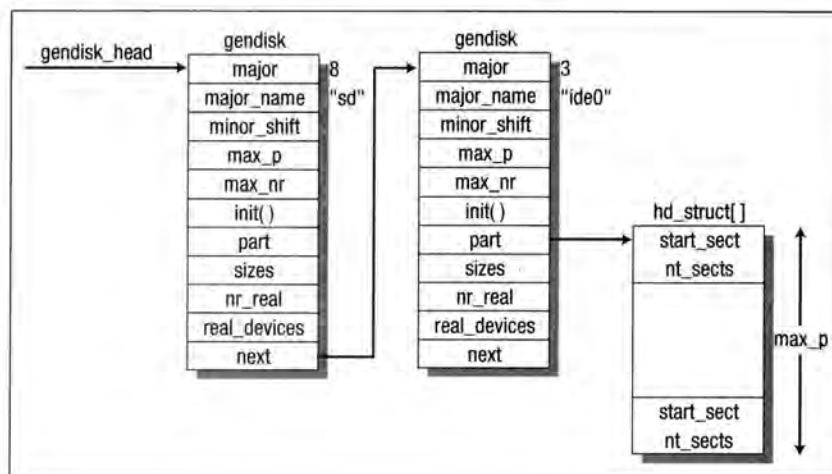


图1-6-5 硬盘表链

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，

[下载](#)

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\_hwif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 `builtin_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

下载

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初始化程序通过向它发送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。

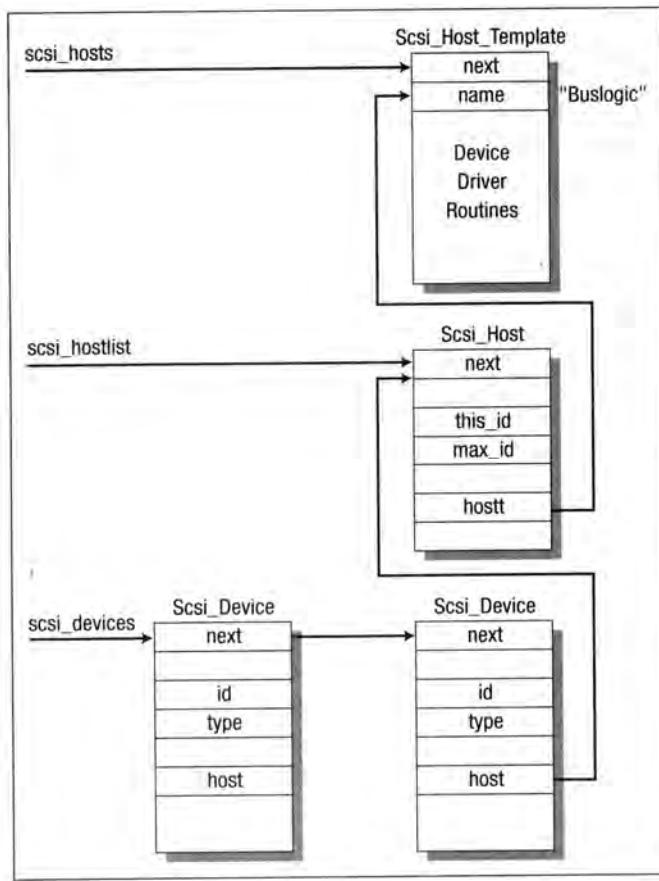


图1-6-6 SCSI数据结构

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

[下载](#)

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/sIN	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_NOTAILERS	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

下载

程序的工作。

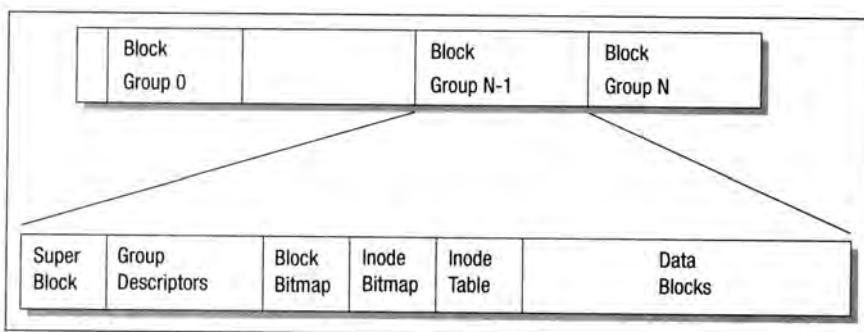


图1-7-1 EXT2文件系统的物理结构

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

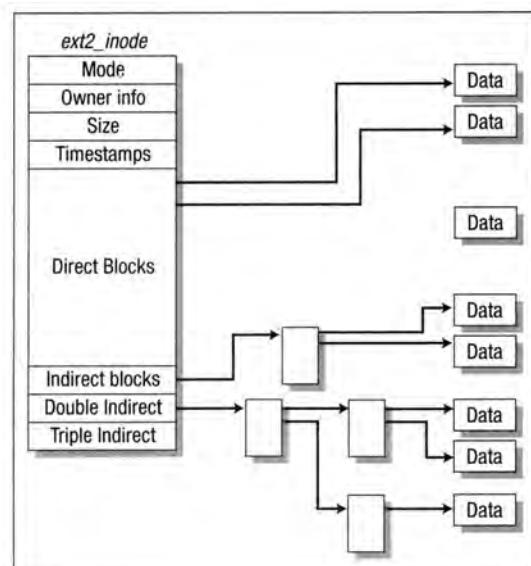


图1-7-2 EXT2的inode结点

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 efsck”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。

下载

### 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。

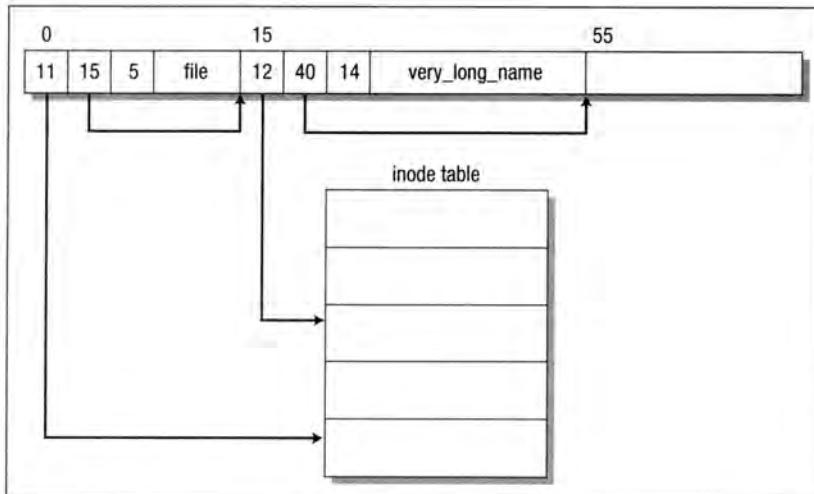


图1-7-3 EXT2系统的目录

- 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

[下载](#)

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。

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

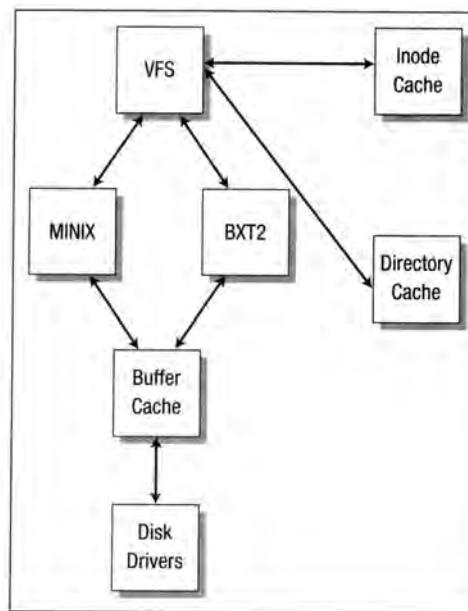


图1-7-4 VFS文件系统的逻辑结构图

方式来加速文件系统对它们对应的物理设备的访问。缓冲区缓存是独立于文件系统的，并被集成成为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 /dev/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储空间中，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。

下载

- 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；在卸载时向内核注销。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\_system\_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。

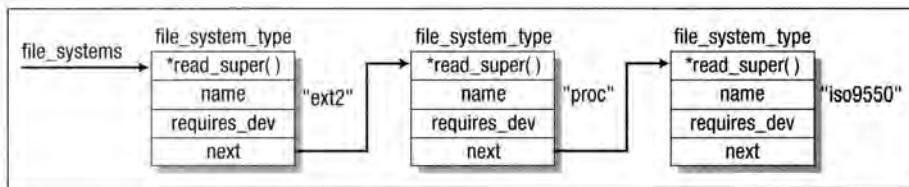


图1-7-5 注册的文件系统

通过查看/proc/filesystems，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命令传送给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。

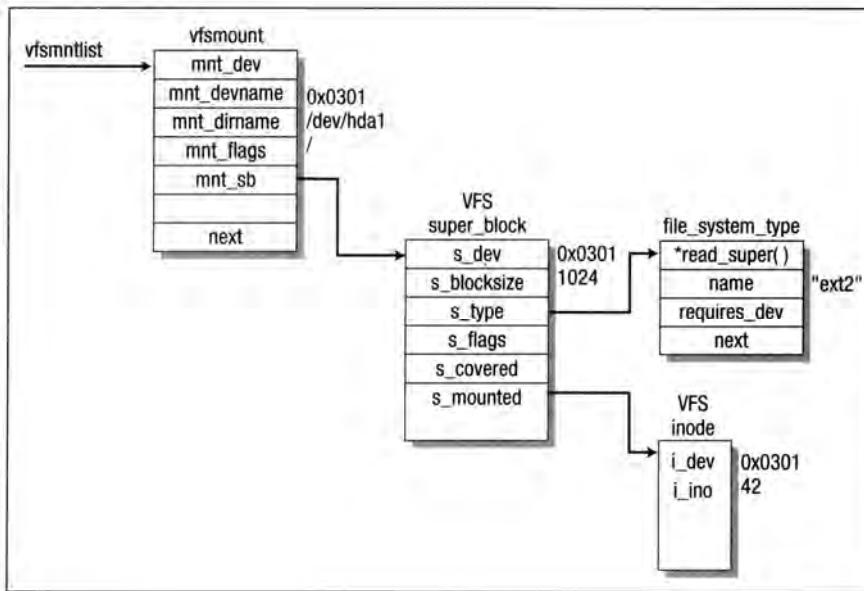


图1-7-6 安装的文件系统

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

[下载](#)

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超级块占用的存储区就被归还给内核的自由存储区池。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

[下载](#)

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：

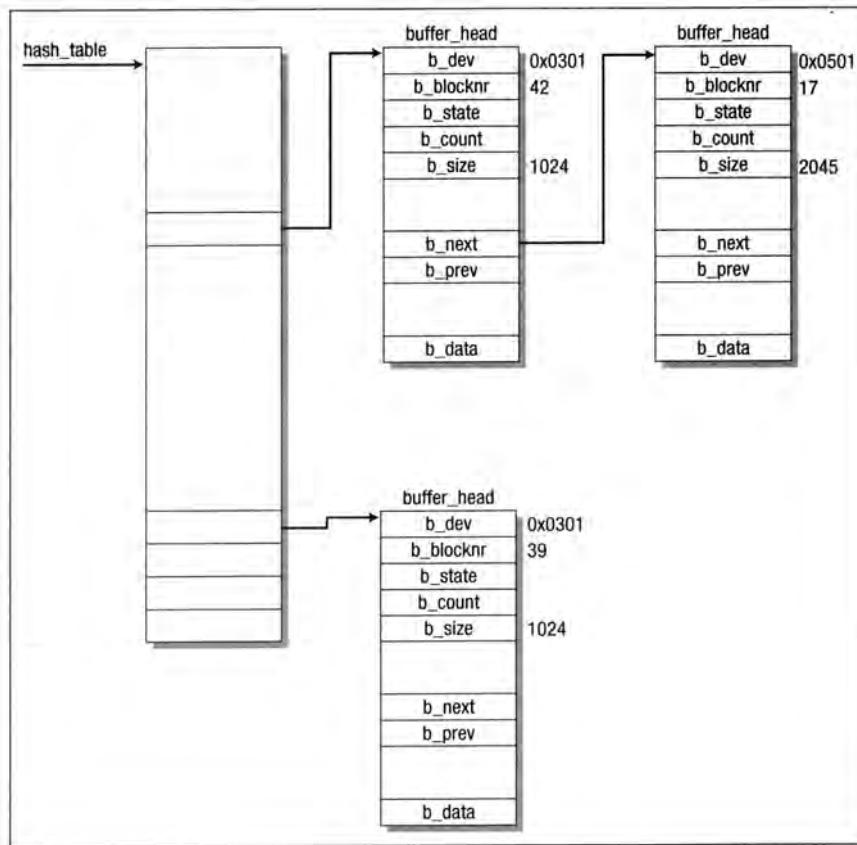


图1-7-7 缓冲区缓存

- 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

[下载](#)

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 `bdfflush` 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的研究，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和全世界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(Domain 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

下载

(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、校验和其他一些有用信息(详见图1-8-1)，其中校验是由源主机利用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 Protocol, 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播图片，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。

VERS	HLEN	SERVICE TYPE	TOTAL LENGTH			
IDENTIFICATION		FLAGS	FRAGMENT			
TTL	PROTOCOL	HEADER	CHECKSUM			
SOURCE IP ADDRESS						
DESTANATION IP ADDRESS						
IP OPTIONS		PADDIN				

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 net/address	Gatevay address	Flags	Refcnt	Use	Iface
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层直接报文。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

下载

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，以太网上将会有许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

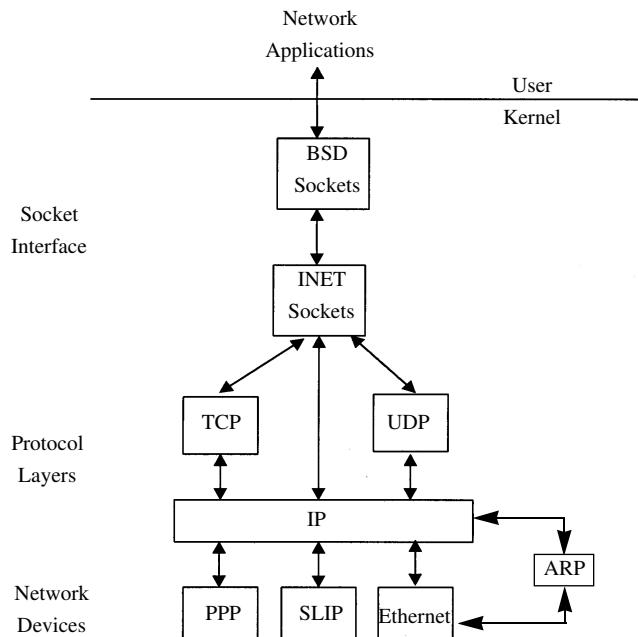


图1-8-4 Linux网络层次结构

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

[下载](#)

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地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。

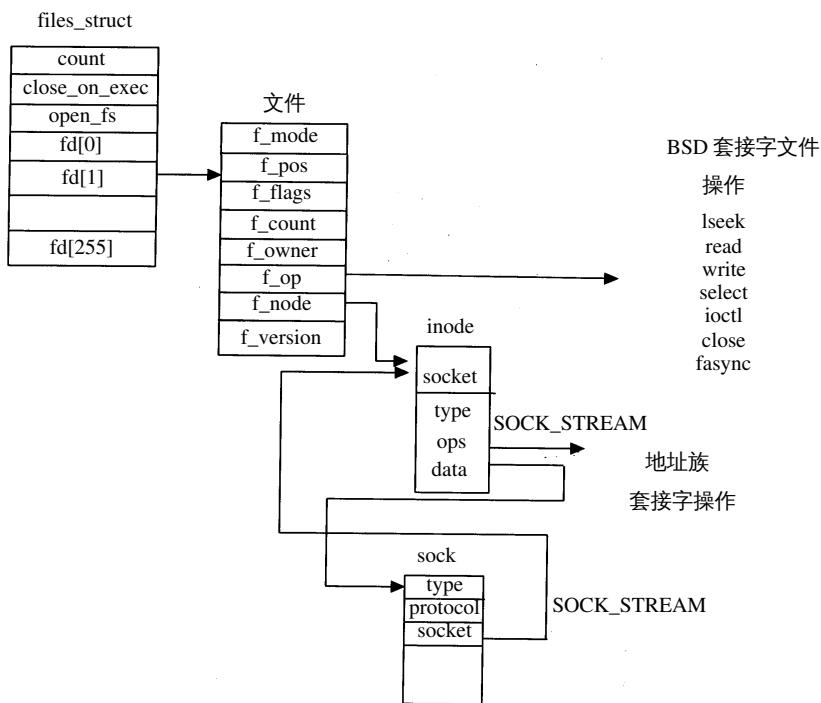


图1-8-5 Linux BSD套接字数据结构

#### 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 `proto_ops` 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。

[下载](#)

#### 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\_listony\_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数据结构的指针。

#### 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：

下载

- 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。

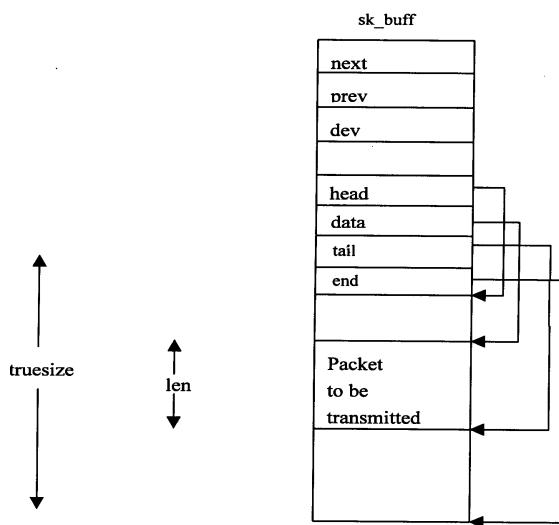


图1-8-6 套接字缓冲区

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。

ETHERNET FRAME				
Destination Ethernet address	Source Ethernet address	Protocol	Data	Checksum

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

下载

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进行重组。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。

由于网络拓扑结构可以随时改变，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

**下载**

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，就将其引用计数增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，就将引用计数减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。

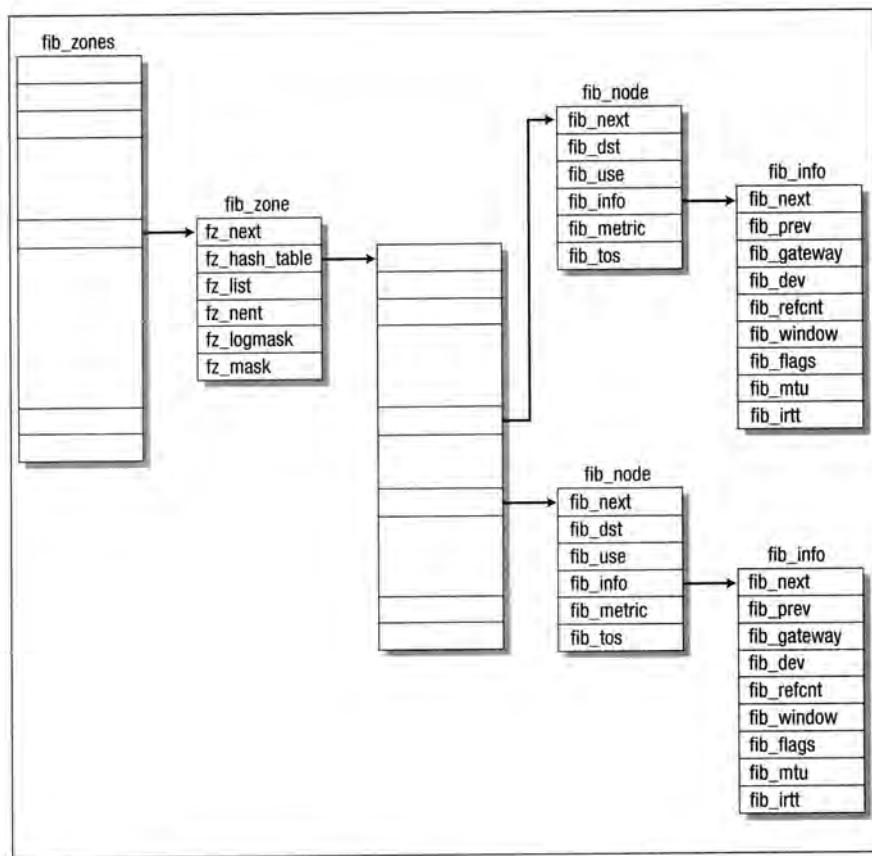


图1-8-9 转发信息数据库

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。

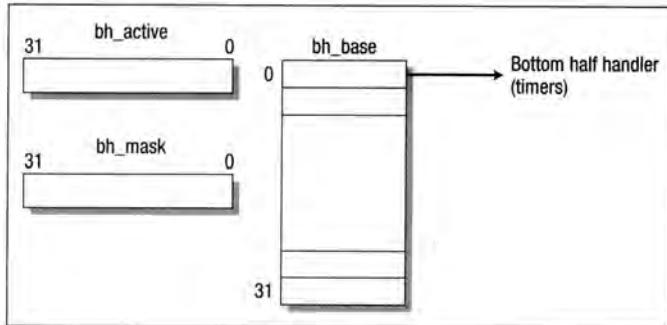


图1-9-1 Bottom Half控制数据结构

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。

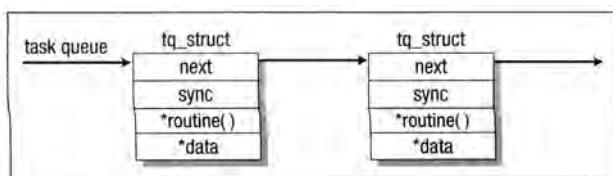


图1-9-2 任务队列

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

[下载](#)

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。任何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。

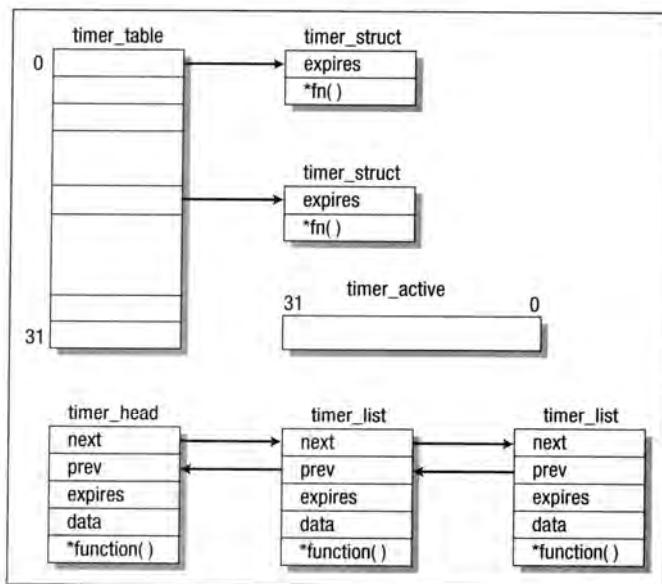


图1-9-3 系统定时器

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。

加入到等待队列中的进程可以是可中断的，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。

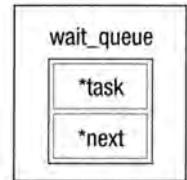


图 1-9-4 等待队列

**下载**

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

下载

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。

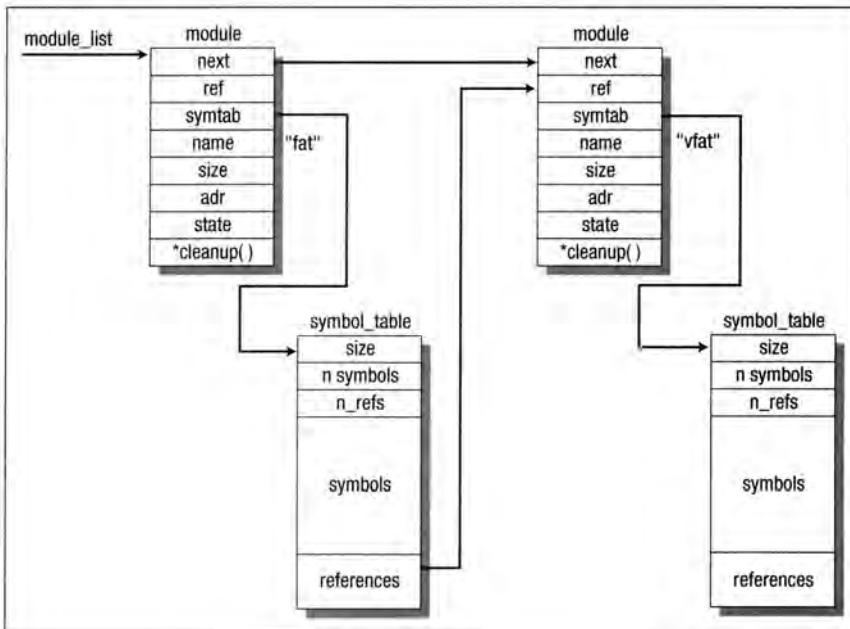


图1-9-5 内核模块表

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。

**下载**

- /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/ext2\_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 */
}
```

下载

```
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);
```

下载

```

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;
```

下载

```
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;
* i_hash_next, *i_hash_prev;
* i_bound_to, *i_bound_by;
* i_mount;
i_count;
i_flags;
i_lock;
i_dirt;
i_pipe;
i_sock;
i_seek;
i_update;
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;
```

下载

```
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
{

```

下载

```
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。

```

下载

```
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  lingeftime;
    int           proc;

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



下载

```
unsigned char          protocol;
volatile unsigned char state;
unsigned char          ack_backlog;
unsigned char          max_ack_backlog;
unsigned char          priority;
unsigned char          debug;
int                  rdbuf;
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;
#ifndef CONFIG_ATALK || defined(CONFIG_ATALK_MODULE)
    struct atalk_sock  af_at;
#endif
#ifndef CONFIG_IPX || defined(CONFIG_IPX_MODULE)
    struct ipx_opt     af_ipx;
#endif
#ifndef CONFIG_INET
    struct inet_packet_opt  af_packet;
#endif
#ifndef 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 ? */

#ifndef 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    *icomm;          /* 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           signal;
    unsigned           blocked;         /* bitmap of masked signals */
```

下载

```
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;
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;

#ifndef __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。

下载

```
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 */
};
```