# Linux Kernel核心中文手册

## Unix/Linux作坊

Chapter 1

Hardware Basic(硬件基础知识)

一个操作系统必须和作为它的基础的硬件系统紧密配合。操作系统需要使用一些只有硬件才能提供的功能。为了完整的了解Linux，你需要了解底层硬件的基础知识。本章对于现代PC的硬件进行了。

1975年1月“Popular Electronics”杂志封面上印出了Altair 8080的图片，一场革命开始了。

Altair 8080，跟随早期的“Star Trek epsode”命名，只需要$397，就可由个人电子爱好者自己组装。它拥有Intel 8080处理器和256字节内存，但是没有屏幕和键盘。以今天的标准来衡量，它太简陋了。它的发明者，Ed Roberts，制造了名词“personal computer“来命名他的发明，但现在，PC这个名词已经用来命名几乎所有你可以不依靠帮助就可以自己运行起来的计算机。用这个定义，甚至一些十分强大的Alpha AXP系统也是PC。

爱好者们看到了Altair的潜力，开始为它写软件，制造硬件。对于这些早期的先驱来讲，它代表着自由：从被神职人员控制和运行的大型批处理的主机系统中逃脱出来的自由。你可以在自己家里甚至厨桌上拥有计算机，这使学院的退学生为此着迷并通宵达旦。与此同时出现大量硬件，在一定程度上各自不同，而软件专家则乐于为这些新机器撰写软件。有讽刺意味的是，IBM在1981年发布了IBM PC并于1982年早期供货，从此定义了现代PC的模型。它拥有Intel 8088处理器，64K内存（可以扩充到256K），两个软驱和一个80x25的彩色图卡(CGA)，用今天的标准衡量，它功能不算很强，但是它销售的不错。1983年，紧接着推出的IBM PC-XT，则拥有一个豪华的10M硬盘。不久大批公司如Compaq开始制造IBM PC的复制品，PC的结构成为了事实的标准。这个事实的标准使大批硬件公司可以在这个不断增长的市场上一起竞争，反过来，可以遏制价格，让用户满意。现代PC承袭了早期PC的许多系统体系特征。甚至基于最强大的Intel Pentium Pro的系统也可以运行Intel 8086的寻址模式。当Linus Torvalds开始开发后来的Linux时，他选择了当时最常见和价格最合理的硬件平台：一台Intel 80386 PC。

从PC的外面看，最明显的部件就是机箱、键盘、鼠标和显示器。在机箱的前面有一些按钮，一个小屏幕显示一些数字，还有一个软驱。现在的大多数系统还有一个CD-ROM期、驱动器。如果你需要保护你的数据，那么还会有一个备份用的磁带机。这些设备一律被看作外设。

虽然CPU管理整个系统，但它并不是唯一的智能设备。所有的外设控制器，例如IDE控制器，也都拥有一定程度的智能。在PC内部（图1.1），你可以看到一个主板，包括CPU或微处理器、内存和一些ISA或PCI外设控制卡的槽位。其中一些控制器，如IDE磁盘控制器可能内置在系统主板上。

1. CPU

CPU，或者说微处理器，是所有计算机系统的心脏。微处理器进行数学运算，逻辑操作并从内存中读取指令并执行指令，进而控制数据流向。计算机发展的早期，微处理器的各种功能模块是由相互分离（并且尺寸上十分巨大）的单元构成。这也是名词“中央处理单元”的起源。现代的微处理器将这些功能模块集中在一块非常小的硅晶片制造的集成电路上。在本书，名词CPU、微处理器和处理器交替使用。

微处理器处理二进制数据：这些数据由1和0组成。这些1和0对应电气开关的开或关。就好像42代表4个10和2个单元，二进制数字由一系列代表2的幂数的数字组成。这里，幂数意味着一个数字用自身相乘的次数。10 的一次幂是10，10的2次幂是10x10，10的3次幂是10x10x10，依此类推。二进制0001是十进制1，二进制数0010是十进制2，二进制0011是十进制3，二进制0100是十进制4，等等。所以，十进制42是二进制101010或者（2+8+32或$2^1+2^3+2^5$）。在计算机程序除了使用二进制表示数字之外，另一种基数，16进制，也经常用到。在这种进制中，每一位数字表示16的幂数。因为十进制数字只是从0到9，在十六进制中10到15分别用字母A，B，C，D，E，F表示。例如，十六进制的E是十进制的14，而十六进制的2A是十进制的42（2个16+10）。用C语言的表示法（本书一直使用），十六进制数字使用前缀“0x”：十六进制的2A写做0x2A。

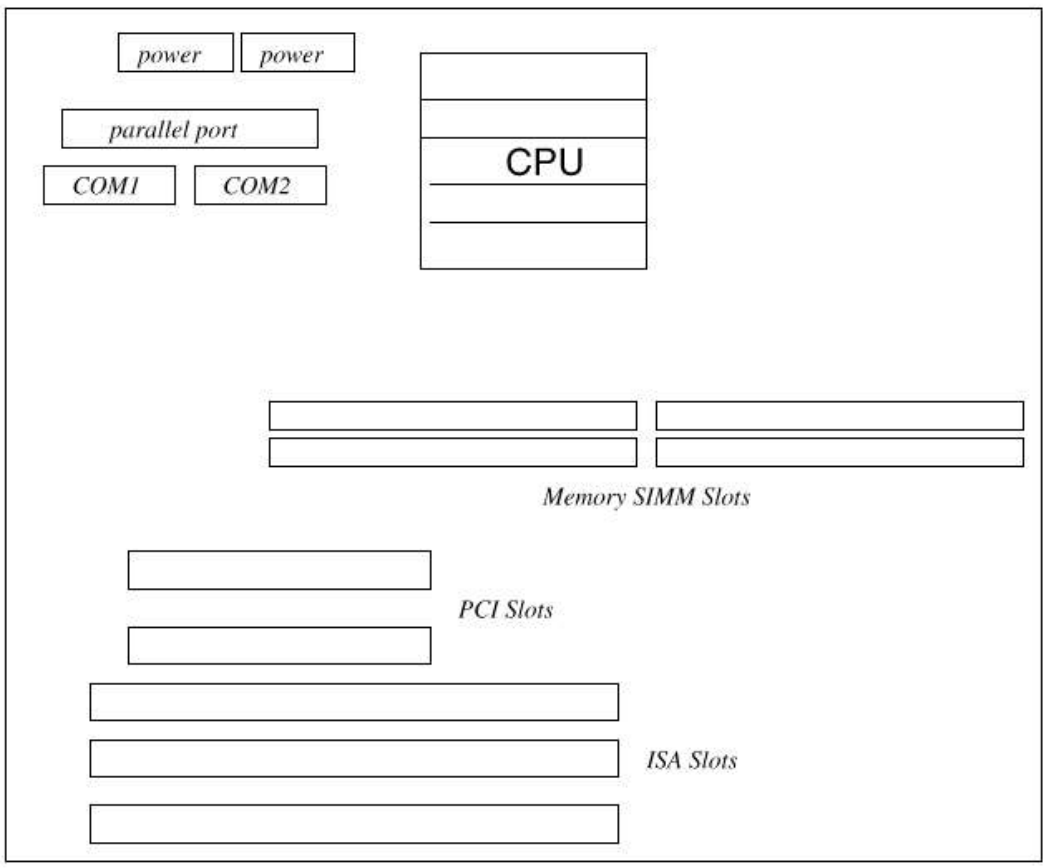



Figure 1.1: A typical PC motherboard.

微处理器可以执行算术运算如加、乘和除，也可以执行逻辑操作例如“X是否大于Y”。

处理器的执行由外部时钟控制。这个时钟，即系统时钟，对处理器产生稳定的时钟脉冲，在每一个时钟脉冲里，处理器执行一些工作。例如，处理器可以在每一个时钟脉冲里执行一条指令。处理器的速度用系统时钟的频率来描述。一个100Mhz的处理器每秒钟接受到100，000，000次时钟脉冲。用时钟频率来描述CPU的能力是一种误解，因为不同的处理器在每一次时钟脉冲中执行的工作量不同。虽然如此，如果所有的条件同等，越快的时钟频率表示处理器的能力越强。处理器执行的指令非常简单，例如：“把内存位置X的内容读到寄存器Y中“。寄存器是微处理器的内部存储空间，用来存储数据并进行操作。执行的操作可能使处理器停止当前操作而转去执行内存中其他地方的指令。正是这些微小的指令集合在一起，赋予现代的微处理器几乎无限的能力，因为它每秒可以执行数百万甚至数十亿的指令。

执行指令时必须从内存中提取指令，指令自身也可能引用内存中的数据，这些数据也必须提取到内存中并在需要的时候保存到内存中去。

一个微处理器内部寄存器的大小、数量和类型完全决定于它的类型。一个Intel 80486处理器和一个Alpha AXP处理器的寄存器组完全不同。另外，Intel是32位宽而Alpha AXP是64位宽。但是，一般来讲，所有特定的处理器都会有一些通用目的的寄存器和少量专用的寄存器。大多数处理器拥有以下特殊用途的专用的寄存器：

Program Counter（PC）程序计数器

这个寄存器记录了下一条要执行的指令的地址。PC的内容在每次取指令的时候自动增加。

Stack Pointer（SP）堆栈指针

处理器必须能够存取用于临时存储数据的大容量的外部读写随机存取内存（RAM）。堆栈是一种用于在外部内存中存放和恢复临时数据的方法。通常，处理器提供了特殊的指令用于将数据压在堆栈中，并在以后需要是取出来。堆栈使用LIFO（后进先出）的方式。换句话说，如果你压入两个值x和y到堆栈中，然后从堆栈中弹出一个值，那么你会得到y的值。

一些处理器的堆栈向内存顶部增长，而另一些向内存的底部增长。还有一些处理器两种方式都可以支持，例如：ARM。

Processor Status（PS）

指令可能产生结果。例如：“X寄存器的内容是否大于Y寄存器的内容？“可能产生真或假的结果。PS寄存器保留这些结果以及处理器当前状态的其他信息。多数处理器至少有两种模式：kernel（核心态）和user（用户态），PS寄存器会纪录能够确定当前模式的那些信息。

## 2. Memory(内存)

所有系统都具有分级的内存结构，由位于不同级别的速度和容量不同的内存组成。

最快的内存是高速缓存存储器，就象它的名字暗示的一样-用于临时存放或缓存主内存的内容。这种内存非常快但是比较昂贵，因此多数处理器芯片上内置有少量的高速缓冲存储器，而大多数高速缓存存储器放在系统主板上。一些处理器用一块缓存内存同时缓存指令和数据，而另一些处理器有两块缓存内存-一个用于指令，另一个用于数据。Alpha AXP处理器有两个内置的内存高速缓存存储器：一个用于数据（D-Cache），另一个用于指令（I-Cache）。它的外部高速缓冲存储器（或B-Cache）将两者混在一起。

最后一种内存是主内存。相对于外部高速缓存存储器而言速度非常慢，对于CPU内置的高速缓存存储器，主内存简直是在爬。

高速缓存存储器和主内存必须保持同步（一致）。换句话说，如果主内存中的一个字保存在高速缓存存储器的一个或多个位置，那么系统必须保证高速缓存存储器和主内存的内容一样。使高速缓冲存储器同步的工作一部分是由硬件完成，另一部分则是由操作系统完成的。对于其它一些系统的主要任务，硬件和软件也必须紧密配合。

### 3. Buses（总线）

系统板的各个组成部分由被称为总线的连接系统互连在一起。系统总线分为三种逻辑功能：地址总线、数据总线和控制总线。地址总线指定了数据传输的内存位置（地址），数据总线保存了传输的数据。数据总线是双向的，它允许CPU读取，也允许CPU写。控制总线包含了各种信号线用于在系统中发送时钟和控制信号。有许多种不同的总线类型，ISA和PCI总线是系统用于连接外设的常用方式。

### 4. Controllers and Peripherals （控制器和外设）

外设指实在的设备，如由系统板或系统板插卡上的控制芯片所控制的图形卡或磁盘。IDE控制芯片控制IDE磁盘，而SCSI控制芯片控制SCSI磁盘。这些控制器通过不同的总线连接到CPU并相互连接。现在制造的大多数系统都是用PCI或ISA总线将系统的主要部件连接在一起。控制器本身也是象CPU一样的处理器，它们可以看作CPU的智能助手，CPU拥有系统的最高控制权。

所有的控制器都是不同的，但是通常它们都有用于控制它们的寄存器。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错误的状态码，另一个寄存器可能用于控制用途，改变控制器的模式。一个总线上的每一个控制器都可以分别被CPU寻址，这样软件设备驱动程序就可以读写它的寄存器进而控制它。IDE电缆是一个好例子，它给了你分别存取总线上每一个驱动器的能力。另一个好例子是PCI总线，允许每一个设备（如图形卡）被独立存取。

### 5. Address Spaces（寻址空间）

连接CPU和主内存的系统总线以及连接CPU和系统硬件外设的总线是分离的。硬件外设所拥有的内存空间称为I/O空间。I/O空间本身可以再进一步划分，但是我们现在先不讨论。CPU可以访问系统内存空间和I/O空间，而控制器只能通过CPU间接访问系统内存。从设备的角度来看，比如软驱控制器，它只能看到它的控制寄存器所在的地址空间（ISA），而非系统内存。一个CPU用不同的指令去访问内存和I/O空间。例如，可能有一条指令是“从I/O地址0x3f0读取一个字节到X寄存器“。这也是CPU通过读写系统硬件外设处于I/O地址空间的寄存器从而控制外设的方法。在地址空间中，普通外设（如IDE控制器，串行端口，软驱控制器等等）的寄存器在PC外设的多年发展中已经成了定例。I/O空间的地址0x3f0正是串行口（COM1）的控制寄存器的地址。

有时控制器需要直接从系统内存读取大量内存，或直接写大量数据到系统内存中去。比如将用户数据写到硬盘上去。在这种情况下，使用直接内存存取（DMA）控制器，允许硬件设备直接存取系统内存，当然，这种存取必须在CPU的严格控制和监管下进行。

### 6. Timer(时钟)

所有操作系统需要知道时间，现代PC包括一个特殊的外设，叫做实时时钟（RTC）。它提供了两样东西：可靠的日期和精确的时间间隔。RTC有自己的电池，所以即使PC没有加电，它仍在运行。这也是为什么PC总是“知道”正确的日期和时间。时间间隔计时允许操作系统精确地调度基本工作。

## Chapter 2

## Software Basic(软件基础)

程序是用于执行特定任务的计算机指令组合。程序可以用汇编语言，一种非常低级的计算机语言来编写，也可以使用和机器无关的高级语言，比如C语言编写。操作系统是一个特殊的程序，允许用户通过它运行应用程序，比如电子表和文字处理等等。本章介绍了基本的编程原理，并简介操作系统的目的和功能。

### 2.1 Computer Languages(计算机语言)

#### 2.1.1.汇编语言

CPU从内存中读取和执行的指令对于人类来讲无法理解。它们是机器代码，精确的告诉计算机要做什么。比如十六进制数0x89E5，是Intel 80486的指令，将寄存器ESP的内容拷贝到寄存器EBP中。早期计算机中最初的软件工具之一是汇编程序，它读入人类可以阅读的源文件，将其装配成机器代码。汇编语言明确地处理对寄存器和对数据的操作，而这种操作对于特定的微处理器而言是特殊的。Intel X86微处理器的汇编语言和Alpha AXP微处理器的汇编语言完全不同。以下Alpha AXP汇编代码演示了程序可以执行的操作类型：

```
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
```

第一条语句（第一行）将寄存器15指定的地址中的内容加载到寄存器16中。第二条指令将紧接着的内存中的内容加载到寄存器17中。第三行比较寄存器16和寄存器17，如果相等，分支到标号100，否则，继续执行第四行，将寄存器17的内容存到内存中。如果内存中的数据相同，就不必存储数据。编写汇编级的程序需要技巧而且十分冗长，容易出错。Linux系统的核心很少的一部分是用汇编语言编写，而这些部分之所以使用汇编语言只是为了提高效率，并且和具体的微处理器相关。

### 2.1.2 The C Programming Language and Compiler (C语言和编译器)

使用汇编语言编写大型程序十分困难，消耗时间，容易出错而且生成的程序不能移植，只能束缚在特定的处理器家族。更好的选择是使用和机器无关的语言，例如C。C允许你用逻辑算法描述程序和要处理的数据。被称为编译程序（compiler）的特殊程序读入C程序，并将它转换为汇编语言，进而产生机器相关的代码。好的编译器生成的汇编指令可以和好的汇编程序员编写的程序效率接近。大部分Linux核心是用C语言编写的。以下的C片断：

```
if (x != y)

x = y;
```

执行了和前面示例中汇编代码完全一样的操作。如果变量x的内容和变量y的内容不一样，变量y的内容被拷贝到变量x。C代码用例程（routine）进行组合，每一个例程执行一项任务。例程可以返回C所支持的任意的数值或数据类型。大型程序比如Linux核心分别由许多的C语言模块组成，每一个模块有自己的例程和数据结构。这些C源代码模块共同构成了逻辑功能比如文件系统的处理代码。

C支持多种类型的变量。一个变量是内存中的特定位置，可用符号名引用。上述的C片断中，x和y引用了内存中的位置。程序员不需要关心变量在内存中的具体位置，这是连接程序（下述）必须处理的。一些变量包含不同的数据例如整数、浮点数等和另一些则包含指针。

指针是包含其它数据在内存中的地址的变量。假设一个变量x，位于内存地址0x80010000， 你可能有一个指针px，指向x。 Px可能位于地址0x80010030。Px的值则是变量x的地址，0x80010000。

C允许你将相关的变量集合成为结构。例如：

Struct {

Int I;

Char b;

} my_struct;

是一个叫做my_struct的数据结构，包括两个元素：一个整数（32位）I和一个字符（8位数据）b。

### 2.1.3 Linkers（连接程序）

连接程序将几个目标模块和库文件连接在一起成为一个单独的完整程序。目标模块是汇编程序或编译程序的机器码输出，它包括机器码、数据和供连接程序使用的连接信息。比如：一个目标模块可能包括程序的所有数据库功能，而另一个目标模块则包括处理命令行参数的函数。连接程序确定目标模块之间的引用关系，即确定一个模块所引用的例程和数据在另一个模块中的实际位置。Linux核心是由多个目标模块连接而成的独立的大程序。

## 2.2 What is an Operating System（什么是操作系统？）

没有软件，计算机只是一堆发热的电子元件。如果说硬件是计算机的心脏，则软件就是它的灵魂。操作系统是允许用户运行应用程序的一组系统程序。操作系统将系统的硬件抽象，呈现在用户和应用程序之前的是一个虚拟的机器。是软件造就了计算机系统的特点。大多数PC可以运行一到多个操作系统，而每一个操作系统从外观和感觉上都大不相同。Linux由不同功能的部分构成，这些部分总体组合构成了Linux操作系统。Linux最明显的部分就是Kernel自身，但是如果没有shell或libraries一样没有用处。

为了了解什么是操作系统，看一看在你输入最简单的命令时发生了什么：

$ls

Mail c images perl

Docs tcl

$

这里的$是登录的shell输出的提示符（此例是bash）：表示shell在等候你（用户）输入命令。输入ls引发键盘驱动程序识别输入的字符，键盘驱动程序将识别的字符传递给shell去处理。shell先查找同名的可执行映象，它找到了/bin/ls, 然后调用核心服务将ls执行程序加载到虚拟内存中并开始执行。ls执行程序通过执行核心的文件子系统的系统调用查找文件。文件系统可能使用缓存的文件系统信息或通过磁盘设备驱动程序从磁盘上读取文件信息,也可能是通过网络设备驱动程序同远程主机交换信息而读取本系统所访问的远程文件的详细信息（文件系统可以通过NFS网络文件系统远程安装）。不管文件信息是如何得到的，ls都将信息输出，通过显示驱动程序显示在屏幕上。

以上的过程看起来相当复杂，但是它说明了即使是最简单的命令也是操作系统各个功能模块之间共同协作的结果，只有这样才能提供给你（用户）一个完整的系统视图。

### 2.2.1 Memory management（内存管理）

如果拥有无限的资源，例如内存，那么操作系统所必须做的很多事情可能都是多余的。所有操作系统的一个基本技巧就是让少量的物理内存工作起来好像有相当多的内存。这种表面看起来的大内存叫做虚拟内存，就是当软件运行的时候让它相信它拥有很多内存。系统将内存分为容易处理的页，在系统运行时将这些页交换到硬盘上。而应用软件并不知道，因为操作系统还使用了另一项技术：多进程。

### 2.2.2 Processes (进程)

进程可以看作一个在执行的程序，每一个进程都是正在运行的特定的程序的独立实体。如果你观察一下你的Linux系统，你会发现有很多进程在运行。例如：在我的系统上输入ps 显示了以下进程：

$ 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

$

如果我的系统拥有多个CPU那么每个进程可能（至少在理论上如此）都在不同的CPU上运行。不幸的是，只有一个，所以操作系统又使用技巧，在短时间内依次运行每一个进程。这个时间段叫做时间片。这种技巧叫做多进程或调度，它欺骗了每一个进程，好像它们是唯一的进程。进程相互之间受到保护，所以如果一个进程崩溃或不能工作，不会影响其他进程。操作系统通过给每一个进程一个独立的地址空间来实现保护，进程只能访问它自己的地址空间。

### 2.2.3 Device Drivers（设备驱动程序）

设备驱动程序组成了Linux核心的主要部分。象操作系统的其他部分一样，它们在一个高优先级的环境下工作，如果发生错误，可能会引发严重问题。设备驱动程序控制了操作系统和它控制的硬件设备之间的交互。比如：文件系统向IDE磁盘写数据块是使用通用块设备接口。驱动程序控制细节，并处理和设备相关的部分。设备驱动程序和它驱动的具体的控制器芯片相关，所以，如果你的系统有一个NCR810的SCSI控制器，那么你需要NCR810的驱动程序。

### 2.2.4 The Filesystems（文件系统）

象Unix一样，在Linux里，系统对独立的文件系统不是用设备标示符来存取（比如驱动器编号或驱动器名称），而是连接成为一个树型结构。Linux在安装新的文件系统时，把它安装到指定的安装目录，比如/mnt/cdrom，从而合并到这个单一的文件系统树上。Linux的一个重要特征是它支持多种不同的文件系统。这使它非常灵活而且可以和其他操作系统良好共存。Linux最常用的文件系统是EXT2，大多数Linux发布版都支持。

文件系统将存放在系统硬盘上的文件和目录用可以理解的统一的形式提供给用户，让用户不必考虑文件系统的类型或底层物理设备的特性。Linux透明的支持多种文件系统（如MS-DOS和EXT2），将所有安装的文件和文件系统集合成为一个虚拟的文件系统。所以，用户和进程通常不需要确切知道所使用的文件所在的文件系统的类型，用就是了。

块设备驱动程序掩盖了物理块设备类型的区别（如IDE和SCSI）。对于文件系统来讲，物理设备就是线性的数据块的集合。不同设备的块大小可能不同，如软驱一般是512字节，而IDE设备通常是1024字节，同样，对于系统的用户，这些区别又被掩盖。EXT2文件系统不管它用什么设备，看起来都是一样的。

## 2.3 Kernet Data Structures（核心数据结构）

操作系统必须纪录关于系统当前状态的许多信息。如果系统中发生了事情，这些数据结构就必须相应改变以反映当前的实际情况。例如：用户登录到系统中的时候，需要创建一个新的进程。核心必须相应地创建表示此新进程的数据结构，并和表示系统中其他进程的数据结构联系在一起。

这样的数据结构多数在物理内存中，而且只能由核心和它的子系统访问。数据结构包括数据和指针（其他数据结构或例程的地址）。乍一看，Linux核心所用的数据结构可能非常混乱。其实，每一个数据结构都有其目的，虽然有些数据结构在多个的子系统中都会用到，但是实际上它们比第一次看到时的感觉要简单的多。

理解Linux核心的关键在于理解它的数据结构和核心处理这些数据结构所用到的大量的函数。本书以数据结构为基础描述Linux核心。论及每一个核心子系统的算法，处理的方式和它们对核心数据结构的使用。

### 2.3.1 Linked Lists（连接表）

Linux使用一种软件工程技术将它的数据结构连接在一起。多数情况下它使用链表数据结构。如果每一个数据结构描述一个物体或者发生的事件的单一的实例，比如一个进程或一个网络设备，核心必须能够找出所有的实例。在链表中，根指针包括第一个数据结构或单元的地址，列表中的每一个数据结构包含指向列表下一个元素的指针。最后元素的下一个指针可能使0或NULL，表示这

是列表的结尾。在双向链表结构中，每一个元素不仅包括列表中下一个元素的指针，还包括列表中前一个元素的指针。使用双向链表可以比较容易的在列表中间增加或删除元素，但是这需要更多的内存存取。这是典型的操作系统的两难情况：内存存取数还是CPU的周期数。

### 2.3.2 Hash Tables

链接表是常用的数据结构，但是游历链接表的效率可能并不高。如果你要寻找指定的元素， 可能必须查找完整个表才能找到。Linux使用另一种技术：Hashing 来解决这种局限。Hash table是指针的数组或者说向量表。数组或向量表是在内存中依次存放的对象。书架可以说是书的数组。数组用索引来访问，索引是数组中的偏移量。再来看书架的例子，你可以使用在书架上的位置来描述每一本书：比如第5本书。

Hash table是一个指向数据结构的指针的数组，它的索引来源于数据结构中的信息。如果你用一个数据结构来描述一个村庄的人口，你可以用年龄作为索引。要找出一个指定的人的数据，你可以用他的年龄作为索引在人口散列表中查找，通过指针找到包括详细信息的数据结构。不幸的是，一个村庄中可能很多人年龄相同，所以散列表的指针指向另一个链表数据结构，每一个元素描述同龄人。即使这样，查找这些较小的链表仍然比查找所有的数据结构要快。

Hash table可用于加速常用的数据结构的访问，在Linux里常用hash table来实现缓冲。缓冲是需要快速存取的信息，是全部可用信息的一个子集。数据结构被放在缓冲区并保留在那里，因为核心经常访问这些结构。使用缓冲区也有副作用，因为使用起来比简单链表或者散列表更加复杂。如果数据结构可以在缓冲区找到（这叫做缓冲命中），那么一切很完美。但是如果数据结构不在缓冲区中，那么必须查找所用的相关的数据结构，如果找到，那么就加到缓冲区中。增加新的数据结构到缓冲区中可能需要废弃一个旧的缓冲入口。Linux必须决定废弃那一个数据结构，风险在于废弃的可能使Linux下一个要访问的数据结构。

### 2.3.3 Abstract Interfaces（抽象接口）

Linux核心经常将它的接口抽象化。接口是以特定方式工作的一系列例程和数据结构。比如：所有的网络设备驱动程序都必须提供特定的例程来处理特定的数据结构。用抽象接口的方式可以用通用的代码层来使用底层特殊代码提供的服务（接口）。例如网络层是通用的，而它由底层符合标准接口的同设备相关的代码提供支持。

通常这些底层在启动时向高一层登记。这个登记过程常通过在链接表中增加一个数据结构来实现。例如，每一个连结到核心的文件系统在核心启动时进行登记（或者如果你使用模块，在文件系统第一次使用时向核心登记）。你可以查看文件/proc/filesystems来检查那些文件系统进行了登记。登记所用的数据结构通常包括指向函数的指针。这是执行特定任务的软件函数的地址。再一次用文件系统登记的例子，每一个文件系统登记时传递给Linux核心的数据结构都包括一个和具体文件系统相关的例程地址，在安装文件系统时必须调用。

# Chapter 3

# Memory Management （内存管理）

内存管理子系统是操作系统的重要部分。从计算机发展早期开始，就存在对于大于系统中物理能力的内存需要。为了克服这种限制，开发了许多种策略，其中最成功的就是虚拟内存。虚拟内存通过在竞争进程之间共享内存的方式使系统显得拥有比实际更多的内存。

虚拟内存不仅仅让你的计算机内存显得更多，内存管理子系统还提供：

Large Address Spaces（巨大的地址空间）操作系统使系统显得拥有比实际更大量的内存。虚拟内存可以比系统中的物理内存大许多倍。

Protection（保护）系统中的每一个进程都有自己的虚拟地址空间。这些虚拟的地址空间是相互完全分离的，所以运行一个应用程序的进程不会影响另外的进程。另外，硬件的虚拟内存机制允许对内存区写保护。这可以防止代码和数据被恶意的程序覆盖。

Memory Mapping（内存映射）内存映射用来将映像和数据映射到进程的地址空间。用内存映射，文件的内容被直接连结到进程的虚拟地址空间。

Fair Physics Memory Allocation（公平分配物理内存）内存管理子系统允许系统中每一个运行中的进程公平地共享系统的物理内存

Shared Virtual Memory（共享虚拟内存）虽然虚拟内存允许进程拥有分离（虚拟）的地址空间，有时你也需要进程之间共享内存。例如，系统中可能有多个进程运行命令解释程序bash。虽然可以在每一个进程的虚拟地址空间都拥有一份bash的拷贝，更好的是在物理内存中只拥有一份拷贝，所有运行bash的进程共享代码。动态连接库是多个进程共享执行代码的另一个常见例子。共享内存也可以用于进程间通讯(IPC)机制，两个或多个进程可以通过共同拥有的内存交换信息。Linux系统支持系统V的共享内存IPC机制。

## 3.1 An Abstract Model of Virtual Memory（虚拟内存的抽象模型）

在考虑Linux支持虚拟内存的方法之前，最好先考虑一个抽象的模型，以免被太多的细节搞乱。

在进程执行程序的时候，它从内存中读取指令并进行解码。解码指令也许需要读取或者存储内存特定位置的内容，然后进程执行指令并转移到程序中的下一条指令。进程不管是读取指令还是存取数据都要访问内存。

在一个虚拟内存系统中，所有的地址都是虚拟地址而非物理地址。处理器通过操作系统保存的一组信息将虚拟地址转换为物理地址。

为了让这种转换更简单，将虚拟内存和物理内存分为适当大小的块，叫做页（page）。页的大小一样。（当然可以不一样，但是这样一来系统管理起来比较困难）。Linux在Alpha AXP系统上使用8K字节的页，而在Intel x86系统上使用4K字节的页。每一页都赋予一个唯一编号：page frame number(PFN 页编号)。在这种分页模型下，虚拟地址由两部分组成：虚拟页号和页内偏移量。假如页大小是4K，则虚拟地址的位11到0包括页内偏移量，位12和以上的位是页编号。每一次处理器遇到虚拟地址，它必须提取出偏移和虚拟页编号。处理器必须将虚拟页编号转换到物理的页，并访问物理页的正确偏移处。为此，处理器使用了页表（page tables）。

图3.1显示了两个进程的虚拟地址空间，进程X和进程Y，每一个进程拥有自己的页表。这些页表将每一个进程的虚拟页映射到内存的物理页上。图中显示进程X的虚拟页号0映射到物理页号1，而进程Y的虚拟页编号1映射到物理页号4。理论上页表每一个条目包括以下信息：

有效标志 表示页表本条目是否有效

本页表条目描述的物理页编号

访问控制信息 描述本页如何使用：是否可以写？是否包括执行代码？

页表通过虚拟页标号作为偏移来访问。虚拟页编号5是表中的第6个元素（0是第一个元素）

要将虚拟地址转换到物理地址，处理器首先找出虚拟地址的页编号和页内偏移量。使用2的幂次的页尺寸，可以用掩码或移位简单地处理。再一次看图3.1，假设页大小是0x2000（十进制8192），进程Y的虚拟地址空间的地址是0x2194，处理器将会把地址转换为虚拟页编号1内的偏移量0x194。

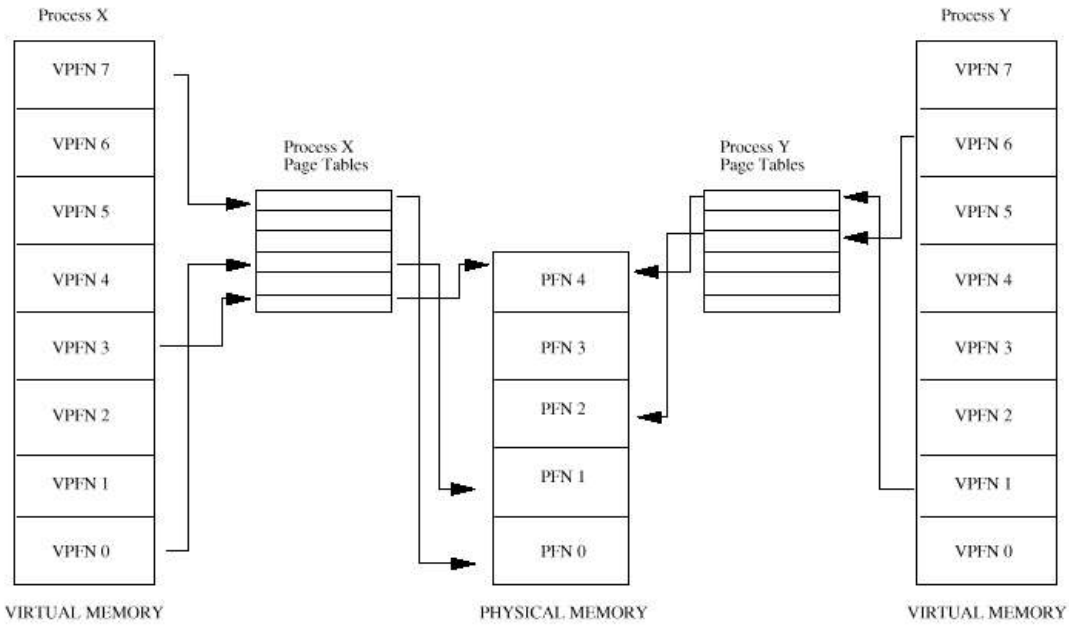



Figure 3.1: Abstract model of Virtual to Physical address mapping

处理器使用虚拟页编号作为索引在进程的页表中找到它的页表的条目。如果该条目有效，处理器从该条目取出物理的页编号。如果本条目无效，就是进程访问了它的虚拟内存中不存在的区域。在这种情况下，处理器无法解释地址，必须将控制权传递给操作系统来处理。

处理器具体如何通知操作系统进程在访问无法转换的无效的虚拟地址，这个方式是和处理器相关的。处理器将这种信息（page fault）进行传递，操作系统得到通知，虚拟地址出错，以及出错的原因。

假设这是一个有效的页表条目，处理器取出物理页号并乘以页大小，得到了物理内存中本页的基础地址。最后，处理器加上它需要的指令或数据的偏移量。

再用上述例子，进程Y的虚拟页编号1映射到了物理页编号4（起始于0x8000 ， 4x 0x2000），加上偏移0x194，得到了最终的物理地址0x8194。

通过这种方式将虚拟地址映射到物理地址，虚拟内存可以用任意顺序映射到系统的物理内存中。例如，图3.1 中，虚拟内存X的虚拟页编号映射到了物理页编号1而虚拟页编号7虽然在虚拟内存中比虚拟页0要高，却映射到了物理页编号0。这也演示了虚拟内存的一个有趣的副产品：虚拟内存页不必按指定顺序映射到物理内存中。

### 3.1.1 Demand Paging

因为物理内存比虚拟内存少得多，操作系统必须避免无效率地使用物理内存。节省物理内存的一种方法是只加载执行程序正在使用的虚拟页。例如：一个数据库程序可能正在数据库上运行一个查询。在这种情况下，并非所有的数据必须放到内存中，而只需要正被检查的数据记录。如果这是个查找型的查询，那么加载程序中增加记录的代码就没什么意义。这种进行访问时才加载虚拟页的技术叫做demand paging。

当一个进程试图访问当前不在内存中的虚拟地址的时候处理器无法找到引用的虚拟页对应的页表条目。例如：图3.1中进程X的页表中没有虚拟页2 的条目，所以如果进程X试图从虚拟页2中的地址读取时，处理器无法将地址转换为物理地址。这时处理器通知操作系统发生page fault。

如果出错的虚拟地址无效意味着进程试图访问它不应该访问的虚拟地址。也许是程序出错，例如向内存中任意地址写。这种情况下，操作系统会中断它，从而保护系统中其他的进程。

如果出错的虚拟地址有效但是它所在的页当前不在内存中，操作系统必须从磁盘映像中将相应的页加载到内存中。相对来讲磁盘存取需要较长时间，所以进程必须等待直到该页被取到内存中。如果当前有其他系统可以运行，操作系统将选择其中一个运行。取到的页被写到一个空闲的页面，并将一个有效的虚拟页条目加到进程的页表中。然后这个进程重新运行发生内存错误的地方的机器指令。这一次虚拟内存存取进行时，处理器能够将虚拟地址转换到物理地址，所以进程得以继续运行。

Linux使用demand paging技术将可执行映像加载到进程的虚拟内存中。当一个命令执行时，包含它的文件被打开，它的内容被映射到进程的虚拟内存中。这个过程是通过修改描述进程内存映射的数据结构来实现，也叫做内存映射（memory mapping）。但是，实际上只有映像的第一部分真正放在了物理内存中。映像的其余部分仍旧在磁盘上。当映像执行时，它产生page fault，Linux使用进程的内存映像表来确定映像的那一部分需要加载到内存中执行。

### 3.1.2 Swapping（交换）

如果进程需要将虚拟页放到物理内存中而此时已经没有空闲的物理页，操作系统必须废弃物理空间中的另一页，为该页让出空间。

如果物理内存中需要废弃的页来自磁盘上的映像或者数据文件，而且没有被写过所以不需要存储，则该页被废弃。如果进程又需要该页，它可以从映像或数据文件中再次加载到内存中。

但是，如果该页已经被改变，操作系统必须保留它的内容以便以后进行访问。这种也叫做dirty page，当它从物理内存中废弃时，被存到一种叫做交换文件的特殊文件中。因为访问交换文件的速度和访问处理器以及物理内存的速度相比很慢，操作系统必须判断是将数据页写到磁盘上还是将它们保留在内存中以便下次访问。

如果决定哪些页需要废弃或者交换的算法效率不高，则会发生颠簸（thrashing）。这时，页不断

地被写到磁盘上，又被读回，操作系统过于繁忙而无法执行实际的工作。例如在图3.1中，如果物理页号1经常被访问，那么就不要将它交换到硬盘上。进程正在使用的也叫做工作集(working set)。有效的交换方案应该保证所有进程的工作集都在物理内存中。

Linux使用LRU（Least Recently Used最近最少使用）的页面技术来公平地选择需要从系统中废弃的页面。这种方案将系统中的每一页都赋予一个年龄，这个年龄在页面存取时改变。页面访问越多，年纪越轻，越少访问，年纪越老越陈旧。陈旧的页面是交换的好候选。

### 3.1.3 Shared Vitual Memory（共享虚拟内存）

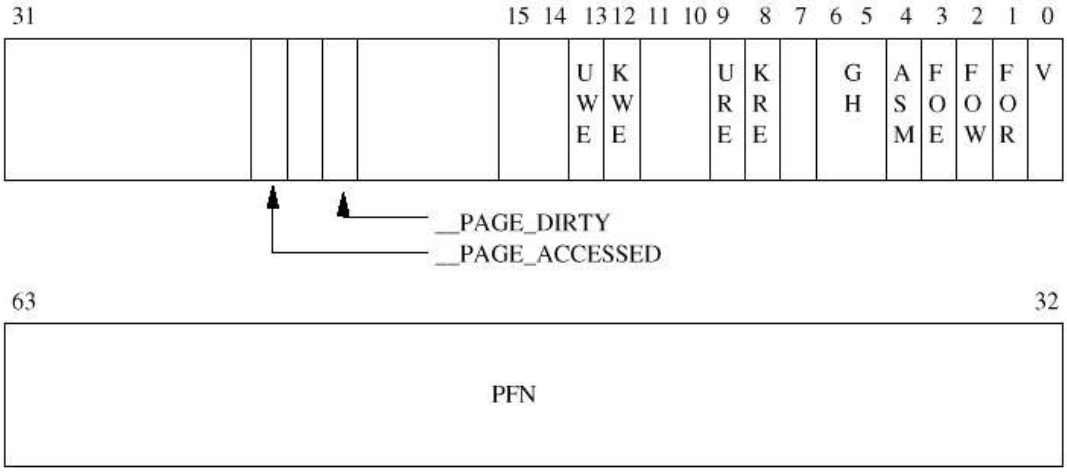
虚拟内存使多个进程可以方便地共享内存。所有的内存访问都是通过页表，每一个进程都有自己的页表。对于两个共享一个物理内存页的进程，这个物理页编号必须出现在两个进程的页表中。

图3.1显示了两个共享物理页号4的进程。对于进程X虚拟页号是4，而对于进程Y虚拟页号是6。这也表明了共享页的一个有趣的地方：共享的物理页不必存在共享它的进程的虚拟内存空间的同一个地方。

### 3.1.4 Physical and Vitual Addressing Modes（物理和虚拟寻址模式）

对于操作系统本身而言，运行在虚拟内存中没有什么意义。如果操作系统必须维护自身的页表，这将会是一场噩梦。大多数多用途的处理器同时支持物理地址模式和虚拟地址模式。物理寻址模式不需要页表，处理器在这种模式下不需要进行任何地址转换。Linux核心运行在物理地址模式。

Alpha AXP处理器没有特殊的物理寻址模式。它将内存空间分为几个区，将其中两个指定为物理映射地址区。核心的地址空间叫做KSEG地址空间，包括从0xfffffc0000000000向上的所有地址。为了执行连接在KSEG的代码（核心代码）或者访问那里的数据，代码必须在核心态执行。Alpha 上的Linux核心连接到从地址0xfffffc0000310000执行。




### 3.1.5 Access Control（访问控制）

页表条目也包括访问控制信息。当处理器使用页表条目将进程的虚拟地址映射到物理地址的时候，它很容易利用访问控制信息控制进程不要用不允许的方式进行访问。

有很多原因你希望限制对于内存区域的访问。一些内存，比如包含执行代码，本质上是只读的代码，操作系统应该禁止进程写它的执行代码。反过来，包括数据的页可以写，但是如果试图执行这段内存应该失败。大多数处理器有两种执行状态：核心态和用户态。你不希望用户直接执行核心态的代码或者存取核心数据结构，除非处理器运行在核心态。

访问控制信息放在PTE（page table entry）中，而且和具体处理器相关。图3.2显示了Alpha AXP的PTE。各个位意义如下：

V 有效，这个PTE是否有效

FOE “Fault on Execute” 试图执行本页代码时，处理器是否要报告page fault，并将控制权传递给操作系统。

ＦＯＷ “Fault on Write” 如上，在试图写本页时产生page fault

FOR “fault on read” 如上，在试图读本页时产生page fault

ASM 地址空间匹配。用于操作系统清除转换缓冲区中的部分条目

KRE 核心态的代码可以读本页

URE 用户态的代码可以读本页

GII 间隔因子，用于将一整块映射到一个转换缓冲条目而非多个。

KWE 核心态的代码可以写本页

UWE 用户态的代码可以写本页

Page frame number 对于V位有效的PTE，包括了本PTE的物理页编号；对于无效的PTE，如果不是0，包括了本页是否在交换文件的信息。

以下两位由Linux定义并使用

_PAGE_DIRTY 如果设置，本页需要写到交换文件中。

_PAGE_ACCESSED Linux 使用，标志一页已经访问过

3.2 Caches（高速缓存）

如果你用以上理论模型来实现一个系统，它可以工作，但是不会太高效率。操作系统和处理器的设计师都尽力让系统性能更高。除了使用更快的处理器、内存等，最好的方法是维护有用信息和数据的高速缓存，这会使一些操作更快。Linux使用了一系列和高速缓存相关的内存管理技术：

Buffer Cache： Buffer cache 包含了用于块设备驱动程序的数据缓冲区。这些缓冲区大小固定（例如512字节），包括从块设备读出的数据或者要写到块设备的数据。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。块设备用设备标识符和要访问的数据块编号作为索引，用来快速定位数据块。块设备只能通过buffer cache存取。如果数据可以在buffer cache中找到，那就不需要从物理块设备如硬盘上读取，从而使访问加快。

参见fs/buffer.c

Page Cache 用来加快对磁盘上映像和数据的访问。它用于缓存文件的逻辑内容，一次一页，并通过文件和文件内的偏移来访问。当数据页从磁盘读到内存中时，被缓存到page cache中。

参见mm/filemap.c

Swap Cache 只有改动过的（或脏dirty）页才存在交换文件中。只要它们写到交换文件之后没有再次修改，下一次这些页需要交换出来的时候，就不需要再写到交换文件中，因为该页已经在交换文件中了，直接废弃该页就可以了。在一个交换比较厉害的系统，这会节省许多不必要和高代价的磁盘操作。

参见mm/swap_state.c mm/swapfile.c

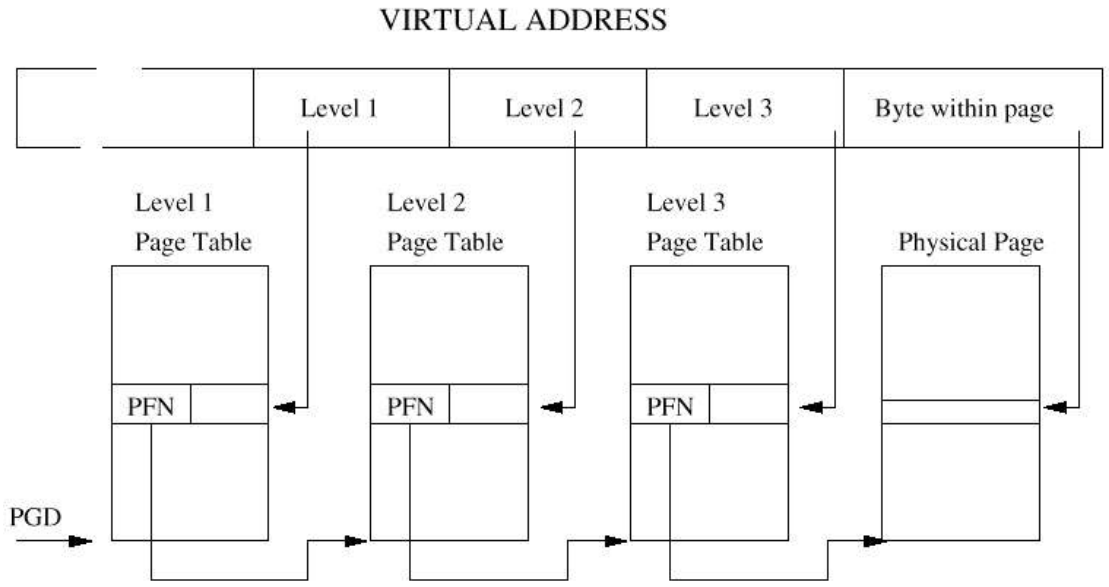



Figure 3.3: Three Level Page Tables

Hardware Cache:硬件高速缓存的常见的实现方法是在处理器里面：PTE的高速缓存。这种情况下，处理器不需要总是直接读页表，而在需要时把页转换表放在缓存区里。CPU里有转换表缓冲区(TLB Translation Look-aside Buffers)，放置了系统中一个或多个进程的页表条目的缓存的拷贝。

当引用虚拟地址时，处理区试图在TLB中寻找。如果找到了，它就直接将虚拟地址转换到物理地址，进而对数据执行正确的操作。如果找不到，它就需要操作系统的帮助。它用信号通知操作系统，发生了TLB missing。一个和系统相关的机制将这个异常转到操作系统相应的代码来处理。操作系统为这个地址映射生成新的TLB条目。当异常清除之后，处理器再次尝试转换虚拟地址，这一次将会成功因为TLB中该地址有了一个有效的条目。

高速缓存的副作用（不管是硬件或其他方式的）在于Linux必须花大量时间和空间来维护这些高速缓存区，如果这些高速缓存区崩溃，系统也会崩溃。

## 3.3 Linux Page Tables（Linux页表）

Linux假定了三级页表。访问的每一个页表包括了下一级页表的页编号。图3.3显示了一个虚拟地址如何分为一系列字段：每一个字段提供了在一个页表中的偏移量。为了将虚拟地址转换为物理地址，处理器必须取得每一级字段的内容，转换为包括该页表的物理页内的偏移，然后读取下一级页表的页编号。重复三次直到包括虚拟地址的物理地址的页编号找到为止。然后用虚拟地址中的最后一个字段：字节偏移量，在页内查找数据。

Linux运行的每一个平台都必须提供转换宏，让核心处理特定进程的页表。这样，核心不需要知道

页表条目的具体结构或者如何组织。通过这种方式，Linux成功地使用了相同的页表处理程序用于Alpha和Intel x86处理器，其中Alpha使用三级页表，而Intel使用二级页表。

参见include/asm/pgtable.h

## 3.4 Page Allocation and Deallocation (页的分配和回收)

系统中对于物理页有大量的需求。例如，当程序映像加载到内存中的时候，操作系统需要分配页。当程序结束执行并卸载时需要释放这些页。另外为了存放核心相关的数据结构比如页表自身，也需要物理页。这种用于分配和回收页的机制和数据结构对于维护虚拟内存子系统的效率也许是最重要的。

系统中的所有的物理页都使用mem_map数据结构来描述。这是一个mem_map_t结构的链表，在启动时进行初始化。每一个mem_map_t（容易混淆的是这个结构也被称为page 结构）结构描述系统中的一个物理页。重要的字段（至少对于内存管理而言）是：

参见include/linux/mm.h

count 本页用户数目。如果本页由多个进程共享，计数器大于1。

Age 描述本页的年龄。用于决定本页是否可以废弃或交换出去。

Map_nr mem_map_t描述的物理页编号。

页分配代码使用free_area向量来查找空闲的页。整个缓冲管理方案用这种机制来支持。只要用了这种代码，处理器使用的页的大小和物理页的机制就可以无关。

每一个free_area单元包括页块的信息。数组中的第一个单元描述了单页，下一个是2页大小的块，下一个是4页大小的块，以此类推，依次向上都是2的倍数。这个链表单元用作队列的开头，有指向mem_map数组中页的数据结构的指针。空闲的页块在这里排队。Map是一个跟踪这么大小的页的分配组的位图。如果页块中的第N块空闲，则位图中的第N位置位。

图3.4显示了free_area结构。单元0有一个空闲页（页编号0），单元2有2个4页的空闲块，第一个起始于页编号4，第二个起始于页编号56。

### 3.4.1 Page Allocation (页分配)

参见mm/page_alloc.c get_free_pages()

Linux使用Buddy算法有效地分配和回收页块。页分配代码试图分配一个由一个或多个物理页组成的块。页分配使用2的幂数大小的块。这意味着可以分配1页大小，2页大小，4页大小的块，依此类推。只要系统有满足需要的足够的空闲页（nr_free_pages > min_free_pages），分配代码就会在free_area中查找满足需要大小的一个页块。Free_area中的每一个单元都有描述自身大小的页块的占用和空闲情况的位图。例如，数组中的第2个单元拥有描述4页大小的块的空闲和占用的分配图。

这个算法首先找它请求大小的内存页块。它跟踪free_area数据结构中的list单元队列中的空闲页的链表。如果请求大小的页块没有空闲，就找下一个尺寸的块（2倍于请求的大小）。继续这一过程一直到遍历了所有的free_area或者找到了空闲页块。如果找到的页块大于请求的页块，则该块将被分开成为合适大小的块。因为所有的块都是2的幂次的页数组成，所以这个分割的过程比较简单，你只需要将它平分就可以了。空闲的块则放到适当的队列，而分配的页块则返回给调用者。

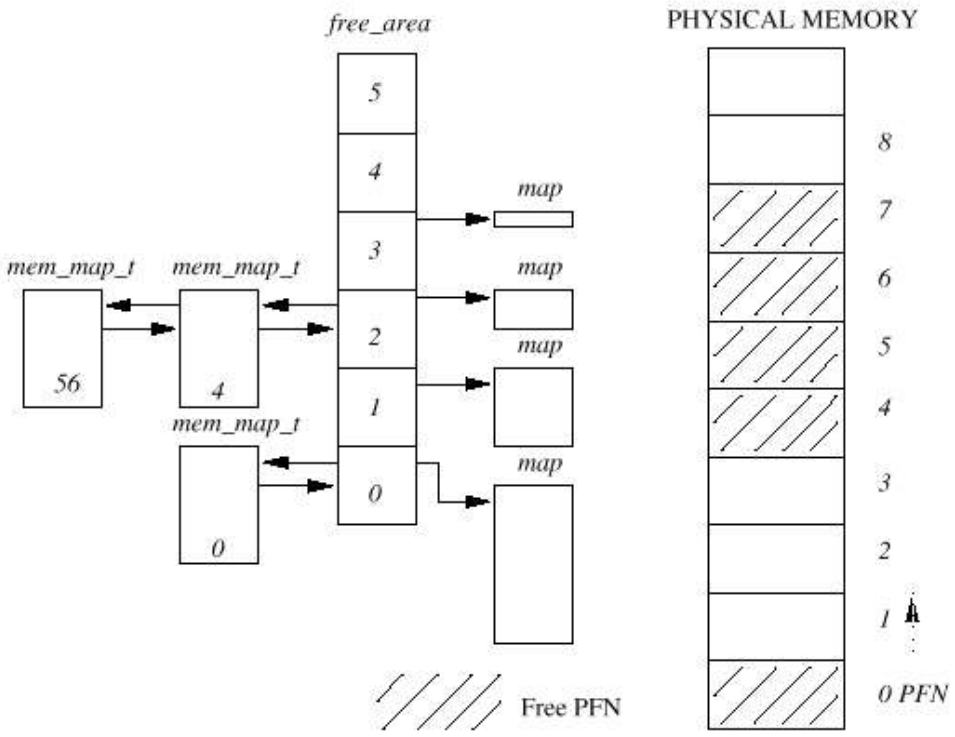



Figure 3.4: The free_area data structure

例如在图3.4中，如果请求2页的数据块，第一个4页块（起始于页编号4）将会被分为两个2页块。起始于页号4的第一个2页块将会被返回给调用者，而第二个2页块（起始于页号6）将会排在free_area数组中的单元1中2页空闲块的队列中。

### 3.4.2 Page Deallocation（页回收）

分配页块的过程中将大的页块分为小的页块，将会使内存更为零散。页回收的代码只要可能就把页联成大的页块。其实页块的大小很重要（2的幂数），因为这样才能很容易将页块组成大的页块。

只要一个页块回收，就检查它的相邻或一起的同样大小的页块是否空闲。如果是这样，就把它和新释放的页块一起组成以一个新的下一个大小的空闲页块。每一次两个内存页块组合成为更大的页块时，页回收代码都要试图将页块合并成为更大的块。这样，空闲的页块就会尽可能的大。

例如，在图3.4，如果页号1释放，那么它会和已经空闲的页号0一起组合并放在free_area的单元1中空闲的2页块队列中。

## 3.5 Memory Mapping （内存映射）

当一个映像执行时，执行映像的内容必须放在进程的虚拟地址空间中。对于执行映像连接到的任意共享库，情况也是一样。执行文件实际并没有放到物理内存，而只是被连接到进程的虚拟内存。这样，只要运行程序引用了映像的部分，这部分映像就从执行文件中加载到内存中。这种映像和进程虚拟地址空间的连接叫做内存映射。

每一个进程的虚拟内存用一个mm_struct 数据结构表示。这包括当前执行的映像的信息（例如bash）和指向一组vm_area_struct结构的指针。每一个vm_area_struct的数据结构都描述了内存区域的起始、进程对于内存区域的访问权限和对于这段内存的操作。这些操作是一组例程，Linux用于管理这段虚拟内存。例如其中一种虚拟内存操作就是当进程试图访问这段虚拟内存时发现（通过page fault）内存不在物理内存中所必须执行的正确操作，这个操作叫做 nopage 操作。Linux请求把执行映像的页加载到内存中的时候用到nopage操作。

当一个执行映像映射到进程的虚拟地址空间时，产生一组vm_area_struct数据结构。每一个vm_area_struct结构表示执行映像的一部分：执行代码、初始化数据（变量）、未初始化数据等等。Linux支持一系列标准的虚拟内存操作，当vm_area_struct数据结构创建时，一组正确的虚拟内存操作就和它们关联在一起。

## 3.6 Demand Paging

只要执行映像映射到进程的虚拟内存中，它就可以开始运行。因为只有映像的最开始的部

分是放在物理内存中，很快就会访问到还没有放在物理内存的虚拟空间区。当进程访问没有有效页表条目的虚拟地址的时候，处理器向Linux报告page fault。Page fault描述了发生page fault的虚拟地址和内存访问类型。

Linux必须找到page fault 发生的空间区所对应的vm_area_struct数据结构（用Adelson-Velskii and Landis AVL树型结构连接在一起）。如果找不到这个虚拟地址对应的vm_area_struct结构，说明进程访问了非法的虚拟地址。Linux将向该进程发信号，发送一个SIGSEGV信号，如果进程没有处理这个信号，它就会退出。

参见 handle_mm_fault() in mm/memory.c

Linux然后检查page faul的类型和该虚拟内存区所允许的访问类型。如果进程用非法的方式访问内存，比如写一个它只可以读的区域，也会发出内存错的信号。

现在Linux确定page fault是合法的，它必须进行处理。Linux必须区分在交换文件和磁盘映像中的页，它用发生page fault的虚拟地址的页表条目来确定。

参见do_no_page() in mm/memory.c

如果该页的页表条目是无效的但非空，此页是在交换文件中。对于Alpha AXP页表条目来讲，有效位置位但是PFN域非空。这种情况下PFN域存放了此页在交换文件（以及那一个交换文件）中的位置。页在交换文件中如何处理在本章后面讨论。

并非所有的vm_area_struct数据结构都有一整套虚拟内存操作，而且那些有特殊的内存操作的也可能没有nopang操作。因为缺省情况下，对于nopage操作，Linux会分配一个新的物理页并创建有效的页表条目。如果这一段虚拟内存有特殊的nopage操作，Linux会调用这个特殊的代码。

通常的Linux nopage操作用于对执行映像的内存映射，并使用page cache将请求的映像页加载到物理内存中。虽然在请求的页调入的物理内存中以后，进程的页表得到更新，但是也许需要必要的硬件动作来更新这些条目，特别是如果处理器使用了TLB。既然page fault得到了处理，就可以扔在一边，进程在引起虚拟内存访问错误的指令那里重新运行。

参见mm/filemap.c 中filemap_nopage()

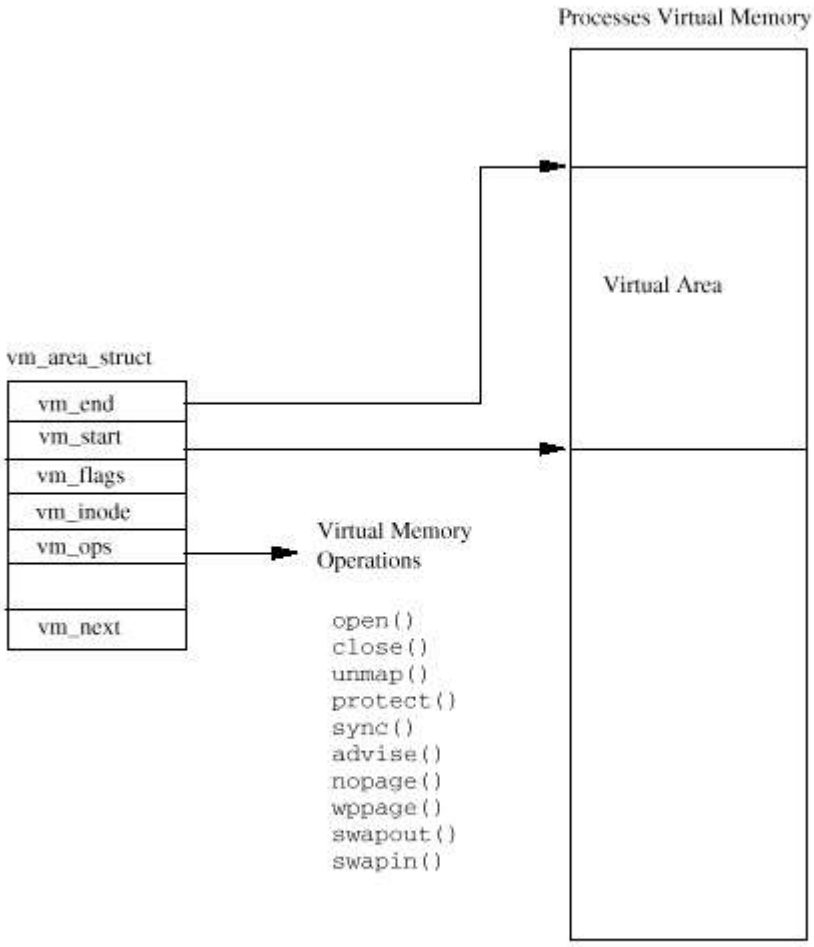



Figure 3.5: Areas of Virtual Memory

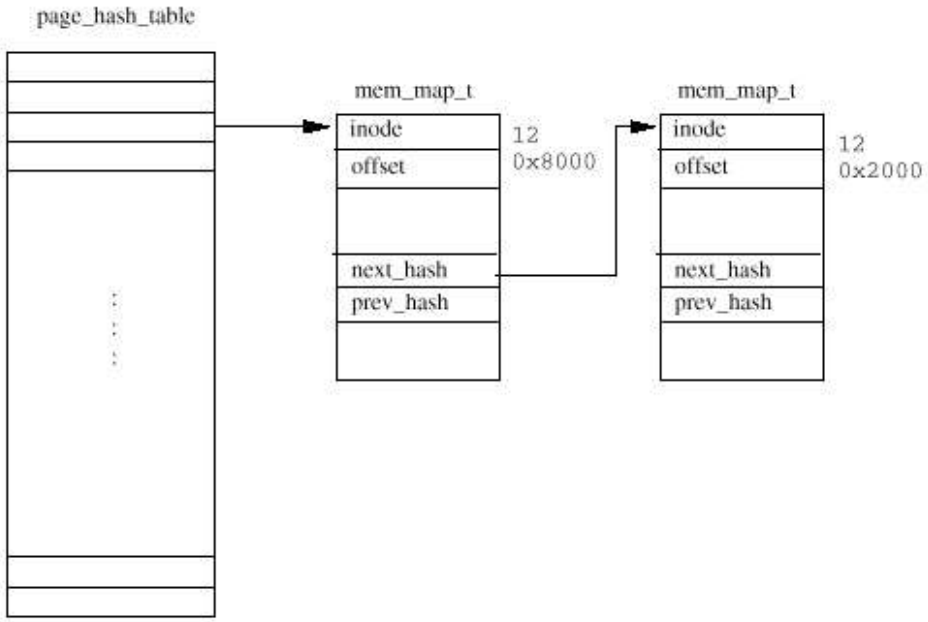



Figure 3.6: The Linux Page Cache

## 3.7 The Linux Page Cache

Linux的page cache的作用是加速对于磁盘文件的访问。内存映射文件每一次读入一页，这些页被存放在page cache中。图3.6显示了page cache，包括一个指向mem_map_t数据结构的指针向量：page_hash_table。Linux中的每一个文件都用一个VFS inode的数据结构标示（在第9章描述），每一个VFS I节点都是唯一的并可以完全确定唯一的一个文件。页表的索引取自VFS 的I节点号和文件中的偏移。

参见linux/pagemap.h

当一页的数据从内存映射文件中读出，例如当demand paging时需要放到内存中的时候，此页通过page cache中读出。如果此页在缓存中，就返回一个指向mem_map_t数据结构的指针给page fault 的处理代码。否则，此页必须从存放此文件的文件系统中加载到内存中。Linux分配物理内存并从磁盘文件中读出该页。如果可能，Linux会启动对文件下一页的读。这种单页的超前读意味着如果进程从文件中顺序读数据的话，下一页数据将会在内存中等待。

当程序映像读取和执行的时候page cache 不断增长。如果页不在需要，将从缓存中删除。比如不再被任何进程使用的映像。当Linux使用内存的时候，物理页可能不断减少，这时Linux可以减小page cache。

## 3.8 Swapping out and Discarding Pages（交换出去和废弃页）

当物理内存缺乏的时候，Linux内存管理子系统必须试图释放物理页。这个任务落在核心交换进程上（kswapd）。核心交换守护进程是一种特殊类型的进程，一个核心线程。核心线程是没有虚拟内存的进程，以核心态运行在物理地址空间。核心交换守护进程名字有一点不恰当，因为它不仅仅是将页交换到系统交换文件上。它的任务是保证系统有足够的空闲页，使内存管理系统有效地运行。

核心交换守护进程（kswapd）在启动时由核心的init 进程启动，并等待核心的交换计时器到期。每一次计时器到期，交换进程检查系统中的空闲页数是否太少。它使用两个变量：free_pages_high和free_pages_low来决定是否释放一些页。只要系统中的空闲页数保持在free_pages_high之上，交换进程什么都不做。它重新睡眠直到它的计时器下一次到期。为了做这种检查，交换进程要考虑正在向交换文件中写的页数，用nr_async_pages来计数：每一次一页排到队列中等待写到交换文件中的时候增加，写完的时候减少。Free_page_low 和free_page_high是系统启动时间设置的，和系统中的物理页数相关。如果系统中的空闲页数小于free_pages_high或者比free_page_low还低，核心交换进程会尝试三种方法来减少系统使用的物理页数：

参见mm/vmscan.c 中的kswapd()

减少buffer cache 和page cache的大小

将系统V的共享内存页交换出去

交换和废弃页

如果系统中的空闲页数低于free_pages_low，核心交换进程将试图在下一次运行前释放6页。否则试图释放3页。以上的每一种方法都要被尝试直到释放了足够的页。核心交换进程记录了它上一次使用的释放物理页的方法。每一次运行时它都会首先尝试上一次成功的方法来释放页。

释放了足够的页之后，交换进程又一次睡眠，直到它的计时器又一次过期。如果核心交换进程释放页的原因是系统空闲页的数量少于free_pages_low，它只睡眠平时的一半时间。只要空闲页数大于free_pages_low，交换进程就恢复原来的时间间隔进行检查。

### 3.8.1 Reducing the size of the Page and Buffer Caches

page 和buffer cache中的页是释放到free_area向量中的好选择。Page Cache，包含了内存映射文件的页，可能有不必要的数据，占去了系统的内存。同样，Buffer Cache ，包括了从物理设备读或向物理设备写的数据，也可能包含了无用的缓冲。当系统中的物理页将要耗尽的时候，废弃这些缓存区中的页相对比较容易，因为它不需要向物理设备写（不象将页从内存中交换出去）。废弃这些页不会产生多少有害的副作用，只不过使访问物理设备和内存映射文件时慢一点。虽然如此，如果公平地废弃这些缓存区中的页，所有的进程受到的影响就是平等的。

每一次当核心交换进程要缩小这些缓存区时，它要检查mem_map页矢量中的页块，看是否可以从物理内存中废弃。如果系统空闲页太低（比较危险时）而核心交换进程交换比较厉害，这个检查的页块大小就会更大一些。页块的大小进行循环检查：每一次试图减少内存映射时都用一个不同的页块大小。这叫做clock算法，就象钟的时针。整个mem_map页向量都被检查，每次一些页。

参见mm/filemap.c shrink_map()

检查的每一页都要判断缓存在page cache 或者buffer cache中。注意共享页的废弃这时不考虑，一页不会同时在两个缓存中。如果该页不在这两个缓冲区中，则mem_map页向量表的下一页被检查。

缓存在buffer cache  ch中的页（或者说页中的缓冲区被缓存）使缓冲区的分配和释放更有效。缩小内存映射的代码试图释放包含检查过的页的缓冲区。如果缓冲区释放了，则包含缓冲区的页也被释放了。如果检查的页是在Linux的page cache 中，它将从page cache 中删除并释放。

参见 fs/buffer.c free_buffer()

如果这次尝试释放了足够的页，核心交换进程就会继续等待直到下一次被周期性地唤醒。因为释放的页不属于任何进程的虚拟内存（只是缓存的页），因此不需要更新进程的页表。如果废弃的缓存页仍然不够，交换进程会试图交换出一些共享页。

### 3.8.2 Swapping Out System V Shared Memory Pages（交换出系统V的共享内存页）

系统V的共享内存是一种进程间通讯的机制，通过两个或多个进程共享虚拟内存交换信息。进程间如何共享内存在第5章详细讨论。现在只要讲讲每一块系统V共享内存都用一个shmid_ds的数据结构描述就足够了。它包括一个指向vm_area_struct链表数据结构的指针，用于共享此内存的每一个进程。Vm_area_struct数据结构描述了此共享内存在每一个进程中的位置。这个系统V的内存中的每一个vm_area_struct结构都用vm_next_shared和vm_prev_shared指针连接在一起。每一个shmid_ds数据结构都有一个页表条目的链表，每一个条目都描述一个共享的虚拟页和物理页的对应关系。

核心交换进程将系统V的共享内存页交换出去时也用clock算法。它每一次运行都记录了上一次交换出去了那一块共享内存的那一页。它用两个索引来记录：第一个是shmid_ds数据结构数组中的索引，第二个是这块共享内存区的页表链中的索引。这样可以共享内存区的牺牲比较公平。

参见ipc/shm.c shm_swap()

因为一个指定的系统V共享内存的虚拟页对应的物理页号包含在每一个共享这块虚拟内存的进程的页表中，所以核心交换进程必须修改所有的进程的页表来体现此页已经不在内存而在交换文件中。对于每一个交换出去的共享页，交换进程必须找到在每一个共享进程的页表中对应的此页的条目（通过查找每一个vm_area_struct指针）如果在一个进程页表中此共享内存页的条目有效，交换进程要把它变为无效，并且标记是交换页，同时将此共享页的在用数减1。交换出去的系统V共享页表的格式包括一个在shmid_ds数据结构组中的索引和在此共享内存区中页表条目的索引。

如果所有共享的内存都修改过，页的在用数变为0，这个共享页就可以写到交换文件中。这个系统V共享内存区的shmid_ds数据结构指向的页表中此页的条目将会换成交换出的页表条目。交换出的页表条目无效但是包含一个指向打开的交换文件的索引和此页在此文件内的偏移量。这个信息用于将此页再取回物理内存中。

## 3.3 Swapping Out and Discarding Pages

交换进程轮流检查系统中的每一个进程是否可以用于交换。好的候选是可以交换的进程（有一些不行）并且有可以从内存中交换出去或废弃的一个或多个页。只有其他方法都不行的时候才会把页从物理内存交换到系统交换文件中。

参见 mm/vmscan.c swap_out()

来自于映像文件的执行映像的大部分内容可以从文件中重新读出来。例如：一个映像的执行指令不会被自身改变，所以不需要写到交换文件中。这些页只是被简单地废弃。如果再次被进程引用，可以从执行映像再次加载到内存中。

一旦要交换的进程确定下来，交换进程就查看它的所有虚拟内存区域，寻找没有共享或锁定的区域。Linux不会把选定进程的所有可以交换出去的页都交换出去，而只是去掉少量的页。如果页在内存中锁定，则不能被交换或废弃。

参见mm/vmscan.c swap_out_vme() 跟踪进程mm_struct中排列的vm_area_struct结构中的vm_next vm_nex指针。

Linux的交换算法使用了页的年龄。每一个页都有一个计数器（放在mem_map_t数据结构中），告诉核心交换进程此页是否值得交换出去。页不用时变老，访问时更新。交换进程只交换老的页。缺省地，页第一次分配时年龄赋值为3。每一次访问，它的年龄就增加3，直到20。每一次系统交换进程运行时它将页的年龄减1使页变老。这个缺省的行为可以更改，所以这些信息（和其他相关信息）都存放在swap_control数据结构中。

如果页太老(年龄age = 0)，交换进程会进一步处理。脏页可以交换出去，Linux在描述此页的PTE中用一个和体系结构相关的位来描述这种页（见图3.2）。但是，并非所有的脏页都需要写到交换文件。每一个进程的虚拟内存区域都可以拥有自己的交换操作（由vm_area_struct中的vm_ops指针指示），如果这样，交换进程会用它的这种方式。否则，交换进程会从交换文件中分配一页，并把此页写到该文件中。

此页的页表条目会用一个无效的条目替换，但是包括了此页在交换文件的信息：此页所在文件内的偏移和所用的交换文件。不管什么方式交换，原来的物理页被放回到free_area重释放。干净（或不脏）的页可以被废弃，放回到free_area中重用。

如果交换或废弃了足够的可交换进程的页，交换进程重新睡眠。下一次唤醒时它会考虑系统中的下一个进程。这样，交换进程轻咬去每一个进程的物理页，直到系统重新达到平衡。这种做法比交换出整个进程更公平。

## 3.9 The Swap Cache（交换缓存）

当把页交换到交换文件时，Linux会避免写不必要写的页。有时可能一个页同时存在于交换文件和物理内存中。这发生于一页被交换出内存然后在进程要访问时又被调入内存的情况下。只要内存中的页没有被写过，交换文件中的拷贝就继续有效。

Linux用swap cache来记录这些页。交换缓存是一个页表条目或者系统物理页的链表。一个交换页有一个页表条目，描述使用的交换文件和它在交换文件中的位置。如果交换缓存条目非0，表示在交换文件中的一页没有被改动。如果此页后来被改动了（被写），它的条目就从交换缓存中删除）

当Linux需要交换一个物理页到交换文件的时候，它查看交换缓存，如果有此页的有效条目，它不需要把此页写到交换文件。因为内存中的此页从上次读到交换文件之后没有被修改过。

交换缓存中的条目是曾经交换出去的页表条目。它们被标记为无效，但是包含了允许Linux找到正确交换文件和交换文件中正确页的信息。

## 3.10 Swapping Page In（交换进）

保存在交换文件中的脏页可能又需要访问。例如：当应用程序要向虚拟内存中写数据，而此页对应的物理页交换到了交换文件时。访问不在物理内存的虚拟内存页会引发page fault。Page fault是处理器通知操作系统它不能将虚拟内存转换到物理内存的信号。因为交换出去后虚拟内存中描述此页的页表条目被标记为无效。处理器无法处理虚拟地址到物理地址的转换，将控制转回到操作系统，告诉它发生错误的虚拟地址和错误的原因。这个信息的格式和处理器如何把控制转回到操作系统是和处理器类型相关的。处理器相关的page faule处理代码必须定位描述包括出错虚拟地址的虚拟内存区的vm_area_struct的数据结构。它通过查找该进程的vm_area_struct数据结构，直到找到包含了出错的虚拟地址的那一个。这是对时间要求非常严格的代码，所以一个进程的vm_area_struct数据结构按照特定的方式排列，使这种查找花费时间尽量少。

参见 arch/i386/mm/fault.c do_page_fault()

执行了合适的和处理器相关的动作并找到了包括错误（发生）的虚拟地址的有效的虚拟内存，page fault的处理过程又成为通用的，并可用于Linux能运行的所有处理器。通用的page fault处理代码查找错误虚拟地址的页表条目。如果它找到的页表条目是交换出去的页，Linux必须把此页交换回物理内存。交换出去的页的页表条目的格式和处理器相关，但是所有的处理器都将这些页标为无效并在页表条目中放进了在交换文件中定位页的必要信息。Linux使用这种信息把此页调回到物理内存中。

参见mm/memory.c do_no_page()

这时，Linux知道了错误（发生）的虚拟地址和关于此页交换到哪里去的页表条目。Vm_area_struct数据结构可能包括一个例程的指针，用于把这块虚拟内存中的页交换回到物理内存中。这是swapin操作。如果这块内存中有swapin操作，Linux会使用它。其实，交换出去的系统V的共享内存之所以需要特殊的处理因为交换的系统V的共享内存页的格式和普通交换页的不同。如果没有swapin操作，Linux假定这是一个普通页，不需要特殊的处理。它分配一块空闲的物理页并将交换出去的页从交换文件中读进来。关于从交换文件哪里（和哪一个交换文件）的信

息取自无效的页表条目。

参见mm/page_alloc.c swap_in()

如果引起page fault的访问不是写访问，页就留在交换缓存中，它的页表条目标记为不可写。如果后来此页又被写，会产生另一个page fault，这时，此页被标志为脏页，而它的条目也从交换缓存中删除。如果此页没有被修改而又需要交换出来，Linux就可以避免将此页写到交换文件，因为此页已经在交换文件中了。

如果将此页从交换文件调回的访问是写访问，这个页就从交换缓存中删除，此页的页表条目页标记为脏页和可写。

## Chapter 4

## Processes （进程）

本章描述进程是什么以及Linux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。程序是存放在磁盘上的包括一系列机器代码指令和数据的可执行的映像，因此，是一个被动的实体。进程可以看作是一个执行中的计算机程序。它是动态的实体，在处理器执行机器代码指令时不断改变。处理程序的指令和数据，进程也包括程序计数器和其他CPU的寄存器以及包括临时数据（例如例程参数、返回地址和保存的变量）的堆栈。当前执行的程序，或者说进程，包括微处理器中所有的当前的活动。Linux是一个多进程的操作系统。进程是分离的任务，拥有各自的权利和责任。如果一个进程崩溃，它不应该让系统中的另一个进程崩溃。每一个独立的进程运行在自己的虚拟地址空间，除了通过安全的核心管理的机制之外无法影响其他的进程。

在一个进程的生命周期中它会使用许多系统资源。它会用系统的CPU执行它的指令，用系统的物理内存来存储它和它的数据。它会打开和使用文件系统中的文件，会直接或者间接使用系统的物理设备。Linux必须跟踪进程本身和它使用的系统资源以便管理公平地管理该进程和系统中的其他进程。如果一个进程独占了系统的大部分物理内存和CPU，对于其他进程就是不公平的。

系统中最宝贵的资源就是CPU。通常系统只有一个。Linux是一个多进程的操作系统。它的目标是让进程一直在系统的每一个CPU上运行，充分利用CPU。如果进程数多于CPU（多数是这样），其余的进程必须等到CPU被释放才能运行。多进程是一个简单的思想：一个进程一直运行，直到它必须等待，通常是等待一些系统资源，等拥有了资源，它才可以继续运行。在一个单进程的系统，比如DOS，CPU被简单地设为空闲，这样等待的时间就会被浪费。在一个多进程的系统中，

同一时刻许多进程在内存中。当一个进程必须等待时操作系统将CPU从这个进程拿走，并将它交给另一个更需要的进程。是调度程序选择了

下一次最合适的进程。Linux使用了一系列的调度方案来保证公平。

Linux支持许多不同的可执行文件格式，ELF是其中之一，Java是另一个。Linux必须透明地管理这些文件，因为进程使用系统的共享的库。

## 4.1 Linux Processes（Linux的进程）

Linux中，每一个进程用一个task_struct（在Linux中task和process互用）的数据结构来表示，用来管理系统中的进程。Task向量表是指向系统中每一个task_struct数据结构的指针的数组。这意味着系统中最大进程数受task向量表的限制，缺省是512。当新的进程创建的时候，从系统内存中分配一个新的task_struct，并增加到task向量表中。为了更容易查找，用current指针指向当前运行的进程。

参见include/linux/sched.h

除了普通进程，Linux也支持实时进程。这些进程必须对于外界事件迅速反应（因此叫做“实时”），调度程序必须和普通用户进程区分对待。虽然task_struct数据结构十分巨大、复杂，但是它的域可以分为以下的功能：

State 进程执行时它根据情况改变状态(state)。Linux进程使用以下状态：（这里漏掉了SWAPPING，因为看来没用到）

Running 进程在运行(是系统的当前进程)或者准备运行（等待被安排到系统的一个CPU上）

Waiting 进程在等待一个事件或资源。Linux区分两种类型的等待进程：可中断和不可中断的（interruptible and uninterruptible）。可中断的等待进程可以被信号中断，而不可中断的等待进程直接等待硬件条件，不能被任何情况中断。

Stopped 进程停止了，通常是接收到了一个信号。正在调试的进程可以在停止状态。

Zombie 终止的进程，因为某种原因，在task 向量表重任旧有一个task_struct数据结构的条目。就想听起来一样，是一个死亡的进程。

Scheduling Information 调度者需要这个信息用于公平地决定系统中的进程哪一个更应该运行。

Identifiers 系统中的每一个进程都有一个进程标识符。进程标识符不是task向量表中的索引，而只是一个数字。每一个进程也都有用户和组（user and group）的标识符。用来控制进程对于系统中文件和设备的访问。

Inter-Process Communication Linux支持传统的UNIX-IPC机制，即信号，管道和信号灯（semaphores），也支持系统V的IPC机制，即共享内存、信号灯和消息队列。关于Linux支持的IPC机制在第5章中描述。

Links 在Linux系统中，没有一个进程是和其他进程完全无关的。系统中的每一个进程，除了初始的进程之外，都有一个父进程。新进程不是创建的，而是拷贝，或者说从前一个进程克隆的（cloned）。每一个进程的task_struct中都有指向它的父进程和兄弟进程（拥有相同的父进程的进程）以及它的子进程的的指针。在Linux系统中你可以用pstree命令看到正在运行的进程的家庭关系。

init(1)-+-crond(98)

|-emacs(387)

|-gpm(146)

|-inetd(110)

|-kerneld(18)

|-kflushd(2)

|-klogd(87)

|-kswapd(3)

|-login(160)---bash(192)---emacs(225)

|-lpd(121)

|-mingetty(161)

|-mingetty(162)

|-mingetty(163)

|-mingetty(164)

|-login(403)---bash(404)---pstree(594)

|-sendmail(134)

|-syslogd(78)

`-update(166)

另外系统中的所有的进程信息还存放在一个task_struct数据结构的双向链表中，根是init进程。这个表让Linux可以查到系统中的所有的进程。它需要这个表以提供对于ps或者kill等命令的支持。

Times and Timers 在一个进程的生命周期中，核心除了跟踪它使用的CPU时间还记录它的其他时间。每一个时间片（clock tick），核心更新jiffies中当前进程在系统和用户态所花的时间综合。Linux也支持进程指定的时间间隔的计数器。进程可以使用系统调用建立计时器，在计时器到期的时候发送信号给自己。这种计时器可以是一次性的，也可是周期性的。

File system 进程可以根据需要打开或者关闭文件，进程的task_struct结构存放了每一个打开的文件描述符的指针和指向两个VFS I节点（inode）的指针。每一个VFS I节点唯一描述一个文件系统中的一个文件或目录，也提供了对于底层文件系统的通用接口。Linux下如何支持文件系统在第9章中描述。第一个I节点是该进程的根（它的主目录），第二个是它的当前或者说pwd目录。Pwd取自Unix命令：印出工作目录。这两个VFS节点本身有计数字段，随着一个或多个进程引用它们而增长。这就是为什么你不能删除一个进程设为工作目录的目录。

Virtual memory 多数进程都有一些虚拟内存（核心线程和核心守护进程没有），Linux核心必须知道这些虚拟内存是如何映射到系统的物理内存中的。

Processor Specific Context 进程可以看作是系统当前状态的总和。只要进程运行，它就要使用处理器的寄存器、堆栈等等。当一个进程暂停的时候，这些进程的上下文、和CPU相关的上下文必须保存到进程的task_struct结构中。当调度者重新启动这个进程的时候，它的上下文就从这里恢复。

## 4.2 Identifiers （标识）

Linux，象所有的Unix，使用用户和组标识符来检查对于系统中的文件和映像的访问权限。Linux系统中所有的文件都有所有权和许可，这些许可描述了系统对于该文件或目录拥有什么样的权限。基本的权限是读、写和执行，并分配了3组用户：文件属主、属于特定组的进程和系统中的其他进程。每一组用户都可以拥有不同的权限，例如一个文件可以让它的属主读写，它的组读，而系统中的其他进程不能访问。

Linux使用组来给一组用户赋予对文件或者目录的权限，而不是对系统中的单个用户或者进程赋予权限。比如你可以为一个软件项目中的所有用户创建一个组，使得只有他们才能够读写项目的源代码。一个进程可以属于几个组（缺省是32个），这些组放在每一个进程的task_struct结构中的groups向量表中。只要进程所属的其中一个组对于一个文件有访问权限，则这个进程就又对于这个文件的适当的组权限。

一个进程的task_struct中有4对进程和组标识符。

**Uid,gid** 该进程运行中所使用的用户的标识符和组的标识符

**Effective uid and gid** 一些程序把执行进程的uid和gid 改变为它们自己的（在VFS I节点执行映像的属性中）。这些程序叫做setuid程序。这种方式有用，因为它可以限制对于服务的访问，特别是那些用其他人的方式运行的，例如网络守护进程。有效的uid 和gid来自setuid程序，而uid和gid 仍旧是原来的。核心检查特权的时候检查有效 uid和gid。

**File system uid and gid** 通常和有效uid和gid相等，检查对于文件系统的访问权限。用于通过NFS安装的文件系统。这时用户态的NFS服务器需要象一个特殊进程一样访问文件。只有文件系

统uid和gid改变（而非有效uid和gid）。这避免了恶意用户向NFS的服务程序发送Kill信号。Kill用一个特别的有效uid和gid发送给进程。

**Saved uid and gid** 这是POSIX标准的要求，让程序可以通过系统调用改变进程的uid和gid。用于在原来的uid和gid改变之后存储真实的uid和gid。

## 4.3 Scheduling （调度）

所有的进程部分运行与用户态，部分运行于系统态。底层的硬件如何支持这些状态各不相同但是通常有一个安全机制从用户态转入系统态并转回来。用户态比系统态的权限低了很多。每一次进程执行一个系统调用，它都从用户态切换到系统态并继续执行。这时让核心执行这个进程。Linux中，进程不是互相争夺成为当前运行的进程，它们无法停止正在运行的其它进程然后执行自身。每一个进程在它必须等待一些系统事件的时候会放弃CPU。例如，一个进程可能不得不等待从一个文件中读取一个字符。这个等待发生在系统态的系统调用中。进程使用了库函数打开并读文件，库函数又执行系统调用从打开的文件中读入字节。这时，等候的进程会被挂起，另一个更加值得的进程将会被选择执行。进程经常调用系统调用，所以经常需要等待。即使进程执行到需要等待也有可能会用去不均衡的CPU事件，所以Linux使用抢先式的调度。用这种方案，每一个进程允许运行少量一段时间，200毫秒，当这个时间过去，选择另一个进程运行，原来的进程等待一段时间直到它又重新运行。这个时间段叫做时间片。

需要调度程序选择系统中所有可以运行的进程中最值得的进程。一个可以运行的进程是一个只等待CPU的进程。Linux使用合理而简单的基于优先级的调度算法在系统当前的进程中进行选择。当它选择了准备运行的新进程，它就保存当前进程的状态、和处理器相关的寄存器和其他需要保存的上下文信息到进程的task_struct数据结构中。然后恢复要运行的新的进程的状态（又和处理器相关），把系统的控制交给这个进程。为了公平地在系统中所有可以运行（runnable）的进程之间分配CPU时间，调度程序在每一个进程的task_struct结构中保存了信息：

参见 kernel/sched.c schedule()

policy 进程的调度策略。Linux有两种类型的进程：普通和实时。实时进程比所有其它进程的优先级高。如果有一个实时的进程准备运行，那么它总是先被运行。实时进程有两种策略：环或先进先出（round robin and first in first out）。在环的调度策略下，每一个实时进程依次运行，而在先进先出的策略下，每一个可以运行的进程按照它在调度队列中的顺序运行，这个顺序不会改变。

Priority 进程的调度优先级。也是它允许运行的时候可以使用的时间量（jiffies）。你可以通过系统调用或者renice命令来改变一个进程的优先级。

Rt_priority Linux支持实时进程。这些进程比系统中其他非实时的进程拥有更高的优先级。这个域允许调度程序赋予每一个实时进程一个相对的优先级。实时进程的优先级可以用系统调用来修改

Coutner 这时进程可以运行的时间量（jiffies）。进程启动的时候等于优先级（priority），每一次时钟周期递减。

调度程序从核心的多个地方运行。它可以在把当前进程放到等待队列之后运行，也可以在系统调用之后进程从系统态返回进程态之前运行。需要运行调度程序的另一个原因是系统时钟刚好把当前进程的计数器(counter)置成了0。每一次调度程序运行它做以下工作：

参见 kernel/sched.c schedule()

kernel work 调度程序运行bottom half handler并处理系统的调度任务队列。这些轻量级的核心线程在第11章详细描述

Current pocess 在选择另一个进程之前必须处理当前进程。

如果当前进程的调度策略是环则它放到运行队列的最后。

如果任务是可中断的而且它上次调度的时候收到过一个信号，它的状态变为RUNNING

如果当前进程超时，它的状态成为RUNNING

如果当前进程的状态为RUNNING则保持此状态

不是RUNNING或者INTERRUPTIBLE的进程被从运行队列中删除。这意味着当调度程序查找最值得运行的进程时不会考虑这样的进程。

Process Selection 调度程序查看运行队列中的进程，查找最值得运行的进程。如果有实时的进程（具有实时调度策略），就会比普通进程更重一些。普通进程的重量是它的counter，但是对于实时进程则是counter 加1000。这意味着如果系统中存在可运行的实时进程，就总是在任何普通可运行的进程之前运行。当前的进程，因为用掉了一些时间片（它的counter减少了），所以如果系统中由其他同等优先级的进程，就会处于不利的位置：这也是应该的。如果几个进程又同样的优先级，最接近运行队列前段的那个就被选中。当前进程被放到运行队列的后面。如果一个平衡的系统，拥有大量相同优先级的进程，那么回按照顺序执行这些进程。这叫做环型调度策略。不过，因为进程需要等待资源，它们的运行顺序可能会变化。

Swap Processes 如果最值得运行的进程不是当前进程，当前进程必须被挂起，运行新的进程。当一个进程运行的时候它使用了CPU和系统的寄存器和物理内存。每一次它调用例程都通过寄存器或者堆栈传递参数、保存数值比如调用例程的返回地址等。因此，当调度程序运行的时候它在当前进程的上下文运行。它可能是特权模式：核心态，但是它仍旧是当前运行的进程。当这个进程要挂起时，它的所有机器状态，包括程序计数器(PC)和所有的处理器寄存器，必须存到进程的task_struct数据结构中。然后，必须加载新进程的所有机器状态。这种操作依赖于系统，不同的CPU不会完全相同地实现，不过经常都是通过一些硬件的帮助。

交换出去进程的上下文发生在调度的最后。前一个进程存储的上下文，就是当这个进程在调度结束的时候系统的硬件上下文的快照。相同的，当加载新的进程的上下文时，仍旧是调度结束时的快照，包括进程的程序计数器和寄存器的内容。

如果前一个进程或者新的当前进程使用虚拟内存，则系统的页表需要更新。同样，这个动作适合体系结构相关。Alpha AXP处理器，使用TLT（Translation Look-aside Table）或者缓存的页表

条目，必须清除属于前一个进程的缓存的页表条目。

### 4.3.1 Scheduling in Multiprocessor Systems（多处理器系统中的调度）

在Linux世界中，多CPU系统比较少，但是已经做了大量的工作使Linux成为一个SMP（对称多处理）的操作系统。这就是，可以在系统中的CPU之间平衡负载的能力。负载均衡没有比在调度程序中更重要的了。

在一个多处理器的系统中，希望的情况是：所有的处理器都繁忙地运行进程。每一个进程都独立地运行调度程序直到它的当前的进程用完时间片或者不得不等待系统资源。SMP系统中第一个需要注意的是系统中可能不止一个空闲（idle）进程。在一个单处理器的系统中，空闲进程是task向量表中的第一个任务，在一个SMP系统中，每一个CPU都有一个空闲的进程，而你可能有不止一个空闲CPU。另外，每一个CPU有一个当前进程，所以SMP系统必须记录每一个处理器的当前和空闲进程。

在一个SMP系统中，每一个进程的task_struct都包含进程当前运行的处理器编号（processor）和上次运行的处理器编号（last_processor）。为什么进程每一次被选择运行时不要在不同的CPU上运行是没什么道理的，但是Linux可以使用processor_mask把进程限制在一个或多个CPU上。如果位N置位，则该进程可以运行在处理器N上。当调度程序选择运行的进程的时候，它不会考虑processor_mask相应位没有设置的进程。调度程序也会利用上一次在当前处理器运行的进程，因为把进程转移到另一个处理器上经常会有性能上的开支。

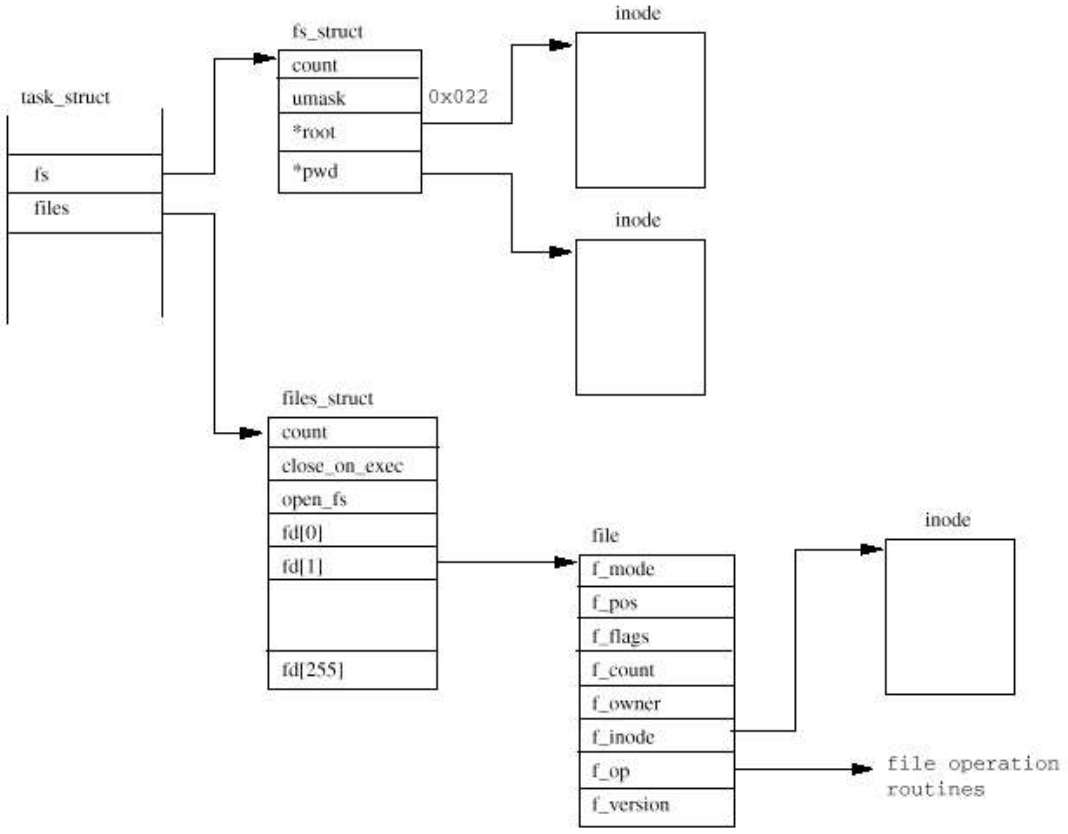



Figure 1.1: A Process's Files

## 4.4 Files（文件）

图4.1显示了描述系统每一个进程中的用于描述和文件系统相关的信息的两个数据结构。第一个fs_struct包括了这个进程的VFS I节点和它的umask。Umask是新文件创建时候的缺省模式，可以通过系统调用改变。

参见include/linux/sched.h

第二个数据结构，files_struct，包括了进程当前使用的所有文件的信息。程序从标准输入读取，向标准输出写，错误信息输出到标准错误。这些可以是文件，终端输入/输出或者世纪的设备，但是从程序的角度它们都被看作是文件。每一个文件都有它的描述符，files_struct包括了指向256个file数据结果，每一个描述进程形用的文件。F_mode域描述了文件创建的模式：只读、读写或者只写。F_pos记录了下一次读写操作在文件中的位置。F_inode指向描述该文件的I节点，f_ops是指向一组例程地址的指针，每一个地址都是一个用于处理文件的函数。例如写数据的函数。这种抽象的接口非常强大，使得Linux可以支持大量的文件类型。我们可以看到，在Linux中pipe也是用这种机制实现的。

每一次打开一个文件，就使用files_struct中的一个空闲的file指针指向这个新的file结构。Linux进程启动时有3个文件描述符已经打开。这就是标准输入、标准输出和标准错误，这都是从创建它们的父进程中继承过来的。对于文件的访问都是通过标准的系统调用，需要传递或返回文件描述

符。这些描述符是进程的fd向量表中的索引，所以标准输入、标准输出和标准错误的文件描述符分别是0，1和2。对于文件的所有访问都是利用file数据结构中的文件操作例程和它的VFS I节点一起来实现的。

## 4.5 Virtual Memory（虚拟内存）

进程的虚拟内存包括多种来源的执行代码和数据。第一种是加载的程序映像，例如ls命令。这个命令，象所有的执行映像一样，由执行代码和数据组成。映像文件中包括将执行代码和相关的程序数据加载到进程地虚拟内存中所需要的所有信息。第二种，进程可以在处理过程中分配（虚拟）内存，比如用于存放它读入的文件的内容。新分配的虚拟内存需要连接到进程现存的虚拟内存中才能使用。第三中，Linux进程使用通用代码组成的库，例如文件处理。每一个进程都包括库的一份拷贝没有意义，Linux使用共享库，几个同时运行的进程可以共用。这些共享库里边的代码和数据必须连接到该进程的虚拟地址空间和其他共享该库的进程的虚拟地址空间。

在一个特定的时间，进程不会使用它的虚拟内存中包括的所有代码和数据。它可能包括旨在特定情况下使用的代码，比如初始化或者处理特定的事件。它可能只是用了它的共享库中一部分例程。如果把所有这些代码都加载到物理内存中而不使用只会是浪费。把这种浪费和系统中的进程数目相乘，系统的运行效率会很低。Linux改为使用demand paging 技术，进程的虚拟内存只在进程试图使用的时候才调入物理内存中。所以，Linux不把代码和数据直接加载到内存中，而修改进程的页表，把这些虚拟区域标志为存在但是不在内存中。当进程试图访问这些代码或者数据，系统硬件会产生一个page fault，把控制传递给Linux核心处理。因此，对于进程地址空间的每一个虚拟内存区域，Linux需要直到它从哪里来和如何把它放到内存中，这样才可以处理这些page fault。

Linux核心需要管理所有的这些虚拟内存区域，每一个进程的虚拟内存的内容通过一个它的task_struct指向的一个mm_struct　mm_struc数据结构描述。该进程的mm_struct数据结构也包括加载的执行映像的信息和进程页表的指针。它包括了指向一组vm_area_struct数据结构的指针，每一个都表示该进程中的一个虚拟内存区域。

这个链接表按照虚拟内存顺序排序。图4.2显示了一个简单进程的虚拟内存分布和管理它的核心数据结构。因为这些虚拟内存区域来源不同，Linux通过vm_area_struct指向一组虚拟内存处理例程（通过vm_ops）的方式抽象了接口。这样进程的所有虚拟内存都可以用一种一致的方式处理，不管底层管理这块内存的服务如何不同。例如，会有一个通用的例程，在进程试图访问不存在的内存时调用，这就是page fault 的处理。

当Linux为一个进程创建新的虚拟内存区域和处理对于不在系统物理内存中的虚拟内存的引用时，反复引用进程的vm_area_struct数据结构列表。这意味着它查找正确的vm_area_struct数据结构所花的事件对于系统的性能十分重要。为了加速访问，Linux也把vm_area_struct数据结构放到一个AVL（Adelson-Velskii and Landis）树。对这个树进行安排使得每一个

vm_area_struct（或节点）都有对相邻的vm_area_struct结构的一个左和一个右指针。左指针指向拥有较低起始虚拟地址的节点，右指针指向一个拥有较高起始虚拟地址的节点。为了找到正确的节点，Linux从树的根开始，跟从每一个节点的左和右指针，直到找到正确的vm_area_struct。当然，在这个树中间释放不需要时间，而插入新的vm_area_struct需要额外的处理时间。

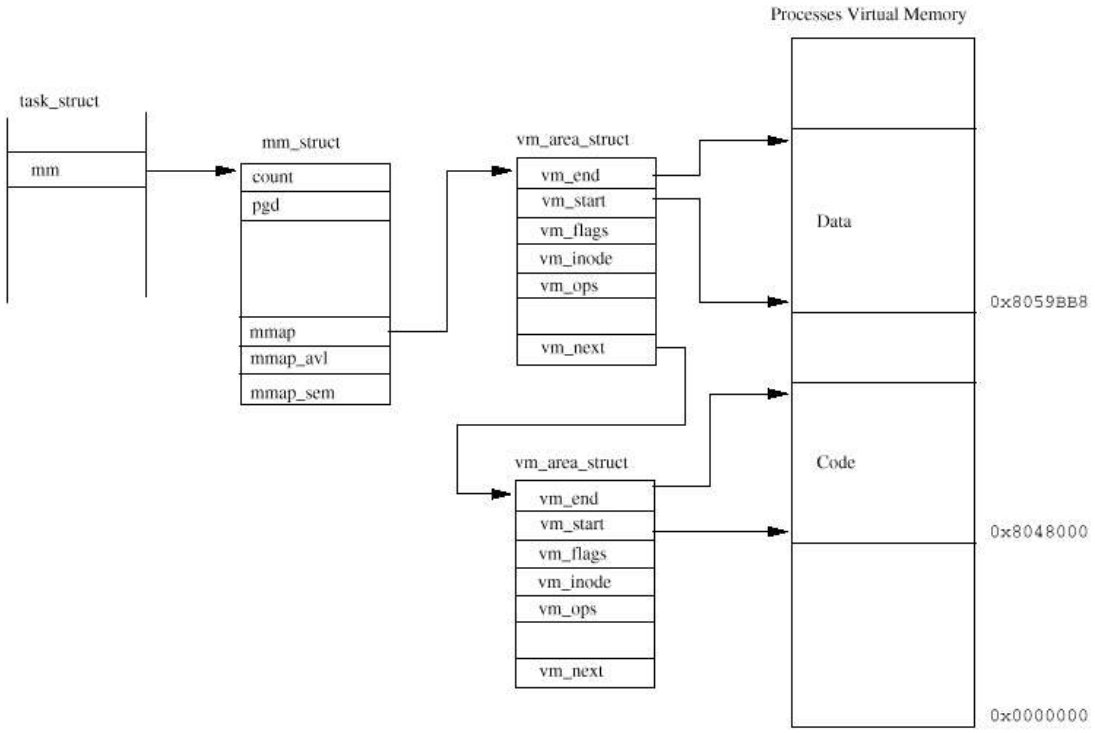



Figure 4.2: A Process's Virtual Memory

当一个进程分配虚拟内存的时候，Linux并不为该进程保留物理内存。它通过一个新的vm_area_struct数据结构来描述这块虚拟内存，连接到进程的虚拟内存列表中。当进程试图写这个新的虚拟内存区域的时候，系统会发生page fault。处理器试图解码这个虚拟地址，但是没有对应该内存的页表条目，它会放弃并产生一个page fault异常，让Linux核心处理。Linux检查这个引用的虚拟地址是不是在进程的虚拟地址空间， 如果是，Linux创建适当的PTE并为该进程分配物理内存页。也许需要从文件系统或者交换磁盘中加载相应的代码或者数据，然后进程从引起page fault的指令重新运行，因为这次该内存实际存在，可以继续。

## 4.6 Creating a Process（创建一个进程）

当系统启动的时候它运行在核心态，这时，只有一个进程：初始化进程。象所有其他进程一样，初始进程有一组用堆栈、寄存器等等表示的机器状态。当系统中的其他进程创建和运行的时候这些信息存在初始进程的task_struct数据结构中。在系统初始化结束的时候，初始进程启动一个核心线程（叫做init）然后执行空闲循环，什么也不做。当没有什么可以做的时候，调度程序会运行这个空闲的进程。这个空闲进程的task_struct是唯一一个不是动态分配而是在核心连接的时候静态定义的，为了不至于混淆，叫做init_task。

Init核心线程或进程拥有进程标识符1，是系统的第一个真正的进程。它执行系统的一些初始化的设置（比如打开系统控制它，安装根文件系统），然后执行系统初始化程序。依赖于你的系统，可能是/etc/init，/bin/init或/sbin/init其中之一。Init程序使用/etc/inittab作为脚本文件创建系统中的新进程。这些新进程自身可能创建新的进程。例如：getty进程可能会在用户试图登录的时候创建一个login的进程。系统中的所有进程都是init核心线程的后代。

新的进程的创建是通过克隆旧的进程，或者说克隆当前的进程来实现的。一个新的任务是通过系统调用创建的（fork或clone），克隆发生在核心的核心态。在系统调用的最后，产生一个新的进程，等待调度程序选择它运行。从系统的物理内存中为这个克隆进程的堆栈（用户和核心）分配一个或多个物理的页用于新的task_struct数据结构。一个进程标识符将会创建，在系统的进程标识符组中是唯一的。但是，也可能克隆的进程保留它的父进程的进程标识符。新的task_struct进入了task 向量表中，旧的（当前的）进程的task_struct的内容拷贝到了克隆的task_struct。

参见kernel/fork.c do_fork()

克隆进程的时候，Linux允许两个进程共享资源而不是拥有不同的拷贝。包括进程的文件，信号处理和虚拟内存。共享这些资源的时候，它们相应的count字段相应增减，这样Linux不会释放这些资源直到两个进程都停止使用。例如，如果克隆的进程要共享虚拟内存，它的task_struct会包括一个指向原来进程的mm_struct的指针，mm_struct的count域增加，表示当前共享它的进程数目。

克隆一个进程的虚拟内存要求相当的技术。必须产生一组vm_area_struct数据结构、相应的mm_struct数据结构和克隆进程的页表，这时没有拷贝进程的虚拟内存。这会是困难和耗时的任务，因为一部分虚拟内存可能在物理内存中而另一部分可能在交换文件中。替代底，Linux使用了叫做“copy on write”的技术，即只有两个进程中的一个试图写的时候才拷贝虚拟内存。任何不写入的虚拟内存，甚至可能写的，都可以在两个进程之间共享二部会有什么害处。只读的内存，例如执行代码，可以共享。为了实现“copy on write”，可写的区域的页表条目标记为只读，而描述它的vm_area_struct数据结构标记为“copy on write”。当一个进程试图写向着这个虚拟内存的时候会产生page fault。这时Linux将会制作这块内存的一份拷贝并处理两个进程的页表和虚拟内存的数据结构。

### 7. Times and Timer（时间和计时器）

核心跟踪进程的CPU时间和其他一些时间。每一个时钟周期，核心更新当前进程的jiffies来表示在系统和用户态下花费的时间总和。

除了这些记账的计时器，Linux还支持进程指定的间隔计时器（interval timer）。进程可以使用这些计时器在这些计时器到期的时候发送给自身信号。支持三种间隔计时器：

参见kernel/itimer.c

Real 这个计时器使用实时计时，当计时器到期，发送给进程一个SIGALRM信号。

Virtual 这个计时器只在进程运行的时候计时，到期的时候，发送给进程一个SIGVTALARM信号。

Profile 在进程运行的时候和系统代表进程执行的时候都及时。到期的时候，会发送SIGPROF信号。

可以运行一个或者所有的间隔计时器，Linux在进程的task_struct数据结构中记录所有的必要信息。可以使用系统调用建立这些间隔计时器，启动、停止它们，读取当前的数值。虚拟和profile计时器的处理方式相同：每一次时钟周期，当前进程的计时器递减，如果到期，就发出适当的信号

参见kernel/sched.c do_it_virtual()， do_it_prof()

实时间隔计时器稍微不同。Linux使用计时器的机制在第11章描述。每一个进程都有自己的timer_list数据结构，当时使用实时计时器的时候，使用系统的timer表。当它到期的时候，计时器后半部分处理把它从队列中删除并调用间隔计时器处理程序。它产生SIGALRM信号并重启动间隔计时器，把它加回到系统计时器队列。

参见：kernel/iterm.c it_real_fn()

8. Executing Programs （执行程序）

在Linux中，象Unix一样，程序和命令通常通过命令解释器执行。命令解释程序是和其他进程一样的用户进程，叫做shell（想象一个坚果，把核心作为中间可食的部分，而shell包围着它，提供一个接口）。Linux中有许多shell，最常用的是sh、bash和tcsh。除了一些内部命令之外，比如cd和pwd，命令是可执行的二进制文件。对于输入的每一个命令，shell在当前进程的搜索路径指定的目录中（放在PATH环境变量）查找匹配的名字。如果找到了文件，就加载并运行。Shell用上述的fork机制克隆自身，并在子进程中用找到的执行映像文件的内容替换它正在执行的二进制映像（shell）。通常shell等待命令结束，或者说子进程退出。你可以通过输入control-Z发送一个SIGSTOP信号给子进程，把子进程停止并放到后台，让shell重新运行。你可以使用shell命令bg让shell向子进程发送SIGCONT信号，把子进程放到后台并重新运行，它会持续运行直到它结束或者需要从终端输入或输出。

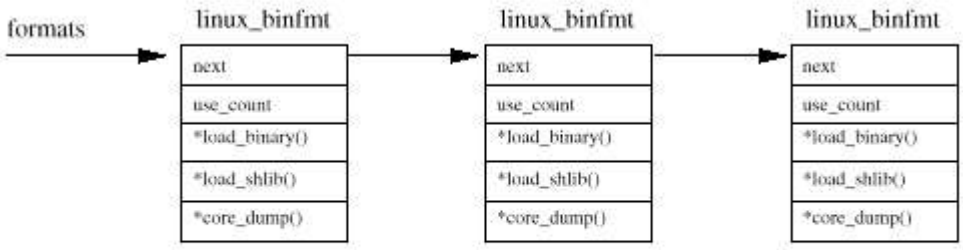



Figure 4.3: Registered Binary Formats

执行文件可以由许多格式甚至可以是一个脚本文件（script file）。脚本文件必须用合适的解释程序识别并运行。例如/bin/sh解释shell script。可执行的目标文件包括了执行代码和数据以及足够的其他信息，时的操作系统可以把它们加载到内存中并执行。Linux中最常用的目标文件类型是ELF，而理论上，Linux灵活到足以处理几乎所有的目标文件格式。

好像文件系统一样，Linux可以支持的二进制格式也是在核心连接的时候直接建立在核心的或者是可以作为模块加载的。核心保存了支持的二进制格式（见图4.3）的列表，当试图执行一个文件的时候，每一个二进制格式都被尝试，直到可以工作。通常，Linux支持的二进制文件是a.out和ELF。可执行文件不需要完全读入内存，而使用叫做demand loading的技术。当进程使用执行映像的一部分的时候它才被调入内存，未被使用的映像可以从内存中废弃。

参见fs/exec.c do_execve()

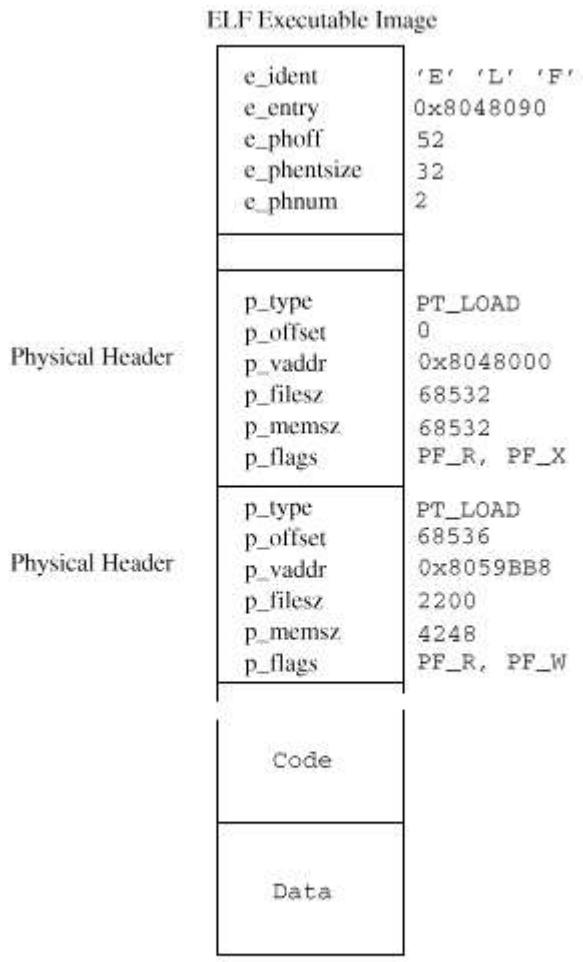



Figure 4.4: ELF Executable File Format

## 1. ELF

ELF（Executable and Linkable Format 可执行可连接格式）目标文件，由Unix系统实验室设计，现在成为Linux最常用的格式。虽然和其他目标文件格式比如ECOFF和a.out相比，有性能上的轻微开支，ELF感觉更灵活。ELF可执行文件包括可执行代码（有时叫做text）和数据（data）。执行映像中的表描述了程序应该如何放到进程的虚拟内存中。静态连接的映像是用连接程序（ld）或者连接编辑器创建的，单一的映像中包括了运行该映像所需要的所有的代码和数据。这个映像也描述了该映像在内存中的布局和要执行的第一部分代码在映像中的地址。

图4.4象是了静态连接的ELF可执行映像的布局。这是个简单的C程序，打印“hello world”然后退出。头文件描述了它是一个ELF映像，有两个物理头（e_phnum是2），从映像文件的开头第52字节开始（e_phoff）。第一个物理头描述映像中的执行代码，在虚拟地址0x8048000，有65532字节。因为它是静态连接的，所以包括输出“hello world”的调用printf（）的所有的库代码。映像的入口，即程序的第一条指令，不是位于映像的起始位置，而在虚拟地址

0x8048090（e_entry）。代码紧接着在第二物理头后面开始。这个物理头描述了程序的数据，将会加载到虚拟内存地址0x8059BB8。这块数据可以读写。你会注意到文件中数据的大小是2200字节（p_filesz）而在内存中的大小是4248字节。因为前2200字节包括预先初始化的数据，而接着的2048字节包括会被执行代码初始化的数据。

参见include/linux/elf.h

当Linux把ELF可执行映像加载到进程的虚拟地址空间的时候，它不是实际的加载映像。它设置虚拟内存数据结构，即进程的vm_area_struct和它的页表。当程序执行了page fault的时候，程序的代码和数据会被放到物理内存中。没有用到的程序部分将不会被放到内存中。一旦ELF 二进制格式加载程序满足条件，映像是一个有效的ELF可执行映像，它把进程的当前可执行映像从它的虚拟内存中清除。因为这个进程是个克隆的映像（所有的进程都是），旧的映像是父进程执行的程序的映像（例如命令解释程序shell bash）。清除旧的可执行映像会废弃旧的虚拟内存的数据结构，重置进程的页表。它也会清除设置的其他信号处理程序，关闭打开的文件。在清除过程的最后，进程准备运行新的可执行映像。不管可执行映像的格式如何，进程的mm_struct中都要设置相同的信息。包括指向映像中代码和数据起始的指针。这些数值从ELF可执行映像的物理头中读入，它们描述的部分也被映射到了进程的虚拟地址空间。这也发生在进程的vm_area_struct数据结构建立和页表修改的时候。mm_struct数据结构中也包括指针，指向传递给程序的参数和进程的环境变量。

ELF Shared Libraries（ELF共享库）

动态连接的映像，反过来，不包含运行所需的所有的代码和数据。其中一些放在共享库并在运行的时候连接到映像中。当运行时动态库连接到映像中的时候，动态连接程序（dynamic linker）也要使用ELF共享库的表。Linux使用几个动态连接程序，ld.so.1，libc.so.1和ld-linux.so.1，都在/lib目录下。这些库包括通用的代码，比如语言子例程。如果没有动态连接，所有的程序都必须有这些库的独立拷贝，需要更多的磁盘空间和虚拟内存。在动态连接的情况下，ELF映像的表中包括引用的所有库例程的信息。这些信息指示动态连接程序如何定位库例程以及如何连接到程序的地址空间。

## 2. Scripts Files

脚本文件是需要解释器才能运行的可执行文件。Linux下有大量的解释器，例如wish、perl和命令解释程序比如tcsh。Linux使用标准的Unix约定，在脚本文件的第一行包括解释程序的名字。所以一个典型的脚本文件可能开头是：

#!/usr/bin/wish

脚本文件加载器试图找出文件所用的解释程序。它试图打开脚本文件第一行指定的可执行文件。如果可以打开，就得到一个指向该文件的VFS I 节点的指针，然后执行它去解释脚本文件。脚本文件的名字成为了参数0（第一个参数），所有的其他参数都向上移动一位（原来的第一个参数成为了第二个参数等等）。加载解释程序和Linux加载其他可执行程序一样。Linux依次尝试各种二进制格式，直到可以工作。这意味着理论上你可以把几种解释程序和二进制格式堆积起来，让Linux的二进制格式处理程序更加灵活。

参见fs/binfmt_script.c do_load_script()

## Chapter 5

## Interprocess Communication Mechanisms（进程间通讯机制）

进程之间互相通讯并和核心通讯，协调它们的行为。Linux支持一些进程间通讯（IPC）的机制。信号和管道是其中的两种，Linux还支持系统V IPC（用首次出现的Unix的版本命名）的机制。

### 5.1 Signals（信号）

信号是Unix系统中使用的最古老的进程间通讯的方法之一。用于向一个或多个进程发送异步事件的信号。信号可以用键盘终端产生，或者通过一个错误条件产生，比如进程试图访问它的虚拟内存中不存在的位置。Shell也使用信号向它的子进程发送作业控制信号。

有一些信号有核心产生，另一些可以由系统中其他有权限的进程产生。你可以使用kill命令（kill –l）列出你的系统的信号集，在我的Linux Intel系统输出：

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

在Alpha AXP Linux系统上编号不同。进程可以选择忽略产生的大多数信号，有两个例外：SIGSTOP（让进程停止执行）和SIGKILL（让进程退出）不可以忽略，虽然进程可以选择它如何处理信号。进程可以阻塞信号，如果它不阻塞信号，它可以选择自己处理或者让核心处理。如果核心处理，将会执行该信号的缺省行为。例如，进程接收到SIGFPE（浮点意外）的缺省动作是产生core并退出。信号没有固有的优先级，如果一个进程同时产生了两个信号，它们会以任意顺序出现在进程中并按任意顺序处理。另外，也没有机制可以处理统一种类的多个信号。进程无法知道它接收了1还是42个SIGCONT信号。

Linux用进程的task_struct中存放的信息来实现信号机制。支持的信号受限于处理器的字长。32位字长的处理器可以有32中信号，而64位的处理器，比如Alpha AXP可以有多达64种信号。当前待处理的信号放在signal域，blocked域放着要阻塞的信号掩码。除了SIGSTOP和SIGKILL，所有的信号都可以被阻塞。如果产生了一个被阻塞的信号，它一直保留待处理，直到被解除阻塞。Linux也保存每一个进程如何处理每一种可能的信号的信息，这些信息放在一个sigaction的数据结构数组中，每一个进程的task_struct都有指针指向对应的数组。这个数组中包括处理这个信号的例程的地址，或者包括一个标志，告诉Linux该进程是希望忽略这个信号还是让核心处理。进程通过执行系统调用改变缺省的信号处理，这些调用改变适当的信号的sigaction和阻塞的掩码。

并非系统中所有的进程都可以向其他每一个进程发送信号，只有核心和超级用户可以。普通进程只可以向拥有相同uid和gid或者在相同进程组的进程发送信号。通过设置task——struct的signal中适当的位产生信号。如果进程不阻塞信号，而且正在等待但是可以中断（状态是Interruptible），那么它的状态被改为Running并确认它在运行队列，通过这种方式把它唤醒。这样调度程序在系统下次调度的时候会把它当作一个运行的候选。如果需要缺省的处理，Linux可以优化信号的处理。例如如果信号SIGWINCH（X window改变焦点）发生而使用缺省的处理程序，则不需要做什么事情。

信号产生的时候不会立刻出现在进程中，它们必须等到进程下次运行。每一次进程从系统调用中退出的时候都要检查它的signal和blocked域，如果有任何没有阻塞的信号，就可以发送。这看起来好像非常不可靠，但是系统中的每一个进程都在调用系统调用，比如向终端写一个字符的过程中。如果愿意，进程可以选择等待信号，它们挂起在Interruptible状态，直到有了一个信号。Linux信号处理代码检查sigaction结构中每一个当前未阻塞的信号。

如果信号处理程序设置为缺省动作，则核心会处理它。SIGSTOP信号的缺省处理是把当前进程的状态改为Stopped，然后运行调度程序，选择一个新的进程来运行。SIGFPE信号的缺省动作是让当前进程产生core（core dump），让它退出。变通地，进程可以指定自己的信号处理程序。这是一个例程，当信号产生的时候调用而且sigaction结构包括这个例程的地址。Linux必须调用进

程的信号处理例程，至于具体如何发生是和处理器相关。但是，所有的CPU必须处理的是当前进程正运行在核心态，并正准备返回到调用核心或系统例程的用户态的进程。解决这个问题的方法是处理该进程的堆栈和寄存器。进程程序计数器设为它的信号处理程序的地址，例程的参数加到调用结构或者通过寄存器传递。当进程恢复运行的时候显得信号处理程序是正常的调用。

Linux是POSIX兼容的，所以进程可以指定调用特定的信号处理程序的时候要阻塞的信号。这意味着在调用进程的信号处理程序的时候改变blocked掩码。信号处理程序结束的时候，blocked掩码必须恢复到它的初始值。因此，Linux在收到信号的进程的堆栈中增加了对于一个整理例程的调用，把blocked掩码恢复到初始值。Linux也优化了这种情况：如果同时几个信号处理例程需要调用的时候，就在它们堆积在一起，每次退出一个处理例程的时候就调用下一个，直到最后才调用整理例程。

## 5.2 Pipes（管道）

普通的Linux shell都允许重定向。例如：

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

把列出目录文件的命令ls的输出通过管道接到pr命令的标准输入上进行分页。最后，pr命令的标准输出通过管道连接到lpr命令的标准输入上，在缺省打印机上打印出结果。管道是单向的字节流，把一个进程的标准输出和另一个进程的标准输入连接在一起。没有一个进程意识到这种重定向，和它平常一样工作。是shell建立了进程之间的临时管道。在Linux中，使用指向同一个临时VFS I节点（本身指向内存中的一个物理页）的两个file数据结构来实现管道。图5.1显示了每一个file数据结构包含了不同的文件操作例程的向量表的指针：一个用于写，另一个从管道中读。这掩盖了和通用的读写普通文件的系统调用的不同。当写进程向管道中写的时候，字节拷贝到了共享的数据页，当从管道中读的时候，字节从共享页中拷贝出来。Linux必须同步对于管道的访问。必须保证管道的写和读步调一致，它使用锁、等待队列和信号（locks，wait queues and signals）。

参见include/linux/inode_fs.h

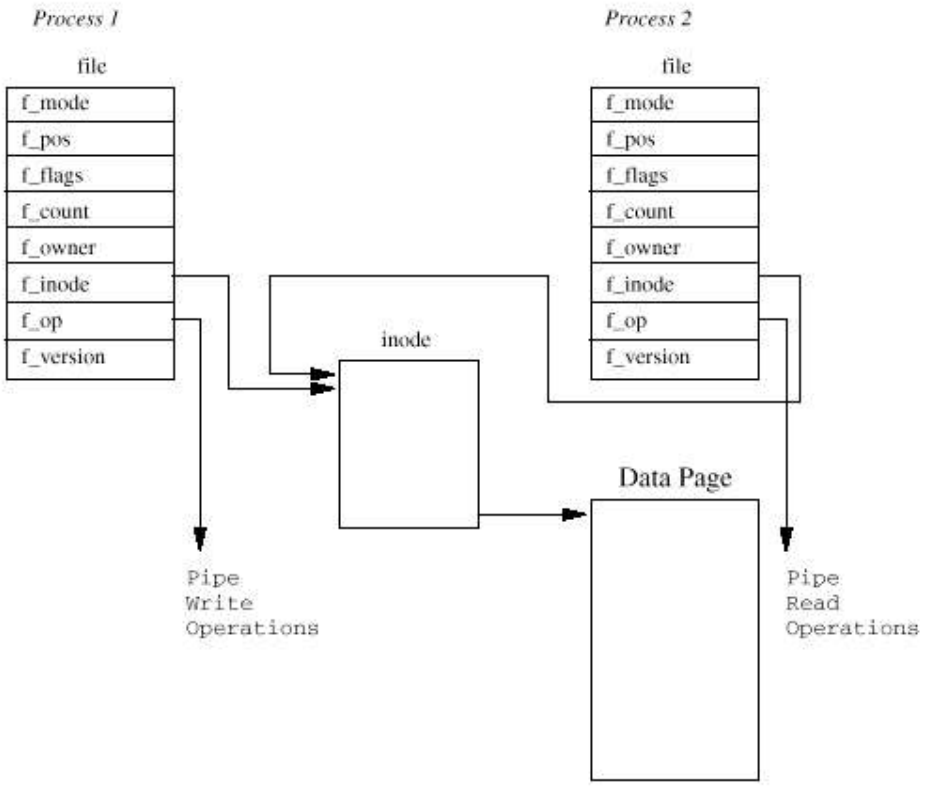



Figure 5.1: Pipes

当写进程向管道写的时候，它使用标准的write库函数。这些库函数传递的文件描述符是进程的file数据结构组中的索引，每一个都表示一个打开的文件，在这种情况下，是打开的管道。Linux系统调用使用描述这个管道的file数据结构指向的write例程。这个write例程使用表示管道的VFS I节点存放的信息，来管理写的请求。如果有足够的空间把所有的字节都写导管到中，只要管道没有被读进程锁定，Linux为写进程上锁，并把字节从进程的地址空间拷贝到共享的数据页。如果管道被读进程锁定或者空间不够，当前进程睡眠，并放在管道I节点的等待队列中，并调用调度程序，运行另外一个进程。它是可以中断的，所以它可以接收信号。当管道中有了足够的空间写数据或者锁定解除，写进程就会被读进程唤醒。当数据写完之后，管道的VFS I 节点锁定解除，管道I节点的等待队列中的所有读进程都会被唤醒。

参见fs/pipe.c pipe_write()

从管道中读取数据和写数据非常相似。进程允许进行非阻塞的读（依赖于它们打开文件或者管道的模式），这时，如果没有数据可读或者管道被锁定，会返回一个错误。这意味着进程会继续运行。另一种方式是在管道的I节点的等待队列中等待，直到写进程完成。如果管道的进程都完成了操作，管道的I节点和相应的共享数据页被废弃。

参见fs/pipe.c pipe_read()

Linux也可以支持命名管道，也叫FIFO，因为管道工作在先入先出的原则下。首先写入管道的数据是首先被读出的数据。不想管道，FIFO不是临时的对象，它们是文件系统中的实体，可以用

mkfifo命令创建。只要有合适的访问权限，进程就可以使用FIFO。FIFO的大开方式和管道稍微不同。一个管道（它的两个file数据结构，VFS I节点和共享的数据页）是一次性创建的，而FIFO是已经存在，可以由它的用户打开和关闭的。Linux必须处理在写进程打开FIFO之前打开FIFO读的进程，以及在写进程写数据之前读的进程。除了这些，FIFO几乎和管道的处理完全一样，而且它们使用一样的数据结构和操作。

### 3. Sockets

注意：写网络篇的时候加上去

#### 1. System V IPC mechanisms （系统V IPC机制）

Linux支持三种首次出现在Unix 系统V（1983）的进程间通讯的机制：消息队列、信号灯和共享内存（message queues， semaphores and shared memory）。系统V IPC机制共享通用的认证方式。进程只能通过系统调用，传递一个唯一的引用标识符到核心来访问这些资源。对于系统V IPC对象的访问的检查使用访问许可权，很象对于文件访问的检查。对于系统V IPC对象的访问权限由对象的创建者通过系统调用创建。每一种机制都使用对象的引用标识符作为资源表的索引。这不是直接的索引，需要一些操作来产生索引。

系统中表达系统V IPC对象的所有Linux数据结构都包括一个ipc_perm的数据结构，包括了创建进程的用户和组标识符，对于这个对象的访问模式（属主、组和其他）和IPC对象的key。Key 用作定位系统V IPC对象的引用标识符的方法。支持两种key：公开和四有的。如果key是公开的,那么系统中的任何进程,只要通过了权限检查,就可以找到对应的系统V IPC对象的引用标识符。系统V IPC对象不能使用key引用，必须使用它们的引用标识符。

参见include/linux/ipc.h

#### 2. Message Queues（消息队列）

消息队列允许一个或多个进程写消息，一个或多个进程读取消息。Linux维护了一系列消息队列的msgque 向量表。其中的每一个单元都指向一个msqid_ds的数据结构，完整描述这个消息队列。当创建消息队列的时候，从系统内存中分配一个新的msqid_ds的数据结构并插入到向量表中

每一个msqid_ds数据结构都包括一个ipc_perm的数据结构和进入这个队列的消息的指针。另外，Linux保留队列的改动时间，例如上次队列写的时间等。Msqid_ds队列也包括两个等待队列：一个用于向消息队列写，另一个用于读。

参见include/linux/msg.h

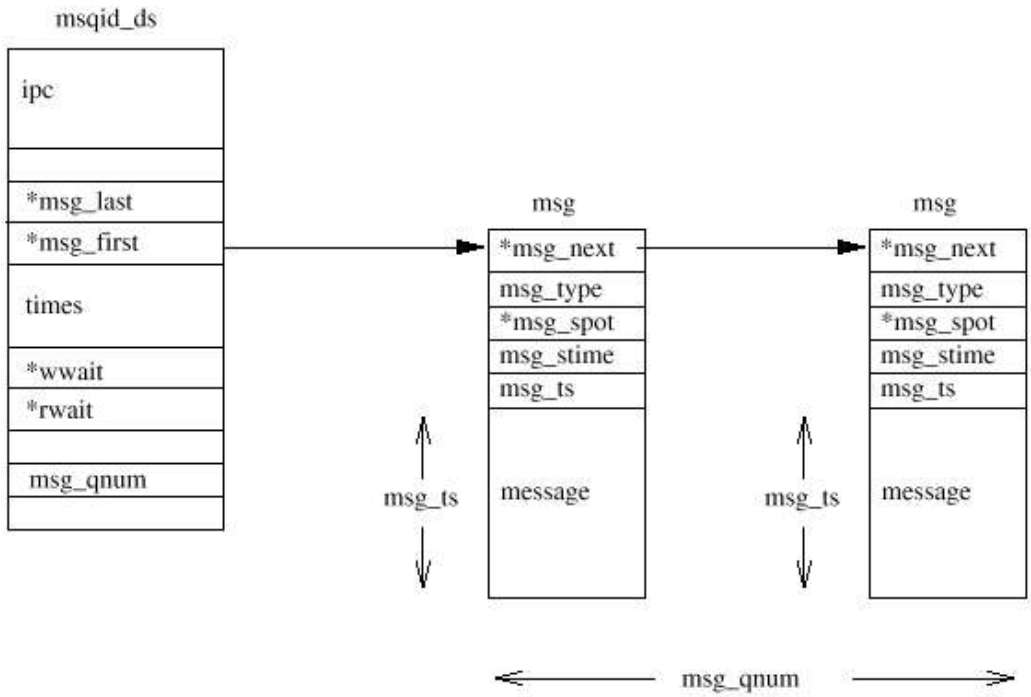



Figure 5.2: System V IPC Message Queues

每一次一个进程试图向写队列写消息，它的有效用户和组的标识符就要和队列的ipc_perm数据结构的模式比较。如果进程可以想这个队列写，则消息会从进程的地址空间写到msg数据结构，放到消息队列的最后。每一个消息都带有进程间约定的，应用程序指定类型的标记。但是，因为Linux限制了可以写的消息的数量和长度，可能会没有空间容纳消息。这时，进程会被放到消息队列的写等待队列，然后调用调度程序选择一个新的进程运行。当一个或多个消息从这个消息队列中读出去的时候会被唤醒。

从队列中读是一个相似的过程。进程的访问权限一样被检查。一个读进程可以选择是不管消息的类型从队列中读取第一条消息还是选择特殊类型的消息。如果没有符合条件的消息，读进程会被加到消息队列的读等待进程，然后运行调度程序。当一个新的消息写到队列的时候，这个进程会被唤醒，继续运行。

### 3. Semaphores（信号灯）

信号灯最简单的形式就是内存中一个位置，它的取值可以由多个进程检验和设置。检验和设置的操作，至少对于关联的每一个进程来讲，是不可中断或者说有原子性：只要启动就不能中止。检验和设置操作的结果是信号灯当前值和设置值

的和，可以是正或者负。根据测试和设置操作的结果，一个进程可能必须睡眠直到信号灯的值被另一个进程改变。信号灯可以用于实现重要区域（critical regions），就是重要的代码区，同一时刻只能有一个进程运行。

比如你有许多协作的进程从一个单一的数据文件读写记录。你可能希望对文件的访问必须严格地协调。你可以使用一个信号灯，初始值1，在文件操作的代码中，加入两个信号灯操作，第一个检查并把信号灯的值减小，第二个检查并增加它。访问文件的第一个进程试图减小信号灯的数值，如果成功，信号灯的取值成为0。这个进程现在可以继续运行并使用数据文件。但是，如果另一个进程需要使用这个文件，现在它试图减少信号灯的数值，它会失败因为结果会是-1。这个进程会被挂起直到第一个进程处理完数据文件。当第一个进程处理完数据文件，它会增加信号灯的数值成为1。现在等待进程会被唤醒，这次它减小信号灯的尝试会成功。

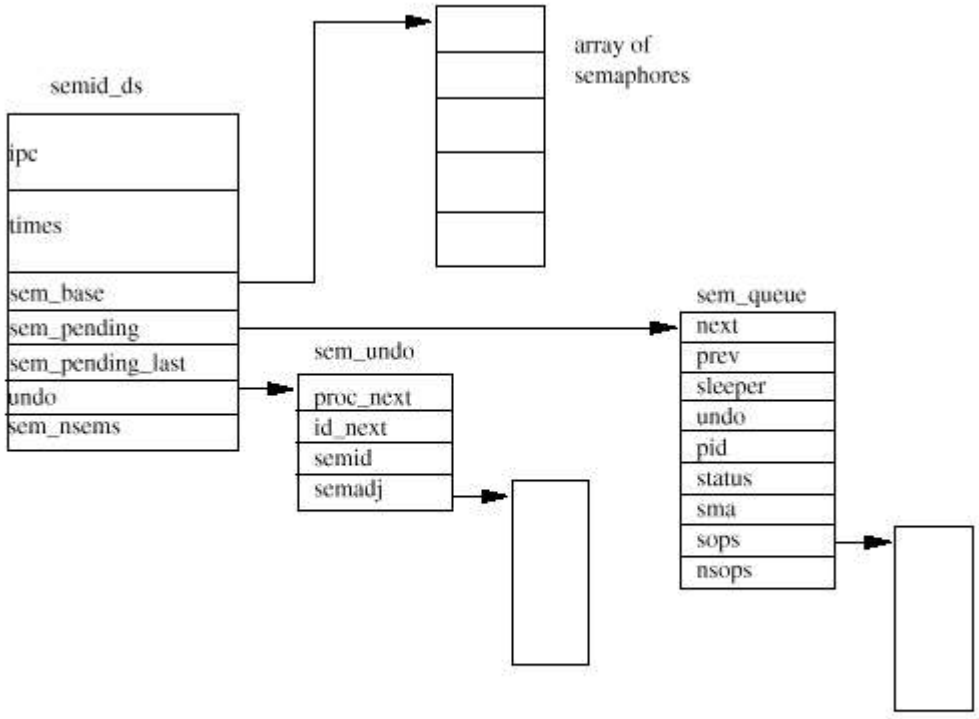



Figure 5.3: System V IPC Semaphores

每一个系统V IPC信号灯对象都描述了一个信号灯数组，Linux使用semid_ds数据结构表达它。系统中所有的semid_ds数据结构都由semary指针向量表指向。每一个信号灯数组中都有sem_nsems，通过sem_base指向的一个sem数据结构来描述。所有允许操作一个系统V IPC信号灯对象的信号灯数组的进程都可以通过系统调用对它们操作。系统调用可以指定多种操作，每一种操作多用三个输入描述：信号灯索引、操作值和一组标志。信号灯索引是信号灯数组的索引，操作值是要增加到当前信号灯取值的数值。首先，Linux检查所有的操作是否成功。只有操作数加上信号灯的当前值大于0或者操作值和信号灯的当前值都是0，操

作才算成功。如果任意信号灯操作失败，只要操作标记不要求系统调用无阻塞，Linux会挂起这个进程。如果进程要挂起，Linux必须保存要进行的信号灯操作的状态并把当前进程放到等待队列重。它通过在堆栈中建立一个sem_queue的数据结构并填满它来实现上述过程。这个新的sem_queue数据结构被放到了这个信号灯对象的等待队列的结尾（使用sem_pending和sem_pending_last指针）。当前进程被放到了这个sem_queue数据结构的等待队列中（sleeper），调用调度程序，运行另外一个进程。

参见include/linux/sem.h

如果所有的信号灯操作都成功，当前的进程就不需要被挂起。Linux继续向前并把这些操作应用到信号灯数组的合适的成员上。现在Linux必须检查任何睡眠或者挂起的进程，它们的操作现在可能可以实施。Linux顺序查找操作等待队列（sem_pending）中的每一个成员，检查现在它的信号灯操作是否可以成功。如果可以它就把这个sem_queue数据结构从操作等待表中删除，并把这种信号灯操作应用到信号灯数组。它唤醒睡眠的进程，让它在下次调度程序运行的时候可以继续运行。Linux从头到尾检查等待队列，直到不能执行信号灯操作无法唤醒更多的进程为止。

在信号灯操作中有一个问题：死锁（deadlock）。这发生在一个进程改变了信号灯的值进入一个重要区域（critical region）但是因为崩溃或者被kill而没有离开这个重要区域的情况下。Linux通过维护信号灯数组的调整表来避免这种情况。就是如果实施这些调整，信号灯就会返回一个进程的信号灯操作前的状态。这些调整放在sem_undo数据结构中，排在sem_ds数据结构的队列中，同时排在使用这些信号灯的进程的task_struct数据结构的队列中。

每一个独立的信号灯操作可能都需要维护一个调整动作。Linux至少为每一个进程的每一个信号灯数组都维护一个sem_undo的数据结构。如果请求的进程没有，就在需要的时候为它创建一个。这个新的sem_undo数据结构同时在进程的task_struct数据结构和信号灯队列的semid_ds数据结构的队列中排队。对信号灯队列中的信号灯执行操作的时候，和这个操作值相抵消的值加到这个进程的sem_undo数据结构的调整队列这个信号灯的条目上。所以，如果操作值为2，那么这个就在这个信号灯的调整条目上增加-2。

当进程被删除，比如退出的时候，Linux遍历它的sem_undo数据结构组，并实施对于信号灯数组的调整。如果删除信号灯，它的sem_undo数据结构仍旧停留在进程的task_struct队列中，但是相应的信号灯数组标识符标记为无效。这种情况下，清除信号灯的代码只是简单地废弃这个sem_undo数据结构。

### 4. Shared Memory（共享内存）

共享内存允许一个或多个进程通过同时出现在它们的虚拟地址空间的内存通讯。这块虚拟内存的页面在每一个共享进程的页表中都有页表条目引用。但是不需要在所有进程的虚拟内存都有相同的地址。象所有的系统V IPC对象一样，对于共享内存区域的访问通过key控制，并进行访问权限检查。内存共享之后，就不再检查进程如何使用这块内存。它们必须依赖于其他机制，比如系统V的信号灯来同步对于内存的访问。

每一个新创建的内存区域都用一个shmid_ds数据结构来表达。这些数据结构保存在shm_segs向量表中。Shmid_ds数据结构描述了这个共享内存取有多大、多少个进程在使用它以及共享内存如何映射到它们的地址空间。由共享内存的创建者来控制对于这块内存的访问权限和它的key是公开或私有。如果有足够的权限它也可以把共享内存锁定在物理内存中。

参见include/linux/sem.h

每一个希望共享这块内存的进程必须通过系统调用粘附（attach）到虚拟内存。这为该进程创建了一个新的描述这块共享内存的vm_area_struct数据结构。进程可以选择共享内存在它的虚拟地址空间的位置或者由Linux选择一块足够的的空闲区域。

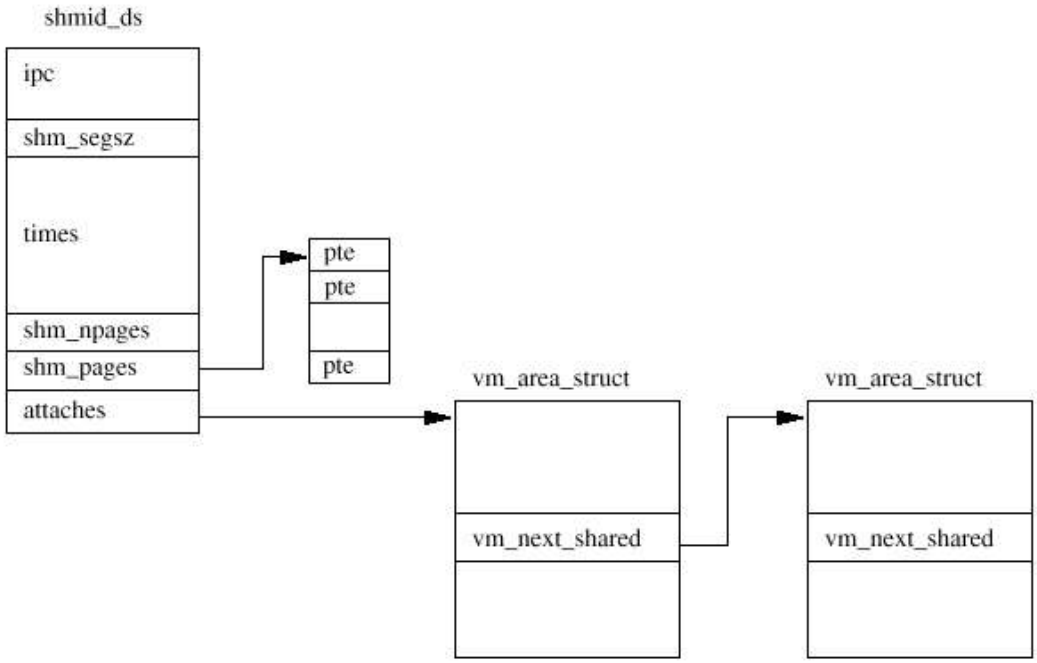



Figure 5.4: System V IPC Shared Memory

这个新的vm_area_struct结构放在由shmid_ds指向的vm_area_struct列表中。通过vm_next_shared和vm_prev_shared把它们连在一起。虚拟内存在粘附的时候其实并没有创建，而发生在第一个进程试图访问它的时候。

在一个进程第一次访问共享虚拟内存的其中一页的时候，发生一个page fault。当Linux处理这个page fault的时候，它找到描述它的vm_area_struct数据结构。这里包含了这类共享虚拟内存的处理例程的指针。共享内存的page fault处理代码在这个shmid_ds的页表条目列表中查找，看是

否存在这个共享虚拟内存页的条目。如果不存在，它就分配一个物理页，并为它创建一个页表条目。这个条目不但进入当前进程的页表，也存到这个shmid_ds。这意味着当下一个进程试图访问这块内存并得到一个page fault的时候，共享内存错误处理代码也会让这个进程使用这个新创建的物理页。所以，是第一个访问共享内存页的进程使得这一页被创建，而随后访问的其他进程使得此页被加到它们的虚拟地址空间。

当进程不再需要共享虚拟内存的时候，它们从中分离（detach）出来。只要仍旧有其他进程在使用这块内存，这种分离只是影响当前的进程。它的vm_area_struct从shmid_ds数据结构中删除，并释放。当前进程的页表也进行更新，使它共享过的虚拟内存区域无效。当共享这块内存的最后一个进程从中分离出的时候，共享内存当前在物理内存中的页被释放，这块共享内存的shmid_ds数据结构也被释放。

如果共享的虚拟内存没有被锁定在物理内存中的话会更加复杂。在这种情况下，共享内存的页可能在系统大量使用内存的时候交换到了系统的交换磁盘。共享内存如何交换初和交换入物理内存在第3章中有描述。

## Chapter 6

## PCI

Peripheral Component Interconnect（PCI），好像它的名字暗示的一样，是描述如何通过一个结构化和可控制的方式把系统中的外设组件连接起来的一个标准。标准的PCI Local Bus规范描述了系统组件电气连接的方法和它们行为的方法。本章探讨Linux核心如何初始化系统的PCI总线和设备。

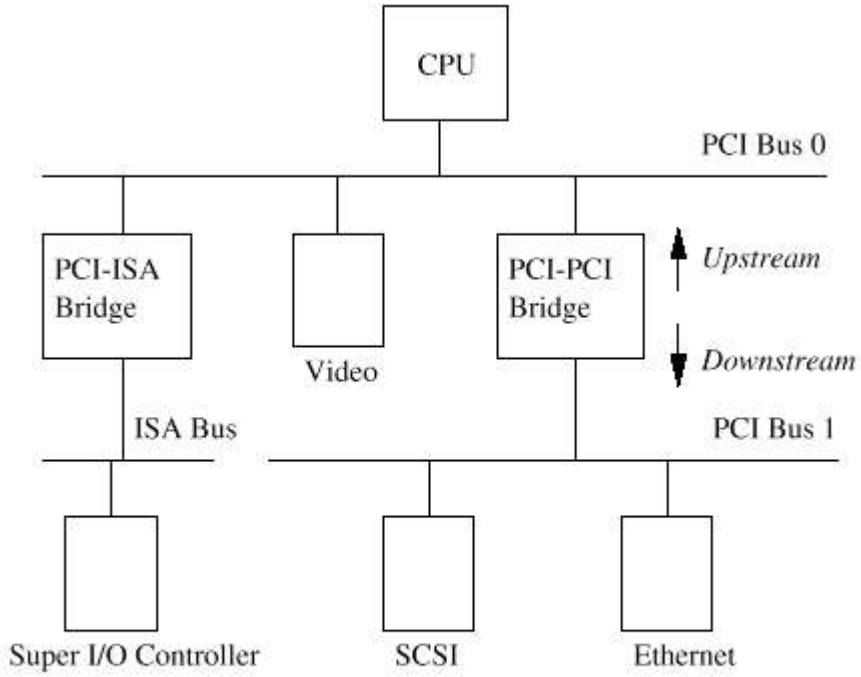



Figure 6.1: Example PCI Based System

图6.1是一个PCI基础的系统的逻辑图。PCI总线和PCI-PCI桥（bridge）是系统组件联系在一起的粘合剂。CUP和video设备连在主要的PCI总线，PCI总线0。一个特殊的PCI设备，PCI-PCI桥把主总线连接到次PCI总线，PCI总线1。按照PCI规范的术语，PCI总线1描述成为PCI-PCI桥的下游而PCI总线0是桥的上游。连接在次PCI总线上的是系统的SCSI和以太网设备。物理上桥、次要PCI总线和这两种设备可以在同一块PCI卡上。系统中的PCI-ISA桥支持老的、遗留的ISA设备，本图显示了一个超级I/O控制芯片，控制键盘、鼠标和软驱。

## 6.1 PCI Address Space（PCI地址空间）

CPU和PCI设备需要访问它们所共享的内存。这些内存让设备驱动程序控制这些PCI设备并在它们之间传递信息。一般地共享的内存包括设备的控制和状态寄存器。这些寄存器用于控制设备和读取它的状态。例如：PCI SCSI设备驱动程序可以读取SCSI设备的状态寄存器，判断它是否可以向SCSI磁盘写一块信息。或者它可以写入控制寄存器让它关闭的设备开始运行。

CPU的使用的系统内存可以用作这种共享内存，但是如果这样的话，每一次PCI设备访问内存，CPU都不得不停顿，等待PCI设备完成。对于内存的访问通常有限制，同一时间只能有一个系统组件允许访问。这会使得系统速度降低。允许系统的外部设备在一个不受控的方式下访问主内存也不是一个好主意。这会非常危险：一个恶意的设备会让系统非常不稳定。

外部设备由它们自己的内存空间。CPU可以访问这些空间，但是设备对于系统内存的访问受到严格的控制，必须通过DMA（Direct Memory Access直接内存存取）通道。ISA设备可以访问两

种地址空间：ISA I/O（输入/输出）和ISA内存。PCI由三中：PCI I/O、PCI内存和PCI配置空间（configuration space）。CPU可以访问所有的地址空间其中PCI I/O和PCI内存地址空间由设备驱动程序使用而PCI配置空间由Linux和心中的PCI初始化代码使用。

Alpha AXP处理器没有对于除了系统地址空间之外的地址空间的天生的访问模式。它需要使用支持芯片来访问象PCI配置空间这样的其他地址空间。它使用了一个地址空间的映射方案，从巨大的虚拟地址空间中偷出一部分映射到PCI地址空间。

## 6.2 PCI Configuration Headers（PCI配置头）

系统中的每一个PCI设备，包括PCI-PCI桥都由一个配置数据结构，位于PCI配置地址空间中。PCI配置头允许系统识别和控制设备。这个头位于PCI配置地址空间的确切位置依赖于设备使用的PCI拓扑。例如，插在PC主板一个PCI槽位的一个PCI显示卡配置头会在一个位置，而如果它被插到另一个PCI槽位则它的头会出现在PCI配置内存中的另一个位置。但是不管这些PCI设备和桥在什么位置，系统都可以发现并使用它们配置头中的状态和配置寄存器来配置它们。

通常，系统的设计使得每一个PCI槽位的PCI配置头都有一个和它在板上的槽位相关的偏移量。所以，举例来说，板上的第一个槽位的PCI配置可能位于偏移0而第二个槽位的在偏移256（所有的头都一样长度，256字节），依此类推。定义了系统相关的硬件机制使得PCI配置代码可以尝试检查一个给定的PCI总线上的所有可能的PCI配置头，试图读取头中的一个域（通常是Vendor Identification 域）得到一些错误，从而知道那些设备存在而那些设备不存在。PCI Local Bus规范描述了一种可能的错误信息：试图读取一个空的PCI槽位的Verdor Identification和Device Indentification域时候返回0xFFFFFFFF。

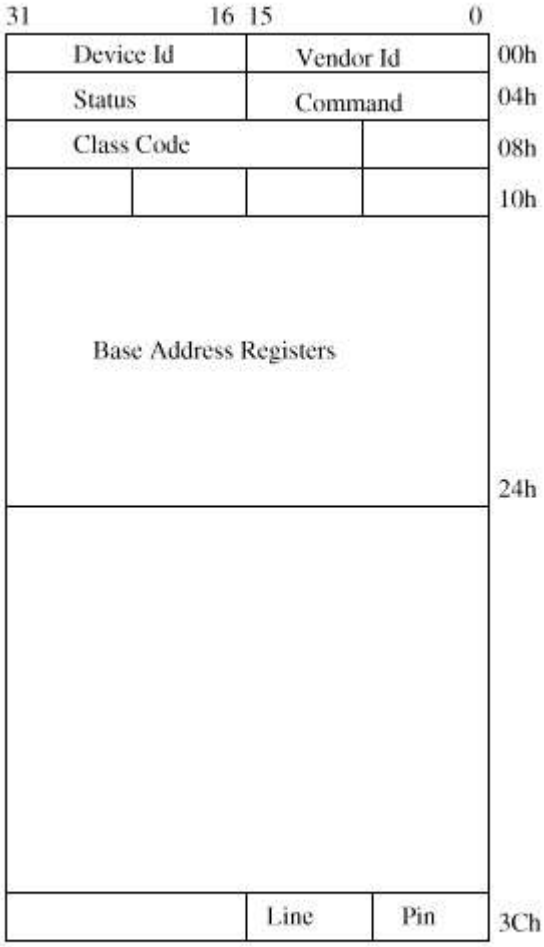



Figure 6.2: The PCI Configuration Header

图6.2显示了256字节的PCI配置头的布局。它包括以下域：

参见include/linux/pci.h

Vendor Identification 唯一的数字，描述这个PCI设备的发明者。Digital的PCI Vendor Identification 是0x1011而Intel是0x8086。

Device Identification 描述设备自身的唯一数字。例如Digital的21141快速以太网设备的设备标识符是0x0009。

Status 此域给除了设备的状态，它的位的含义由PCI Local Bus规范规定。

Command 系统通过写这个域控制这个设备。例如：允许设备访问PCI I/O内存。

Class Code 标识了设备的类型。对于每一种设备都有标准分类：显示、SCSI等等。对于SCSI的类型编码是0x0100。

Base Address Registers 这些寄存器用于确定和分配设备可以使用的PCI I/O和PCI内存的类型、大小和位置。

Interrupt Pin PCI卡的物理管脚中的4个用于向PCI总线传递中断。标准中把它们标记为A、B、C和D。Interrupt Pin域描述了这个PCI设备使用那个管脚。通常对于一个设备来说这时硬件决定的。就是说每一次系统启动的时候，这个设备都使用同一个中断管脚。这些信息允许中断处理子系统管理这些设备的中断。

Interrupt Line PCI配置头中的Interrupt Line域用于在PCI初始化代码、设备驱动程序和Linux的中断处理子系统之间传递中断控制。写在这里的数字对于设备驱动程序来讲是没有意义的，但是它可以让中断处理程序正确地把一个中断从PCI设备发送到Linux操作系统中正确的设备驱动程序的中断处理代码处。Linux如何处理中断参看第7章。

## 6.3 PCI I/O and PCI Memory Address（PCI I/O和PCI内存地址）

这两种地址空间用于设备和CPU上运行的Linux核心的它们的设备驱动程序通讯。例如：DECchip 21141快速以太网设备把它的内部寄存器映射到了PCI I/O空间。然后它的Linux设备驱动程序通过读写这些寄存器来控制设备。显示驱动程序通常使用大量的PCI内存空间来放置显示信息。

直到PCI系统建立起来并使用PCI配置头中的Command域打开了设备对于这些地址空间的访问为止，设备都无法访问这些空间。应该注意的是只有PCI配置代码读写PCI配置地址，Linux的设备驱动程序只是读写PCI I/O和PCI内存地址。

## 6.4 PCI-ISA Bridges（PCI-ISA桥）

这种桥把对于PCI I/O和PCI内存地址空间的访问转换成为ISA I/O和ISA内存访问，用来支持ISA设备。现在销售的多数系统都包括几个ISA总线插槽和几个PCI总线插槽。这种向后的兼容的需要会不断减少，将来会有只有PCI的系统。在早期的Intel 8080基础的PC时代，系统中的ISA设备的ISA 地址空间（I/O和内存）就被固定下来。甚至一个S5000 Alpha AXP基础的计算机系统的ISA软驱驱动器的ISA I/O地址也会和第一台IBM PC一样。PCI规范保留了PCI I/O和PCI内存的地址空间中的较低的区域保留给系统中的ISA外设并使用一个PCI-ISA桥把所有对于这些区域的PCI内存访问转换为ISA访问。

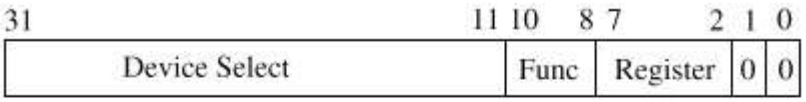



Figure 6.3: Type 0 PCI Configuration Cycle

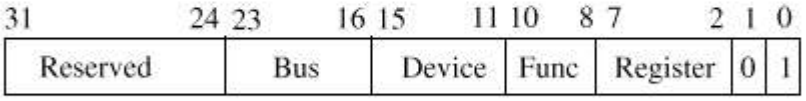



Figure 6.4: Type 1 PCI Configuration Cycle

## 6.5 PCI-PCI Bridges（PCI-PCI桥）

PCI-PCI桥是特殊的PCI设备，把系统中的PCI总线粘和在一起。简单系统中只有一个PCI总线，当时单个PCI总线可以支持的PCI设备的数量有电气限制。使用PCI-PCI桥增加更多的PCI总线允许系统支持更多的PCI设备。这对于高性能的服务器尤其重要。当然，Linux完全支持使用PCI-PCI桥的使用。

### 6.5.1 PCI-PCI Bridges: PCI I/O and PCI Memory Windows

PCI-PCI桥只向下游传递对于PCI I/O和PCI内存读和写的一个子集。例如在图6.1中，只有读和写的地址属于SCSI或者以太网设备的时候PCI-PCI桥才会把读写的地址从PCI总线0传递到总线1，其余的都被忽略。这种过滤阻止了不必要的地址信息遍历系统。为了达到这个目的，PCI-PCI桥必须编程设置它们必须从主总线向次总线通过的PCI I/O和PCI内存地址空间访问的基础（base）和限制。一旦系统中的PCI-PCI桥设置好，只要Linux设备驱动程序只是通过这些窗口存取PCI I/O和PCI内存空间，PCI-PCI桥是不可见的。这是个重要的特性，使得Linux的PCI设备驱动程序的作者的日子好过了。但是它也让Linux下的PCI-PCI桥在一定程度上需要技巧才能配置，我们不久就会看到。

### 6.5.2 PCI-PCI Bridges: PCI Configuration Cycles and PCI Bus Numbering（PCI-PCI桥：PCI配置cycle和PCI总线编号）

既然CPU的PCI初始化代码可以定位不在主PCI总线上的设备，必须有一种机制使得桥可以决定是否把配置cycle从它的主接口传递到次接口上。一个cycle就是它显示在PCI总线上的地址。PCI规范定义了两种PCI地址配置格式：类型0和类型1，分别在图6.3和图6.4中显示。类型0的PCI配置cycle不包含总线号，被这个PCI总线上的所有的PCI设备解释用于PCI地址配置。配置cycle的位

32：11看作是设备选择域。设计系统的一个方法是让每一个位选择一个不同的设备。这种情况下为11可能选择槽位0的PCI设备，位12选择槽位1的PCI设备，依此类推。另一种方法是把设备的槽位号直接写到位31：11中。一个系统使用哪一种机制依赖于系统的PCI内存控制器。

类型1的PCI配置cycle包括一个PCI总线号，这种配置循环被除了PCI-PCI桥之外的所有PCI设备忽略。所有看到了类型1的PCI配置cycle的PCI-PCI桥都可以把这些信息向它们的下游传送。一个PCI-PCI桥是否忽略PCI配置循环或者向它的下游传递，依赖于这个桥是如何配置的。每一个PCI-PCI桥都有一个主总线接口号和一个次总线接口号。主总线接口离CPU最近而次总线接口是离CPU最远的。每一个PCI-PCI桥都还有一个附属总线编号，这是在第二个总线接口之外可以桥接的最大的PCI总线数目。或者说，附属总线编号是PCI-PCI桥下游的最大的PCI总线编号。当PCI-PCI桥看到一个类型1的PCI配置cycle的时候，它做以下事情：

如果指定的总线编号不在桥的次总线编号和总线的附属编号之间就忽略它。

如果指定的总线编号和桥的次总线编号符合就把它转变成为类型0的配置命令

如果指定的总线编号大于次要总线编号而小于或等于附属总线编号，就不改变地传递到次要总线接口上。

所以，如果我们希望寻址图6.9的拓扑中总线3上的设备1，我们必须从CPU生成一个类型1的配置命令。桥1不改变地传递到总线1，桥2忽略它但是桥3把它转换成一个类型0的配置命令，并把它发送到总线3，使设备1响应它。

每一个独立的操作系统负责在PCI配置阶段分配总线编号，但是不管使用哪一种编码方案，对于系统中所有的PCI-PCI桥，以下陈述都必须是正确的：

所有位于一个PCI-PCI桥后面的PCI总线的编码都必须在次总线编号和附属总线编号之间（包含）

如果违背了这条规则，则PCI-PCI桥将无法正确地传递和转换类型1的PCI配置cycle，系统无法成功地找到并初始化系统中的PCI设备。为了完成编码方案，Linux按照特定的顺序配置这些特殊设备。参看6.6.2节对于Linux PCI桥和总线编码方案的描述以及一个可以工作的例子。

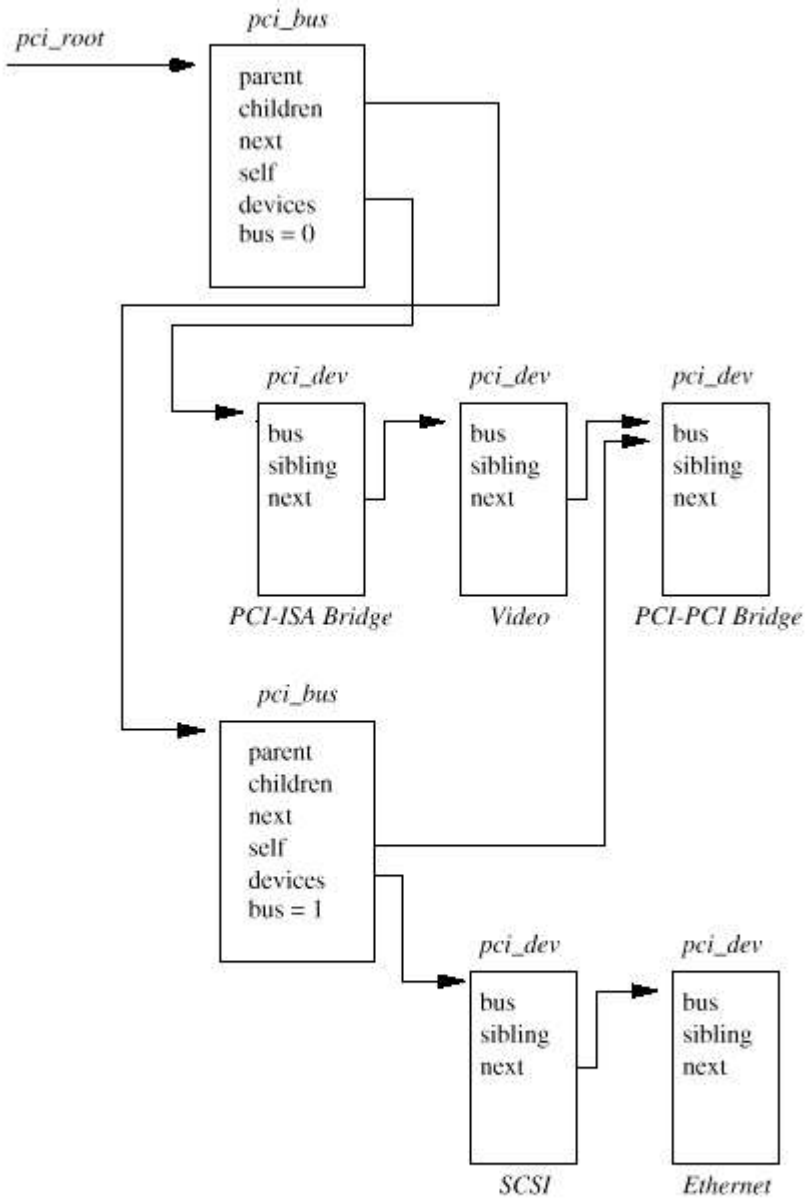



Figure 6.5: Linux Kernel PCI Data Structures

## 6.6 Linux PCI Initialization（Linux PCI初始化过程）

Linux中PCI初始化代码分为三个逻辑部分：

PCI Device Driver 这个伪设备驱动程序从总线0开始查找PCI系统，定位系统中所有的PCI设备和桥。它建立一链接的数据结构的列表，描述系统的拓扑。另外，它还为系统中所有的桥编码。

参见drivers/pci/pci.c and include/linux/pci.h

PCI BIOS 这个软件层提供了PCI BIOS ROM规范中描述的服务。即使Alpha AXP没有BIOS服务，在Linux核心也有提供了相同的功能的等价代码。

参见arch/*/kernel/bios32.c

PCI Fixup 系统相关的整理代码，整理和系统相关的在PCI初始化最后的内存疏松的情况。

参见arch/*/kernel/bios32.c

### 6.6.1 Linux Kernel PCI Data Structures（Linux核心的PCI数据结构）

当Linux核心初始化PCI系统的时候它建立反映系统真实的PCI拓扑结构的数据结构。图6.5显示了数据结构之间的关系，它用来描述了图6.1中示例的PCI系统。

每一个PCI设备（包括PCI-PCI桥）都用一个pci_dev的数据结构描述。每一个PCI总线用一个pci_bus的数据结构描述。结果是一个PCI总线的树型结构，每一个总线上有粘附着一些子PCI设备。因为一个PCI总线只能通过PCI-PCI桥达到（除了主PCI总线，总线0），每一个pci_bus都包括一个它要通过的PCI设备的指针（这个PCI-PCI桥）。这个PCI设备是这个PCI总线的父总线的一个子设备。

图6.5中没有显示的还有一个指向系统中所有的PCI设备的指针：pci_devices。系统中所有的PCI设备的pci_dev的数据结构都排在这个队列中。Linux核心使用这个队列快速查找系统中所有的PCI设备。

### 6.6.2 The PCI Device Driver（PCI 设备驱动程序）

PCI设备驱动程序完全不是一个真正的设备驱动程序，只是系统初始化的时候操作系统调用的一个函数。PCI初始化代码必须扫描系统中所有的PCI总线，查找系统中所有的PCI设备（包括PCI-PCI桥接设备）。它使用PCI BIOS代码来查看它当前扫描的PCI总线上的每一个可能的槽位是否被占用。如果这个PCI槽位占用，它就建立一个描述这个设备的pci_dev数据结构，并把它链接到已知PCI设备的列表中（由pci_deivices指向）。

参见drivers/pci/pci.c Scan_bus()

PCI初始化代码从PCI总线0开始扫描。它试图读出每一个可能的PCI槽位中每一个可能的PCI设备的Vendor Identification和Device Identification域。当它找到了占用的槽位它就建立一个pci_dev数据结构来描述它。PCI初始化代码所建立的所有的pci_dev数据结构（包括所有的PCI-PCI桥）都链接到一个链接表：pci_devices。

如果找到的设备是一个PCI-PCI桥，则建立一个pci_bus的数据结构，并链接到pci_root指向的由pci_bus和pci_dev数据结构组成的树上。PCI的初始代码可以判断PCI设备是否PCI-PCI桥，因为它的分类编码（class code）是0x060400。然后Linux核心配置它刚刚找到的PCI-PCI桥的另一端的PCI总线（下游）。如果找到更多的PCI-PCI桥，它们都一样被配置。这个过程成为深度（depthwize）算法：系统在宽度搜索之前先在深度展开。看图6.1，Linux会首先配置PCI总线1和它的以太网和SCSI设备，然后配置PCI总线0上的显示设备。

在Linux向下游查找PCI总线的时候它必须配置介入的PCI-PCI桥的次总线和附属总线编号。这些在下面的6.6.2节详细描述：

Configuring PCI-PCI Bridges – Assigning PCI Bus Numbers（配置PCI-PCI桥-分配PCI总线编号）

对于传送通过它们进行的PCI I/O、PCI内存或者PCI配置地址空间的读写，PCI-PCI桥必须直到以下：

Primary Bus Number 刚好在PCI-PCI桥上游的总线编号

Secondary Bus Number 刚好在PCI-PCI桥下游的总线编号

Subordinate Bus Number 从这个桥向下可以达到的所有总线中最高的总线编号。

PCI I/O and PCI Memory Windows 从这个PCI-PCI桥向下的所有的地址的PCI I/O地址空间和PCI 内存空间的窗口的base和size。

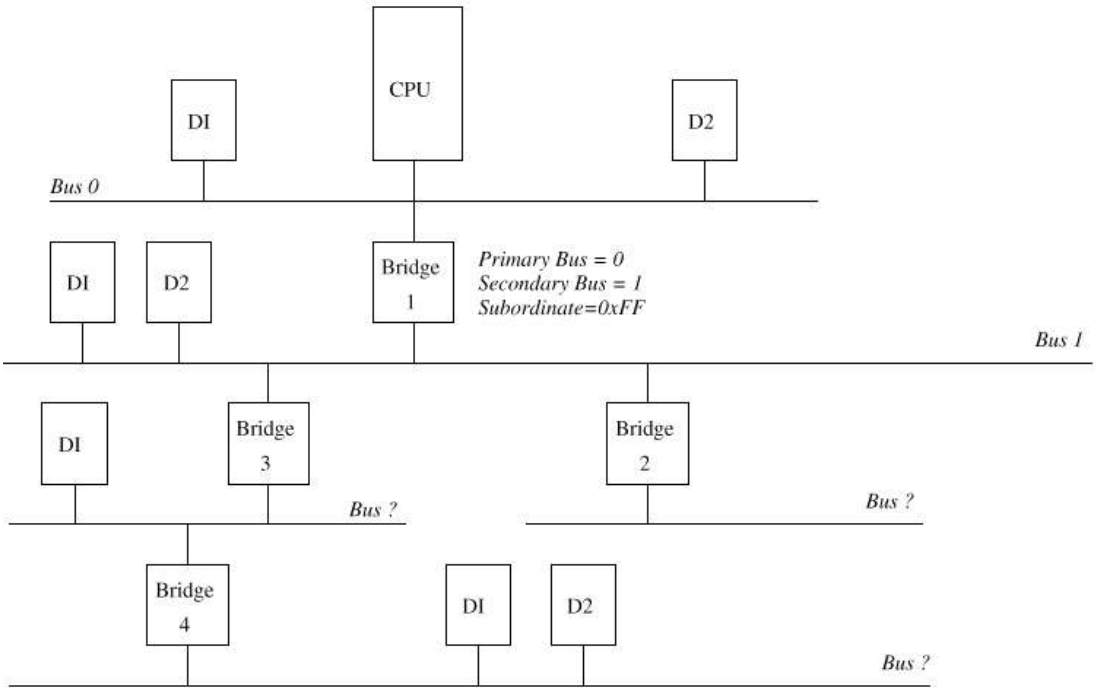



Figure 6.6: Configuring a PCI System: Part 1

问题是当你希望配置任何指定的PCI-PCI桥的时候你并不知道这个桥的附属总线数目。你不知道是否下游还有其他PCI-PCI桥。就算知道，你也不知道它们将会被分配什么编号。答案是使用一个深度递归算法（depthwise recursive algorithm）。在每一个总线上找到任何PCI-PCI桥的时候都就给它们分配编号。对于找到的每一个PCI-PCI桥，就给它的次总线分配编号，并给它分配临时的附属总线编号0xFF，并扫描它的下游所有的PCI-PCI桥并分配编号。这看起来相当复杂，但是下面的实际例子能使这个过程更清楚。

PCI-PCI Bridge Numbering： Step 1 参考图6.6中的拓扑，扫描找到的第一个桥是桥1（Bridge1）。桥1下游的PCI总线编号为1，桥1分配一个次总线号1和一个临时的附属总线编号0xFF。这意味着指定PCI总线1或更高的所用的类型1的PCI配置地址会穿过桥1到达PCI总线1。如果它们的总线编号是1，就转换成为类型0的配置cycle，否则对于其他的总线编号就不变。这也正是Linux PCI初始化代码需要做的，这样才能访问并扫描PCI总线1。

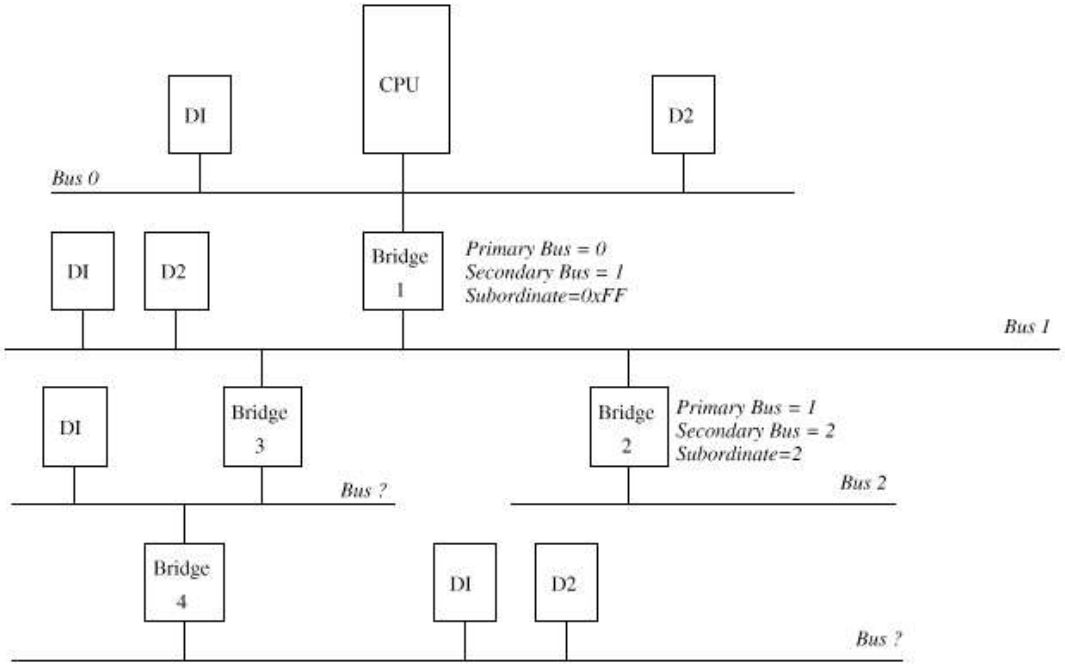



Figure 6.7: Configuring a PCI System: Part 2

PCI-PCI Bridge Numbering： Step 2 Linux使用深度算法，所以初始化代码开始扫描PCI总线1。这是它找到了PCI-PCI桥2，桥2之外没有其他的PCI-PCI桥，所以它的附属总线编号成为2，和它的次接口一样。图6.7显示了总线和PCI-PCI桥这时是如何编码的。

PCI-PCI Bridge Numbering：Step 3 PCI初始化代码回来扫描PCI总线1，找到了另一个PCI-PCI桥3。它的主总线接口赋值1而它的次总线接口是3，它的附属总线编号是0xFF。图6.8显示了系统这时是如何配置的。带有总线编号1、2或3的类型1的PCI配置cycle现在可以正确地传送到适当的PCI总线。

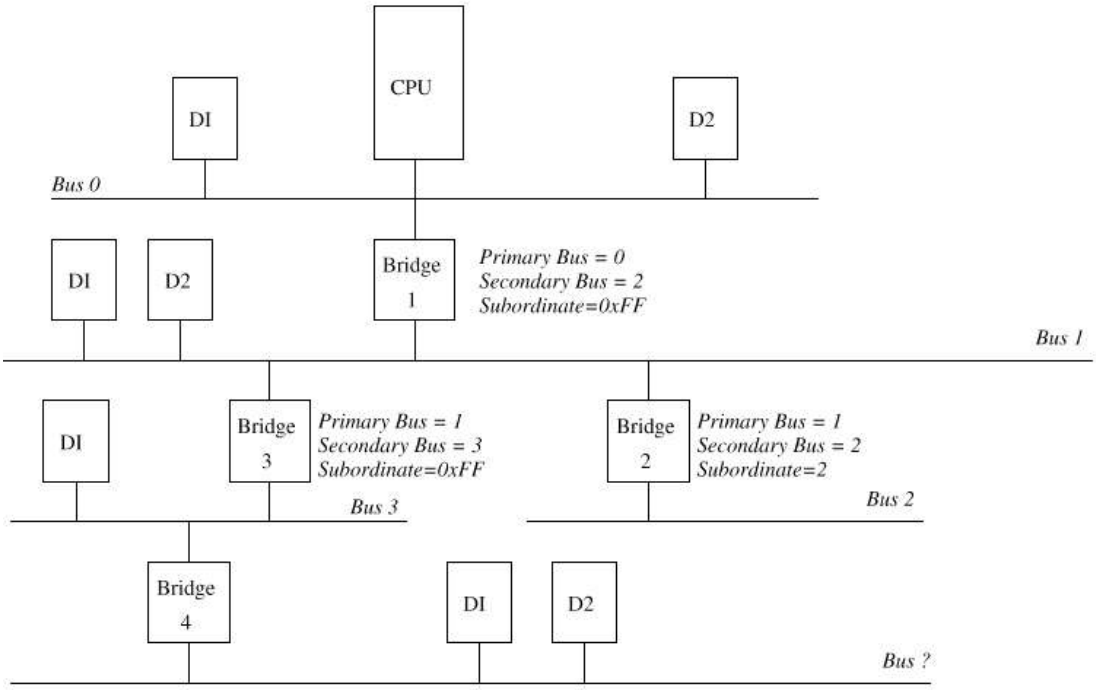



Figure 6.8: Configuring a PCI System: Part 3

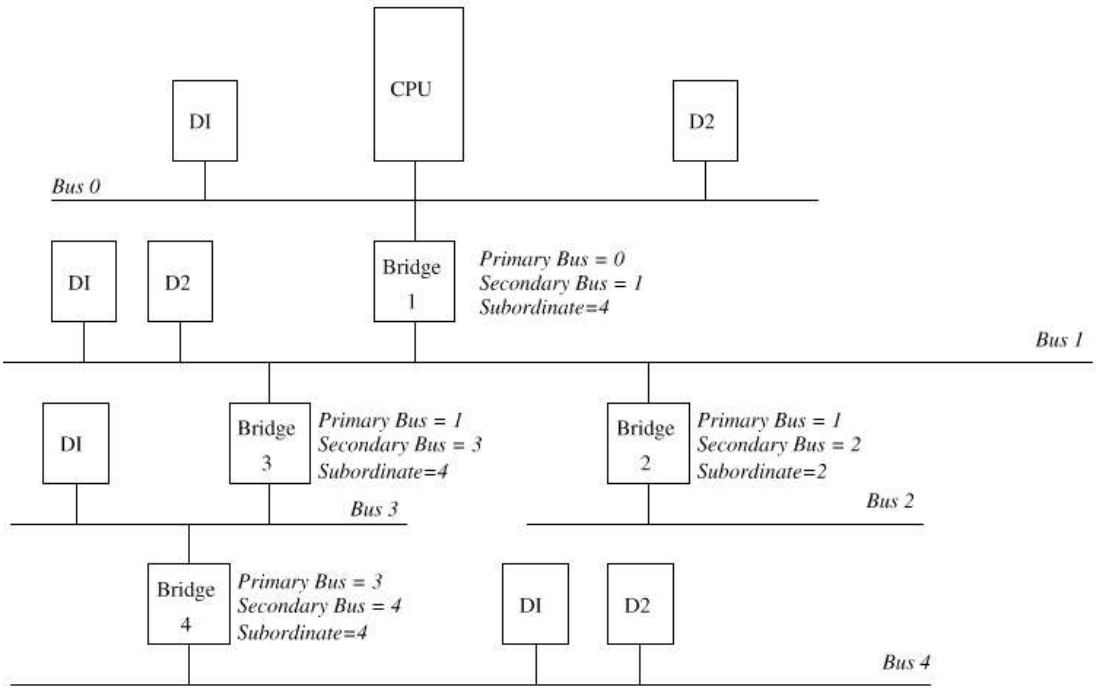



Figure 6.9: Configuring a PCI System: Part 4

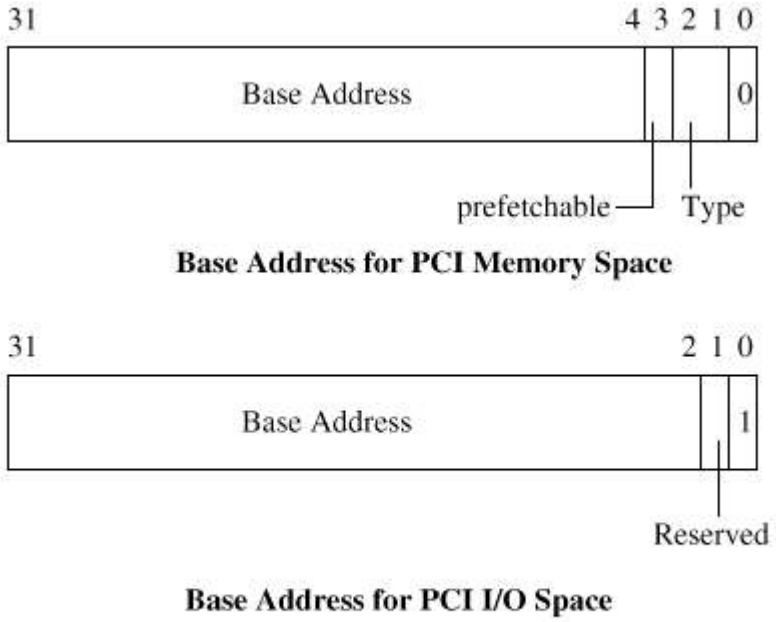



Figure 6.10: PCI Configuration Header: Base Address Registers

### 6.6.3 PCI BIOS Functions（PCI BIOS函数）

PCI BIOS函数是通用的跨平台的一系列标准例程。例如，它们对于Intel和Alpha AXP系统都是一样的。它们允许CPU控制对于所有PCI地址空间的访问。只有Linux核心和设备驱动程序需要使用它们。

参见arch/*/kernel/bios32.c

### 6.6.4 PCI Fixup

Alpha AXP系统上的PCI整理代码比Intel（基本不做任何事情）要做更多的工作。对于Intel系统，启动时候运行的系统BIOS，已经完全配置了PCI系统。Linux不需要做更多的事情，只是映射PCI的配置。对于非Intel系统，需要做更多的配置：

参见arch/kernel/bios32.c

对于每一个设备分配PCI I/O和PCI内存空间

对于系统重的每一个PCI-PCI桥必须配置PCI I/O和PCI内存地址窗口

对于设备产生Interrupt Line值，这些控制设备的中断处理

下面描述这些代码如何工作。

## Finding Out How Much PCI I/O and PCI Memory Space a Device Needs

（找出一个设备需要多少PCI I/O和PCI内存空间）

查询找到的每一个PCI设备，找出它需要多少PCI I/O和内存地址空间。为此，把每一个Base Address Register都写成1然后读出来。设备会在不关心的地址位返回1，有效地指定了需要的地址空间。

用两个基本的基础地址寄存器（Base Address Register），第一种指示设备的寄存器以及PCI I/O和PCI 内存空间必须在哪一个地址空间。这通过寄存器的0位表示。图6.10显示了PCI 内存和PCI I/O的基础地址寄存器的两种形式。

为了找出每一个给定的基础地址寄存器需要多少地址空间，需要向所有的寄存器写并读出来。设备会把不关心的地址位设为0，这样就有效地指明了需要的地址空间。这种设计暗示了使用的所有的地址空间都是2的指数，本质上是对齐的。

例如，在你初始化DECChip 21142 PCI快速以太网设备的时候，它告诉你在PCI I/O或PCI内存空间它需要0x100字节的地址。初始化代码为它分配空间。在它分配空间之后，21142的控制和状态寄存器就可以在这些地址见到。

## Allocating PCI I/O and PCI Memory to PCI-PCI Bridges and Devices

（为PCI-PCI桥和设备分配PCI I/O和PCI内存）

象所有的内存一样，PCI I/O和PCI内存空间是有限的，其中有一些相当紧缺。对于非Intel系统的PCI整理代码（和Intel系统的BIOS代码）必须有效地为每一个设备分配它需要的内存量。分配给一个设备的PCI I/O和PCI内存的分配必须自然对齐。例如，如果一个设备请求PCI I/O地址0xB0，那么分配的地址就必须是0xB0的倍数。另外，分配给任何桥的PCI I/O和PCI内存地址的基础必须分别对齐4K和1M的边界。下游的设备给定的地址空间必须位于它所有的上游的PCI-PCI桥的内存范围中间。所以有效地分配地址空间是有比较困难的问题。

Linux使用的算法依赖于用PCI设备驱动程序建立的总线/设备树所描述的每一个设备，它按照PCI I/O内存递增的顺序分配地址空间。又是使用递归算法，遍历PCI初始化代码所建立的pci_bus和pci_dev数据结构。BIOS整理代码从PCI总线的根（pci_root所指）开始：

分别把当前的全局PCI I/O和内存的基础分别对齐在4K和1M的边界

对于当前总线上的每一个设备（按照需要的PCI I/O内存顺序排列）

-分配它的PCI I/O和/或PCI内存

-将全局的PCI I/O和内存的基础按照合适的量移动

-允许设备使用给定的PCI I/O和PCI内存

分别为当前总线下游的所有总线分配空间，注意这会改变全局的PCI I/O和内存基础。

分别把当前的全局PCI I/O和内存的基础对齐在4K和1M的边界，同时指出当前的PCI-PCI桥所需要的PCI I/O和PCI内存的窗口的基础和大小

对于连接在当前总线上的PCI-PCI桥，设置它的PCI-PCI I/O和PCI内存地址和限制。

打开PCI-PCI桥上桥接PCI I/O和PCI内存访问的功能。这意味着如果任何在桥的主PCI总线上看到的PCI I/O和PCI内存地址如果位于它的PCI I/O和PCI内存地址窗口的话就会被桥接到它的次总线。

以图6.1的PCI系统作为PCI整理代码的例子：

Align the PCI base （初始的）PCI I/O是0x4000，PCI内存是0x100000。这样允许PCI-ISA桥把所有低于此的地址都转换到ISA地址。

The Video Device 请求0x200000的PCI内存，因为必须按照要求的大小对齐，所以我们从PCI内存0x200000开始分配，PCI内存基础地址移到了0x400000，而PCI I/O地址仍旧是0x4000。

The PCI-PCI Bridges 我们现在穿过PCI-PCI桥，在那里分配内存。注意因为我们不需要对其基础地址因为它们已经正确对齐了。

The Ethernet Device 它在PCI I/O和PCI内存空间都请求0xB0字节。它被分配在PCI I/O地址

0x4000，PCI内存0x400000。PCI内存的基础移到了0x4000B0，PCI I/O的基础成为0x40B0。

The SCSI Device 它请求0x1000的PCI内存，所以它在对齐之后分配在0x401000。而PCI I/O的基础地址还是0x40B0，PCI内存的基础移到了0x402000。

The PCI-PCI Bridge’s PCI I/O and Memory Windows 我们现在返回到桥，把它的PCI I/O窗口设置成为0x4000和0x40B0之间，它的PCI内存窗口在0x400000和0x402000之间。这意味着PCI-PCI桥会忽略对于显示设备的PCI内存访问，如果是对以太网或者SCSI设备的访问就可以通过。

## Chapter 7 Interrupts and Interrupt Handling（中断和中断处理）

本章探讨Linux核心如何处理中断。虽然核心有用于处理中断的通用机制和接口，大部分中断处理的细节还是和体系结构相关的。

Linux使用大量不同的硬件来完成许多不同的任务。显示设备驱动显示器，IDE设备驱动磁盘等等。你可以同步地驱动这些设备，就是你可以发出一个请求执行一些操作（比如把一块内存写到磁盘）然后等待操作结束。这种方式，虽然可以工作，但是非常没有效率，操作系统当它等待每一个操作完成的时候会花费大量时间“忙于什么也不做”（busy doing nothing）。一个好的，更有效的方法是做出了请求然后去作其他更有用的事情，然后当设备完成请求的时候被设备中断。在这种方案下，系统中同一时刻可能有许多设备的请求在同时发生。

让设备中断CPU当前的工作必须有一些硬件的支持。大多数，如果不是所有的话，通用目的的处理器比如Alpha AXP都使用相似的方法。CPU的一些物理管脚的电路只要改变电压（例如从+5V到-5V）就会让CPU停止正在做的工作，开始执行处理中断的特殊代码：中断处理代码。这些管脚之一可能连接一个内部适中，每一个1000分之一秒就接收一个中断，其他的也许连接到系统的其他设备，比如SCSI控制器。

系统通常使用一个中断控制器把设备的中断集合在一起，然后把信号传送到CPU的一个单一的中断管脚。这可以节省CPU的中断管教，也给设计系统带来了灵活性。中断控制器有掩码和状态寄存器，用于控制这些中断。设置掩码寄存器的位可以允许和禁止中断，状态寄存器返回系统中当前的中断。

一些系统中的中断可能是硬连接的，例如实时时钟的内部时钟可能永久地连接到中断控制器的第3管脚。但是，另一些管脚连接什么可能由在特定的ISA或者PCI槽位插入什么控制卡决定。例如，中断控制器的第4管脚可能和PCI槽位0相连，可能某一天有一个以太网卡，当时后来可能是一块

SCSI控制卡。每一个系统都有它自己的中断中转机制，操作系统必须足够灵活才能处理。

大多数现代的通用目的微处理器用相同的方式处理中断。发生硬件中断的时候，CPU停止它正在运行的指令，跳到内存中一个位置运行，这里或者包含中断处理代码或者是跳到中断处理代码的指令。这种代码通常在CPU的特殊模式下工作：中断模式，通常，这种模式下其他中断不能产生。这里也有例外：一些CPU将中断划分级别，更高级别的中断可以发生。这意味着写第一级的中断处理程序必须非常小心。中断处理程序通常都有自己的堆栈，用来存放CPU的执行状态（CPU所有的通用寄存器和上下文）并处理中断。一些CPU有一组只在中断模式下存在的寄存器，中断处理代码可以使用这些寄存器来存储它需要保存的大部分上下文信息。

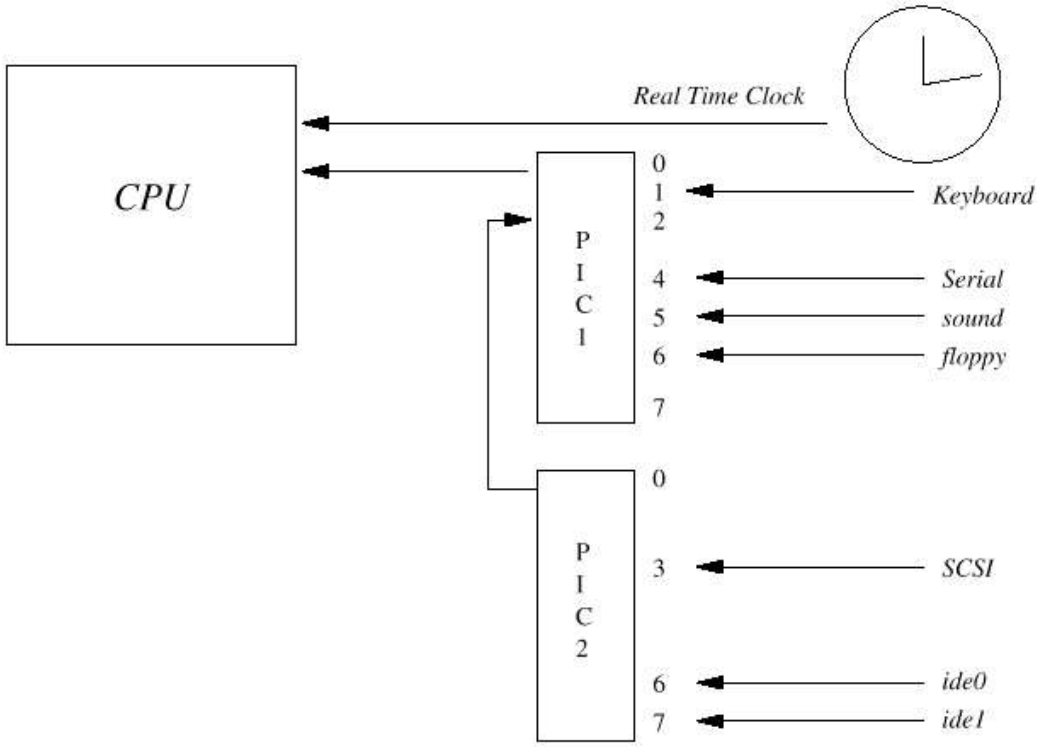



Figure 7.1: A Logical Diagram of Interrupt Routing

当处理完中断，CPU的状态恢复，中断结束。CPU会继续做它在中断发生之前做的事情。重要的事中断处理程序必须尽可能地有效，通常操作系统不能经常或者长时间阻塞中断。

## 7.1 Programmable Interrupt Controllers（可编程中断控制器）

系统设计师可以任意使用他们希望用的中断体系结构，但是IBM PC都使用Intel 82C59A-2 CMOS可编程中断控制器或者它的衍生物。这种控制器在PC最初的时候就使用了。它可通过寄存器编程，这些寄存器在ISA地址空间的众所周知的位置。甚至很现代的逻辑芯片组都在ISA内存的相同位置保留了等价的寄存器。非Intel的系统，例如Alpha AXP PC不受这些体系限制，通常使用不同的中断控制器。

图7.1显示了两个串联在一起的8位控制器：每一个都有一个掩码和一个中断状态寄存器，PIC1和PIC2。掩码寄存器位于地址0x21和0xA1，而状态寄存器位于0x20和0xA0。在掩码寄存器的一个特殊位写1允许一种中断，写0可以禁止它。所以向位3写1允许中断3，写0会禁止它。不幸的是（也是让人气恼的），中断掩码寄存器只可以写，你无法读回你所写的值。这意味着Linux必须为它设置的掩码（mask）寄存器保留一份本地拷贝。它在中断允许和禁止的例程中修改这些保存的掩码，每一次都要把整个掩码写到寄存器中。

当产生中断信号，中断处理程序读取两个中断状态寄存器（ISR）。它把0x20的ISR看作16位的中断寄存器的第8位，0xA0中的ISR看作高8位。所以，发生在0xA0的ISR的第1位的中断被看作是中断9。PCI1的第2位不可用，因为它用作串联PIC2的中断，任何PIC2的中断都会使PIC1的第2位置位。

## 7.2 Initializing the Interrupt Handling Data Structures（初始化中断处理数据结构）

当设备驱动程序要求控制系统的中断的时候建立核心的中断处理数据结构。为此，设备驱动程序使用一系列Linux核心服务，用来请求一个中断、允许它和禁止它。这些设备驱动程序调用这些例程来登记它们的中断处理例程的地址。

参见arch/*/kernel/irq.c request_irq() enable_irq() and disable_irq()

PC体系结构为了方便把一些中断固定下来，所以驱动程序在初始化的时候只需要简单地请求它的中断。软盘设备驱动程序就是这样：它总是请求中断6。但是也可能一个设备驱动程序不知道设备会使用什么中断。对于PCI设备驱动程序这不是问题，因为它们总是知道它们的中断编号。不幸的是对于ISA设备没有什么简单的办法找到它们的中断号码，Linux允许设备驱动程序探查它们的中断来解决这个问题。

首先，设备驱动程序让设备产生中断，然后系统中所有没有分配的中断都允许了。这意味着设备等待处理的中断现在会通过可编程中断控制器传递。Linux读取中断状态寄存器然后把它的内容返回到设备驱动程序。非0 的结果表示在探查中发生了一或多个中断。驱动程序现在关闭探查，并禁止所有位分配的中断。如果ISA设备驱动程序成功地找到了它的IRQ号，它就可以想平常一样地请求控制它。

参见arch/*/kernel/irq.c irq_probe_*()

PCI系统比ISA系统更加动态。ISA设备的中断通常用硬件设备上的跳线来设置，对于设备驱动程序是固定的。反过来，PCI设备的中断是在系统启动的时候由PCI BIOS或者PCI子系统在PCI初始化的时候分配的。每一个PCI设备可以使用4个中断管脚其中之一：A、B、C或D。这时设备制造的时候确定的，大多数设备缺省用中断管脚A。每一个PCI槽位的PCI中断线（interrupte

line）A、B、C和D都转到中断控制器。所以槽位4的管脚A可能转到了中断控制器的第6管脚，槽位4的管脚B可能转到了中断控制器的管脚7，依此类推。

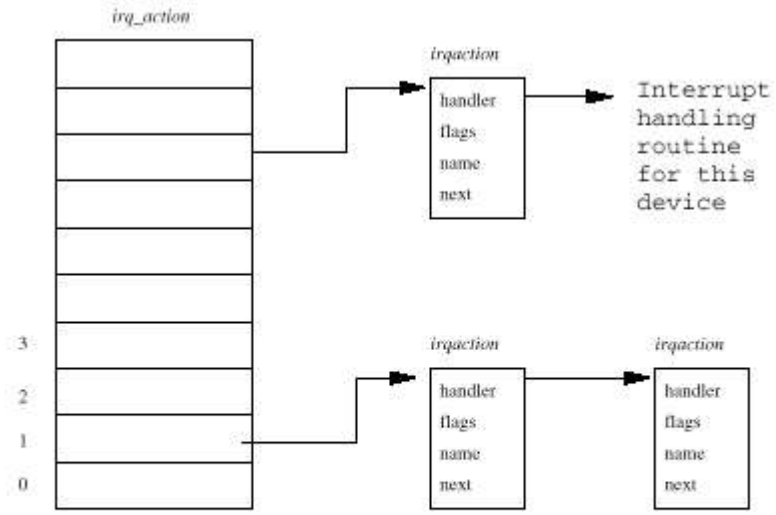



Figure 7.2: Linux Interrupt Handling Data Structures

PCI中断如何被转发（路由route）完全是和系统相关的，必须有一些理解这种PCI中断路由拓扑的设置代码。在Intel PC上，这是启动的时候的系统BIOS代码完成的。但是对于没有BIOS的系统（例如Alpha AXP系统），Linux进行这种设置。PCI设置代码把中断控制器的管脚编号写到每一个设备的PCI配置头中。它使用它知道的PCI中断路有拓扑和设备的PCI槽位以及它正在使用的PCI中断管脚来决定中断管脚（或者说IRQ）编号。设备使用的中断管脚就确定下来并放到PCI配置头的一个域。它把这个信息写到中断线（interrupte line）域（这是为此目的保留的）。当设备驱动程序运行的时候，它读取这个信息，并使用它向Linux核心请求对中断的控制。

参见arch/alpha/kernel/bios32.c

系统中可能使用许多PCI中断资源。例如，当使用PCI-PCI桥的时候。中断来源的数目可能超过系统的可编程中断控制器的管脚数目。这种情况下，PCI设备可以共享中断：中断控制器上的一个管脚接收来自多于一个PCI设备的中断。Linux让第一个请求一个中断的源宣称（declare）它是否可以共享，这样来支持中断共享。共享中断结果是irq_action向量表中的一个条目可以指向几个irqaction的数据结构。当发生了一个共享的中断的时候，Linux会调用这个源的所有的中断处理程序。所有可以共享中断的设备驱动程序（都应该是PCI设备驱动程序）必须预备在没有中断服务的时候被调用。

## 7.3 Interrupt Handling（中断处理）

Linux中断处理子系统的一个主要任务是把中断转送到（route）正确的中断处理代码段。这种代码必须了解系统的中断拓扑。例如，如果软驱控制器在中断控制器的管脚6发生中断，它必须可以识别出中断是来自软驱，并把它转送到软驱设备驱动程序的中断处理程序代码。Linux使用一系列

数据结构的指针，包含了处理系统中断的例程的地址。这些例程属于系统中的设备的设备驱动程序，每一个设备驱动程序必须负责在驱动程序初始化的时候请求它想要的中断。图7.2显示了irq_action是一个指向irqaction数据结构的指针的向量表。每一个irqaction数据结构都包括了这个中断处理程序的信息，包括中断处理例程的地址。不同体系的中断数目和如何处理是不同的，通常，不同系统之间，Linux中断处理代码是和体系结构相关的。这意味着irq_action向量表的大小依赖于中断源的数目而不同。

当发生中断的时候，Linux必须首先通过读取系统的可编程中断控制器的状态寄存器确定它的来源。然后把这个来源转换成irq_action向量表中的偏移。例如，从软驱控制器来的中断控制器管脚6的中断会转为中断处理程序向量表中的第7个指针。如果发生的中断没有对应的中断处理程序，Linux核心会记录下一个错误，否则，它会调用这个中断源的所有的irqaction数据结构中的中断处理例程。

当Linux核心调用设备驱动程序的中断处理例程的时候，它必须有效地判断为什么被中断，并进行响应。为了找出中断的原因，设备驱动程序会读取中断设备的状态寄存器。设备可能回应：发生了一个错误或者完成了一个请求的操作。例如软驱控制器可能报告它已经把软驱的读磁头定位到了软盘正确的扇区。一旦确定了中断的原因，设备驱动程序可能还需要做更多的工作。如果是这样，Linux核心有机制允许延迟这个操作稍候进行。这可以避免让CPU在中断模式下花费太多时间。详细描述参见设备驱动程序章（第8章）

## Chapter 8

## Device Drivers（设备驱动程序）

操作系统其中一个目的就是向用户掩盖系统硬件设备的特殊性。例如，虚拟文件系统呈现了安装的文件系统的一个统一的试图，而和底层的物理设备无关。本章描述Linux核心是如何管理系统中的物理设备的。

CPU不是系统中唯一的智能设备，每一个物理设备都由它自己的硬件控制器。键盘、鼠标和串行口由SuperIO芯片控制，IDE磁盘由IDE控制器控制，SCSI磁盘由SCSI控制器控制，等等。每一个硬件控制器都由自己的控制和状态控制器（CSR），不同的设备之间是不同的。一个Adaptec 2940 SCSI控制器的CSR和NCR 810 SCSI控制器的完全不同。CSR用于启动和停止设备，初始化设备和诊断它的问题。管理这些硬件控制器的代码不是放在每一个应用程序里边，而是放在Linux核心。这些处理或者管理硬件控制器的软件脚做设备驱动程序。Linux核心的设备驱动程序本质上是特权的、驻留内存的低级的硬件控制例程的共享库。是Linux的设备驱动程序在处理它们管理的设备的特质。

UN*X的一个基本特点是它抽象了设备的处理。所有的硬件设备都象常规文件一样看待：它们可以使用和操作文件相同的、标准的系统调用来进行打开、关闭和读写。系统中的每一个设备都用一个设备特殊文件代表。例如系统中第一个IDE硬盘用/dev/had表示。对于块（磁盘）和字符设备，这些设备特殊文件用mknod命令创建，并使用主（major）和次（minor）设备编号来描述设备。网络设备也用设备特殊文件表达，但是它们由Linux在找到并初始化系统中的网络控制器的时候创建。同一个设备驱动程序控制的所有设备都由一个共同的major设备编号。次设备编号用于在不同的设备和它们的控制器之间进行区分。例如，主IDE磁盘的不同分区都由一个不同的次设备编号。所以，/dev/hda2，主IDE磁盘的第2个分区的主设备号是3，而次设备号是2。Linux使用主设备号表和一些系统表（例如字符设备表chrdevs）把系统调用中传递的设备特殊文件（比如在一个块设备上安装一个文件系统）映射到这个设备的设备驱动程序中。

参见fs/devices.c

Linux支持三类的硬件设备：字符、块和网络。字符设备直接读写，没有缓冲区，例如系统的串行端口/dev/cua0和/dev/cua1。块设备只能按照一个块（一般是512字节或者1024字节）的倍数进行读写。块设备通过buffer cache访问，可以随机存取，就是说，任何块都可以读写而不必考虑它在设备的什么地方。块设备可以通过它们的设备特殊文件访问，但是更常见的是通过文件系统进行访问。只有一个块设备可以支持一个安装的文件系统。网络设备通过BSD socket接口访问，网络子系统在网络章（第10章）描述。

Linux有许多不同的设备驱动程序（这也是Linux的力量之一）但是它们都具有一些一般的属性：

Kernel code 设备驱动程序和核心中的其他代码相似，是kenel的一部分，如果发生错误，可能严重损害系统。一个写错的驱动程序甚至可能摧毁系统，可能破坏文件系统，丢失数据。

Kenel interfaces 设备驱动程序必须向Linux核心或者它所在的子系统提供一个标准的接口。例如，终端驱动程序向Linux核心提供了一个文件I/O接口，而SCSI设备驱动程序向SCSI子系统提供了SCSI设备接口，接着，向核心提供了文件I/O和buffer cache的接口。

Kernel mechanisms and services 设备驱动程序使用标准的核心服务例如内存分配、中断转发和等待队列来完成工作

Loadable Linux大多数的设备驱动程序可以在需要的时候作为核心模块加载，在不再需要的时候卸载。这使得核心对于系统资源非常具有适应性和效率。

Configurable Linux设备驱动程序可以建立在核心。哪些设备建立到核心在核心编译的时候是可以配置的。

Dynamic 在系统启动，每一个设备启动程序初始化的时候它查找它管理的硬件设备。如果一个设备驱动程序所控制的设备不存在并没有关系。这时这个设备驱动程序只是多余的，占用很少的系统内存，而不会产生危害。

## 8.1 Poling and Interrupts（轮询和中断）

每一次给设备命令的时候，例如“把读磁头移到软盘的第42扇区“，设备驱动程序可以选择它如何判断命令是否执行结束。设备驱动程序可以轮询设备或者使用中断。

轮询设备通常意味着不断读取它的状态寄存器，直到设备的状态改变指示它已经完成了请求。因为设备驱动程序是核心的一部分，如果驱动程序一直在轮询，核心在设备完成请求之前不能运行其他任何东西，会是损失惨重的。所以轮询的设备驱动程序使用一个系统计时器，让系统在晚些时候调用设备驱动程序中的一个例程。这个定时器例程会检查命令的状态，Linux的软盘驱动程序就是这样工作的。使用计时器进行轮询是一种最好的接近，而更加有效的方法是使用中断。

中断设备驱动程序在它控制的硬件设备需要服务的时候会发出一个硬件中断。例如：一个以太网设备驱动程序会在设备在网络上接收到一个以太网报文的时候被中断。Linux核心需要有能力把中断从硬件设备转发到正确的设备驱动程序。这通过设备驱动程序向核心登记它所使用的中断来实现。它登记中断处理程序例程的地址和它希望拥有的中断编号。你通过/proc/interrupts可以看到设备驱动使用了哪些中断和每一类型的中断使用了多少次：

0: 727432 timer

1: 20534 keyboard

2: 0 cascade

3: 79691 + serial

4: 28258 + serial

5: 1 sound blaster

11: 20868 + aic7xxx

13: 1 math error

14: 247 + ide0

15: 170 + ide1

对于中断资源的请求发生在驱动程序初始化的时间。系统中的一些中断是固定的，这是IBM PC体系结构的遗留物。例如软驱磁盘控制器总是用中断6。其他中断，例如PCI设备的中断，在启动的时候动态分配。这时设备驱动程序必须首先找出它所控制的设备的中断号，然后才能请求拥有这个中断（的处理权）。对于PCI中断，Linux支持标准的PCI BIOS回调（callback）来确定系统中设备的信息，包括它们的IRQ。

一个中断本身是如何转发到CPU依赖于体系结构。但是在大多数的体系上，中断都用一种特殊的模式传递，而停止系统中发生其他中断。设备驱动程序在它的中断处理例程中应该做尽可能少的工作，使得Linux核心可以结束中断并返回到它中断之前的地方。收到中断后需要做大量工作的设备驱动程序可以使用核心的bottom half handler或者任务队列把例程排在后面，以便在以后调用。

## 8.2 Direct Memory Access （DMA）

当数据量比较少的时候用中断驱动的设备驱动程序向设备或者通过设备传输数据工作地相当好。例如，一个9600波特率的modem每一毫秒（1/1000秒）大约可以传输一个字符。如果中断延迟，就是从硬件设备发出中断到开始调用设备驱动程序中的中断处理程序所花的时间比较少（比如2毫秒），那么数据传输对系统整体的映像就非常小。9600波特率的modem数据传出只会占用0.002%的CPU处理时间。但是对于高速的设备，比如硬盘控制器或者以太网设备，数据传输速率相当高。一个SCSI设备每秒可以传输高达40M字节的信息。

直接内存存取，或者说DMA，就是发明来解决这个问题的。一个DMA控制器允许设备不需要处理器的干预而和系统内存创树数据。PC的ISA DMA控制器由8个DMA通道，其中7个可用于设备驱动程序。每一个DMA通道都关联一个16位的地址寄存器和一个16位的计数寄存器（count register）。为了初始化一次数据传输，设备驱动程序需要建立DMA通道的地址和计数寄存器，加上数据传输的方向，读或写。当传输结束的时候，设备中断PC。这样，传输发生的时候，CPU可以作其他事情。

使用DMA的时候设备驱动程序必须小心。首先，所有的DMA控制器都不了解虚拟内存，它只能访问系统中的物理内存。因此，需要进行DMA传输的内存必须是物理内存中连续的块。这意味着你不能对于一个进程的虚拟地址空间进行DMA访问。但是你可以在执行DMA操作的时候把进程的物理也锁定到内存中。第二：DMA控制器无法访问全部的物理内存。DMA通道的地址寄存器表示DMA地址的首16位，跟着的8位来自于页寄存器（page register）。这意味着DMA请求限制在底部的16M内存中。

DMA通道是稀少的资源，只有7个，又不能在设备驱动程序之间共享。象中断一样，设备驱动程序必须有能力发现它可以使用哪一个DMA通道。象中断一样，一些设备有固定的DMA通道。比如软驱设备，总是用DMA通道2。有时，设备的DMA通道可以用跳线设置：一些以太网设备用这种技术。一些更灵活的设备可以告诉它（通过它们的CSR）使用哪一个DMA通道，这时，设备驱动程序可以简单地找出一个可用的DMA通道。

Linux使用dma_chan数据结构向量表（每一个DMA通道一个）跟踪DMA通道的使用。Dma_chan数据结构只有两个玉：一个字符指针，描述这个DMA通道的属主，一个标志显示这个DMA通道是否被分配。当你cat /proc/dma的时候显示的就是dma_chan向量表。

## 8.3 Memory（内存）

设备驱动程序必须小心使用内存。因为它们是Linux核心的一部分，它们不能使用虚拟内存。每一次设备驱动程序运行的时候，可能是接收到了中断或者调度了一个buttom half handler或任务队列，当前的进程都可能改变。设备驱动程序不能依赖于一个正在运行的特殊进程。象核心中其他部分一样，设备驱动程序使用数据结构跟踪它控制的设备。这些数据结构可以在设备驱动程序的代码部分静态分配，但是这会让核心不必要地增大而浪费。多数设备驱动程序分配核心的、不分页的内存存放它们的数据。

Linux核心提供了核心的内存分配和释放例程，设备驱动程序正是使用了这些例程。核心内存按照2的幂数的块进行分配。例如128或512字节，即使设备驱动程序请求的数量没有这么多。设备驱动程序请求的字节数按照下一个块的大小取整。这使得核心的内存回收更容易，因为较小的空闲块可以组合成更大的块。

请求核心内存的时候Linux还需要做更多的附加工作。如果空闲内存的总数太少，物理页需要废弃或者写到交换设备。通常，Linux 会挂起请求者，把这个进程放到一个等待队列，直到有了足够的物理内存。不是所有的设备驱动程序（或者实际是Linux的核心代码）希望发生这样的事情，核心内存分配例程可以请求如果不能立刻分配内存就失败。如果设备驱动程序希望为DMA访问分配内存，它也需要指出这块内存是可以进行DMA的。因为需要让Linux核心明白系统中哪些是连续的可以进行DMA的内存，而不是让设备驱动程序决定。

## 8.4 Interfacing Device Drivers with the Kernel（设备驱动程序和核心接口）

Linux核心必须能够用标准的方式和它们作用。每一类的设备驱动程序：字符、块和网络，都提供了通用的接口供核心在需要请求它们的服务的时候使用。这些通用的接口意味着核心可以完全相同地看待通常是非常不同的设备和它们的设备驱动程序。例如，SCSI和IDE磁盘的行为非常不同，但是Linux核心对它们使用相同的接口。

Linux非常地动态，每一次Linux核心启动，它都可能遇到不同的物理设备从而需要不同的设备驱动程序。Linux允许你在核心建立的时间通过配置脚本包含设备驱动程序。当启动的时候这些设备驱动程序初始化，它们可能没发现它们可以控制的任何硬件。其他驱动程序可以在需要的时候作为核心模块加载。为了处理设备驱动程序的这种动态的特质，设备驱动程序在它们初始化的时候向核心登记。Linux维护已经登记的设备驱动程序列表，作为和它们接口的一部分。这些列表包括了例程的指针和支持这一类设备的接口的信息。

### 8.4.1 Character Devices（字符设备）

字符设备，Linux最简单的设备，象文件一样访问。应用程序使用标准系统调用打开、读取、写和关闭，完全好像这个设备是一个普通文件一样。甚至连接一个Linux系统上网的PPP守护进程使用的modem，也是这样的。当字符设备初始化的时候，它的设备驱动程序向Linux核心登记，在chrdevs向量表增加一个device_struct数据结构条目。这个设备的主设备标识符（例如对于tty设备是4），用作这个向量表的索引。一个设备的主设备标识符是固定的。Chrdevs向量表中的每一个条目，一个device_struct数据结构，包括两个元素：一个登记的设备驱动程序的名称的指针和一个指向一组文件操作的指针。这块文件操作本身位于这个设备的字符设备驱动程序中，每一个都处理特定的文件操作比如打开、读、写和关闭。/proc/devices中字符设备的内容来自chrdevs向量表

参见include/linux/major.h

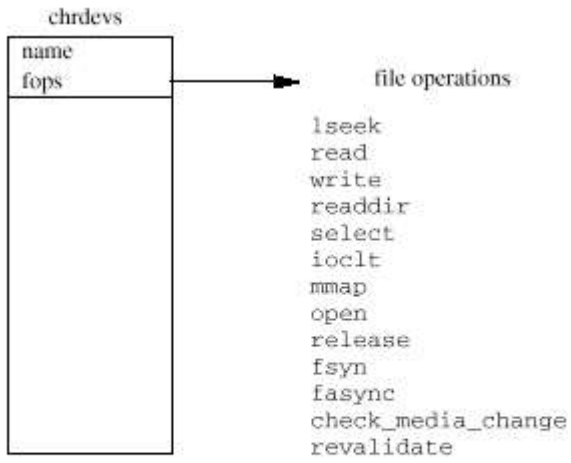



Figure 8.1: Character Devices

当代表一个字符设备（例如/dev/cua0）的字符特殊文件打开，核心必须做一些事情，从而去掉用正确的字符设备驱动程序的文件操作例程。和普通文件或目录一样，每一个设备特殊文件都用VFS I节点表达。这个字符特殊文件的VFS inode（实际上所有的设备特殊文件）都包括设备的major和minor标识符。这个VFS I节点由底层的文件系统（例如EXT2），在查找这个设备特殊文件的时候根据实际的文件系统创建。

参见fs/ext2/inode.c ext2_read_inode()

每一个VFS I节点都联系着一组文件操作，依赖于I节点所代表的文件系统对象不同而不同。不管代表一个字符特殊文件的VFS I节点什么时候创建，它的文件操作被设置成字符设备的缺省操作。这只有一种文件操作：open操作。当一个应用程序打开这个字符特殊文件的时候，通用的open文件操作使用设备的主设备标识符作为chrdevs向量表中的索引，取出这种特殊设备的文件操作块。它也建立描述这个字符特殊文件的file数据结构，让它的文件操作指向设备驱动程序中的操作。然后应用程序所有的文件系统操作都被映射到字符设备的文件操作。

参见fs/devices.c chrdev_open() def_chr_fops

## 8.4.2 Block Devices（块设备）

块设备也支持象文件一样被访问。这种为打开的块特殊文件提供正确的文件操作组的机制和字符设备的十分相似。Linux用blkdevs向量表维护已经登记的块设备文件。它象chrdevs向量表一样，使用设备的主设备号作为索引。它的条目也是device_struct数据结构。和字符设备不同，块设备进行分类。SCSI是其中一类，而IDE是另一类。类向Linux核心登记并向核心提供文件操作。一种块设备类的设备驱动程序向这种类提供和类相关的接口。例如，SCSI设备驱动程序必须向SCSI子系统提供接口，让SCSI子系统用来对核心提供这种设备的文件操作

参见fs/devices.c

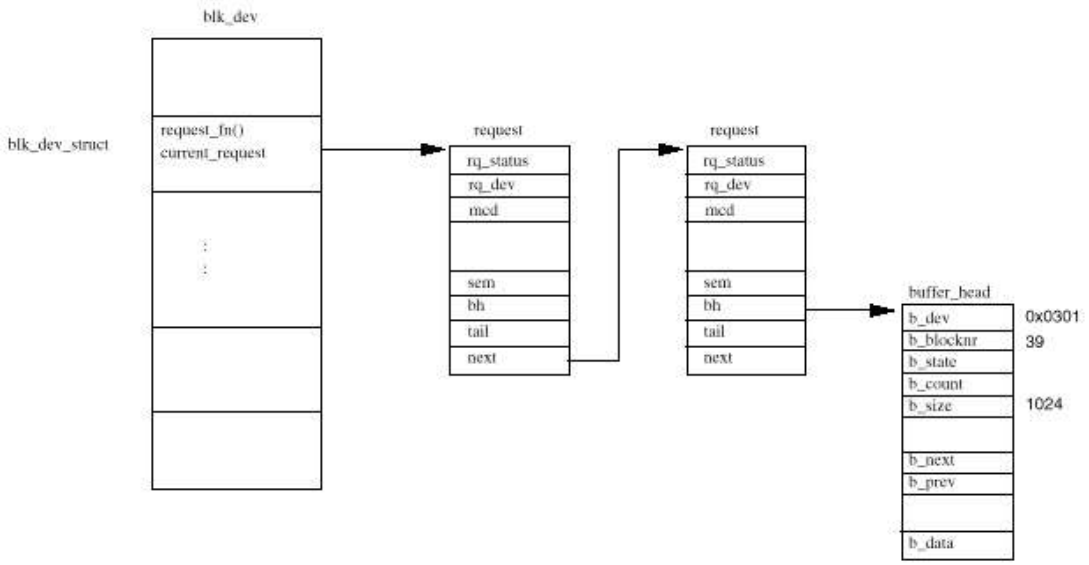



Figure 8.2: Buffer Cache Block Device Requests

每一个块设备驱动程序必须提供普通的文件操作接口和对于buffer cache的接口。每一个块设备驱动程序填充blk_dev向量表中它的blk_dev_struct数据结构。这个向量表的索引还是设备的主设备号。这个blk_dev_struct数据结构包括一个请求例程的地址和一个指针，指向一个request数据结构的列表，每一个都表达buffer cache向设备读写一块数据的一个请求。

参见drivers/block/ll_rw_blk.c include/linux/blkdev.h

每一次buffer cache希望读写一块数据到或从一个登记的设备的时候它就在它的blk_dev_struc中增加一个request数据结构。图8.2显示了每一个request都有一个指针指向一个或多个buffer_head数据结构，每一个都是一个读写一块数据的请求。这个buffer_head数据结构被锁定（buffer cache），可能会有一个进程在等待这个缓冲区的阻塞进程完成。每一个request结构都是从一个静态表，all_request表中分配的。如果这个request增加到一个空的request列表，就调用驱动程序的request函数处理这个request队列。否则，驱动程序只是简单地处理request队列中的每一个请求。

一旦设备驱动程序完成了一个请求，它必须把每一个buffer_head结构从request结构中删除，标记它们为最新的，然后解锁。对于buffer_head的解锁会唤醒任何正在等待这个阻塞操作完成的进程。这样的例子包括文件解析的时候：必须等待EXT2文件系统从包括这个文件系统的块设备上读取包括下一个EXT2目录条目的数据块，这个进程将会在将要包括目录条目的buff_head队列中睡眠，直到设备驱动程序唤醒它。这个request数据结构会被标记为空闲，可以被另一个块请求使用。

## 8.5 Hard Disks（硬盘）

硬盘把数据存放在转动的磁碟上，提供了一个更永久存储数据的方式。为了写入数据，微小的磁头把磁碟表面的一个微小的点磁化。通过磁头可以探测指定的微粒是否被磁化，从而可以读出数据。

一个磁盘驱动器由一个或多个磁碟组成，每一个都用相当光滑的玻璃或者陶瓷制成，并覆盖上一层精细的金属氧化物。磁碟放在一个中心轴上面，并按照稳定的速度转动。转动速度根据型号不同从3000到1000RPM（转/每分钟）。磁盘的读/写磁头负责读写数据，每一个磁碟有一对，每一面一个。读/写磁头和磁碟表面并没有物理的接触，而是在一个很薄的空气垫（十万分之一英寸）上面漂浮。读写磁头通过一个驱动器在磁碟表面移动。所有的磁头都粘在一起，一起在磁碟表面移动。

每一个磁碟的表面都分成多个狭窄的同心环，叫做磁道（track）。磁道0是最外面的磁道，最高编号的磁道是最接近中心轴的磁道。一个柱面（cylinder）是相同编号磁道的组合。所以每一个磁碟的每一面的所有的第5磁道就是第5柱面。因为柱面数和磁道数相同，所以磁盘的尺寸常用柱面来描述。每一个磁道分成扇区。一个扇区是可以从硬盘读写的最小数据单元，也就是磁盘的块大小。通常扇区大小是512字节，扇区大小通常是在制造磁盘的时候进行格式化的时候设定的。

磁盘通常用它的尺寸（geometry）描述：柱面数、磁头数和扇区数。例如，启动的时候Linux这样描述我的IDE磁盘：

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

这意味着它由1050柱面（磁道），16头（8个磁碟）和63个扇区/磁道。对于512字节的扇区或块大小，磁盘的容量是529200K字节。这和磁盘声明的516M的存储能力不符合，因为一些扇区用作存储磁盘的分区信息。一些磁盘可以自动找出坏的扇区，对其进行重新索引。

硬盘可以再分为分区。一个分区是分配用于特定目的的一大组扇区。对磁盘分区允许磁盘用于几个操作系统或多个目的。大多数单个磁盘的Linux系统都由3个分区：一个包含DOS文件系统，另一个是EXT2文件系统，第三个是交换分区。硬盘的分区用分区表描述，每一个条目用磁头、扇区和柱面号描述分区的起止位置。对于用fdisk格式化的DOS磁盘，可以有4个主磁盘分区。不是分区表所有的4个条目都必须用到。Fdisk支持三种类型的分区：主分区、扩展分区和逻辑分区。扩展分区不是真正的分区，它可以包括任意数目的逻辑分区。发明扩展分区和逻辑分区是为了突破4个主分区的限制。下面是一个包括2个主分区的磁盘的fdisk的输出：

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

它显示了第一个分区开始于柱面或磁道0，磁头1和扇区1，直到柱面477，扇区32和磁头63。因为一个磁道由32个扇区和64个读写磁头，这个分区的柱面都是完全包括的。Fdisk缺省把分区对齐在柱面的边界。它从最外面的柱面（0）开始向内，朝向中心轴，扩展478个柱面。第2个分区，交换分区，开始于下一个柱面（478）并扩展到磁盘最里面的柱面。

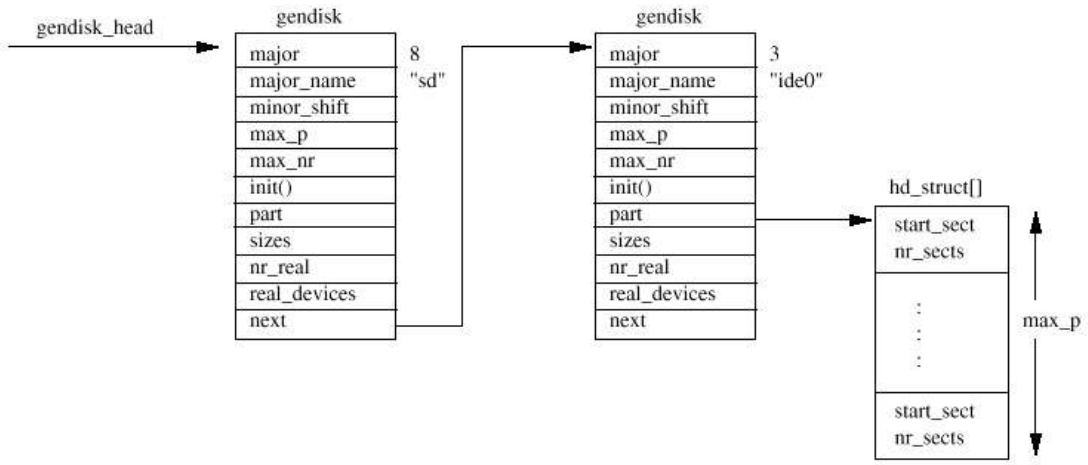



Figure 8.3: Linked list of disks

在初始化的时候Linux映射系统中的硬盘的拓扑结构。它找出系统中有多少个硬盘以及硬盘的类型。Linux还找出每一个磁盘如何分区。这些都是由gendisk_head指针列表指向的一组gendisk数据结构的列表表达。对于每一个磁盘子系统，例如IDE，初始化的时候生成gendisk数据结构表示它找到的磁盘。这个过程和它登记它的文件操作和在blk_dev数据结构中增加它的条目发生在同一时间。每一个gendisk数据结构都由一个唯一的主设备号，和块特殊设备的相同。例如，SCSI磁盘子系统会创建一个独立的gendisk条目（“sd”），主设备号是8（所有SCSI磁盘设备的主设备号）。图8.3显示了两个gendisk条目，第一个是SCSI磁盘子系统，第二个是IDE磁盘控制器。这里是ide0，主IDE控制器。

虽然磁盘子系统在初始化的时候会建立相应的gendisk条目，Linux只是在进行分区检查的时候才用到。每一个磁盘子系统必须维护自己的数据结构，让它自己可以把设备的主设备号和次设备号映射到物理磁盘的分区上。不管什么时候读写块设备，不管是通过buffer cache或者文件操作，核心都根据它在块特殊设备文件（例如/dev/sda2）中找到的主设备号和次设备号把操作定向到合适的设备。是每一个设备驱动程序或子系统把次设备号映射到真正的物理设备上。

### 8.5.1 IDE Disks（IDE磁盘）

今天Linux系统中最常用的磁盘是IDE磁盘（Integrated Disk Electronic）。IDE和SCSI一样是一个磁盘接口而不是一个I/O总线。每一个IDE控制器可以支持最多2个磁盘，一个是master，另一个是slave。Master和slave通常用磁盘上的跳线设置。系统中的第一个IDE控制器叫做主IDE控制器，下一个叫从属控制器等等。IDE可以从/向磁盘进行3.3M/秒的传输，IDE磁盘的最大尺寸是538M字节。扩展IDE或EIDE把最大磁盘尺寸增加到8.6G字节，数据传输速率高达16.6M/秒。IDE和EIDE磁盘比SCSI磁盘便宜，大多数现代PC都有一个或更多的主板上的IDE控制器。

Linux按照它发现的控制器的顺序命名IDE磁盘。主控制器上的主磁盘是/dev/had，slave磁盘是/dev/hdb。/dev/hdc是次IDE控制器上的master磁盘。IDE子系统向Linux登记IDE控制器而不是

磁盘。主IDE控制器的主标识符是3，次IDE控制器的标识符是22。这意味着如果一个系统有两个IDE控制器，那么在blk_dev和blkdevs向量表中在索引3和22会有IDE子系统的条目。IDE磁盘的块特殊文件反映了这种编号：磁盘/dev/had和/dev/hdb，都连接在主IDE控制器上，主设备号都是3。核心使用主设备标识符作为索引，对于这些块特殊文件的IDE子系统进行的所有的文件或者buffer cache操作都被定向到相应的IDE子系统。当执行一个请求的时候，IDE子系统负责判断这个请求是针对哪一个IDE磁盘。为此，IDE子系统使用设备特殊文件中的次设备号，这些信息允许它把请求定向到正确的磁盘的正确的分区。/dev/hdb，主IDE控制器上的slave IDE磁盘的设备标识符是（3，64）。它的第一个分区（/dev/hdb1）的设备标识符是（3，65）。

### 8.5.2 Initializing the IDE Subsystem（初始化IDE子系统）

IBM PC的大部分历史中都有IDE磁盘。这期间这些设备的接口发生了变化。这让IDE子系统的初始化过程比它第一次出现的时候更加复杂。

Linux可以支持的最大IDE控制器数目是4。每一个控制器都用一个ide_hwifs向量表中的一个ide_hwif_t数据结构表示。每一个ide_hwif_t数据结构包含两个ide_drive_t数据结构，分别表示可能支持的master和slave IDE驱动器。在IDE子系统初始化期间，Linux首先查看在系统的CMOS内存中记录的磁盘的信息。这种用电池做后备的内存在PC关机的时候不会丢失它的内容。这个CMOS内存实际上在系统的实时时钟设备里面，不管你的PC开或者关，它都在运行。CMOS内存的位置由系统的BIOS设置，同时告诉Linux系统中找到了什么IDE控制器和驱动器。Linux从BIOS中获取找到的磁盘的尺寸（geometry），用这些信息设置这个驱动器的ide_hwif_t的数据结构。大多数现代PC使用PCI芯片组例如Intel的82430 VX芯片组，包括了一个PCI EIDE控制器。IDE子系统使用PCI BIOS回调（callback）定位系统中的PCI （E）IDE控制器。然后调用这些芯片组的询问例程。

一旦发现一个IDE接口或者控制器，就设置它的ide_hwif_t来反映这个控制器和上面的磁盘。操作过程中IDE驱动程序向I/O内存空间的IDE命令寄存器写命令。主IDE控制器的控制和状态寄存器的缺省的I/O地址是0x1F0-0x1F7。这些地址是早期的IBM PC约定下来的。IDE驱动程序向Linux的buffer cache和VFS登记每一个控制器，分别把它加到blk_dev和blkdevs向量表中。IDE驱动程序也请求控制适当的中断。同样，这些中断也有约定，主IDE控制器是14，次IDE控制器是15。但是，象所有的IDE细节一样，这些都可以用核心的命令行选项改变。IDE驱动程序在启动的时候也为每一个找到的IDE控制器在gendisk列表中增加一个gendisk条目。这个列表稍后用于查看启动时找到的所有的硬盘的分区表。分区检查代码明白每一个IDE控制器可以控制两个IDE磁盘。

### 8.5.3 SCSI Disks（SCSI磁盘）

SCSI（Small Computer System Interface小型计算机系统接口）总线是一种有效的点对点的数据总线，每个总线支持多达8个设备，每个主机可以有一或者多个。每一个设备都必须由一个唯一的标识符，通常用磁盘上的跳线设置。数据可以在总线上的任意两个设备之间同步或者异步传

输，可以用32位宽的数据传输，速度可能高达40M/秒。SCSI总线可以在设备之间传输数据和状态信息，发起者（initiator）和目标（target）之间的事务会涉及多达8个不同的阶段。你可以通过SCSI总线上的5种信号判断出当前的阶段。这8个阶段是：

BUS FREE 没有设备有总线的控制权，当前没有发生任何事务。

ARBITRATION （仲裁）一个SCSI设备试图得到SCSI总线的控制权，它在地址管脚上声明（assert）它的SCSI标识符。最高编号的SCSI标识符成功。

SELECTION 一个设备通过仲裁成功地得到了SCSI总线的控制权，现在它必须向它要发送命令的SCSI目标发送信号。它在地址管脚上声明目标的SCSI标识符。

RESELECTION SCSI设备在处理请求的过程中可能断线，目标会重新选择发起者。并非所有的SCSI设备都支持这一阶段。

COMMAND 6、10或者12字节的命令可以从发起者发送到目标。

DATA IN，DATA OUT在这一阶段，数据在发起者和目标之间传输。

STATUS 在完成了所有的命令，进入这一阶段。允许目标向发起者发送一个状态字节，表示成功或失败。

MESSAGE IN，MESSAGE OUT在发起者和目标之间传递的附加信息。

Linux SCSI子系统由两个基本元素组成，每一个都用数据结构表示：

Host 一个SCSI host是一个物理的硬件，一个SCSI控制器。NCR810 PCI SCSI控制器是一个SCSI host的例子。如果一个Linux系统有多于一个同类型的SCSI控制器，每一个实例都分别用一个SCSI host表示。这意味着一个SCSI设备驱动程序可能控制多于一个控制器的实例。SCSI host通常总是SCSI命令的发起者（initiator）。

Device SCSI设备通常是磁盘，但是SCSI标准支持多种类型：磁带、CD-ROM和通用（generic）SCSI设备。SCSI设备通常都是SCSI命令的目标。这些设备必须不同地对待。例如可移动介质如CD-ROM或磁带，Linux需要探测介质是否取出。不同的磁盘类型有不同的主设备编号，允许Linux把块设备请求定向到合适的SCSI类型。

Initializing the SCSI Subsystem（初始化SCSI子系统）

初始化SCSI子系统相当复杂，反映出SCSI总线和设备的动态的实质。Linux在启动的时候初始化SCSI子系统：它查找系统中的SCSI控制器（SCSI host），并探测每一个SCSI总线，查找每一个

设备。然后初始化这些设备，让Linux核心的其余部分可以通过普通的文件和buffer cache块设备操作访问它们。这个初始化过程有四个阶段：

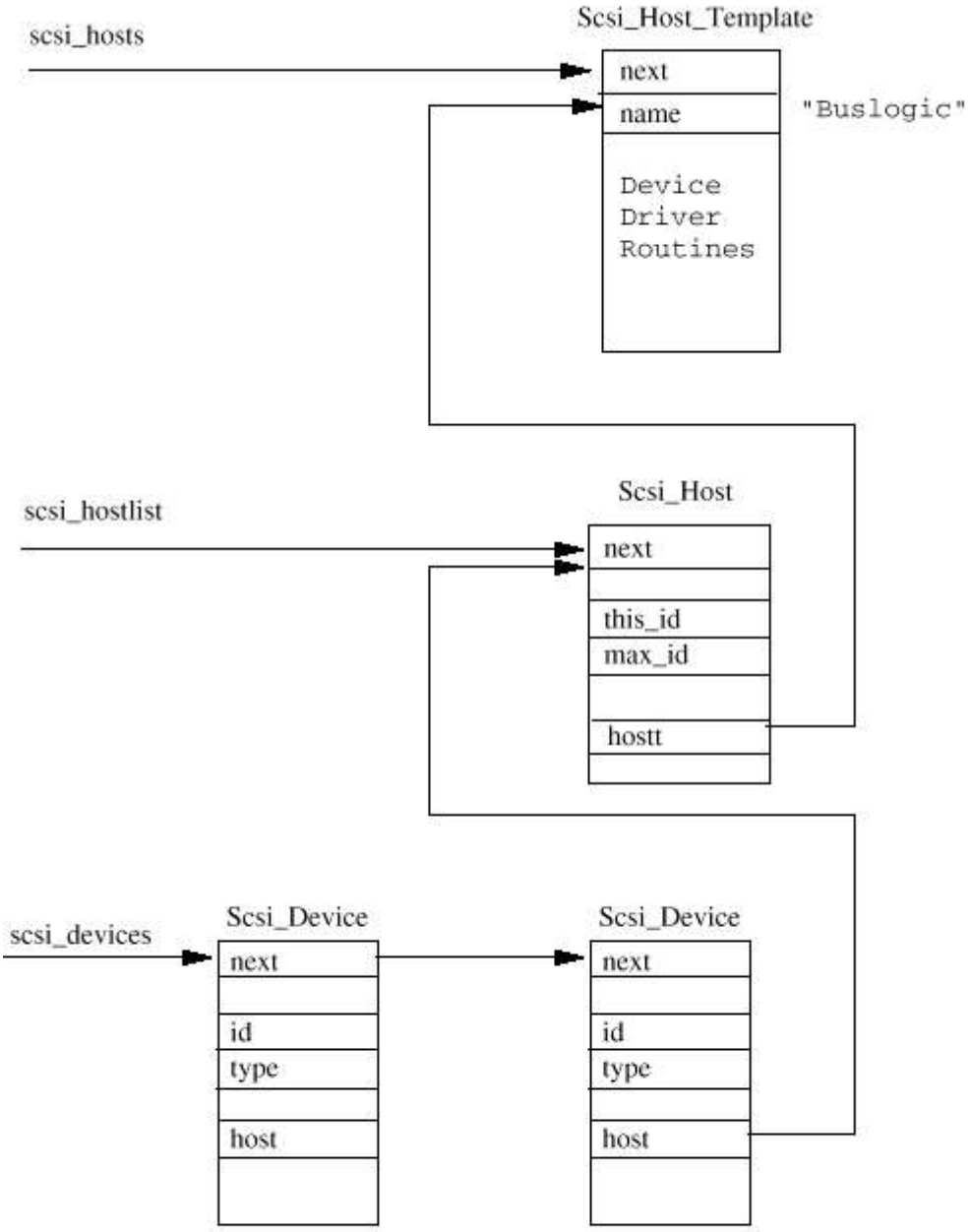



Figure 8.4: SCSI Data Structures

首先，Linux找出核心建立的时候建立到核心的哪一个SCSI host适配器或控制器有可以控制的硬件。每一个内建的SCSI host在buildin_scsi_hosts向量表中都有一个Scsi_Host_Template的条目。这个Scsi_Host_Template数据结构包括例程的指针，这些例程可以执行和SCSI host相关的动作例如探测这个SCSI host上粘附了什么SCSI设备。这些例程在SCSI子系统配置期间被调用，是支持这种host类型的SCSI设备驱动程序的一部分。每一个查到的SCSI控制器（有真实的SCSI设备粘附），它的Scsi_Host_Template数据结构都加到scsi_hosts列表中，表示有效的SCSI host。每一个探测到的host类型的每一个实例都用scsi_hostlist列表中的一个Scsi_Host数据结构表示。例如一个系统有两个NCR810 PCI SCSI控制器，在这个列表中会有两个Scsi_Host条目，

每一个控制器一个。每一个Scsi_Host指向的Scsi_Host_Template表示它的设备驱动程序。

现在每一个SCSI host都找到了，SCSI子系统必须找到每一个host总线上的所有的SCSI设备。SCSI设备编号从0到7，每一个设备编号或者SCSI标识符在它所粘附的SCSI总线上都是唯一的。SCSI标识符通常用设备上的跳线设置。SCSI初始化代码通过向每一个设备发送TEST_UNIT_READY命令来查找一个SCSI总线上的每一个SCSI设备。当一个设备回应，再向它发送一个ENQUIRY命令来完成它的判别。这向Linux给出Vendor的名称和设备的型号和修订号。SCSI命令用一个Scsi_Cmnd数据结构来表示，这些命令通过调用这个SCSI host的Scsi_Host_Template数据结构中的设备驱动程序例程传递给设备驱动程序。每一个找到的SCSI设备用一个Scsi_Device数据结构表示，每一个都指向它的父Scsi_Host。所有的Scsi_Device数据结构都加到scsi_devices列表中。图8.4显示了主要的数据结构和其他数据结构的关系。

有四种SCSI设备类型：磁盘、磁带、CD和通用（generic）。每一种SCSI类型都分别向核心登记，有不同的主块设备类型。但是，它们只有在一个或多个给定的SCSI设备类型的设备找到的时候才登记自己。每一个SCSI类型，例如SCSI磁盘，维护它自己的设备表。它用这些表把核心的块操作（文件或buffer cache）定向到正确的设备驱动程序或SCSI host。每一个SCSI类型都用一个Scsi_Type_Template数据结构表示。它包括这种类型的SCSI设备的信息和执行多种任务的例程的地址。SCSI子系统使用这些模板调用每一种SCSI设备类型的SCSI类型例程。换句话说，如果SCSI子系统希望粘附一个SCSI磁盘设备，它会调用SCSI 磁盘类型的例程。如果探测到某类型的一个或多个SCSI设备，它的Scsi_Type_Templates的数据结构就加到了scsi_devicelist列表中。

SCSI子系统初始化的最后阶段是调用每一个登记的Scsi_Device_Template的完成函数。对于SCSI磁盘类型让所有的SCSI磁盘转动起来并记录它们的磁盘尺寸。它也把表示所有SCSI磁盘的gendisk数据结构增脚的磁盘的链接列表中，如图8.3。

Delivering Block Device Requests（传递块设备请求）

一旦Linux初始化了SCSI子系统，就可以使用SCSI设备了。每一个有效的SCSI设备类型都在核心中登记自己，所以Linux可以把块设备请求定向到它那里。这些请求可能是通过blk_dev的buffer cache请求或者是通过blkdevs的文件操作。拿一个由一个或多个EXT2文件系统分区的SCSI磁盘驱动器为例，当它的EXT2分区安装上的时候核心的缓冲区请求是如何定向到正确的SCSI磁盘呢？

每一个向/从一个SCSI磁盘分区读/写一块数据的请求都会在blk_dev向量表中这个SCSI磁盘的current_request列表中加入一个新的request数据结构。如果这个request列表正在处理，那么buffer cache不需要做什么。否则它必须让SCSI磁盘子系统处理它的请求队列。系统中的每一个SCSI磁盘用一个Scsi_Disk数据结构表示。它们保存在rscsi_disks向量表中，用SCSI磁盘分区的次设备号的一部分作为索引。例如，/dev/sdb1主设备号8，次设备号17，它的所以是1。每一个Scsi_Disk的数据结构包括一个指向表示这个设备的Scsi_Device数据结构的指针。Scsi_Device又

指向一个“拥有它”的Scsi_Host数据结构。Buffer cache中的request数据结构转换成为描述需要发送到SCSI设备的SCSI命令的Scsi_Cmd数据结构中，并在表示这个设备的Scsi_Host数据结构中排队。一旦适当的数据块读/写之后，会由各自的SCSI设备驱动程序处理。

## 8.6 Network Devices（网络设备）

一个网络设备，只要关系到Linux的网络子系统，是一个发送和接收数据包的实体。通常是一个物理的设备，例如一个以太网卡。但是一些网络设备是纯软件的，例如loopback设备，用于向自己发送数据。每一个网络设备用一个device数据结构表示。网络设备驱动程序在核心启动网络初始化的时候向Linux登记它控制的设备。Device数据结构包括这个设备的信息和允许大量支持的网络协议使用这个设备的服务的函数的地址。这些函数多数和使用这个网络设备传输数据有关。设备使用标准的网络支持机制，向适当的协议层传输接收的数据。传输和接收的所有的网络数据（包packets）都用sk_buff数据结构表示，这是灵活的数据结构，允许网络协议头很容易地增加和删除。网络协议层如何使用网络设备，它们如何使用sk_buff数据结构来回传递数据，在网络章（第10章）有详细的描述。本章集中在device数据结构以及网络设备如何被发现和初始化。

参见include/linux/netdevice.h

device数据结构包括网络设备的信息：

Name 不象块和字符设备，它们的设备特殊文件用mknod命令创建，网络设备特殊文件在系统的网络设备发现并初始化的时候自然出现。它们的名字是标准的，每一个名字都表示了它的设备类型。同种类型的多个设备从0向上依次编号。因此以太网设备编号为/dev/eth0、/dev/eth1、/dev/eth2等等。一些常见的网络设备是：

/dev/ethN 以太网设备

/dev/slN SLIP设备

/dev/pppN PPP设备

/dev/lo loopback 设备

Bus Information 这是设备驱动程序控制设备需要的信息。Irq是设备使用的中断。Base address是设备的控制和状态寄存器在I/O内存种的地址。DMA通道是这个网络设备使用的DMA通道号。所有这些信息在启动时设备初始化的时候设置。

Interface Flags 这些描述了这个网络设备的特性和能力。

IFF_UP 接口up ，正在运行

IFF_BROADCAST 设备的广播地址有效

IFF_DEBUG 设备的debug选项打开

IFF_LOOPBACK 这是一个loopback设备

IFF_POINTTOPOINT 这是点对点的连接（SLIP and PPP）

IFF_NOTRAILERS No network trailers

IFF_RUNNING 分配了资源

IFF_NOARP 不支持ARP协议

IF_PROMISC 设备在混合（promiscuous）接收模式，它会接收所有的包，不管它们的地址是谁。

IFF_ALLMULTI 接收所有的IP Multicast帧

IFF_MULTICAST 可以接收IP multicast帧

Protocal Information 每一个设备都描述它可以被网络协议层如何使用：

Mtu 不包括需要增加的链路层的头这个网络能够传输的最大尺寸的包。这个最大值用于协议层例如IP，来选择一个合适的包大小进行发送。

Family family显示了设备可以支持的协议族。所有Linux网络设备都支持的family是AF_INET，Internet地址family。

Type 硬件接口类型描述了这个网络设备连接的介质。Linux网络设备支持多种介质类型。包括Ethernet、X.25，Token Ring、Slip、PPP和Apple Localtalk。

Addresses device 数据结构保存一些和这个网络设备相关的地址，包括IP地址

Packet Queue 这是一个sk_buff的包队列，等待网络设备进行传输

Support Functions 每一个设备都提供了一组标准的例程，让协议层调用，作为对于设备链路层的接口的一部分。包括设置和帧传输例程，以及增加标准帧头和收集统计信息的例程。这些统计信息可以用ifcnfig看到

8.6.1 Initializing Network Devices（初始化网络设备）

网络设备驱动程序象其他Linux设备驱动程序一样，可以建立到Linux核心中。每一个可能的网络设备都用dev_base列表指针指向的网络设备列表中的一个device数据结构表示。如果需要设备相关的操作，网络层调用网络设备服务例程（放在device数据结构中）其中的一个。但是，初始的时候，每一个device数据结构只是放了初始化或者探测例程的地址。

网络驱动程序必须解决两个问题。首先，不是所有建立在Linux核心的网络设备驱动程序都会有控制的设备；第二，系统中的以太网设备总是叫做/dev/eth0、/dev/eth1等等，而不管底层的设备驱动程序是什么。“丢失“网络设备的问题容易解决。在调用每一个网络设备的初始化例程的时候，它返回一个状态，显示它是否定位到了它驱动的控制器的一个实例。如果驱动程序没有找到任何设备，它由dev_base指向的device列表中的条目就被删除。如果驱动程序可以找到一个设备，它就用这个设备的信息和网络设备驱动程序中的支持函数的地址填充device数据结构其余的部分。

第二个问题，就是动态地分配以太网设备到标准的/dev/ethN设备特殊文件上，用更优雅的方式解决。Device列表中有8个标准的条目：eth0、eth1到eth7。所有条目的初始化例程都一样。它顺序尝试建立在核心的每一个以太网设备驱动程序，直到找到一个设备。当驱动程序找到它的以太网设备，它就填充它现在拥有的ethN的device数据结构。这时网络驱动程序也要初始化它控制的物理硬件，并找出它使用的IRQ、DMA等等。驱动程序可能找到它控制的网络设备的几个实例，在这种情况下，它就占用几个/dev/ethN的device数据结构。一旦所有的8个标准的/dev/ethN都分配了，就不会再探测更多的以太网设备。

## Chapter 9

## The File System（文件系统）

本章描述Linux如何维护它支持的文件系统中的文件。描述了虚拟文件系统（Virtual File System VFS）并解释了Linux核心中真实的文件系统如何被支持

Linux的一个最重要的特点之一使它可以支持许多不同的文件系统。这让它非常灵活，可以和许多其他操作系统共存。在写作本章的时候，Linux可一直支持15种文件系统：ext、ext2、xia、minix、umsdos、msdos、vfat、proc、smb、ncp、iso9660、sysv、hpfs、affs和ufs，而且不容置疑，随着时间流逝，会加入更多的文件系统。

在Linux中，象Unix一样，系统可以使用的不同的文件系统不是通过设备标识符（例如驱动器编号或设备名称）访问，而是连接成一个单一的树型的结构，用一个统一的单个实体表示文件系统。Linux在文件系统安装的时候把它加到这个单一的文件系统树上。所有的文件系统，不管什么类型，都安装在一个目录，安装的文件系统的文件掩盖了这个目录原来存在的内容。这个目录叫做安装目录或安装点。当这个文件系统卸载的时候，安装目录自己的文件又可以显现出来。

当磁盘初始化的时候（比如用fdisk），利用一个分区结构把物理磁盘划分成一组逻辑分区。每一个分区可以放一个文件系统，例如一个EXT2文件系统。文件系统在物理设备的块上通过目录、软链接等把文件组织成逻辑的树型结构。可以包括文件系统的设备是块设备。系统中的第一个IDE磁盘驱动器的第一个分区，IDE磁盘分区/dev/hda1，是一个块设备。Linux文件系统把这些块设备看成简单的线性的块的组合，不知道也不去关心底层的物理磁盘的尺寸。把对设备的特定的块的读的请求映射到对于设备有意义的术语：这个块保存在硬盘上的磁道、扇区和柱面，这是每一个块设备驱动程序的任务。一个文件系统不管它保存在什么设备上，都应该用同样的方式工作，有同样的观感。另外，使用Linux的文件系统，是否这些不同的文件系统在不同的硬件控制器的控制下的不同的物理介质上都是无关紧要的（至少对于系统用户是这样）。文件系统甚至可能不在本地系统上，它可能是通过网络连接远程安装的。考虑以下的例子，一个Linux系统的根文件系统在一个SCSI磁盘上。

A E boot etc lib opt tmp usr

C F cdrom fd proc root var sbin

D bin dev home mnt lost+found

不管是操作这些文件的用户还是程序都不需要知道/C实际上是在系统的第一个IDE磁盘上的一个安装的VFAT文件系统。本例中（实际是我家中的Linux系统），/E是次IDE控制器上的master IDE磁盘。第一个IDE控制器是PCI控制器，而第二个是ISA控制器，也控制着IDE CDROM，这些也都无关紧要。我可以用一个modem和PPP网络协议拨号到我工作的网络，这时，我可以远程安装我的Alpha AXP Linux系统的文件系统到/mnt/remote。

文件系统中的文件包含了数据的集合：包含本章源的文件是一个ASCII文件，叫做filesystems.tex。一个文件系统不仅保存它包括的文件的数据，也保存文件系统的结构。它保存了Linux用户和进程看到的所有的信息，例如文件、目录、软链接、文件保护信息等等。另外，它必须安全地保存这些信息，操作系统的基本的一致性依赖于它的文件系统。没有人可以使用一个随机丢失数据和文件的操作系统（不知道是否有，虽然我曾经被拥有的律师比Linux开发者还多的操作系统伤害过）。

Minix是Linux的第一个文件系统，有相当的局限，性能比较差。它的文件名不能长于14个字符（这仍然比8.3文件名要好），最大的文集大小是64M字节。第一眼看去，64M字节好像足够大，但是设置中等的数据库需要更大的文件大小。第一个专为Linux设计的文件系统，扩展文件系统或EXT（Extend File System），在1992年4月引入，解决了许多问题，但是仍然感到性能低。所以，1993年，增加了扩展文件系统第二版，或EXT2。这种文件系统在本章稍后详细描述。

当EXT文件系统增加到Linux的时候进行了一个重要的开发。真实的文件系统通过一个接口层从操

作系统和系统服务中分离出来，这个接口叫做虚拟文件系统或VFS。VFS允许Linux支持许多（通常是不同的）文件系统，每一个都向VFS表现一个通用的软件接口。Linux文件系统的所有细节都通过软件进行转换，所以所有的文件系统对于Linux核心的其余部分和系统中运行的程序显得一样。Linux的虚拟文件系统层允许你同时透明地安装许多不同的文件系统。

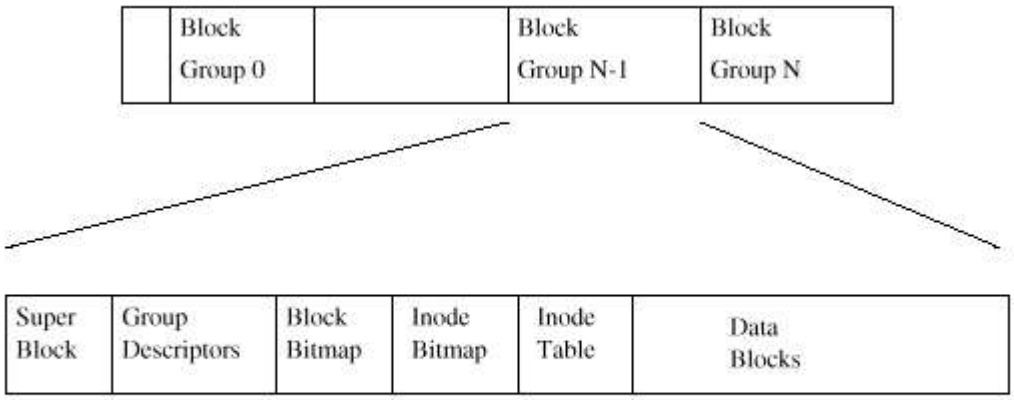



Figure 9.1: Physical Layout of the EXT2 File system

Linux虚拟文件系统的实现使得对于它的文件的访问尽可能的快速和有效。它也必须保证文件和文件数据正确地存放。这两个要求相互可能不平等。Linux VFS在安装和使用每一个文件系统的时候都在内存中高速缓存信息。在文件和目录创建、写和删除的时候这些高速缓存的数据被改动，必须非常小心才能正确地更新文件系统。如果你能看到运行的核心中的文件系统的数据结构，你就能够看到文件系统读写数据块，描述正在访问的文件和目录的数据结构会被创建和破坏，同时设备驱动程序会不停地运转，获取和保存数据。这些高速缓存中最重要的是Buffer Cache，在文件系统访问它们底层的块设备的时候结合进来。当块被访问的时候它们被放到Buffer Cache，根据它们的状态放在不同的队列中。Buffer Cache不仅缓存数据缓冲区，它也帮助管理块设备驱动程序的异步接口。

## 9.1 The Second Extended File System (EXT2)

EXT2被发明（Remy Card）作为Linux一个可扩展和强大的文件系统。它至少在Linux社区中是最成功的文件系统，是所有当前的Linux发布版的基础。EXT2文件系统，象所有多数文件系统一样，建立在文件的数据存放在数据块中的前提下。这些数据块都是相同长度，虽然不同的EXT2文件系统的块长度可以不同，但是对于一个特定的EXT2文件系统，它的块长度在创建的时候就确定了（使用mke2fs）。每一个文件的长度都按照块取整。如果块大小是1024字节，一个1025字节的文件会占用两个1024字节的块。不幸的是这一意味着平均你每一个文件浪费半个块。通常计算中你会用磁盘利用来交换CPU对于内存的使用，这种情况下，Linux象大多数操作系统一样，为了较少CPU的负载，使用相对低效率的磁盘利用率来交换。不是文件系统中所有的块都包含数据，一些块必须用于放置描述文件系统结构的信息。EXT2用一个inode数据结构描述系统中的每一个文件，定义了系统的拓扑结构。一个inode描述了一个文件中的数据占用了哪些块以及文件的访问权限、文件的修改时间和文件的类型。EXT2文件系统中的每一个文件都用一个inode描述，而每一个inode都用一个独一无二的数字标识。文件系统的inode都放在一起，在inode表中。EXT2的目录是简单的特殊文件（它们也使用inode描述），包括它们目录条目的inode的指针。

图9.1显示了一个EXT2文件系统占用了一个块结构的设备上一系列的块。只要提到文件系统，块设备都可以看作一系列能够读写的块。文件系统不需要关心自身要放在物理介质的哪一个块上，这是设备驱动程序的工作。当一个文件系统需要从包括它的块设备上读取信息或数据的时候，它请求对它支撑的设备驱动程序读取整数数目的块。EXT2文件系统把它占用的逻辑分区划分成块组（Block Group）。每一个组除了当作信息和数据块来存放真实的文件和目录之外，还复制对于文件系统一致性至关重要的信息。这种复制的信息对于发生灾难，文件系统需要恢复的时候是必要的。下面对于每一个块组的内容进行了详细的描述。

### 9.1.1 The EXT2 Inode（EXT2 I节点）

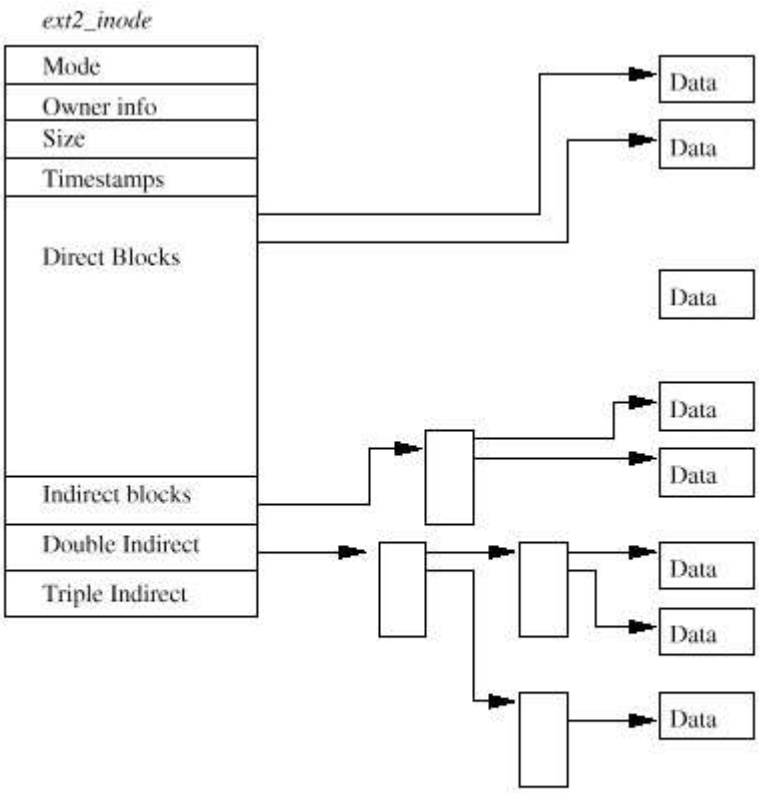



Figure 9.2: EXT2 Inode

在EXT2文件系统中，I节点是建设的基石：文件系统中的每一个文件和目录都用一个且只用一个inode描述。每一个块组的EXT2的inode都放在inode表中，还有一个位图，让系统跟踪分配和未分配的I节点。图9.2显示了一个EXT2 inode的格式，在其他信息中，它包括一些域：

参见include/linux/ext2_fs_i.h

mode 包括两组信息：这个inode描述了什么和用户对于它的权限。对于EXT2，一个inode可以描述一个文件、目录、符号链接、块设备、字符设备或FIFO。

Owner Information 这个文件或目录的数据的用户和组标识符。这允许文件系统正确地进行文件

访问权限控制

Size 文件的大小（字节）

Timestamps 这个inode创建的时间和它上次被修改的时间。

Datablocks 指向这个inode描述的数据的块的指针。最初的12个是指向这个inode描述的数据的物理块，最后的3个指针包括更多级的间接的数据块。例如，两级的间接块指针指向一个指向数据块的块指针的块指针。的这意味着小于或等于12数据块大小的文件比更大的文件的访问更快。

你应该注意EXT2 inode可以描述特殊设备文件。这些不是真正的文件，程序可以用于访问设备。/dev下所有的设备文件都是为了允许程序访问Linux的设备。例如mount程序用它希望安装的设备文件作为参数。

### 9.1.2 The EXT2 Superblock（EXT2超级块）

超级块包括这个文件系统基本大小和形状的描述。它里面的信息允许文件系统管理程序用于维护文件系统。通常文件系统安装时只有块组0中的超级块被读取，但是每一个块组中都包含一个复制的拷贝，用于系统崩溃的时候。除了其他一些信息，它包括：

参见include/linux/ext2_fs_sb.h

Magic Number 允许安装软件检查这是否是一个EXT2文件系统的超级块。对于当前版本的EXT2是0xEF53。

Revision Level major和minor修订级别允许安装代码确定这个文件系统是否支持只有在这种文件系统特定修订下才有的特性。这也是特性兼容域，帮助安装代码确定哪些新的特征可以安全地使用在这个文件系统上。

Mount Count and Maximum Mount Count 这些一起允许系统确定这个文件系统是否需要完全检查。每一次文件系统安装的时候mount count增加，当它等于maximum mount count的时候，会显示告警信息“maximal mount count reached，running e2fsck is recommended”。

Block Group Number 存放这个超级块拷贝的块组编号。

Block Size 这个文件系统的块的字节大小，例如1024字节。

Blocks per Group 组中的块数目。象块大小一样，这是文件系统创建的时候确定的。

Free Blocks 文件系统中空闲块的数目

Free Inodes 文件系统中空闲的inode

First Inode 这是系统中第一个inode的编号。一个EXT2根文件系统中的第一个inode是‘/’目录

的目录条目

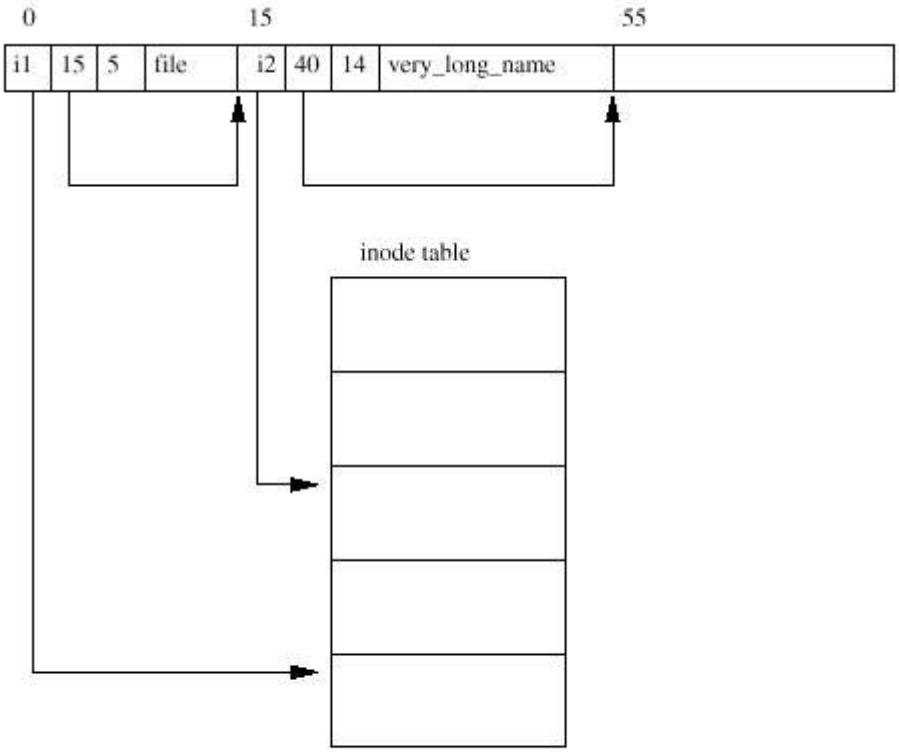



Figure 9.3: EXT2 Directory

9.1.3 The EXT2 Group Descriptor（EXT2组描述符）

每一个块组都有一个数据结构描述。象超级块，所有得亏组的组描述符在每一块组都进行复制。每一个组描述符包括以下信息：

参见include/linux/ext2_fs.h ext2_group_desc

Blocks Bitmap 这个块组的块分配位图的块编号，用在块的分配和回收过程中

Inode Bitmap 这个块组的inode位图的块编号。用在inode的分配和回收过程中。

Inode Table 这个块组的inode table的起始块的块编号。每一个EXT2 inode数据结构表示的inode在下面描述

Free blocks count，Free Inodes count，Used directory count

组描述符依次排列，它们一起组成了组描述符表（group descriptor table）。每一个块组包括块组描述符表和它的超级块的完整拷贝。只有第一个拷贝（在块组0）实际被EXT2文件系统使用。其他拷贝，象超级块的其他拷贝一样，只有在主拷贝损坏的时候才使用。

### 9.1.4 EXT2 Directories（EXT2目录）

在EXT2文件系统中，目录是特殊文件，用来创建和存放对于文件系统中的文件的访问路径。图9.3显示了内存中一个目录条目的布局。一个目录文件，是一个目录条目的列表，每一个目录条目包括以下信息：

参见include/linux/ext2_fs.h ext2_dir_entry

inode 这个目录条目的inode。这是个放在块组的inode表中的inode数组的索引。图9.3叫做file的文件的目录条目引用的inode 是 i1。

Name length 这个目录条目的字节长度

Name 这个目录条目的名字

每一个目录中的前两个条目总是标准的“.”和“..”,分别表示“本目录”和“父目录”。

### 9.1.5 Finding a File in a EXT2 File System（在一个EXT2文件系统中查找一个文件）

Linux的文件名和所有的Unix文件名的格式一样。它是一系列目录名，用“/”分隔，以文件名结尾。一个文件名称的例子是/home/rusling/.cshrc，其中/home和/rusling是目录名，文件名是.cshrc。象其它Unix系统一样，Linux不关心文件名本身的格式：它可以任意长度，由可打印字符组成。为了在EXT2文件系统中找到代表这个文件的inode，系统必须逐个解析目录中的文件名直到得到这个文件。

我们需要的第一个inode是这个文件系统的根的inode。我们通过文件系统的超级块找到它的编号。为了读取一个EXT2 inode我们必须在适当的块组中的inode表中查找。举例，如果根的inode编号是42，那么我们需要块组0中的inode表中的第42个inode。Root inode是一个EXT2目录，换句话说root inode的模式描述它是一个目录，它的数据块包括EXT2目录条目。

Home是这些目录条目之一，这个目录条目给了我们描述/home目录的inode编号。我们必须读取这个目录（首先读取它的inode，然后读取从这个inode描述的数据块读取目录条目），查找rusling条目，给出描述/home/rusling目录的inode编号。最后，我们读取描述/home/rusling目录的inode指向的目录条目，找到.cshrc文件的inode编号，这样，我们得到了包括文件里信息的数据块。

### 9.1.6 Changing the size of a File in an EXT2 File System（在EXT2文件系统中改变一个文件的大小）

文件系统的一个常见问题是它趋于更多碎片。包含文件数据的块分布在整个文件系统，数据块越分散，对于文件数据块的顺序访问越没有效率。EXT2文件系统试图克服这种情况，它分配给一个文件的新块物理上和它的当前数据块接近或者至少和它的当前数据块在一个块组里面。只有这个失败了它才分配其它块组中的数据块。

无论何时一个进程试图象一个文件写入数据，Linux文件系统检查数据是否会超出文件最后分配块的结尾。如果是，它必须为这个文件分配一个新的数据块。直到这个分配完成，该进程无法运行，它必须等待文件系统分配新的数据块并把剩下的数据写入，然后才能继续。EXT2块分配例程所要做的第一个事情是锁定这个文件系统的EXT2超级块。分配和释放块需要改变超级块中的域，Linux文件系统不能允许多于一个进程同一时间都进行改变。如果另一个进程需要分配更多的数据块，它必须等待，直到这个进程完成。等待超级块的进程被挂起，不能运行，直到超级块的控制权被它的当前用户释放。对于超级块的访问的授权基于一个先来先服务的基础（first come first serve），一旦一个进程拥有了超级块的控制，它一直维持控制权到它完成。锁定超级块之后，进程检查文件系统是否有足够的空闲块。如果没有足够的空闲块，分配更多的尝试会失败，进程交出这个文件系统超级块的控制权。

如果文件系统中有足够的空闲块，进程会试图分配一块。如果这个EXT2文件系统已经建立了预分配的数据块，我们就可以取用。预分配的块实际上并不存在，它们只是分配块的位图中的保留块。VFS inode用两个EXT2特有的域表示我们试图分配新数据块的文件：prealloc_block and prealloc_count，分别是预分配块中第一块的编号和预分配块的数目。如果没有预分配块或者预分配被禁止，EXT2文件系统必须分配一个新的数据块。EXT2文件系统首先查看文件最后一个数据块之后数据块是否空闲。逻辑上，这是可分配的效率最高的块，因为可以让顺序访问更快。如果这个块不是空闲，继续查找，在随后的64块中找理想的数据块。这个块，虽然不是最理想，但是至少和文件的其它数据块相当接近，在一个块组中。

参见fs/ext2/balloc.c ext2_new_block()

如果这些块都没有空闲的，进程开始顺序查看所有其它块组直到它找到空闲的块。块分配代码在这些块组中查找8个空闲数据块的簇。如果无法一次找到8个，它会降低要求。如果希望进行块预分配，并允许，它会相应地更新prealloc_block和prealloc_count。

不管在哪里找到了空闲的数据块，块分配代码会更新块组的块位图，并从buffer cache中分配一个数据缓冲区。这个数据缓冲区使用支撑文件系统的设备标识符和分配块的块编号来唯一标识。缓冲区中的数据被置为0，缓冲区标记为“dirty”表示它的内容还没有写到物理磁盘上。最后，超级块本身也标记位“dirty”，显示它进行了改动，然后它的锁被释放。如果有进程在等待超级块，那么队列中第一个进程就允许运行，得到超级块的排它控制权，进行它的文件操作。进程的数据写到新的数据块，如果数据块填满，整个过程重复进行，再分配其它数据块

## 9.2 The Virtual File System（虚拟文件系统VFS）

图9.4显示了Linux核心的虚拟文件系统和它的真实的文件系统之间的关系。虚拟文件系统必须管理任何时间安装的所有的不同的文件系统。为此它管理描述整个文件系统（虚拟）和各个真实的、安装的文件系统的数据结构。

相当混乱的是，VFS也使用术语超级块和inode来描述系统的文件，和EXT2文件系统使用的超级块和inode的方式非常相似。象EXT2的inode，VFS的inode描述系统中的文件和目录：虚拟文件系统的内容和拓扑结构。从现在开始，为了避免混淆，我会用VFS inode和VFS超级块以便同EXT2的inode和超级块区分开来。

参见fs/*

当每一个文件系统初始化的时候，它自身向VFS登记。这发生在系统启动操作系统初始化自身的时候。真实的文件系统自身建立在内核中或者是作为可加载的模块。文件系统模块在系统需要的时候加载，所以，如果VFAT文件系统用核心模块的方式实现，那么它只有在一个VFAT文件系统安装的时候才加载。当一个块设备文件系统安装的时候，（包括root文件系统），VFS必须读取它的超级块。每一个文件系统类型的超级块的读取例程必须找出这个文件系统的拓扑结构，并把这些信息映射到一个VFS超级块的数据结构上。VFS保存系统中安装的文件系统的列表和它们的VFS超级块列表。每一个VFS超级块包括了文件系统的信息和完成特定功能的例程的指针。例如，表示一个安装的EXT2文件系统的超级块包括一个EXT2相关的inode的读取例程的指针。这个EXT2 inode读取例程，象所有的和文件系统相关的inode读取例程一样，填充VFS inode的域。每一个VFS超级块包括文件系统中的一个VFS inode的指针。对于root文件系统，这是表示“/”目录的inode。这种信息映射对于EXT2文件系统相当高效，但是对于其他文件系统相对效率较低。

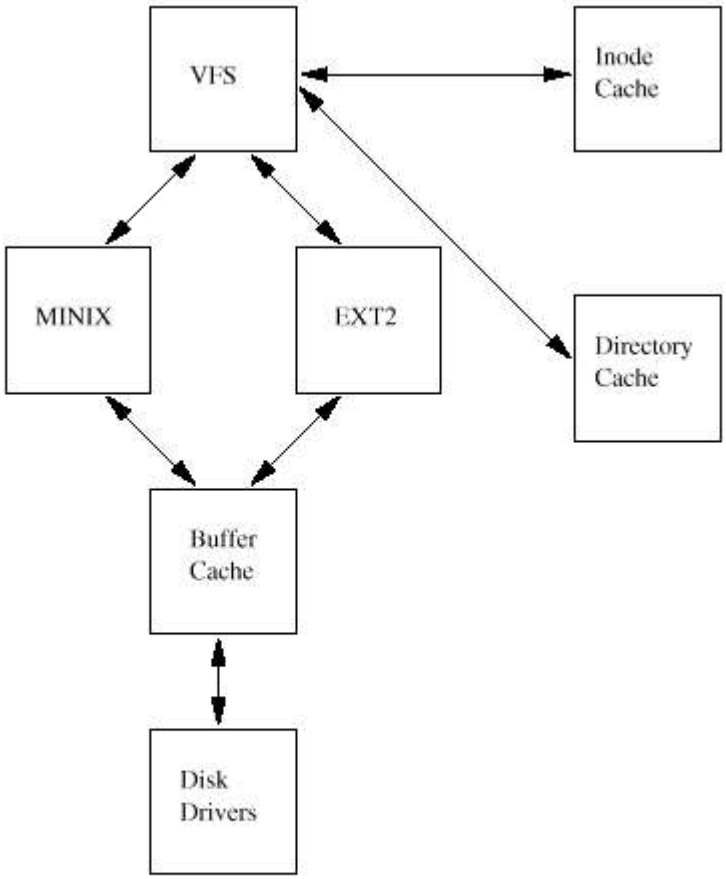



Figure 9.4: A Logical Diagram of the Virtual File System

当系统的进程访问目录和文件的时候，调用系统例程，游历系统中的VFS inode。例如再一个目录中输入ls或者cat一个文件，让VFS查找代表这个文件系统的VFS inode。映为系统中的每一个文件和目录都用一个VFS inode代表，所以一些inode会被重复访问。这些inode保存在inode cache，这让对它们的访问更快。如果一个inode不在inode cache中，那么必须调用一个和文件系统相关的例程来读取适当的inode。读取这个inode的动作让它被放到了inode cache，以后对这个inode的访问会让它保留在cache中。较少使用的VFS inode 会从这个高速缓存中删除。

参见fs/inode.c

所有的Linux文件系统使用一个共同的buffer cache来缓存底层的设备的数据缓冲区，这样可以加速对于存放文件系统的物理设备的访问，从而加快对文件系统的访问。这个buffer cache独立于文件系统，集成在Linux核心分配、读和写数据缓冲区的机制中。让Linux文件系统独立于底层的介质和支撑的设备驱动程序有特殊的好处。所有的块结构的设备向Linux核心登记，并表现为一个统一的，以块为基础的，通常是异步的接口。甚至相对复杂的块设备比如SCSI设备也是这样。当真实的文件系统从底层的物理磁盘读取数据的，引起块设备驱动程序从它们控制的设备上读取物理块。在这个块设备接口中集成了buffer cache。当文件系统读取了块的时候，它们被存放到了所有的文件系统和Linux核心共享的全局的buffer cache中。其中的buffer（缓冲区）用它们的块编号和被读取设备的一个唯一的标识符来标记。所以，如果相同的数据经常需要，它会从buffer cache中读取，而不是从磁盘读取（会花费更多时间）。一些设备支持超前读（read ahead），数据块会预先读取，以备以后可能的读取。

参见fs/buffer.c

VFS也保存了一个目录查找的缓存，所以一个常用的目录的inode可以快速找到。作为一个试验，试着对于你最近没有列表的目录列表。第一次你列表的时候，你会注意到短暂的停顿，当时第二次你列表的时候，结果会立即出来。目录缓存本身不存储目录里的inode，这是inode cache负责的，目录缓存只是存储目录项目全称和它们的inode编号。

参见fs/dcache.c

### 9.2.1 The VFS Superblock（VFS超级块）

每一个安装的文件系统都用VFS超级块表示。除了其它信息，VFS超级块包括：

参见include/linux/fs.h

Device 这是包含文件系统的块设备的设备标识符。例如，/dev/hda1，系统中的第一个IDE磁盘，设备标识符是0x301

Inode pointers 其中的mounted inode指针指向该文件系统的第一个inode。Covered inode指针指向文件系统安装到的目录的inode。对于root文件系统，它的VFS超级块中没有covered指针。

Blocksize 文件系统块的字节大小，例如1024字节。

Superblock operations 指向一组本文件系统超级块例程的指针。除了其他类型之外，VFS使用这些例程读写inode和超级块

File System type 指向这个安装的文件系统的file_system_type数据结构的一个指针

File System Specific 指向这个文件系统需要的信息的一个指针

### 9.2.2 The VFS Inode

象EXT2文件系统，VFS中每一个文件、目录等等都用一个且只用一个VFS inode代表。每一个VFS inode中的信息使用文件系统相关的例程从底层的文件系统中获取。VFS inode只在核心的内存中存在，只要对系统有用，就一直保存在VFS inode cache中。除了其它信息，VFS inode包括一些域：

参见include/linux/fs.h

device 存放这个文件（或这个VFS inode代表的其它实体）的设备的设备标识符。

Inode nunber 这个inode的编号，在这个文件系统中唯一。Device 和inode number的组合在整个虚拟文件系统中是唯一的。

Mode 象EXT2一样，这个域描述这个VFS inode代表的东西和对它的访问权限。

User ids 属主标识符

Times 创建、修改和写的时间

Block size 这个文件的块的字节大小，例如1024字节

Inode operations 指向一组例程地址的指针。这些例程和文件系统相关，执行对于这个inode的操作，例如truncate这个inode代表的文件

Count 系统组件当前使用这个VFS inode的数目。Count 0意味着这个inode是空闲，可以废弃或者重用。

Lock 这个域用于锁定这个VFS inode。例如当从文件系统读取它的时候

Dirty 显示这个VFS inode是否被写过，如果这样，底层的文件系统需要更新。

File system specific information

### 9.2.3 Registering the File Systems（登记文件系统）

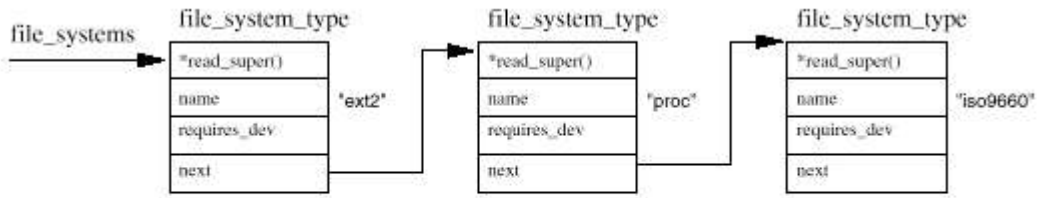



Figure 9.5: Registered File Systems

当你建立Linux核心的时候，你会被提问是否需要每一个支持的文件系统。当核心建立的时候，文件系统初始化代码包括对于所有内建的文件系统的初始化例程的调用。Linux文件系统也可以建立成为模块，这种情况下，它们可以在需要的时候加载或者手工使用insmod加载。当家在一个文件系统模块的时候，它自身向核心登记，当卸载的时候，它就注销。每一个文件系统的初始化例程都向虚拟文件系统注册自身，并用一个file_system_type数据结构代表，这里面包括文件系统的名称和一个指向它的VFS超级块的读取例程的指针。图9.5显示file_system_type数据结构被放到了由file_systems指针指向的一个列表中。每一个file_system_type数据结构包括以下信息：

参见fs/filesystems.c sys_setup()

参见 include/linux/fs.h file_system_type

Superblock read routine 在这个文件系统的一个实例安装的时候，由VFS调用这个例程

File Systme name 文件系统的名称，例如ext2

Device needed 是否这个文件系统需要一个设备支持？并非所有的文件系统需要一个设备来存放。例如/proc文件系统，不需要一个块设备

你可以检查/proc/filesystems来查看登记了哪些文件系统，例如：

ext2

nodev proc

iso9660

9.2.4 Mounting a File System（安装一个文件系统）

当超级用户试图安装一个文件系统的时候，Linux核心必须首先验证系统调用中传递的参数。虽然mount可以执行一些基本的检查，但是它不知道这个核心建立是可以支持的文件系统或者提议的安装点是否存在。考虑以下的mount 命令：

$ mount –t iso9660 –o ro /dev/cdrom /mnt/cdrom

这个mount命令会传递给核心三部分信息：文件系统的名称、包括这个文件系统的物理块设备和这个新的文件系统要安装在现存的文件系统拓扑结构中的哪一个地方。

虚拟文件系统要做的第一件事情是找到这个文件系统。它首先查看file_systems指向的列表中的每一个file_system_type数据结构，查看所有已知的文件系统。如果它找到了一个匹配的名称，它就直到这个核心支持这个文件系统类型，并得到了文件系统相关例程的地址，去读取这个文件系统的超级块。如果它不能找到一个匹配的文件系统名称，如果核心内建了可以支持核心模块按需加载（参见第12章），就可以继续。这种情况下，在继续之前，核心会请求核心守护进程加载适当的文件系统模块。

参见fs/super.c do_mount()

参见fs/super.c get_fs_type()

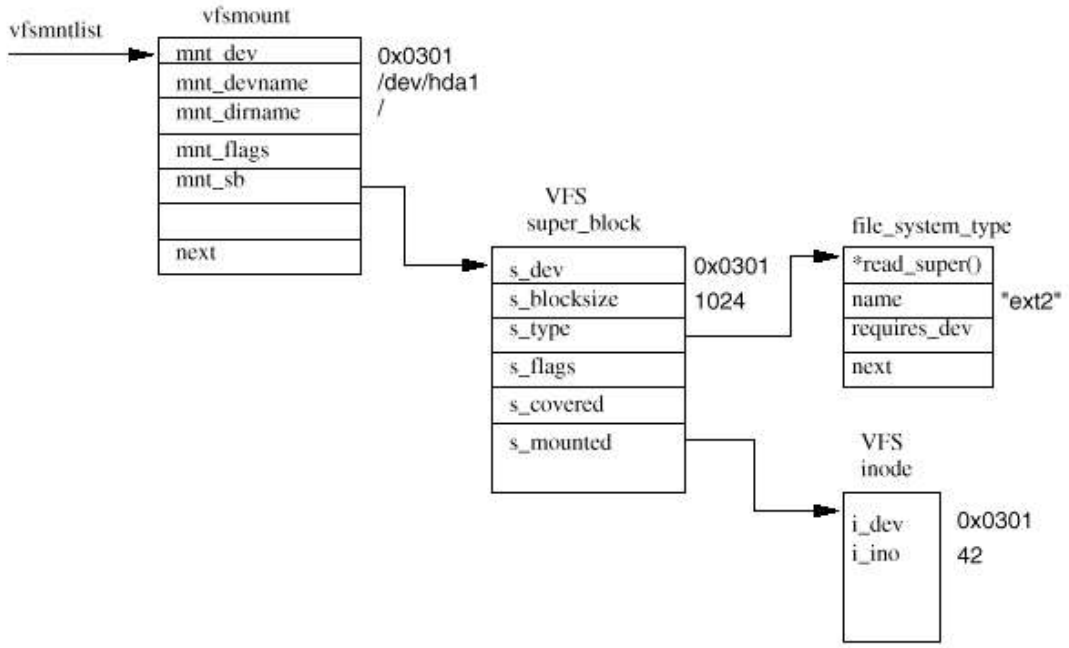



Figure 9.6: A Mounted File System

第二步，如果mount传递的物理设备还没有安装，必须找到即将成为新的文件系统的安装点的目录的VFS inode。这个VFS inode可能在inode cache或者必须从支撑这个安装点的文件系统的块设备上读取。一旦找到了这个inode，就检查它是否是一个目录，而且没有其他文件系统安装在那里。同一个目录不能用作多于一个文件系统的安装点。

这时，这个VFS安装代码必须分配以一个VFS超级块并传递安装信息给这个文件系统的超级块读取例程。系统所有的VFS超级块都保存在super_block数据结构组成的super_blocks向量表中，必须为这次安装分配一个结构。超级块读取例程必须根据它从物理设备读取得信息填充VFS超级块的域。对于EXT2文件系统而言，这种信息的映射或者转换相当容易，它只需要读取EXT2的超级块并填到VFS超级块。对于其他文件系统，例如MS DOS文件系统，并不是这么简单的任务。不管是什么文件系统，填充VFS超级块意味着必须从支持它的块设备读取描述该文件系统的信息。如果块设备不能读取或者它不包含这种类型的文件系统，mount命令会失败。

每一个安装的文件系统用一个vfsmount数据结构描述，参见图9.6。它们在vfsmntlist指向的一个列表中排队。另一个指针， vfsmnttail指向列表中最后一个条目，而mru_vfsmnt指针指向最近使用的文件系统。每一个vfsmount结构包括存放这个文件系统的块设备的设备编号，文件系统安装的目录和一个指向这个文件系统安装时所分配的VFS超级块的指针。VFS超级块指向这一类型的文件系统的file_system_type数据结构和这个文件系统的root inode。这个inode在这个文件系统加载过程中一直驻留在VFS inode cache中。

参见fs/super.c add_vfsmnt()

### 9.2.5 Finding a File in the Virtual File System（在虚拟文件系统中查找一个文件）

为了找到一个文件在虚拟文件系统中的VFS inode，VFS必须依次名称，一次一个目录，找到中间的每一个的目录的VFS inode。每一个目录查找都要调用和文件系统相关的查找例程（地址放在代表父目录的VFS inode中）。因为在文件系统的VFS超级块中总是有文件系统的root inode，并在超级块中用指针指示，所以整个过程可以继续。每一次查找真实文件系统中的inode的时候，都要检查这个目录的目录缓存。如果目录缓存中没有这个条目，真实文件系统要么从底层的文件系统要么从inode cache中获得VFS inode。

### 9.2.6 Creating a File in the Virtual File System（在虚拟文件系统中创建一个文件）

### 9.2.7 Unmounting a File System（卸载一个文件系统）

我的工作手册通常把装配描述成为拆卸的反过程，但是对于卸载文件系统有些不同。如果系统有东西在使用文件系统的一个文件，那么这个文件系统就不能被卸载。例如，如果一个进程在使用/mnt/cdrom目录或它的子目录，你就不能卸载/mnt/cdrom。如果有东西在使用要卸载的文件系统，那么它的VFS inode会在VFS inode cache中。卸载代码检查整个inode列表，查找属于这个文件系统所占用的设备的inode。如果这个安装的文件系统的VFS超级块是dirty，就是它被修改过了，那么它必须被写回到磁盘上的文件系统。一旦它写了磁盘，这个VFS超级块占用的内存就被返回到核心的空闲内存池中。最后，这个安装的vmsmount数据结构也从vfsmntlist中删除并释放。

参见fs/super.c do_umount()

参见fs/super.c remove_vfsmnt()

## 8. The VFS Inode Cache

当游历安装的文件系统的时候，它们的VFS inode不断地被读取，有时是写入。虚拟文件系统维护一个inode cache，用于加速对于所有安装的文件系统的访问。每一次从inode cache读出一个VFS inode，系统就可以省去对于物理设备的访问。

参见fs/inode.c

VFS inode cache用散列表（hash table）的方式实现，条目是指针，指向既有同样hash value的VFS inode 列表。一个inode的hash value从它的inode 编号和包括这个文件系统的底层的物理设备的设备编号中计算出来。不论何时虚拟文件系统需要访问一个inode，它首先查看VFS inode cache。为了在inode hash table中查找一个inode，系统首先计算它的hash value，然后用它作为inode hash table的索引。这样得到了具有相同hash value的inode的列表的指针。然后它一次读取所有的inode直到它找到和它要找的inode具有相同的inode编号

和相同的设备标识符的inode为止。

如果可以在cache中找到这个inode，它的count就增加，表示它有了另一个用户，文件系统的访问继续进行。否则必须找到一个空闲的VFS inode让文件系统把inode读入到内存。如何得到一个空闲的inode，VFS有一系列选择。如果系统可以分配更多的VFS inode，它就这样做：它分配核心页并把它们分成新的、空闲的inode，放到inode列表中。系统中所有的VFS inode除了在inode hash table中之外也在一个由first_inode指向的列表。如果系统已经拥有了它允许有的所有的inode，它必须找到一个可以重用的inode。好的候选是哪些使用量（count）是0的inode：这表示系统当前没有使用它们。真正重要的VFS inode，例如文件系统的root inode，已经有了一个大于0的使用量，所以永远不会选做重用。一旦定位到一个重用的候选，它就被清除。这个VFS inode可能是脏的，这时系统必须等待它被解锁然后才能继续。在重用之前这个VFS inode的候选必须被清除。

虽然找到了一个新的VFS inode，还必须调用一个和文件系统相关的例程，用从底层的真正的文件系统中毒取得信息填充这个inode。当它填充的时候，这个新的VFS inode的使用量为1，并被锁定，所以在它填入了有效的信息之前除了它没有其它进程可以访问。

为了得到它实际需要的VFS inode，文件系统可能需要访问其它一些inode。这发生在你读取一个目录的时候：只有最终目录的inode是需要的，但是中间目录的inode也必须读取。当VFS inode cache使用过程并填满时，较少使用的inode会被废弃，较多使用的inode会保留在高速缓存中。

## 9. The Directory Cache（目录缓存）

为了加速对于常用目录的访问，VFS维护了目录条目的一个高速缓存。当真正的文件系统查找目录的时候，这些目录的细节就被增加到了目录缓存中。下一次查找同一个目录的时候，例如列表或打开里边的文件，就可以在目录缓存中找到。只有短的目录条目（最多15字符）被缓存，不过这是合理的，因为较短的目录名称是最常用的。例如：当X服务器启动的时候，/usr/X11R6/bin非常频繁地被访问。

参见fs/dcache.c

目录缓存中包含一个hash table，每一个条目都指向一个具有相同的hash value的目录缓存条目的列表。Hash 函数使用存放这个文件系统的设备的设备编号和目录的名称来计算在hash table中的偏移量或索引。它允许快速找到缓存的目录条目。如果一个缓存在查找的时候花费时间太长，或根本找不到，这样的缓存是没有用的。

为了保持这些cache有效和最新，VFS保存了一个最近最少使用（LRU）目录缓存条目的列表。当一个目录条目第一次被放到了缓存，就是当它第一次被查找的时候，它被加到了第一级LRU列表的最后。对于充满的cache，这会移去LRU列表前面存在的条目。当这个目录条目再一次被访问的时候，它被移到了第二个LRU cache列表的最后。同样，这一次它移去了第二级LRU cache列表前面的二级缓存目录条目。这样从一级和二级LRU列表中移去目录条目是没有问题的。这些条目之所以在列表的前面只是因为它们最近没有被访问。如果被访问，它们会在列表的最后。在二

级LRU缓存列表中的条目比在一级LRU缓存列表中的条目更加安全。因为这些条目不仅被查找而且曾经重复引用。

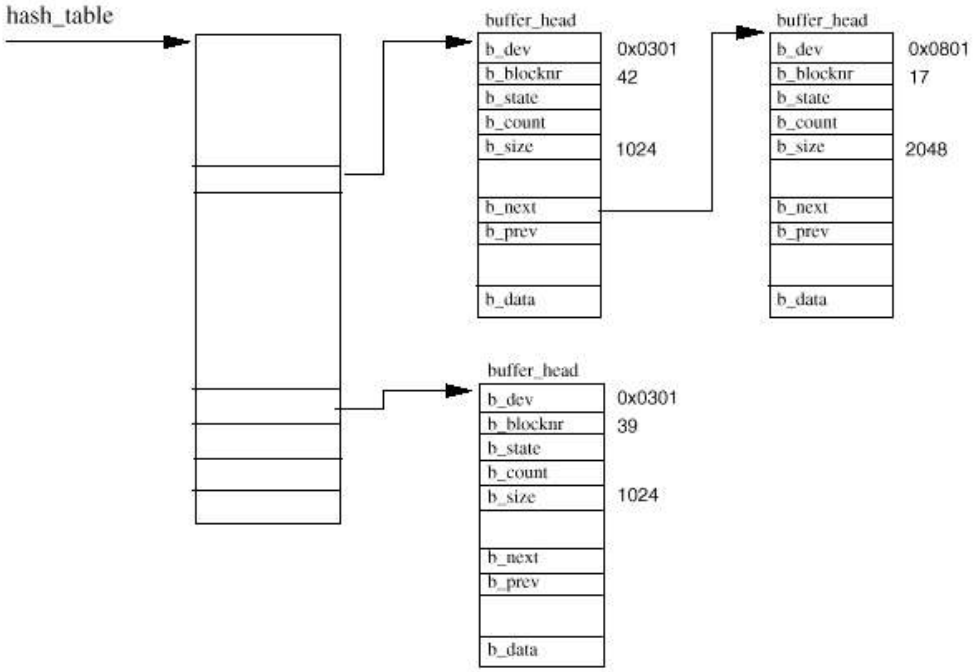



Figure 9.7: The Buffer Cache

## 2. The Buffer Cache

当使用安装的文件系统的时候，它们会对块设备产生大量的读写数据块的请求。所有的块数据读写的请求都通过标准的核心例程调用，以buffer_head数据结构的形式传递给设备驱动程序。这些数据结构给出了设备驱动程序需要的所有信息：设备标识符唯一标识了设备，块编号告诉了驱动程序读去哪一块。所有的块设备被看成同样大小的块的线性组合。为了加速对于物理块设备的访问，Linux维护了一个块缓冲区的缓存。系统中所有的块缓冲区动保存在这个buffer cache，甚至包括那些新的、未使用的缓冲区。这个缓存区被所有的物理块设备共享：任何时候缓存区中都有许多块缓冲区，可以属于任何一个系统块设备，通常具有不同的状态。如果在buffer cache中有有效的数据，这就可以节省系统对于物理设备的访问。任何用于从/向块设备读取/写入数据的块缓冲区都进入这个buffer cache。随着时间的推移，它可能从这个缓存区中删除，为其它更需要的缓冲区让出空间，或者如果它经常被访问，就可能一直留在缓存区中。

这个缓存区中的块缓冲区用这个缓冲区所属的设备标识符和块编号唯一标识。这个buffer cache由两个功能部分组成。第一部分是空闲的块缓冲区列表。每一个同样大小的缓冲区（系统可以支持的）一个列表。系统的空闲的块缓冲区当第一次创建或者被废弃的时候就在这些列表中排队。当前支持的缓冲区大小是512、1024、2048、4096和8192字节。第二个功能部分是缓存区（cache）本身。这是一个hash table，是一个指针的向量表，用于链接具有相同hash index的buffer。Hash index从数据块所属的设备标识符和块编号产生出来。图9.7显示了这个hash table和一些条目。块缓冲区要么在空闲列表之一，要么在buffer cache中。当它们在buffer cache的时候，它们也在LRU列表中排队。每一个缓冲区类型一个LRU列表，系统使用这些类型在一

种类型的缓冲区上执行操作。例如，把有新数据的缓冲区写到磁盘上。缓冲区的类型反映了它的状态，Linux当前支持以下类型：

clean 未使用，新的缓冲区（buffer）

locked 锁定的缓冲区，等待被写入

dirty 脏的缓冲区。包含新的有效的数据，将被写到磁盘，但是直到现在还没有调度到写

shared 共享的缓冲区

unshared 曾经共享的缓冲区，但是现在没有共享

不论何时文件系统需要从它的底层的物理设备读取一个缓冲区的时候，它都试图从buffer cache中得到一个块。如果它不能从buffer cache中得到一个缓冲区，它就从适当大小的空闲列表中取出一个干净的缓冲区，这个新的缓冲区会进入到buffer cache中。如果它需要的缓冲区已经在buffer cache中，那么它可能是也可能不是最新。如果它不是最新，或者它是一个新的块缓冲区，文件系统必须请求设备驱动程序从磁盘上读取适当的数据块。

象所有的高速缓存一样，buffer cache必须被维护，这样它才能有效地运行，并在使用buffer cache的块设备之间公平地分配缓存条目。Linux使用核心守护进程bdflush在这个缓存区上执行大量的整理工作，不过另一些是在使用缓存区的过程中自动进行的。

### 9.3.1 The bdflush Kernel Daemon（核心守护进程bdflsuh）

核心守护进程bdflush是一个简单的核心守护进程，对于有许多脏的缓冲区（包含必须同时写到磁盘的数据的缓冲区）的系统提供了动态的响应。它在系统启动的时候作为一个核心线程启动，相当容易混淆，它叫自己位“kflushd”，而这是你用ps显示系统中的进程的时候你会看得的名字。这个进程大多数时间在睡眠，等待系统中脏的缓冲区的数目增加直到太巨大。当缓冲区分配和释放的时候，就检查系统中脏的缓冲区的数目，然后唤醒bdflush。缺省的阈值是60%，但是如果系统非常需要缓冲区，bdflush也会被唤醒。这个值可以用updage命令检查和设置：

#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).

不管什么时候写入了数据，成为了脏的缓冲区，所有的脏的缓冲区都链接在BUF_DIRTY LRU列表中，bdflush会尝试把合理数目的缓冲区写到它们的磁盘中。这个数目也可以用update命令检查和设置，缺省是500（见上例）。

9.3.2 The update Process（update 进程）

update 命令不仅仅是一个命令，它也是一个守护进程。当以超级用户身份（系统初始化）运行的时候，它会定期把所有旧的脏缓冲区写到磁盘上。它通过调用系统服务例程执行这些任务，或多或少和bdflush的任务相同。当生成了一个脏缓冲区的时候，都标记上它应该被写到它自己的磁盘上的系统时间。每一次update运行的时候，它都查看系统中所有的脏的缓冲区，查找具有过期的写时间的缓冲区。每一个过期的缓冲区都被写到磁盘上。

参见fs/buffer.c sys_bdflush()

3. The /proc File System

/proc文件系统真实地体现了Linux虚拟文件系统的能力。它实际上并不存在（这也是Linux另外一个技巧），/proc和它的子目录以及其中的文件都不存在。但是为什么你可以cat /proc/devices？ /proc文件系统，象一个真正的文件系统一样，也向虚拟文件系统登记自己，但是当它的文件和目录被打开，VFS执行调用请求它的inode的时候，/proc文件系统才利用核心中的信息创建这些文件和目录。例如，核心的/proc/devices文件是从核心描述它的设备的数据结构中产生出来的。

/proc文件系统代表了一个用户可读的窗口，进入核心的内部工作空间。一些Linux子系统，例如第12章描述的Linux核心模块，都在/proc文件系统中创建条目。

### 4. Device Special Files

Linux，象所有版本的Unix一样，把它的硬件设备表示成为特殊文件。例如，/dev/null是空设备。设备文件不在文件系统中占用任何数据空间，它只是设备驱动程序的一个访问点。EXT2文件系统和Linux的VFS都把设备文件作为特殊类型的inode。有两种类型的设备文件：字符和块特殊文件。在核心内部本身，设备驱动程序都实现文件的基本操作：你可以打开、关闭等等。字符设备允许字符模式的I/O操作，而块设备要求所有的I/O通过buffer cache。当对于一个设备文件执行一个I/O请求的时候，它被转到系统中适当的设备驱动程序。通常这不是一个真正的设备驱动程序，而是一个用于子系统的伪设备驱动程序（pseudo-device driver）例如SCSI设备驱动程序层。设备文件用主设备编号（标识设备类型）和次类型来引用（用于标识单元，或者主类型的实例）。例如，对于系统中的第一个IDE控制器上的IDE磁盘，主设备编号是3，IDE磁盘的第一个分区的次设备编号应该是1，所以，ls –l /dev/hda1输出

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

参见/include/linux/major.h中所有Linux的主设备编号

在核心中，每一个设备用一个kdev_t数据类型唯一描述。这个类型有两个字节长，第一个包括设备的次设备编号，第二个包括主设备编号。上面的IDE设备在核心中保存为0x0301。代表一个块或者字符设备的EXT2 inode把设备的主和次设备号放在它的第一个直接块指针那里。当它被VFS读取的时候，代表它的VFS inode数据结构的I_rdev域被设成正确的设备标识符。

参见include/linux/kdev_t.h

## Chapter 10

## Networks（网络）

Linux和网络几乎是同义词。实际上Linux是Internet或WWW的产物。它的开发者和用户使用web交换信息、想法、代码而Linux自身也常用于支持一些组织的联网需求。本章描述了Linux如何支持统称为TCP/IP的网络协议。

TCP/IP协议设计用来支持连接在ARPANET上的计算机之间的通讯。ARPANET是美国政府投资的一个美国的研究网络。ARPANET是一些网络概念的先驱，例如报文交换和协议分层，让一种协议利用其它协议提供的服务。ARPANET于1988年退出，但是它的后继者（NSF NET和Internet）发展的甚至更大。现在所知的World Wide Web是在ARPANET中发展的，它本身也是由TCP/IP协议支持的。Unix在ARPANET上大量使用，第一个发布的网络版的Unix是4.3BSD。

Linux的网络实现是基于4.3BSD的模型，它支持BSD socket（和一些扩展）和全系列的TCP/IP网络功能。选择这种编程接口是因为它的流行程度，而且可以帮助程序在Linux和其它Unix平台之间移植。

## 10.1 An Overview of TCP/IP Networking（TCP/IP网络概览）

本节为TCP/IP网络的主要原理给出了一个概览。这并不是一个详尽的描述。要更详细的描述，阅读第10本参考书（附录）。

在一个IP网络中，每一个机器都分配一个IP地址，这是一个32位的数字，唯一标识这一台机器。WWW是一个非常巨大、不断增长的IP网络，每一个连接在上面的机器都分配了一个独一无二的IP地址。IP地址用点分隔的四个数字表示，例如，16.42.0.9。IP地址实际上分为两个部分：网络地址和主机地址。这些地址的大小（尺寸）可能不同（有几类IP地址），以16.42.0.9为例，网络地址是16.42，主机地址是0.9。主机地址可以进一步划分成为子网（subnetwork）和主机地址。再次以16.42.0.9为例，子网地址可以是16.42.0，主机地址为16.42.0.9。对于IP地址进行进一步划分允许各个组织划分它们自己的网络。例如，假设16.42是ACME计算机公司的网络地址，16.42.0可以是子网0，16.42.1可以是子网1。这些子网可以在分离的大楼里，也许通过电话专线或者甚至通过微波连接。IP地址由网络管理员分配，使用IP子网是分散网络管理任务的一个好办法。IP子网的管理员可以自由地分配他们自己子网内的IP地址。

但是，通常IP地址难于记忆，而名字更容易记忆。Linux.acme.com比16.42.0.9更好记。必须使用一种机制把网络名字转换为IP地址。这些名字可以静态地存在/etc/hosts文件中或者让Linux询问一个分布式命名服务器（Distributed Name Server DNS）来解析名字。这种情况下，本地主机必须知道一个或多个DNS服务器的IP地址，在/etc/resolv.conf中指定。

不管什么时候你连接另外一台机器的时候，比如读取一个web page，都要使用它的IP地址和那台机器交换数据。这种数据包括在IP报文（packet）中，每一个报文都有一个IP头（包括源和目标机器的IP地址，一个校验和和其它有用的信息。这个校验和是从IP报文的数据中得到的，可以让IP报文的接收者判断传输过程中IP报文是否损坏（可能是一个噪音很大的电话线）。应用程序传输的数据可能被分解成容易处理的更小的报文。IP数据报文的大小依赖于连接的介质而变化：以太网报文通常大于PPP报文。目标主机必须重新装配这些数据报文，然后才能交给接收程序。如果你通过一个相当慢的串行连接访问一个包括大量图形图像的web页，你就可以用图形的方式看出数据的分解和重组。

连接在同一个IP子网的主机可以互相直接发送IP报文，而其它的IP报文必须通过一个特殊的主机（网关）发送。网关（或路由器）连接在多于一个子网上，它们会把一个子网上接收的IP报文重新发送到另一个子网。例如，如果子网16.42.1.0和16.42.0.0通过一个网关连接，那么所有从子网0发送到子网1的报文必须先发送到网关，这样才能转发。本地的主机建立一个路由表，让它可以把要转发的IP报文发送到正确的机器。对于每一个IP目标，在路由表中都有一个条目，告诉Linux要到达目标需要先把IP报文发送到那一台主机。这些路由表是动态的，而且当应用程序使用网络

和网络拓扑变化的时候不断改变。

IP协议是传输层协议，被其他协议使用，携带它们的数据。传输控制协议（TCP）是一个可靠的端到端的协议，使用IP传送和接收它的报文。象IP报文有自己的头一样，TCP也有自己的头。TCP是一个面向连接的协议，两个网络应用程序通过一个虚拟的连接连接在一起，甚至它们中间可能会有许多子网、网关和路由器。TCP在两个应用程序之间可靠地传送和接收数据，并且保证不会有丢失和重复的数据。当TCP使用IP传送它的报文的时候，在IP报文中包含的数据就是TCP报文自身。每一个通讯的主机的IP层负责传送和接收IP报文。用户数据报协议（UDP）也使用IP层传送它的报文，但是不象TCP，UDP不是一个可靠的协议，它只提供数据报服务。其它协议也可以使用IP意味着当接收到IP报文，接收的IP层必须知道把这个IP报文中包含的数据交给哪一个上层协议。为此，每一个IP报文的头都有一个字节，包含一个协议标识符。当TCP请求IP层传输一个IP报文的时候IP报文的头就说明它包含一个TCP报文。接收的IP层，使用这个协议标识符来决定把接收到的数据向上传递给哪一个协议，在这种情况下，是TCP层。当应用程序通过TCP/IP通讯的时候，它们不但必须指定目标的IP地址，也要指定目标应用程序的端口（port）地址。一个端口地址唯一标识一个应用程序，标准的网络应用程序使用标准的端口地址：例如web服务器使用端口80。这些已经注册的端口地址可以在/etc/services中查到。

协议分层不仅仅停留在TCP、UDP和IP。IP协议本身使用许多不同的物理介质和其它IP主机传输IP报文。这些介质自己也可能增加它们自己的协议头。这样的例子有以太网层、PPP和SLIP。一个以太网允许许多主机同时连接在一个物理电缆上。每一个传送的以太帧可以被所有连接的主机看到，所以每一个以太网设备都有一个独一无二的地址。每一个传送到那个地址的以太网帧会被那个地址的主机接收，而被连接到这个网络的其它主机忽略掉。这个独一无二的地址当每一个以太网设备制造的时候内建在设备里边，通常保存在以太网卡的SROM中。以太地址由6个字节长，例如，可能是08-00-2b-00-49-4A。一些以太网地址保留用于多点广播，用这种目标地址发送的以太网帧会被网络上的所有的主机接收。因为以太网帧中可能运载许多不同的协议（作为数据），和IP报文一样，它们的头中都包含一个协议标识符。这样以太网层可以正确地接收IP报文并把数据传输到IP层。

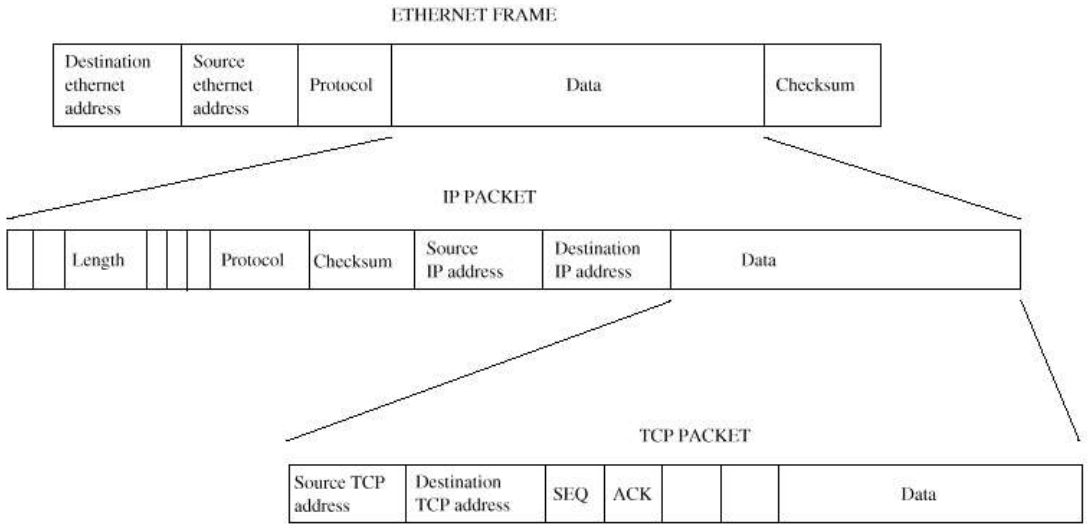



Figure 10.1: TCP/IP Protocol Layers

为了通过多种连接协议，例如通过以太网来传输IP报文，IP层必须找出这个IP主机的以太网地址。这是因为IP地址只是一个寻址的概念，以太网设备自己有自己的物理地址。IP地址可以由网络管理员根据需要分配和再分配，而网络硬件则只响应具有它自己物理地址的以太网帧，或者特

殊的多点广播地址（所有的机器都必须接收）。Linux使用地址解析协议（ARP）让机器把IP地址转换成真实的硬件地址例如以太网地址。为了得到一个IP地址所联系的硬件地址，一个主机会发送一个ARP请求包，包含它希望转换的IP地址，发送到一个多点广播地址，让网络上所有的点都可以收到。具有这个IP地址的目标主机用一个ARP回应来应答，这中间包括了它的物理硬件地址。APR不仅仅限制在以太网设备，它也可以解析其它物理介质的IP地址，例如FDDI。不能进行ARP的设备会有标记，这样Linux就不需要试图对它们进行ARP。也有一个相反的功能，反向ARP，或RARP，把物理地址转换到IP地址。这用于网关，回应对于代表远端网络的IP地址的ARP请求。

## 10.2 The Linux TCP/IP Networking Layers（Linux TCP/IP网络分层）

象网络协议一样，图10.2显示了Linux对于internet 协议地址族的实现就好像一系列连接的软件层。BSD socket由只和BSD socket相关的通用的socket管理软件来支持。支持这些的是INET socket层，它管理以IP为基础的协议TCP和UDP的通讯端点。UDP是一个无连接的协议，而TCP是一个可靠的端到端的协议。当传送UDP报文的时候，Linux不知道也不关心它们是否安全到达目的地。TCP报文进行了编号，TCP连接的每一端都要确保传送的数据正确地接收到。IP层包括了网际协议（Internet Protocol）的代码实现。这种代码在传送的数据前增加IP头，而且知道如何把进来的IP报文转送到TCP或者UDP层。在IP层之下，支持Linux联网的是网络设备，例如PPP和以太网。网络设备并非总是表现为物理设备：其中一些比如loopback设备只是纯粹的软件设备。不象标准的Linux设备用mknod命令创建，网络设备只有在底层的软件找到并且初始化它们之后才出现。你只有在建立俄一个包含恰当的以太望设备驱动程序的核心之后你才能看到设备文件/dev/eth0。ARP协议位于IP层和支持ARP的协议之间。

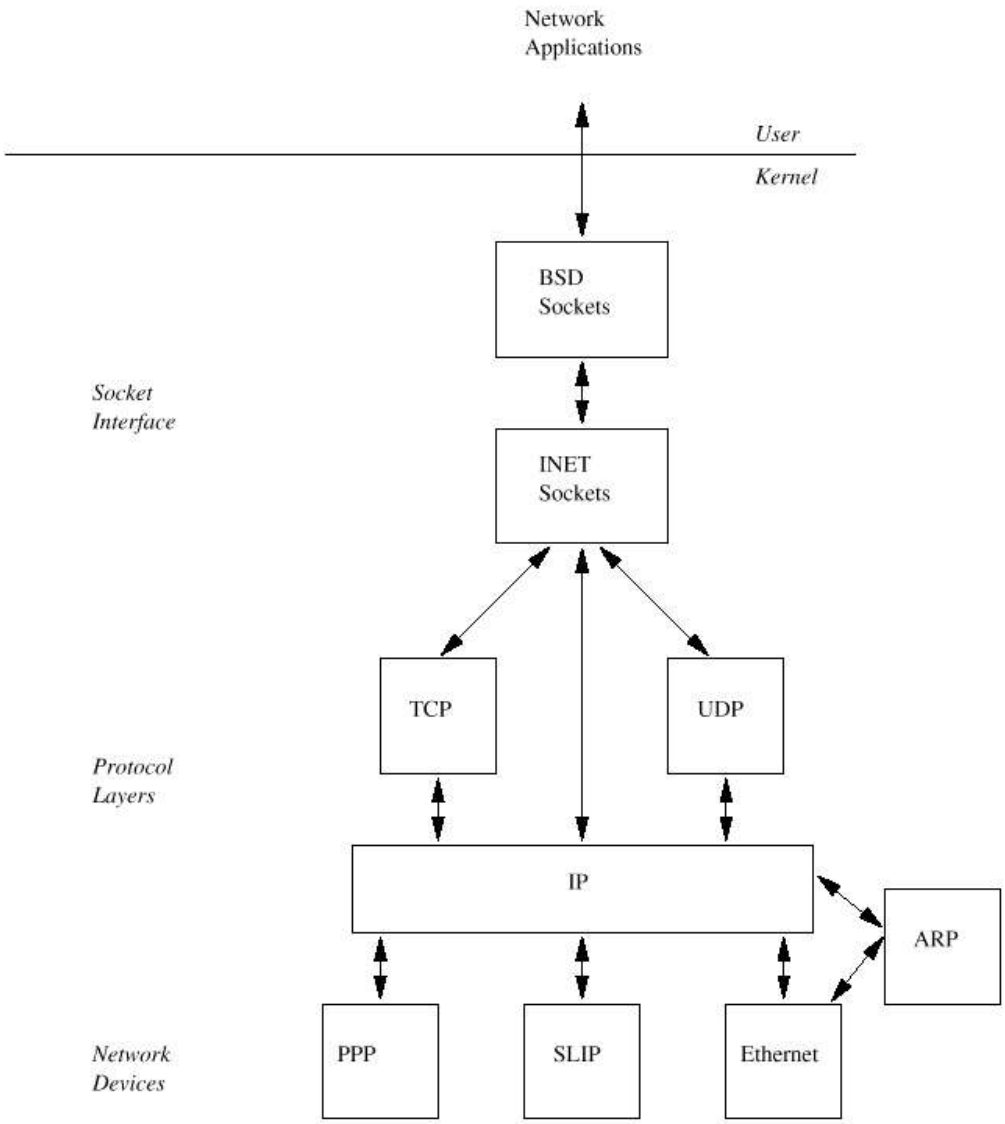



Figure 10.2: Linux Networking Layers

## 10.3 The BSD Socket Interface（BSD socket 接口）

这是一个通用的接口，不仅仅支持多种形式的联网，也是一种进程间通讯机制。一个socket描述了通讯连接的一端，两个通讯进程每一个都会有一个socket，描述它们之间通讯连接的自己部分。Socket可以想象成一种特殊形式的管道，但是和管道不同，socket对于可以容纳的数据量没有限制。Linux支持几种类型的socket，这些类叫做address families（地址族）。这是因为每一类都有自己通讯寻址方式。Linux支持以下socket address families或domain：

UNIX Unix domain sockets,

INET The Internet address family supports communications via

TCP/IP protocols

AX25 Amateur radio X25

IPX Novell IPX

APPLETALK Appletalk DDP

X25 X25

有几种socket类型，每一种都代表了连接上支持的服务的类型。并非所有的address families都支持所有类型的服务。Linux BSD socket支持以下socket类型。

Stream 这种socket提供了可靠的、双向顺序的数据流，保证传输过程中数据不会丢失、损坏或重复。Stream socket在INET address family中由TCP协议支持

Datagram 这种socket也提供了双向的数据传输，但是和stream socket不同，它不保证消息会到达。甚至它到达了也不保证它们会顺序到达或没有重复或损坏。这种类型的socket在Internet address family中由UDP协议支持。

RAW 这允许进程直接（所以叫“raw”）访问底层的协议。例如，可以向一个以太网设备打开一个raw socket，观察raw IP数据流。

Reliable Delivered Messages 这很象数据报但是数据保证可以到达

Sequenced Packets 象stream socket但是数据报文大小是固定的

Packet 这不是标准的BSD socket类型，它是Linux特定的扩展，允许进程直接在设备层访问报文

使用socket通讯的进程用一个客户服务器的模型。服务器提供服务，而客户使用这种服务。一个这样的例子是一个Web 服务器，提供web page和一个web 客户（或浏览器），读取这些页。使用socket的服务器，首先创建一个socket，然后为它bind一个名字。这个名字的格式和socket的address family有关，它是服务器的本地地址。Socket的名字或地址用sockaddr数据结构指定。一个INET socket会绑定一个IP端口地址。注册的端口编号可以在/etc/services中看到：例如，web服务器的端口是80。在socket上绑定一个地址后，服务器就listen进来的对于绑定的地址的连接请求。请求的发起者，客户，创建一个socket，并在上面执行一个连接请求，指定服务器的目标地址。对于一个INET socket，服务器的地址是它的IP地址和它的端口地址。这些进来的请求必须通过大量的协议层，找到它的路径，然后就在服务器的监听端口等待。一旦服务器接收到了进来的请求，它可以接受（accept）或者拒绝它。如果要接受进来的请求，服务器必须创建一个新的socket来接受它。一旦一个socket已经用于监听进来的连接请求，它就不能再用于支持一个连接。连接建立之后，两端都可以自由地发送和接收数据。最后，当一个连接不再需要的时候，它可以被关闭。必须小心，保证正确地处理正在传送的数据报文。

一个BSD socket上的操作的确切意义依赖于它底层的地址族。建立一个TCP/IP连接和建立一个业余无线电X.25连接有很大的不同。象虚拟文件系统一样，Linux在和独立的地址族相关的软件所支

持的BSD socket层抽象了BSD socket和应用程序之间的socket接口。当核心初始化的时候，建立在核心的地址族就向BSD socket接口登记自己。稍后，当应用程序创建和使用BSD socket的时候，在BSD socket和它的支撑地址族之间建立一个联系。这种联系是通过交叉的数据结构和地址族支持例程表实现的。例如，当应用程序创建一个新的socket的时候，BSD socket接口就使用地址族相关的socket创建例程。

当配置核心的时候，一组地址族和协议都建立到了protocols向量表中。每一个都用它的名称（例如“INET”）和它的初始化例程的地址来代表。当启动的时候，socket接口初始化，每一个协议的初始化代码都要被调用。对于socket地址族，它们里边会登记一系列协议操作。这都是一些例程，每一个都执行一个和地址族相关的特殊操作。登记的协议操作保存在pops向量表中，这个向量表保存指向proto_ops数据结构的指针。Proto_ops数据结构包括协议族类型和一批和特定地址族相关的socket操作例程的指针。Pops向量表用地址族的标识符作为索引，例如Internet address family的标识符（AF_INET是2）。

参见include/linux/net.h

## 10.4 The INET Socket Layer

INET socket层支持包含TCP/IP协议的internet address family。象上面讨论的，这些协议是分层的，每一个协议都使用其它协议的服务。Linux的TCP/IP代码和数据结构反映了这种分层。它和BSD socket层的接口是通过网络初始化的时候它向BSD socket层登记的internet address family socket操作进行的。这些和其它登记的地址族一起放在pops向量表中。BSD socket层通过调用在登记的proto_ops数据结构中的INET层的socket支持例程完成它的工作。例如，一个地址族是INET的BSD socket创建请求会使用底层的INET socket创建函数。每一次操作BSD socket层都把代表BSD socket的socket数据结构传递给INET层。INET socket层使用它自己的数据结构socket，连接到BSD socket数据结构，而不是用TCP/IP相关的信息把BSD socket搞乱。这种连接参见图10.3。它使用BSD socket中的data指针把sock数据结构和BSD socket数据结构连接起来。这意味着后续的INET socket调用可以很容易地获取这个sock 数据结构。在创建的时候sock数据结构的协议操作指针也被建立，这些指针依赖于请求的协议。如果请求TCP，则sock数据结构的协议操作指针会指向TCP连接所需要的一系列TCP协议的操作。

参见include/net/sock.h

### 10.4.1 Creating a BSD Socket（创建一个BSD Socket）

创建一个新的socket的系统调用需要传递它的地址族的标识符、socket的类型和协议。首先，用请求的地址族在pops向量表中查找一个匹配的地址族。它可能是一个使用核心模块实现的特殊的地址族，如果这样，kerneld核心进程必须加载这个模块，我们才能继续。然后分配一个新的socket数据结构来表示这个BSD socket。实际上这个socket数据结构物理上是VFS inode数据结构的一部分，分配一个socket实际上就是分配一个VFS inode。这看起来比较奇怪，除非你考虑

让socket可以用和普通文件一样的方式进行操作。象所有文件都用VFS inode 数据结构表示一样，为了支持文件操作，BSD socket也必须用一个VFS inode数据结构表示。

这个新创建的BSD socket数据结构包括一个指针指向和地址族相关的socket例程，这个指针被设置到从pops向量表中取出的proto_ops数据结构。它的类型被设置成请求的socket类型：SOCK_STREAM、SOCK_DGRAM等等其中之一，然后用proto_ops数据结构中保存的地址调用和地址族相关的创建例程。

然后从当前进程的fd向量表中分配一个空闲的文件描述符，它所指向的file数据结构也被初始化。这包括设置文件操作指针，指向BSD socket接口支持的BSD socket文件操作例程。所有将来的操作会被定向到socket接口，依次通过调用支撑的地址族的操作例程传递到相应的地址族。

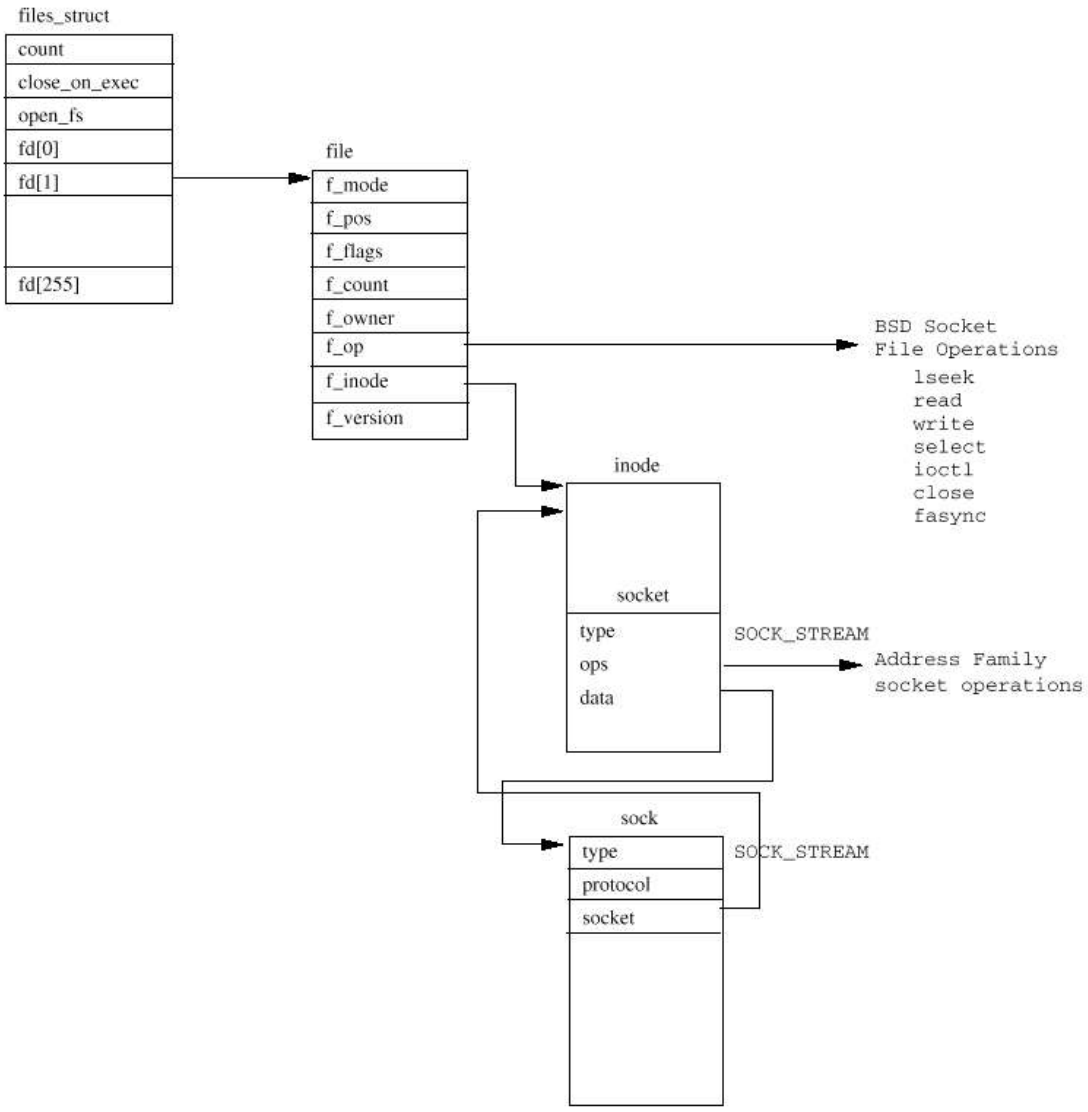



Figure 10.3: Linux BSD Socket Data Structures

## 10.4.2 Binding an Address to an INET BSD Socket（为一个INET BSD socket绑定一个地址）

为了监听进来的网际连接请求，每一个服务器必须创建一个INET BSD socket并把自己的地址绑定到它上面。Bind的操作大部分由INET socket 层处理，另一些需要底层的TCP和UDP协议层的支持。已经绑定了一个地址的socket不能用于其它通讯。这意味着这个socket的状态必须是TCP_CLOSE。传递给bind操作的sockaddr包括要绑定的IP地址和一个端口号（可选）。通常，绑定的地址会是分配给支持INET地址族的网络设备的地址之中的一个，而且接口必须是开启的并能够使用。你可以用ifconfig命令看系统中哪一个网络接口当前是激活的。IP地址也可以是IP广播地址（全是1或0）。这是意味着“发送给每一个人”的特殊地址。如果这个机器作为一个透明的proxy或者防火墙，这个IP地址也可以设置成任何IP地址。不过只有具有超级用户特权的进程可以绑定任意IP地址。这个绑定的IP地址被存在sock数据结构的recv_addr和saddr域中。它们分别用于hash查找和发送IP地址。端口号是可选的，如果没有设置，会向支撑的网络请求一个空闲的。按照惯例，小于1024的端口号不能被没有超级用户特权的进程使用。如果底层的网络分配端口号，它总是分配一个大于1024的端口。

当底层的网络设备接收报文的时候，这些报文必须被转到正确的INET和BSD socket才能被处理。为此，UDP和TCP维护hash table，用于查找进来的IP信息的地址，把它们转到正确的socket/sock对。TCP是一个面向连接的协议，所以处理TCP报文比处理UDP报文所包括的信息要多。

UDP维护一个已经分配的UDP端口的hash table，udp_table。这包括sock数据结构的指针，用一个根据端口号的hash函数作为索引。因为UDP hash table比允许的端口号要小的多（udp_hash只有128，UDP_HTABLE_SIZE）表中的一些条目指向一个sock数据结构的链表，用每一个sock的next 指针连接在一起。

TCP更加复杂，因为它维护几个hast table 。但是，在绑定操作中，TCP实际上并不把绑定的sock数据结构加到它的hash table中，它只是检查请求的端口当前没有被使用。在listen操作中sock数据结构才加到TCP的hash table中。

### 10.4.3 Making a Connection to an INET BSD Socket

一旦创建了一个socket，如果没有用于监听进来的连接请求，它就可以用于建立向外的连接请求。对于无连接的协议，比如UDP，这个socket操作不需要做许多，但是对于面向连接的协议如TCP，它涉及在两个应用程序之间建立一个虚拟电路。

一个向外的连接只能在一个正确状态的INET BSD socket上进行：就是说还没有建立连接，而且没有用于监听进来的连接。这意味着这个BSD socket数据结构必须在SS_UNCONNECTED状态。UDP协议不在两个应用程序之间建立虚拟连接，所有发送的消息都是数据报，发出的消息可能到到也可能没有到达它的目的地。但是，它也支持BSD socket的connect操作。在一个UDP

INET BSD socket上的一个连接操作只是建立远程应用程序的地址：它的IP地址和它的IP端口号。另外，它也要建立一个路由表条目的缓存区，这样，在这个BSD socket上发送的UDP数据报不需要在检查路由表数据库（除非这个路由变成无效）。这个缓存的路由信息被INET sock数据结构中的ip_route_cache指针指向。如果没有给出地址信息，这个BSD socket发送的消息就自动使用这个缓存的路由和IP地址信息。UDP把sock的状态改变成为TCP_ESTABLISHED。

对于在一个TCP BSD socket上进行的连接操作，TCP必须建立一个包括连接信息的TCP消息，并发送到给定的IP目标。这个TCP消息包括连接的信息：一个独一无二的起始消息顺序编号、发起主机可以管理的消息的最大尺寸、发送和接收的窗口大小等等。在TCP中，所有的消息都编了号，初始顺序编号用作第一个消息编号。Linux选择一个合理的随机数以避免恶意的协议攻击。每一个从TCP连接的一端发送，被另一端成功接收的消息被确认，告诉它成功地到达，而且没有损坏。没有确认的消息会被重发。发送和接收窗口大小是确认前允许的消息的数目。如果接收端的网络设备支持的最大消息尺寸比较小，则这个连接会使用两个中间最小的一个。执行向外的TCP连接请求的应用程序现在必须等待目标应用程序的响应，是接受还是拒绝这个连接请求。对于期望进来的消息的TCP sock，它被加到了tcp_listening_hash，这样进来的TCP消息可以定向到这个sock数据结构。TCP也启动计时器，这样如果目标应用程序对于请求不响应，向外的连接请求会超时。

### 10.4.4 Listening on an INET BSD Socket

一旦一个socket拥有了一个绑定的地址，它就可以监听指定这个绑定地址的进来的连接请求。一个网络应用程序可以不绑定地址直接在一个socket上监听，这种情况下，INET socket层找到一个未用的端口号（对于这种协议而言），自动把它绑定到这个socket上。这个socket的listen函数把socket变成TCP_LISTEN的状态，并且执行所需的和网络相关的工作，一边允许进来的连接。

对于UDP socket，改变socket的状态已经足够，但是TCP已经激活它现在要把socket的sock数据结构加到它的两个hash table中。这是tcp_bound_hash和tcp_listening_hash 表。这两个表都通过一个基于IP端口号的hash函数进行索引。

不论何时接收到一个对于激活的监听socket的进来的TCP连接请求，TCP都要建立一个新的sock数据结构表示它。这个sock数据结构在它最终被接受之前成为这个TCP连接的buttom half。它也克隆包含连接请求的进来的sk_buff并把它排在监听的sock数据结构的receive_queue队列中。这个克隆的sk_buff包括一个指针，指向这个新创建的sock数据结构。

### 10.4.5 Accepting Connection Requests

UDP不支持连接的概念，接受INET socket的连接请求只应用于TCP协议，在一个监听的sock上进行接受（accept）操作会从原来的监听的socket克隆出一个新的socket数据结构。然后这个accept操作传递给支撑的协议层，在这种情况下，是INET去接受任何进来的连接请求。如果底层的协议，比如UDP不支持连接，INET协议层的accept操作会失败。否则，连接的请求会传递到真正的协议，在这里，是TCP。这个accept操作可能是阻塞，也可能是非阻塞的。在非阻塞的情况下，如果没有需要accept的进来的连接，这个accept操作会失败，而新创建的socket数据结构会被废弃。在阻塞的情况下，执行accept操作的网络应用程序会被加到一个等待队列，然后挂起，直到接收到一个TCP的连接请求。一旦接收到一个连接请求，包含这个请求的sk_buff会被废弃，这个sock数据结构被返回到INET socket层，在这里它被连接到先前创建的新的socket数据结构。这个新的socket的文件描述符（fd）被返回给网络应用程序，应用程序就可以用这个文件描述符对这个新创建的INET BSD socket进行socket操作。

## 10.5 The IP Layer（IP层）

### 10.5.1 Socket Buffers

使用分成许多层，每一层使用其它层的服务，这样的网络协议的一个问题是，每一个协议都需要在传送的时候在数据上增加协议头和尾，而在处理接收的数据的时候需要删除。这让协议之间传送数据缓冲区相当困难，因为每一层都需要找出它的特定的协议头和尾在哪里。一个解决方法是在每一层都拷贝缓冲区，但是这样会没有效率。替代的，Linux使用socket 缓冲区或者说sock_buffs在协议层和网络设备驱动程序之间传输数据。Sk_buffs包括指针和长度域，允许每一协议层使用标准的函数或方法操纵应用程序数据。

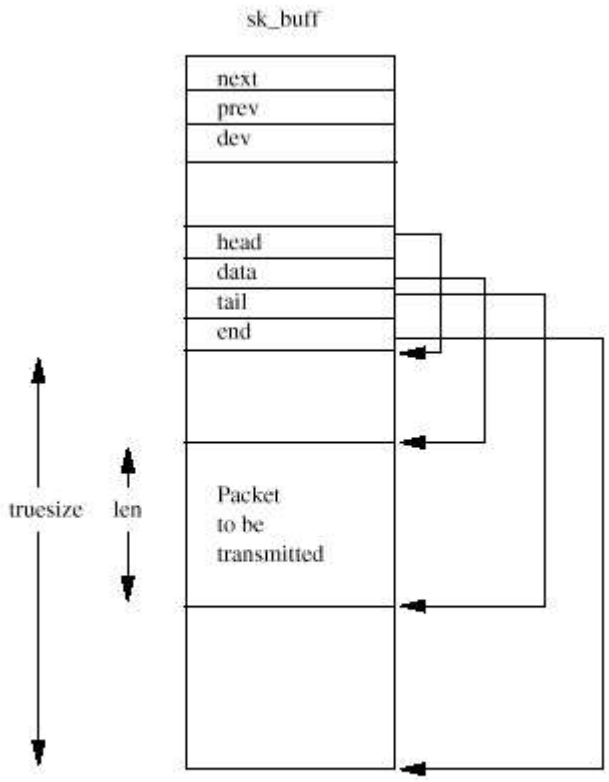



Figure 10.4: The Socket Buffer (sk_buff)

图10.4显示了sk_buff数据结构：每一个sk_buff都有它关联的一块数据。Sk_buff有四个数据指针，用于操纵和管理socket缓冲区的数据

参见include/linux/skbuff.h

head 指向内存中的数据区域的起始。在sk_buff和它相关的数据块被分配的时候确定的。

Data 指向协议数据的当前起始为止。这个指针随着当前拥有这个sk_buff 的协议层不同而变化。

Tail 指向协议数据的当前结尾。同样，这个指针也随拥有的协议层不同而变化。

End 指向内存中数据区域的结尾。这是在这个sk_buff分配的时候确定的。

另有两个长度字段len和truesize，分别描述当前协议报文的长度和数据缓冲区的总长度。Sk_buff处理代码提供了标准的机制用于在应用程序数据上增加和删除协议头和尾。这种代码安全地操纵了sk_buff中的data、tail和len字段。

Push 这把data 指针向数据区域的起始移动，并增加len字段。用于在传送的数据前面增加数据或协议头

参见include/linux/skbuff.h skb_push()

Pull 把data指针从数据区域起始向结尾移动，并减少len字段。用于从接收的数据中删除数据或协议头。

参见include/linux/skbuff.h skb_pull()

Put 把tail指针向数据区域的结尾移动并增加len字段，用于在传输的数据尾部增加数据或协议信息

参见include/linux/skbuff.h skb_put()

trim 把tail指针向数据区域的开始移动并减少len字段。用于从接收的数据中删除数据或协议尾

参见include/linux/skbuff.h skb_trim()

sk_buff数据结构也包括一些指针，使用这些指针，在处理过程中这个数据结构可以存储在sk_buff的双向环形链表中。有通用的sk_buff例程，在这些列表的头和尾中增加sk_buffs和删除其中的sk_buff。

10.5.2 Receiving IP Packets

第8 章描述了Linux的网络设备驱动程序如何建立到核心以及被初始化。这产生了一系列device数据结构，在dev_base列表中链接在一起。每一个device数据结构描述了它的设备并提供了一组回调例程，当需要网络驱动程序工作的时候网络协议层可以调用。这些函数大多数和传输数据以及网络设备的地址有关。当一个网络设备从它的网络上接收到数据报文的时候，它必须把接收到的数据转换到sk_buff数据结构。这些接收的sk_buff在接收的时候被网络驱动程序增加到backlog队列。如果backlog队列增长的太大，那么接收的sk_buff就被废弃。如果有工作要执行，这个网络的button half标记成准备运行。

参见net/core/dev.c netif_rx()

当网络的bottom half处理程序被调度程序调用的时候，它首先处理任何等待传送的网络报文，然后才处理sk_buff的backlog  backlo队列，确定接收到的报文需要传送到那个协议层。当Linux网络层初始化的时候，每一个协议都登记自己，在ptype_all列表或者ptype_base hash table中增加一个packet_type的数据结构。这个packet_type数据结构包括协议类型，一个网络驱动设备的指针，一个协议的数据接收处理例程的指针和一个指针，指向这个列表或者hash table下一个packet_type数据类型。Ptype_all链表用于探测（snoop）从任意网络设备上接收到的所有的数据报文，通常不使用。Ptype_base hash table使用协议标识符hash，用于确定哪一种协议应该接收进来的网络报文。网络的bottom half把进来的sk_buff的协议类型和任一表中的一个或多个packet_type条目进行匹配。协议可能会匹配一个或多个条目，例如当窥测所有的网络通信的时候，这时，这个sk_buff会被克隆。这个sk_buff被传递到匹配的协议的处理例程。

参见net/core/dev.c net_bh()

参见net/ipv4/ip_input.c ip_recv()

### 10.5.3 Sending IP Packets

报文在应用程序交换数据的过程中传送，或者也可能是为了支持已经建立的连接或为了建立连接而由网络协议产生产生。不管数据用什么方式产生，都建立一个包含数据的sk_buff，并当它通过协议层的时候增加许多头。

这个sk_buff需要传递到进行传输的网络设备。但是首先，协议，例如IP，需要决定使用哪一个网络设备。这依赖于这个报文的最佳路由。对于通过modem连接到一个网络的计算机，比如通过PPP协议，这种路由选择比较容易。报文应该要么通过loopback设备传送给本地主机，要么传送到PPP modem连接的另一端的网关。对于连接到以太网的计算机而言，这种选择比较困难，因为网络上连接了许多计算机。

对于传送的每一个IP报文，IP使用路由表解析目标IP地址的路由。对于每一个IP目标在路由表中进行的查找，成功就会返回一个描述要使用的路由的rtable数据结构。包括使用的源IP地址，网络device数据结构的地址，有时候还会有一个预先建立的硬件头。这个硬件头和网络设备相关，包含源和目的物理地址和其它同介质相关的信息。如果网络设备是以太网设备，硬件头会在图10.1中显示，其中的源和目的地址会是物理的以太网地址。硬件头和路由缓存在一起，因为在这个路由传送的每一个IP报文都需要追加这个头，而建立这个头需要时间。硬件头可能包含必须使用ARP协议才能解析的物理地址。这时，发出的报文会暂停，直到地址解析成功。一旦硬件地址被解析，并建立了硬件头，这个硬件头就被缓存，这样以后使用这个接口的IP报文就不需要进行ARP。

参见include/net/route.h

### 10.5.4 Data Fragmentation

每一个网络设备都有一个最大的报文尺寸，它无法传送或接收更大的数据报文。IP协议允许这种数据，会把数据分割成网络设备可以处理的报文大小的更小的单元。IP协议头包含一个分割字段，包含一个标记和分割的偏移量。

当要传输一个IP报文的时候，IP查找用来发送IP报文的网络设备。通过IP路由表来查找这个设备。每一个设备都有一个字段描述它的最大传输单元（字节），这是mtu字段。如果设备的mtu比等待传送的IP报文的报文尺寸小，那么这个IP报文必须被分割到更小的碎片（mtu大小）。每一个碎片用一个sk_buff代表：它的IP头标记了它被分割，以及这个IP报文在数据中的偏移量。最后一个报文被标记为最后一个IP碎片。如果在分割成碎片的过程中，IP无法分配一个sk_buff，这

次传送就失败。

接收IP碎片比发送更难，因为IP碎片可能以任意顺序被接收，而且它们必须在重组之前全部接收到。每一次一个IP报文被接收的时候，都检查它是否是一个IP碎片。收到一个消息的第一个碎片，IP就建立一个新的ipq数据结构，并连接到等待组装的IP碎片的ipqueue列表中。当更多的IP碎片接收到的时候，就查到正确的ipq数据结构并建立一个新的ipfrag数据结构来描述这个碎片。每一个ipq数据结构都唯一描述了一个成为碎片的IP接收帧，包括它的源和目标IP地址，上层协议标识符和这个IP帧的标识符。当接收到所有的碎片的时候，它们被组装在一起成为一个单一的sk_buff，并传递到下一个协议层去处理。每一个ipq包括一个计时器，每一次接收到一个有效的碎片的时候就重新启动。如果这个计时器过期，这个ipq数据结构和它的ipfrag就被去除，并假设这个消息在传输过程中丢失了。然后由高层的协议负责重新传输这个消息。

参见net/ipv4/ip_input.c ip_rcv()

## 10.6 The Address Resolution Protocol (ARP)

地址解析协议的任务是提供IP地址到物理硬件地址的转换，例如以太网地址。IP在它把数据（用一个sk_buff的形式）传送到设备驱动程序进行传送的时候才需要这种转换。它进行一些检查，看这个设备是否需要一个硬件头，如果是，这个报文的硬件头是否需要重建。Linux缓存硬件头以免频繁地重建。如果硬件头需要重建，它就调用和设备相关的硬件头重建例程。所有的一台设备使用相同的通用的头重建例程，然后使用ARP服务把目标的IP地址转换到物理地址。

参见net/ipv4/ip_output.c ip_build_xmit()

参见net/ethernet/eth.c rebuild_header()

ARP协议本身非常简单，包含两种消息类型：ARP请求和ARP应答。ARP请求包括需要转换的IP地址，应答（希望）包括转换的IP地址和硬件地址。ARP请求被广播到连接到网络的所有的主机，所以，对于一个以太网所有连在以太网上的机器都可以看到这个ARP请求。拥有这个请求中包括的IP地址的机器会回应这个ARP请求，用包含它自己物理地址的ARP应答。

Linux中的ARP协议层围绕着一个arp_table数据结构的表而建立。每一个描述一个IP和物理地址的对应。这些条目在IP地址需要转换的时候创建，随着时间推移变得陈旧的时候被删除。每一个arp_table数据结构包含以下域：

Last used 这个ARP条目上一次使用的时间

Last update 这个ARP条目上一次更新的时间

Flags 描述这个条目的状态：它是否完成等等

IP address 这个条目描述的IP地址

Hardware address 转换（翻译）的硬件地址

Hardware header 指向一个缓存的硬件头的指针

Timer 这是一个timer_list的条目，用于让没有回应的ARP请求超时

Retries 这个ARP请求重试的次数

Sk_buff queue 等待解析这个IP地址的sk_buff条目的列表

ARP表包含一个指针（arp_tables向量表）的表，把arp_table的条目链接在一起。这些条目被缓存，以加速对它们的访问。每一个条目用它的IP地址的最后两个字节做表的索引进行查找，然后跟踪这个条目链，直到找到正确的条目。Linux也缓存从arp_table条目预先建立的硬件头，用hh_cache数据结构的形式进行缓存。

当请求一个IP地址转换的时候，没有对应的arp_table条目，ARP必须发送一个ARP请求消息。它在表中创建一个新的arp_table条目，并把需要地址转换的包括了网络报文的sk_buff放到这个新的条目的sk_buff队列。它发出一个ARP请求并让ARP过时计时器运行。如果没有回应，ARP会重试几次。如果仍旧没有回应，ARP会删除这个arp_table条目。任何排队等待这个IP地址进行转换的sk_buff数据结构会被通知，由传输它们的上层协议负责处理这种失败。UDP不关心丢失的报文，但是TCP会在一个建立的TCP连接上试图重新发送。如果这个IP地址的属主用它的硬件地址应答，这个arp_table条目标记为完成，任何排队的sk_buff会被从对队列中删除，继续传送。硬件地址被写到每一个sk_buff的硬件头中。

ARP协议层也必须回应指明它的IP地址的ARP请求。它登记它的协议类型（ETH_P_ARP），产生一个packet_type数据结构。这意味着网络设备接收到的所有的ARP报文都会传给它。象ARP应答一样，这也包括ARP请求。它使用接收设备的device数据结构中的硬件地址产生ARP应答。

网络拓扑结构不断变化，IP地址可能被重新分配到不同的硬件地址。例如，一些拨号服务为它建立的每一个连接分配一个IP地址。为了让ARP表中包括最新的条目，ARP运行一个定期的计时器，检查所有的arp_table条目，看哪一个超时了。它非常小心，不删除包含包含一个或多个缓存的硬件头的条目。删除这些条目比较危险，因为其它数据结构依赖它们。一些arp_table条目是永久的，并被标记，所以它们不会被释放。ARP表不能增长的太大：每一个arp_table条目都要消耗一些核心内存。每当需要分配一个新的条目而ARP表到达了它的最大尺寸的时候，就查找最旧的条目并删除它们，从而修整这个表。

## 10.7 IP Routing

IP路由功能确定发向一个特定的IP地址的IP报文应该向哪里发送。当传送IP报文的时候，会有许多选择。目的地是否可以到达？如果可以，应该使用哪一个网络设备来发送？是不是有不止一个网络设备可以用来到达目的地，哪一个最好？IP路由数据库维护的信息可以回答这些问题。有两个数据库，最重要的是转发信息数据库（Forwarding Information Database）。这个数据库是已知IP目标和它们最佳路由的详尽的列表。另一个小一些，更快的数据库，路由缓存（route cache）用于快速查找IP目标的路由。象所有缓存一样，它必须只包括最常访问的路由，它的内容是从转发信息数据库中得来的。

路由通过BSD socket接口的IOCTL请求增加和删除。这些请求被传递到具体的协议去处理。INET协议层只允许具有超级用户权限的进程增加和删除IP路由。这些路由可以是固定的，或者是动态的，不断变化的。多数系统使用固定路由，除非它们本身是路由器。路由器运行路由协议，不断地检查所有已知IP目标的可用的路由。不是路由器的系统叫做末端系统（end system）。路由协议用守护进程的形式来实现，例如GATED，它们也使用BSD socket接口的IOCTL来增加和删除路由。

### 10.7.1 The Route Cache

不论何时查找一个IP路由的时候，都首先在路由缓存中检查匹配的路由。如果在路由缓存中没有匹配的路由，才查找转发信息数据库。如果这里也找不到路由，IP报文发送会失败，并通知应用程序。如果路由在转发信息数据库而不在路由缓存中，就为这个路由产生一个新的条目并增加到路由缓存中。路由缓存是一个表（ip_rt_hash_table），包括指向rtable数据结构链的指针。路由表的索引是基于IP地址最小两字节的hash 函数。这两个字节通常在目标中有很大不同，让hash value可以最好地分散。每一个rtable条目包括路由的信息：目标IP地址，到达这个IP地址要使用的网络设备（device结构），可以使用的最大的信息尺寸等等。它也有一个引用计数器（refrence count），一个使用计数器（usage count）和上次使用的时间戳（在jiffies中）。每一次使用这个路由的时候这个引用计数器就增加，显示利用这个路由的网络连接数目，当应用程序停止使用这个路由的时候就减少。使用计数器每一次查找路由的时候就增加，用来让这个hash条目链的rtable条目变老。路由缓存中所有条目的最后使用的时间戳用于定期检查这个rtable是否太老。如果这个路由最近没有使用，它就从路由表中废弃。如果路由保存在路由缓存中，它们就被排序，让最常用的条目在hash链的前面。这意味着当查找路由的时候找到这些路由会更快。

参见net/ipv4/route.c check_expire()

### 10.7.2 The Forwarding Information Database

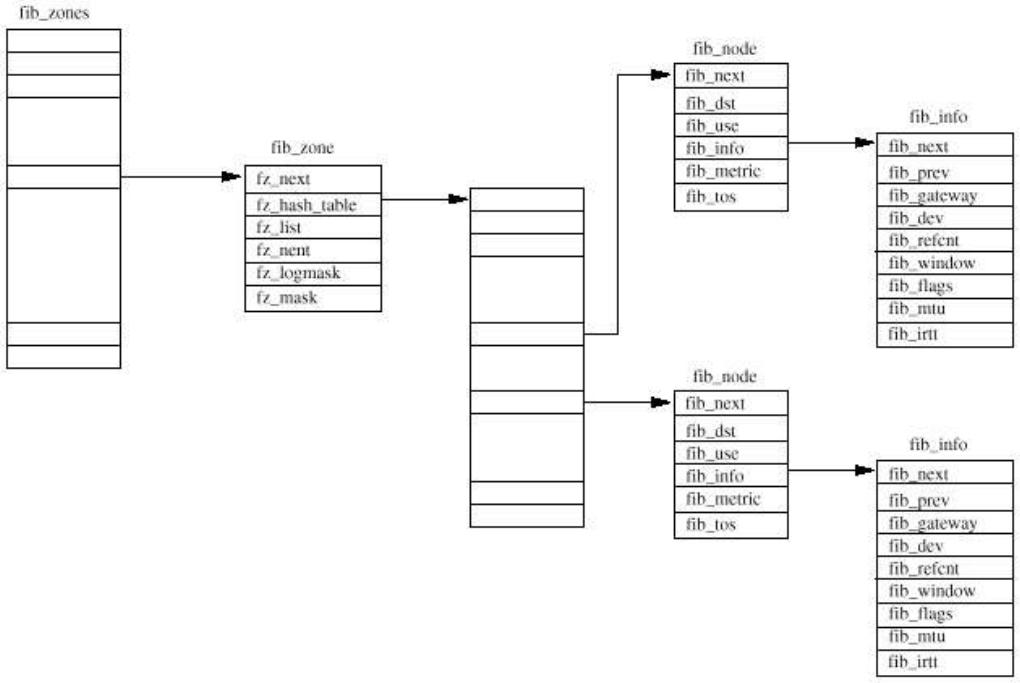



Figure 10.5: The Forwarding Information Database

转发信息数据库（图10.5显示）包含了当时从IP的观点看待系统可用的路由。它是非常复杂的数据结构，虽然它已经进行了合理有效的安排，但是它对于参考而言并不是一个快速的数据库。特别是如果每一个传输的IP报文都在这个数据库中查找目标会非常慢。这也是为什么要有路由缓存：加速已经知道最佳路由的IP报文的传送。路由缓存从这个转发信息数据库得到，表示了它最常用的条目。

每一个IP子网用一个fib_zone数据结构表示。所有这些都被fib_zones hash表指向。Hash索引取自IP子网掩码。所有通向同一子网的路由都用排在每一个fib_zone数据结构的fz_list队列中得的成对的fib_node和fib_info数据结构来描述。如果这个子网的路由数目变得太大，就生成一个hash table，让fib_node数据结构的查找更容易。

对于同一个IP子网，可能存在多个路由，这些路由可能穿过多个网关之一。IP路由层不允许使用相同的一个网关对于一个子网有多于一个路由。换句话说，如果对于一个子网有多个路由，那么要保证每一个路由都是用不同的网关。和每一个路由关联的是它的量度（metric），这是用来衡量这个路由的益处。一个路由的量度，基本上，是它在到达目标子网之前必须跳过的子网数目。这个量度越高，路由越差。

# Chapter 11

# Kernel Mechanisms （核心机制）

本章描述了Linux核心需要提供的一些一般的任务和机制，让核心的其余部分可以有效地工作。

## 11.1 Bottom Half Handling

通常在核心中会有这样的时候：你不希望执行工作。一个好例子是在中断处理的过程中。当引发了中断，处理器停止它正在执行的工作，操作系统把中断传递到适当的设备驱动程序。设备驱动程序不应该花费太多时间来处理中断，因为在这段时间，系统中的其他东西都不能运行。通常一些工作可以在稍后的时候进行。Linux发明了boffom half处理程序，这样设备驱动程序和Linux核心的其它部分可以把可以稍后作的工作排队。图11.1显示了同bottom half 处理相关的核心数据结构。有多达32个不同的bottom half处理程序：bh_base是一个指针的向量表，指向核心的每一个bottom half处理例程，bh_active和bh_mask按照安装和激活了哪些处理程序设置它们的位。如果bh_mask的位N设置，则bh_base中的第N个元素会包含一个bottom half例程的地址。如果bh_active的第N位设置，那么一旦调度程序认为合理，就会调用第N位的bottom half处理程序。这些索引是静态定义的：timer bottom half 处理器优先级最高（索引0），console bottom half处理程序优先级次之（index 1）等等。通常bottom half 处理例程会有和它关联的任务列表。例如这个immediate buttom half handler 通过包含需要立即执行的任务的immediate任务队列（tq_immediate）来工作。

参见include/linux/interrupt.h

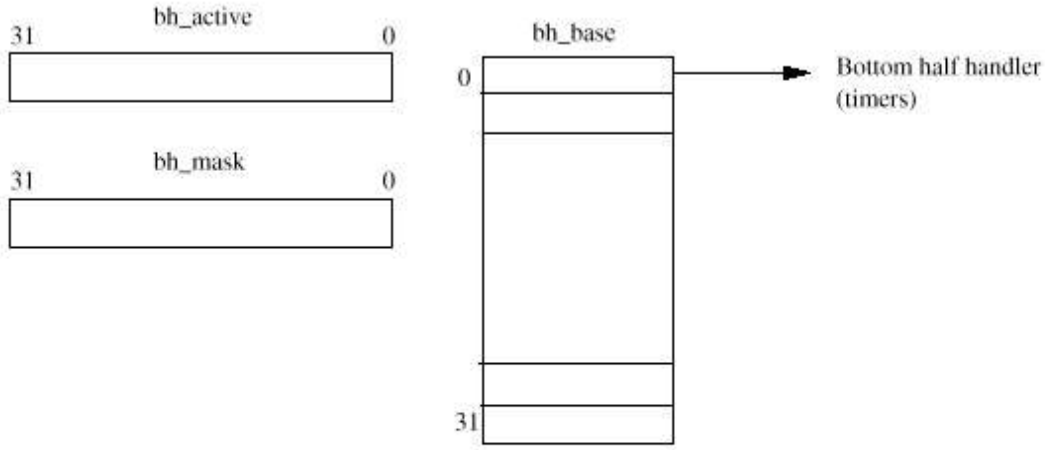



Figure 11.1: Bottom Half Handling Data Structures

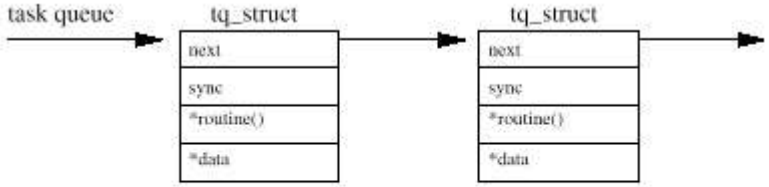



Figure 11.2: A Task Queue

核心的一些bottom half处理程序和设备有关，但是其它的是更一般的：

TIMER 这个处理程序在每一次系统定时时钟中断被标记成为激活，用来驱动核心的时钟队列机制

CONSOLE 这个处理程序用来处理控制台消息

TQUEUE 这个处理程序用来处理TTY消息

NET 这个处理程序用来处理通用的网络处理

IMMEDIATE 通用的处理程序，一些设备驱动程序用来排列稍后进行的工作

设备驱动程序或者核心的其它部分，需要调度稍后进行的工作的时候，它就在适当的系统队列中增加这个工作，例如时钟队列，然后就发送信号到核心，一些bottom half处理需要进行。它通过设置bh_active中的合适的位来做到这点。如果驱动程序在immediate队列排列了一些东西并希望immediate bottom half 处理程序会运行并处理它的时候就设置第8位。每一次系统调用的最后，把控制权返回调用程序之前都检查bh_active的位掩码。如果有任意位被设置，相应的激活的bottom half 处理例程就被调用。首先检查位0，然后1直到位31。调用每一个bottom half 处理例程调用的时候就清除bh_active中相应的位。Bh_active是易变的：它只在调用调度程序之间有意义，通过设置它，当没有需要作的工作的时候可以不调用相应的bottom half 处理程序。

Kernel/softirq.c do_bottom_half()

## 11.2 Task Queues（任务队列）

任务队列是核心用来把工作推迟到以后的方法。Linux由一个通用的机制，把工作排列在队列中并在稍后的时间进行处理。任务队列通常和bottom half处理程序一起使用：当timer bottom half 处理程序运行的时候处理计时器任务队列。任务队列是一个简单的数据结构，参见图11.2，包括一个tq_struct数据结构的单链表，每一个包括例程的指针和指向一些数据的指针。

参见include/linux/tqueue.h 当这个任务队列的单元被处理的时候调用这个例程，数据的指针会传递给它。

核心的任何东西，例如设备驱动程序，都可以创建和使用任务队列，但是有三个任务队列是由核心创建和管理的：

timer 这个队列用于排列在下一个系统时钟之后尽可能运行的工作。每一个时钟周期，都检查这个队列，看是否有条目，如果有，时钟队列的bottom half 处理程序被标记为激活。当调度在一次运行的时候，就处理这个时钟队列bottom half 处理程序以及其它bottom half 处理程序。不要把这个队列和系统计时器混淆，那是一个更复杂的机制

immediate 这个队列也是在调度程序处理激活的bottom half处理程序的时候被处理。这个

immediate bottom half 处理程序没有timer 队列bottom half 处理程序优先级高，所以这些任务会迟疑写运行。

Scheduler 这个任务队列由调度程序直接处理。它用于支持系统中的其它任务队列，这种情况下，要运行的任务会是一个处理任务队列（例如设备驱动程序）的例程。

当处理任务队列的时候，指向队列中的一个单元的指针从队列中删除，用一个null指针代替。实际上，这种删除是一个不能被中断的原子操作。然后为队列中的每一个单元顺序调用它的处理例程。队列中的单元通常是静态分配的数据。但是没有一个固有的机制来废弃分配的内存。任务队列处理例程只是简单地移到列表中的下一个单元。保证正确地清除任何分配的核心内存是任务本身的工作。

## 11.3 Timers

一个操作系统都需要有能力把一个活动调度到将来的一个时间，这需要一种机制让活动可以调度到相对准确的时间去运行。任何希望支持一个操作系统的微处理器都需要一个可编程间隔适中，定期中断处理器。这个定期的中断就是系统时钟周期（system clock tick），它就象一个节拍器，指挥系统的活动。Linux用非常简单的方式看待时间：它从系统启动的时候开始用时钟周期测量时间。任何系统时间都基于这种量度，叫做jiffers，和全局变量同名。

Linux有两种类型的系统计时器，每一种都排列例程，在特定的系统时间调用，但是实现的方式上它们有轻微的不同。图11.3显示了两种机制。第一种，旧的计时器机制，有一个静态的数组，有32个指向timer_struct数据结构的指针和一个激活的时钟的掩码，timer_active。计时器放在这个计时器表中的什么位置是静态定义的（和bottom half 处理程序中的bh_base不同）。条目在系统初始化的时候被加到这个表中。第二种机制，使用一个timer_list数据结构的链接表中，按照过期时间的数据排列。

参见include/linux/timer.h

每一种方法都使用jiffies中的时间作为过期时间，这样一个希望运行5秒的计时器会有一个可以换算为5秒的jiffies单元加上当前系统时间得到计时器过期时的系统时间（以jiffies为单位）。每一次系统时钟周期，timer bottom half处理程序被标记为激活，所以当下一次调度程序运行的时候，会处理计时器队列。Timer bottom half处理程序会处理全部两种类型的系统计时器。对于旧的系统计时器，检查timer_active位掩码中置了位的。如果一个激活的计时器过期（过期时间小于当前的系统jiffies），就调用它的计时器例程，并清除它的激活位。对于新的系统计时器，检查timer_list数据结构的链接表中的条目。每一个过期的计时器从这个列表中删除并调用它的例程。新的计时器机制的优点在于它可以向计时器例程传递参数。

参见kernel/sched.c timer_bh() run_old_timers() run_timer_list()

## 11．4 Wait Queues（等待队列）

许多时候一个进程必须等待一个系统资源。例如，一个进程可能需要描述文件系统中一个目录的VFS inode，但是这个inode可能不在buffer cache钟。这时，系统必须等待这个inode从包含这个文件系统的物理介质中取出来，然后才能继续。

Linux核心使用一个简单的数据结构，一个等待队列（见图11.4），包含一个指向进程的task_struct的指针和一个指向等待队列中下一个元素的指针。

参见include/linux/wait.h

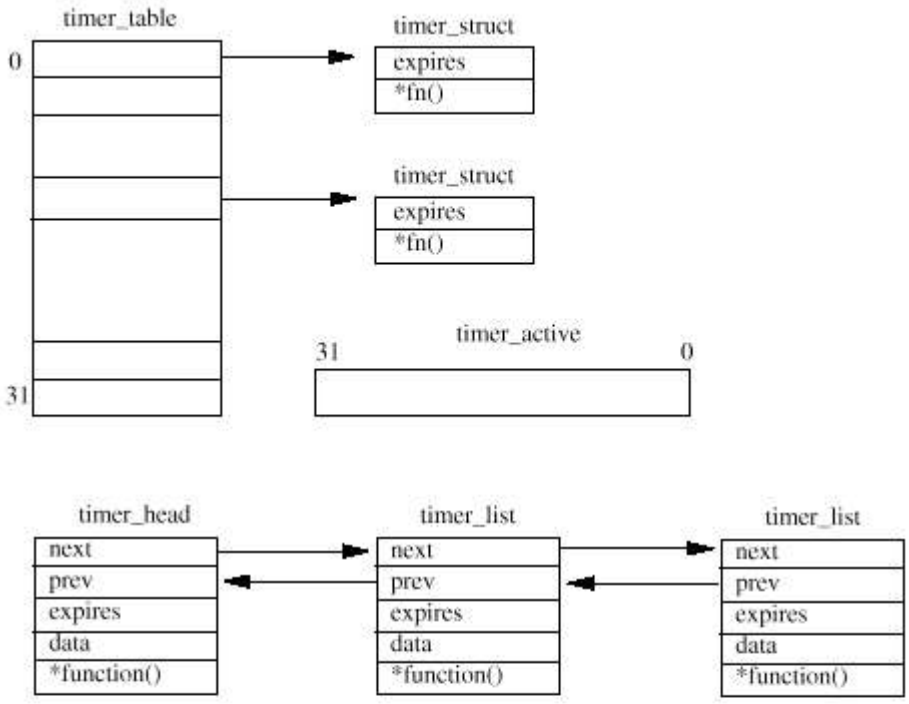



Figure 11.3: System Timers

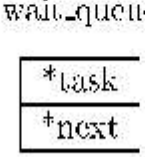



Figure 11.4: Wait Queue

当进程被增加到了一个等待队列的结尾的时候，它们可能时可被中断或者不可中断的。可中断的进程在等待队列等待的过程中可以被事件中断，例如过期的计时器或者发送来的信号等事件。等待进程的状态会反映出来，可以是INTERRUPTIBLE或者UNINTERRUPTIBLE。因为这个进程现在

不能继续运行，就开始运行调度程序，当它选择了一个新的进程运行的时候，这个等待的进程就会被挂起。

当处理等待队列的时候，等待队列中的每一个进程的状态都被设置位RUNNING。如果进程从运行队列中删除了，它就被放回到运行队列。下一次运行调度程序的时候，在等待队列的进程现在就成为运行的候选，因为它们不再等待了。当一个等待队列的进程被调度的时候，首先要作的是把自己从等待队列中删除。等待队列可以用于同步访问系统资源，Linux用这种方式实现它的信号灯。

## 11.5 Buzz Locks

通常叫做spin locks，这是保护一个数据结构或代码段的一个原始方法。它们一次只允许一个进程处于一个重要的代码区域。Linux使用它们来限制对于数据结构中的域的访问，它利用一个整数字段作为锁。每一个希望进入这个区域的进程试图把锁的起始值从0变为1。如果当前致使1，进程重新尝试，在一个紧凑的代码循环中旋转（spin）。对于保存这个锁的内存位置的访问必须具有原子性，读取它的值、检查它是0然后把它改为1，这个动作不能被其他任何进程打断。多数CPU结构通过特殊的指令为此提供支持，但是你也可以使用未缓存的主内存实现这种buzz lock。

当属主进程离开这个重要的代码区域的时候，它减小这个buzz lock，让它的值返回到0。任何在这个锁上循环的进程现在会读到0，第一个做到的进程会把它增加到1并进入这个重要区域。

## 11.6 Semaphores（信号灯）

信号灯用于保护重要的代码区域或数据结构。记住，对于重要数据结构例如描述一个目录的VFS inode的每一次访问都是通过核心为进程执行的。如果允许一个进程改变另一个进程使用的重要的数据结构是非常危险的。实现的方法之一是在要访问的重要的代码片上使用一个buzz lock，虽然这是最简单的方法但是不会有太好的系统性能。Linux使用信号灯实现一次只允许一个进程访问重要的代码和数据区域：所有其它希望访问这个资源的进程会被迫等待直到信号灯空闲。等待的进程被刮起，系统中的其它进程和平时一样正常运行。

一个Linux信号灯数据结构包括以下信息：

参见include/asm/semaphore.h

count 这个字段记录了希望使用这个资源的进程数。正的数值表示这个资源可用。负值或0表示有进程在等待。起始值1表示同一时间有一个且只有一个进程可以使用这个资源。当进程希望使用这个资源的时候它们减小这个count，当结束对这个资源的使用的时候，它们增加这个count

waking 等待这个资源的进程数，这也是等待当资源空闲的时候被唤醒的进程数。

Wait queue 当进程等待这个资源的时候它们被放到这个等待队列

Lock 当访问waking域所用的buzz lock

假设信号灯的起始值是1，第一个到来的进程会看到count是正的，把它减少1，成为0。这个进程现在“拥有”这个受到信号灯保护的重要的代码片或资源。当进程离开这个重要区域的时候它增加信号灯的count。最理想的情况是没有其它进程竞争这个重要区域的所有权。Linux实现的信号灯在这种最常见的情况下工作的非常高效。

如果另一个进程希望进入这个重要区域，而它已经被一个进程拥有，它也会减少这个count。因为这个count现在是-1，这个进程不能进入这个重要区域。它必须等待直到拥有的进程退出来。Linux让等待的进程睡眠直到拥有权的进程退出这个重要区域把它唤醒。等待进程把它自己加到信号灯的等待队列中，并循环检查waking字段的值，调用调度程序直到waking非0。

这个重要区域的属主增加信号灯的count，如果它小于或等于0，那么还有进程在睡眠，等待这个资源。理想情况下，信号灯的count会返回到它的起始值1，这样就不需要做什么工作。所有权的进程增加waking计数器并唤醒在信号灯等待队列中睡眠的进程。当等待的进程被唤醒之后，waking计数器现在是1，它知道它现在可以进入这个重要区域。它减少waking计数器，把它返回0并继续。所有对于这个信号灯的waking字段的访问都用信号灯的lock这个buzz lock来保护。

## Chapter 12

## Modules

本章描述Linux核心如何只在需要的时候才动态加载函数，例如文件系统。

Linux是一个完整的核心，就是说，它是一个单一的巨大的程序，核心的功能组件可以访问它的所有的内部数据结构以及例程。另一种方法是使用一个微内核的结构，核心的功能片被分成独立的单元，互相之间有严格的通讯机制。这样通过配置进程向核心增加新的组件不花多少时间。比如你希望增加一个NCR 810 SCSI卡的SCSI驱动程序，你不需要把它连接到核心。否则你不得不配置并建立一个新的核心才能使用这个NCR 810。作为一种变通，Linux允许在你需要的时候动态地加载和卸载操作系统的组件。Linux的模块是可以在系统启动之后任何时候动态连接到核心的代码块。它们可以在不被需要的时候从核心删除并卸载。大多数Linux核心模块是设备驱动程序，伪

设备驱动程序比如网络驱动程序或文件系统。

你可以使用insmod和rmmod命令明确地加载和卸载Linux核心模块，或者在需要这些模块的时候由核心自己要求核心守护进程（kerneld）加载和卸载这些模块。在需要的时候动态地加载代码相当有吸引力，因为它让核心可以保持最小而且核心非常灵活。我当前的Intel核心大量使用模块，它只有406K大小。我通常只适用VFAT文件系统，所以我建立我的Linux核心，当我安装一个VFAT分区的时候自动加载VFAT文件系统。当我卸载VFAT文件系统的时候，系统探测到我不再需要VFAT文件系统模块，把它从系统中删除。模块也可以用来尝试新的核心代码而不需要每次都创建和重启动核心。但是，没有这么好的事情，使用核心模块通常伴随轻微的性能和内存开支。一个可加载模块必须提供更多的代码，这种代码和额外的数据结构会占用更多一点的内存。另外因为间接访问核心资源也让模块的效率轻微降低。

一旦Linux核心加载，它就和普通核心代码一样成为核心的一部分。它和任何核心代码拥有相同的权利和义务：换句话说，Linux核心模块和所有的核心代码或设备驱动程序一样可能让核心崩溃。

既然模块在需要的时候可以使用核心资源，它们必须能够找到这些资源。比如一个模块需要调用kmalloc()，核心内存分配例程。当建立的时候（build），模块不知道内存中kmalloc()在哪里，所以当这个模块加载的时候，在模块能够工作之前，核心必须整理模块对于kmmalloc()的所有的引用。核心在核心符号表中保存了所有核心资源的列表，所以当模块加载的时候它可以解析模块中对于这些资源的引用。Linux允许模块堆栈（堆砌），就是一个模块需要另一个模块的服务。例如VFAT文件系统模块需要FAT 文件系统模块的服务，因为VFAT文件系统或多或少是FAT文件系统上的扩展。一个模块需要另一个模块的服务或资源的情况和一个模块需要核心自己的服务和资源的情况非常相似，只不过这时请求的服务在另一个，此前已经加载的模块钟。当每一个模块加载的时候，核心修改它的符号表，把这个新加载的模块的所有输出的资源或符号加到核心符号表中。这意味着，当下一个模块加载的时候，它可以访问已经加载的模块的服务。

当时图卸载一个模块的时候，核心需要知道这个模块不在用，它还需要一些方法来通知它准备卸载的模块。用这种方法模块可以在它从核心删除之前释放它占用的任何的系统资源，例如核心内存或中断。当模块卸载的时候，核心把这个模块输出到核心符号表中所有的符号都删除。

除了写的不好的可加载模块可能破坏操作系统之外，还有另一个危险。如果你加载一个为比你当前运行的核心要早或迟的核心建立的模块会发生什么？如果这个模块执行一个核心例程而提供了错误的参数就会引起问题。核心可以选择防止这种情况，当模块加载的时候进行严格的版本检查。

## 12.1 Loading a Module（加载一个模块）

用两种方法可以加载一个核心模块。第一种使用insmod命令手工把它插入到核心。第二种，更聪明的方法是在需要的时候加载这个模块：这叫做按需加载（demand loading）。当核心发现需要一个模块的时候，例如当用户安装一个不在核心的文件系统的时候，核心会请求核心守护进程（kerneld）试图加载合适的模块。

Kerneld和insmod，lsmod以及rmmod都在modules程序包中。

核心守护进程通常是拥有超级用户特权的一个普通的用户进程。当它启动的时候（通常是在系统启动的时候启动），它打开一个通向核心的IPC通道。核心使用这个连接向kerneld发送消息，请求它执行大量的任务。Kerneld的主要功能是加载和卸载核心模块，但是它也可以执行其它任务，比如需要的时候在串行线上启动PPP连接，不需要的时候把它关闭。Kerneld本身并不执行这些任务，它运行必要的程序比如insmod来完成工作。Kerneld只是核心的一个代理，调度它的工作。

参见include/linux/kerneld.h

insmod 命令必须找到它要加载的被请求的核心模块。按徐加载的核心模块通常放在/lib/mmodules/kernel-version目录里边。核心模块和系统中的其它程序一样是连接程序的目标文件，但是它们被连接成可以重定位的映像。就是没有连接到特定地址去运行的映像。它们可以是a.out或elf格式的目标文件。Insmod指向一个特权的系统调用，找出系统的输出符号。它们以符号名称和值（例如它的地址）的形式成对存放。核心的输出符号表放在核心维护的模块列表中的第一个module数据结构，用module_list指针指向。只有在核心编译和连接的时候特殊指定的符号才加到这个表中，而并非核心的每一个符号都输出它的模块。例如符号“request_irq”是一个系统例程，当一个驱动程序希望控制一个特定的系统中断的时候必须调用它。在我当前的核心上，它的值是0x0010cd30。你可以检查文件/proc/ksyms或使用ksyms工具简单地查看输出的核心符号和它们的值。Ksyms工具可以向你显示所有的输出的核心符号或者只显示哪些加载模块输出的符号。Insmod把模块读取到它的虚拟内存，使用核心的输出符号来整理这个模块对于核心例程和资源的未解析的引用。这个整理过程是用向内存中的模块映像打补丁的方式进行，insmod物理上把符号的地址写到模块的合适的位置。

参见kernel/module.c kernel_syms() include/linux/module.h

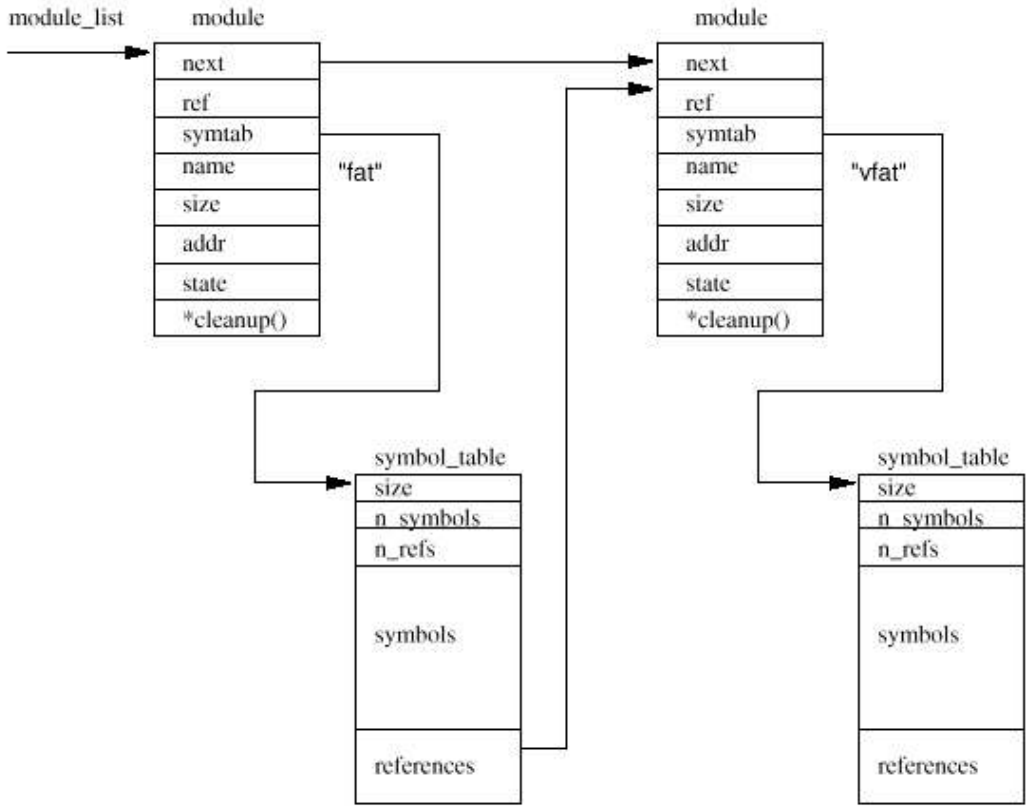



Figure 12.1: The List of Kernel Modules

当insmod整理完了模块对于输出的核心符号的引用之后，他向核心请求足够的空间放置新的核心，又是通过特权的系统调用。核心分配一个新的module数据结构和足够的核心内存来存放这个新的模块，并把它放置到核心的模块列表的最后。这个新的模块被标记为UNINITIALIZED。图12.1显示了核心模块列表的后面两个模块：FAT和VFAT被加载到了内存。图中没有显示的有列表的第一个模块：这是一个伪模块，用于放置核心的输出符号表。你可以使用命令lsmod列出所有加载的核心模块和它们之间的依赖关系。Lsmod只是简单地把从核心module数据结构列表中提取的/proc/modules重新安排了格式。核心为模块分配的内存映射到insmod进程的地址空间，所以它可以访问它。Insmod把模块拷贝到分配的空间，并把它重定位，这样它就可以从被分配的核心地址运行。必须进行重定位，因为一个模块不能期待在两次被加载到相同的地址或者在两个不同的Linux系统上被加载到相同的地址。这一次，重定位又关系到要用适当的地址为模块的映像打补丁。

参见kernel/module.c create_module()

新的模块也向核心输出符号，Insmod建立一个输出映像表。每一个核心模块必须包含模块初始化和模块清除的历程，这些符号必须是专用的而不是输出的，但是insmod必须知道它们的地址，能把它们传递给核心。所有这些做好之后，Insmod现在准备初始化这个模块，它执行一个特权的系统调用，把这个模块的初始化和清除例程的地址传递给核心。

参见kernel/module.c sys_init_module()

当一个新的模块加到核心的时候，它必须更新核心的符号表并改变被新的模块使用的模块。其它模块依赖的模块必须在它们的符号表之后维护一个引用列表，用它们的module数据结构指向。图12.1显示了VFAT文件系统模块依赖于FAT文件系统模块。所以FAT模块包含一个到VFAT模块的引用：这个引用在VFAT模块加载的时候增加。核心调用模块的初始化例程，如果成功，它开始安装这个模块。模块的清除例程的地址保存在它的module数据结构中，当这个模块卸载的时候核心会去调用。最后，模块的状态被设置为RUNNING。

## 12.2 Unloading a Module

模块可以使用rmmod命令删除，但是kerneld可以把所有不用的按需加载的模块从系统中删除。每一次它的空闲计时器到期的时候，kerneld执行系统调用，请求从系统删除所有的不需要的按需加载的模块。这个计时器的值由你在启动kerneld的时候设定：我的kerneld每180秒检查一次。如果你安装了一个iso9660 CD ROM而你的iso9660文件系统是一个可加载模块，那么，在CD ROM卸载不久，iso9660模块会从核心中删除。

如果核心中的其它组件依赖于一个模块，它就不能被删除。例如如果你安装了一个或更多的VFAT文件系统，你就不能卸载VFAT模块。如果你检查ls输出，你会看到每一个模块关联一个计数器。例如：

Module: #pages: Used by:

msdos 5 1

vfat 4 1 (autoclean)

fat 6 [vfat msdos] 2 (autoclean)

这个计数器（count）是依赖于这个模块的核心实体的数目。在上例中，vfat和msdos都依赖于fat模块，所以fat模块的计数器是2。Vfat和msdos模块的依赖数都是1，因为它们都有一个安装的文件系统。如果我加载另外一个VFAT文件系统，那么vfat模块的计数器会变成2。一个模块的计数器放在它的映像的第一个长字中（longword）。

因为它也放置AUTOCLEAN和VISITED标志，所以这个字段有一些轻微过载。这些标志都用于按需加载模块。这些模块被标记为AUTOCLEAN，这样系统可以识别出哪些它可以自动卸载。VISITED标志表示这个模块被一个或多个系统组件使用：只要另一个组件使用它就设置这个标志。每一次kerneld请求系统删除不用的按需加载的模块的时候，它都查看系统中所有的模块，找到合适的候选。它只查看标记为AUTOCLEAN而且状态是RUNNING的模块。如果这个候选的

VISITED标记被清除，那么它就删除这个模块，否则它就清除这个VISITED标记，继续查找系统中的下一个模块。

假设一个模块可以被卸载，就调用它的清除例程（cleanup），让它释放它所分配的核心资源。这个module数据结构被标记为DELTED，从核心模块列表中删除。任何其它的它所依赖的模块的引用表被修改，这样它们不再把它当作一个依赖者。这个模块需要的所有的核心内存被释放。

参见kernel/module.c delete_module()

## Chapter 13

## The Linux Kernel Sources（Linux核心源程序）

本章描述了你应该在Linux核心源程序的什么地方开始查看特定的核心功能。

本书不依赖‘C’语言的知识或要求你有Linux核心源程序才能理解Linux核心如何工作。而是说，练习查看核心源程序能够对于Linux操作系统有一个深入地理解。本章给出核心源程序的概览：它们如何组织，你应该从哪里开始查找特定的代码。

## Where to Get The Linux Kernel Sources（从哪里得到Linux核心源程序）

所有的主要的Linux分发（Craftworks，Debian，Slackware，RedHat 等等）中间都有核心源程序。通常L安装在你的Linux系统上的Linux核心都是用这些源程序建立的。实际上这些源程序显得有些过时，所以你可能希望得到附录C提到的web站点得到最新的源程序。它们放在ftp://ftp.cs.helsinki.fi和其它所有的镜像的web站点。Helsinki的web站点最新，但是其它站点例如MIT和Sunsite也不会太落后。

如果你无法访问web，还有许多CDROM厂家用非常合理的费用提供世界主要web站点的块找。一些甚至提供预订服务，按季或月进行更新。你的本地的Linux用户组也是一个源程序的好的来源。

Linux核心源程序有一个非常简单的编号系统。任何偶数的核心（例如2.0.30）都是一个稳定的发行的核心，而任何奇数的核心（例如2.1.42）都是一个开发中的核心。本书基于稳定的2.0.30源代码。开发版的核心具有所有的最新特点和所有最新的设备的支持，但是它们可能不稳定，可能不是你所要的，但是让Linux社团测试最新核心是很重要的。这样可以让整个社团都进行测试。记住，即使你测试非生产用核心，最好也要备份你的系统。

对于核心源程序的改动作为patch文件分发。工具patch可以对于一系列源文件应用一系列修改。例如，如果你有2.0.29的源程序树，而你希望转移到2.0.30，你可以取到2.0.30的patch文件，并把这些patch（编辑）应用到源程序树上：

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

这样可以不用拷贝整个源程序树，特别对于慢速的串行连接。一个核心补丁（正式和非正式的）的好来源是http://www.linuxhq.com

How The Kernel Sources Are Arranged（核心源程序如何组织）

在源程序树的最上层你会看到一些目录：

arch arch子目录包括所有和体系结构相关的核心代码。它还有更深的子目录，每一个代表一种支持的体系结构，例如i386和alpha。

Include include子目录包括编译核心所需要的大部分include文件。它也有更深的子目录，每一个支持的体系结构一个。Include/asm是这个体系结构所需要的真实的include目录的软链接，例如include/asm-i386。为了改变体系结构，你需要编辑核心的makefile，重新运行Linux的核心配置程序

Init 这个目录包含核心的初始化代码，这时研究核心如何工作的一个非常好的起点。

Mm 这个目录包括所有的内存管理代码。和体系结构相关的内存管理代码位于arch/*/mm/，例如arch/i386/mm/fault.c

Drivers 系统所有的设备驱动程序在这个目录。它们被划分成设备驱动程序类，例如block。

Ipc 这个目录包含核心的进程间通讯的代码

Modules 这只是一个用来存放建立好的模块的目录

Fs 所有的文件系统代码。被划分成子目录，每一个支持的文件系统一个，例如vfat和ext2

Kernel 主要的核心代码。同样，和体系相关的核心代码放在arch/*/kernel

Net 核心的网络代码

Lib 这个目录放置核心的库代码。和体系结构相关的库代码在arch/*/lib/

Scripts 这个目录包含脚本（例如awk和tk脚本），用于配置核心

## Where to Start Looking（从哪里开始看）

看像Linux核心这么巨大复杂的程序相当困难。它就像一个巨大的线球，显示不出终点。看核心的一部分代码通常会引到查看其它几个相关的文件，不就你就会忘记你看了什么。下一节给你一个提示，对于一个给定的主题，最好看源程序树的那个地方。

## System Startup and Initialization（系统启动和初始化）

在一个Intel系统上，当loadlin.exe或LILO把核心加载到内存并把控制权交给它的时候，核心开始启动。这一部分看arch/i386/kernel/head.S。head.S执行一些和体系结构相关的设置工作并跳到init/main.c中的main()例程。

## Memory Management（内存管理）

代码大多在mm但是和体系结构相关的代码在arch/*/mm。Page fault处理代码在mm/memory.c中，内存映射和页缓存代码在mm/filemap.c 中。Buffer cache 在mm/buffer.c中实现，交换缓存在mm/swap_state.c和mm/swapfile.c中。

## Kernel

大部分相对通用的代码在kernel，和体系结构相关的代码在arch/*/kernel。调度程序在kernel/sched.c，fork代码在kernel/fork.c。bottom half 处理代码在include/linux/interrupt.h。task_struct数据结构可以在include/linux/sched.h中找到

## PCI

PCI伪驱动程序在drivers/pci/pci.c，系统范围的定义在include/linux/pci.h。每一种体系结构都有一些特殊的PCI BIOS代码，Alpha AXP的位于arch/alpha/kernel/bios32.c

Interprocess Communication

全部在ipc目录。所有系统V IPC对象都包括ipc_perm数据结构，可以在include/linux/ipc.h中找到。系统V消息在ipc/msg.c中实现，共享内存在ipc/shm.c中，信号灯在ipc/sem.c。管道在ipc/pipe.c中实现。

Interrupt Handling

核心的中断处理代码几乎都是和微处理器（通常也和平台）相关。Intel中断处理代码在arch/i386/kernel/irq.c它的定义在incude/asm-i386/irq.h。

Device Drivers（设备驱动程序）

Linux核心源代码的大部分代码行在它的设备驱动程序中。Linux所有的设备驱动程序源代码都在drivers中，但是它们被进一步分类：

/block 块设备驱动程序比如ide（ide.c）。如果你希望查看所有可能包含文件系统的设备是如何初始化的，你可以看drivers/block/genhd.c中的device_setup()。它不仅初始化硬盘，也初始化网络，因为你安装nfs文件系统的时候需要网络。块设备包括基于IDE和SCSI设备。

/char 这里可以查看基于字符的设备比如tty，串行口等。

/cdrom Linux所有的CDROM代码。在这里可以找到特殊的CDROM设备（比如Soundblaster CDROM）。注意ide CD驱动程序是drivers/block中的ide-cd.c，而SCSI CD驱动程序在drivers/scsi/scsi.c中

/pci PCI伪驱动程序。这是一个观察PCI子系统如何被映射和初始化的好地方。Alpha AXP PCI整理代码也值得在arch/alpha/kernel/bios32.c中查看

/scsi 在这里不但可以找到所有的Linux支持的scsi设备的驱动程序，也可以找到所有的SCSI代码

/net 在这里可以找到网络设备驱动程序比如DEC Chip 21040 PCI以太网驱动程序在tulip.c中

/sound 所有的声卡驱动程序的位置

File Systems（文件系统）

EXT2文件系统的源程序都在fs/ext2/子目录，数据结构的定义在include/linux/ext2_fs.h,ext2_fs_i.h和ext2_fs_sb.h中。虚拟文件系统的数据结构在include/linux/fs.h中描述，代码是fs/*。Buffer cache和update 核心守护进程都是用fs/buffer.c实现的

Network（网络）

网络代码放在net子目录，大部分的include文件在include/net。BSD socket代码在net/socket.c，Ipv4 INET socket 代码在net/ipv4/af_inet.c中。通用协议的支持代码（包括sk_buff处理例程）在net/core中，TCP/IP网络代码在net/ipv4。网络设备驱动程序在drivers/net

Modules（模块）

核心模块代码部分在核心，部分在modules包中。核心代码全部在kernel/modules.c，数据结果和核心守护进程kerneld的消息则分别在include/linux/module.h和include/linux/kerneld.h中。你可能也希望在include/linux/elf.h中查看一个ELF目标文件的结构

Appendix A（附录A）

Linux Data Structures（Linux数据结构）

本附录列出了本书中描述的Linux使用的主要的数据结构。为了在页面上访得下，它们经过了少量的编辑。

Block_dev_struct

block_dev_struct数据结构用于登记可用的块设备，让buffer cache使用。它们放在blk_dev向量表中。

参见include/linux/blkdev.h

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

buffer_head

buffer_head数据结构存放buffer cache中一个块缓冲区的信息。

参见include/linux/fs.h

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

```
/* First cache line: */
unsigned long b_blocknr; /* block number */
kdev_t b_dev; /* device (B_FREE = free) */
kdev_t b_rdev; /* Real device */
unsigned long b_rsector; /* Real buffer location on disk */
struct buffer_head *b_next; /* Hash queue list */
struct buffer_head *b_this_page; /* circular list of buffers in one
page */
/* Second cache line: */
unsigned long b_state; /* buffer state bitmap (above) */
struct buffer_head *b_next_free;
unsigned int b_count; /* users using this block */
unsigned long b_size; /* block size */
/* Non-performance-critical data follows. */
char *b_data; /* pointer to data block */
unsigned int b_list; /* List that this buffer appears */
unsigned long b_flushtime; /* Time when this (dirty) buffer
* should be written */
unsigned long b_lru_time; /* Time when this buffer was
* last used. */
struct wait_queue *b_wait;
struct buffer_head *b_prev; /* doubly linked hash list */
struct buffer_head *b_prev_free; /* doubly linked list of buffers */
struct buffer_head *b_reqnext; /* request queue */
};
```

device

系统中的每一个网络设备都用一个device数据结构表示

参见include/linux/netdevice.h

```
struct device
{
/*
* This is the first field of the "visible" part of this structure
* (i.e. as seen by users in the "Space.c" file). It is the name
* the interface.
*/
char *name;
/* I/O specific fields */
unsigned long rmem_end; /* shmem "recv" end */
unsigned long rmem_start; /* shmem "recv" start */
unsigned long mem_end; /* shared mem end */
unsigned long mem_start; /* shared mem start */
unsigned long base_addr; /* device I/O address */
unsigned char irq; /* device IRQ number */
/* Low-level status flags. */
volatile unsigned char start, /* start an operation */
interrupt; /* interrupt arrived */
unsigned long tbusy; /* transmitter busy */
struct device *next;
/* The device initialization function. Called only once. */
int (*init)(struct device *dev);
/* Some hardware also needs these fields, but they are not part of
```

```
the usual set specified in Space.c. */
unsigned char if_port; /* Selectable AUI,TP, */
unsigned char dma; /* DMA channel */
struct enet_statistics* (*get_stats)(struct device *dev);
/*
* This marks the end of the "visible" part of the structure. All
* fields hereafter are internal to the system, and may change at
* will (read: may be cleaned up at will).
*/
/* These may be needed for future network-power-down code. */
unsigned long trans_start; /* Time (jiffies) of
last transmit */
unsigned long last_rx; /* Time of last Rx */
unsigned short flags; /* interface flags (BSD)*/
unsigned short family; /* address family ID */
unsigned short metric; /* routing metric */
unsigned short mtu; /* MTU value */
unsigned short type; /* hardware type */
unsigned short hard_header_len; /* hardware hdr len */
void *priv; /* private data */
/* Interface address info. */
unsigned char broadcast[MAX_ADDR_LEN];
unsigned char pad;
unsigned char dev_addr[MAX_ADDR_LEN];
unsigned char addr_len; /* hardware addr len */
unsigned long pa_addr; /* protocol address */
unsigned long pa_brdaddr; /* protocol broadcast addr*/
```

```
unsigned long pa_dstaddr; /* protocol P-P other addr*/
unsigned long pa_mask; /* protocol netmask */
unsigned short pa_alen; /* protocol address len */
struct dev_mc_list *mc_list; /* M'cast mac addrs */
int mc_count; /* No installed mcasts */
struct ip_mc_list *ip_mc_list; /* IP m'cast filter chain */
__u32 tx_queue_len; /* Max frames per queue */
/* For load balancing driver pair support */
unsigned long pkt_queue; /* Packets queued */
struct device *slave; /* Slave device */
struct net_alias_info *alias_info; /* main dev alias info */
struct net_alias *my_alias; /* alias devs */
/* Pointer to the interface buffers. */
struct sk_buff_head buffs[DEV_NUMBUFFS];
/* Pointers to interface service routines. */
int (*open)(struct device *dev);
int (*stop)(struct device *dev);
int (*hard_start_xmit) (struct sk_buff *skb,
struct device *dev);
int (*hard_header) (struct sk_buff *skb,
struct device *dev,
unsigned short type,
void *daddr,
void *saddr,
unsigned len);
int (*rebuild_header)(void *eth,
struct device *dev,
```

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

device_struct

device_struct数据结构用于登记字符和块设备（存放这个设备的名称和可能进行的文件操作）。Chrdevs和blkdevs向量表中的每一个有效的成员都分别代表一个字符或块设备。

参见fs/devices.c

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

file

每一个打开的文件、socket等等都用一个file数据结构代表

参见include/linux/fs.h

```
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 */
};
```

file_struct

file_struct数据结构描述了一个进程打开的文件

参见include/linux/sched.h

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

fs_struct

参见include/linux/sched.h

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

gendisk

gendisk数据结构存放硬盘的信息。用在初始化过程中找到磁盘，探测分区的时候

参见include/linux/genhd.h

```
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

VFS inode数据结构存放磁盘上的一个文件或目录的信息

参见include/linux/fs.h

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

```
off_t i_size;
time_t i_atime;
time_t i_mtime;
time_t i_ctime;
        unsigned long i_blksize;
unsigned long i_blocks;
unsigned long i_version;
unsigned long i_nrpages;
struct semaphore i_sem;
struct inode_operations *i_op;
struct super_block *i_sb;
struct wait_queue *i_wait;
struct file_lock *i_flock;
struct vm_area_struct *i_mmap;
struct page *i_pages;
struct dquot *i_dquot[MAXQUOTAS];
struct inode *i_next, *i_prev;
struct inode *i_hash_next, *i_hash_prev;
struct inode *i_bound_to, *i_bound_by;
struct inode *i_mount;
unsigned short i_count;
unsigned short i_flags;
unsigned char i_lock;
unsigned char i_dirt;
unsigned char i_pipe;
unsigned char i_sock;
unsigned char i_seek;
```

```
unsigned char i_update;
unsigned short i_writecount;
union {
        struct pipe_inode_info pipe_i;
        struct minix_inode_info minix_i;
        struct ext_inode_info ext_i;
        struct ext2_inode_info ext2_i;
        struct hpfs_inode_info hpfs_i;
        struct msdos_inode_info msdos_i;
        struct umsdos_inode_info umsdos_i;
        struct iso_inode_info isofs_i;
        struct nfs_inode_info nfs_i;
        struct xiafs_inode_info xiafs_i;
        struct sysv_inode_info sysv_i;
        struct affs_inode_info affs_i;
        struct ufs_inode_info ufs_i;
        struct socket socket_i;
        void *generic_ip;
} u;
};
```

ipc_perm

ipc_perm数据结构描述一个系统V IPC对象的访问权限

参见include/linux/ipc.h

```
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

irqaction数据结构描述系统的中断处理程序

参见include/linux/interrupt.h

```
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

Linux理解的每一个二进制文件格式都用一个linux_binfmt数据结构表示

参见include/linux/binfmt.h

```
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

mem_map_t数据结构（也叫做page）用于存放每一个物理内存页的信息

参见include/linux/mm.h

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

```
unsigned long map_nr; /* page->map_nr == page - mem_map */
} mem_map_t;
```

mm struct

mm_struct数据结构用于描述一个任务或进程的虚拟内存

参见include/linux/sched.h

```
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

系统中的每一个PCI总线用一个pci_bus数据结构表示

参见include/linux/pci.h

```
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

系统中的每一个PCI设备，包括PCI-PCI和PCI-ISA桥设备都用一个pci_dev数据结构代表

参见include/linux/pci.h

```
/*
* 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

request用于向系统中的块设备发出请求。请求都是从/向buffer cache读/写数据块

参见include/linux/blkdev.h

```
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

每一个rtable数据结构都存放向一个IP主机发送报文的路由的信息。Rtable数据结构在IP route缓存中使用。

参见include/net/route.h

```
struct rtable
{
struct rtable *rt_next;
__u32 rt_dst;
__u32 rt_src;
__u32 rt_gateway;
atomic_t rt_refcnt;
atomic_t rt_use;
unsigned long rt_window;
atomic_t rt_lastuse;
struct hh_cache *rt_hh;
struct device *rt_dev;
```

```
unsigned short rt_flags;
unsigned short rt_mtu;
unsigned short rt_irtt;
unsigned char rt_tos;
};
```

semaphore

信号灯用于保护重要数据结构和代码区域。

参见include/asm/semaphore.h

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

sk_buff

sk_buff数据结构当网络数据在协议层之间移动的过程中描述网络数据。

参见include/linux/sk_buff.h

```
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

每一个sock数据结构都存放一个BSD socket中和协议相关的信息。例如，对于一个INET socket，这个数据结构会存放所有的TCP/IP和UDP/IP相关的信息

参见include/linux/net.h

```
struct sock
{
/* This must be first. */
struct sock *sklist_next;
struct sock *sklist_prev;
struct options *opt;
atomic_t wmem_alloc;
atomic_t rmem_alloc;
unsigned long allocation; /* Allocation mode */
__u32 write_seq;
__u32 sent_seq;
__u32 acked_seq;
__u32 copied_seq;
__u32 rcv_ack_seq;
unsigned short rcv_ack_cnt; /* count of same ack */
__u32 window_seq;
__u32 fin_seq;
__u32 urg_seq;
```

```
__u32 urg_data;
__u32 syn_seq;
int users; /* user count */
/*
* Not all are volatile, but some are, so we
* might as well say they all are.
*/
volatile char dead,
             urginline,
     intr,
     blog,
     done,
     reuse,
     keepopen,
     linger,
     delay_acks,
     destroy,
     ack_timed,
     no_check,
     zapped,
     broadcast,
     nonagle,
     bsdism;
unsigned long lingertime;
int proc;
struct sock *next;
struct sock **pprev;
```

```
struct sock *bind_next;
struct sock **bind_pprev;
struct sock *pair;
int hashent;
struct sock *prev;
struct sk_buff *volatile send_head;
struct sk_buff *volatile send_next;
struct sk_buff *volatile send_tail;
struct sk_buff_head back_log;
struct sk_buff *partial;
struct timer_list partial_timer;
long retransmits;
struct sk_buff_head write_queue,
receive_queue;
struct proto *prot;
struct wait_queue **sleep;
__u32 daddr;
__u32 saddr; /* Sending source */
__u32 rcv_saddr; /* Bound address */
unsigned short max_unacked;
unsigned short window;
__u32 lastwin_seq; /* sequence number when we last
        updated the window we offer */
__u32 high_seq; /* sequence number when we did
        current fast retransmit */
volatile unsigned long ato; /* ack timeout */
volatile unsigned long lrcvtime; /* jiffies at last data rcv */
```

```
volatile unsigned long idletime; /* jiffies at last rcv */
unsigned int bytes_rcv;
/*
* mss is min(mtu, max_window)
*/
unsigned short mtu; /* mss negotiated in the syn's */
volatile unsigned short mss; /* current eff. mss - can change */
volatile unsigned short user_mss; /* mss requested by user in ioctl */
volatile unsigned short max_window;
unsigned long window_clamp;
unsigned int ssthresh;
unsigned short num;
volatile unsigned short cong_window;
volatile unsigned short cong_count;
volatile unsigned short packets_out;
volatile unsigned short shutdown;
volatile unsigned long rtt;
volatile unsigned long mdev;
volatile unsigned long rto;
volatile unsigned short backoff;
int err, err_soft; /* Soft holds errors that don't
cause failure but are the cause
of a persistent failure not
just 'timed out' */
unsigned char protocol;
volatile unsigned char state;
unsigned char ack_backlog;
```

```
unsigned char max_ack_backlog;
unsigned char priority;
unsigned char debug;
int rcvbuf;
int sndbuf;
unsigned short type;
unsigned char localroute; /* Route locally only */
/*
* This is where all the private (optional) areas that don't
* overlap will eventually live.
*/
union
{
struct unix_opt af_unix;
#if defined(CONFIG_ATALK) || defined(CONFIG_ATALK_MODULE)
struct atalk_sock af_at;
#endif
#if defined(CONFIG_IPX) || defined(CONFIG_IPX_MODULE)
struct ipx_opt af_ipx;
#endif
#ifdef CONFIG_INET
struct inet_packet_opt af_packet;
#ifdef CONFIG_NUTCP
struct tcp_opt af_tcp;
#endif
#endif
} protinfo;
```

```
/*
* IP 'private area'
*/
int ip_ttl; /* TTL setting */
int ip_tos; /* TOS */
struct tcphdr dummy_th;
struct timer_list keepalive_timer; /* TCP keepalive hack */
struct timer_list retransmit_timer; /* TCP retransmit timer */
struct timer_list delack_timer; /* TCP delayed ack timer */
int ip_xmit_timeout; /* Why the timeout is running */
struct rtable *ip_route_cache; /* Cached output route */
unsigned char ip_hdrincl; /* Include headers ? */
#ifdef CONFIG_IP_MULTICAST
int ip_mc_ttl; /* Multicasting TTL */
int ip_mc_loop; /* Loopback */
char ip_mc_name[MAX_ADDR_LEN]; /* Multicast device name */
struct ip_mc_socklist *ip_mc_list; /* Group array */
#endif
/*
* This part is used for the timeout functions (timer.c).
*/
int timeout; /* What are we waiting for? */
struct timer_list timer; /* This is the TIME_WAIT/receive
* timer when we are doing IP
*/
struct timeval stamp;
/*
```

```
* Identd
*/
struct socket *socket;
/*
* Callbacks
*/
void (*state_change)(struct sock *sk);
void (*data_ready)(struct sock *sk,int bytes);
void (*write_space)(struct sock *sk);
void (*error_report)(struct sock *sk);
};
```

socket

每一个socket数据结构都存放一个BSD socket的信息。它不会独立存在，实际上是VFS inode数据结构的一部分

参见include/linux/net.h

```
struct socket {
short type; /* SOCK_STREAM, ... */
socket_state state;
long flags;
struct proto_ops *ops; /* protocols do most everything */
void *data; /* protocol data */
struct socket *conn; /* server socket connected to */
struct socket *iconn; /* incomplete client conn.s */
struct socket *next;
```

```
struct wait_queue **wait; /* ptr to place to wait on */
struct inode *inode;
struct fasync_struct *fasync_list; /* Asynchronous wake up list */
struct file *file; /* File back pointer for gc */
};
```

task_struct

每一个task_struct描述系统中的一个任务或进程

参见include/linux/sched.h

```
struct task_struct {
/* these are hardcoded - don't touch */
volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
long counter;
long priority;
unsigned long signal;
unsigned long blocked; /* bitmap of masked signals */
unsigned long flags; /* per process flags, defined below */
int errno;
long debugreg[8]; /* Hardware debugging registers */
struct exec_domain *exec_domain;
/* various fields */
struct linux_binfmt *binfmt;
struct task_struct *next_task, *prev_task;
struct task_struct *next_run, *prev_run;
unsigned long saved_kernel_stack;
unsigned long kernel_stack_page;
int exit_code, exit_signal;
```

```
/* ??? */
unsigned long personality;
int dumpable:1;
int did_exec:1;
int pid;
int pgrp;
int tty_old_pgrp;
int session;
/* boolean value for session group leader */
int leader;
int groups[NGROUPS];
/*
* pointers to (original) parent process, youngest child, younger sibling,
* older sibling, respectively. (p->father can be replaced with
* p->p_pptr->pid)
*/
struct task_struct *p_opptr, *p_pptr, *p_cptr,
*p_ysptr, *p_osptr;
struct wait_queue *wait_chldexit;
unsigned short uid,euid,suid,fsuid;
unsigned short gid,egid,sgid,fsgid;
unsigned long timeout, policy, rt_priority;
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
long utime, stime, cutime, cstime, start_time;
/* mm fault and swap info: this can arguably be seen as either
```

```
mm-specific or thread-specific */
unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
int swappable:1;
unsigned long swap_address;
unsigned long old_maj_flt; /* old value of maj_flt */
unsigned long dec_flt; /* page fault count of the last time */
unsigned long swap_cnt; /* number of pages to swap on next pass */
/* limits */
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
/* file system info */
int link_count;
struct tty_struct *tty; /* NULL if no tty */
/* ipc stuff */
struct sem_undo *semundo;
struct sem_queue *semsleeping;
/* ldt for this task - used by Wine. If NULL, default_ldt is used */
struct desc_struct *ldt;
/* tss for this task */
struct thread_struct tss;
/* filesystem information */
struct fs_struct *fs;
/* open file information */
struct files_struct *files;
/* memory management info */
struct mm_struct *mm;
```

```
/* signal handlers */
struct signal_struct *sig;
#ifdef __SMP__
int processor;
int last_processor;
int lock_depth; /* Lock depth.
We can context switch in and out
of holding a syscall kernel lock... */
#endif
};
```

timer_list

timer_list数据结构用于实现进程的实时计时器。

参见include/linux/timer.h

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

tq_struct

每一个任务队列（tq_struct）数据结构都存放正在排队的工作的信息。通常是一个设备驱动程序需要的任务，但是不需要立即完成。

参见include/linux/tqueue.h

```
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

每一个vm_area_struct数据结构描述一个进程的一个虚拟内存区

参见include/linux/mm.h

```
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 */
};
```

## Append B（附录B）

## The Alpha AXP Processor

Alpha AXP体系结构是一个为了速度而设计的64位的加载/存储（load/store）RISC体系结构。所有的寄存器都是64位长的：32个整数寄存器和32个浮点寄存器。第31个整数寄存器和第31个浮点寄存器用于null操作：读取它们得到0，写向它们没有任何结果。所有的指令和内存操作（不管是读或写）都是32位。只要具体的实现遵循这种体系结构，允许不同的实现方式。

没有直接在内存中操作数值的指令：所有的数据操作都是在寄存器之间进行。所以，如果你希望增加内存中一个计数器，你必须首先把它读到一个积存其中，修改之后再写回去。只有通过一个指令写向一个寄存器或内存位置，而另一个读取这个寄存器或内存位置，指令之间才能过相互作用。Alpha AXP的一个有趣的特点是它有可以产生标志位的指令，比如测试两个整数是否相等，这个结构不是存放到处理器的一个状态寄存器，而是存放到第三个寄存器。初看上去比较奇怪，但是不依赖于状态寄存器意味着可以更容易地让这个CPU每一个循环执行多个指令。执行过程中使用无关的寄存器的指令不需要互相等待，如果只有一个状态寄存器则必须等待。没有对于内存的直接操作以及数目众多的寄存器对于同时多条指令也有帮助。

Alpha AXP体系结构使用一系列子例程，叫做体系结构的授权库代码（privileged architecture library code PALcode）。PALcode和操作系统、Alpha AXP体系的CPU具体实现和系统硬件相关。这些子例程向操作系统提供上下文切换、中断、异常和内存管理的基本支持。这些子例程可以被硬件调用或通过CALL_PAL指令调用。PALcode用标准的Alpha AXP汇编程序编写，带有一些特殊扩展的实现，用于提供直接访问低级硬件的功能，例如内部处理器的寄存器。PALcode在PAL模式运行，这是一个特权的模式，会停止一些系统事件的发上，并允许PALcode完全控制系统的物理硬件。

# Appendix C（附录C）

## Useful Web and FTP Sites（有用的web和ftp站点）

http://www.azstarnet.com/~axplinux This is David Mosberger-Tang's Alpha AXP Linux web site and it is the place to go for all of the Alpha AXP HOW-TOs. It also has a large number of pointers to Linux and Alpha AXP specific information such as CPU data sheets.

http://www.redhat.com/ Red Hat's web site. This has a lot of useful pointers.

ftp://sunsite.unc.edu This is the major site for a lot of free software. The Linux specfic software is held in pub/Linux.

http://www.intel.com Intel's web site and a good place to look for Intel chip information.

http://www.ssc.com/lj/index.html The Linux Journal is a very good Linux magazine and well worth the yearly subscription for its excellent articles.

http://www.blackdown.org/java-linux.html This is the primary site for information on Java on Linux.

ftp://tsx-11.mit.edu/ ~ftp/pub/linux MIT's Linux ftp site.

ftp://ftp.cs.helsinki.fi/pub/Software/Linux/Kernel Linus's kernel sources.

http://www.linux.org.uk The UK Linux User Group.

http://sunsite.unc.edu/mdw/linux.html Home page for the Linux Documentation Project,

http://www.digital.com Digital Equipment Corporation's main web page.

http://altavista.digital.com DIGITAL's Altavista search engine. A very good place to search for information within the web and news groups.

http://www.linuxhq.com The Linux HQ web site holds up to date offical and unoffical patches as well as advice and web pointers that help you get the best

set of kernel sources possible for your system.

http://www.amd.com The AMD web site.

http://www.cyrix.com Cyrix's web site.

Appendix D(附录D)

The GNU General Public

License

Printed below is the GNU General Public License (the GPL or copyleft), under which Linux is licensed. It is reproduced here to clear up some of the confusion about Linux's copyright status|Linux is not shareware, and it is not in the public domain. The bulk of the Linux kernel is copyright c 1993 by Linus Torvalds, and other software and parts of the kernel are copyrighted by their authors. Thus, Linux is copyrighted, however, you may redistribute it under the terms of the GPL printed below.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

D.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software{to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using

it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not re ect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in e
ect making the program proprietary. To prevent this, we have made it

clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

D.2 Terms and Conditions for Copying, Distribu-

tion, and Modification 0. This License applies to any program or other work which contains a notice

placed by the copyright holder saying it may be distributed under the terms of this General Public License. The \Program", below, refers to any such program or work, and a \work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term \modification".) Each licensee is addressed as \you". Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent ofhaving been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appro-

priately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it,

thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights

to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program. In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the correspond-ing source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to dis-tribute corresponding source code. (This alternative isallowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface defi-nition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary

form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law ifyou do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement

or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any par-ticular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity ofany such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which isimplemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permit-

ted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and \any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMIT-TED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM \AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIM-ITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO

THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH
YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME
THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORREC-
TION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAWOR AGREED
TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER
PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM
AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, IN-
CLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUEN-
TIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE
THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA
OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED
BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO
OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER
OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.
END OF TERMS AND CONDITIONS

D.3 Appendix: How to Apply These Terms to Your
New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way toachieve this is to make itfree software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the \copyright" line and a pointer to where the full notice is found.

hone line to give the program's name and a brief idea of what it does.i

`Gnomovision' (which makes passes at compilers) written by James Hacker.

hsignature of Ty Cooni, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Glossary（名词表）

Argument Functions and routines are passed arguments to process.

ARP Address Resolution Protocol. Used to translate IP addresses into physical hardware addresses.

Ascii American Standard Code for Information Interchange. Each letter of the alphabet is represented by an 8 bit code. Ascii is most often used to store written characters.

Bit A single bit of data that represents either 1 or 0 (on or off).

Bottom Half Handler Handlers for work queued within the kernel.

Byte 8 bits of data,

C A high level programming language. Most of the Linux kernel is written in C.

CPU Central Processing Unit. The main engine of the computer, see also microprocessor and processor.

Data Structure This is a set of data in memory comprised of fields,

Device Driver The software controlling a particular device, for example the NCR 810 device driver controls the NCR 810 SCSI device.

DMA Direct Memory Access.

ELF Executable and Linkable Format. This object file format designed by the Unix System Laboratories is now firmly established as the most commonly used

format in Linux.

EIDE Extended IDE.

Executable image A structured file containing machine instructions and data. This file can be loaded into a process's virtual memory and executed. See also program.

Function A piece of software that performs an action. For example, returning the bigger of two numbers.

IDE Integrated Disk Electronics.

Image See executable image.

IP Internet Protocol.

IPC Interprocess Communiction.

Interface A standard way of calling routines and passing data structures. For example, the interface between two layers of code might be expressed in terms of routines that pass and return a particular data structure. Linux's VFS is a good example of an interface.

IRQ Interrupt Request Queue.

ISA Industry Standard Architecture. This is a standard, although now rather dated, data bus interface for system components such as oppy disk drivers.

Kernel Module A dynamically loaded kernel function such as a filesystem or a device driver.

Kilobyte A thousand bytes of data, often written as Kbyte,

Megabyte A million bytes of data, often written as Mbyte,

Microprocessor A very integrated CPU. Most modern CPUs are Microprocessors.

Module A file containing CPU instructions in the form of either assembly language instructions or a high level language like C.

Object file A fille containing machine code and data that has not yet been linked with other object files or libraries to become an executable image.

Page Physical memory is divided up into equal sized pages.

Pointer A location in memory that contains the address of another location in memory,

Process This is an entity which can execute programs. A process could be thought of as a program in action.

Processor Short for Microprocessor, equivalent toCPU.

PCI Peripheral Component Interconnect. A standard describing how the peripheral components of a computer system may be connected together.

Peripheral An intelligent processor that does work on behalf of the system's CPU. For example, an IDE controller chip,

Program A coherent set of CPU instructions that performs a task, such as printing \hello world". See also executable image.

Protocol A protocol is a networking language used to transfer application data between two cooperating processes or network layers.

Register A location within a chip, used to store information or instructions.

Routine Similar to a function except that, strictly speaking, routines do not return values.

SCSI Small Computer Systems Interface.

Shell This is a program which acts as an interface between the operating system and a human user. Also called a command shell, the most commonly used shell in Linux is the bash shell.

SMP Symmetrical multiprocessing. Systems with more than one processor which fairly share the work amongst those processors.

Socket A socket represents one end of a network connection, Linux supports the BSD Socket interface.

Software CPU instructions (both assembler and high level languages like C) and data. Mostly interchangable with Program.

System V A variant ofUnix

TM

produced in 1983, which included, amongst other

things, System V IPC mechanisms.

TCP Transmission Control Protocol.

Task Queue A mechanism for deferring work in the Linux kernel.

UDP User Datagram Protocol.

Virtual memory A hardware and software mechanism for making the physical memory in a system appear larger than it actually is.

Bibliography（参考书目）

[1] Richard L. Sites. Alpha Architecture Reference Manual Digital Press

[2] Matt Welsh and Lar Kaufman. Running Linux O'Reilly & Associates, Inc, ISBN 1-56592-100-3

[3] PCI Special Interest Group PCI Local Bus Specification

[4] PCI Special Interest Group PCI BIOS ROM Specification

[5] PCI Special Interest Group PCI to PCI Bridge Architecture Specification

[6] Intel Peripheral Components Intel 296467, ISBN 1-55512-207-8

[7] Brian W. Kernighan and Dennis M. Richie The C Programming Language Prentice Hall, ISBN 0-13-110362-8

[8] Steven Levy Hackers Penguin, ISBN 0-14-023269-9

[9] Intel Intel486 Processor Family: Programmer's Reference Manual Intel

[10] Comer D. E. Interworking with TCP/IP, Volume 1 - Principles, Protocols and Architecture Prentice Hall International Inc

[11] David Jagger ARM Architectural Reference Manual Prentice Hall, ISBN 0-13-736299-4

**Unix/Linux作坊**