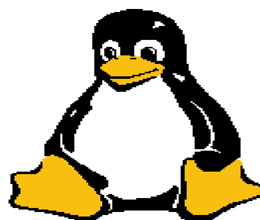
彎曲評論

科技 · 人物 · 潮流



Linux核心（The Linux Kernel）

（下）

"First, they ignored us; then they laughed at us; then they fighted us; then we win."

--From 1st Linux Conference

原著：David A Rusling

编译：陈怀临等

第八章 设备驱动程序

操作系统的目的之一就是掩盖掉各种硬件的特殊性。使得系统中的硬件设备对于用户而言是透明的，例如，不管底层是什么样的物理设备，虚拟文件系统提供一个一致的，安装好的文件系统。本章将描述 Linux 核心如何管理系统中的物理设备。

系统中 CPU 不是唯一的智能设备，每一个物理外设都有其设备控制器。键盘，鼠标和串行接口由多功能卡(SuperIO)控制，IDE 磁盘由 IDE 控制器掌握，SCSI 磁盘有 SCSI 控制器控制。每一个于硬件控制器都有其自己的控制和状态寄存器(CSR)。这些 CSR 在不同的设备中是不一样的。

一个 Adaptec 2940 SCSI 控制器的 CSR 与 NCR810 SCSI 控制器差别很大。CSR 用来启动和停止一个设备，用来初始化一个设备和检测故障。用来管理系统中硬件控制器的代码位于 Linux 核心中，而不是在每个应用程序中。用来管理硬件控制器的软件通常叫做设备驱动程序。Linux 核心的设备驱动程序基本上是一些共享库(Shared Library),在库中含有一些特权的，常住内存的，一些用来处理底层硬件的例程。Linux 的设备驱动程序用来处理各种硬件的多样性。

操作系统的基本功能之一是对设备处理的抽象化。所有的物理设备被当做正规的文件来处理，可以被"打开"，"关闭"，"读"和"写"，就像我们用系统调用处理文件一样。(译者注："文件"是一个逻辑上的概念；设备是一个实体。这里谈的是把设备抽象在/dev 文件系统下。)系统中每一个设备都对应一个设备特殊文件(device special file)，例如，系统中的第一个 IDE 磁盘的设备文件名是/dev/hda。对于块设备(如，磁盘)和字符设备，它们的的设备特殊文件通常是通过 mknod 命令用主设备号和次设备号来描述和创建。(译者注：主设备号和次设备号用来定位系统中两个表。一个主设备对应一个设备驱动程序。次设备的含义是系统中可以存在多个设备属于同一类，比如多个 IDE 磁盘。 但它们只需要一个同样的设备驱动程序来管理。)网络设备也同样是一个设备特殊文件，但它是由 Linux 核心来创建当系统发现并初始化网络控制器的时候。被同样一个设备驱动程序所管理的所有设备拥有一个同样的主设备号。次设备号用来区分不同的设备和设备控制器。例如，每个 IDE 磁盘主设备的每个分区都有个不同的次设备号。所以，/dev/hda2，这个第 2 个分区的主设备号是 3，次设备号是 2。Linux 将系统调用中(比如将一个文件系统安装在一个块设备上)传递过来的设备特殊文件名映射到相应的设备驱动程序(根据其相应的主设备名)和许多系统表中，如字符设备表，chrdevs。

Linux 支持三种硬件设备类型：字符，块和网络设备。字符设备的读写不需要缓冲，例如系统的串行接口/dev/cua0 和/dev/cua1。块设备的读和写只能以块的单位来进行，块的大小一般是 512 字节或 1024 字节。块设备的读写是通过缓冲 Cache 并且可以被随机存取。随机存取意味著你可以定位块设备的任一个块并进行读取；块设备的存取可以通过其设备特殊文件，但更通常的是通过文件系统。只有块设备支持文件系统的安装(Mount)。网络设备的存取是通过 BSD 的 Socket 接口和网络子系统(请参阅网络章节)。

Linux 支持许多不同的设备驱动程序(Linux 的优点之一)。它们都具备一些共同的属性：

核心态：

设备驱动程序是核心的一部份，就象核心中其他代码一样，如果不正确运行，会严重地毁坏系统。一个写的不好的驱动程序 可能使系统崩溃，并可能将文件系统打乱丢失数据.(译者注：作者在这提"核心态"的目的是指核心态下运行的代码可以几乎完全控制一个系统。)

核心接口：

设备驱动程序必须提供一个标准的接口给 Linux 核心或相应的子系统。例如，终端驱动程序提供一个文件 I/O 接口给 Linux 核心；SCSI 设备驱动程序提供一个 SCSI 设备接口给 SCSI 子系统。SCSI 设备接口提供文件 I/O，SCSI 子系统提供缓冲机制。

核心机制和服务：

设备驱动程序利用标准的核心服务，如内存分配，中断传送，等待队列来运行。

可装卸的：

大多数的 Linux 设备驱动程序可以在需要时被载进系统作为核心的一个模块；可以被卸下当不再被使用。这使的核心的自适应性非常好，系统的资源可以有效地被利用。(译者注：读者可以联想一下 Windows 操作系统中的 DLL(Dynamic Link Library)的概念)

可重构的：

Linux 设备驱动程序可以被构造进核心。当核心重新编译时，那些设备就是可重构的。

动态的：

当系统启动时，每一个设备驱动程序进行初始化，寻找其控制的设备。如果核心中一个设备驱动程序所对应的控制设备不存在(译者注：例如没有安装 SCSI 磁盘虽然系统有 SCSI 驱动程序)，也没有关系。这种情况下，系统中只不过是多了一个"多余的"驱动程序，占用了一些系统内存而已。对系统本身无碍。

8.1 检测与中断

每次设备接受一个命令，例如，"移动读磁头到软盘的第 42 扇区"，为了知道这个命令是否完成，设备驱动程序有两种选择：(不断地)检测这个设备或使用中断。(译者注："不断地"可以理解为："while(!(read_device_status_register()));"
)

检测一个设备意味著频繁地读(设备的)状态寄存器直到状态寄存器值的变化显示该设备已经完成请求。如果一个设备驱动程序是核心的一部份，上述行为将是一种灾难性的因为核心什么其他的也

不能作直到设备完成服务请求(译者注：这种方法极大地牺牲了系统的并发性。例如，其他进程全部被阻塞因为在核心态时，进程是不可被抢先的(或被剥夺的。)。一个替代的方法是使用一个系统定时器，设备驱动程序每隔一定时间调用设备驱动程序中的一个例程去检测服务命令是否完成。Linux 的软盘驱动程序就是这样工作的(译者注：不知道这种方法的优点何在？)。一种更有效的方法是使用中断。

中断驱动的设备驱动程序意味著：任何时候，它所管理的设备需要被处理时，该设备会发出一个中断。例如，每当一个 Ethernet 网卡控制器从网络上接收一个 Ethernet 数据包时，系统将会接收到一个中断。Linux 核心需要能够传送这个来自设备的中断到相应的设备驱动程序。这是通过该设备驱动程序(在初始化时)登记它所管理的中断号来达到的(译者注：请参阅中断处理章节)。它并且登记对应该中断的中断处理程序的地址。读者可以通过/proc/interrupts 来查阅哪一个中断别哪一个设备驱动程序所使用和其中断的类型。

```
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
```

这个申请中断资源的过程发生在驱动程序初始化的时候。系统中有一些中断号的使用是固定的，这是由于 IBM PC 体系结构的习惯遗留(Legacy)而来。例如，软盘控制器将一直使用中断 6。其他中断，如 PCI 设备的中断是在系统启动时动态分配的(译者注：请注意 ISA 设备与 PCI 设备在中断号占用方面的区别)。这种情况下，设备驱动程序在登记/申请系统中一个中断号之前，将首先探测它所管理的设备所将占用的 IRQ。对 PCI 中断，Linux 支持标准的 PCI BIOS 回调函数，以用来决定系统中设备的信息，包括其中断号。

一个中断如何被传递到 CPU 中，不同的硬件体系结构有不同的方法。但大多数系统中，中断的传递是通过一种特殊的模式，在这种模式下，系统其他的中断不会发生(译者注：这与处理中断时，屏蔽掉同等级的中断不是一回事，这里讲的是"传递"中断)。一个设备驱动程序的中断处理例程要尽可能地简单快速，从而 Linux 核心可以能够很快地撤销(Dismiss)这个中断并回到被中断之前

的现场(译者注：系统被中断时，有可能一个进程正在用户态下运行)。需要为接收/处理中断作很多工作的设备驱动程序可以使用核心的 bottom half handlers 或任务队列。该任务队列存放着那些将被待后调用的函数例程。

8.2 直接内存存取-DMA(Direct Memeory Access)

当数据量很小的情况下，使用中断驱动的设备驱动程序来从/向硬件设备传递数据是合理的，可以工作的很好。例如，一个 9600 波特率的 Modem 的传输速率近似于没毫秒(millisecond)一个字符。如果中断的延迟，硬件设备发出中断和设备驱动程序处理该中断的时间非常小(比如 2 毫秒)，那么数据传输的总体系统影响也非常小。这个 9600 波特率的 Modem 数据传输只要占用 0.002%的 CPU 处理时间。但是对于高速设备，比如硬盘控制器或 Ethernet 设备，它们的的传输速率要高很多。 一个 SCSI 设备能达到 40M 字节每秒。

直接内存存取，或 DMA，被提出用来解决传输上述大批量数据的问题。一个 DMA 控制器允许设备与内存之间发送或接收数据，但不影响处理器 CPU。PC 的 ISA DMA 控制器有 8 个 DMA 通道。第 7 个通道被用来为设备驱动程序服务。每一个 DMA 通道与一个 16 位的地址寄存器器和一个 16 位的计数寄存器相关联。当想要发起一次数据交换时，设备驱动程序设置相应 DMA 通道的地址，计数寄存器的大小，这次数据传输的方向(读或写)。然后通知设备可以启动 DMA 操作。当 DMA 结束时，设备才中断系统。因此，在数据传输的过程中，CPU 可以作其他的事情。

在使用 DMA 时，设备驱动程序必须额外小心。首先，对于 DMA 控制器而言，没有虚拟内存的概念，它所面对的，存取的是系统中的物理内存。因此被 DMA 的内存必须是一块连续的物理内存块。这意味著你不能通过 DMA 去"直接"存取进程的虚拟空间地址。当然一个方法是在 DMA 期间，可以锁住一个进程的一些物理页面，防止操作系统将其对换到 swap 空间上，从而保证 DMA 正确地完成。

DMA 通道是"短缺"资源，只有 7 个通道。而且通道不能被设备驱动程序间共享。就象中断一样，一个设备驱动程序必须能够知道哪一个 DMA 通道它要使用。有些设备使用固定的中断号，就象有些设备使用固定的中断号一样。例如，软驱设备使用的 DMA 通道一直是通道 2。有时一个设备的 DMA 通道可以由跳线来设置。许多以太(Ethernet)设备使用这种技术。一些更灵活的设备可以通过其 CSR 得知当前系统中哪些 DMA 通道是空着的。从而设备驱动程序可以随便挑选一个 DMA 通道使用。

Linux 通过一个向量数据结构 dma_chan(每一个 DMA 通道对应一个这样的数据结构)来掌握 DMA 通道的使用情况。dma_chan 结构中只包含两个域：一个指向一个字符串的指针，这个指针描述了这个 DMA 通道拥有者。另外一个域是一个标志，用来显示当前的 DMA 通道是空着的还是已被占据。当你使用命令"cat /proc/dma"时，其实是核心中的向量 dma_chan 被打印出来了。

8.3 存储器

在使用内存时，设备驱动程序要小心，因为它们是核心的一部份，故不能使用虚拟内存(译者注：作者在上一节和这里反复强调〝虚拟内存〞是因为运行在不同〝虚拟内存〞空间中的用户态进程之间不会发生冲突。操作系统的内存管理机制将负责。)。每一次设备驱动程序因为来了中断，或者 bottom half 或任务队列中的句柄被调度到而运行，当前的进程有可能被剥夺。所以设备驱动程序不能依赖于一个特殊的运行的进程，虽然设备驱动程序运行在一个进程的上下文上。象核心中的其他部份一样，设备驱动程序使用数据结构来管理跟踪它所控制的设备。这些数据结构可以静态地分配，作为设备驱动程序代码的(数据的)一部份，但这样会使得核心变的太大，造成资源的浪费。大多数设备驱动程序采用从核心中动态分配非页面的内存用来存储数据。

Linux 提供核心内存分配和释放的例程以供设备驱动程序使用。核心内存的分配是以 2 的幂次方为单位的。例如，128 字节或 512 字节即使设备驱动程序需要的内存量少于这些值。设备驱动程序申请的(被分配的)字节数被〝凑〞到下一个块的边界处。这种方法使得内存的释放回收更容易因为系统可以将这些小的空闲块合并成更大的内存块。(译者注：以 2 的幂次方为单位进行内存分配可以减少系统中内存被弄的零碎。
)

当核心内存被申请时，Linux 有可能要作许多额外的工作。如果剩余的内存太下的话，一些物理页面需要被丢弃或写进对换磁盘空间。通常地，Linux 将这个处理挂起并放到一个等待队列中直到系统中有足够的物理内存。当然不是所有的设备驱动程序(至少 Linux 核心代码)都希望这样被处理。所以当不能立刻分配内存时，核心内存分配例程可以直接返回一个〝失败〞。如果设备驱动程序希望用 DMA 与被分配的内存来交换数据，它可以指定这片内存是 DMA'able 的。这种情况下 Linux 核心需要了解系统中什么地方构成了 DMA'able 的内存。

8.4 设备驱动程序与核心的接口

Linux 核心必须能够通过一些标准的方法来和设备驱动程序接口。每一类设备驱动程序，(字符，块和网络)都提供一个一致的，共同的接口给核心以用来核心向它们申请服务。这些共同接口

(common interfaces)意味著核心可以将这些不同的设备和其驱动程序一样来对待。例如，SCSI 和 IDE 磁盘的行为是不同的。但 Linux 核心对它们使用一个同样的接口进行操作。

Linux 是非常动态的，可重构的。每次一个 Linux 核心启动时，可能遇到不同的物理设备，因此需要不同的相应的设备驱动程序。在核心重新构建(Build)的时候，Linux 允许通过配置文件将设备驱动程序带进核心。当这些驱动程序在机器启动时初始化的时候，有可能系统中并没不存在相应的物理设备。有些驱动程序可以在需要时被装载进入核心。为了处理设备驱动程序的这种动态特性，系统要求设备驱动程序在初始化时向系统进行登记。Linux 核心负责维护一些含有登记了的设备驱动程序的表。这些表中包含了一些例程(rountines)的指针和其他一些信息以用来支持核心与那些设备的接口。

8.4.1 字符设备

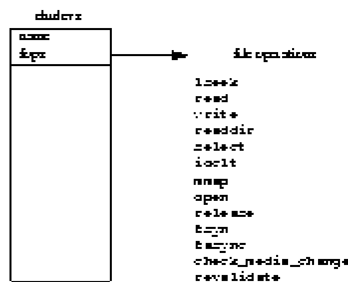


图 8.1 字符设备

字符设备，Linux 中最简单的设备，是通过”文件‟的形式被存取。应用程序使用标准的系统调用‟打开”，‟读”，‟写”，和‟关闭”字符设备就像它是一个文件一样，即使这个设备是一个被 PPP 监控程序(Daemon)用来将 Linux 系统连接上网的 Modem。当一个字符设备初始化时，它的设备驱动程序在 Linux 核心中登记，通过添加一个入口项(Entry)在含有 device_struct 数据结构的 chrdevs 向量中。这个设备的主设备号(例如，4 对于 tty 设备)被用来作为其在这个向量的索引。一个设备的主索引号是固定的。

chrdevs 向量的每一个入口项是一个 device_struct 数据结构，含有两个元素。一个指向那个登记”在这个入口处‟的设备驱动程序名字的指针；一个指向一系列文件操作函数地址的指针。这些文件操作函数位于这个字符设备的驱动程序里并负责处理相应的具体的文件操作如：打开，读，写和关闭。文件/proc/devices 中对于字符设备的内容是从 chrdevs 向量获取的。

当一个代表一个字符设备的字符特殊文件被打开时(例如/dev/cua0)，系统必须正确地工作保证相应的字符设备驱动程序的文件操作例程被调用。就象一个普通文件或目录一样，每一个设备特殊文件对应一个 VFS inode。这个 VFS inode(数据结构)中含有这个设备的主和次设备号。VFS inode 是当一个特殊设备文件名被查询时，由文件系统所创见。

每一个 VFS inode 与一套文件操作相联系。每当一个代表字符特殊文件的 VFS inode 被创建时，对应这个 VFS inode 的文件操作被设置成缺省的字符设备操作。

当一个字符特殊文件被一个应用程序打开时，这个"open"操作将使用这个设备的主设备号作为 chrdevs 向量的索引来查找对应这个设备的文件操作集的(例程的)地址。并且还要设置一个描述这个字符特殊文件的数据结构--file，使得 file 结构中关于文件操作的指针指向设备驱动程序中相应的部份。经过这些之后，所有用户层的文件操作将被映射到对于这个字符设备的设备驱动程序提供的文件操作。

8.4.2 块设备

块设备同样支持以文件的形式被存取。当遇到打开块设备操作时，用来提供一套对应的文件操作的机制与字符设备基本上是一样的。Linux 在 blkdevs 向量中维护登记了的块设备。与 chrdevs 向量一样，blkdevs 向量使用设备的主设备号作为其索引。向量的每一个入口仍是一个 device_struct 数据结构。与字符设备不同的是，这些数据结构是属于块设备的。SCSI 设备和 IDE 设备是其中两个例子。这些设备数据结构在核心中登记并为核心提供对应于其设备的文件操作。对应于某类设备的设备驱动程序提供实现这些接口的细节。例如，一个 SCSI 设备驱动程序必须为 SCSI 子系统提供接口。SCSI 子系统利用这些接口，提供给核心一个一致的文件接口。

除了文件操作接口，每个块设备还必须提供缓冲区接口。每一个块设备驱动程序在一个 blk_dev 向量中添加其入口。blk_dev 向量的每个元素是一个 blk_dev_struct 数据结构。向量的索引仍然是设备的主设备号。blk_dev_struct 数据结构中含有一个请求例程的地址和一个指向"request"数据结构的指针。每一个"request"数据结构代表了一个从缓冲区到驱动程序的读或写数据块的请求。

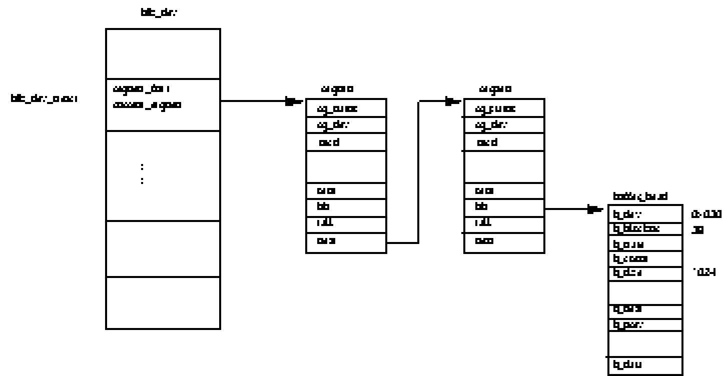


图 8.2 块设备的缓冲

每次一个缓冲区想要读或写一块数据从/到一个登记了的设备，它将插入一个"request"数据结构在blk_dev_struct中。由图 8.2所示，每一个申请含有一个指向一个或多个”buffer_head"的数据结构。每一个buffer_head是读或写一个块数据的请求(译者注：请参阅Linux数据结构章节)。Buffer_head结构是被缓冲区锁住的。因此有可能存在一个进程正在等待对这个缓冲区操作的完成。每一个“request"数据结构是从一个静态的链表中(all_requests)分配而来。如果一个请求(request)被加在一个空的请求队列上，设备驱动程序的请求函数(译者注：blk_dev_struc结构中函数指针所指向的函数)将被立即调用来处理这个请求队列。否则，驱动程序将顺序地处理请求队列中的所有请求。

一旦设备驱动程序完成一个请求，它必须从这个请求中移去每一个 buffer_head 结构，将它们标志成为更新并释放对其的锁。对一个 buffer_head 锁的释放将唤醒所有在睡眠中等待这个块操作完成的进程。一个例子是：当要解释一个文件名时，EXT2 文件系统必须从块设备中读取下一个 EXT2 目录项。这个进程将睡眠在那个含有目录项的 buffer_head 上直到被设备驱动程序唤醒。这个 request 数据结构将被回收从而可以被其他的块请求使用。

8.5 硬盘

磁盘将数据保存在磁盘片上，提供一种持久的存储方式。为了写数据，一个很小的磁头在磁盘片的表面上磁化小微粒(minute particles)。数据也通过磁头来读写。磁头能检测一个小微粒是否被磁化。

一个磁盘有一个或多个磁盘片(platters)组成。每个磁盘片的表面分成一些小的同心圆---磁道(track)。磁道 0 是最外层的磁道，最大编号的磁道最靠近圆心。一个柱面(cylinder)是指一个有同样编号的磁道集合。因此所有磁片上的所有磁道 5 构成了柱面 5。因为柱面的数目等于磁道德数目，我们经常看见人们使用柱面来描述磁盘。每一个磁道分为一些扇区(sectors)。一个扇区是一个硬盘读或写的最小单位。一个扇区的大小就是一个块的大小(译者注：换句话说，磁盘的读写是以块为单位的)。通常一个扇区的大小是 512 字节。一个扇区的大小通常是在磁盘格式化的时候就被确定了。

一个磁盘通常用其几何参数来描述，柱面的数目，磁头的数目和扇区的数目。例如，在启动时，Linux 描述一个 IDE 磁盘：

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

上述意味著这个磁盘含有 1050 个柱面，16 个磁头(8 个磁片)和 63 个扇区/每个磁道。如果每个扇区(或每个块)大小是 512 字节，这个磁盘的大小是 529200 字节。这个大小与系统声称的516M 大小不一致。这是因为磁盘的一些扇区已被用来存放磁盘分区信息。例外，一些磁盘可以自动地发现坏扇区并重心索引磁盘以绕过这些坏扇区。

硬盘可以更深一步地分为一些分区(partitions)。一个分区是一个用来作某个特殊用途的扇区的集合。将一个磁盘分区允许这个磁盘被几个操作系统使用，或允许用来作不同的用途。许多Linux 系统只有一个磁盘，但分为 3 个分区。一个含有 DOS 文件系统；一个含有 EXT2 文件系统；第 3 个是对换分区(译者注：用于虚拟内存管理系统)。一个硬盘的分区由一个分区表来描述。分区表中的每一个条目(entry)通过磁头，扇区和柱面，描述了这个分区的起始和结束地址。fdisk 支持 3 类分区类型。主分区，扩展分区和逻辑分区。扩展分区不是一个真正的分区，可以含有任意数目的逻辑分区。扩展和逻辑分区的发明是用来绕过系统中只允许 4 个主分区的限制。下面是用 fdisk 对一个含有 2 个主分区的磁盘分区的信息：

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

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

Expert command (m for help): p

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

Nr	AF	Hd	Sec	Cyl	Hd	Sec	Cyl	Start	Size	ID
1	00	1	1	0	63	32	477	32	978912	83
2	00	0	1	478	63	32	509	978944	65536	82
3	00	0	0	0	0	0	0	0	0	00
4	00	0	0	0	0	0	0	0	0	00

上面所示，第一个分区起始于柱面或磁道 0，磁头 1 和扇区 1，共延伸到 477 柱面，扇区 32 和磁头 63。因为一个磁道有 32 个扇区，整个磁盘有 64 个读/写磁头，所以这个分区占据的柱面是完整的。确省地，fdisk 自动将分区依照柱面的边界对齐。它起始于最外层的柱面(0)并向里面延伸 478 个柱面。第二个分区是对换分区，起始于下一个柱面(478)并一直延伸到最里面的那个柱面。

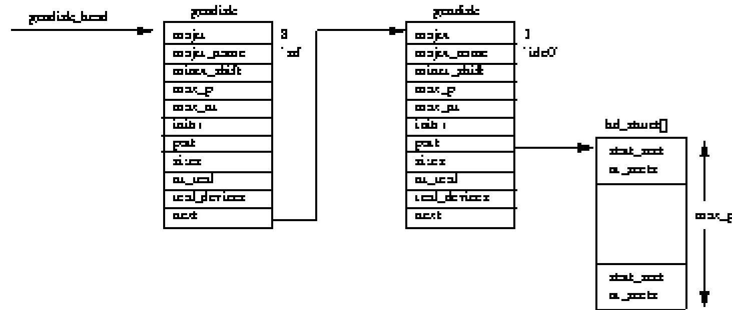


图 8.3 磁盘的链接表

在系统初始化期间，Linux 将映射系统中所有硬盘的拓朴。它发现有多少硬盘和其类型。另外，Linux 发现这些磁盘是如何分区的。这一切将体现在由指针 gendisk_head 指向的，一个元素为 gendisk 数据结构的链表中。当每个磁盘子系统初始化时，例如 IDE，它负责产生代表它所发现的磁盘的 gendisk 数据结构。这个行为发生在与登记它的文件操作接口，插入一个入口项在 blk_dev 数据结构中的同一时间。每一个 gendisk 数据结构有一个唯一的主设备号。这个主设备号与该块设备的主设备号一致。例如，SCSI 磁盘子系统产生一个单一的 gendisk 记录”sd�となど。记录中，其主设备号是 8。系统中所有的 SCSI 磁盘设备都拥有这个同样的主设备号。图 8.3 所示

是两个 gendisk 记录,第一个是为 SCSI 磁盘子系统，第二个是为 IDE 磁盘控制器，ide0, 主磁盘控制器。

虽然磁盘子系统在初始化时创建 gendisk 记录，这些记录只在 Linux 进行分区检测时使用。然而，每个磁盘子系统维护一个自己的数据结构，以用来映射设备的主设备号和次设备号到物理磁盘的分区中。任何时刻当一个块设备被读或写，不管是来自缓冲区还是文件操作，核心将使用在块设备特殊文件中发现的主设备号(例如，/dev/sda2)，引导读或写操作指向正确的设备。值得注意的是各个设备驱动程序负责映射次设备号到具体的物理设备中。

8.5.1 IDE 磁盘

在 Linux 系统中，最常见的是 IDE(Integrated Disk Electronic)磁盘。IDE 是一个磁盘接口，而不是一个 I/O 总线，例如象 SCSI。每个 IDE 控制器可以支持多达 2 个磁盘。一个主(master)磁盘；一个副(slave)磁盘。主和副磁盘是通过设置磁盘的跳线来完成的。系统中的第一个 IDE 控制器叫做主 IDE 控制器。依此类推，下一个叫做第二 IDE 控制器。IDE 接口可以达到 3.3M 的传输速率。IDE 磁盘容量最大是 538M 字节。扩展 IDE(EIDE)可以达到 8.6G 字节和 16.6M 字节的传输速率每秒。IDE 和 EIDE 磁盘比 SCSI 磁盘要便宜。大多数 PC 机都有一个或多个 IDE 控制器。

Linux 依据发现 IDE 控制器的顺序对其上的 IDE 磁盘命名。在主控制器上的主磁盘是 /dev/hda，副磁盘是/dev/hdb。/dev/hdc 是第二个 IDE 控制器 上的主磁盘。IDE 子系统在核心中登记 IDE 主控制器，而不是磁盘。主控制器的主设备号是 3；副 IDE 控制器的主设备号是 22。这意味著如果系统有两个 IDE 控制器，在向量 blk_dev 和 blkdevs 中将插入两个 IDE 子系统记录在向量索引 3 和 22 的地方。从设备特殊文件名中可以体现这点。在主 IDE 控制器上的磁盘/dev/hda 和/dev/hdb 的主设备号是 3。任何对这两个设备特殊文件的操作都会被核心根据被访问的主设备号传到其对应的 IDE 子系统中。IDE 子系统将负责是哪一个 IDE 磁盘被申请，通过设备特殊文件的次设备号。次设备号里含有信息关于哪一个分区和哪一个磁盘。/dev/hdb 的设备标识是(3,64)。该磁盘上的第一个分区(/dev/hdb1)的设备标识是(3,65)。

8.5.2 IDE 子系统的初始化

IDE 磁盘一直贯穿在 IBM PC 机的历史。在这个期间，IDE 接口发生了许多变化。这使得 IDE 子系统的初始化变得越来越复杂。

Linux 最多可以支持 4 个 IDE 控制器。每个控制器将体现在向量 ide_hwifs 的 ide_hwif_t 数据结构中。每个 ide_hwif_t 中含有两个 ide_drive_t 数据结构，对应于可能的主和副 IDE 驱动器。IDE 子系统初始化期间，Linux 首先通过系统的 CMOS 中信息查看是否有磁盘。CMOS 的位置由系统的 BIOS 设定并可以告诉 Linux 什么 IDE 控制器和磁盘驱动器在系统中。Linux 从 BIOS 中得到磁盘的几何描述信息并为这些驱动器设立 ide_hwif_t 数据结构。目前越来越多的 PC 机使用包含了 PCI EIDE 控制器的 PCI 芯片集(chipsets)(例如，Intel's 82430 VX)。IDE 子系统使用 PCI BIOS 的回调(callback)来定位系统中的 PCI E(IDE)控制器，然后调用 PCI 专门的为这些控制器准备的例程(来初始化核心的向量结构)。

一旦每个 IDE 接口和控制器被发现，相应的 ide_hwif_t 数据结构将被建立以反映这些控制器和其上的磁盘。在运行期间，IDE 驱动程序向 I/O 空间的 IDE 命令寄存器写入命令。主 IDE 控制器的控制和状态寄存器的缺省 I/O 地址在 0x1F0 - 0x1F7。IDE 驱动程序在 Linux 的块缓冲数据结构中登记每个控制器，在 blk_dev 和 blkdevs 分别加入记录。IDE 驱动器还将申请占有某个中断。对于主 IDE 控制器，约定的中断号是 14；第二个 IDE 控制器是 15。然而，上述都可以别核心的命令选项所覆盖。IDE 驱动程序还要为每个 IDE 控制器在 gendisk 链中加入一个 gendisk 记录。这个链表用来查找所有在启动时发现的磁盘的分区表信息。

8.5.3 SCSI 磁盘

SCSI(Small Computer System Interface)总线是一个快速的端到端的数据总线。每个 SCSI 总线上支持 8 个设备，其中包含一或多个 hosts。每个 SCSI 设备必须有一个唯一的标识名(一般通过磁盘的跳线来设置)。总线上的两个设备可以同步地或异步地 32 位地交换数据。速率可达 40M 字节。SCSI 总线可在设备之间数据和状态信息，并且在发起者(initiator)和目标(target)之间，一个单一的事务(transaction)可以包含多达 8 个不同状态的信息。我们可以从来自总线上的 5 个信号来辨别 SCSI 总线上当前的状态(phase)。

BUS FREE (总线空)
 当前没有设备正占据总线。没有活跃的事务处理。
ARBITRATION (仲裁)
 一个 SCSI 设备试图占据总线。它通过将其 SCSI 标识数据"插入"地址线。(如有竞争，)SCSI 标识最大的获得总线。
SELECTION (选择)

当一个设备通过仲裁，成功地获得总线后，它必须向目标(Target)发出信号表示它想要对目标发出命令。这是通过将目标的 SCSI 标识放入地址线而达到的。

RESELECTION

SCSI 设备有可能断开连接在一个请求处理过程中。目标有可能等会儿重新选择发起者(initiator)。不是所有的 SCSI 设备都支持这个状态。

COMMAND

在此状态下，可以从发起者传到目标端传送 6,10 或 12 字节的命令。

DATA IN, DATA OUT

在这些状态下，数据在发起者和目标端之间传送。

STATUS

当所有命令都完成时，可以进入这个状态。允许目标发出一个状态字节以向发起者表明成功或失败。

MESSAGE IN, MESSAGE OUT

附加的信息将在发起端和目标端传送。

Linux 的 SCSI 子系统有两个基本元素组成。每一个都通过一些数据结构在核心中得到体现。

host

一个 SCSI host 是一个物理硬件，即一个 SCSI 控制器。NCR810 PCI SCSI 控制器就是一个 SCSI host 的例子。如果一个 Linux 系统存在多于一个的，同样类型的 SCSI 控制器，每一个将对应于不同的 SCSI host。这意味著一个 SCSI 设备驱动程序可能控制多于一个的 SCSI 控制器。SCSI host 一般都是 SCSI 命令的发起者。

device(设备)

最常见的 SCSI 设备就是 SCSI 磁盘。但 SCSI 标准还支持其他的几类设备，如磁带，CD-ROM 和通用(generic)SCSI 设备。SCSI 设备一般都是 SCSI 命令的目标。这些设备必须不同的对待。例如，可移动的介质 CD-ROMs 或磁带，Linux 需要检测介质是否被移动了。不同的设备类型有不同的主设备号，Linux 可以依此引导不同的对块设备的请求到其相应的 SCSI 设备类型上。

SCSI 子系统的初始化

由于 SCSI 总线和设备的动态特性，SCSI 子系统的初始化比较复杂。Linux 在系统启动时初始化 SCSI 子系统。它首先发现系统中的 SCSI 控制器(即 SCSI hosts)然后探测在所有 SCSI 总线上

的所有的设备。然后初始化这些设备。通过向核心提供一套规范的文件操作和缓冲区操作例程集，使得对于 Linux 核心系统来说“可见”。这个初始化的过程分为 4 个阶段：

首先，Linux 检测在核心构建时已加入核心的那些 SCSI hosts 或控制器上是否有设备需要控制。上面每个 SCSI hosts 在 builtin_scsi_hosts 向量中有一个入口记录相对应。每个记录是一个 Scsi_Host_Template 数据结构。Scsi_Host_Template 中含有一些函数指针。这些函数用来执行 SCSI hosts 的一些特定功能，如检测什么设备正挂在 SCSI hosts 上。这些例程 SCSI 子系统所调用，属于这种 hosts 设备类型的设备驱动程序的一部份。每个在其上存在 SCSI 设备的 SCSI host 将 Scsi_Host_Template 数据结构加入一个 Scsi_Host 结构到一个 scsi_hostslist 列表 链中。例如，如果一个系统有两个 NCR810 PCI SCSI 控制器，系统数据结构中将有两个 Scsi_Host 记录在 scsi_hostslist 列表链中。每一个 Scsi_Host 指向代表起设备驱动程序的 Scsi_Host_Template。

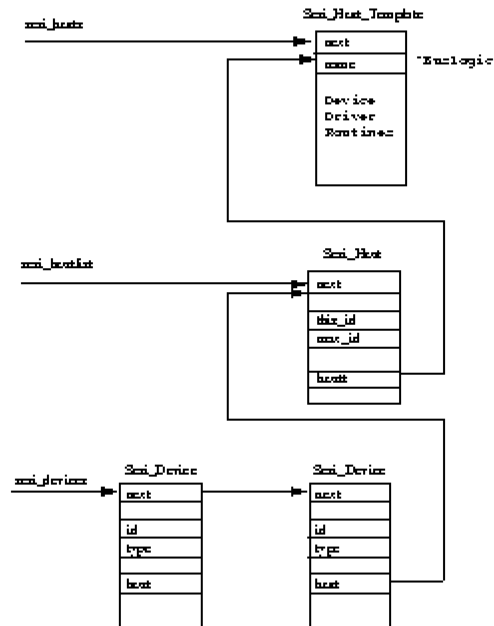


图 8.4 SCSI 数据结构

到现在，系统中所有的 SCSI host 都已备发现，SCSI 子系统必须知道在每个 host 上是些什么 SCSI 设备。SCSI 设备是按照 0-7 来标号的。每个 SCSI 设备的标号在其所安装的 host 上是唯一的。SCSI 标号通常是由设备上的跳线来设置的。SCSI 初始化代码通过发出 TEST_UNIT_READY 命令来探测一个 SCSI 总线上的 SCSI 设备。当一个设备存在并回答时，它的 SCSI 标识信息(包括厂商，设备型号和版本号)被读取，通过一个 ENQUIRY 命令。SCSI 命令

含在一个 Scsi_Cmnd 数据结构中。这些 Scsi_Cmnd 数据被传递到属于这个 SCSI host 的设备驱动程序的相关函数中。这些函数的指针在先前已被登记在 Scsi_Host_Template 结构中。每个已发现的 SCSI 设备将对应一个 Scsi_Device 数据结构.每个 Scsi_Device 数据结构指向其 host 的数据结构 Scsi_Host。所有的 Scsi_Device 结构链在一个叫做 scsi_devices 的链表上(译者注：Scsi_Device：SCSI 设备；Scsi_Host：SCSI host 或控制器；Scsi_Host_Template：SCSI host 的设备驱动程序入口)。图 8.4 所示是上述数据结构的关系。

SCSI 设备有四种：磁盘，磁带，CD 和 generic。每一种都分别在核心中以不同的主块设备号进行登记。当然，只有在发现系统中存在相关的 SCSI 设备时才会进行登记。每个 SCSI 类型，例如 SCSI 磁盘，维护一套其自己的表数据结构。这些表用来将来自核心的块操作请求映射到相应的设备驱动程序上或相应的 SCSI host 上。每一个 SCSI 类型在核心中对应于一个 Scsi_Device_Template 数据结构。这个结构中含有这种设备的信息和各种针对这种设备的例程地址。SCSI 子系统使用这些模板来调用对应于每种 SCSI 设备的 SCSI 类型函数。换句话说，如果 SCSI 子系统想要”attach‟(添加)一个 SCSI 磁盘，它将调用 SCSI 磁盘类型的”attach‟函数。Scsi_Device_Template 数据结构全部挂在 scsi_devicelist 链表上。(译者注：请注意 Scsi_Device_Template 与 Scsi_Host_Template 的关系和区别)

SCSI 子系统初始化的最后一步是对应于每一个登记了的 Scsi_Device_Template 调用”完成‟(finish)函数。对应于 SCSI 磁盘类型，这意味著旋转机器上所有的 SCSI 磁盘然后读取它们的几何参数。(根据获得的几何参数)，核心填写为每个 SCSI 磁盘填写 gendisk 数据结构在 gendisk 链中。

传送块设备请求

一旦 Linux 完成 SCSI 子系统的初始化，SCSI 设备就可以被使用了。每个存在相应设备的设备类型在核心进行了登记，从而 Linux 可以正确地将块设备请求定位/传送到正确的设备上。这些请求可以是来自 blk_dev 的缓冲区操作或来自 blkdevs 的文件操作。举一个例子，如果有一个 SCSI 磁盘含有一个或两个 EXT2 文件系统分区，当其中一个 EXT2 文件系统已被安装(mounted)，核心的缓冲区申请如何被定位到正确的磁盘上？

每个读或写一块 SCSI 磁盘数据的请求都导致在 blk_dev 向量中的 current_request 链表中加入一个新的 request 结构。如果这个申请队列正在被处理，缓冲区不需要做其他的事情。否则，必须提醒 SCSI 磁盘子系统去处理 request 队列。系统中每个 SCSI 磁盘对应于一个

Scsi_Disk 数据结构在 rscsi_disks 向量中。rscsi_disks 的索引使用了部份 SCSI 磁盘分区的次设备号信息。例如，/dev/sdb1 的主设备号是 8，次设备号是 17，其在 rscsi_disks 中的索引号是 1。 每个 Scsi_Disk 结构中含有一个指针指向代表这个设备的 Scsi_Device 结构。然后通过 Scsi_Device 指向对应的 Scsi_Host 结构(译者注：对应于这个 SCSI 磁盘所属的 SCSI 磁盘控制器)。 从缓冲区来的 request 结构被转换成描述 SCSI 命令的 Scsi_Cmd 结构并将其放入这个对应的 Scsi_Host 的队列中。当一旦要求的块数据被读或写完成之后，SCSI 设备驱动程序将会处理这些 Scsi_Cmd 结构。

8.6 网络设备

从 Linux 的网络子系统的角度而言，一个网络设备是用来发送和接收数据的一个"实体"或一个"东西"，比如一个 ethernet 网卡。每个网络设备在核心中对应于一个 device 数据结构。核心启动时，网络设备驱动程序初始化并登记其控制的设备。这个 device 结构中包含了关于这个设备的信息和一些函数的地址。这些函数被用来对各种高层的网络协议提供底层支撑。它们大多数是关于在物理网络设备上传输数据。网络设备使用标准的网络机制将接收到的数据向高层网络协议传送。所有传送的和接收的数据报(packets)都对应于 sk_buff 数据结构。sk_buff 是非常灵活的数据结构，允许网络协议头(network protocol headers)很轻松地被加入和移去。网络协议层如何使用网络设备，如何使用 sk_buff 来回传递数据，请参阅第 10 章网络。本章关于网络方面的重点是网络设备数据结构和网络设备如何被检错与初始化。

device 数据结构含有网络设备的如下信息：

Name

与用 mknod 命令来创建设备特殊文件的块和字符设备不同的是，网络设备特殊文件是当系统网络设备被发现并初始化时出现的。它们的名字是标准的。每一个名字代表了它是哪一种网络设备类型。属于同一类型的设备的名字从数字 0 开始往上走。因此 ethernet 设备名是 /dev/eth0,/dev/eth1,/dev/eth2 等等 。下面是一些通用的网络设备名：

/dev/ethN Ethernet 设备

/dev/slN SLIP 设备

/dev/pppN PPP 设备

/dev/lo Loopback 设备

Bus Information

这个信息被设备驱动程序用来控制设备。irq 数据是这个设备使用的中断。base address 是设备的控制和状态寄存器在 I/O 空间的地址。DMA channel 是这个网络设备用的 DMA 通道。所有的上述信息在设备初始化时被设置。

Interface Flags

用来描述网络设备的特性和能力：

IFF_UP	(网络)接口在运行，
IFF_BORADCAST	device 中的广播地址是有效的，
IFF_DEBUG	设备的调试功能已被打开，
IFF_LOOPBACK	当前设备是一个 loopback 设备，
IFF_POINTTOPOINT	这是个点到点的连接(SLP 和 PPP)，
IFF_NOTRAILERS	没有网络跟踪(No network trailers)，
IFF_RUNNING	分配的资源，
IFF_NOARY	不支持 ARP 协议，
IFF_PROMISC	设备处在混杂接收模式，将接收任何网上数据包，
IFF_ALLMULTI	接收所有的 IP 多点广播(multicast)数据帧，
IFF_MULTICAST	能够接收 IP 多点广播(multicast)数据帧。

Protocol Information

通过这些信息，设备描述自己将如何被网络协议层所使用。

mtu 该设备能传输的最大报文大小(不包括所需要的报文头)。这个最大值协议层被用来，例如 IP，选择适当的发送报文的大小。

Family 显示该设备可以支持的协议族。所有 Linux 网络设备支持的协议族是 AF_INET, Internet 地址族。

Type 这个硬件接口类型描述该网络设备正与什么介质相连。在 Linux 网络设备中，可以支持很多种不同的设备，包括 Ethernet, X.25, Token Ring, Slip, PPP 和 Apple Localtalk。

Address

device 数据结构含有许多与该设备相关的地址，例如：IP 地址。

Packet Queue

sk_buff 报文的队列。等待在这个网络设备上传输。

Support Functions

每个设备提供一套标准的例程作为该设备连接层接口一部份。

从而协议层可以进行调用。这些例程包括：设置和帧传输例程；

添加标准报文帧头和收集统计信息的例程。这些统计信息可以通过

ifconfig 命令来查看。

8.6.1 网络设备的初始化

与其他 Linux 设备驱动程序一样，网络设备驱动程序也可以被预先构造在核心中。 每一个潜在的网络设备都对应于一个 device 数据结构。这些结构组成一个由 dev_base

指向的链表。如果网络层需要网络设备完成一个特定的任务，它调用 一个地址已在 device 结构中的网络设备服务例程。在最开始，device 结构中只含有初始化(initialization)和探测(probe)函数的地址。

网络设备驱动程序需要解决两个问题。首先，不是所有的已构建在核心中的网络设备驱动程序都有相应的外设存在。第二，不管什么样的设备驱动程序，系统中的 ethernet 设备名始终是 /dev/eth0, /dev/eth1 等等。网络设备不存在的问题很容易解决。因为当为每个网络设备初始化的例程被调用时，其函数返回值将显示物理设备是否存在。如果不存在，这个驱动程序对应的 device 数据结构将从被 dev_base 指向的链表中被移去。如果驱动程序确实发现一个设备，device 结构的剩余部份将被填充。这些包括设备信息和设备驱动程序中的支撑函数的地址。

第二个问题是关于如何动态地将标准的/dev/ethN 设备特殊文件名赋值给系统中的 ethernet 设备。 在 device 链中，有 8 个标准记录。从 eth0, eth1 到 eth7。它们的初始化都是一样的：依次检测是否每个 ethernet 设备驱动程序有相应的物理设备存在直到发现一个。当找到一个 ethernet 设备时，驱动程序填充其当前占据的 ethN device 数据结构。与此同时，网络设备驱动程序初始化刚刚找到的物理设备，找出该物理设备想要占据的 IRQ 号，DMA 通道等等。一个驱动程序有可能检测到几个它所控制的网络设备 ，在这种情况下，驱动程序将占据几个 /dev/ethN device 数据结构。当所有的 8 个标准的/dev/ethN 都被分配光之后，系统将不检测其他的 ethernet 设备。

第 9 章 文件系统

这章描述 Linux 内核怎么在它支持的文件系统中维护文件，描述虚拟文件系统 (VFS) 讲述了 Linux 内核的真实文件系统是怎么被支持的。

Linux 的最重要的特徵之一是它的为许多不同的文件系统的支持。这使其非常灵活从而与许多另外的操作系统可以很好的共存。 在本书写作的时候， Linux 已支持 15 种文件系统; ext , ext2 ,xia , minix , umsdos ,msdos , vfat , proc , smb , ncp , iso9660 , sysv , hpfs , affs 及 ufs , 并且没有疑问，将来支持的文件类型将被增加的更多。

在 Linux 中 ，因为它是 Unix 的一种,系统可以使用的不同文件系统，不能向 Windows 或 DOS 一样通过设备标识符存取 (例如一个驱动器数字或一个驱动器命名)， 而是它们被构建成为一个单一的层次树状结构以作为代表文件系统的实体。 Linux 通过安装一个文件系统将该新文件系统加入它的文件系统树中。所有的文件系统，不管是什么类型，都安装在文件系统树的一个目录上并且该文件系统之上的文件将掩盖掉这个安装目录中原来存在的内容。这个目录称为安装目录或安装点。当文件系统被卸掉之后，安装目录中原来的文件才再次可见。

当磁盘被初始化时 (使用 fdisk , 例如) 磁盘上存在着一个分区结构把物理的磁盘划分成很多逻辑的分区结构。每个分区可以拥有一个单个的文件系统，例如一 EXT2 文件系统。文件系统通过在物理设备上的目录，软联接等等来组织文件以形成一个逻辑的层次结构。能包含文件系统的设备称为块设备。 IDE 磁盘分区 /dev/hda1 , 系统中的第一个 IDE 磁盘驱动器分区， 是一个块设备。 Linux 文件系统认为这些块设备是块的简单的，线性的组合， 他们不知道(关心)底层的物理磁盘的几何学分布。一个读块设备的请求到具体的物理参数的映射过程由块设备驱动程序来负责，如相应的磁道，扇区和块所在的柱面。 一个文件系统，不管位于什么具体的设备上，必须保持一个同样的方式和接口来进行操作。使用 Linux 的文件系统时， 即使这些不同的文件系统在不同的物理的媒介上，由不同的硬件控制器控制着，对于系统用户而言，应该是透明的，没有关系的。文件系统可能甚至不在本地的磁盘系统上， 而是一个网络安装的磁盘。考虑如下一个 Linux 系统，它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
```

用户和操作在上述文件系统上的程序都不需要知道 /C 是一个安装的 VFAT 文件系统位于系统的第一个 IDE 磁盘上。在上述例子中， /E 是在第二 个 IDE 控制器上的主 IDE 磁盘。第一 IDE 控制器是否是一个 PCI 控制器，第二个控制器是否是一个 ISA 控制器(控制 IDE CDROM 的那个)，在这里没有关系。我能拨号上网进入我工作的机器，使用一个调制解调器和 PPP 协议。在这种情况中我能远程地装我的 Alpha AXP Linux 系统的文件系统在上本地的 /mnt/remote 目录上.(译者注：作者在这里解释了半天，其目的是让读者理解：文件系统(File System)是操作系统中抽象出来的一个概念。其具体的物理结构组织对于用户和用户进程是透明的，不用关心的。)

一个文件系统的文件是数据的集合。一个文件系统不仅含有文件系统中的文件而且含有文件系统的结构。它包含 Linux 用户和进程所能看见的文件，目录联结，文件保护信息等等。 而且它必须安全地保持那个信息，操作系统的基本完整取决于它的文件系统。没人将使用随机丢失数据和文件的一个操作系统。

Minix , Linux 的第一个文件系统有相当的局限性并且缺乏很好的性能。

它的文件名不能比 14 个字符长 (它仍然比 8.3 文件名好一些) 并且最大的文件大小是 64MBytes 。64Mbytes 可能乍看之下似乎足够大但是大文件大小是必要的以用来保持数据库系统。第一个，具体地说，为 Linux 设计的，文件系统, 扩充文件系统, 或 EXT , 在 1992 年 4 月被引入，其解决了很多问题但是仍然缺乏一个很好的性能。

因此，在 1993 ，第二扩大文件系统, 或 EXT2 , 被增加到 Linux 文件系统中。

这个文件系统将在本章被详细描述。

当 EXT 文件系统被增加进 Linux 时，一个重要的关于文件系统的开发技术发生了。真实的文件系统通过一个叫做虚拟文件系统(VFS)的接口层，而从操作系统和系统服务中被逻辑地分离开来。

VFS 允许 Linux 支持许多不同的，文件系统。每一个文件系统提交一个相同的软件接口给 VFS。 Linux 文件系统的所有细节被软件解释从而所有的，不同的，文件系统对 Linux 内核，对在系统运行的程序而言显得相同。 Linux 的虚拟的文件系统层允许你同时透明地安装许多不同的文件系统。

(译者注：细心的读者不难发现，在工业界，通过提供一个接口(Interface)标准,透明地屏蔽掉实现的不同性，多样性是一个常用的方法。如 POSIX 标准，PCI 标准等等。这个思路几乎可以在任何一个技术中得到采用。标准是整合工业界竞争的必然手段和结果。)

Linux 虚拟文件系统的实现要使得对文件的存取要尽可能的快和高效。文件和文件中的数据要正确地被维护。上述这两个要求是互相限制的。 当文件系统被安装和使用时，Linux VFS 在内存中保存其信息。当文件和目录被创造，写和删除时，在这些缓存里的数据要被修改更新，以正确地更新文件系统。如果在运行的核心中观察文件系统的数据结构，你将能看到数据块正被文件系统读和写。描述正在被存取的文件和目录的数据结构，在核心中被创建和删除。设备驱动程序总是在那里存取和保存数据。这些缓存(Cache)中最重要的是是缓冲区缓存(Buffer Cache)， 它是一个文件系统存取底层块设备的方法和途径。当数据块被存取时，它们被放进缓冲区缓存并且根据它们的状态而放入各种各样的队列中。缓冲区缓存不仅缓存数据缓冲区，它也与块设备驱动程序一起管理异步接口。

9.1 第二扩充文件系统 (EXT2)

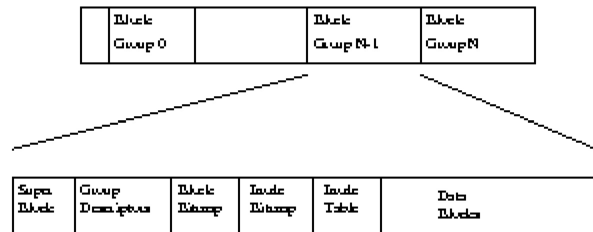


图 9.1 ： EXT2 文件系统的物理的布局

第二扩充文件系统被设计 (由 Remy Card)作为 Linux 的一个可扩展的，强有力的文件系统。它也是到目前为止在 Linux 领域最成功的文件系统并且被当前 Linux 的所有的分发(Distribution)所支持。

EXT2 文件系统，象很多文件系统一样，其构造的前提假设是文件中保持的数据被放在数据块中。这些数据块大小都一样，并且，尽管块大小在不同的 EXT2 文件系统之间可以变化，但当它被创造时(使用mke2fs)，EXT2 中块大小就被定下来 。每个文件的大小都被调整为块大小的整数倍。如果块大小是 1024 个字节，那么 1025个字节的一个文件将占据 2 1024 字节的块。不幸的是，这意味着平均而言每个文件将浪费半个数据块。通常在考虑计算时，存储器和磁盘空间的使用率与CPU的使用效率之间是一种折衷(Trade off)。Linux ，与大多数操作系统一样，选择相对低效的磁盘使用以便在 CPU 上减少负载(workload)。文件系统中，不是所有的块都含有文件数据，其中一些必须被用来描述文件系统的结构信息。 EXT2 通过inode 数据结构描述每个文件。并已此定义文件系统的拓扑。一个inode 描述一个文件中的数据占据哪些块，文件的修正时间，存取权利和文件类型等等。EXT2 文件系统中，每个文件被 一个inode描述并且每个 inode 有一个唯一的数字标识。文件系统的inodes 一起被放在一个 inode 表中。EXT2 目录是一种特殊的文件 (也被inodes 描述)，包含一些指针，指向目录入口的各个文件或子目录的 inodes 。

图9.1 给出了一个在一个块设备上占据一系列块的EXT2文件系统布局。就每个文件系统而言，块设备只是能被读并且写的一系列数据块而已。一个文件系统不需要担心一个数据块放在物理的媒介上的何处。物理分布是设备的设备驱动程序的工作。无论何时一个文件系统需要从包含它的块设备读信息或数据，它请求设备驱动程序读出一个整数倍数的数据块。EXT2文件系统划分其占据的逻辑分区成为数据块组(Block Group)。

除了保持其中的文件和目录的信息之外，每个组还复制那些对于文件系统的完整性至关重要的信息和数据。这个信息的备份是非常需要的当灾难发生并且文件系统需要恢复的时候。下面的章节将更详细的描述每个数据块组的内容。

9.1.1 EXT2 Inode

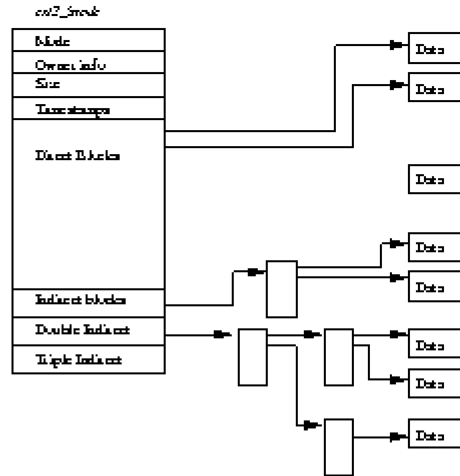


图 9.2 ： EXT2 Inode

在 EXT2 文件系统中， inode 是最基本的积木;文件系统的每个文件和目录被一个并且仅仅被一个 inode 所描述。每个块组的 inodes 被存放在一个 inode 表中。该表与系统中的一张位图一起，使得系统可以追踪分配了的 indoes 和没有分配的 inodes 的情况。图 9.2 显示出一个 EXT2 inode 的格式， 在其包含的信息之中，它包含下列域：

模式(Mode) 这个域含有两个信息;这 inode 描述什么并且用户拥有的允许。对 EXT2 而言 ，一个 inode 能描述文件，目录，符号连接，块设备，字符设备或 FIFO。

拥有者信息(Owner Information)

这个文件或目录的主人的用户和组标识符。这允许文件系统正确允许的该 inode 的各项存取。大小(Size) 以字节为单位的文件的大小。

时间戳(Timestamps)

inode 被创造的时间和它最后一次被修改的时间。

数据块(Datablocks)

到包含这个 inode 描述的数据的块的指针。第一 12 个指针是到包含这 inode 所描述的数据的块的指针。最后 3 个指针包含间接的，越来越多层的，最后描述了数据的，物理块的间接指针。例如， 两倍间接块指针指向一个数据块。该数据块中每个入口又是指向一个数据块指针的指针。这种方式意味着小于或等于 12 个数据块的文件比更大的文件存取起来要更快些。

应该注意的是， EXT2 inodes 可以描述特殊的设备文件。这些不是真实的文件而是程序能够使用来存取设备的句柄。所有在/dev 下的设备文件都在那里以允许程序存取 Linux 的设备。例如 mount 程序将想要安装的设备文件作为一个参数来引用。

9.1.2 EXT2 超级块(Superblock)

Superblock 包含一个文件系统的基本大小和其形状的描述。文件系统管理器使该信息来维持文件系统。当文件系统被安装时，通常仅仅在数据块组 0 的 Superblock 被读进内存。在系统的每个其他块组中也含有一个超级块的副本拷贝以防止文件系统崩溃。在其含有的信息之中：

Magic Number

这允许安装软件根据这个域来检查此确实是一个 EXT2 文件系统的 Superblock 。当前的 EXT2 版本是 0xEF53 .

Revision Level

主和次 Revision Level 使得安装代码可以决定这个文件系统是否只是特别地支持某个版本的文件系统性能。

其特徵相容性域值可以帮助安装代码决定哪些新特徵可以在这个文件系统上被使用，

Mount Count and Maximum Mount Count

这些域在一起，允许系统决定是否文件系统应该被检查。文件系统被安装时，安装数每次被增加，并且当它等于最大的安装数时，系统将显示警告消息"到达最大的安装数目，推荐运行 e2fsck"。

Block Group Number

拥有该 Superblock 的这个拷贝的块组数字，

Block Size

这个文件系统的块的大小，例如 1024 个字节，

Blocks per Group

在一个组中块的数目。当文件系统被创造时，象块大小一样这个值是被固定下来的，

Free Blocks 在当前文件系统中的空余的块的数目，

Free Inodes

在当前文件系统中的空余的 Inodes 的数目。

First Inode

文件系统中的第一个 inode 的 inode 号码。在一个 EXT2 根文件系统的第一个 inode 将是目录入口项 / 目录。

9.1.3 EXT2 组描述符

每个块组(Block Group)有一个数据结构来描述它。象 Superblock 一样，所有块组的组描述符在每个块组中有一份拷贝以防止在文件系统崩溃。

每个组描述符包含下列信息：

块位图(Blocks Bitmap)

当前块组中块的分配位图的块号码。在块分配和回收期间被使用。

Inode 位图

当前块组的 inode 分配位图的块号码。这在 inode 分配和回收期间被使用，

Inode 表

这个块组的 inode 表的开始块的块号码。每个 inode 由一个 EXT2 inode 数据结构来描述。 空余块数, 空余 Inodes 数, 已使用的目录数(Free Block count,Free Inodes count,Used directory count) 组描述符挨个儿存放并且一起组成为描述符表。每个块组，在其 Superblock 的拷贝以后包含组描述符的全部表项。 系统中仅仅第一个拷贝 (块组 0) 实际上被 EXT2 文件系统使用。另外的拷贝, 象 Superblock 的拷贝一样, 只是以防主拷贝崩溃.(译者注：读者可以回忆 DOS 中的两个 FAT 表的使用；当查找一个文件时，系统只用第一个 FAT 表；另外一个 FAT 表作备份使用以防 FAT 链表指针混乱。有趣的是 DOS 中并不保存多个 0 扇区。总的来说，多个超级块和组描述符数据结构的使用是为了保证数据结构的一致性。如当使用 fsck 检查文件系统时，如两个相应的数据结构不一致，那就说明文件系统非正常的操作发生，如调电，非正常关机等等。)

9.1.4 EXT2 目录

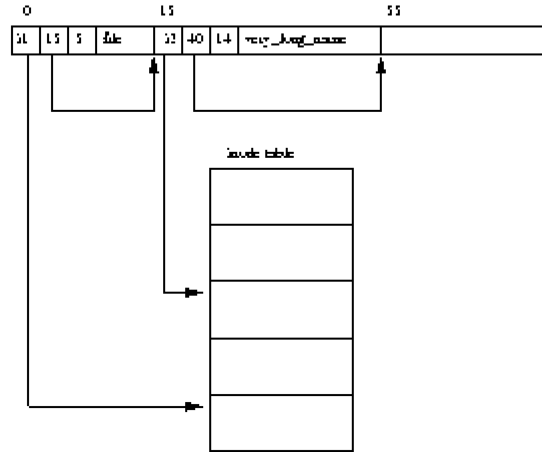


图 9.3 ： EXT2 目录

在 EXT2 文件系统中，目录是被用来创造并且在文件系统中保持存取路径到文件的特殊文件。图 9.3 显示出内存中一个目录入口的布局。

一个目录文件是一系列目录入口的一张表，每一个人口项包含下列信息：

inode 为这个目录入口项的 inode 号码。这是被保存在块组 Inode 表中的 inodes 的数组的索引。

在图 9.3 中，文件 file 的目录入口 是一个指向 inode i1 的指针。

名字长度(name length)

这个目录入口的以字节记的长度，如 16 字节等等。(译者注：换句话说，每个目录项的长度是不定长的)

名字(name)

这个目录入口的名字，如文件的名字或子目录的名字等等。

每个目录的起先两个入口项总是是“ . ”并且“ .. ”，分别意味着这个当前目录和“上一级目录”的入口。

9.1.5 在一个 EXT2 文件系统中寻找一个文件

一个 Linux 文件名的格式和 Unix 一样。它是一系列由“ / “”)分开的目录名组成，最后以文件的名字结束。例如一个文件名是 /home/rusling/.cshrc， 在这里/home 及/rusling

是目录名字。文件的名字是 .cshrc .象所有的其他的 Unix 系统一样，Linux 并不特别着重对文件名的格式本身。它可以是任何长度，由可打印的字符组成。为了发现代表一个文 inode，一个 EXT2 系统必须一个目录一次的逐层分析这个组合的文件名的直到我们最终找到该文件。

我们需要的第一个 inode 是文件系统根(root)的 inode。我们可以得到它的值在文件系统的 superblock 中 。为了读取一个 EXT2 inode ，我们必须在适当的块组的 inode 表从寻找它。如果，例如，根 inode 号码是 42，我们将从块组 0 的 inode 表中读取第 42 个 inode 。根 inode 为一个 EXT2 目录，换句话说 inode 作为一个目录。其指向的数据块包含 EXT2 目录入口的数据。

home 只是"/"中许多目录入口的一个。 从其在"/"中的入口项，我们可以得知描述其的 inode 的号码。我们必须读这个目录 (首先读它的 inode，然后从该 inode 指向的数据块读取 "/home"下的目录入口数据)来发现 rusling。从得到的数据中，我们得到/home/rusling 目录 inode 的号码的入口 。最后我们读入指向描述目录/home/rusling 的 inode 数据。并从其指向的数据块中发现.csshrc 的 inode 数值。从该 inode 中，我们可以定位包含该文件数据的数据块。

9.1.6 在一个 EXT2 文件系统中改变一个文件的大小

文件系统一个普编的问题是文件数据块组织的碎片趋势(译者注：数据块物理存放位置的不连续性，离散性)。保持文件的数据的块在整个文件系统中分布。这使得顺序存取一个文件的数据块的效率随着数据块的分离越来越差。 EXT2 文件系统通过将一个新分配的数据块放在靠近当前块的地方，或至少在一个同样的块组，来克服上述效率的问题。只有当上述行为失败时(译者注：如当前块组已满)，文件系统才分配在另外的块组的数据块。

无论何时进程试图写数据进一个文件， Linux 文件系统检查看数据将写入的位置是否已越过文件的最后分配的数据块。如果是的，它必须为这个文件分配新数据块。直到分配完成，进程不能运行;必须等到文件系统分配一个新数据块并且将余下数据写入到这个新的数据块中之后。EXT2 数据块分配算法要做的第一件事情是锁住 EXT2 文件系统的 Superblock。分配和释放数据块都要改变 superblock 内的域值，文件系统不能允许超过一个的 Linux 进程同时这种变化。如果另外的进程更需要分配数据块,它将必须等待直到这进程完成了。等待 superblock 的进程被挂起,不能继续运行，直到 superblock 的控制被它的当前的占有者所放弃。 superblock 的存取基于先来，先服务的基础(FIFO)并且一旦进程获得 superblock 的控制，它拥有该控制直到它完成了操作。获得并锁住了 superblock 后 ， 进程检查文件系统中是否有足够的自由数据块。

如果没有足够的可分配物理数据块，分配块的尝试将失败并且进程将放弃这个文件系统的 superblock 控制。(译者注：superlock 或 inode 被读进内存后是共享的数据区,所以在存取时要加锁。在操作系统中，文件系统是个非常需要保护的资源。不同的文件句柄可以指向同一个文件。对文件的操作是一个完全并发的操作过程。在一个进程读一个文件的同时，其内容可以被其他进程或同一个进程内部的线程(Thread)所改写。因此操作系统核心的锁机制是非常重要的。译者强烈建议读者阅读相关内部算法。可参见贝齐的著作。)

如果在文件系统中有足够的自由块，进程试着分配一个。

如果 EXT2 文件系统被设计成有预先分配数据块的功能，我们可以从中取一个。预先分配的数据块其实并不实际上存在，它们只是在分配的块位图中被预先保留而已。代表正在试图分配数据块给那个文件的 VFS inode 的新数据块有两个 EXT2 特定的域， prealloc_block 及 prealloc_count ，preallocated 是第一个预先分配的数据块的块号码。preallocated 是当前预先分配的数据块已经有多少。如果当前没有预先分配的块或块 preallocation 功能没被打开，EXT2 文件系统必须从头开始分配一个新块。EXT2 文件系统首先查看在该文件的最后那个数据块之后的数据块是否是空余的。从逻辑上而言，这是分配方案中最有效的块因为它使得做顺序存取更加快捷。如果该块不是空余的，系统扩大搜索范围并且在该理想块的 64 块范围内寻找数据块。这个寻找到的块，尽管不是最理想的，但还是相当靠近并且与另外属于这个文件的其他数据块属于同一个数据块组。

如果甚至上述块也不是空余的，进程开始依次在其他的块组里进行查找直到它发现空余的块。块分配代码在块组中寻找 一个 有 8 个空余的数据块簇。如果它不能发现 8 个数据块在一起，它将要求设置较少些。如果需要或启动了块预分配(preallocation)功能，它将更新 prealloc_block 及 prealloc_count 位值(译者注：系统总是尽力的要把文件的数据块放在相邻的物理位置以提高文件数据查找效率。这里的机关是，如果一个连续 8 个数据块或少点的连续数据块被发现，即使不预先分配占有，在下一次分配空间时，极有可能系统得到最佳的分配方案---连续分配。如果预约功能打开，则确保下次分配的最佳性)
无论哪里系统发现空余的块，块分配代码更新该目标块组的块位图(译者注：标记该物理块已被占用。请回忆在 PC 下使用 NORTON 软件查看磁盘空间使用时的情景)并且在缓冲区缓存(Buffer cache)中分配一个数据缓冲区。那个数据缓冲区被文件系统支持的对应的设备标识符唯一定位(1：1)并且与刚刚分配的物理块号码也是唯一对应的。然后缓冲区的数据被清零-- zero'd，而且缓冲区的状态被标记”dirty" 以表示该缓冲区的内容还没被最后写入对应的物理磁盘块。最后，

superblock 自己被标记作为"dirty"以表示已被改变，然后被解锁。如果有任何进程正在等待 superblock ，在队列中的第一个被允许再次运行并且将为它的文件操作获得 superblock 的独占控制。进程的数据被写到新数据块(译者注：其实是先写入其对应的数据缓冲区中)，并且，如果那个数据块已被充满，进程将重复上述块分配行为从而得到一个新的数据块.（译者注：这里讲的 superblock,indoe, buffer cache 全是核心中的共享数据结构，所以存在与物理磁盘上的映象的一致性问题。在文件系统中，一个非常重要的是：所有的块数据都是先写入 Buffer Cache。而 Buffer Cache 也是被多进程，多线程所共享的。）

9.2 虚拟文件系统 (VFS)

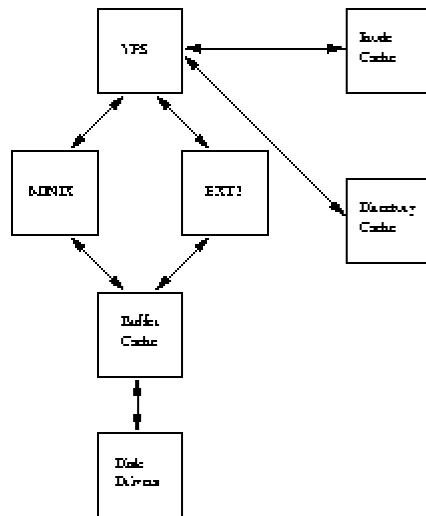


图 9.4 ： 虚拟的文件系统的逻辑图表

图 9.4 显示了 Linux 核心的虚拟文件系统与真实文件系统的关系。虚拟文件系统必须管理在任何时间被安装的，不同的文件系统。为了做到这一点，它在核心中维持描述全部的数据结构为整个(虚拟) 文件系统和真实的, 安装的文件系统。

值得注意的是，VFS，象 EXT2 文件系统一样， 同样使用 superblocks 和 inodes 来描述系统的文件。象 EXT2 inodes 一样，VFS inodes 在系统内用来描述文件和目录;虚拟文件系统的内容和结构拓扑。从现在起, 为了避免混乱, 我们将用 VFS indoes VFS superblocks 以区别 EXT2 inodes 和 superblocks。

当每个文件系统被初始化时，它向 VFS 登记自己。这个过程通常发生在当操作系统在系统引导时间初始化自己的时候。真实的文件系统要么是被嵌入了核心或是作为可装载的模块。系统模块当系统需要它们时被装载，因此，例如，如果 VFAT 文件系统作为一个核心模块被实现，那么只有当被装载(mount)的时候,一个 VFAT 文件系统才被装入核心。当一个基于块设备的文件系统被安装时(这包括根文件系统)，VFS 必须读入它的 superblock 。每种文件系统类型的 superblock 读例程必须了解其相应文件系统的拓扑组织结构并将该信息映射到 VFS superblock 数据结构之上。 VFS 保持系统中所有已安装的文件系统的 VFS superblocks 并组织成一个链表。每个 VFS superblock 包含相应的信息和能执行特殊功能的例程的指针。例如,一个安装了的 EXT2 文件系统的 superblock 包含一个指向读取一个特定的 EXT2 inode 结构的例程指针。这个 EXT2 inode 读取例程，象文件系统的所有其他特定的 inode 读例程一样，在一个 VFS inode 中填写相关域。文件系统中每个 VFS superblock 包含一个指针指向其相应的第一个 VFS inode 。对于根("/")文件系统，这是代表的"/" 目录的 inode。这个信息映射的过程对于 EXT2 文件系统是很有效的但是对于另外其他的文件系统其效率要差些。

当系统的进程存取目录和文件时，与 VFS inodes 处理相关的系统例程在系统核心中被调用。

例如，键入 ls 以显示一个目录或 cat 以显示一个文件导致虚拟文件系统查找代表那个文件系统的相应 VFS inodes。 因为在系统中每个文件和目录都对应于一个 VFS inode，因此会有很多 inodes 将反复的被存取。这些 inodes 被存放在使它们的存取更快的 inode 缓存。如果一 inode 不在 inode 缓存，那么特定的例程必须被请的一个文件系统命令读适当的 inode 缓冲中。如果一个 inode 不在 inode 缓冲中，则必须调用一个特定的例程来读入一个 inode。读 inode 的行为导致一个 inode 被放进 inode 缓存中并且其他相续的对该 inode 的存取将使得该 inode 保持在缓存中。不常用的 VFS inodes 会从核心 inode 缓存中被挪走。

所有的 Linux 文件系统使用相同的的缓冲区缓存(Buffer Cache)机制来缓冲来自底层的数据。这个机制使得文件系统对物理数据存储设备的存取得到加快。

这个缓冲区缓存是独立于文件系统的，被集成入 Linux 核心机制中用来分配和读写缓冲区和。这个机制的最大优点是它使得 Linux 文件系统独立于底层的物理介质，独立于设备驱动程序。所有的块设备在 Linux 核心中登记自己，提供一个一致的，基于块的，异步的接口。即使复杂的 SCSI 设备也如此。当真实的文件系统要从底层物理设备读取数据时，其结果是触发一个块设备驱动程序向它们控制的设备发出读物理块的请求。集成在块设备接口里的就是缓冲区缓存。当文件系

统读入了数据块后，它们被存放在这个全局的缓冲区缓存中，被文件系统和 Linux 核心所共享。在其内的缓冲区数据通过块号码和对应于其设备的标识符被系统唯一标识。因此，如果同样的数据经常被需要使用，数据将从缓冲区缓存被检索而非从磁盘读入。一些设备支持提前读取功能，系统"猜测"要被读取的数据块并事先将其读入到缓冲区缓冲中。

VFS 也保留一个存放目录查找的缓存以便经常被使用的目录的 inodes 能快速被发现。

作为一个试验，试着列出你最近没列出的一个目录。你列出它的第一次，你可以注意响应时间有一点停顿，但是 第二次列此目录时速度却是非常快的。目录缓存并不存储目录的 inodes 本身；这些应该在 inode 缓存中，目录缓存只简单地存储目录名到其相应的 indoe 号码间的映射信息。

9.2.1 VFS Superblock

每个安装了的文件系统都被一个 VFS superblock 所表示;在其信息之中， VFSsuperblock 包含：

设备(Device) 这是这个文件系统所依赖的块设备的设备标识符。例如， /dev/hda1 ，系统中的第一个 IDE 硬盘有一个设备标识符 0x301 ，Inode 指针 mounted inode 指针指向这个文件系统的第一个 inode。 covered inode 指针指向代表这个文件系统安装点目录的 inode。根文件系统的 VFS superblock 没有 covered 指针，

块大小(Blocksize)

这个文件系统的字节的块大小，例如 1024 个字节，

Superblock 操作

一个指向这个文件系统的一套 superblock 例程的一个指针。与其他信息在一起使用，这些例程被 VFS 用来读和写该文件系统的 inodes 和 superblocks 。

文件系统类型(File System Type)

一个指向被安装文件系统中 file_system_type 数据结构的一个指针，

File System specific 一个指针指向这个文件系统所特定需要的一些信息，

9.2.2 VFS Inode

象 EXT2 文件系统一样，VFS 系统中每个文件，目录等等被一个而且仅仅被一个 VFS inode 所表示。

通过一些特殊的文件系统例程，每个 VFS inode 的构建信息都来自于底层的文件系统。 VFS inodes 仅仅在核心中，内存中才存在，并且只当他们对系统有用时才存在。 VFS inodes 包含下列域：

设备(device) 这是保持该 VFS inode 所代表的文件所在的设备的设备标识符

inode 号码

这是该 inode 的号码并且此号码在这个文件系统以内是唯一的。device 和 inode 号码的组合在虚拟文件系统中是唯一的，

模式(mode) 象 EXT2 中这个域一样，它描述这个 VFS inode 代表了什么和相应的存取权利。

用户 ids

该 VFS inode 拥有者的标识符，

时间 创造，修正和写的时间，

块大小

这个文件的块的大小，例如 1024 个字节，

inode 操作

指向一块例程地址的一个指针。这些例程对文件系统是特定的并且它们可以为这个 inode 完成相关操作，例如，截断被这个 inode 所代表的文件。

count

系统中当前使用这个 VFS inode 的统计数字。一个 count 是 0 的 inode 是空余的或可以被从内存中抛弃的。

锁(lock) 这个域被用来锁住一个 VFS inode ，例如，当它正从文件系统中被读取时，

dirty

显示这 VFS inode 是否被写了，如果是，底层相应的文件系统需要修改，以保持一致性，

文件系统特定的信息(file system specific information)

9.2.3　登记文件系统

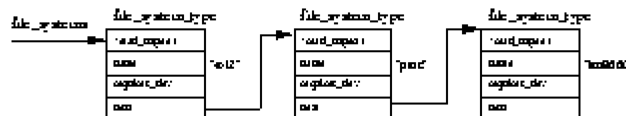


图 9.5 ：登记的文件系统

当你构造 Linux 核心时，你会被问到你是否想要构建支持的每个文件系统。当核心被构造时，文件系统初始代码中含有所有被构造文件系统的初始化代码的调用入口。

Linux 文件系统也可以作为模块来被构造，并且，在这种情况中，它们可以是当需要时或手工安载时(使用 insmod 命令)，才被装入。无论何时一个文件系统模块被装载，它向核心登记自己；当被卸掉时，从核心中撤消登记。每个文件系统的初始化代码在虚拟文件系统 VFS 中登记自己，通过提供 file_system_type 数据结构，在这个数据结构中，含有文件系统的名和一个指向其 VFS superblock 读例程的指针。图 9.5 显示出 file_system_type 数据结构被放进 file_system 的一个链表中。
每个 file_system_type 数据结构包含下列信息：

Superblock 读例程

当一个文件系统的实例被安装时，该例程被 VFS 调用，

文件系统名字

这个文件系统的名字，例如 ext2 ，

设备需要(Device Needed)

这个文件系统需要一台设备支持吗？不是所有的文件系统需要一台设备来支持。例如，/proc 文件系统，不要求一个块设备，

你可以通过查看/proc/filesystems 来获知系统中什么文件系统被登记了。例如：

```
        ext2
 nodev  proc  iso9660
```

9.2.4 安装一个文件系统

当超级用户试图安装一个文件系统时， Linux 核心必须首先验证在系统调用中被传递的参数。尽管 mount 做一些基本的检查，它不知道核心中哪些文件系统是否已经被构建，不知道是否一个安装点实际上存在。考虑下列安装命令：

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

本安装(mount)命令将传递给核心 3 个信息;文件系统的名字, 含有该文件系统的物理块设备和这个新要安装的文件系统将被安装在现有文件系统拓扑结构中的什么地方。

虚拟文件系统必须做的第一事情是找到该文件系统。

为了做到这一点，核心浏览上节讲述的文件系统链表，通过遍历由 file_systems 指向的 file_system_type 数据结构。

如果发现一个匹配的名字，这表明核心当前支持这种文件系统类型并且得到如何读取这个文件系统的 superblock 的例程地址。如果它不能发现一匹配文件系统名字，系统核心会查看是否自己被构建为动态地装载核心模块。(参见 模块章)。在这种情况中核心将请求核心监控程序将相应的文件系统调入。

下一步，如果指定的物理设备还没有被安装, 系统必须发现这个文件系统的安装点目录的 VFS inode。这个 VFS inode 可能已在核心的 inode 缓存中，或它需要从支持安装点的文件系统的块设备被读取进来。一旦 inode 被找到，系统将检查它是否一个目录并且没有其他的文件系统已经被安装在那里了。同一个目录不能为多个文件系统作为安装点。

然后，VFS 安装代码必须分配一个 VFS superblock 数据结构并且将相关的安装信息传递给这个文件系统的 superblock 读例程。
系统的所有 VFS superblocks 结构被放在 super_blocks 向量中。其元素是 super_block 数据结构。 superblock 读例程必须基于它从物理设备读到的信息填写 VFS superblock 记录域。对于一个 EXT2 文件系统，这个映射或信息的翻译的过程是相当容易的, 它简单地读取 EXT2 superblock 并且相应填写 VFS superblock 。对于另外的文件系统，例如 MS DOS 文件系统，事情就不那么简单。无论什么文件系统, 填写 VFS superblock 意味着文件系统必须从支持它的块设备读入一些信息。 如果块设备不能被读或如果它不含有这类文件系统， 安装命令将失败。

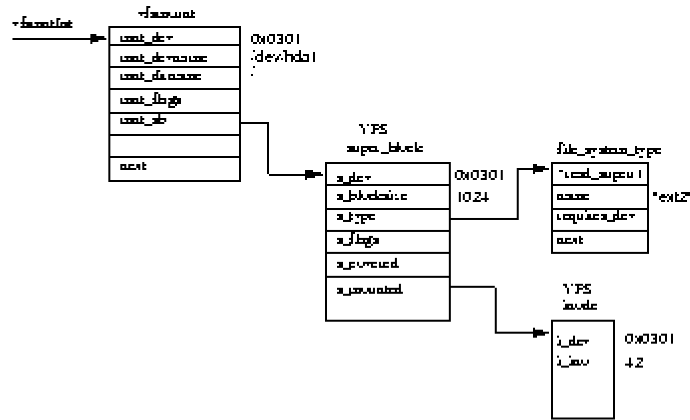


图 9.6 ：一个安装的文件系统

每个被安装的文件系统被一个 vfsmount 数据结构所描述;参见图 9.6，它们被链在一个 vfsmntlist 的链表上。另外一个指针， vfsmnttail 指向上述链表的最后入口。mru_vfsmnt 指针指向最近最多使用了的文件系统。每个 vfsmount 结构包含该文件系统对应的底层设备的设备号，这个文件系统被安装的目录，和一个指针指向其 VFS superblock。如我们已经知道的， VFS superblock 指向这种文件系统的 file_system_type 数据结构，然后指向这个文件系统的根 inode。如果该文件系统没有被卸出核心，这个 root inode 一直呆在 VFS inode 缓冲中。

9.2.5　在虚拟文件系统中查找一个文件

为了在虚拟文件系统中发现一个文件的 VFS inode ，VFS 必须一次一个目录地解释名字，寻找代表名字中间的，那些目录的各个 VFS inode。逐层找到父目录的 inode 是很容易的，因为我们总是可以由其 VFS superblock 得到每个文件系统的根的 VFS inode。每次当真实的文件系统探寻一个目录 inode 时，它首先在目录缓冲中探查这个目录。如果在当前目录缓存中没有入口，真实的文件系统就从底层的文件系统或从 inode 缓存获取其 VFS inode 。

9.2.6　在虚拟文件系统创建一个文件

9.2.7　(卸掉)Unmounting 一个文件系统

如果系统还正在使用一个文件系统的文件，一个文件系统是不能被卸下的。例如，你不能 umount /mnt/cdrom 如果进程正在使用它或它的子目录。如果一个将要被卸下的文件系统正在被使用，那么有可能在 VFS inode 缓存中存在属于这个文件系统的 VFS inodes；系统的代码在核心 inodes 的链表中进行查找这个文件系统占据的设备拥有的 inodes。如果这个安装的文件系统的 VFS superblock 是 dirty 的，这说明它被修改了，那么它必须被写回到在磁盘上的文件系统中。一旦它被写回磁盘，VFS superblock 占据的存储空间就可以被释放。最后，最后对应于该文件系统的 vfsmount 数据结构也被从 vfsmntlist 链表中断开并且释放其占据的空间。

9.2.8 VFS Inode 缓存(cache)

当一个安装的文件系统被浏览时，其 VFS inodes 不断地被读或写。虚拟文件系统维持一个 inode 缓存以加快文件系统的存取。每次一个 VFS inode 从 inode 缓存中被读取，系统就可以节省读取物理设备的存取时间。

VFS inode 缓存的实现是一个其入口是 VFS inodes 链表指针的一张哈希表。同一个链表中的 inodes 拥有相同的哈希值。一个 inode 的哈希值的计算是通过其 inode 的数值和包含其文件系统的物理设备的标识符。无论何时虚拟文件系统存取一个 inode 时，它首先查看 VFS inode 缓存。为了在 VFS inode 缓冲中查找一个 inode，系统首先计算它对应的哈希值然后将其作为索引值进入 inode 哈希表。然后通过读取这个拥有相同哈希值的 inode 链表并挨个儿比较每个 inode 的 inode 数字和一样的设备标识符直到发现为止。

如果一个 inode 在 inode 缓存中被发现，该 inode 的计数(count)值被增加以显示出它还有另外的用户,然后系统接着继续存取文件。否则一个空余的 VFS inode 必须被发现以便文件系统能够从存储器读取 inode 。 至于 VFS 怎么得到一空余的 inode 有很多选择。如果系统可以分配更多的 VFS inodes 空间，事情就解决了;它分配一些核心存储页并且将它们分成一个个新的,空余的 inodes 并且把它们放进核心中的 inode 表。系统中所有的 VFS inodes 都在一个被指针 first_inode 指向的一张链表中。当然也在那个哈希表中。如果系统已经拥有了它可以被允许有的 inodes 的数量，它必须发现一个好的候选 inode 被重用(resue)。好的候选 inode 是一个当前使用计数为 0 的 inodes; 这表示当前系统不再需要这些 inode。那些重要的 VFS inodes , 例如文件系统的根 inodes 的使用计数总是比零大，所以从来不会被选中作为重用候选的 inode。一旦一个候选 inode 被选定，它将被清理。 这个 VFS inode 有可能是 dirty

的，在这种情况中，它需要被写回到文件系统。这个 inode 也可能当前被加锁了，在这种情况中系统必须等待直到它被解锁。VFS inode 必须在重用之前被清理。

当新的 VFS inode 被发现后，一个特定的文件系统例程必须被调用，将从底层真实文件系统读取来的信息来填充这个已准备好了的 inode 数据结构。 当它正在被填写的过程中，这个新的 VFS inode 的使用计数值为 1 并且被加锁，从而其他的实体不能对这个数据结构进行任何操作直到它已含有完整的数据结构。

为了得到一个实际上需要的 VFS inode，文件系统可能需要存取若干个另外的 inodes 。当你读一个目录时，这种情况就会发生;仅仅那个最后的目录的 inode 是我们所需要得，但是那些中间目录的 inodes 也必须被读取。当 VFS inode 缓存机制被使用并且被充满时，那些较少被使用的 inodes 将被丢弃。较多被使用的 inodes 将在缓存中留下。(译者注：细心的读者不难发现，其实在文件系统中存在许多的机制都是为了克服读取物理设备带来的效率代价。译者在这里提出一个问题供大家思考：这些众多的缓冲机制的缺点是什么，特别是随着计算机系统内存价格越来越便宜的时候。这里有很多的工作和思考可以做。)

9.2.9 目录缓存

为了加快对那些被通常使用的目录的存取，VFS 维持目录入口的缓存。

当目录被真实的文件系统查寻时，它们的细节被加进目录缓冲。当下一次同样的目录被查寻时，例如列目录或打开在其中的一个
文件，系统将在目录缓存中找到其信息。仅仅短的目录入口 (15 字节长度) 才被缓冲。这是合理的因为短目录名字是被最经常使用的。例如， /usr/X11R6/bin， 当 X 服务器运行时，该文件通常被频繁地被存取。

目录缓存由一张哈希表组成，其每个入口指向具有同样哈希值的目录缓存的一个链表。哈希函数使用支持该文件系统的设备的设备标识和目录名来作为哈希值的计算□通过哈希表，可以使得一个目录项快速的被找到。一个需要花费很多查找时间的缓冲机制是没有意义的。

为了保持一个最新的，正确的缓冲，VFS 维护一些基于 LRU(Least Recently Used)算法的目录缓冲链表。当一个目录项第一次被放进这个缓存时(它第一次被查找时)， 它被增加到第一层 LRU 链表的链尾。在一个已经充满的缓存区情况下，这将从 LRU 表的前面挤掉一个已经存在的目录入口项。当这个新目录项再次被存取时，它被放到第二层 LRU 缓存表的链尾。这个行为也可

能挤掉一个在第二层 LRU 缓存表中链头的一个目录项。这种在 LRU 链表中的链头元素的替换或挪走是很好的策略。其原因是在链头的元素意味着它们在最近没有被存取。如果他们有被存取，它们的位置将是靠近链表的链尾。在第二层 LRU 中的缓存数据比在第一层的要安全。这里的内涵是在第二层的数据不仅仅是被查寻了一下而已，而是被经常地访问。

9.3 缓冲区缓存(Buffer Cache)

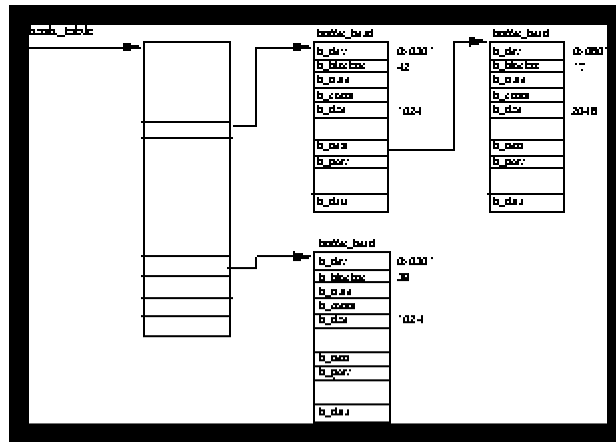


图 9.7 ：缓冲区缓存

当安装的文件系统被使用时，它们产生很多对块设备的数据块的读和写请求。所有的块数据读和写请求以 buffer_head 数据结构的形式传递给设备驱动程序经由一些标准的核例程调用。这个数据结构里含有了块设备驱动程序所需要的所有信息;标识一个设备的设备标识符和要读取得数据块的号码。所有的块设备都是具有同样大小的数据块的线性组合。为了加快物理块设备的存取，Linux 维持一个块缓冲区的缓存。系统中所有的块缓冲区都被放在这个缓冲区缓存的每个地方，甚至包括最新的，还没被使用的缓冲区。这个缓存被系统中所有的物理块设备所共享;在任何一个时间里，在缓存(cache)中都有许多块缓冲区(Block Buffer)，它们可能属于系统中的任何一个块设备而且这些数据处于不同的状态。如果从缓冲区缓存中可以得到有效的数据，这就将节省系统去访问物理设备的时间。任何一个被用来从块设备读取或写数据的块缓冲区都进入这个缓冲区缓存(Buffer Cache)。随着时间的推移，将来它可能从缓存被移走以为那些更合适的缓冲区(Buffer),当然如果它(a Block Buffer)经常被存取，它就可以在缓存里留下。

在缓存内的块缓冲区都通过其对应块设备的标识符合其块号码来唯一标识。缓冲区缓存由两个功能部份组成。第一部份是空余的块缓冲区的链表。对应于每种支持的缓冲区大小，系统中有一个相应的链表。当系统中的块缓冲被创建和被丢弃时，它们就被挂到这些相应的链表上。当前 Linux 系统支持的缓冲区大小是 512 ， 1024 ， 2048 ， 4096 和 8192 字节。第二个功能部份是其缓存(cache)本身。这是一张哈希表。每个入口是指向由指针串起来的缓冲区链表的指针。哈希值索引的计算是有数据块拥有的设备标识符和块号码来产生。图 9.7 所示是这个哈希表和几个入口。块缓冲区要么是在空余的链表之中，或在哈希缓冲区缓存中。当他们在缓冲区缓存中时，它们也被链在 LRU 链中。对每种缓冲区类型都有一张 LRU 链表。它们被系统用来执行对这种类型缓冲相关的操作。例如,写缓冲区中的新数据到磁盘中。缓冲区的类型反映它的状态。Linux 当前支持下列类型：

乾净(clean) 闲置的, 新的缓冲区，
锁(locked) 被锁的缓冲区, 等待被写，
脏(dirty) 脏的缓冲区。这些包含新, 有效的数据，将被写但是到目前为止没被安排写到磁盘上去，
分享(shared) 共享的缓冲区，
独占的（unshared ）
曾经是共享的缓冲区但是目前不是，

无论何时当一个文件系统需要从它底层的物理设备读一个缓冲区(Buffer)时，它试着从缓冲区缓存(Buffer cache)得到。如果它不能从缓冲区缓存得到一个缓冲区，然后它将从适当大小的空余的链表中得到一个乾净的(clean)新的缓冲区。这个缓冲区将被放入将缓冲区缓存。如果它需要的缓冲区在缓冲区缓存中，它可能已含有最新的数据。如果它不是最新的，或如果它仅仅是一个新块缓冲区，文件系统必须请求设备驱动程序从磁盘读取适当的数据块。

象所有的缓存一样，缓存必须被高效地维持以便它有效的，公平地为块设备分配缓存入口。Linux 使用 bdflush 监控程序执行这些 cache 的看护工作。

9.3.1 bdflush 核心监控程序

bdflush 核心监控进程是一个当系统中有太多"dirty"缓冲区时，提供动态相应的一个简单的核心监控进程；含有数据的缓冲区必须在一定的时间内写入磁盘。它在系统启动时间作为一个核心线程而运行，它把自己称为" kflushd "进程。你如果用 ps 命令在系统中显示进程列表，你会看到这个名字。大多数情况下，这个监控进程在系统中睡眠直到系统中 dirty 的缓冲区的数目

变得太大。每次当缓冲区被分配和丢弃时，系统检查 dirty 的缓冲区的数目。如果系统中的缓冲区 dirty 的数目超过了一个百分比，bdflush 将被弄醒。缺省阀值是 60%。 但是，如果系统急需缓冲区， bdflush 将被无条件地唤醒。这个域值可以被观察和修改，通过 update 命令。

```
# 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).
```

系统中所有的 dirty 缓冲区都被链进一个 BUF_DIRTY LRU 链表中一旦它们变得 dirty。bdflush 试着将一定合理数目的这些数据写入磁盘。再次强调的是这个数目是可以通过 update 命令来调节的。其缺省值是 500。

9.3.2 更新(update) 进程

update 命令不仅仅是一个命令;它也是一个监控进程。当作为超级用户运行时 (在系统初始化期间)， 它周期性地刷新所有旧的 dirty 的缓冲区到磁盘上。它通过调用一个与 bdflush 功能差不多的系统服务例程来完成这个任务。

无论何时一个 dirty 的缓冲区出现时，系统给它标识上一个它应该被写入磁盘的系统时间。每次 update 运行时，它查看系统中所有的 dirty 的缓冲区并将已经到期的刷入磁盘。

9.4 /proc 文件系统

/proc 文件系统确实显示了 Linux 虚拟文件系统的强大。它其实并不实际存在。/proc 目录，其子目录和它的文件实际上也不存在。

但你如何能 cat /proc/devices 呢? /proc 文件系统, 就象一个真实的文件系统一样，在虚拟文件系统中登记自己。然而, 当其下的文件或目录被 open 的时候，VFS 对它进行调用请求 inodes 时候， /proc 文件系统利用核心中的信息来创造那些文件和目录。例如, 核心的 /proc/devices 文件是从核心中描述设备的数据结构中产生。

/proc 文件系统提供给一个用户了解核心内部工作的可读窗口。一些 Linux 子系统, 例如 Linux 核心模块，在 /proc 文件系统中都有信息入口项。

9.5 设备特殊文件

Linux , 象所有的 Unix 版本一样，将硬件设备作为特殊文件来对待。例如, /dev/null 是空设备。一个设备文件不占有文件系统的任何数据空间, 仅仅是对设备驱动程序的一个存取点。EXT2 文件系统和 Linux VFS 都使用特殊类型的 inode 来实现设备文件。系统中有两种特殊设备文件类型;字符和块设备文件。在核心自己内部，设备驱动程序实现文件的语义：你能打开他们, 关上他们等等。字符设备允许以字符模式进行所有的 I/O 操作；块设备要求 I/O 操作通过缓冲区缓存(Buffer Cache)的模式。当一个针对设备文件的 I/O 请求到来时，该请求被提交给相应的设备驱动程序。经常这并不是系统的一个真正的设备驱动程序，而是一个针对一些子系统的伪设备驱动程序。例如 SCSI 子系统设备驱动层。设备文件通过一个主设备号(它标明设备类型)，和一个次设备号(主设备类型的一个实例)来标识。例如, 系统的第一个 IDE 控制器上的 IDE 磁盘的主设备号是 3。一个 IDE 磁盘的第一个分区的次设备号是 1。因此， ls - l /dev/hda1 将给出：

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

在核心里，每台设备都唯一的由一个 kdev_t 数据类型来描述, 这是一个 2 个字节的整数, 第一个字节包含次设备号；第二个字节包含主设备号。

上述 IDE 设备在核心中的数据就是 0x0301。一个代表块或字符设备的 EXT2 inode 在它的第一个直接的块指针中保持着该设备的主设备和次设备号。当它被 VFS 读取时，代表它的 VFS inode 数据结构在其 i_rdev 域中设定上述正确的设备标识符信息。

第 10 章 网 络

“网络”与“Linux”在某种意义上是同义词，因为 Linux 是 Internet 与 WWW(World Wide Web)的产物。Linux 的开发者与用户利用 web 来互通信息，交换代码。Linux 本身也经常用于对各种网络应用的支持。本章主要讨论了 Linux 中 TCP/IP 协议的实现。

TCP/IP 协议最初是为了支持在 ARPANET（一个由美国政府资助的带研究性质的网络）中计算机间的通讯而设计的。在 ARPANET 的研究中首次提出了诸如报文交换、协议分层（即一层协议使用另一层的服务）等网络概念。ARPANET 于 1988 年停止使用，不过它的后继系统（NSF1 NET 与 Internet）则得到了更为广泛的发展。众所周知的 WWW（World Wide Web）就是利用 TCP/IP 协议在 ARPANET 上发展起来的。UnixTM 系统在 ARPANET 中得到了广泛的应用。最早发布的 UnixTM 网络版本是 4.3 BSD。Linux 中网络部份的实现是建立在 4.3 BSD 上的，因而 Linux 支持 BSD sockets (包括其扩展)及其全部 TCP/IP 协议。由于在此之前 4.3 BSD 以得到广泛的应用，Linux 选择 BSD sockets 这一编程接口有利于它与其他 UnixTM 平台键应用程序的移植性。

10.1 TCP/IP 概述

本节对 TCP/IP 的主要内容进行了并非详尽的概述。
在一个使用 IP 协议的网络中，每一台计算机都被赋予一个 IP 地址，IP 地址是一个用来唯一标志机器的 32 位整数。比如 WWW 就是一个庞大的、并且在不断增长的 IP 网络，接入该网的所有计算机都有一个唯一的 IP 地址。IP 地址是由四个相互间用黑点隔开的整数表示的，比如，16.42.0.9。事实上每个 IP 地址由两部份组成：网络地址（network address）与主机地址(host address)。这两部份在 32 位 IP 地址中所占位数随地址类型的不同而不同。仍以 16.42.0.9 为例，其中 16.42 是网络地址，0.9 是主机地址。另外，IP 地址还可用子网地址（subnet address）与机器地址(host address)进行分类。在 16.42.0.9 中，子网地址为 16.42.0，机器地址为 16.42.0.9。IP 地址的这种划分为人们进一步划分他们的网络提供了手

段。假设 16.42 是 ACME Computer Company 的网络地址；则 16.42.0 与 16.42.1 就可以分别划分为子网 0 与子网 1 的子网地址。这些子网可以分布在不同的建筑中，相互之间可以由电话线连接，也可以由微波连接。IP 地址由网络管理员分配，建立 IP 子网是一种将网络管理权合理分配给 IP 子网管理员的较好方法。IP 子网管理员可以对本子网内部的 IP 地址自由分配。

一般来说，IP 地址是较难记忆的，而名字则相对较易记忆，比如 linux.acme.com 要比 16.42.0.9 好记得多。这样就要求有一种从名字到 IP 地址的转换机制。与 IP 地址对应的名字可以在文件/etc/hosts 中指定；也可以由 DNS 服务器（Distributed Name Server）来动态解析，在这种情况下，本地计算机必须知道一个或多个 DNS 服务器的 IP 地址，用户可以通过文件/etc/resolv.conf 来配置。

每当你连接到另外一台机器，比如当你在阅读一个网页时，IP 地址就用来与那台机器进行数据交换。交换的数据被封装在若干个 IP 包中，在每个 IP 包的报文头中包含有源机器 IP 地址、目的机器的 IP 地址、校验和以及其他一些有用信息。校验和是根据 IP 报文中的数据计算出来的，IP 包的接收方可以根据它来判断报文在传输过程中是否由于传输线路噪音等原因受到破坏。应用程序的传输数据一般会被分成几个较小的数据包，以简化处理。IP 包的大小根据物理介质的不同而变化；比如以太网的报文包一般要比 PPP 网的大。目的机器在接受到数据包后要将它们重新组装成原来的数据才能交给应用程序。当你通过一个较慢的连接访问一个含有大量图片的网站时，你就能够感觉到数据分片与组装的过程。

同一个 IP 子网上的主机之间可直接传送数据。在其他情况下，IP 数据包将被首先送往一个指定的机器，即网关（或路由器）。网关一般与多个 IP 子网相连，它把从一个子网上接受到的、而目的地址在另一个子网的 IP 包转发出去。比如，如果两个子网 16.42.1.0 与 16.42.0.0 被一个网关连接在一起，那么从子网 0 发送到子网 1 的数据包都将被传送到网关，由网关来决定传输路径。每台本地机都有一个路由表以使 IP 包能够正确的传送。对于每一个目的地址，在路由表中都有一项来指出发往这一地址的数据应送到那一台机器才能使其能到达正确的目的地。另外，路由表是随着应用程序对网络的应用以及网络拓扑关系的变化而动态变化的。

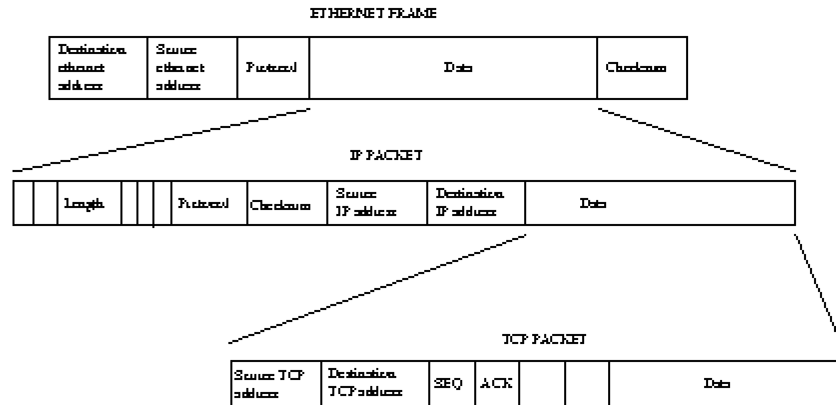


图 10.1 TCP/IP 协议层

IP 协议处于传输层，这一层为其它层的协议的数据传送提供服务。TCP（Transmission Control Protocol)协议是一种可靠的端到端网络协议，它就是使用 IP 的服务来完成自身数据的发送与接收的。与 IP 报文相似，TCP 报文也有自己的报文头。TCP 是一个面向连接的协议，即网络上两台计算机在进行数据传输时，虽然在它们之间可能存在许多子网、网关和路由器，但 TCP 进行连接时，两台机器上的应用程序之间存在一条虚连接。通过 TCP 进行的应用数据的传送与接收都是可靠的，即不会丢失数据或数据重复。当 TCP 利用 IP 进行数据传输时，IP 包中数据部份就是整个 TCP 包，IP 层协议对自身协议包的传输负责。UDP(User Datagram Protocol)协议是另一种利用 IP 协议进行传输的传输层协议，与 TCP 所不同的是，UDP 协议的传输不是可靠的，而仅仅提供数据报（DATAGRAM）服务。这种对 IP 协议的使用使得 IP 协议在接收到数据后必须有能力判断所接收到的数据应该传给哪一个上层协议。为了实现这一点，在每一个 IP 包的报文头中都有一个字节作为协议标志用于指出该包的数据部份是那种协议的报文。比如当 TCP 请求 IP 层提供服务时，在相应的 IP 包的报文头中协议标志就指出该包中包含一个 TCP 包。接收方的 IP 层就利用这一标志来决定应该把数据上传给那种协议（在本例中，应该交给 TCP 层）。应用程序在利用 TCP/IP 进行网络传输时，不仅要指出 IP 地址，还要指出对方应用程序的端口号，端口号与应用程序一一对应，标准的网络服务使用标准的端口号，比如 web 服务器使用 80 号端口。端口地址的使用情况可以在文件/etc/services 中看到。

协议的层次结构并非仅限于 TCP,UDP 与 IP，IP 协议层本身就使用许多不同的物理介质来传输 IP 报文，这些介质进行传送时也有自己的协议与协议头。以太网层就是一个较好的例子，其他类似的例子还有 PPP 与 SLIP 等协议。以太网允许多台主机同时连接到一个物理电缆上，在其上传输的所有以太帧对连接在该网上的所有机器都是可见的，因此每一台机器都应有唯一的物理地址，从而以太帧只能被具有其目的地址的机器所接收，而其他机器将忽略之。物理地址是内嵌在以太网卡中

并且在网卡制造时确定，通常这一地址是保存在网卡上的一个 SROM2 中。以太网的物理地址有 6 个字节，比如 08-00-2b-00-49-A4。某些特殊的以太网物理地址被保留用作广播，即以这些地址作为目的地址的以太网物理帧将为该网上所有的主机所接收。以太帧可以通过在数据帧的帧头上包含一个标志的方式来传送多种不同协议（数据），这就使得以太网能够正确地接收 IP 数据包并将其上传给 IP 协议层。

为了能够通过象以太网这样的支持多连接的物理网传送 IP 报文，IP 层协议必须能获得对方主机的以太网物理地址。因为 IP 地址仅仅是一种概念上的地址体系，以太网物理设备自身有它们自己的地址体系。另一方面，IP 地址并不固定，可以为网络管理员随意设置与修改，而以太网卡则只接收与自身地址相对应的以太帧或广播帧。Linux 利用 ARP（Address Resolution Protocol）来进行从 IP 地址向诸如以太网的物理地址等物理地址的转换工作。当一个主机想知道与一个 IP 地址相对应的物理地址时，该主机就通过广播的方式向网上的所有机器发送 ARP 请求报文，该报文中包含有希望被解析的 IP 地址。当使用此 IP 地址的机器接收到这一请求报文时，就用 ARP 回答报文回应，在该报文中包含其物理地址。ARP 协议并不仅仅局限于以太网设备，它还可为其它诸如 FDDI 等物理介质解析 IP 地址。对于那些不能使用 ARP 进行解析的物理介质在 Linux 中都被作标记，从而系统运行时将不会使用 ARP。另外还有一种协议 RARP(ReverseARP) 用于反向解析，即将物理地址解析为 IP 地址。这种协议一般用于网关，它对包含远地网的 IP 地址地址 ARP 请求进行回应。

10.2 Linux 中的 TCP/IP 网络层次结构

和网络协议一样，Linux 在实现互连网地址协议簇时也将其实现为一系列相互依赖的以层次结构组织的软件。BSD 套接字由一个基本的套接字管理软件支持。而为其提供支持的下层软件为 INET 套接字层，这一层软件统一管理利用 TCP 与 UDP 进行的端与端之间的通信。正如上文所述 UDP (User Datagram Protocol)是一个面向非连接的通信协议，而 TCP (Transmission Control Protocol)则是一个面向连接的可靠的端与端通信协议。当使用 UDP 进行数据的传输时，Linux 不知道也不关心数据包是否正确地到达了目的端；而 TCP 的数据包则被编号，从而使通信对方能够确认数据被正确地接收。IP 层也包含 Internet Protocol 的实现代码，这些代码将 IP 报文头加到被传输数据之前并且负责将接收到的 IP 报文上传给 TCP 或 UDP 层。在 IP 层之下的是网络设备层，比如 PPP 与 ethernet，由它们负责 Linux 的物理连接。网络设备并不总是指物理设备，也有可能指软件设备比如回溯设备。与使用 mknod 命令所生成的 Linux 设备所不同的是，网络设备只有当低层软件存在并且由这些软件完成了设备的初始化工作之后才能生成。在利用合适的以太网设备构造了一个内核之后，你就会看到 /dev/eth0。ARP 协议处于 IP 层与为 ARP 提供寻址支持的协议之间。

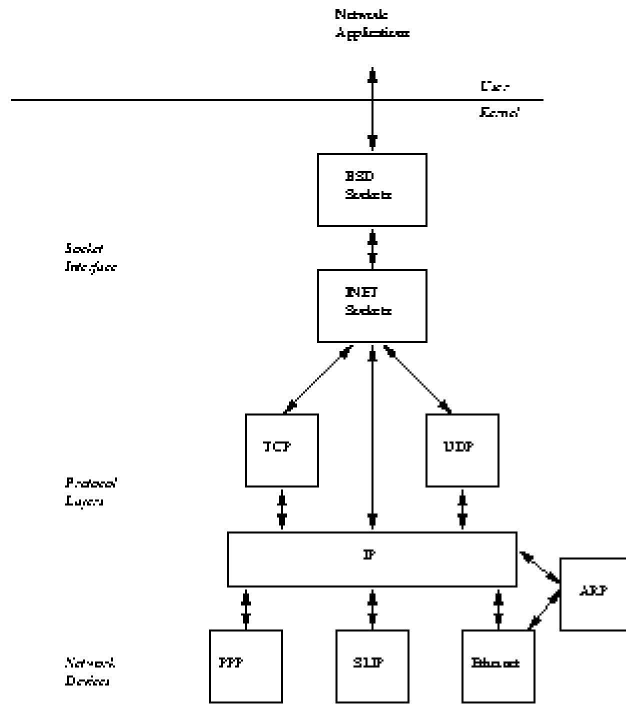


图 10.2 Linux 中的网络层次

10.3 BSD 套接字

BSD 套接字是一个通用的系统接口，它不仅支持各种形式的网络连接，同时还是一种进程间的通信机制。套接字可以描述通信连接一端的运行状态，对于参与通信的两个不同进程有不同的套接字与之对应。事实上，套接字可以被看作为一种管道，只不过与管道不同的是，它对于其中所能容纳的数据量没有限制。Linux 支持多种不同类型的套接字，由于不同类型的套接字在通信时进行寻址的方法不同，套接字又被称为地址族。Linux 支持以下几种套接字地址族或套接字域：

UNIX　Unix 域的套接字

INET　该域的套接字提供对通过 TCP/IP 的通信

AX25　适用于 Amateur radio X25 协议

IPX　适用于 Novell IPX 协议

APPLETALK　适用于 Appletalk DDP 协议

X25　适用于 X25 协议

套接字系统中存在几种不同的套接字类型，不同的类型代表了不同的网络服务，但并不是每一种地址族都支持所有类型的服务。LinuxBSD 套接字系统支持以下几种套接字类型：

Stream
这种类型的套接字提供可靠的、双向、有序的数据流传输，这种通信可以保证数据不被丢失、破坏或重复。在 Internet(INET)地址族中，Stream 类型的套接字由 TCP 协议支持。

Datagram
这种类型的套接字也提供双向的数据传输，只不过与 stream 类型的套接字不同的是，它不保证数据能够达目的地，即使数据抵达目的地了，也不能保证数据的正确性,即它不能保证数据不被破坏或重复。在 Internet 地址族中，这种类型的套接字由 UDP 协议支持。

Raw
该类型的套接字允许进程直接对下层协议进行操作，这也是为何称其为 raw 的原因。比如，可以使用该类型的套接字直接对以太网设备层进行操作从而的到原始的 IP 数据包。

Reliable Delivered Messages
该类型与 datagram 基本相同，唯一区别在于它保证数据能够到达目的地。

Sequenced Packets
该类型与 stream 类型相似，唯一区别在于该类型的数据包大小是固定的。

Packet　这种类型不是标准的 BSD 套接字类型,它是 Linux 自身对套接字系统的扩展，它允许进程直接对设备层的数据包进行操作。

利用套接字进行通信的进程使用客户机/服务器模式，即服务器提供某些服务，客户机则使用这些服务。比如，一个 Web 服务器提供一些 web 网页，web 客户端程序比如浏览器等就可以浏览这些网页。如果一个服务端程序要通过套接字提供服务，那它首先要创建一个套接字然后再为其绑定一个名字，这种名字实际上指明了服务程序在本地系统中的地址，名字的格式与套接字所属的地址族相关，套接字的名字或地址用 sockaddr 这一数据结构来指定。对于 INET 域的套接字还要绑定一个 IP 端口地址（端口号），端口号的使用情况可以在文件/etc/services 中看到；比如，web 服务器所使用的端口号为 80。在套接字绑定地址以后，服务程序就在该地址上侦听针对这一地址的服务请求。客户方程序提出服务请求时，也要创建一个套接字然后指定目的地址并建立一个与对方套接字的连接。对于 INET 域的套接字目的地址就是服务器的 IP 地址与其端口号。客户方的服务请求到达服务器后要通过各层协议一直到达服务程序的侦听套接字，服务程序在发现服

务请求后只有两种选择：接受与拒绝。如果一个服务请求被接受，服务程序就会建立一个新的套接字而不是使用侦听套接字来继续完成相应的服务操作，因为用于侦听的套接字是不能用于支持连接的。在正确地建立了连接后，连接的双方就可以方便地进行数据的传送与接收，当此连接不再需要时，可以被关闭。在整个过程中，都有相应的措施保证数据能够被正确的处理。

在 BSD 套接字上操作的具体含义取决于相应的地址族。比如建立一个 TCP/IP 连接和建立一个 amateur radio X.25 连接是不同的。类似于虚拟文件系统，Linux 将 BSDsocket 层中与应用程序关系密切的那一部份抽取出来成为 socket 界面，并有专门的软件支持。在内核启动时，各地址协议族连同其 BSD socket 界面一起在内核中登记注册，这样，在应用程序使用 BSD socket 时，就可以方便地在 BSDsocket 与相应支持软件之间建立联系。这种联系是通过交叉连接的数据结构以及其它一些信息表建立的。比如在创建新的 socket 时，就有相应的创建例程进行支持。

内核在配置时，各层协议的对应号码被填入协议向量中，协议向量中的每一项都由相应的名来表示，比如，"INET"与其初始化例程的地址。在系统引导时，套接字接口被初始化，与此同时，各协议的初始化例程也将被调用。对于该套接字协议而言，这一步骤将会产生一组协议操作，即一组例程，其中每一个例程都将执行针对该地址族的一些特定操作。所有这些协议操作被保存在一个指向 proto_ops 结构的指针向量 pops 中。

proto_ops 结构包含地址族的类型以及一组指向针对该地址族的套接字操作例程指针。

Pops 向量可以由地址族标识符来检索，比如，Internet 地址族对应于 AF_INET（在 Linux 中被赋值 2）。

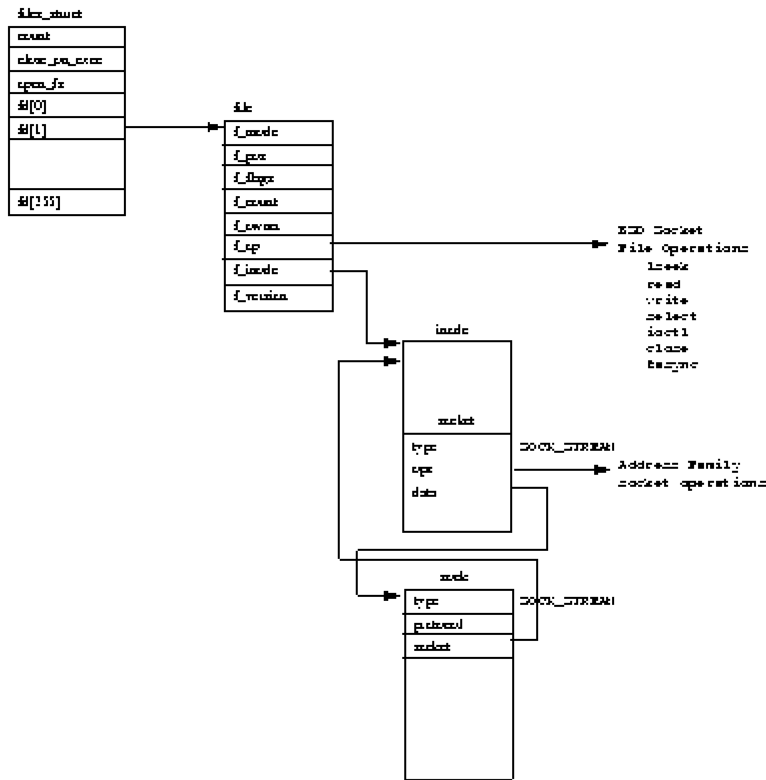


图 10.3　Linux 中 BSD Socket 的数据结构

10.4　INET 套接字系统

INET 套接字系统支持包括 TCP/IP 协议在内整个 internet 地址协议族。由上文讨论可知，整个协议族是层次化的，下层的协议为上层的协议提供服务，这种层次化结构也体现在 Linux 的 TCP/IP 源代码所使用的数据结构中。各层协议与 BSD 套接字层的操作界面是一组 Internet 域的 socket 操作，这些操作一般在网络初始化时在 BSD 套接字层中登记，这些登记信息连同其他协议族的信息一同保存在 pops 向量中。BSD 套接字系统通过 proto_ops 结构调用 INET 套接字层所支持的例程来完成相应的工作，比如，BSD 套接字使用 INET 域中的地址提出请求时就要使用到 INET 层的例程。在该操作中，BSD 套接字系统把代表 BSD 套接字的数据结构交给 INET 层，INET 层并不将 BSD socket 直接与下层的 TCP/IP 协议的信息连接在一起，而是使用自身的数据结构 sock 与 BSD socket 进行联系，如图 10.3。BSD socket 使用自身结构中的一个指针与 sock 结构连接起来，这样就使以后的 socket 操作能够比较方便的访问 sock 结构中的数据。在 sock 结构中有一个协议操作指针，它指向的内容随被请求服务的协议的不同而不同。例如，如

果请求 TCP 服务,则 sock 结构中协议指针将指向一组 TCP 的协议操作组，以便于进行 TCP 连接时使用。

10.4.1 BSD Socket 的建立

使用系统调用建立一个新的 socket 时要用到地址族、socket 类型与协议标志三个参数。

地址族标识用于在 pops 中搜索对应的地址族信息。有时一个特定的地址族由一个特定的内核模块来实现，在这种情况下 kerneld daemon 就必须将这一模块加载程序才能继续执行。BSD socket 是由一个 socket 数据结构来表示的。在实现上，socket 结构是一种 VFS inode 结构，生成一个新的 socket 的结构也就等于申请一个 VFS inode，之所以这样是因为 socket 可以象文件句柄那样用于对普通文件的操作，但文件都是用 VFS inode 来表示的，因此为了实现对文件操作的支持，BSD sockets 也必须用 VFS inode 结构来实现。

在新创建的 BSD socket 数据结构中包含一个特殊的指针，该指针指向面向地址族的 socket 例程，这些例程的具体信息被保存在 proto_ops 结构中，并且可以根据 opos 结构中的信息进行检索，其中 socket 类型可以是 SOCK_STREAM, SOCK_DGRAM 等。那些面向地址域的创建例程可以通过保存在 proto_ops 结构中的地址来调用。

一个新的文件描述符是从当前进程的文件描述符向量中分配的，同时该描述符所指向的文件数据结构也被初始化。这一过程包括对文件操作指针的设置，使之指向一组由 BSD socket 界面所支持的 BSD socket 文件操作例程，任何进一步的操作都将传送给 socket 界面，而后者则会调用其相应的地址族操作例程从而将操作请求传给下层提供支持的地址协议。

10.4.2 INET BSD Socket 与地址的绑定

为了能够侦听 internet 上的连接请求，服务器创建一个 INET BSD socket 并要给其绑定一个地址，该绑定操作一般是在 TCP 与 UDP 协议的支持下，由 INET socket 层来完成。一个 socket 一旦被绑定了地址，就不能用于其它通信，即该 socket 的状态将是 TCP_CLOSE。在进行绑定操作时所使用的参数中 sockaddr 应该包含被绑定的 IP 地址，并最好同时提供端口号。一般情况下，用于绑定的 IP 地址应有相应网络设备提供支持，它应支持 INET 地址族并且其相应的界面可以提供服务。用户可以用 ifconfig 命令来查看哪一个网络界面正在被使用。上面的 IP 地址还有可能是全 1 或全 0 的广播地址，意为要将数据传给所有的机器。如果一个机器充当代理或防火墙则它的 IP 地址可以被指定为任何 IP，但只有拥有超级用户权力的进程才能绑定任何一个 IP 地址。被绑定的 IP 地址被保存在 sock 结构中的 recv_addr 与 saddr 域中，分别用于哈希表的搜索与发送数据，其中端口号是可选的，如果用户没有指定，则系统会自动申请一个空余的端

口，按照惯例，小于 1024 的端口号在没有得到管理员的允许的情况下不应使用，网络系统自己分配端口号时，一般也只分配大于 1024 的端口。

下层网络设备必须将接收到的数据包交给正确的 INET 与 BSD 套接字才能进行处理。为此，UDP 与 TCP 利用一些哈希表进行输入 IP 数据的信息搜索从而将数据包正确地交给 socket/sock 数据。由于 TCP 是一个面向连接的的协议，因而处理 TCP 包所需的信息要比处理 UDP 包多。

UDP 中有一个哈希表 udp_hash，它保存了所有已分配的 UDP 端口。事实上，该表保存的是指向 sock 数据结构的指针，可以利用端口号进行检索。由于 UDP 哈希表的表项要比可能的端口号数量少的多（udp_hash 只有 UDP_HTABLE_SIZE=128 个表项），因而该表的某些表项是一个 sock 数据组成的链表。

与 UDP 相比，TCP 协议相对就比较复杂，它要维护多个哈希表。但在绑定操作时，已被绑定的 sock 数据并不立即加入 TCP 的哈希表中，只是检查一下所使用的的端口号是否已被使用。Sock 数据是在执行侦听操作时插入哈希表中的。

10.4.3 利用 INET BSD Socket 创建连接

一个 socket 被创建之后，如果还没有用于侦听连接请求，该 socket 可用来发出连接请求。在进行连接时，面向连接的 TCP 协议需要在双方之间建立一条虚拟链路，而对面向非连接的 UDP 协议而言则不需作这些工作。

连接请求的发出只能在某些处于特定状态的 INET BSD
socket 上进行，也就是说，只有那些还没有建立连接并且也没有用于侦听连接请求的 socket 上才能发出连接请求。在系统中，BSD socket 应处于 SS_UNCONNECTED 状态。UDP 协议不需建立虚连接，而直接以数据报的形式进行通信，同时发出的数据不能保证能够到达目的地。但该协议支持对 BSDsocket 上的连接，只不过在建立连接时，不建立虚连接而只设置远地程序的 IP 地址与 IP 端口，同时设置路由缓存，以使以后在该 socket 上传送的 UDP 数据报不再需要使用路由数据库（除非已有的路由不再有效）。
路由缓存由 INET sock 结构中的 ip_route_cache 指向，在没有提供地址信息时，路由缓存中的信息（路由与 IP 地址）将被自动地使用。同时，该 socket 的状态也将被设为 TCP_ESTABLISHED。

在 TCP BSD socket 上进行连接时，TCP 协议将向对方发送一个包含连接信息的 TCP 数据包。该数据包中有消息的初始序列号、可接受数据的大小、发送与接收窗口大小等连接信息。在 TCP 中，所有的数据都是被编号的，初始序列号就是第一个消息的编号。Linux 一般选择一个随机的编号作为初始序列号以防恶意的破坏。通过 TCP 连接发送的所有数据在对方成功接收后都会得到确认，没被确认的数据将被重发。发送与接收窗口的大小是指已发送但尚未被确认的数据包的最大数量。网络传输数据包的大小取决于下层网络设备，当通信双方的大小不一时，取较小者。发出 TCP 连接请求的一方必须等待对方的回应（接受或拒绝），此时它必须将自身加入 tcp_listening_hash 表中，从而传入的数据能够交给它；与此同时，TCP 还启动若干时钟以便当对方没有回答时进行超时处理。

10.4.4 在 INET BSD Socket 上的侦听

一旦一个 socket 被绑定了地址后，它就可以侦听针对于它的连接请求。有时 socket 在没有绑定时也可以侦听连接请求，在这种情况下，INET socket 层会为该 socket 自动地绑定一个尚未使用的端口号。在此之后 socket 层的侦听函数 listen 会将该 socket 置为 TCP_LISTEN 并执行其它一些操作从而可以处理连接请求。

对 UDP 协议而言，以上的操作已经足够了，但 TCP 还要将该 socket 中的 sock 数据加入两张哈希表：tcp_bound_data 与 tcp_listening_hash，这两张表都是由一个基于 IP 端口号的哈希函数来进行检索的。

对于一个正在侦听的 socket，当它接收到一个连接请求时，TCP 协议就为其创建一个新的 sock 数据来处理与该连接相关的通信操作。在 sock 结构中，还包含用于保存连接请求信息的 sk_buff 结构的复制体，该 sk_buff 结构被放入用于侦听的 sock 结构中的一个队列中：receive_queue，sk_buff 中有一个指针指向新创建的 sock 结构。

10.4.5 连接请求的接受

UDP 协议不支持连接，TCP 协议中才涉及 INET socket 上连接请求的接受。一个处于侦听状态的 socket 执行接受操作时会在该侦听 socket 的基础上克隆（clone）一个新的 socket，然后就将接受操作交给下层支持协议（如 INET）来处理。如果该协议的支撑协议不支持连接，如 UDP，连接接受操作就会失败，否则接受操作将被进一步交给完成实质性操作的协议如 TCP。连接的接受操作可以分为阻塞方式与非阻塞方式两种。对于阻塞方式而言，执行接受操作的应用进程会在一个等待队列中等待直到接收到一个 TCP 连接请求为止。对非阻塞方式而言，应用进程不进行等待，如果没有连接请求该操作就失败返回，新生成的 socket 数据也将废弃。在一个连接请求被正确地接收后，用于保存请求信息的 sk_buff 结构就被废弃，同时将一个 sock 结构返回给

INET 层并被放入新生成的 socket 结构中。网络应用程序此时会得到一个新 socket 的文件描述符，由该描述符就可继续进行网络应用的后续操作。

10.5 IP 层

10.5.1 Socket 缓冲

在层次化的网络协议结构中，一层协议将为另一层提供服务，因此每一协议在接收与发送数据时，都要对数据的协议头与协议尾进行处理，其中包括确定协议头与协议尾的位置，这样就使在各层协议间数据的传递很困难。一种解决方法就是在各层之间拷贝数据，这种方法一般效率不高，在 Linux 中使用另一种方法，即 socket 缓存方法。Linux 使用 sk_buffs 结构来完成协议层之间的数据传送，该结构包含一些指针与长度域以使个协议能利用标准的函数或方法来处理应用数据。

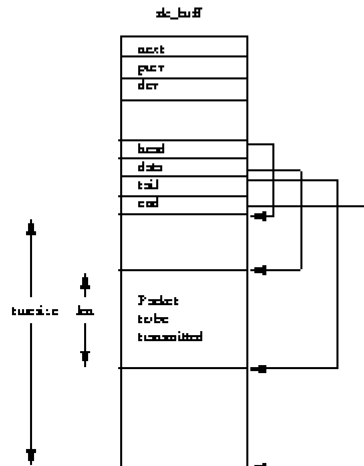


图 10.4 socket 缓冲区(sk_buff)

Sk_buff 的数据结构如图 10.4 所示，sk_buff 使用以下四个指针对其自身的数据块进行操作：

Head

指向整个数据区的开始位置，当 sk_buff 结构及其相关的数据块被分配后，该指针的值就确定了。

data

指向当前协议数据区的开始位置，该指针的值随 sk_buff 所处协议层的不同而改变。

tail
指向当前协议数据区的结束位置，与 data 类似，该指针的值也随 sk_buff 所处协议层
的不同而改变。
end
指向整个数据区域的结束位置，与 head 类似，该指针的值也在 sk_buff 结构分配时确
定。

在 sk_buff 中有两个长度域：len 与 truesize，分别描述协议数据与整个数据缓存的大小。系统对 sk_buff 进行处理的代码提供了对应用数据增删协议头与协议尾的操作，这些操作能正确地对 sk_buff 中的头指针、尾指针以及长度域进行正确的修改。
push
该操作将一块数据加到数据区的头部，同时修改 len 的值。该操作用于将协议头等数据加到传输数据上。
pull
该操作修改 data 指针，使之与数据区的尾不接近，同时减少 len 的值。在删除接收数据中的协议头时用到这一操作。
put
该操作将 tail 指针向数据区的尾移动，同时增加 len 的值。该操作用于将协议信息加到数据的尾部。
Trim
该操作将 tail 指针向数据区的头移动，同时减少 len 的值。该操作用于协议信息从数
据的尾部删除。
sk_buff 结构中还包含用于将多个 sk_buff 链接成双向链表的辅助指针，同时系统提供向该链表增删 sk_buff 结构的例程。

10.5.2　IP 包的接收
Chapter 设备驱动程序描述了 Linux 中的网络设备是如何加载到内核并初始化的，此过程的结果是生成 dev_base 链表，该链表的表项描述了设备信息。每一个表项都包含了一组回调函数，当上层协议需要网络设备提供服务时就调用这些函数，它们一般用于数据的传输以及对网络设备地址的操作。当网卡接收到数据后就将其转换成 sk_buff 结构，然后这些 sk_buff 将被放在 backlog 队列中。

backlog 队列的大小有一定的限制，超过该限制后收到的 sk_buff 将被丢弃，同时，网络系统的也被标识为运行状态。

底层网络协议运行时，先处理发送出去的数据包，然后处理 sk_buff 的 backlog 队列，将数据传给对应的协议进行处理。

Linux 的网络层协议被启动并初始化后，每一协议都将自身的 packet_type 数据加到 ptype_all 链表或 ptype_base 哈希表中，从而在系统中登记。Packet_type 结构中包含了协议类型、指向网络设备的指针、指向进行接收处理的协议例程的指针以及指向链表或哈希表中下一个 packet_type 项的指针。ptype_all 链用来探测那些已接收到，但还没有被正常使用的数据包。Ptype_base 表利用协议标志符对接收到的数据进行哈希排列，同时决定由哪一个上层协议来处理接收到的网络数据。在接收数据时，系统将 sk_buff 数据与两张表中的 packet_type 表项进行匹配。某些情况下，比如对网络数据进行侦听时，会有多个表项与之匹配，这时 sk_buff 将被复制为多份。在此之后，sk_buff 数据将被传给匹配的协议例程进行处理。

10.5.3 IP 包的发送

当程序需要交换数据或网络协议要建立连接时，就要发送数据包。但不管因为何种原因发送数据包，都要分配一个 sk_buff 结构来存储接收到的数据，当该 sk_buff 结构在各协议之间传递时，还会被加上各种协议头信息。

只有把 sk_buff 交给网络设备层后数据才能被传递，为了实现这一点，首先要由某一协议（如 IP）来决定数据应由哪种设备传送，这一判断一般取决于哪一条路径最好。对于那些通过 modem 上网的系统来说，它们使用 PPP 协议，路由的选择由于网络的结构的简单性而变的容易，数据要么通过 loopback 设备交给本地机，要么交给在 modem 另一端的网关。对于以太网上的计算机而言，路由的选择则比较困难，因为在同一个网上连接有许多计算机。在发送 IP 数据包时，IP 协议使用路由表来解析目的地的 IP 地址，解析出的 IP 地址会以 rtable 结构的形式返回，该结构包含了相应的 IP 地址和网卡的物理地址，有时还会包含一个硬件信息头。硬件信息头一般都是针对特定网卡的，其中包含了源机与目的机的物理地址以及其它一些与传输媒体有关的信息。对于以太网的情况，硬件信息头就是图 10.1 中所示的那样，其源与目的地址就是相应以太网地址。由于在同一条路由上发送 IP 包所用的硬件信息头都是相同的，为了减少重复构造的开销，每一信息头都是与对应的路由一同保存在内存缓冲里的。有时，硬件信息头中的物理地址必须通过 ARP 解析之后才能得到，在这种情况下，报文就要等到地址被解析之后才能发送。一旦物理地址被成功的解析并创建了相应的硬件信息头，该信息就被保存起来以便以后使用同一条路由发送 IP 包时就不用重复 ARP 的解析了。

10.5.4 数据分片

对每一个网络而言，网卡所能发送数据桢的大小是有限制的，为了适应这种限制，IP 协议能够将数据包分割成若干较小的报文，在 IP 协议头中也有一个分片域，指出数据分片的偏移量。

在 IP 包准备发送时，IP 协议就在 IP 路由表中找到与其对应的网络设备。每一个网络设备都有一个 mtu 数据描述其所能发送的数据桢的最大值，如果 mtu 小于 IP 包的大小，IP 协议就必须将自身的 IP 包分割成 mtu 大小的分片。每一分片仍用 sk_buff 来保存，只不过其 IP 协议头将标记它为一个分片，同时指出该分片在原来数据中的偏移量，最后一个分片也要有标记来指明那是最后一个分片。但如果在分片过程中，IP 无法分配新的 sk_buff，则传输失败。

对 IP 分片的接收要比发送困难，因为 IP 分片可以以任何顺序到达，而接收方只有在所有的分片都到达后才能进行组装。因而接收方每收到一个 IP 包都要检查它是否一个 IP 包的分片。当收到第一个数据分片时，IP 协议就建立一个新的 ipq 数据结构，同时该数据将被链入 IP 分片重组队列 ipqueue 中，随着更多的 IP 分片的到达，就能识别真正的 ipq 结构，同时生成新的 ipfrag 结构对分片进行描述。ipq 结构描述了 IP 包的分片信息，包括源与目的地的 IP 地址，上层协议标志符以及本 IP 包的标志。当所有的分片都收到后，IP 就将它们组装成一个 sk_buff，同时上传给相应协议继续处理。在 ipq 中还设有一个时钟，该时钟在每一个 IP 包到达时启动，如果时钟超时，该 ipq 与 ipfrag 数据就将被废弃，系统认为相应数据在传送中丢失，上层协议必须重传。

10.6 地址解析协议 ARP

ARP（Address Resolution Protocol）用来将 IP 地址转换成与其对应的物理地址，比如在以太网中就是以太地址。在 IP 将数据以 sk_buff 的形式交给网卡进行传送时就需要这种转换。

ARP 协议会进行多种检测来判断相应设备是否需要硬件信息头，如果需要，就要为该数据包创建。在 Linux 系统中，硬件信息头会被缓存以避免重复的创建。硬件信息头由相应设备提供的例程来创建。对以太网设备而言，它们使用相同的创建例程，其中在进行 IP 地址向物理地址的转换时就要用到 ARP。

ARP 协议是比较简单的，只包含 ARP 请求与 ARP 回答两种报文类型。ARP 请求报文中提供了需要进行解析的 IP 地址，而 ARP 回答报文则提供了与该 IP 地址相对应的物理地址。ARP 进行工作时，将 ARP 请求报文在本网中进行广播，以使网上的所有机器都能收到该请求报文。拥有 ARP 请求报文中的 IP 地址的机器就用 ARP 回答报文进行回答，在该报文中提供其物理地址。

在 Linux 中，ARP 协议是以 arp_table 这一表为核心进行工作的，该表的每一项对应一个由 IP 地址向物理地址的转变。该表中的表项在进行 IP 地址解析时创建，当这种解析长时间不用时就会被删除。Arp_table 结构包含以下各域：

last used　记录 ARP 表最后一次被使用的时间，
last updated　记录 ARP 表最后一次被修改的时间，
flags　描述表项的状态，比如是否完整等等，
IP address　解析所涉及到的 IP 地址
hardware address　解析出的硬件地址
hardware header　指向缓存中硬件信息头的指针，
timer　这是一个 timer_list 表项，用于当 ARP 请求没有回应时的超时操作，
retries　该 ARP 请求被使用的次数，
sk_buff queue　等待对 IP 地址进行解析的 sk_buff 队列

ARP 表包含一组指针（又称为 arp_tables 向量），它们指向 arp_table 中的表项链，arp_table 的表项一般被缓存以便于访问，操作时以 IP 地址的最后两个字节作为索引检索到相应的指针，然后再在该链中进行严格的匹配从而找到相应的表项。Linux 还针对 arp_table 中的表项预先建立硬件头信息并保存在 hh_cache 结构中。

当一个表项的 IP 地址没有在 arp_table 中出现时，ARP 协议就发送相应的 ARP 请求报文进行解析。此时，要在 arp_table 表中建立新的一项，并将需要进行转换的 sk_buff 链接在该表项上。ARP 在发出 ARP 请求报文的同时，还要设置一个 ARP 超时时钟。在对于其 ARP 请求没有回应的情况下，ARP 本身会重复几次操作，如果全部失败，就将新建的 arp_table 表项删除，对应的 sk_buff 也将交给上层协议进行处理。UDP 对这种情况不进行任何处理，而 TCP 则可能会在已有的 TCP 连接上重传几次。但如果网络对于 ARP 请求进行了正确的回答，ARP 就能进行正确的地址转换，从而新建立的 arp_table 表项就被标识为完整的，链在该项上的 sk_buff 结构也就能继续进行传送了，在此之前，物理地址将被置入每一个 sk_buff 中。

ARP 协议必须对那些针对于自身 IP 地址的 ARP 请求报文进行回答，此时协议将对其协议类型(ETH_P_ARP)进行登记，同时生成 packet_type 类型的数据结构。这意味着物理层所收到的所有 ARP 报文都应传给 ARP 协议。

网络的拓扑结构可能随着时间的变化而变化，相同的 IP 地址也可能被赋予不同的物理地址，例如，拨号上网服务一般就将同一 IP 地址用于不同的连接。为了使 ARP 表中的数据不过期，ARP 使用一个周期性时钟对 arp_table 进行维护，将超时的表项删除。维护时要注意的一点就是不要删除保存有硬件信息头的表项，因为其它的数据结构有可能要使用这些信息。Arp_table 中有一些表项是永久性的，不能删除。由于 arp_table 使用的是内核空间，故该表不能太大，当该表已经到达所允许的上限值时，如果还要增加新的表项，就将不得不删除表中存在时间最长的项。

10.7 IP 路由

IP 的路由功能决定应如何发送 IP 报文。事实上，传送 IP 包时要进行大量的信息判断，比如目的地能否到达？如果目的地能到达，应使用那一个网卡？如果有多个网卡可供使用，用哪一个更好些？IP 路由信息数据库就是用来提供回答这些问题的信息。该数据库主要包含两个部份，其中发送信息数据库（Forwarding Information Database）是最重要的一个，它包含了所有已知的 IP 及到达该 IP 的最佳路由。另外，还有一个路由 cache，用于快速搜索 IP 路由。与其它 cache 一样，路由 cache 只包含发送信息数据库中最常使用的那些路由。

路由是通过 BSD socket 界面的 IOCTL 请求进行填加与删除的，这些操作都将交给相应的协议进行处理。INET 协议层只允许拥有超级用户权力的进程对 IP 路由进行操作。IP 路由可以是固定的，也可以是随时间动态变化的。除路由器之外的大部份系统都使用固定的 IP 路由。而路由器则不停地运行路由协议，对已有的路由进行检查。非路由器系统一般都称为端系统，其中的路由协议以后台程序（如 GATED）的形式对路由进行维护，当然，也可以在 BSD socket 的 IOCTL 请求下对路由进行显式的增、删操作。

10.7.1 路由 Cache

在查询 IP 路由时，系统首先在路由 cache 中查找,在找不到的情况下再在发送信息库中搜索。如果两次都找不到，该 IP 包的发送就不能成功，由应用程序进行出错处理。如果找到的路由在发送信息库中但不在路由 cache 中，则路由 cache 就将被修改，即在 cache 中创建一个新表项并将该路由信息加入。路由 cache 事实上也是一个指针（ip_rt_hash_table），

其中每一项都指向一条由 rtable 数据组成的链表。对该表的检索也是以 IP 地址的后两字节作为关键字进行，因为这两个字节能够较好地区别两个不同目的地，从而较好地对该表进行哈希操作。Rtable 的表项描述了有关路由的信息，比如目的 IP 地址、对应的网卡、在该网卡上每次能传送数据的最大量等等，同时还包含索引计数、使用计数以及最后一次使用的时戳等信息。其中索引计数值表示使用该路由的网络连接的数量，它随着连接的变化而变化。使用记数信息记录了路由表被

使用的次数，同时它被用于对 rtable 中的表项进行排序。最近使用时戳被用来周期性的检测路由 cache，将过时的表项（即长时间不用的表项）删除。路由 cache 中的各路由按其使用频率进行排列，使用频繁的排在前面，以便于路由的搜索操作。

10.7.2 发送信息数据库（Forwarding Information Database）

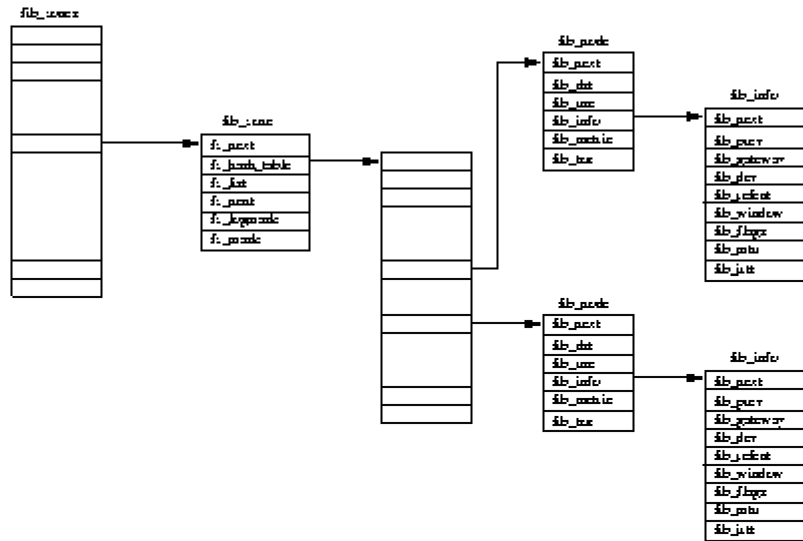


图 10.5 发送信息数据库

发送信息数据库（如图 10.5 所示）指出了某时刻系统所能使用的 IP 路由。系统虽然对数据库进行了合理有效的组织，但由于使用的数据结构较复杂，搜索的效率并不是很高，尤其是在对每一个 IP 包都进行搜索时。系统使用路由 cache 来解决这一问题，路由 cache 保存了一些已知的较优路径以加速 IP 包的传送。如上文所述，路由 cache 是根据发送信息数据库来建立的。

IP 子网信息保存在 fib_zone 结构中，而一个系统中的所有 fib_zone 结构都由 fib_zones

哈希表中的指针进行控制。用 IP 子网的掩码作为对该表进行哈希操作的关键字。同一子网中的路由信息保存在 fib_node 与 fib_info 这一对数据结构中，多条路由的信息就以队列的形式组织，并由 fib_zone 中的 fz_list 指针指向该队列。如果某一子网中的路由数目很大，系统就会使用一中哈希函数来进行对 fib_node 的搜索。

在某些情况下，通往某一 IP 子网的路由有多条并且这些路由可选择多个网关。但在 IP 层，不允许通往同一子网的多条路由使用同一个网关，即如果到达某一子网有多条路由，那么这些路由必须

使用互不相同的网关。同时，还有一个参数（metric）描述每一条路由的优越性，事实上，该参数记录了为了到达目的子网所必须经过的 IP 子网数，显然，参数值越大，路由越差。

第 11 章 核心机制

本章描述了 Linux 核心的一些基本任务和机制，有了它们， 核心中的其他部份才能够在一起有效地运转.

11.1 Bottom Half Handling (任务的延迟处理)

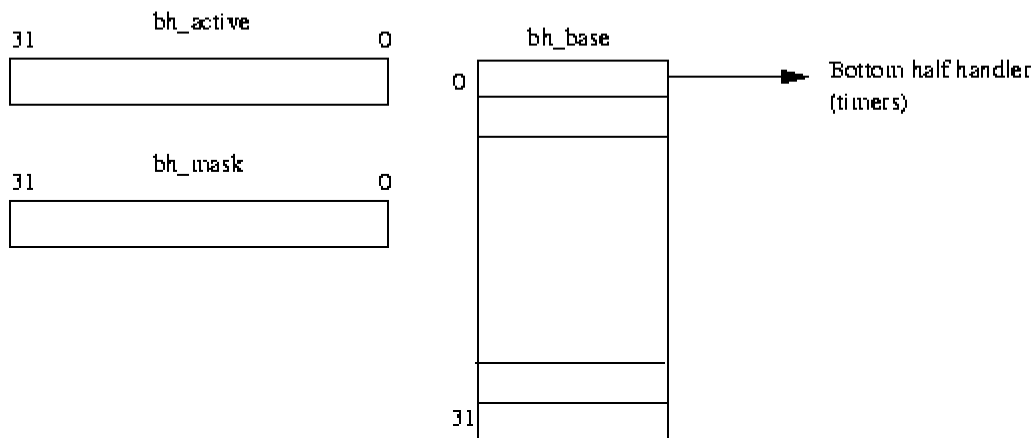


图 11.1 : Bottom Half Handling 数据结构

核心中经常会遇到一些需要推迟处理的工作。 中断处理的过程就是一个很好的例子。 当中断发生时， 处理器停下正在做的工作， 操作系统把中断事件通知给相应的设备驱动程序。 设备驱动程序不应当用过多的时间处理这个中断， 因为这时系统的其他部份全部停了下来。 通常总是有一部份工作不必非要现在就做完， 晚一点处理也行， (从而让系统的其他部份能够早点恢复运行)。 Linux 中设计了 bottom half handlers 使设备驱动程序和核心的其他部份能够排队处理延迟的工作。 图 11.1 中给出了同 bottom half handling 有关的核心数据结构。

核心中最多可以有 32 个不同的 bottom half handler， bh_base 是一个指针向量，这些指针就指向核心的各个 bottom half haldling 过程。 bh_active 和 bh_mask 中的 bit 位指示了系统中安装的 handler 和正在进行处理的 handler。 如果 bh_mask 的第 N bit 位是 1， 那么 bh_base 向量中的第 N 个指针份量就指向一个 bottom half 处理过程。 如果 bh_active 的第 N bit 位是 1， 那么核心中的调度程序就会在适当的时候调用这个 handler。 Handler 的顺序是静态定义的: timer bottom half haldler 优先级最高, console bottom half handler 其次， ... 通常 bottom half handling 过程具有自己的 task 列表。 例如， immediate bottom half handler 就用 immediate tasks queue (tq_immediate) 来排队需要及时处理的 task。

核心中有些 bottom half handler 是设备专用的， 下面一些则是通用的:

TIMER (定时器)

这个 handler 每当系统的时钟中断发生时就激活， 系统用这个 handler 来驱动核心的 timer 队列.

CONSOLE (控制台)

这个 handler 处理控制台消息。

TQUEUE

这个 handler 处理 tty 消息。

NET

这个 handler 负责网络方面的处理工作。

IMMEDIATE

这是一个通用的 handler， 一些设备驱动程序用它来排队需要推迟处理的工作。

当设备驱动程序或者核心的其他部份需要安排推迟完成的工作时， 就把工作加到系统队列去， 比如 timer 队列， 然后通知核心有 bottom half handling 需要完成， 方法就是设

置 bh_active 的相应的 bit 位。 如果驱动程序把工作加入了 immediate 队列并要求 immedia bottom half handler 来处理它， 就会设置 bh_active 的第 8 bit 位。 在每次系统调用结束，返回调用进程之前， 核心就要检查 bh_active ， 如果有某个 bit 位被设置了， 相应的 handler 就会被调用。 检查的顺序是先第 0 bit， 最后是第 31 bit。

当某个 handler 被调用了之后， bh_active 中的相应 bit 就被清除。 所以 bh_active 的值只在瞬间有效。 它只在对调度器的调用之间有意义， 是为了避免无谓地调用这些 handling 例程。

11.2 Task Queues (任务队列)

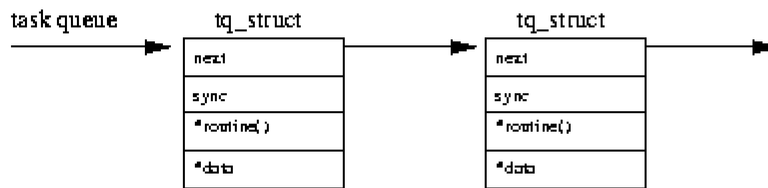


图 11.2: 一个任务队列

核心中使用任务队列管理推迟的工作。 Linux 有一个通用的机制来对工作排队， 以便以后处理。

任务队列经常和 bottom half handler 一起使用。 例如： 当 timer queue buttom half handler 运行时， 它要处理 timer 任务队列， 提供定时器服务。 任务队列是一个简单的数据结构， 请参看图 11.2， 它包括一个 tq_struct 数据结构的单链表， tq_struct 包括一个例程的地址以及一个指针指向一些数据。

当这个任务队列被处理的时候， 每个 tq_struct 单元中的例程就会被调用， 参数则是那个指针。

核心的各个部份， 例如设备驱动程序， 都能创建并使用任务队列。 核心自己创建和管理着下面 3 个队列：

1。 timer (定时器)

这个队列中的工作在下一次系统时钟中断发生时就会被处理。 每当时钟滴答一次，就检查一下这个队列是不是有工作要处理， 如果有， timer queue bottom half handler 就被标记为 active. 当调度器下次被调用的时候， 就会调用 timer queue bottom half handler。 不要把这个队列和系统定时器混淆起来， 那是一个复杂得多的机制。

2。 immediate (尽快)

这个队列也会被调度器处理。 但是它的优先级没有定时器队列那么高。

3。 scheduler (调度器)

这个任务队列由调度器直接处理。 这是被用来支持系统中的其他任务队列的。 也就是说， 它的任务是一个个处理任务队列的例程， 例如， 设备驱动程序的任务队列的处理就会被添加到这里。

处理任务队列的时候， 队列中指向第一个节点的指针被删除， 代之以一个空指针。 实际上， 这个操作不可中断， 是个 "原子" 操作。 然后队列中的每个节点的处理例程都会被依次调用。 队列中的节点通常是静态分配的数据。 但是没有一个统一的机制来时方所分配的内存。 负责处理任务列表的例程在调用完节点的处理例程之后， 就简单地移向下一个节点。 那个任务自己应该担负起释放核心内存的责任。

11.3 Timers (定时器)

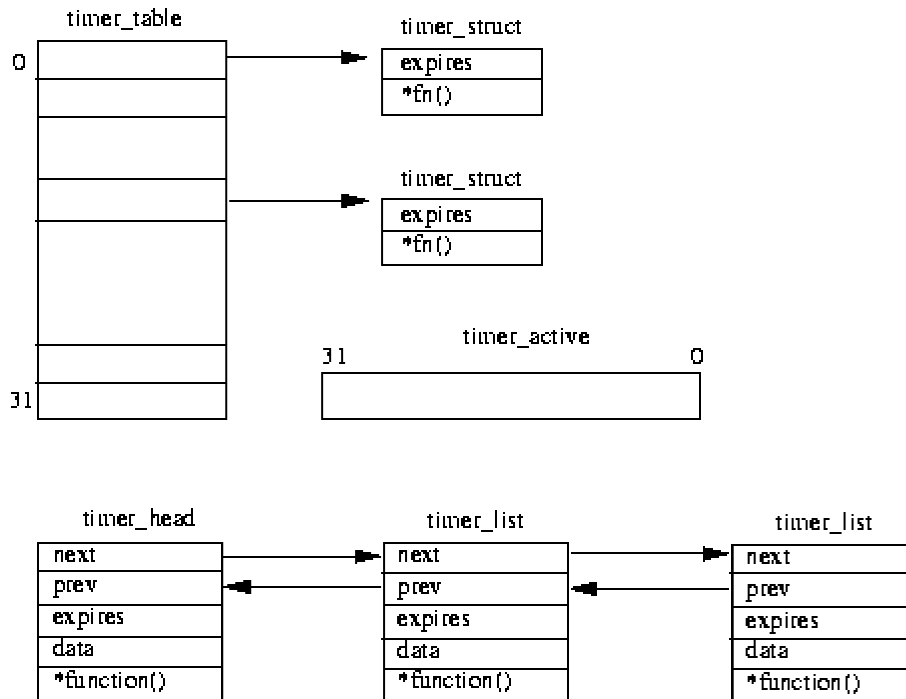


图 11.3: 系统定时器

操作系统经常需要能把一项活动安排在将来某个时间进行。 需要有一种机制保证能够把任务安排到一个相对比较精确的时间进行处理。 任何能够支持操作系统的微处理器必定有一个可编程的间隔定时器， 能够周期性地中断处理器。 这个周期性的中断就是系统时钟的滴答。 如果把系统比作一个交响乐队， 它就像一个节拍器， Linux 对于时间的理解很简单： 它从系统启动开始记录系统时钟的滴答数目， 以此衡量时间。 系统中用到时间的地方都以此为衡量的基础， 这就是所说的 jiffy 单位的由来。 核心中有一个全局变量叫 jiffies, 就是这个记录的值。

Linux 中有两种系统定时器， 都能把事务排队等到某个系统时间来处理。 但是这两者的实现稍有不同。 图 11.3 中对比了这两种机制。

第一种是老式的定时器机制， 有一个静态的数组， 包含 32 个 timer_struct 数据结构以及一个用屏蔽位指示活动定时器的整数： timer_active 。

由于定时器列表中的定时器是静态定义的，（就像 bottom half handler 表中的 bh_base），表中的表项一般都是在系统初始化的时候加入的。

第二种新机制使用了 time_list 数据结构的链表把定时器按到达时间的早晚从早到晚链接起来。

两种机制都使用了 jiffy 作为时间单位。因此，一个 5 秒的定时器要把 5 秒转换为 jiffy 单位，然后和当前系统时间相加，以此设置定时器。每当系统时钟滴答一次，timer bottom half handler 就被标记为 active，这样当调度器被运行时，就会处理定时器队列。timer bottom half handler 对两种系统定时器都要处理。

对于老式的系统定时器，要检查 timer_active 屏蔽位找出设定了相应位的定时器，如果某个定时器设定的时间已到(它设定的时间小于当前系统时间)，它的定时器例程就会被调用。并且对应的屏蔽位会被清除。对于新式的系统定时器，则检查 timer_list 链表中的节点，每个到期的定时器会被从链表中清除，并且节点中的定时器例程会被调用。新机制的好处是在调用定时器例程的时候能够传递一个参数。（当然新机制的显而易见的好处是不限制定时器的数目，这是老式的机制不能做到的，译者注)。

11.4 Wait Queues (等待队列)

进程经常需要等待系统资源。例如，进程可能需要一个目录的 VFS i 节点而这个节点可能不在文件系统的高速缓冲区中。这时，进程就必须等待系统从包含这个 i 节点的物理介质上面得到这个节点的内容才能继续运行。

wait_queue

图 11.4：等待队列

Linux 核心使用了一个很简单的数据结构， 叫做等待队列(见 图 11.4)， 它包含一个指向进程的 task_struct 结构的指针， 还有一个指向队列中下一个节点的指针。

被加进这个队列的进程既有不可中断的(uninterruptible)有又可以中断(interruptible)的(进程的状态中有 UNINTERRUPTIBLE 和 INTERRUPTIBLE 来标志). 可中断的进程可能会被事件打断， 比如它的定时器到时了， 或者有信号发送给这个进程。 因为现在这个进程不能继续运行， 调度器会运行， 当它选择了新的进程运行时， 这个等待的进程就被暂停了。

在处理等待队列的时候， 队列中的每个进程的状态都被设置为 RUNNING (运行态)。 如果这个进程已经被移出了运行队列(run queue)，它又会被放回去。 当调度器下次运行时，等待队列中的进程就又有机会被选中运行了， 因为它们已经不再等待什么了。 当处于等待队列中的进程被调度运行时， 它做的第一件事就是把自己从等待队列中移出。 等待队列可以被用于同步对系统资源的访问， 在 Linux 系统中用于实现 semaphore (信号量， 下面会讲到)。

11.5 Buzz Locks (Buzz 锁)

更多的叫法是转锁(spin lock)。 它是保护数据结构或者一段代码的最基本的方法。 它一次只允许一个进程处于危险(可能产生冲突)的代码区域(临界区)。 在 Linux 中， Buzz 锁被用于限制对数据结构中的域的访问。 使用一个整数作为锁， 初始值为 1。 每个想进入临界区的进程试图把锁的值从 0 改变为 1。 如果当前的值就是 1， 进程就会再试一次。 通过一个紧致的循环反复尝试。 对保存锁的值的内存区域的访问必须是原子化的， 读取当前数值， 检查是不是 0 以及把它改为 1 这些操作必须一次完成， 不能被别的进程中断。 大多数 CPU 提供了特殊的指令来实现这一点， 但是也能够使用不缓冲的主存来实现 buzz 锁。

当拥有这个锁的进程离开临界区之后， 它需要释放锁， 把锁的值改为 0。 其它正在转这把锁的进程现在就会读出 0。 第一个来读的进程将会再把它改为 1 并进入临界区。

11.6 Semaphores (信号量)

信号量被用于保护代码的临界区或数据结构。 请注意每次对临界数据的访问， 例如访问一个目录的 VFS i 节点， 是由核心代码代表用户进程进行的。 允许一个进程能够改变改变另一个进程的也在使用的临界数据结构是非常危险的。 一种办法是使用 buzz 锁来防止冲突， 但是这种最简单的方法性能很差。 Linux 系统中使用信号量来做保护： 只有得到信号量保护的资源的那个进程能够访问临界区， 而其它等待资源的进程被暂停。

Linux 中的信号量数据结构包含下列信息：

count (计数)

这里记录需要使用资源的进程的数目。 一个正的数值表明这个资源现在可用。 一个负的数值和 0 表明现在有进程在等待这个资源。 如果初始数值为 1， 那么同一个时刻只能有一个进程能够使用这个资源。 如果进程需要使用这个资源， 就把计数值减 1， 使用完资源之后再把这个值加 1。

waking (唤醒)

这是正在等待资源的进程的数目。当资源被释放的时候，就有这么多进程等待被唤醒。

wait queue (等待队列)

当进程等待这个资源的时候， 它们被放在这个等待队列里面。

lock

一个 buzz 锁用来保护对 waking 域的访问。

假设信号量的初始值是 1， 第一个来访的进程看到数值 1， 并把它减 1 变为 0。 这个进程就"拥有"了被信号量锁保护的资源。 当进程不再需要使用资源之后， 就把信号量的计数加 1。 最理想的情况就是没有其它的进程竞争这个资源。 Linux 对信号量的实现在这种最常见的情况下效率很高。

当另一个进程也想同时访问临界区的时候， 它也要把计数减 1。 因为现在计数为 -1， 所以这个进程不能进入临界区。 Linux 让这个等待的进程睡眠(进入等待状态)。 直到拥有资源的进程释放资源的时候来唤醒它。 等待的进程把自己添加到信号量的等待队列上， 然后进入循环， 检查 waking 域的值并调用调度器直到 waking 域的值不为 0。

拥有资源的进程在释放资源的时候， 增加信号量的计数， 并检查， 在理想情况下， 计数被恢复为 1， 没有什么事情要做； 如果不大于 0， 说明还有进程在等待这个资源。 这时， 就增加 waking 域的值， 并唤醒一个在信号量的等待队列里的进程。 当这个等待的进程被唤醒的时候， waking 现在是 1， 它就知道自己现在拥有这个资源， 可以进入临界区了。 它就减

少 waking 的值， 然后继续运行。 对 waking 域的访问要通过一个 buzz 锁来保护， 这个锁使用信号量的 lock 域作为自己的值。

第 12 章 模块

本章将讲述 Linux 内核是如何按需动态装载和卸掉模块的。

Linux 是单内核结构，也就是说， 它是一个大程序，其中任一函数都可以访问公共数据结构和其它函数调用。 (作为操作系统)另外一种可能的结构是多核式的， 各功能块自成一体， 相互之间由严格的通信机制相连。单核结构在添加新模块时，一种方法是重新调整设置，所以非常费时。比如，你想在内核中加一个 NCR 810 SCSI 的驱动程序， 你必须重新设置， 重建内核。这也有另外一个办法, Linux 允许动态装载和卸掉模块. Linux 模块是一段可以在机器起动后任意时间被动态连接的代码. 在不需要时, 它们可以被从内核中卸掉. 大多数 Linux 模块是设备驱动程序或伪设备驱动程序, 如网络驱动程序, 文件系统等.

你可以使用 insmod 和 rmmod 命令来装载和卸掉 Linux 模块, 内核自己也可以调用内核驻留程序(Kerneld) 来按需要装载和卸掉模块.

按需动态装载模块可以使内核保持最小, 并更具灵活性. 我现在的 Intel 内核由于大量使用动态装载模块, 只有 406 K 字节. 例如， 我很少用到 VFAT 文件系统, 所以我让 Linux 内核只在我装载 VFAT 分区时, 才自动上载 VFAT 文件系统. 当我卸掉 VFAT 分区时, 内核会检测到, 并自动卸掉 VFAT 文件系统. 当测试新程序时, 你如果不想每次都重建内核, 动态装载模块是非常有用的. 但是, 运用模块会多消耗一些内存, 并对速度有一定影响. 并且模块装载程序是一段代码, 它的数据将占用一部份内存. 这样还会造成不能直接访问内核资源, 效率不高的问题.

一旦 Linux 模块被装载后, 它就和一般内核代码一样, 对其它内核代码, 享受同样的访问权限。换句话说， Linux 内核模块可以像其它内核代码, 或驱动程序一样使系统崩溃。

模块可以使用内核资源，但首先它需知道怎样调用．例如，一个模块要调用 Kmalloc() (内核内存分配程序)．但在模块建立时，它并不知道到哪儿去找 Kmalloc()，所以在它被装载时，内核必须先设定模块中所有 Kmalloc() 调用的函数指针．内核有一张所有资源调用的列表，在模块被装载时，内核重设所有资源调用的函数指针．Linux 允许栈式模块，即一个模块调用另一个模块的函数．例如，由于 VFAT 文件系统可以看成是 FAT 文件系统的超集，所以 VFAT 文件系统模块需要调用 FAT 文件系统提供的服务。一个模块调用另一模块的资源与调用内核资源很相似。唯一的不同是被调用的模块需被先载入。一个模块被载入后，内核将修改它的内核符号表(KERAEL SYMOBOL TABLE)，加入新载入模块提 供的所有资源和符号。所以另一个模块被载入时， 它就可以调用所有已载入模块提供的服务。

当卸掉一模块时，内核先确定该模块不会再被调用，然后通过某种方式通知它。在该模块被内核卸掉以前，该模块须释放所有占用的系统资源。例如，内存或中断，当模块被卸掉后，内核从内核符号表中删除所有该模块提供的资源。

如果模块代码不严谨，它将使整个操作系统崩溃。另一个问题，如果你载入的是为其它版本服务的模块，那怎么办？例如，一个模块调用一个内函数，但提供了错误的输入参数，这将导致运行错误。但内核可以在模块被载入时选择性地通过严格版本检查来杜绝这种现象。

12。1 载入一个模块

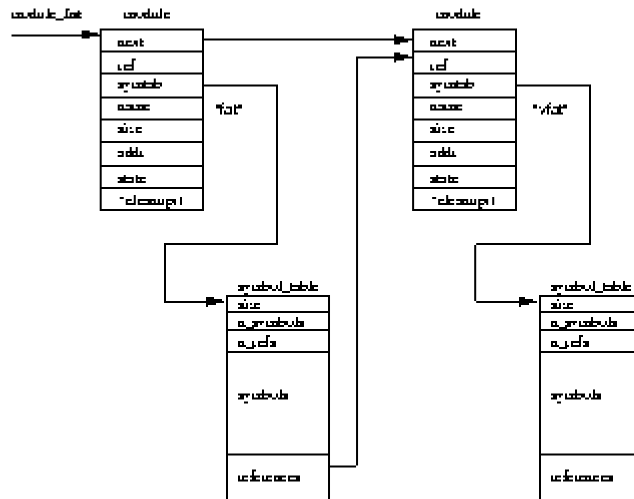


图 12.1 核心模块链表

载入模块有两种方法。第一种是通过 INSTALL 命令来载入；

另一种更聪明的方法是在模块被调用时自动载入，这叫所需载入(DEMAND LOADING)。

例如，当用户在装一个不在内核中的文件系统，内核会自动调用内核驻 留程序(KERNELD)
来载入对应的处理模块。

内核驻留程序是一个具有超级用户极限的普通用户程序。当它被启动时(通常在系统
启动时)，它将打开一个和内核之间的进程间通信管道(IPC CHANNEL)。 内核将利用这
条管道来通知进程驻留程序去完成各种任务。

内核驻留程序的主要任务是载入和卸掉模块，它也能完成其她一些任务。如按需打开和关掉一条通过串口的 DDD LINK。KERNELD 自己并不完成这些任务。它将调用如 INSMOD 这样的命令来完成，KERNELD 只是一个内核的代理，协调完成各项任务。

载入模块时，INSMODE 命令必须先找到要被载入的模块。可所需载入的模块通常被放在/LIB/MODULES/KERNEL-VERSION 下，这些模块与一般系统程序都是已连接好的目标代码，不同处在于模块是可重定位的映像文件。也就是说，模块并不是从一个固定的地址开始执行的。模块可以是 a.out， 也可以是 ELF 格式的目标代码。INSMODE 通过一个有系统权限的调用来找到内核中可被调用的资源。

系统(资源)符号由名和值俩部份组成。内核用 MODULE_LIST 指针指向其管理的所有模块所串成的链表。内核的输出符号表在第一个 MODULE 数据结构中，并不是内核所有的符号都能被模块调用，可调用符号必须被加入输出符号表中，而输出符号表是与内核一起编译连接的。例如，当一驱动程序想控制某一系统中断时，她需调用”REQUEST_IRQ”这样一个系统函数，在我机器的内核中，它现在的值是 0x0010cd30，你可以看/PROC/KSYMS 文件或用 KSYMS 来查询。KSYMS 命令可以显示所有内核输出符号的值，也可以显示载入模块的输出符号的值。当 INSMOD 载入模块时，它先将模块载入虚存，根据内核输出符号，重设所有内核资源函数调用的指针。即在模块的函数调用处写入对应符号的物理地址。当 INSMOD 重设完内核输出符号的地址后，它将调用一个系统函数，要 求内核分配足够的空间。内存就会分配一个新的 MODULE 数据结构和足够的内存来装

载这个新模块，并把这个 MODULE 数据结构放在模块链表的最后， 置成未初使化(UNINITALIZED)。

表 12。1 显示的是内核载入 FAT 和 VFAT 两模块后的模块链表。链表的第一模块并没有显示出来，那是一个伪模块，只是用来记录内核的输出符号表。你可以用 ISMOD 命令来列出所有载入模块及它们之间的关系。ISMOD 只是格式化的输出记录内核链表的/PROC/MODULES 文件。INSMOD 可以访问内核分配给新载入模块的内存，它先将模块写入这块内存，然后对它进行重定位处理，使模块可以从这个地址开始执行。由于每次模块被载入时，无论在不在同一台机器上，都不大可能分配到相同的内存地址，所以重定位(即重设它的函数指针)是必须的。

新载入模块也可以输出符号，INSMOD 会为这些符号建一个表。另外，每一个模块必须有自己的初始和清理(即析构)函数。这两个函数不能被输出，但它的地址将在初使化时由 INSMOD 传给内核。

当一个新模块被载入内核时，它要更新系统符号表及被它调用的模块。内核中被调用模块都需在其符号表的最后保留一列指向调用模块的指针。图 12。1 显示 VFAT 文件系统依赖于 FAT 文件系统，所以，在 FAT 模块中有一个指向 VFAT 的指针，这个指针是在 VFAT 被载入时加入的。内核将调用模块的初使化函数，如果成功，它将继续完成安装新模块的任务。模块的清理函数的地址将被存在它的 MODULE 数据结构中。当模块被卸掉时，它将被调用。到这里模块的状态被置为"运行"(RUNNING)。

12。2 卸掉模块

用 RMMOD 命令可以卸掉一个指定模块，但按需载入模块没用时，它会被内核自动卸掉，KERNELD 每次被激活时，它会调用一个系统函数将所有没用的模块从内核中卸掉。例如，如果你装了一个 ISO9660 的 CDROM，并且它的文件系统是一个按需载入模块，那么当你卸掉 CDROM 后不久，ISO9660 文件系统也会被从内核中卸掉。你可以在起动 KERNELD 时，设置其被激活的时间间隔，我的 KERNELD 每 180 秒被激活一次。

当还在被其他模块调用时，模块是不能被卸掉的。例如，当你还在用 VFAT 文件系统时，VFAT 模块不会被卸掉。当你看 ISMOD 命令的输出时，你会发现每个模块都带有一个计数器。这个计 数器记录依赖于该模块的模块数。在上面的例子中， VFAT 和 MSDOS 都依赖于 FAT 模块，所以 FAT 模块的计数器为 2，VFAT 和 MODOS 的都为 1，表示只有文件系统依赖于它们。如果我再装

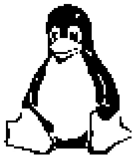
入一个 VFAT 文件系统，VFAT 模块的计数器将变成 2。模块的计数器是它映像的第一个长字(LONGWORD)。

这个长字同时也记录了 AUTOCLEAN 和 VISITED 两个标志，只有按需载入模块才用到这两个标志。AUTOCLEAN 用来使系统识别哪一个模块需被自动卸掉。VISITED 标志表示该模块是否还在被其它模块调用。每次 KERNELD 试图卸掉已没用的按需载入模块时，系统检查所有模块。它只注意标为 AUTOCLEAN 并正在运行的模块，如果这个模块没有设置 VISTIED 标志，它将被卸掉。否则，系统就清掉 VISTIED 标志，并继续检查下一模块。

当一个模块可以被卸掉时，系统会调用它的清理函数来释放它所占用的所有系统资源。

该模块的 MODULE 数据结构将被标为 DELEDTED，并从模块链表中去除，所有它依赖的模块会修改它们的指针，表示该模块已不再依赖它们了。所有该模块占用的内存将被释放掉。

第 13 章 Linux 源代码结构



本章介绍 Linux 原代码的组织。从而有兴趣的读者可以阅读原码。

本书并不要求读者具有 C 语言的经验和必须手边有 Linux 原代码。然而阅读原代码对理解 Linux 操作系统的一个非常好的练习方法。本章给出一个核心原码的综述和其目录组织。

从哪哩得到 Linux Kernel 原码

(译者注：这里略去原作者有点旧的一段。)

通常来讲，读者可以访问：www.redhat.com 得到Linux系统。另外我们强烈鼓励读者访问：www.linuxhq.com

和www.linux.com

Linux 核心原码版本组织是一个简单的编码系统。任何偶数编码的核心版本(如 2.0.30)都是一个稳定的，正式发布的版本。任何齐数的版本表示其是一个正在开发中的核心。本书基于 2.0.30 核心版本。我们鼓励用户尝试开发中的版本。从而整个 Linux"社区"可以来测试最新的代码。

核心代码的变动词藏以 patch 的形式发布。Linux 提供一个实用命令 patch 来对原代码进行更新。例如，你当前有一个 2.0.29 的原码但想要更新到 2.0.30 上。你要做的是：得到 2.0.30 的 path 文件并相应地将你的原码更新.

$ cd /usr/src/linux
$ patch -p1 < elf.h中定义了ELF文件的结构格式。 linux 中。include herneld.h module.h和include 其数据结构和kerneld核心监控程序定义在include modules.c中。核心模块代码一部份在核心中，一部份在modules包中。核心中的部份在kernel 模块 net中。ipv4 中。网络设备驱动程序在drivers IP网络代码在net core中。TCP 在net af_inet.c 中。协议的一些通用代码(包含sk_buff处理例程) ipv4 socket代码在net INET socket.c 中。IP版本 4 net中。BSD 网络代码的include文件在include 网络 buffer.c。核心监控程序update也在这里。 *。缓冲区代码在fs 代码在fs fs.h。相应 ext2_fs.h和 ext2_fs_sb.h。虚拟文件系统数据结构定义在include ext2_fs.h， ext2 目录下。其数据结构定义在include 为EXT2 文件系统的原码在fs 文件系统 支持声卡的驱动程序。 sound 网络代码。 net 含有所有的SCSI代码和Linux所支持的SCSI设备驱动程序 scsi PCI伪驱动程序的代码。 pci scsi下的scsi.c中。 代码在drivers CD的 block的ide-cd.c中。SCSI CDROM的代码在drivers Linux关于CDROM的代码。请注意关于IDE cdrom 字符设备如ttys, 串行口和鼠标等。 char nfs文件系统。块设备包含基于IDE和SCSI的设备。 我们需要网络来"mount" genhd.c中的device_setup()函数。该初始化过程不仅对硬盘，也对网络初始化因为 block drivers 块设备驱动程序，如ide.c。如果你想看所有可能含有文件系统的设备的初始化，你应该看 并且分为下列次子目录： 绝大多数Linux核心代码是关于设备驱动程序。所有的Linux设备驱动程序原码在drivers子目录中。 设备驱动程序 irq.h中。 asm-i386 irq.c中。其定义在include kernel i386 arch 核心中中断处理的代码基本上是与特定的硬件体系结构相关的。Intel的中断处理代码在 中断处理 pipe.c中。 sem.c中。管道在ipc shm.c中。信号灯(semaphores)在ipc memory)在ipc (shared msg.c中。共享内存 V消息传递(message)的实现在ipc ipc.h中。System 其定义在include IPC对象包括一个 ipc_perm数据结构。 V 进程间通信所有的代码都在ipc目录中。所有的System 进程间通信 bios32.c中。 alpha AXP的部份在arch BIOS代码。Alpha 的PCI pci.h中。每中结构有其特定 pic.c中。系统定义在include PCI伪驱动程序在drivers PCI sched.h中找到。

task_struct数据结构可以在include interrupt.h中。 Half处理代码在include fork.c中。Bottom 中。fork代码在kernel sched.c kernel中。调度代码在kernel * 大多数通用的代码在kernel目录中。与硬件相关的在arch 核心 swap_state.c中。 缓冲代码在mm buffer.c中。对换 filemap.c中。缓冲区cache代码在mm 中。内存映象和页面缓冲代码在mm memory.c mm中。页面错处理代码在mm 大多数代码在mm目录中。少量与体系结构有关的在arch 内存管理 main.c中的main()函数。 相关的设置工作，然后跳转到init head.S关于上述过程。Head.S做一些与体系结构 核心进入开始阶段。请阅读 对于基于Intel的系统，当loadlin.exe或LILO将核心装载进入内存中并将控制交给核心时， 系统初启和初始化 从而使得读者的阅读更容易一些。 阅读象Linux核心这样一个巨大的复杂的系统是令人畏缩的。下面提供一些好的建议和介绍， 从哪开始了解Linux核心 该目录含有一些scripts工具，如awk和tk等等。核心构建时用得着。 scripts lib中。 该目录含有核心的库代码。与硬件相关的部份在arch lib 关于网络的代码。 kernel中。 主要的核心代码。与硬件相关的部份在arch vfat和ext2 等。 文件系统代码。细分为一些子目录。每个子目录对应一种文件系统。如 含有装载模块代码的目录。 modules 该目录含有核心的进程间通信代码。 ipc 如block。 所有的的设备驱动程序的目录。根据设备类型，进一步划分为一些子目录。 fault.c mm。例如，arch 该目录中含有所有的内存管理代码。与体系结构相关的内存管理代码在 该目录中是核心的初始化代码。建议读者阅读核心代码从这开始。 核心配置程序。 如果想要改变体系结构，你要编辑核心的makefile并且从新运行Linux asm-i386。 asm子目录是一个软连接，其真正指向的是对应于一个体系结构的目录，例如，include include 该目录下放着系统构建所需的大多数include文件。它也包含一些对应于不同体系结构的子目录。 。例如，i386 和Alpha。 该目录下放着所有的与体系结构有关的核心代码。该目录下还包含其它子目录分别对应不同的体系结构linux下，你可以看见许多目录： src usr 在Linux原码的根目录 核心原码的组织 这样你就不需要从新拷贝所有的原码。一个非常好的提供核心patch站点是：www.linuxhq.com

第 14 章 Linux数据结构

该附录描述了 Linux 使用的主要数据结构。

block_dev_struct

block_dev_struct 数据结构用来登记块设备以被缓冲区使用。这些结构被存放在 blk_dev vector 中。

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

buffer_head

The buffer_head 数据结构用来存放缓冲区中的一个数据块的信息。

```
/* bh state bits */
#define BH_Uptodate  0   /* 1 if the buffer contains valide 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

系统中每一个网络设备都对应于一个设备数据结构。

```
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 数据结构用来登记字符和块设备(含有相应的名字和对此设备的文件操作集)。每一个 chrdevs and blkdevs 向量的入口对应一个字符或块设备。

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

file

每个打开的文件， socket 接口都对应一个文件数据结构。

```
struct file {
  mode_t f_mode;
  loff_t f_pos;
  unsigned short f_flags;
```

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

files_struct

files_struct 数据结构描述一个正被打开的文件。

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

fs_struct

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

gendisk

gendisk 数据结构含有一个硬盘的信息。当系统初始化，发现硬盘和检测分区参数时，此结构被填充。

```
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 数据结构包含一个磁盘中的文件或目录的信息。(译者注：从逻辑概念"文件到物理上的"数据块"映射。)

```
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 结构描述了一个 System V IPC 对象的存取权限。.

```
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 结构用来描述系统的中断句柄。

```
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

每个 Linux 可以理解的二进制文件格式对应一个 linux_binfmt 数据结构。

```
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(或叫做页面) 用来存放每个物理内存页面的信息。

```
typedef struct page {
  /* these must be first (free area handling) */
  struct page        *next;
  struct page        *prev;
```

```
  struct inode        *inode;
  unsigned long       offset;
  struct page         *next_hash;
  atomic_t            count;
  unsigned            flags;     /* atomic flags, some possibly
                                    updated asynchronously */
  unsigned            dirty:16,
                      age:8;
  struct wait_queue   *wait;
  struct page         *prev_hash;
  struct buffer_head *buffers;
  unsigned long       swap_unlock_entry;
  unsigned long       map_nr;    /* page->map_nr == page - mem_map */
} mem_map_t;
```

mm_struct

mm_struct 用来描述一个任务活一个进程的虚拟内存空间。

```
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

系统中的每个 PCI 总线对应一个 pci_bus 结构。

```
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

系统中的每个 PCI 设备，包括 PCI-PCI 和 PCI-ISA 桥设备都分别对应于一个 pci_dev 结构。

```
/*
 * 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 结构用来向系统中的块设备发出请求。 这些请求是关于读或写缓冲区中的数据块。

```
struct request {
    volatile int rq_status;
#define RQ_INACTIVE            (-1)
#define RQ_ACTIVE              1
#define RQ_SCSI_BUSY           0xffff
#define RQ_SCSI_DONE           0xfffe
#define RQ_SCSI_DISCONNECTING  0xffe0

    kdev_t rq_dev;
    int cmd;        /* READ or WRITE */
    int errors;
    unsigned long sector;
    unsigned long nr_sectors;
    unsigned long current_nr_sectors;
    char * buffer;
    struct semaphore * sem;
    struct buffer_head * bh;
    struct buffer_head * bhtail;
    struct request * next;
};
```

rtable

每个 rtable 结构含有对应一个 IP 主机的路由信息。rtable 结构在 IP 路由缓冲中被使用。

```
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

Semaphores 被用来保护临界数据和临界区代码。

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

sk_buff

sk_buff 结构被用来当数据在网络协议之间移动时描述网络数据。

```
struct sk_buff
{
  struct sk_buff      *next;       /* Next buffer in list*/
  struct sk_buff      *prev;       /* Previous buffer in list*/
  struct sk_buff_head *list;       /* List we are on */
  int                 magic_debug_cookie;
  struct sk_buff      *link3;      /* Link for IP protocol level buffer
chains */
  struct sock         *sk;         /* Socket we are owned by          */
  unsigned long       when;        /* used to compute rtt's           */
  struct timeval      stamp;       /* Time we arrived                 */
  struct device       *dev;        /* Device we arrived on/are leaving
by */
  union
```

```
{
    struct tcphdr   *th;
    struct ethhdr   *eth;
    struct iphdr    *iph;
    struct udphdr   *uh;
    unsigned char   *raw;
    /* for passing file handles in a unix domain socket */
    void            *filp;
} h;

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

struct iphdr        *ip_hdr;     /* For IPPROTO_RAW
*/
unsigned long       len;         /* Length of actual data
*/
unsigned long       csum;        /* Checksum
*/
__u32               saddr;       /* IP source address
*/
__u32               daddr;       /* IP target address
*/
__u32               raddr;       /* IP next hop address
*/
__u32               seq;         /* TCP sequence number
*/
__u32               end_seq;     /* seq [+ fin] [+ syn] + datalen
*/
__u32               ack_seq;     /* TCP ack sequence number
*/
unsigned char       proto_priv[16];
volatile char       acked,       /* Are we acked ?
*/
```

```
                      used,        /* Are we in use ?
*/
                      free,        /* How to free this buffer
*/
                      arp;         /* Has IP/ARP resolution finished
*/
  unsigned char       tries,       /* Times tried
*/
                      lock,        /* Are we locked ?
*/
                      localroute,  /* Local routing asserted for this
frame */
                      pkt_type,    /* Packet class
*/
                      pkt_bridged, /* Tracker for bridging
*/
                      ip_summed;   /* Driver fed us an IP checksum
*/
#define PACKET_HOST         0        /* To us
*/
#define PACKET_BROADCAST    1        /* To all
*/
#define PACKET_MULTICAST    2        /* To group
*/
#define PACKET_OTHERHOST    3        /* To someone else
*/
  unsigned short      users;       /* User count - see datagram.c,tcp.c
*/
  unsigned short      protocol;    /* Packet protocol from driver.
*/
  unsigned int        truesize;    /* Buffer size
*/
  atomic_t            count;       /* reference count
*/
  struct sk_buff      *data_skb;   /* Link to the actual data skb
*/
  unsigned char       *head;       /* Head of buffer
*/
```

```
  unsigned char       *data;       /* Data head pointer
*/
  unsigned char       *tail;       /* Tail pointer
*/
  unsigned char       *end;        /* End pointer
*/
  void                (*destructor)(struct sk_buff *); /* Destruct
function */
  __u16               redirport;   /* Redirect port
*/
};
```

sock

每个 sock 数据结构维护一个关于 BSD socket 的协议信息。例如，对一个 INET 类型的 socket，此数据结构将含有所有 TCP/IP 和 UDP/IP 的相关信息。

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

    struct options          *opt;
    atomic_t                wmem_alloc;
    atomic_t                rmem_alloc;
    unsigned long           allocation;       /* Allocation mode */
    __u32                   write_seq;
    __u32                   sent_seq;
    __u32                   acked_seq;
    __u32                   copied_seq;
    __u32                   rcv_ack_seq;
    unsigned short          rcv_ack_cnt;      /* count of same ack */
    __u32                   window_seq;
    __u32                   fin_seq;
    __u32                   urg_seq;
    __u32                   urg_data;
    __u32                   syn_seq;
    int                     users;            /* user count */
  /*
```

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

 struct sock              *next;
 struct sock              **pprev;
 struct sock              *bind_next;
 struct sock              **bind_pprev;
 struct sock              *pair;
 int                      hashent;
 struct sock              *prev;
 struct sk_buff           *volatile send_head;
 struct sk_buff           *volatile send_next;
 struct sk_buff           *volatile send_tail;
 struct sk_buff_head      back_log;
 struct sk_buff           *partial;
 struct timer_list        partial_timer;
 long                     retransmits;
 struct sk_buff_head      write_queue,
                          receive_queue;
```

```
    struct proto            *prot;
    struct wait_queue       **sleep;
    __u32                   daddr;
    __u32                   saddr;          /* Sending source */
    __u32                   rcv_saddr;      /* Bound address */
    unsigned short          max_unacked;
    unsigned short          window;
    __u32                   lastwin_seq;    /* sequence number when
we last
                                               updated the window we
offer */
    __u32                   high_seq;       /* sequence number when
we did
                                               current fast
retransmit */
    volatile unsigned long  ato;            /* ack timeout */
    volatile unsigned long  lrcvtime;       /* jiffies at last data
rcv */
    volatile unsigned long  idletime;       /* jiffies at last rcv */
    unsigned int            bytes_rcv;
/*
 *    mss is min(mtu, max_window)
 */
    unsigned short          mtu;            /* mss negotiated in the
syn's */
    volatile unsigned short mss;            /* current eff. mss - can
change */
    volatile unsigned short user_mss;       /* mss requested by user
in ioctl */
    volatile unsigned short max_window;
    unsigned long           window_clamp;
    unsigned int            ssthresh;
    unsigned short          num;
    volatile unsigned short cong_window;
    volatile unsigned short cong_count;
    volatile unsigned short packets_out;
    volatile unsigned short shutdown;
    volatile unsigned long  rtt;
```

```
    volatile unsigned long  mdev;
    volatile unsigned long  rto;

    volatile unsigned short backoff;
    int                     err, err_soft;    /* Soft holds errors that
don't
                                                 cause failure but are
the cause
                                                 of a persistent
failure not
                                                 just 'timed out' */
    unsigned char           protocol;
    volatile unsigned char  state;
    unsigned char           ack_backlog;
    unsigned char           max_ack_backlog;
    unsigned char           priority;
    unsigned char           debug;
    int                     rcvbuf;
    int                     sndbuf;
    unsigned short          type;
    unsigned char           localroute;       /* Route locally only */
/*
 *    This is where all the private (optional) areas that don't
 *    overlap will eventually live.
 */
    union
    {
          struct unix_opt   af_unix;
#if defined(CONFIG_ATALK) || defined(CONFIG_ATALK_MODULE)
        struct atalk_sock   af_at;
#endif
#if defined(CONFIG_IPX) || defined(CONFIG_IPX_MODULE)
        struct ipx_opt      af_ipx;
#endif
#ifdef CONFIG_INET
        struct inet_packet_opt  af_packet;
#ifdef CONFIG_NUTCP
        struct tcp_opt      af_tcp;
```

```
#endif
#endif
    } protinfo;
/*
 *    IP 'private area'
 */
    int                     ip_ttl;           /* TTL setting */
    int                     ip_tos;           /* TOS */
    struct tcphdr           dummy_th;
    struct timer_list       keepalive_timer;  /* TCP keepalive hack */
    struct timer_list       retransmit_timer; /* TCP retransmit timer
*/
    struct timer_list       delack_timer;     /* TCP delayed ack timer
*/
    int                     ip_xmit_timeout;  /* Why the timeout is
running */
    struct rtable           *ip_route_cache;  /* Cached output route */
    unsigned char           ip_hdrincl;       /* Include headers ? */
#ifdef CONFIG_IP_MULTICAST
    int                     ip_mc_ttl;        /* Multicasting TTL */
    int                     ip_mc_loop;       /* Loopback */
    char                    ip_mc_name[MAX_ADDR_LEN]; /* Multicast
device name */
    struct ip_mc_socklist   *ip_mc_list;      /* Group array */
#endif

/*
 *    This part is used for the timeout functions (timer.c).
 */
    int                      timeout;         /* What are we waiting
for?
*/
    struct timer_list        timer;           /* This is the
TIME_WAIT/receive

                                               * timer when we are
doing
IP
```

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

};
```

socket

每个 socket 结构包含一个 BSD socket 信息。这个结构不是独立存在的，而是作为一个 VFS 数据结构的一个部份。

```
struct socket {
  short                type;          /* SOCK_STREAM, ...             */

  socket_state         state;
  long                 flags;
  struct proto_ops     *ops;          /* protocols do most everything */

  void                 *data;         /* protocol data                */

  struct socket        *conn;         /* server socket connected to   */

  struct socket        *iconn;        /* incomplete client conn.s     */

  struct socket        *next;
  struct wait_queue    **wait;        /* ptr to place to wait on      */

  struct inode         *inode;
  struct fasync_struct *fasync_list; /* Asynchronous wake up list    */
```

```
  struct file           *file;        /* File back pointer for gc     */
};
```

task_struct

每个 task_struct 数据结构描述系统中的进程或任务。（译者注：请注意结构中已考虑 SMP 的结构）

```
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 结构用来实现对进程的定时器的操作。

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

tq_struct

每个任务队列 task queue (tq_struct) 含有已在排队的任务信息。一般来说，这些任务是设备驱动程序所需的，但不是必须"立刻"被完成。（译者注：千万注意，

这个队列不是进程调度队列，而是一个函数(如设备驱动程序调用的一些核心函数)队列。结构中含有函数指针和被传递的参数。请回忆递归时栈结构中的内容。不同的是队列是 FIFO，而栈是 LIFO)

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

vm_area_struct

每个 vm_area_struct 结构含有一个进程虚拟内存的每个部份的描述。

```
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 */
};
```

第 15 章 Alpha AXP 处理器

Alpha AXP 处理器是一种 64 位的 RISC 处理器，它在设计时以提高运行速度作为主要目标。在 AXP 中，所有的寄存器都是 64 位的，其中包括 32 个整数寄存器与 32 个浮点数寄存器。第 31 号整数寄存器与 31 号浮点数寄存器只在空操作（null）中使用，即从这两个寄存器中读时将得到 0，而向其中写不会产生任何结果。另外，AXP 处理指令也是 32 位的，内存操作只有读和写两种。

在 AXP 中，读与写之外的操作不能直接在内存中进行，而只能通过寄存器。比如，若想实现一个计数器，就得先将内存中保存的计数器值读到寄存器中，修改后再写回内存中。寄存器与内存还是不同指令之间交互的媒介，即在 AXP 中只能先将一条指令的执行结果写入寄存器或内存，然后由另一寄存器读入这一值。AXP 的另一个值得注意的特徵就是在该系统中有专门用来产生标志的指令。比如进行两个寄存器内容的比较时，比较的结果并不是存储在某个状态寄存器中，而是存在另外一个独立的寄存器中。这种看似奇怪的设计使得指令操作不再依赖于状态寄存器，从而有利于提高 CPU 内部指令的并行性，因为在这种体系结构中，不相关的寄存器不需要因为要使用状态寄存器而进行等待。上文中所提到的不能对内存直接进行操作也同样有利于多指令间的并行操作。

AXP 中使用了一组例程，我们称之为特权系统库代码（privileged architecture library code，PALcode）。PALcode 用于操作系统、AXP 中 CPU 以及系统硬件的设计。这些例程提供了一些操作系统原语用于进行中断处理、内存管理等。例程可由硬件或 CALL_PAL 指令调用。PALcode 是用标准的 Alpha AXP 汇编语言书写的，并在功能上进行了扩展，从而能够对低层的硬件（比如处理器内的寄存器）进行操作。PALcode 在 PALmode 模式下运行，它一种特权级模式，能暂停其他事件的运行从而能及时地完成 PALcode 对硬件的控制操作。

第 16 其他Web和下载站点

请参阅:

kernel.org - The Linux Kernel Archives, the official kernel repository

kernelnewbies.org - information for (beginning) Linux kernel developers

'Linux Kernel in a Nutshell' - free (Creative Commons Attribution-ShareAlike 2.5 license) downloadable book on the Linux kernel

Man pages documentation for Linux kernel

第 17 GNU

GNU General Public License

第 18 章 术语表

Argument

参数

函数和例程中可以带入参数进行处理。

ARP

地址转换协议。被用来将 IP 地址转换成物理硬件地址，如网卡地址。ARP 是 TCP/IP 协议族中一个非常重要的协议。

Ascii

Ascii 代表着 American Standard Code for Information Interchange. 字母表中的每个字母代表一个 8 位的编码。Ascii 被用来存储″可写‶的字符。

Bit

值域为 0 或 1 的一个二进制数据位。

Bottom Half Handler

核心里在队列中的任务的句柄或指针。

Byte

字节，8 位数据。

C

一种高级编程语言，Linux 基本上是用 C 编写的。

CPU

Central Processing Unit(中央处理单元)。

Data Structure

数据结构。

Device Driver

设备驱动程序。用来控制一个特定设备类的软件。例如，NCR 810 设备驱动程序控制

NCR 810 SCSI 设备。」

DMA

Direct Memory Access(直接内存存取)

ELF

Executable and Linkable Format(可执行与可连接格式).

EIDE

扩展 IDE.

Executable image

可执行映象。一个含有指令和数据的文件。可以被调进虚拟内存而执行。

Function

函数

IDE

Integrated Disk Electronics.

Image

参阅可执行映象。

IP

Internet Protocol(网际协议).

IPC

Interprocess Communiction(进程间通讯)

Interface

接口。接口是一个抽象的概念。实现中，通常指一些函数或例程接口。

IRQ

Interrupt Request Queue(中断申请队列).

ISA

Industry Standard Architecture. This is a standard, although now rather

dated, data bus interface for system components such as floppy disk drivers.

Kernel Module
一个可以动态地被装载的核心部份，如文件系统和设备驱动程序。

Kilobyte
1024 字节。

Megabyte
一兆字节或 1024K 字节。

Microprocessor
微处理器。

Module
模块。一个含有指令的文件。

Object file
目标文件。或*.o 文件。指一个含有指令和数据的文件。但这个文件尚未与其所需要的其他目标文件或库相连接以形成一个可执行文件。

Page
物理内存被分成许多同样大小的页面。是虚拟内存管理调度的最小单位。

Pointer
指针

Process
进程。一个正在执行的程序。

Processor
处理器的简称。

PCI

Peripheral Component Interconnect. 一个外设总线。

Peripheral

外围设备

Program

程序。

Protocol

协议。通常指两个实体间"对话"的一种事先格式约定。

Register

寄存器

Routine

例程。与函数类似，除了不返回值。

SCSI

Small Computer Systems Interface(小型计算机接口).

Shell

command shell。Linux 缺省用的是 bash shell.

SMP

Symmetrical multiprocessing. Systems(对称多处理系统)。

Socket

一个 socket 代表着一个网络连接。Linux 支持 BSD Socket 接口。.

Software

软件

System V

Unix 的一个版本，发布于 1983。这个版本中，引进了著名的士 System V IPC 机制。

TCP

Transmission Control Protocol(传输控制协议).

Task Queue

任务队列

UDP

User Datagram Protocol(用户数据报协议).

Virtual memory

虚拟内存