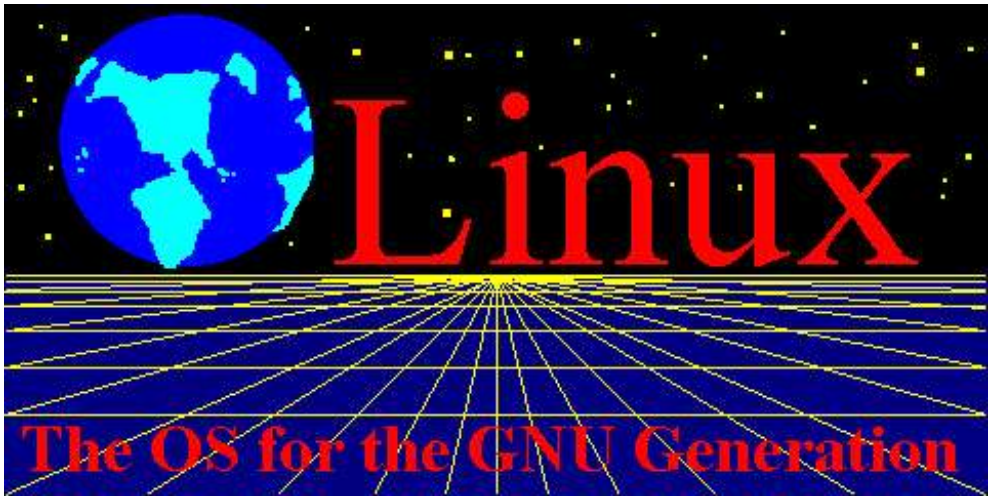
Linux Kernel 中文版



原著：David A Rusling

翻译：Banyan(hejin99@hotmail.com)

Fifa(lifei@263.net)

整理：wujiboy(wujiboy@163.com)

目 录

本书是为那些想了解 Linux 内核工作原理的 Linux 狂热爱好者而写。 它并非一本内部手册。主要描叙了 Linux 设计的原理与机制；以及 Linux 内核怎样工作及其原因。

Linux 还在不断改进；本书基于目前比较流行且性能稳定的 2.0.33 核心。

Legal Notice

前言

Linux 是互连网上的独特现象。虽然它是由学生的业余爱好发展而来，但是现在它已经成为最为流行的免费操作系统。对很多人来说，Linux 是一个谜。免费的东西怎么会变得如此有价值？在个由少数软件公司统治的世界，由一帮 HACKER 们编写的东西是怎样与那些公司的产品竞争的？ 这些软件是如何分发给分布在世界各个角落，希望得到稳定产品的人们的？事实上 Linux 的确稳定而富有竞争力。许多大学与研究机构都使用 Linux 完成他们的日常计算任务。人们在家用 PC 上使用 Linux,许多公司也在使用它--尽管他们并不总是乐意承认这点。Linux 主要用来浏览 WEB，管理 WEB 站点，撰写与发送 EMAIL，以及玩游戏。Linux 绝对不是玩具而是具有专业水平的操作系统，它的爱好者遍及世界。

Linux 的源头要追溯到最古老的 UNIX。1969 年，Bell 实验室的 Ken Thompson 开始利用一台闲置的 PDP-7 计算机开发了一种多用户，多任务操作系统。很快，Dennis Richie 加入了这个项目，在他们共同努力下诞生了最早的 UNIX。Richie 受一个更早的项目——MULTICS 的启发，将此操作系统命名为 Unix。早期 UNIX 是用汇编语言编写的，但其第三个版本用一种崭新的编程语言 C 重新设计了。C 是 Richie 设计出来并用于编写操作系统的程序语言。通过这次重新编写，Unix 得以移植到更为强大的 DEC PDP-11/45 与 11/70 计算机上运行。后来发生的一切，正如他们所说，已经成为历史。Unix 从实验室走出来并成为了操作系统的主流，现在几乎每个主要的计算机厂商都有其自有版本的 Unix.

Linux 起源于一个学生的简单需求。Linus Torvalds,Linux 的作者与主要维护者，在其上大学时所买得起的唯一软件是 Minix. Minix 是一个类似 Unix，被广泛用来辅助教学的简单操作系统。Linus 对 Minix 不是很满意，于是决定自己编写软件。他以学生时代熟悉的 Unix 作为原型， 在一台 Intel 386 PC 上开始了他的工作。他的进展很快，受工作成绩的鼓舞，他将这项成果通过互连网与其他同学共享，主要用于学术领域。有人看到了这个软件并开始分发。每当出现新问题时，有人会立刻找到解决办法并加入其中，很快的， Linux 成为了一个操作系统。值得注意的是 Linux 并没有包括 Unix 源码。它是按照公开的 POSIX 标准重新编写的。Linux 大量使用了由麻省剑桥免费软件基金的 GNU 软件，同时 Linux 自身也是用它们构造而成。

许多人将 Linux 视作简单工具并将其放入 CDROM 中来分发。很多 Linux 使用者使用它来编写应用程序或者运行别人编写的应用程序。这些人热切的阅读 HOWTO 手册，当系统的一部分被正确的设置时，他们总是激动不已，失败时则沮丧气馁。只有少部分人敢于编写设备驱动程序并将核心的补丁提供给 Linus Torvalds，Linus Torvalds 从每个志愿者那里接收补充代码与对核心的修改代码。

这种情形听起来象非常混乱，但 Linus 进行了非常严格的质量控制并由他负责将所有的新代码加入核心。只有少部分人对 Linux 核心贡献了源代码。 大多数 Linux 的使用者并不关心系统是如何工作，或者如何组合在一起的。这种情况令人惋惜，因为阅读 Linux 源代码提供了一个学习操作系统的绝好机会。这不仅仅因为它写得好，还因为它的源码是可以免费得到的。因为虽然作者们对其软件保留版权，但是在免费软件基金的 GNU 公开授权下源代码是可以自由分发的。第一眼看去，源码是非常复杂的。但是通过进一步观察你

可以发现源码目录中包含有 Kernel,mm 以及 net 的目录， 不过要想知道这些目录中包含了那些代码以及代码是如何工作的就需要对 Linux 的总体结构与目标有较深入的理解。简而言之，这也是本书所希望达到的目标，为读者提供一个 Linux 如何工作清晰的印象。当你将文件从一个目录拷到另一个目录或者阅读电子邮件时，不妨在脑海中勾勒一下系统中正在发生什么事情，我还清楚的记得当我感到第一次认识到操作系统真的在工作时的兴奋。这种兴奋正是我想将它带给本书的读者的。

我第一次接触 Linux 在 1994 年下半年当我拜访 Jim Paradis 时，当时他正在致力于将 Linux 移植到 Alpha AXP 处理器系统上。从 1984 年开始，我曾经在 DEC 公司任职，主要工作是网络与通讯。1992 年我开始为新成立的 Digital Semiconductor 分部工作。此分部的任务是全面进入商用芯片市场并销售芯片，特别是 Alpha AXP 系列处理器以及 DEC 以外的 Alpha AXP 系统板。当首次听到 Linux 时我便立刻意识到了这是一个有趣的机会。Jim 的狂热是鼓惑人心的，我也开始帮他一起工作。在工作中，我越来越喜欢这个操作系统及创造它的工程师团体。

Alpha AXP 仅仅是 Linux 可以运行的多种平台中的一个。大多数 Linux 核心工作在基于 Intel 处理器 的系统上，但非 Intel 系统的 Linux 用户也越来越多。它们是 Alpha AXP, ARM, MIPS, Sparc 与 Power PC。 虽然我可以根据上叙任何一种平台来编写本书的内容，但是我的技术知识与背景让我主要根据 Alpha AXP 处理器和 ARM 处理器来编写。这是本书有时使用非 Intel 硬件来描叙一些重要观点。值得注意的是，不管运行在哪种平台上，95%的 Linux 核心代码都是相同的。同样，本书 95%的内容是关于 Linux 内核的机器无关部分的讨论。

本书对读者的知识与经验没有任何要求。我相信对于某一事物的兴趣是鼓励自学的必要因素。不过对于计算机，或者 PC 和 C 程序语言的了解将有助于读者从有关材料中获益。

本书的组织

本书并不是特意一本 Linux 的内部手册。相反它是对操作系统的介绍，同时以 Linux 作为示例。书中每一章遵循“从共性到特性”的原则。它们将首先给出核心子系统的概叙，然后进行尽可能的详细描叙。 我不会用 routine_X()调用 routine_Y()来增加 bar 数据结构中 foo 域的值这种方式来描叙核心算法。 你自己可以通过阅读代码发现它。每当需要理解一段代码时，我总是将其数据结构画出来。这样我发现了许多相关的核心数据结构以及它们之间的关系。 每一章都是非常独立的，就象 Linux 核心子系统一样。当然有时它们还是有联系的，比如说，如果你没有理解虚拟内存工作原理就无法描叙进程。 硬件基本概念一章对现代 PC 做了简要介绍。操作系统必须与硬件系统紧密结合在一起协同工作。操作系统需要一些只能够由硬件提供的服务。为了全面理解 Linux，你必须了解有关硬件的基础知识。 软件基本概念一章介绍了软件基本原理与 C 程序语言。讨论了建立 Linux 这样的操作系统的工具并且给出了操作系统的目标与功能的概叙。 内存管理这章描叙了 Linux 如何处理物理内存以及虚拟存储技术。 进程管理描叙了进程的概念以及 Linux 核心是如何创建、管理与删除系统中的进程。 进程间及进程与核心间通讯以协调它们的活动。Linux 支持大量进程间通讯（IPC）机制。信号与管道是 其中的两种，Linux 同时还支持系统 V IPC 机制。这些进程间通讯机制在 IPC 一章中描叙。 外部设备互连（PCI）标准已经成为 PC 上低价位高数传率的总线标准。PCI 一章将描叙 Linux 核心是如何初始化并使用 PCI 总线

及设备的。 中断及中断处理一章将着重于 Linux 核心对中断的处理。虽然处理中断有通用的机制与接口，但某些细节是与硬件及 CPU 体系结构相关的。 Linux 的一个长处是其对现代 PC 的硬件设备强有力的支持。设备驱动程序一章将描叙 Linux 核心是如何控制系统中的物理设备。 文件系统一章描叙了 Linux 核心是如何维护它所支持的文件系统中的文件。同时还描叙了虚拟文件系统（VFS）及 Linux 核心的每种文件系统是如何得到支持。 网络与 Linux 几乎是同义的。在某种意义上 Linux 是 WWW 时代互连网的产物。其开发者通过 Web 来交换信息及代码。网络一章描叙了 Linux 是如何支持 TCP/IP 这些网络协议。 核心机制一章主要讨论能使 Linux 核心其他部分有效工作而由核心所提供的一些通用任务与机制。 动态模块一章描叙 Linux 核心是如何仅在需要时动态加载某些模块，比如文件系统。处理器一章给出了目前 Linux 可以在其上运行的一些处理器的简要介绍。 资源一章则提供了有关 Linux 核心资源的有用信息。

第一章 硬件基础

操作系统必须与基本硬件系统密切协作。它需要那些仅仅能够由硬件提供的服务。为了全面理解 Linux 操作系统，你必须要懂得一些有关硬件的知识。本章将对硬件：现代 PC 做一个简要的介绍。 当 1975 年一月的"Popular Electronics"杂志以 Altair 8080 的图片作为封面时，一场革命开始了。 家用电器爱好者能独立组装出来的 Altair 8080，当时价格仅仅为 397 美圆。这种带有 256 字节内存的 8080 处理器还没有显示器与键盘，以今天的标准来看，它是微不足道的。它的创造者， Ed Roberts，发明了"personal computer"来描叙他的新发明。但现在 PC 这一术语已被用来称呼那些自己就可以携带的计算机。在这个定义上，非常强劲的计算机如 Alpha AXP 也可称为 PC。 狂热的 HACKER 们看到了 Altair 的巨大潜力，于是他们开始为它编写软件和设计硬件。对早期的先驱来说这意味者某种自由；一种从顽固的超级批处理主机中解放出来的自由。滚滚而来的财富让许多着迷于此（一台可以放在厨房餐桌上的计算机）的大学生纷纷退学。许多五花八门的硬件开始出现，软件 HACKER 们忙着为这些新机器编写软件。有意思的是 IBM 首先坚定的进行现代 PC 的设计和制造并于 1982 年推出产品。该产品的构造是：8080 CPU、64K 字节主存、两个软盘驱动器以及 25 行 80 列的彩色 CGA 显示器。虽然以现在观点看那些都不是多么先进的东西但当时销售情况却很好。紧接着，1983 年，带有昂贵的 10MB 硬盘驱动器的 IBM PC-XT 出现了。在 IBM PC 体系结构成为事实上的标准不久之后，大量仿制者如 COMPAQ 公司出现了。由于这种事实标准的存在，多个硬件公司在这一快速增长的市场上进行了激烈竞争。但用户却从低价中获益。许多早期 PC 中的结构特征还保留在现代 PC 系统中。比如 Intel 公司最先进的 Pentium Pro 处理器还保留着 Intel 8086 的寻址模式。当 Linus Torvalds 开始写 Linux 时，他选择了当时最广泛使用同时价格合理的 Intel 80386 PC。 图 1.1 典型的 PC 主板示意图。 从 PC 的外部来看，最引人注目的是机箱，键盘，鼠标以及显示器。在机箱前部有一些按钮，一个微型显示器显示着一些数字，此外还有一个软驱。今天的大多数机器还包含一个 CD ROM，另外，如果想保护你的数据，还可以添加一个磁带机作为备份用。这些设备统称为外部设备。 尽管 CPU 是系统的总管，但是它仅仅是一个智能设备。所有的这些外设控制器都具有某种层度的智能，如 IDE 控制器。在 PC 内部，你可以看到一个包括 CPU 或者微处理器，主存和许多 ISA 或 PCI 外设控制器插槽的主板（图 1.1）。有些控制器，如 IDE 磁盘控制器必须建立在系统板上。

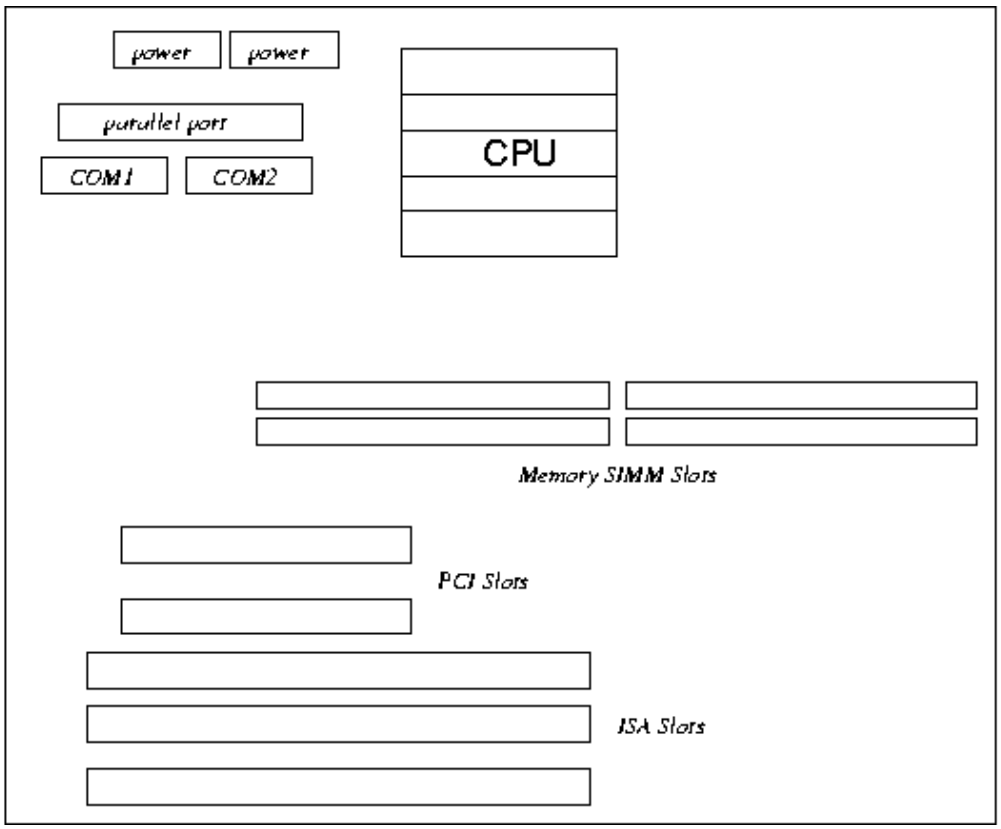


图 1.1: 典型的 PC 主板.

1.1 CPU

CPU，或者微处理器，是计算机系统的核心。微处理器进行计算或者逻辑操作并且管理来自主存的指令并执行它。在计算机的早期时代，微处理器的功能部件使用的是分立元件（外型很大）。 这就是中央处理单元这一名词的由来。现代微处理器将部件结合到小型硅片上的集成电路中。在本书中 CPU 和微处理器及处理器有相同的意义。 微处理器的操作对象是二进制数据；数据由 0 和 1 组成。 1 和 0 对应着电子开关的开路与断路状态。正如十进制的 42 表示有 4 个 10 和一个 2 一样，一个二进制数是一系列表示 2 的次幂的二进制数字组成。二进制 0001 对应十进制的 1，二进制的 0010 对应十进制 的 2，二进制的 0011 表示 3，而 0100 对应 4。十进制 42 的二进制表示为 101010。但是在计算机程序中， 人们常用十进制来表示数而不是直接使用二进制。 在需要使用二进制数时，人们往往使用 16 进制数。如十进制数只能从 0 到 9 一样，16 进制数可以从 0 疏导 15，其中 10 到 15 分别用字母 A、B、C、D、E 及 F 来表示。这样 16 进制的 2A 的十进制表示为 42- 2*16+10=42。在 C 程序语言中，16 进制数的前缀为"0x"；16 进制的 2A 写成 0x2A。 微处理器可以执行如加、乘和除以及象"X 是否比 Y 大"这种逻辑运算。 处理器的执行由外部时钟来监控。这个时钟称为系统时钟，它每隔相同的时间间隔就向 CPU 发送一个脉冲。在每个时钟脉冲上，处理器都会做一些工作。比如，处理器每个时钟脉冲上执行一条指令。处理器的速度一般以系统时钟的速率来描叙。一个 100MHz 的处理器每秒将接收 100,000,000 个时钟滴答。

但是用 CPU 的时钟频率来描叙 CPU 的工作能力是不正确的，因为它们执行的指令不相同。然而，快速的时钟可以在某种程度上代表高性能的 CPU。处理器执行的指令是非常简单的；例如"将内存 X 处的内容读入寄存器 Y"。寄存器是微处理器的内部存储部件，用来存储数据并对数据执行某些指令。有些指令有可能使处理器停止当前的工作而跳转到内存中另外一条指令执行。现代微处理器的紧凑设计使得它有可能每秒执行上百万甚至亿条指令。 指令执行前必须从内存中取出来。指令自身要使用的数据也必须从内存中取出来并放置在适当的地方。 微处理器中寄存器的大小、数量以及类型都取决于微处理器的类型。Intel 80486 处理器和 Alpha AXP 有迥然不同的寄存器，最明显的区别在于 Intel 寄存器为 32 位而 Alpha AXP 为 64 位。一般来说，任何处理器都有许多通用寄存器和少量专用寄存器。许多微处理器有以下几种特定的寄存器。

程序计数器(PC)

此寄存器包含下条指令执行的地址。每当取回一条指令时，PC 的内容将自动增加。

堆栈指针(SP)

微处理器经常需要访问存储临时数据的外部 RAM。堆栈是一种便捷的存放临时数据的方法，处理器提供了特殊指令来将数值压入堆栈然后将其从堆栈中弹出。 堆栈以后进先出(LIFO)的方式工作。换句话说，如果你压入两个值 X 和 Y，然后执行弹栈操作，你将取到 Y 的值。 有些处理器的堆栈从内存顶部向下增长而有些相反。但有的处理器同时支持这两种方式，如 ARM。

处理机状态字(PS)

指令的执行将得到执行结果；比如"寄存器 X 中的内容要大于寄存器 Y 中的内容？"将得到正确或错误作为结果。PS 寄存器包含着这些信息及有关处理器当前状态的其他信息。例如大多数处理器至少有两种执行方式，核心（或管态）与用户方式。PS 寄存器包含表示当前执行方式的信息。

1.2 内存

所有计算机系统都有一个由不同速度与大小的存储器组成的层次结构。最快的的存储器是高速缓存，它被用来暂存主存中的内容。这种存储器速度非常快但非常昂贵，大多数处理器都有少量的片上高速缓存或者将其放在主板上。有些处理器的高速缓存既包含数据也包含指令，但有些将其分成两部分。 Alpha AXP 处理器有两个内部高速缓存，一个用来缓存数据（D-Cache）而另一个用来缓存指令（I- Cache）。而外部高速缓存（B-Cache）将两者混合。这样，相对外部高速缓存存储器，主存的速度非常慢。 高速缓存与主存中的内容必须保持一致。换句话说，对应于地址空间的同一个位置，如果该位置的数据被缓存入高速缓存，则其内容必须和主存中的一致。保证高速缓存一致性的工作由硬件和操作系统共同分担。 这就是在系统中硬件和软件必须紧密协作的原因。

1.3 总线

主板上分立的部件通过称为总线的线路连接在一起。系统总线的功能在逻辑上被划分为三部分： 地址总线、数据总线和控制总线。地址总线为数据传输指明内存位置（地址）。数据总线包含传输的数据。数据总线是双向的；它允许数据读入 CPU 也支持从 CPU 读出

来。控制总线则包含几条表示路由分时和系统的控制信号。当然还有其他一些总线存在，例如 ISA 和 PCI 总线是将外设连接到系统的常用方式。

1.4 控制器与外设

外设是一些物理设备，比如说图象卡或者磁盘，它们受控于位于主板或者主板上插槽中的控制芯片。 IDE 磁盘被 IDE 控制器芯片控制而 SCSI 磁盘由 SCSI 磁盘控制器芯片控制。这些控制器通过各种总线连接到 CPU 上或相互间互连。目前制造的大多数系统使用 PCI 和 ISA 总线来连接主要系统部件。控制器是一些类似 CPU 的处理器，它们可以看做 CPU 的智能帮手。CPU 则是系统的总控。 虽然所有这些控制器互不相同，但是它们的寄存器的功能类似。运行在 CPU 上的软件必须能读出或者写入这些控制寄存器。其中有一个寄存器可能包含指示错误的状态码。另一个则用于控制目的，用来改变控制器的运行模式。在总线上的每个控制器可以被 CPU 所单独寻址，这是软件设备驱动程序能写入寄存器并能控制这些控制器的原因。

1.5 地址空间

系统总线将 CPU 与主存连接在一起并且和连接 CPU 与系统硬件外设的总线隔离开。一般来说，硬件外设存在的主存空间叫 I/O 空间。I/O 空间还可以进一步细分，但这里我们不再深究。CPU 既可以访问系统内存空间又可以访问 I/O 空间内存，而控制器自身只能在 CPU 协助下间接的访问系统内存。从设备的角度来看，比如说软盘控制器，它只能看到在 ISA 总线上的控制寄存器而不是系统内存。典型的 CPU 使用不同指令来访问内存与 I/O 空间。例如，可能有一条指令"将 I/O 地址 0x3F0 的内容读入到寄存器 X"。这正是 CPU 控制系统硬件设备的方式：通过读写 I/O 地址空间上的外设寄存器。在 I/O 空间中通用外设(IDE 控制器、串行口、软盘控制器等等)上的寄存器经过多年的 PC 体系结构发展基本保持不变。I/O 地址空间 0x3f0 是串行口(COM1)的控制寄存器之一。 有时控制器需要直接从系统主存中读写大量数据。例如当用户将数据写入硬盘时。在这种情况 下，直接内存访问(DMA)控制器将用来允许硬件外设直接访问系统主存，不过这将处于 CPU 的严格监控下。

1.6 时钟

所有的操作系统都必须准确的得到当前时间，所以现代 PC 包含一个特殊的外设称为实时时钟(RTC)。它提供了 两种服务：可靠的日期和时间以及精确的时间间隔。RTC 有其自身的电池这样即使 PC 掉电时它照样可以工作，这就是 PC 总是"知道"正确时间和日期的原因。而时间间隔定时器使得操作系统能进行准确的调度工作。

第二章 软件基础

程序是执行某个特定任务的计算机指令集合。程序可以用多种程序语言来编写：从低级计算机语言-汇编语言到高级的、与机器本身无关的语言入 C 程序语言。操作系统是一个

允许用户运行如电子表格或者字处理软件等应用程序的特殊程序。本章将介绍程序设计的基本原则，同时给出操作系统设计目标与功能的概述。

2.1 计算机编程语言

2.1.1 汇编语言

那些 CPU 从主存读取出来执行的指令对人类来说是根本不可理解的。它们是告诉计算机如何准确动作的机器代码。在 Intel 80486 指令中 16 进制数 0x89E5 表示将 ESP 寄存器的内容拷入 EBP 寄存器。为最早的计算机设计的工具之一就是汇编器，它可以将人们可以理解的源文件汇编成机器代码。汇编语言需要显式的操作寄存器和数据，并且与特定处理器相关。比如说 Intel X86 微处理器的汇编语言与 Alpha AXP 微处理器的汇编语言决然不同。以下是一段 Alpha AXP 汇编指令程序：

```
    ldr r16, (r15)      ; Line 1
    ldr r17, 4(r15)     ; Line 2
    beq r16,r17,100     ; Line 3
    str r17, (r15)      ; Line 4
100:                    ; Line 5
```

第一行语句将寄存器 15 所指示的地址中的值加载到寄存器 16 中。接下来将邻接单元内容加载到寄存器 17 中。 第三行语句比较寄存器 16 和寄存器 17 中的值，如果相等则跳转到标号 100 处，否则继续执行第四行语句：将 寄存器 17 的内容存入内存中。如果寄存器中值相等则无须保存。汇编级程序一般冗长并且很难编写，同时还容易出错。 Linux 核心中只有很少一部分是用汇编语言编写，并且这些都是为了提高效率或者是需要兼容不同的 CPU。

2.1.2 C 编程语言和编译器

用汇编语言编写程序是一件困难且耗时的工作。同时还容易出错并且程序不可移植：只能在某一特定处理器 家族上运行。而用 C 语言这样的与具体机器无关的语言就要好得多。C 程序语言允许用它所提供的逻辑算法来 描叙程序同时它提供编译器工具将 C 程序转换成汇编语言并最终产生机器相关代码。好的编译器能产生和汇编语言程序相接近的效率。Linux 内核中大部分用 C 语言来编写，以下是一段 C 语言片段：

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

它所执行的任务和汇编语言代码示例中相同。如果变量 X 的值和变量 Y 的不相同则将 Y 的内容赋予 X。C 代码被 组织成子程序，单独执行某一任务。子程序可以返回由 C 支持的任何数据类型的值。较庞大的程序如 Linux 核心由许多单独的 C 源代码模块组成，每个

模块有其自身的子程序与数据结构。这些 C 源代码模块将相关函数组合起来完成如文件处理等功能。 C 支持许多类型的变量，变量是一个通过符号名称引用的内存位置。在以上的例子中，X 和 Y 都是内存中的位置。程序员并不关心变量放在什么地方，这些工作由连接程序来完成。有些变量包含不同类型的数据，整数和浮点数，以及指针。 指针是那些包含其他数据内存位置或者地址的变量。假设有变量 X，位于内存地址 0x80010000 处。你可以使用指针变量 px 来指向 X，则 px 的值为 0x80010000。 C 语言允许相关变量组合起来形成数据结构，例如：

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

这是一个叫做 my_struct 的结构，它包含两个元素，一个是 32 位的整数 i,另外一个是 8 位的字符 b。

2.1.3 连接程序

连接程序是一个将几个目标模块和库过程连接起来形成单一程序的应用。目标模块是从汇编器或者编译器中产生的机器代码，它包含可执行代码和数据，模块结合在一起形成程序。例如一个模块可能包含程序中所有的数据库函数而另一个主要处理命令行参数。连接程序修改目标模块之间的引用关系，使得在某一模块中引用的数据或者子程序的确存在于其他模块中。Linux 核心是由许多目标模块连接形成的庞大程序。

2.2 操作系统概念

如果没有软件，计算机只不过是一堆发热的电子器件。如果将硬件比做计算机的心脏则软件就是它的灵魂。操作系统是一组系统程序的集合，它提供给用户运行应用软件的功能。操作系统对系统硬件进行抽象，它提供给系统用户一台虚拟的机器。大多数 PC 可以运行一种或者多种操作系统，每个操作系统都有不同的外观。Linux 由许多独立的功能段组成。比如 Linux 内核，如果没有库函数和外壳程序，内核是没有什么用的。 为了理解操作系统到底是什么，思考一下当你敲入一个简单命令时，系统中发生了什么：

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

$符号是由用户登录外壳(这里指 Bash)提供的提示符。它表示正在等待用户敲入一些命令。敲入 ls 命令，首先键盘驱动程序识别出敲入的内容。然后键盘驱动将它们传递给外壳程序，由外壳程序来负责查找同名的可执行程序(ls)。 如果在/bin/ls 目录中找到了 ls，则调用核心服务将 ls 的可执行映象读入虚拟内存并开始执行。ls 调用核心的文件子系统来寻找那些文件是可用的。文件系统使用缓冲过的文件系统信息，或者调用磁盘设备驱动从磁盘

上读取信息。当然 ls 还可能引起网络驱动程序和远程机器来交换信息以找出关于系统要访问的远程文件系统信息(文件系统可以通过网络文件系统或者 NFS 进行远程安装)。当得到这些信息后，ls 将这些信息通过调用视频驱动写到显示器屏幕上。 以上这些听起来十分复杂。这个非常简单命令的处理过程告诉我们操作系统是一组协同工作的函数的集合，它们给所有的用户对系统有一致的印象。

2.2.1 内存管理

由于资源的有限，比如内存，操作系统处理事务的过程看起来十分冗长。操作系统的一个基本功能就是使一个只有少量物理内存的系统工作起来象有多得多的内存一样。这个大内存叫为虚拟内存。其思想就是欺骗系统中运行的软件，让它们认为有大量内存可用。系统将内存划分成易于处理的页面，在系统运行时将这些页面交换到硬盘上去。 由于有另外一个技巧:多处理的存在，这些软件更加感觉不到系统中真实内存的大小。

2.2.2 进程

进程可以认为是处于执行状态的程序,每个进程有一个特定的程序实体。观察以下 Linux 系统中的进程，你会发现有比你想象的要多得多的进程存在。比如，在我的系统中敲入 ps 命令，将得到以下结果：

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

如果系统有许多个 CPU，则每个进程可以运行在不同的 CPU 上。不幸的是，大多数系统中只有一个 CPU。这样 操作系统将轮流运行几个程序以产生它们在同时运行的假象。这种方式叫时间片轮转。同时这种方法还骗过了进程使它们都认为只有自己在运行。进程之间被隔离开，以便某个进程崩溃或者误操作不会影响到别的进程。操作系统通过为每个进程提供分立的地址空间来作到这一点。

2.2.3 设备驱动

设备驱动组成了 Linux 核心的主要部分。象操作系统的其他部分一样，它们运行在高权限环境中且一旦出错 将引起灾难性后果。设备驱动控制操作系统和硬件设备之间的相互操作。例如当文件系统通过使用通用块设备接口来对 IDE 磁盘写入数据块。设备驱动负责处理所有设备相关细节。设备驱动与特定的控制器芯片有关，如果系统中有一个 NCR810 SCSI 控制卡则需要有 NCR810 SCSI 的驱动程序。

2.2.4 文件系统

Linux 和 Unix 一样，系统中的独立文件系统不是通过设备标志符来访问，而是通过表示文件系统的层次树结构来访问。当 Linux 添加一个新的文件系统到系统中时，会将它 mount 到一个目录下，比如说/mnt/cdrom。 Linux 的一个重要特征就是支持多种文件系统。这使得它非常灵活并且可与其他操作系统并存。Linux 中最常用的文件系统是 EXT2 文件系统，它在大多数 Linux 分发版本中都得到了支持。 文件系统提供给用户一个关于系统的硬盘上文件和目录的总体映象，而不管文件的类型和底层物理设备的特性。 Linux 透明地支持多种文件系统并将当前安装的所有文件和文件系统集成到虚拟文件系统中去。所以，用户和进程一般都不知道某个文件位于哪种文件系统中，他们只是使用它。 块设备驱动将物理块设备类型（例如 IDE 和 SCSI）和文件系统中的差别隐藏起来，物理设备只是数据块的线性存储集合。设备的不同导致块大小的不同，从软盘设备的 512 字节到 IDE 磁盘的 1024 字节。这些都隐藏了起来，对系统用户来说这都是不可见的。不管设备类型如何，EXT2 文件系统看起来总是一样。

2.3 核心数据结构

操作系统可能包含许多关于系统当前状态的信息。当系统发生变化时，这些数据结构必须做相应的改变以反映这些情况。例如，当用户登录进系统时将产生一个新的进程。核心必须创建表示新进程的数据结构，同时 将它和系统中其他进程的数据结构连接在一起。大多数数据结构存在于物理内存中并只能由核心或者其子系统来访问。数据结构包括数据和指针；还有其他数据结构的地址或者子程序的地址。它们混在一起让 Linux 核心数据结构看上去非常混乱。尽管可能被几个核心子系统同时用到，每个数据结构都有其专门的用途。理解 Linux 核心的关键是理解它的数据结构以及 Linux 核心中操纵这些数据结构的各种函数。本书把 Linux 核心的 描叙重点放在数据结构上，主要讨论每个核心子系统的算法，完成任务的途径以及对核心数据结构的使用。

2.3.1 连接列表

Linux 使用的许多软件工程的技术来连接它的数据结构。在许多场合下，它使用 linked 或者 chained 数据结构。 每个数据结构描叙某一事物，比如某个进程或网络设备，核心必须能够访问到所有这些结构。在链表结构中，个根节点指针包含第一个结构的地址，而在每个结构中又包含表中下一个结构的指针。表的最后一项必须是 0 或者 NULL，以表明这

是表的尾部。在双向链表中，每个结构包含着指向表中前一结构和后一结构的指针。使用双向链表的好处在于更容易在表的中部添加与删除节点，但需要更多的内存操作。这是一种典型的操作系统开销与 CPU 循环之间的折中。

2.3.2 散列表

链表用来连接数据结构比较方便，但链表的操作效率不高。如果要搜寻某个特定内容，我们可能不得不遍历整个链表。Linux 使用另外一种技术:散列表来提高效率。散列表是指针的数组或向量，指向内存中连续的相邻数据集合。散列表中每个指针元素指向一个独立链表。如果你使用数据结构来描叙村子里的人，则你可以使用年龄作为索引。为了找到某个人的数据，可以在人口散列表中使用年龄作为索引，找到包含此人特定数据的数据结构。但是在村子里有很多人的年龄相同，这样散列表指针变成了一个指向具有相同年龄的人数据链表的指针。搜索这个小链表的速度显然要比搜索整个数据链表快得多。 由于散列表加快了对数据结构的访问速度，Linux 经常使用它来实现 Caches。Caches 是保存经常访问的信息的子集。经常被核心使用的数据结构将被放入 Cache 中保存。Caches 的缺点是比使用和维护单一链表和散列表更复杂。寻找某个数据结构时，如果在 Cache 中能够找到（这种情况称为 cache 命中），这的确很不错。但是如果没有找到，则必须找出它，并且添加到 Cache 中去。如果 Cache 空间已经用完则 Linux 必须决定哪一个结构将从其中抛弃，但是有可能这个要抛弃的数据就是 Linux 下次要使用的数据。

2.3.3 抽象接口

Linux 核心常将其接口抽象出来。接口指一组以特定方式执行的子程序和数据结构的集合。例如，所有的网络设备驱动必须提供对某些特定数据结构进行操作的子程序。通用代码可能会使用底层的某些代码。例如网络层代码是通用的，它得到遵循标准接口的特定设备相关代码的支持。 通常在系统启动时，底层接口向更高层接口注册(Register)自身。这些注册操作包括向链表中加入结构节点。例如，构造进核心的每个文件系统在系统启动时将其自身向核心注册。文件/proc/filesysems 中可以看到已经向核心注册过的文件系统。注册数据结构通常包括指向函数的指针，以文件系统注册为例，它向 Linux 核心注册时必须将那些 mount 文件系统连接时使用的一些相关函数的地址传入。

第三章 存储管理

存储管理子系统时操作系统中最重要的组成部分之一。在早期计算时代，由于人们所需要的内存数目远远大于物理内存，人们设计出了各种各样的策略来解决此问题，其中最成功的是虚拟内存技术。它使得系统中为有限物理内存竞争的进程所需内存空间得到满足。

虚拟内存技术不仅仅可让我们可以使用更多的内存，它还提供了以下功能：

巨大的寻址空间

操作系统让系统看上去有比实际内存大得多的内存空间。虚拟内存可以是系统中实际物理空间的许多倍。每个进程运行在其独立的虚拟地址空间中。这些虚拟空间相互之

间都完全隔离开来，所以进程间不会互相影响。同时，硬件虚拟内存机构可以将内存的某些区域设置成不可写。这样可以保护代码与数据不会受恶意程序的干扰。

内存映射

内存映射技术可以将映象文件和数据文件直接映射到进程的地址空间。在内存映射中，文件的内容被直接连接到进程虚拟地址空间上。

公平的物理内存分配

内存管理子系统允许系统中每个运行的进程公平地共享系统中的物理内存。

共享虚拟内存

尽管虚拟内存允许进程有其独立的虚拟地址空间，但有时也需要在进程之间共享内存。例如有可能系统中有几个进程同时运行 BASH 命令外壳程序。为了避免在每个进程的虚拟内存空间内都存在 BASH 程序的拷贝，较好的解决办法是系统物理内存中只存在一份 BASH 的拷贝并在多个进程间共享。动态库则是另外一种进程间共享执行代码的方式。共享内存可用来作为进程间通讯(IPC)的手段，多个进程通过共享内存来交换信息。 Linux 支持 SYSTEM V 的共享内存 IPC 机制。

3.1 虚拟内存的抽象模型

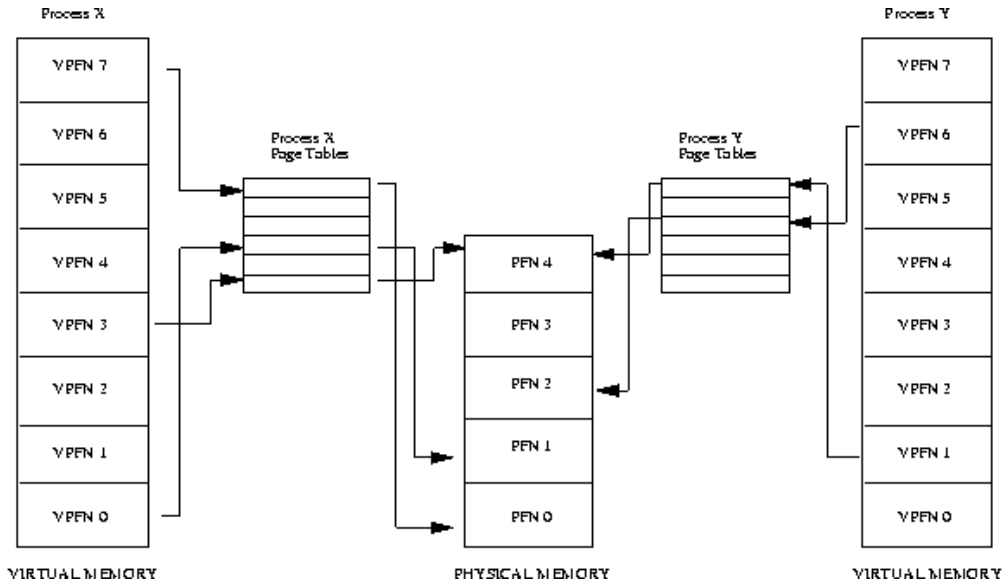


图 3.1 虚拟地址到物理地址映射的抽象模型

在讨论 Linux 是如何具体实现对虚拟内存的支持前，有必要看一下更简单的抽象模型。

在处理器执行程序时需要将其从内存中读出再进行指令解码。在指令解码之前它必须向内存中某个位置取出或者存入某个值。然后执行此指令并指向程序中下一条指令。在此过程中处理器必须频繁访问内存，要么取指取数，要么存储数据。

虚拟内存系统中的所有地址都是虚拟地址而不是物理地址。通过操作系统所维护的一系列表格由处理器实现由虚拟地址到物理地址的转换。

为了使转换更加简单，虚拟内存与物理内存都以页面来组织。不同系统中页面的大小可以相同，也可以不同，这样将带来管理的不便。Alpha AXP 处理器上运行的 Linux 页面大小为 8KB，而 Intel X86 系统上使用 4KB 页面。每个页面通过一个叫页面框号的数字来

标示(PFN) 。

页面模式下的虚拟地址由两部分构成：页面框号和页面内偏移值。如果页面大小为4KB，则虚拟地址的 11：0 位表示虚拟地址偏移值，12 位以上表示虚拟页面框号。处理器处理虚拟地址时必须完成地址分离工作。在页表的帮助下，它将虚拟页面框号转换成物理页面框号，然后访问物理页面中相应偏移处。

图 3.1 给出了两个进程 X 和 Y 的虚拟地址空间，它们拥有各自的页表。这些页表将各个进程的虚拟页面映射到内存中的物理页面。在图中，进程 X 的虚拟页面框号 0 被映射到了物理页面框号 4。理论上每个页表入口应包含以下内容：
有效标记，表示此页表入口是有效的
页表入口描叙的物理页面框号
访问控制信息。用来描叙此页可以进行哪些操作，是否可写？是否包含执行代码?
虚拟页面框号是为页表中的偏移。虚拟页面框号 5 对应表中的第 6 个单元（0 是第一个）。

为了将虚拟地址转换为物理地址，处理器首先必须得到虚拟地址页面框号及页内偏移。一般将页面大小设为 2 的次幂。将图 3.1 中的页面大小设为 0x2000 字节（十进制为 8192）并且在进程 Y 的虚拟地址空间中某个地址为 0x2194，则处理器将其转换为虚拟页面框号 1 及页内偏移 0x194。

处理器使用虚拟页面框号为索引来访问处理器页表，检索页表入口。如果在此位置的页表入口有效，则处理器将从此入口中得到物理页面框号。如果此入口无效，则意味着处理器存取的是虚拟内存中一个不存在的区域。在这种情况下，处理器是不能进行地址转换的，它必须将控制传递给操作系统来完成这个工作。

某个进程试图访问处理器无法进行有效地址转换的虚拟地址时，处理器如何将控制传递到操作系统依赖于具体的处理器。通常的做法是：处理器引发一个页面失效错而陷入操作系统核心，这样操作系统将得到有关无效虚拟地址的信息以及发生页面错误的原因。

再以图 3.1 为例，进程 Y 的虚拟页面框号 1 被映射到系统物理页面框号 4，则在物理内存中的起始位置为 0x8000(4 * 0x2000)。加上 0x194 字节偏移则得到最终的物理地址 0x8194。

通过将虚拟地址映射到物理地址，虚拟内存可以以任何顺序映射到系统物理页面。例如，在图 3.1 中，进程 X 的虚拟页面框号 0 被映射到物理页面框号 1 而虚拟页面框号 7 被映射到物理页面框号 0，虽然后者的虚拟页面框号要高于前者。这样虚拟内存技术带来了有趣的结果：虚拟内存中的页面无须在物理内存保持特定顺序。

3.1.1 请求换页

在物理内存比虚拟内存小得多的系统中，操作系统必须提高物理内存的使用效率。节省物理内存的一种方法是仅加载那些正在被执行程序使用的虚拟页面。比如说，某个数据库程序可能要对某个数据库进行查询操作，此时并不是数据库的所有内容都要加载到内存中去，而只加载那些要用的部分。如果此数据库查询是一个搜索查询而无须对数据库进行添加记录操作，则加载添加记录的代码是毫无意义的。这种仅将要访问的虚拟页面载入的技术叫请求换页。

当进程试图访问当前不在内存中的虚拟地址时，处理器在页表中无法找到所引用地址的入口。在图 3.1 中，对于虚拟页面框号 2，进程 X 的页表中没有入口，这样当进程 X 试图访问虚拟页面框号 2 内容时，处理器不能将此地址转换成物理地址。这时处理器通知操作系统有页面错误发生。

如果发生页面错的虚拟地址是无效的，则表明进程在试图访问一个不存在的虚拟地址。这可能是应用程序出错而引起的，例如它试图对内存进行一个随机的写操作。此时操作系统将终止此应用的运行以保护系统中其他进程不受此出错进程的影响。

如果出错虚拟地址是有效的，但是它指向的页面当前不在内存中，则操作系统必须将此页面从磁盘映象中读入到内存中来。由于访盘时间较长，进程必须等待一段时间直到页面被取出来。如果系统中还存在其他进程，操作系统就会在读取页面过程中的等待过程中选择其中之一来运行。读取回来的页面将被放在一个空闲的物理页面框中，同时此进程的页表中将添加对应此虚拟页面框号的入口。最后进程将从发生页面错误的地方重新开始运行。此时整个虚拟内存访问过程告一段落，处理器又可以继续进行虚拟地址到物理地址转换，而进程也得以继续运行。

Linux 使用请求换页将可执行映象加载到进程的虚拟内存中。当命令执行时，可执行的命令文件被打开，同时其内容被映射到进程的虚拟内存。这些操作是通过修改描叙进程内存映象的数据结构来完成的，此过程称为内存映射。然而只有映象的起始部分被调入物理内存，其余部分仍然留在磁盘上。当映象执行时，它会产生页面错误，这样 Linux 将决定将磁盘上哪些部分调入内存继续执行。

3.1.2 交换

如果进程需要把一个虚拟页面调入物理内存而正好系统中没有空闲的物理页面，操作系统必须丢弃位于物理内存中的某些页面来为之腾出空间。

如果那些从物理内存中丢弃出来的页面来自于磁盘上的可执行文件或者数据文件，并且没有修改过则不需要保存那些页面。当进程再次需要此页面时，直接从可执行文件或者数据文件中读出。

但是如果页面被修改过，则操作系统必须保留页面的内容以备再次访问。这种页面被称为 dirty 页面， 当从内存中移出来时，它们必须保存在叫做交换文件的特殊文件中。相对于处理器和物理内存的速度，访问交换文件的速度是非常缓慢的，操作系统必须在将这些 dirty 页面写入磁盘和将其继续保留在内存中做出选择。

选择丢弃页面的算法经常需要判断哪些页面要丢弃或者交换，如果交换算法效率很低，则会发生"颠簸"现象。在这种情况下，页面不断的被写入磁盘又从磁盘中读回来，这样一来操作系统就无法进行其他任何工作。以图 3.1 为例，如果物理页面框号 1 被频繁使用，则页面丢弃算法将其作为交换到硬盘的侯选者是不恰当的。一个进程当前经常使用的页面集合叫做工作集。高效的交换策略能够确保所有进程的工作集保存在物理内存中。

Linux 使用最近最少使用（LRU）页面衰老算法来公平地选择将要从系统中抛弃的页面。这种策略为系统中的每个页面设置一个年龄，它随页面访问次数而变化。页面被访问的次数越多则页面年龄越年轻；相反则越衰老。年龄较老的页面是待交换页面的最佳侯选者。

3.1.3 共享虚拟内存

虚拟内存让多个进程之间可以方便地共享内存。所有的内存访问都是通过每个进程自身的页表进行。对于两个共享同一物理页面的进程，在各自的页表中必须包含有指向这一物理页面框号的页表入口。

图 3.1 中两个进程共享物理页面框号 4。对进程 X 来说其对应的虚拟页面框号为 4 而

进程 Y 的为 6。这个有趣的现象说明：共享物理页面的进程对应此页面的虚拟内存位置可以不同。

3.1.4 物理与虚拟寻址模式

操作系统自身也运行在虚拟内存中的意义不大。如果操作系统被迫维护自身的页表那将是一个令人恶心的方案。多数通用处理器同时支持物理寻址和虚拟寻址模式。物理寻址模式无需页表的参与且处理器不会进行任何地址转换。Linux 核心直接运行在物理地址空间上。

Alpha AXP 处理器没有特殊的物理寻址模式。它将内存空间划分为几个区域并将其中两个指定为物理映射地址。核心地址空间被称为 KSEG 地址空间，它位于地址 0xfffffc0000000000 以上区域。为了执行位于 KSEG 的核心代码或访问那里的数据，代码必须在核心模式下执行。Alpha 上的 Linux 核心从地址 0xfffffc0000310000 开始执行.

3.1.5 访问控制

页表入口包含了访问控制信息。由于处理器已经将页表入口作为虚拟地址到物理地址的映射，那么可以很方便地使用访问控制信息来判断处理器是否在以其应有的方式来访问内存。

诸多因素使得有必要严格控制对内存区域的访问。有些内存，如包含执行代码的部分，显然应该是只读的，操作系统决不能允许进程对此区域的写操作。相反包含数据的页面应该是可写的， 但是去执行这段数据肯定将导致错误发生。多数处理器至少有两种执行方式：核心态与用户态。任何人都不会允许在用户态下执行核心代码或者在用户态下修改核心数据结构。

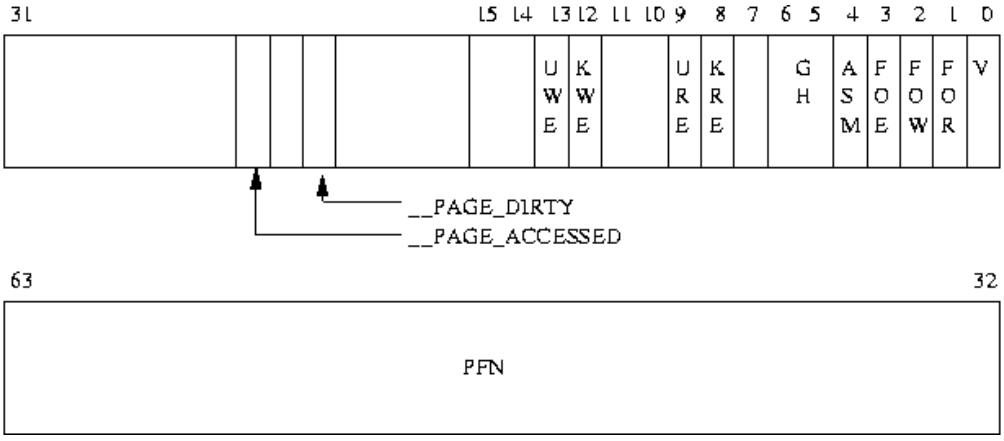


图 3.2 Alpha AXP 页表入口

页表入口中的访问控制信息是处理器相关的；图 3.2 是 Alpha AXP 处理器的 PTE(Page Table Entry)。这些位域的含义如下：

V 有效，如果此位置位，表明此 PTE 有效

FOE “执行时失效”，无论合时只要执行包含在此页面中的指令，处理器都将报告页面错误并将控制传递

FOW “写时失效”，除了页面错误发生在对此页面的写时，其他与上相同。

FOR “读时失效”，除了页面错误发生在对此页面的读时，其他与上相同。
ASM 地址空间匹配。被操作系统用于清洗转换缓冲中的某些入口。
KRE 运行在核心模式下的代码可以读此页面。
URE 运行在用户模式下的代码可以读此页面。
GH 将整个块映射到单个而不是多个转换缓冲时的隐含粒度。
KWE 运行在核心模式下的代码可以写此页面。
UWE 运行在用户模式下的代码可以写此页面。
page frame number 对于 V 位置位的 PTE，此域包含了对应此 PTE 的物理页面框号；
对于无效 PTE，此域不为 0，它包含了页面在交换文件中位置的信息。
以下两位由 Linux 定义并使用。
_PAGE_DIRTY 如果置位，此页面要被写入交换文件。
_PAGE_ACCESSED Linux 用它表示页面已经被访问过。

3.2 高速缓冲

如果用上述理论模型来实现一个系统，它可能可以工作，但效率不会高。操作系统设计者和处理器设计者都在努力以提高系统的性能。除了制造更快的 CPU 和内存外，最好的办法是在高速缓冲中维护有用信息和数据以加快某些操作。Linux 使用了许多与高速缓冲相关的内存管理策略。

Buffer Cache

这个 buffer cache 中包含了被块设备驱动使用的数据缓冲。

这些缓冲的单元的大小一般固定(例如说 512 字节)并且包含从块设备读出或者写入的信息块。块设备是仅能够以固定大小块进行读写操作的设备。所有的硬盘都是块设备。

利用设备标志符和所需块号作索引可以在 buffer cache 中迅速地找到数据。块设备只能够通过 buffer cache 来存取。如果数据在 buffer cache 中可以找到则无需从物理块设备(如硬盘)中读取，这样可以加速访问。

Page Cache

用来加速硬盘上可执行映象文件与数据文件的存取。

它每次缓冲一个页面的文件内容。页面从磁盘上读入内存后缓存在 page cache 中。

Swap Cache

只有修改过的页面存储在交换文件中。

只要这些页面在写入到交换文件后没有被修改，则下次此页面被交换出内存时，就不必再进行更新写操作，这些页面都可以简单的丢弃。在交换频繁发生的系统中，Swap Cache 可以省下很多不必要且耗时的磁盘操作。

Hardware Caches

一个常见的 hardware cache 是处理器中的页表入口 cache。处理器不总是直接读取页表而是在需要时缓存页面的转换。这种 cache 又叫做转换旁视缓冲(Translation Look-aside Buffers)，它包含系统中一个或多个处理器的页表入口的缓冲拷贝。

当发出对虚拟地址的引用时，处理器试图找到相匹配的 TLB 入口。如果找到则直接将虚拟地址转换成物理地址并对数据进行处理。如果没有找到则向操作系统寻求帮助。处理器将向操作系统发出 TLB 失配信号，它使用一个特定的系统机制来将此异常通知操作系统。操作系统则为此地址匹配对产生新的 TLB 入口。当操作系统清除此异常时，处理器将再次进行虚拟地址转换。由于此时在 TLB 中已经有相应的入口，这次操作将成功。

使用高速缓存的缺点在于 Linux 必须消耗更多的时间和空间来维护这些缓存，并且当缓存系统崩溃时系统也将崩溃。

3.3 Linux 页表

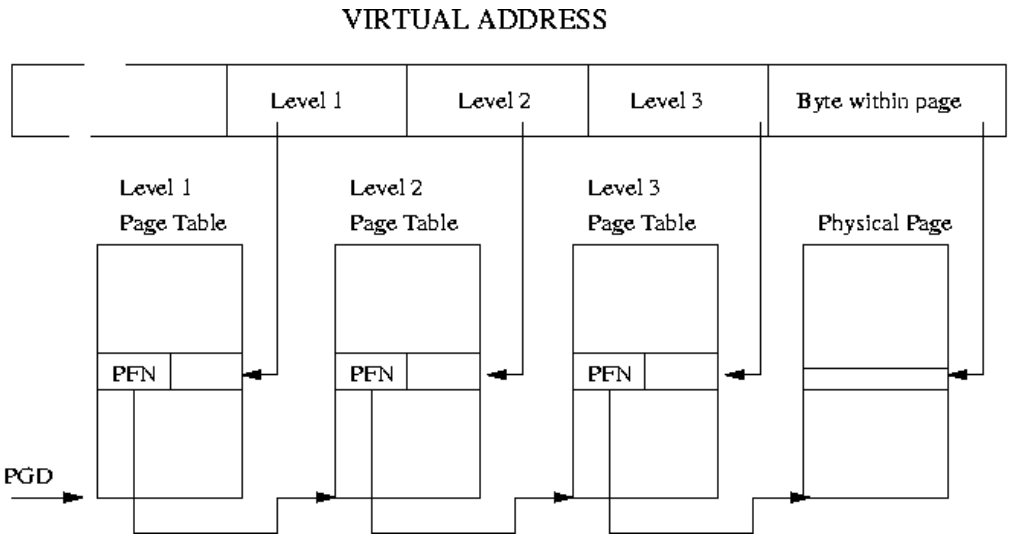


图 3.3 Linux 的三级页表结构

Linux 总是假定处理器有三级页表。每个页表通过所包含的下级页表的页面框号来访问。图 3.3 给出了虚拟地址是如何分割成多个域的，每个域提供了某个指定页表的偏移。为了将虚拟地址转换成物理地址，处理器必须得到每个域的值。这个过程将持续三次直到对应于虚拟地址的物理页面框号被找到。最后再使用虚拟地址中的最后一个域，得到了页面中数据的地址。

为了实现跨平台运行，Linux 提供了一系列转换宏使得核心可以访问特定进程的页表。这样核心无需知道 页表入口的结构以及它们的排列方式。

这种策略相当成功，无论在具有三级页表结构的 Alpha AXP 还是两级页表的 Intel X86 处理器中，Linux 总是使用相同的页表操纵代码。

3.4 页面分配与回收

对系统中物理页面的请求十分频繁。例如当一个可执行映象被调入内存时，操作系统必须为其分配页面。当映象执行完毕和卸载时这些页面必须被释放。物理页面的另一个用途是存储页表这些核心数据结构。虚拟内存子系统中负责页面分配与回收的数据结构和机制可能用处最大。

系统中所有的物理页面用包含 mem_map_t 结构的链表 mem_map 来描叙，这些结构在系统启动时初始化。每个 mem_map_t 描叙了一个物理页面。其中与内存管理相关的重要域如下：

count

记录使用此页面的用户个数。当这个页面在多个进程之间共享时，它的值大于 1。

age

此域描叙页面的年龄，用于选择将适当的页面抛弃或者置换出内存时。

map_nr

记录本 mem_map_t 描叙的物理页面框号。

页面分配代码使用 free_area 数组来寻找和释放页面，此机制负责整个缓冲管理。另外此代码与处理器使用的页面大小和物理分页机制无关。

free_area 中的每个元素都包含页面块的信息。数组中第一个元素描叙 1 个页面，第二个表示 2 个页面大小的块而接下来表示 4 个页面大小的块，总之都是 2 的次幂倍大小。list 域表示一个队列头，它包含指向 mem_map 数组中 page 数据结构的指针。所有的空闲页面都在此队列中。map 域是指向某个特定页面尺寸的页面组分配情况位图的指针。当页面的第 N 块空闲时，位图的第 N 位被置位。

图 free-area-figure 画出了 free_area 结构。第一个元素有个自由页面（页面框号 0），第二个元素有 4 个页面大小的 2 个自由块，前一个从页面框号 4 开始而后一个从页面框号 56 开始。

3.4.1 页面分配

Linux 使用 Buddy 算法来有效的分配与回收页面块。页面分配代码每次分配包含一个或者多个物理页面的内存块。页面以 2 的次幂的内存块来分配。这意味着它可以分配 1 个、2 个和 4 个页面的块。只要系统中有足够的空闲页面来满足这个要求(nr_free_pages > min_free_page)，内存分配代码将在 free_area 中寻找一个与请求大小相同的空闲块。free_area 中的每个元素保存着一个反映这样大小的已分配与空闲页面 的位图。例如，free_area 数组中第二个元素指向一个反映大小为四个页面的内存块分配情况的内存映象。

分配算法首先搜寻满足请求大小的页面。它从 free_area 数据结构的 list 域着手沿链来搜索空闲页面。如果没有这样请求大小的空闲页面，则它搜索两倍于请求大小的内存块。这个过程一直将持续到 free_area 被搜索完或找到满足要求的内存块为止。如果找到的页面块大于请求的块则对其进行分割以使其大小与请求块匹配。由于块大小都是 2 的次幂所以分割过程十分简单。空闲块被连进相应的队列而这个页面块被分配给调用者。

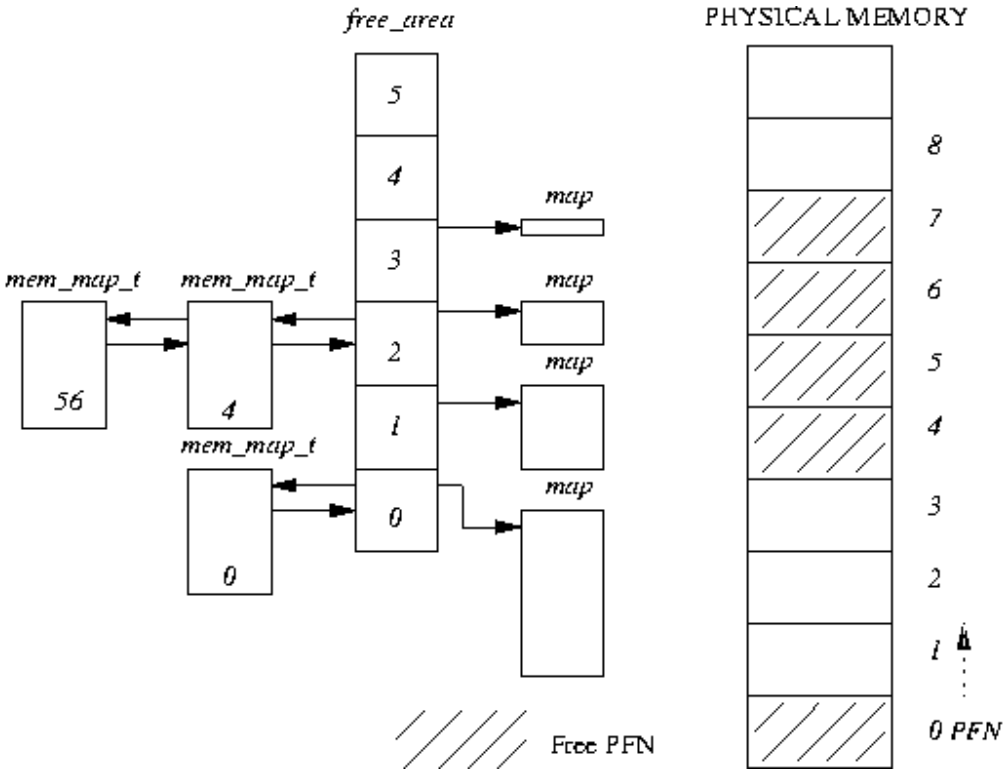


图 3.4 free_area 数据结构

在图 3.4 中，当系统中有大小为两个页面块的请求发出时，第一个 4 页面大小的内存块（从页面框号 4 开始）将分成两个 2 页面大小的块。前一个，从页面框号 4 开始的，将分配出去返回给请求者，而后一个，从页面框号 6 开始，将被添加到 free_area 数组中表示两个页面大小的空闲块的元素 1 中。

3.4.2 页面回收

将大的页面块打碎进行分配将增加系统中零碎空闲页面块的数目。页面回收代码在适当时机下要将这些页面结合起来形成单一大页面块。事实上页面块大小决定了页面重新组合的难易程度。

当页面块被释放时，代码将检查是否有相同大小的相邻或者 buddy 内存块存在。如果有，则将它们结合起来形成一个大小为原来两倍的新空闲块。每次结合完之后，代码还要检查是否可以继续合并成更大的页面。最佳情况是系统的空闲页面块将和允许分配的最大内存一样大。

在图 3.4 中，如果释放页面框号 1，它将和空闲页面框号 0 结合作为大小为 2 个页面的空闲块排入 free_area 的第一个元素中。

3.5 内存映射

映象执行时，可执行映象的内容将被调入进程虚拟地址空间中。可执行映象使用的共享库同样如此。然而可执行文件实际上并没有调入物理内存，而是仅仅连接到进程的虚拟内存。当程序的其他部分运行时引用到这部分时才把它们从磁盘上调入内存。将映象连接到进程虚拟地址空间的过程称为内存映射。

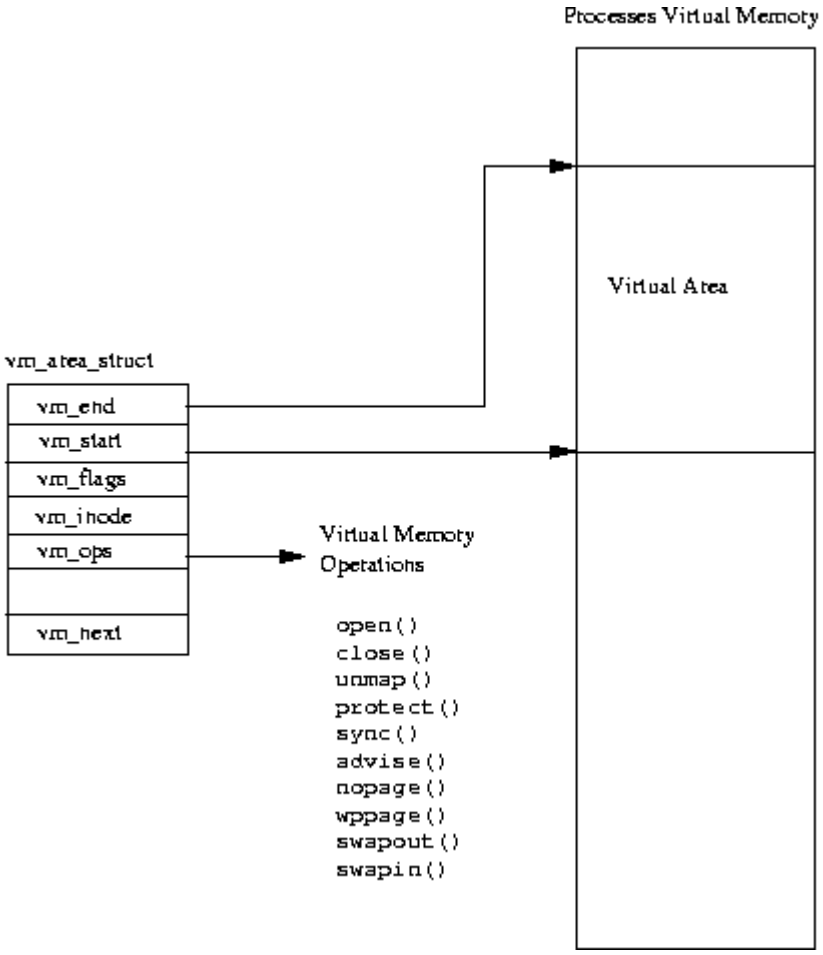


图 3.5 虚拟内存区域

每个进程的虚拟内存用一个 mm_struct 来表示。它包含当前执行的映象（如 BASH）以及指向 vm_area_struct 的大量指针。每个 vm_area_struct 数据结构描叙了虚拟内存的起始与结束位置，进程对此内存区域的存取权限以及一组内存操作函数。这些函数都是 Linux 在操纵虚拟内存区域时必须用到的子程序。其中一个负责处理进程试图访问不在当前物理内存中的虚拟内存(通过页面失效)的情况。此函数叫 nopage。它用在 Linux 试图将可执行映象的页面调入内存时。

可执行映象映射到进程虚拟地址时将产生一组相应的 vm_area_struct 数据结构。每个 vm_area_struct 数据结构表示可执行映象的一部分：可执行代码、初始化数据(变量)、未初始化数据等等。Linux 支持许多标准的虚拟内存操作函数，创建 vm_area_struct 数据结构时

有一组相应的虚拟内存操作函数与之对应。

3.6 请求换页

当可执行映象到进程虚拟地址空间的映射完成后，它就可以开始运行了。由于只有很少部分的映象调入内存，所以很快就会发生对不在物理内存中的虚拟内存区域的访问。当进程访问无有效页表入口的虚拟地址时，处理器将向 Linux 报告一个页面错误。

页面错误带有失效发生的虚拟地址及引发失效的访存方式。Linux 必须找到表示此区域的 vm_area_struct 结构。对 vm_area_struct 数据结构的搜寻速度决定了处理页面错误的效率，而所有 vm_area_struct 结构是通过一种 AVL(Adelson-Velskii and Landis) 树结构连在一起的。如果无法找到 vm_area_struct 与此失效虚拟地址的对应关系，则系统认为此进程访问了非法虚拟地址。这时 Linux 将向进程发送 SIGSEGV 信号，如果进程没有此信号的处理过程则终止运行。

如果找到此对应关系，Linux 接下来检查引起该页面错误的访存类型。如果进程以非法方式访问内存，比如对不可写区域进行写操作，系统将产生内存错误的信号。

如果 Linux 认为页面出错是合法的，那么它需要对这种情况进行处理。

首先 Linux 必须区分位于交换文件中的页面和那些位于磁盘上的可执行映象。Alpha AXP 的页表中有可能存在有效位没有设置但是在 PFN 域中有非 0 值的页表入口。在这种情况下，PFN 域指示的是此页面在交换文件中的位置。如何处理交换文件中的页面将在下章讨论。

不是所有的 vm_area_struct 数据结构都有一组虚拟内存操作函数，它们有的甚至没有 nopage 函数。这是因为 Linux 通过分配新的物理页面并为其创建有效的页表入口来修正这次访问。如果这个内存区域存在 nopage 操作函数，Linux 将调用它。

一般 Linux nopage 函数被用来处理内存映射可执行映象，同时它使用页面 cache 将请求的页面调入物理内存中去。

当请求的页面调入物理内存时，处理器页表也必须更新。更新这些入口必须进行相关硬件操作，特别是处理器使用 TLB 时。这样当页面失效被处理完毕后，进程将从发生失效虚拟内存访问的位置重新开始运行。

3.7 Linux 页面 cache

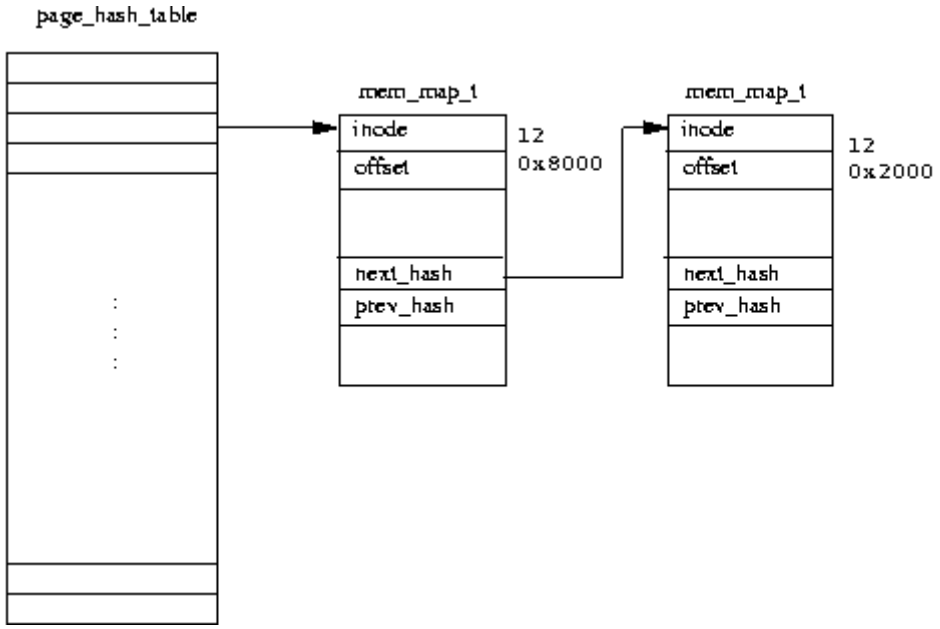


图 3.6 Linux 页面 Cache

Linux 使用页面 cache 的目的是加快对磁盘上文件的访问。内存映射文件以每次一页的方式读出并将这些页面存储在页面 cache 中。图 3.6 表明页面 cache 由 page_hash_table，指向 mem_map_t 数据结构的指针数组组成。

Linux 中的每个文件通过一个 VFS inode（在文件系统一章中讲叙）数据结构来标识并且每个 VFS inode 都是唯一的，它可以并仅可以描叙一个文件。页表的索引从文件的 VFS inode 和文件的偏移中派生出来。

从一个内存映射文件中读出页面，例如产生换页请求时要将页面读回内存中，系统尝试从页面 cache 来读出。如果页面在 cache 中，则返回页面失效处理过程一个指向 mem_map_t 数据结构；否则此页面将从包含映象的文件系统中读入内存并为之分配物理页面。

在映象的读入与执行过程中，页面 cache 不断增长。当不再需要某个页面时，即不再被任何进程使用时，它将被从页面 cache 中删除。

3.8 换出与丢弃页面

当系统中物理内存减少时，Linux 内存管理子系统必须释放物理页面。这个任务由核心交换后台进程(kswapd)来完成。

核心交换后台进程是一种特殊的核心线程。它是没有虚拟内存的进程，在物理地址空间上以核心态运行。核心交换后台进程的名字容易使人误解，其实它完成的工作比仅仅将页面交换到系统的交换文件中要多得多。其目标是保证系统中有足够的空闲页面来维持内存管理系统运行效率。

此进程由核心的 init 进程在系统启动时运行，被核心交换定时器周期性的调用。

当定时器到时后，交换后台进程将检查系统中的空闲页面数是否太少。它使用两个变量：free_pages_high 和 free_page_low 来判断是否该释放一些页面。只要系统中的空闲页面数大于 free_pages_high，核心交换后台进程不做任何工作；它将睡眠到下一次定时器到时。在检查中，核心交换后台进程将当前被写到交换文件中的页面数也计算在内，它使用 nr_async_pages 来记录这个数值；当有页面被排入准备写到交换文件队列中时，它将递增一次，同时当写入操作完成后递减一次。如果系统中的空闲页面数在 free_pages_high 甚至 free_pages_low 以下时，核心交换后台进程将通过三个途径来减少系统中使用的物理页面的个数：

减少缓冲与页面 cache 的大小，

将系统 V 类型的内存页面交换出去，

换出或者丢弃页面。

如果系统中空闲页面数低于 free_pages_low，核心交换后台进程将在下次运行之前释放 6 个页面。否则它只释放 3 个。以上三种方法将依次使用直到系统释放出足够的空闲页面。当核心交换后台进程试图释放物理页面时它将记录使用的最后一种方法。下一次它会首先运行上次最后成功的算法。

当释放出足够页面后，核心交换后台进程将再次睡眠到下次定时器到时。如果导致核心交换后台进程释放页面的原因是系统中的空闲页面数小于 free_pages_low，则它只睡眠平时的一半时间。一旦空闲页面数大于 free_pages_low 则核心交换进程的睡眠时间又会延长。

3.8.1 减少 Page Cache 和 Buffer Cache 的大小

Page Cache 和 Buffer cache 中的页面将被优先考虑释放到 free_area 数组中。Page Cache 中包含的是内存映射文件的页面，其中有些可能是不必要的，它们浪费了系统的内存。而 Buffer Cache 中包含的是从物理设备中读写的缓冲数据，有些可能也是不必要的。当系统中物理页面开始耗尽时，从这些 cache 中丢弃页面比较简单（它不需要象从内存中交换一样，无须对物理设备进行写操作）。除了会使对物理设备及内存映射文件的访问速度降低外，页面丢弃策略没有太多的副作用。如果策略得当，则所有进程的损失相同。

每次核心交换后台进程都会尝试去压缩这些 cache。

它首先检查 mem_map 页面数组中的页面块看是否有可以从物理内存中丢弃出去的。当系统中的空闲页面数降低 到一个危险水平时，核心后台交换进程频繁进行交换，则检查的页面块一般比较大。检查的方式为轮转，每次试图压缩内存映象时，核心后台交换进程总是检查不同的页面块。这是众所周知的 clock 算法，每次在整个 mem_map 页面数组中对页面进行检查。

核心后台交换进程将检查每个页面看是否已经被 page cache 或者 buffer cache 缓冲。读者可能已经注意到共享页面不在被考虑丢弃的页面之列，这种页面不会同时出现在这两种 cache 中。如果页面不在这两者中任何一种之中时，它将检查 mem_map 页面数组中的下一个页面。

缓存在 buffer cache(或者页面中的缓冲被缓存)中的页面可以使缓冲分配和回收更加有效。内存压缩代码将 力图释放在受检页面中包含的缓冲区。

如果页面中包含的所有缓冲区都被释放，这个页面也将被释放。如果受检页面在 Linux 的 page cache 中，则它会从 page cache 中删除并释放。

如果释放出来了足够的页面，核心交换后台进程将等待到下一次被唤醒。这些被释放

的页面都不是任何进程虚拟内存的一部分，这样无须更新页表。如果没有足够的缓冲页面丢弃则交换进程将试图将一些共享页面交换出去。

3.8.2 换出系统 V 内存页面

系统 V 共享内存是一种用来在进程之间通过共享虚拟内存来实现进程通讯的机制。进程是如何共享内存将在 IPC 一章中详细讨论。现在只需要说明系统 V 共享内存的任何区域都可以用一个 shmid_ds 数据结构来表示就足够了。 此结构包含一个指向 vm_area 的链表指针，vm_area 是为每个共享此虚拟内存区域设计的结构。它们之间通过 vm_next_shared 和 vm_prev_shared 指针来连接。每个 shmid_ds 数据结构包含一个页表入口，每个入口描叙物理页面与共享虚拟页面之间的映射关系。

核心交换后台进程同样使用 clock 算法来将系统 V 共享内存页面交换出去。

每次运行时，它要记得哪个共享虚拟内存区域的哪个页面是最后一个被交换出去的。两个索引可以协助它完成这项工作，其一是一组 shmid_ds 数据结构的索引，另一个是系统 V 共享内存区域的页表入口链表的索引。 这能够保证对系统 V 共享内存区域作出公平的选择。

由于对于给定的系统 V 共享虚拟内存的物理页面框号被保存在所有共享此虚拟内存区域进程的页表中，核心 交换后台进程必须同时修改所有的页表以表示页面不再在内存而在交换文件中。对于每个要交换出去的共享 页面，核心交换后台进程可以在每个共享进程的页表中的页表入口中找到它们(通过 vm_area_struct 数据结 构)。如果对应此系统 V 共享内存的页面的进程页表入口是有效的，它可以将其转变成无效，这样换出页表入口和共享页面的用户数将减一。换出系统 V 共享页表入口的格式中包含一个对应于一组 shmid_ds 数据结构的索引以及一个对系统 V 共享内存区域的页表入口索引。

如果所有共享进程的页表都被修改后此页面的记数为 0 则共享页面可以被写到交换文件中。同样指向此系统 V 共享内存区域的 shmid_ds 数据结构链表中的页表入口也被换出页表入口代替。换出页表入口虽然无效但是它包含一组打开的交换文件的索引，同时还能找到换出页面在文件中的偏移。当页面重新被带入物理内存时，这些信息十分有用。

3.8.3 换出和丢弃页面

交换后台进程依次检查系统中的每个进程以确认谁最适合交换出去。

比较好的候选者是那些可以被交换出去（有些是不可被交换出去的）并且只有一个或者几个页面在内存中的进程。只有那些包含的数据无法检索的页面才会从物理内存中交换到系统交换文件中去。

可执行映象的许多内容都可以从映象文件中读出并且可以很容易重读出来。例如，映象中的可执行指令不能被映象本身修改，所以决不会写到交换文件中去。这些页面直接丢弃就可以。当进程再次引用它们时，只需要从可执行映象文件中读入内存即可。

一旦确定了将要被交换出去的进程，交换后台进程将搜索其整个虚拟内存区域以找到那些没有共享或者加锁的区域。

Linux 并不会将选中的进程的整个可交换页面都交换出去，它只删除一小部分页面。

如果内存被加锁则页面不能被交换或者丢弃。

Linux 交换算法使用页面衰老算法。每个页面有一个计数器来告诉核心交换后台进程

这个页面是否值得交换出 去（此计数器包含在 mem_map_t 结构中）。当页面没有使用或者没有找到时将会衰老；交换后台进程仅仅交换 出那些老页面。缺省操作是：当页面被首次分配时，其年龄初始值为 3，每次引用其年龄将加 3，最大值为 20。 每次核心交换后台进程运行它来使页面衰老-将年龄减 1。这个缺省操作可以改变并且由于这个原因它们被存储在 swap_control 数据结构中。

如果页面变老了(age=0)，则交换后台进程将进一步来处理它。dirty 页面可以被交换出去。Linux 在 PTE 中使 用一个硬件相关位来描叙页面的这个特性（见图 3.2）。然而不是所有的 dirty 页面都有必要写入到交换文件 中去。进程的每个虚拟内存区域可能有其自身的交换操作(由 vm_area_struct 结构中的 vm_ops 指针表示)，在 交换时使用的是这些方法。否则，交换后台进程将在交换文件中分配一个页面并将页面写到设备上去。

页面的页表入口被标志成无效但是它包含了页面在在交换文件中位置的信息，包括一个表示页面在交换文件中位置的偏移值以及使用的是哪个交换文件。但是不管使用的是哪种交换算法，以前那个物理页面将被标志成空闲并放入 free_area 中。Clean（或者 not dirty）的页面可以丢弃同时放入 free_area 以备重新使用。

如果有足够的可交换进程页面被交换出去或丢弃，则交换后台进程将再次睡眠。下次它醒来时将考虑系统中 的下一个进程。通过这种方法，交换后台进程一点一点地将每个进程的可交换或可丢弃物理页面收回知道系 统再次处于平衡状态。这比将整个进程交换出去要公平得多。

3.9 The Swap Cache

当将页面交换到交换文件中时，Linux 总是避免页面写，除非必须这样做。当页面已经被交换出内存但是当有进程再次访问时又要将它重新调入内存。只要页面在内存中没有被写过，则交换文件中的拷贝是有效的。

Linux 使用 swap cache 来跟踪这些页面。这个 swap cache 是一个页表入口链表，每个对应于系统中的物理页面。这是一个对应于交换出页面的页表入口并且描叙页面放置在哪个交换文件中以及在交换文件中的位置。 如果 swap cache 入口为非 0 值，则表示在交换文件中的这一页没有被修改。如果此页被修改（或者写入）。 则其入口从 swap cache 中删除。

当 Linux 需要将一个物理页面交换到交换文件时，它将检查 swap cache，如果对应此页面存在有效入口，则 不必将这个页面写到交换文件中。这是因为自从上次从交换文件中将其读出来，内存中的这个页面还没有被修改。

swap cache 中的入口是已换出页面的页表入口。它们虽被标记为无效但是为 Linux 提供了页面在哪个交换文件中以及文件中的位置等信息。

3.10 页面的换入

保存在交换文件中的 dirty 页面可能被再次使用到，例如，当应用程序向包含在已交换出物理页面上的虚拟内存区域写入时。对不在物理内存中的虚拟内存页面的访问将引发页面错误。由于处理器不能将此虚拟地址转换成物理地址，处理器将通知操作系统。由于已被交换出去，此时描叙此页面的页表入口被标记成无效。处理器不能处理这种虚拟地址到物理地址的转换，所以它将控制传递给操作系统，同时通知操作系统页面错误的地址与原因。这些信息的格式以及处理器如何将控制传递给操作系统与具体硬件有关。

处理器相关页面错误处理代码将定位描叙包含出错虚拟地址对应的虚拟内存区域的 vm_area_struct 数据结构。 它通过在此进程的 vm_area_struct 中查找包含出错虚拟地址的位置直到找到为止。这些代码与时间关系重大，进程的 vm_area_struct 数据结构特意安排成使查找操作时间更少。

执行完这些处理器相关操作并且找到出错虚拟地址的有效内存区域后，页面错处理过程其余部分和前面类似。

通用页面错处理代码为出错虚拟地址寻找页表入口。如果找到的页表入口是一个已换出页面，Linux 必须将其 交换进入物理内存。已换出页面的页表入口的格式与处理器类型有关，但是所有的处理器将这些页面标记成无效并把定位此页面的必要信息放入页表入口中。Linux 利用这些信息以便将页面交换进物理入内存。

此时 Linux 知道出错虚拟内存地址并且拥有一个包含页面位置信息的页表入口。vm_area_struct 数据结构可能包含将此虚拟内存区域交换到物理内存中的子程序:swapin。如果对此虚拟内存区域存在 swapin 则 Linux 会使用它。这是已换出系统 V 共享内存页面的处理过程-因为已换出系统 V 共享页面和普通的已换出页面有少许不同。如果没有 swapin 操作，这可能是 Linux 假定普通页面无须特殊处理。

系统将分配物理页面并将已换出页面读入。关于页面在交换文件中位置信息从页表入口中取出。

如果引起页面错误的访问不是写操作则页面被保留在 swap cache 中并且它的页表入口不再标记为可写。如果 页面随后被写入，则将产生另一个页面错误，这时页面被标记为 dirty，同时其入口从 swap cache 中删除。 如果页面没有被写并且被要求重新换出，Linux 可以免除这次写，因为页面已经存在于交换文件中。

如果引起页面从交换文件中读出的操作是写操作，这个页面将被从 swap cache 中删除并且其页表入口被标记 成 dirty 且可写。

第四章 进程管理

本章重点讨论 Linux 内核如何在系统中创建、管理以及删除进程。

进程在操作系统中执行特定的任务。而程序是存储在磁盘上包含可执行机器指令和数据的静态实体。进程或者任务是处于活动状态的计算机程序。

进程是一个随执行过程不断变化的实体。和程序要包含指令和数据一样，进程也包含程序计数器和所有 CPU 寄存器的值，同时它的堆栈中存储着如子程序参数、返回地址以及变量之类的临时数据。当前的执行程序，或者说进程，包含着当前处理器中的活动状态。Linux 是一个多处理操作系统。进程具有独立的权限与职责。如果系统中某个进程崩溃，它不会影响到其余的进程。每个进程运行在其各自的虚拟地址空间中，通过核心控制下可靠的通讯机制，它们之间才能发生联系。

进程在生命期内将使用系统中的资源。它利用系统中的 CPU 来执行指令，在物理内存来放置指令和数据。使用文件系统提供的功能打开并使用文件，同时直接或者间接的使用物理设备。Linux 必须跟踪系统中每个进程以及资源，以便在进程间实现资源的公平分配。如果系统有一个进程独占了大部分物理内存或者 CPU 的使用时间，这种情况对系统中的其它进程是不公平的。

系统中最宝贵的资源是 CPU，通常系统中只有一个 CPU。Linux 是一个多处理操作系

统，它最终的目的是:任何时刻系统中的每个 CPU 上都有任务执行，从而提高 CPU 的利用率。如果进程个数多于 CPU 的个数，则有些进程必须等待到 CPU 空闲时才可以运行。多处理是的思路很简单；当进程需要某个系统资源时它将停止执行并等待到资源可用时才继续运行。单处理系统中，如 DOS，此时 CPU 将处于空等状态，这个时间将被浪费掉。在多处理系统中，因为可以同时存在多个进程，所以当某个进程开始等待时，操作系统将把 CPU 控制权拿过来并交给其它可以运行的进程。调度器负责选择适当的进程来运行，Linux 使用一些调度策略以保证 CPU 分配的公平性。

Linux 支持多种类型的可执行文件格式，如 ELF，JAVA 等。由于这些进程必须使用系统共享库，所以对它们的管理要具有透明性。

4.1 Linux 进程

为了让 Linux 来管理系统中的进程，每个进程用一个 task_struct 数据结构来表示（任务与进程在 Linux 中可以混用）。数组 task 包含指向系统中所有 task_struct 结构的指针。

这意味着系统中的最大进程数目受 task 数组大小的限制，缺省值一般为 512。创建新进程时，Linux 将从系统内存中分配一个 task_struct 结构并将其加入 task 数组。当前运行进程的结构用 current 指针来指示。

Linux 还支持实时进程。这些进程必须对外部时间作出快速反应（这就是“实时”的意思），系统将区分对待这些进程和其他进程。虽然 task_struct 数据结构庞大而复杂，但它可以分成一些功能组成部分：

State

进程在执行过程中会根据环境来改变 state。Linux 进程有以下状态：

Running

进程处于运行（它是系统的当前进程）或者准备运行状态（它在等待系统将 CPU 分配给它）。

Waiting

进程在等待一个事件或者资源。Linux 将等待进程分成两类；可中断与不可中断。可中断等待进程可以被信号中断；不可中断等待进程直接在硬件条件等待，并且任何情况下都不可中断。

Stopped

进程被停止，通常是通过接收一个信号。正在被调试的进程可能处于停止状态。

Zombie

这是由于某些原因被终止的进程，但是在 task 数据中仍然保留 task_struct 结构。 它象一个已经死亡的进程。

Scheduling Information

调度器需要这些信息以便判定系统中哪个进程最迫切需要运行。

Identifiers

系统中每个进程都有进程标志。进程标志并不是 task 数组的索引，它仅仅是个数字。每个进程还有一个用户与组标志，它们用来控制进程对系统中文件和设备的存取权限。

Inter-Process Communication

Linux 支持经典的 Unix IPC 机制，如信号、管道和信号灯以及系统 V 中 IPC 机制，包括共享内存、信号灯和消息队列。我们将在 IPC 一章中详细讨论 Linux 中 IPC 机制。

Links

Linux 系统中所有进程都是相互联系的。除了初始化进程外，所有进程都有一个父进程。新进程不是被创建，而是被复制，或者从以前的进程克隆而来。每个进程对应的 task_struct 结构中包含有指向其父进程和兄弟进程（具有相同父进程的进程）以及子进程的指针。我们可以使用 pstree 命令来观察 Linux 系统中运行进程间的关系：

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

另外，系统中所有进程都用一个双向链表连接起来，而它们的根是 init 进程的 task_struct 数据结构。这个链表被 Linux 核心用来寻找系统中所有进程，它对 ps 或者 kill 命令提供了支持。

Times and Timers

核心需要记录进程的创建时间以及在其生命期中消耗的 CPU 时间。时钟每跳动一次，核心就要更新保存在 jiffies 变量中，记录进程在系统和用户模式下消耗的时间量。Linux 支持与进程相关的 interval 定时器，进程可以通过系统调用来设定定时器以便在定时器到时后向它发送信号。这些定时器可以是一次性的或者周期性的。

File system

进程可以自由地打开或关闭文件，进程的 task_struct 结构中包含一个指向每个打开文件描叙符的指针以及指向两个 VFS inode 的指针。每个 VFS inode 唯一地标记文件中的一个目录或者文件，同时还对底层文件系统提供统一的接口。Linux 对文件系统的支持将在 filesystem 一章中详细描叙。这两个指针，一个指向进程的根目录，另一个指向其当前或者 pwd 目录。pwd 从 Unix 命令 pwd 中派生出来， 用来显示当前工作目录。这两个 VFS inode 包含一个 count 域，当多个进程引用它们时，它的值将增加。这就是为什么你不能删除进程当前目录，或者其子目录的原因。

Virtual memory

多数进程都有一些虚拟内存（核心线程和后台进程没有），Linux 核心必须跟踪虚拟内存与系统物理内存的映射关系。

Processor Specific Context

进程可以认为是系统当前状态的总和。进程运行时，它将使用处理器的寄存器以及堆栈等等。进程被挂起时，进程的上下文-所有的 CPU 相关的状态必须保存在它的 task_struct 结构中。当调度器重新调度该进程时，所有上下文被重新设定。

4.2 Identifiers

和其他 Unix 一样，Linux 使用用户和组标志符来检查对系统中文件和可执行映象的访问权限。Linux 系统中所有的文件都有所有者和允许的权限，这些权限描叙了系统使用者对文件或者目录的使用权。基本的权限是读、写和可执行，这些权限被分配给三类用户：文件的所有者，属于相同组的进程以及系统中所有进程。每类用户具有不同的权限，例如一个文件允许其拥有者读写，但是同组的只能读而其他进程不允许访问。

Linux 使用组将文件和目录的访问特权授予一组用户，而不是单个用户或者系统中所有进程。如可以为某个软件项目中的所有用户创建一个组，并将其权限设置成只有他们才允许读写项目中的源代码。一个进程可以同时属于多个组（最多为 32 个），这些组都被放在进程的 task_struct 中的 group 数组中。只要某组进程可以存取某个文件，则由此组派生出的进程对这个文件有相应的组访问权限。

task_struct 结构中有四对进程和组标志符：

uid, gid

表示运行进程的用户标志符和组标志符。

effective uid and gid

有些程序可以在执行过程中将执行进程的 uid 和 gid 改成其程序自身的 uid 和 gid（保存在描叙可执行映象的 VFS inode 属性中）。这些程序被称为 setuid 程序，常在严格控制对某些服务的访问时使用，特别是那些为别的进程而运行的进程，例如网络后台进程。有效 uid 和 gid 是那些 setuid 执行过程在执行时变化出的 uid 和 gid。当进程试图访问特权数据或代码时，核心将检查进程的有效 gid 和 uid。

file system uid and gid

它们和有效 uid 和 gid 相似但用来检验进程的文件系统访问权限。如运行在用户模式下的 NFS 服务器存取文件时，NFS 文件系统将使用这些标志符。此例中只有文件系统 uid 和 gid 发生了改变（而非有效 uid 和 gid）。这样可以避免恶意用户向 NFS 服务器发送 KILL 信号。

saved uid and gid

POSIX 标准中要求实现这两个标志符，它们被那些通过系统调用改变进程 uid 和 gid 的程序使用。当进程的原始 uid 和 gid 变化时，它们被用来保存真正的 uid 和 gid。

4.3 调度

所有进程部分时间运行于用户模式，部分时间运行于系统模式。如何支持这些模式，底层硬件的实现各不相同，但是存在一种安全机制可以使它们在用户模式和系统模式之间来回切换。用户模式的权限比系统模式下的小得多。进程通过系统调用切换到系统模式继续执行。此时核心为进程而执行。在 Linux 中，进程不能被抢占。只要能够运行它们就不

能被停止。当进程必须等待某个系统事件时，它才决定释放出 CPU。例如进程可能需要从文件中读出字符。一般等待发生在系统调用过程中，此时进程处于系统模式；处于等待状态的进程将被挂起而其他的进程被调度管理器选出来执行。

进程常因为执行系统调用而需要等待。由于处于等待状态的进程还可能占用 CPU 时间，所以 Linux 采用了预加载调度策略。在此策略中，每个进程只允许运行很短的时间：200 毫秒，当这个时间用完之后，系统将选择另一个进程来运行，原来的进程必须等待一段时间以继续运行。这段时间称为时间片。

调度器必须选择最迫切需要运行而且可以执行的进程来执行。

可运行进程是一个只等待 CPU 资源的进程。Linux 使用基于优先级的简单调度算法来选择下一个运行进程。当选定新进程后，系统必须将当前进程的状态，处理器中的寄存器以及上下文状态保存到 task_struct 结构中。同时它将重新设置新进程的状态并将系统控制权交给此进程。为了将 CPU 时间合理的分配给系统中每个可执行进程，调度管理器必须将这些时间信息也保存在 task_struct 中。

policy

应用到进程上的调度策略。系统中存在两类 Linux 进程:普通与实时进程。实时进程的优先级要高于其它进程。如果一个实时进程处于可执行状态，它将先得到执行。实时进程又有两种策略：时间片轮转和先进先出。在时间片轮转策略中，每个可执行实时进程轮流执行一个时间片，而先进先出策略每个可执行进程按各自在运行队列中的顺序执行并且顺序不能变化。

priority

调度管理器分配给进程的优先级。同时也是进程允许运行的时间（jiffies）。系统调用 renice 可以改变进程的优先级。

rt_priority

Linux 支持实时进程，且它们的优先级要高于非实时进程。调度器使用这个域给每个实时进程一个相对优先级。同样可以通过系统调用来改变实时进程的优先级。

counter

进程允许运行的时间(保存在 jiffies 中)。进程首次运行时为进程优先级的数值，它随时间变化递减。

核心在几个位置调用调度管理器。如当前进程被放入等待队列后运行或者系统调用结束时，以及从系统模式返回用户模式时。此时系统时钟将当前进程的 counter 值设为 0 以驱动调度管理器。每次调度管理器运行时将进行下列操作：

kernel work

调度管理器运行底层处理程序并处理调度任务队列。kernel 一章将详细描叙这个轻量级核心线程。

Current process

当选定其他进程运行之前必须对当前进程进行一些处理。

如果当前进程的调度策略是时间片轮转，则它被放回到运行队列。

如果任务可中断且从上次被调度后接收到了一个信号，则它的状态变为 Running。

如果当前进程超时，则它的状态变为 Running。

如果当前进程的状态是 Running，则状态保持不变。那些既不处于 Running 状态又不是可中断的进程将会从运行队列中删除。这意味着调度管理器选择运行进程时不会将这些进程考虑在内。

Process selection

调度器在运行队列中选择一个最迫切需要运行的进程。如果运行队列中存在实时进程（那些具有实时调度策略的进程），则它们比普通进程更多的优先级权值。普通进程的权值是它的 counter 值，而实时 进程则是 counter 加上 1000。这表明如果系统中存在可运行的实时进程，它们将总是在任何普通进程之前运行。如果系统中存在和当前进程相同优先级的其它进程，这时当前运行进程已经用掉了一些时间片，所以它将处在不利形势（其 counter 已经变小）；而原来优先级与它相同的进程的 counter 值显然比它大，这样位于运行队列中最前面的进程将开始执行而当前进程被放回到运行队列中。在存在多个相同优先级进程的平衡系统中，每个进程被依次执行，这就是 Round Robin 策略。然而由于进程经常需要等待某些资源，所以它们的运行顺序也常发变化。

Swap processes

如果系统选择其他进程运行，则必须被挂起当前进程且开始执行新进程。进程执行时将使用寄存器、物理内存以及 CPU。每次调用子程序时，它将参数放在寄存器中并把返回地址放置在堆栈中，所以调度管理器总是运行在当前进程的上下文。虽然可能在特权模式或者核心模式中，但是仍然处于当前运行进程中。当挂起进程的执行时，系统的机器状态，包括程序计数器(PC)和全部的处理器寄存器，必须存储在进程的 task_struct 数据结构中。同时加载新进程的机器状态。这个过程与系统类型相关，不同的 CPU 使用不同的方法完成这个工作，通常这个操作需要硬件辅助完成。

进程的切换发生在调度管理器运行之后。以前进程保存的上下文与当前进程加载时的上下文相同，包括进程程序计数器和寄存器内容。

如果以前或者当前进程使用了虚拟内存，则系统必须更新其页表入口，这与具体体系结构有关。如果处理器使用了转换旁视缓冲或者缓冲了页表入口(如 Alpha AXP)，那么必须冲刷以前运行进程的页表入口。

4.3.1 多处理器系统中的调度

在 Linux 世界中，多 CPU 系统非常少见。但是 Linux 上已经做了很多工作来保证它能运行在 SMP（对称多处理）机器上。Linux 能够在系统中的 CPU 间进行合理的负载平衡调度。这里的负载平衡工作比调度管理器所做的更加明显。

在多处理器系统中，人们希望每个处理器总处与工作状态。当处理器上的当前进程用完它的时间片或者等待系统资源时，各个处理器将独立运行调度管理器。SMP 系统中一个值得注意的问题是系统中不止一个 idle 进程。在单处理器系统中，idle 进程是 task 数组中的第一个任务，在 SMP 系统中每个 CPU 有一个 idle 进程，同时每个 CPU 都有一个当前进程，SMP 系统必须跟踪每个处理器中的 idle 进程和当前进程。

在 SMP 系统中，每个进程的 task_struct 结构中包含着当前运行它的处理器的编号以及上次运行时处理器的编号。 把进程每次都调度到不同 CPU 上执行显然毫无意义，Linux 可以使用 processor_mask 来使得某个进程只在一个或者几个处理器上运行:如果 N 位置位，则进程可在处理器 N 上运行。当调度管理器选择新进程运行时，它不会考虑一个在其 processor_mask 中在当前处理器位没有置位的进程。同时调度管理器将给予上次在此处理器中运行的进程一些优先权，因为将进程迁移到另外处理器上运行将带来性能的损失。

4.4 文件

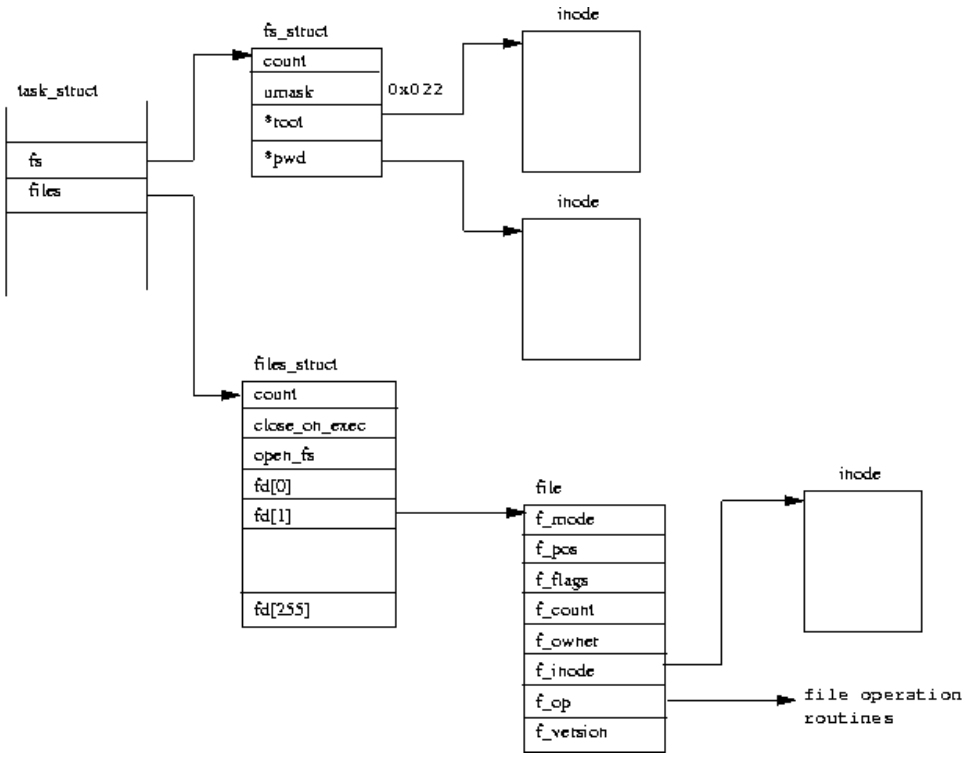


图 4.1 进程所使用的文件

图 4.1 给出了两个描叙系统中每个进程所使用的文件系统相关信息。第一个 fs_struct 包含了指向进程的 VFS inode 和其屏蔽码。这个屏蔽码值是创建新文件时所使用的缺省值，可以通过系统调用来改变。

第二个数据结构 files_struct 包含了进程当前所使用的所有文件的信息。程序从标准输入中读取并写入到标准输出中去。任何错误信息将输出到标准错误输出。这些文件有些可能是真正的文件，有的则是输出/输入终端或者物理设备，但程序都将它们视为文件。每个文件有一个描叙符，files_struct 最多可以包含 256 个文件数据结构，它们分别描叙一个被当前进程使用的文件。f_mode 域表示文件将以何种模式创建：只读 、读写还是只写。f_pos 中包含下一次文件读写操作开始位置。f_inode 指向描叙此文件的 VFS inode， f_ops 指向一组可以对此文件进行操作的函数入口地址指针数组。这些抽象接口十分强大，它们使得 Linux 能够支持多种文件类型。在 Linux 中，管道是用我们下面要讨论的机制实现的。

每当打开一个文件时，位于 files_struct 中的一个空闲文件指针将被用来指向这个新的文件结构。Linux 进 程希望在进程启动时至少有三个文件描叙符被打开，它们是标准输入，标准输出和标准错误输出，一般进程 会从父进程中继承它们。这些描叙符用来索引进程的 fd 数组，所以标准输入，标准输出和标准错误输出分别 对应文件描叙符 0，1 和 2。每次对文件的存取都要通过文件数据结构中的文件操作子程序和 VFS inode 一起来完成，

4.5 虚拟内存

进程的虚拟内存包括可执行代码和多个资源数据。首先加载的是程序映象，例如 ls。ls 和所有可执行映象一样，是由可执行代码和数据组成的。此映象文件包含所有加载可执行代码所需的信息，同时还将程序数据连接进入进程的虚拟内存空间。然后在执行过程中，进程定位可以使用的虚拟内存，以包含正在读取的文件内容。新分配的虚拟内存必须连接到进程已存在的虚拟内存中才能够使用。 最后 Linux 进程调用通用库过程，比如文件处理子程序。如果每个进程都有库过程的拷贝，那么共享就变得没有意义。而 Linux 可以使多个进程同时使用共享库。来自共享库的代码和数据必须连接进入进程的虚拟地址空间以及共享此库的其它进程的虚拟地址空间。

任何时候进程都不同时使用包含在其虚拟内存中的所有代码和数据。虽然它可以加载在特定情况下使用的那些代码，如初始化或者处理特殊事件时，另外它也使用了共享库的部分子程序。但如果将这些没有或很少使用的代码和数据全部加载到物理内存中引起极大的浪费。如果系统中多个进程都浪费这么多资源，则会大大降低的系统效率。Linux 使用请求调页技术来把那些进程需要访问的虚拟内存带入物理内存中。核心将进程页表中这些虚拟地址标记成存在但不在内存中的状态，而无需将所有代码和数据直接调入物理内存。当进程试图访问这些代码和数据时，系统硬件将产生页面错误并将控制转移到 Linux 核心来处理之。这样对于处理器地址空间中的每个虚拟内存区域，Linux 都必须知道这些虚拟内存从何处而来以及如何将其载入内存以处理页面错误。

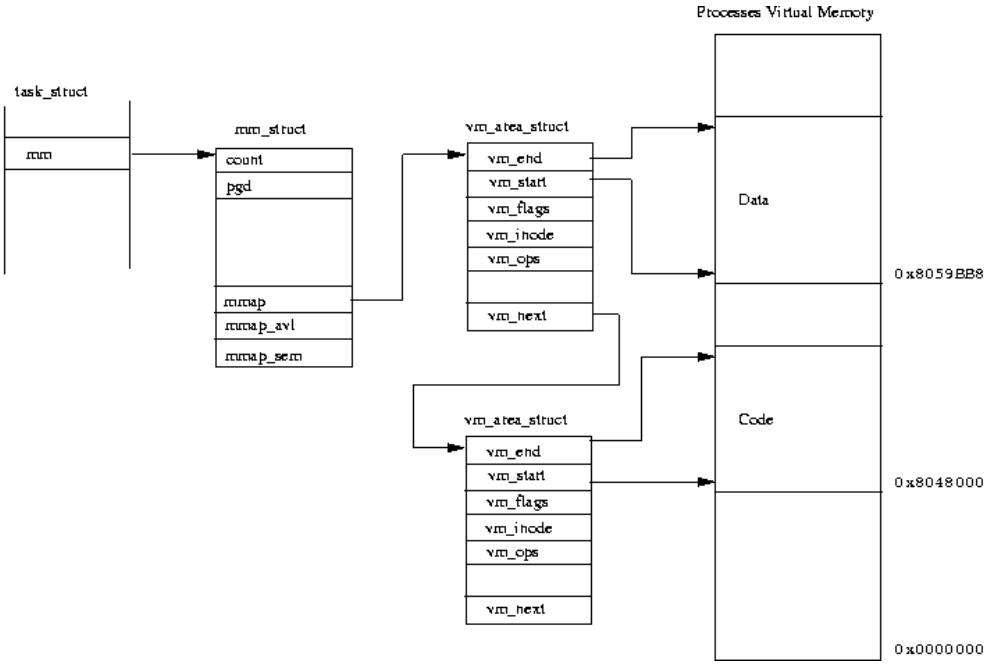


图 4.2 进程的虚拟内存

Linux 核心需要管理所有的虚拟内存地址，每个进程虚拟内存中的内容在其 task_struct 结构中指向的 vm_area_struct 结构中描叙。进程的 mm_struct 数据结构也包含了已加载可执行映象的信息和指向进程页表 的指针。它还包含了一个指向 vm_area_struct 链表的指针，每个指针代表进程内的一个虚拟内存区域。

此链表按虚拟内存位置来排列，图 4.2 给出了一个简单进程的虚拟内存以及管理它的核心数据结构分布图。 由于那些虚拟内存区域来源各不相同，Linux 使用 vm_area_struct 中指向一组虚拟内存处理过程的指针来抽 象此接口。通过使用这个策略，所有的进程虚拟地址可以用相同的方式处理而无需了解底层对于内存管理的区别。如当进程试图访问不存在内存区域时，系统只需要调用页面错误处理过程即可。

为进程创建新虚拟内存区域或处理页面不在物理内存错误时，Linux 核心重复使用进程的 vm_area_struct 数据结构集合。这样消耗在查找 vm_area_struct 上的时间直接影响了系统性能。Linux 把 vm_area_struct 数据结构以 AVL(Adelson-Velskii and Landis)树结构连接以加快速度。在这种连接中，每个 vm_area_struct 结构有一个左指针和右指针指向 vm_area_struct 结构。左边的指针指向一个更低的虚拟内存起始地址节点而右边的指针指向一个更高虚拟内存起始地址节点。为了找到某个的节点，Linux 从树的根节点开始查找，直到找到正确的 vm_area_struct 结构。插入或者释放一个 vm_area_struct 结构不会消耗额外的处理时间。

当进程请求分配虚拟内存时，Linux 并不直接分配物理内存。它只是创建一个 vm_area_struct 结构来描叙此虚拟内存，此结构被连接到进程的虚拟内存链表中。当进程试图对新分配的虚拟内存进行写操作时，系统将产生页面错。处理器会尝试解析此虚拟地址，但是如果找不到对应此虚拟地址的页表入口时，处理器将放弃解析并产生页面错误异常，由 Linux 核心来处理。Linux 则查看此虚拟地址是否在当前进程的虚拟地址空间中。如果是 Linux 会创建正确的 PTE 并为此进程分配物理页面。包含在此页面中的代码或数据可能需要从文件系统或者交换磁盘上读出。然后进程将从页面错误处开始继续执行，由于物理内存已经存在，所以不会再产生页面异常。

4.6 进程创建

系统启动时总是处于核心模式，此时只有一个进程：初始化进程。象所有进程一样，初始化进程也有一个由堆栈、寄存器等表示的机器状态。当系统中有其它进程被创建并运行时，这些信息将被存储在初始化进程的 task_struct 结构中。在系统初始化的最后，初始化进程启动一个核心线程（init）然后保留在 idle 状态。 如果没有任何事要做，调度管理器将运行 idle 进程。idle 进程是唯一不是动态分配 task_struct 的进程，它的 task_struct 在核心构造时静态定义并且名字很怪，叫 init_task。

由于是系统的第一个真正的进程，所以 init 核心线程（或进程）的标志符为 1。它负责完成系统的一些初始化设置任务（如打开系统控制台与安装根文件系统），以及执行系统初始化程序，如/etc/init, /bin/init 或者 /sbin/init ，这些初始化程序依赖于具体的系统。init 程序使用/etc/inittab 作为脚本文件来创建系统中的新进程。这些新进程又创建各自的新进程。例如 getty 进程将在用户试图登录时创建一个 login 进程。系统中所有进程都是从 init 核心线程中派生出来。

新进程通过克隆老进程或当前进程来创建。系统调用 fork 或 clone 可以创建新任务，复制发生在核心状态下的核心中。在系统调用的结束处有一个新进程等待调度管理器选择它去运行。系统从物理内存中分配出来一个新的 task_struct 数据结构，同时还有一个或多个包含被复制进程堆栈（用户与核心）的物理页面。然后创建唯一地标记此新任务的进程标志符。但复制进程保留其父进程的标志符也是合理的。新创建的 task_struct 将被放入 task 数组中，另外将被复制进程的 task_struct 中的内容页表拷入新的 task_struct 中。

复制完成后，Linux 允许两个进程共享资源而不是复制各自的拷贝。这些资源包括文

件、信号处理过程和虚拟内存。进程对共享资源用各自的 count 来记数。在两个进程对资源的使用完毕之前，Linux 绝不会释放此资源，例如复制进程要共享虚拟内存，则其 task_struct 将包含指向原来进程的 mm_struct 的指针。mm_struct 将增加 count 变量以表示当前进程共享的次数。

复制进程虚拟空间所用技术的十分巧妙。复制将产生一组新的 vm_area_struct 结构和对应的 mm_struct 结构，同时还有被复制进程的页表。该进程的任何虚拟内存都没有被拷贝。由于进程的虚拟内存有的可能在物理内存中，有的可能在当前进程的可执行映象中，有的可能在交换文件中，所以拷贝将是一个困难且繁琐的工作。Linux 使用一种"copy on write"技术：仅当两个进程之一对虚拟内存进行写操作时才拷贝此虚拟内存块。但是不管写与不写，任何虚拟内存都可以在两个进程间共享。只读属性的内存，如可执行代码，总是可以共享的。为了使"copy on write"策略工作，必须将那些可写区域的页表入口标记为只读的，同时描叙它们的 vm_area_struct 数据都被设置为"copy on write"。当进程之一试图对虚拟内存进行写操作时将产生页面错误。这时 Linux 将拷贝这一块内存并修改两个进程的页表以及虚拟内存数据结构。

4.7 时钟和定时器

核心跟踪着进程的创建时间以及在其生命期中消耗的 CPU 时间。每个时钟滴答时，核心将更新当前进程在系统 模式与用户模式下所消耗的时间（记录在 jiffies 中）。
除了以上记时器外，Linux 还支持几种进程相关的时间间隔定时器。

进程可以使用这些定时器在到时时向它发送各种信号，这些定时器如下：

Real 此定时器按照实时时钟记数，当时钟到期时，向进程发送 SIGALRM 信号。

Virtual 此定时器仅在进程运行时记数，时钟到期时将发送 SIGVTALRM 信号。

Profile 此定时器在进程运行和核心为其运行时都记数。当到时时向进程发送 SIGPROF 信号。

以上时间间隔定时器可以同时也可以单独运行，Linux 将所有这些信息存储在进程的 task_struct 数据结构中。通过系统调用可以设置这些时间间隔定时器并启动、终止它们或读取它们的当前值。Virtual 和 Profile 定时器以相同方式处理。

每次时钟滴答后当前进程的时间间隔定时器将递减，当到时之后将发送适当的信号。

Real 时钟间隔定时器的机制有些不同，这些将在 kernel 一章中详细讨论。每个进程有其自身的 timer_list 数 据结构，当时间间隔定时器运行时，它们被排入系统的定时器链表中。当定时器到期后，底层处理过程将把它从队列中删除并调用时间间隔处理过程。此过程将向进程发送 SIGALRM 信号并重新启动定时器，将其重新放入系统时钟队列。

4.8 程序执行

象 Unix 一样，Linux 程序通过命令解释器来执行。命令解释器是一个用户进程，人们将其称为 shell 程序。

在 Linux 中有多个 shell 程序，最流行的几个是 sh、bash 和 tcsh。除了几个内置命令如 cd 和 pwd 外，命令都是一个可执行二进制文件。当键入一个命令时，Shell 程序将搜索包含在进程 PATH 环境变量中查找路径中的目录来定位这个可执行映象文件。如果找到，则

它被加载且执行。shell 使用上面描叙的 fork 机制来复制自身然后用找到的二进制可执行映象的内容来代替其子进程。一般情况下，shell 将等待此命令的完成或者子进程的退出。你可以通过按下 control-Z 键使子进程切换到后台而 shell 再次开始运行。同时还可以使用 shell 命令 bg 将命令放入后台执行，shell 将发送 SIGCONT 信号以重新启动进程直到进程需要进行终端输出和输入。

可执行文件可以有许多格式，甚至是一个脚本文件。脚本文件需要恰当的命令解释器来处理它们；例如 /bin/sh 解释 shell 脚本。可执行目标文件包含可执行代码和数据，这样操作系统可以获得足够的信息将其加载到内存并执行之。Linux 最常用的目标文件是 ELF，但是理论上 Linux 可以灵活地处理几乎所有目标文件格式。

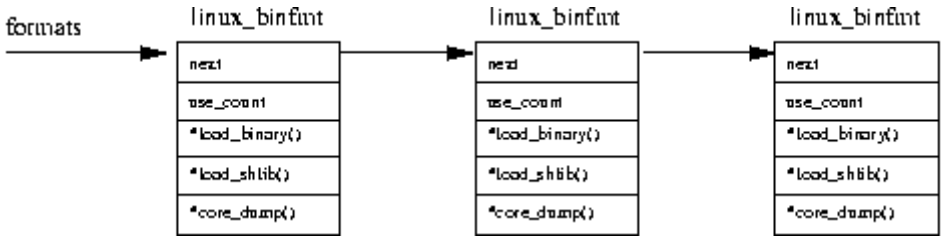


图 4.3 已注册的二进制格式

通过使用文件系统，Linux 所支持的二进制格式既可以构造到核心又可以作为模块加载。核心保存着一个可以支持的二进制格式的链表（见图 4.3），同时当执行一个文件时，各种二进制格式被依次尝试。

Linux 上支持最广的格式是 a.out 和 ELF。执行文件并不需要全部读入内存，而使用一种请求加载技术。进程使用的可执行映象的每一部分被调入内存而没用的部分将从内存中丢弃。

4.8.1 ELF

ELF（可执行与可连接格式）是 Unix 系统实验室设计的一种目标文件格式，现在已成为 Linux 中使用最多的格式。但与其它目标文件格式相比，如 ECOFF 和 a.out，ELF 的开销稍大，它的优点是更加灵活。ELF 可执行文件中包含可执行代码，即正文段：text 和数据段：data。位于可执行映象中的表描叙了程序应如何放入进程的虚拟地址空间中。静态连接映象是通过连接器 ld 得到，在单个映象中包含所有运行此映象所需代码和数据。此映象同时也定义了映象的内存分布和首先被执行的代码的地址。

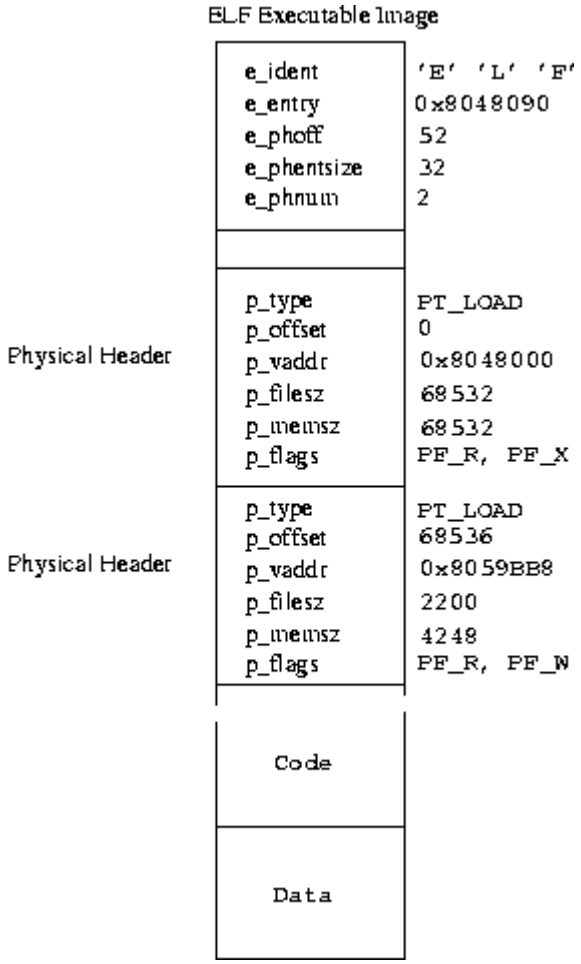


图 4.4 ELF 可执行文件格式

图 4.4 给出了一个静态连接的 ELF 可执行映象的构成。

这是一个打印"Hello World"并退出的简单 C 程序。文件头将其作为一个带两个物理文件头(e_phnum = 2)的 ELF 映象来描叙，物理文件头位于映象文件起始位置 52 字节处。第一个物理文件头描叙的是映象中的可执行代码。它从虚拟地址 0x8048000 开始，长度为 65532 字节。这是因为它包含了 printf()函数库代码以输出"Hello World"的静态连接映象。映象的入口点，即程序的第一条指令，不是位于映象的起始位置而在虚拟地址 0x8048090（e_entry）处。代码正好接着第二个物理文件头。这个物理文件头描叙了此程序使用的数据，它被加载到虚拟内存中 0x8059BB8 处。这些数据是可读并可写的。你可能已经注意到了数据 的大小在文件中是 2200 字节(p_filesz)但是在内存中的大小为 4248 字节。这是因为开始的 2200 字节包含的是预先初始化数据而接下来的 2048 字节包含的是被执行代码初始化的数据。

当 Linux 将一个 ELF 可执行映象加载到进程的虚拟地址空间时，它并不真的加载映象。首先它建立其虚拟内存数据结构：进程的 vm_area_struct 树和页表。当程序执行时将产生页面错，引起程序代码和数据从物理内存中取出。程序中没有使用到的部分从来都不会加载到内存中去。一旦 ELF 二进制格式加载器发现这个映象是有效的 ELF 可执行映象，它将把进程的当前可执行映象从虚拟内存冲刷出去。当进程是一个复制映象时（所有的进程都是），父进程执行的是老的映象程序，例如象 bash 这样的命令解释器。同时还将清除任

何信号处理过程并且关闭打开的文件，在冲刷的最后，进程已经为新的可执行映象作好了准备。不管可执行映象是哪种格式，进程的 mm_struct 结构中将存入相同信息，它们是指向映象代码和数据的指针。当 ELF 可执行映象从文件中读出且相关程序代码被映射到进程虚拟地址空间后，这些指针的值都被确定下来。同时 vm_area_struct 也被建立起来，进程的页表也被修改。mm_struct 结构中还包含传递给程序和进程环境变量的参数的指针。

ELF 共享库

另一方面，动态连接映象并不包含全部运行所需要的代码和数据。其中的一部分仅在运行时才连接到共享库中。ELF 共享库列表还在运行时连接到共享库时被动态连接器使用。Linux 使用几个动态连接器，如 ld.so.1，libc.so.1 和 ld-linux.so.1，这些都放置在/lib 目录中。这些库中包含常用代码，如 C 语言子程序等。如果没有动态连接，所有程序将不得不将所有库过程拷贝一份并连接进来，这样将需要更多的磁盘与虚拟内存空间。通过动态连接，每个被引用库过程的信息都可以包含在 ELF 映象列表中。这些信息用来帮助动态连接器定位库过程并将它连入程序的地址空间。

4.8.2 脚本文件

脚本文件的运行需要解释器的帮助。Linux 中有许许多多的解释器；例如 wish、perl 以及命令外壳程序 tcsh。 Linux 使用标准的 Unix 规则，在脚本文件的第一行包含了脚本解释器的名字。典型的脚本文件的开头如下：

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

此时命令解释器会试着寻找脚本解释器。 然后它打开此脚本解释器的执行文件得到一个指向此文件的 VFS inode 并使此脚本解释器开始解释此脚本。这时脚本文件名变成了脚本解释器的 0 号参数(第一个参数)并且其余参数向上挪一个位置(原来的第一个参数变成第二个)。脚本解释器的加载过程与其他可执行文件相同。Linux 会逐个尝试各种二进制可执行格式直到它可以执行。

第五章 进程间通讯机制

进程在核心的协调下进行相互间的通讯。Linux 支持大量进程间通讯(IPC)机制。除了信号和管道外，Linux 还支持 Unix 系统 V 中的 IPC 机制。

5.1 信号

信号是 Unix 系统中的最古老的进程间通讯方式。它们用来向一个或多个进程发送异步事件信号。信号可以从键盘中断中产生，另外进程对虚拟内存的非法存取等系统错误环境下也会有信号产生。信号还被 shell 程序用来向其子进程发送任务控制命令。

系统中有一组被详细定义的信号类型，这些信号可以由核心或者系统中其它具有适当

权限的进程产生。使用 kill 命令(kill -l)可以列出系统中所有已经定义的信号。在我的系统(Intel 系统)上运行结果如下：

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

当我在 Alpha AXP 中运行此命令时，得到了不同的信号个数。除了两个信号外，进程可以忽略这些信号中的绝大部分。其一是引起进程终止执行的 SIGSTOP 信号，另一个是引起进程退出的 SIGKILL 信号。 至于其它信号，进程可以选择处理它们的具体方式。进程可以阻塞信号，如若不阻塞，则可以在自行处理此信号和将其转交核心处理之间作出选择。如果由核心来处理此信号，它将使用对应此信号的缺省处理方法。 比如当进程接收到 SIGFPE(浮点数异常)时，核心的缺省操作是引起 core dump 和进程的退出。信号没有固有的相对优先级。如果在同一时刻对于一个进程产生了两个信号，则它们将可能以任意顺序到达进程并进行处理。同时 Linux 并不提供处理多个相同类型信号的方式。即进程无法区分它是收到了 1 个还是 42 个 SIGCONT 信号。

Linux 通过存储在进程 task_struct 中的信息来实现信号。信号个数受到处理器字长的限制。32 位字长的处理器最多可以有 32 个信号而 64 位处理器如 Alpha AXP 可以有最多 64 个信号。当前未处理的信号保存在 signal 域中，并带有保存在 blocked 中的被阻塞信号的屏蔽码。除了 SIGSTOP 和 SIGKILL 外，所有的信号都能被阻塞。当产生可阻塞信号时，此信号可以保持一直处于待处理状态直到阻塞释放。Linux 保存着每个进程处理每个可能信号的信息，它们保存在每个进程 task_struct 中的 sigaction 数组中。这些信息包括进程希望处理的信号所对应的过程地址，或者指示是忽略信号还是由核心来处理它的标记。通过系统调用，进程可以修改缺省的信号处理过程，这将改变某个信号的 sigaction 以及阻塞屏蔽码。

并不是系统中每个进程都可以向所有其它进程发送信号：只有核心和超级用户具有此权限。普通进程只能向具有相同 uid 和 gid 的进程或者在同一进程组中的进程发送信号。信号是通过设置 task_struct 结构中 signal 域里的某一位来产生的。如果进程没有阻塞信号并且处于可中断的等待状态，则可以将其状态改成 Running，同时如确认进程还处在运行队列中，就可以通过信号唤醒它。这样系统下次发生调度时，调度管理器将选择它运行。如果进程需要缺省的信号处理过程，则 Linux 可以优化对此信号的处理。例如 SIGWINCH（X 窗口的焦点改变）信号，其缺省处理过程是什么也不做。

信号并非一产生就立刻交给进程，而是必须等待到进程再次运行时才交给进程。每次进程从系统调用中退出前，它都会检查 signal 和 blocked 域，看是否有可以立刻发送的非阻塞信号。这看起来非常不可靠，但是系统中每个进程都在不停地进行系统调用，如向终端输出字符。当然进程可以选择去等待信号，此时进程将一直处于可中断状态直到信号出现。对当前不可阻塞信号的处理代码放置在 sigaction 结构中。

如果信号的处理过程被设置成缺省则由核心来应付它。SIGSTOP 信号的缺省处理过程是将当前进程的状态改变成为 Stopped 并运行调度管理器以选择一个新进程继续运行。

SIGFPE 的缺省处理过程则是引起 core dump 并使进程退出。当然，进程可以定义其自身的信号处理过程。一旦信号产生，这个过程就将被调用。它的地址存储在 sigaction 结构中。核心必须调用进程的信号处理例程，具体如何去做依赖于处理器类型，但是所有的 CPU 必须处理这个问题：如果信号产生时，当前进程正在核心模式下运行并且马上要返回调用核心或者系统例程的进程，而该进程处在用户模式下。解决这个问题需要操纵进程的堆栈及寄存器。进程的程序计数器被设置成其信号处理过程的地址，而参数通过调用框架或者寄存器传递到处理例程中。当进程继续执行时，信号处理例程好象普通的函数调用一样。

Linux 是 POSIX 兼容的，所以当某个特定信号处理例程被调用时，进程可以设定哪个信号可以阻塞。这意味着可以在进程信号处理过程中改变 blocked 屏蔽码。当信号处理例程结束时，此 blocked 屏蔽码必须设置成原有值。 因此，Linux 添加了一个过程调用来进行整理工作，通过它来重新设置被发送信号进程调用栈中的原有 blocked 屏蔽码。 对于同一时刻几个信号处理过程，Linux 通过堆栈方式来优化其使用，每当一个处理过程退出时，下一个处理过程必须等到整理例程结束后才执行。

5.2 管道

一般的 Linux shell 程序都允许重定向。如

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

在这个管道应用中，ls 列当前目录的输出被作为标准输入送到 pr 程序中，而 pr 的输出又被作为标准输入送到 lpr 程序中。管道是单向的字节流，它将某个进程的标准输出连接到另外进程的标准输入。但是使用管道的进程都不会意识到重定向的存在，并且其执行结果也不会有什么不同。shell 程序负责在进程间建立临时的管道。

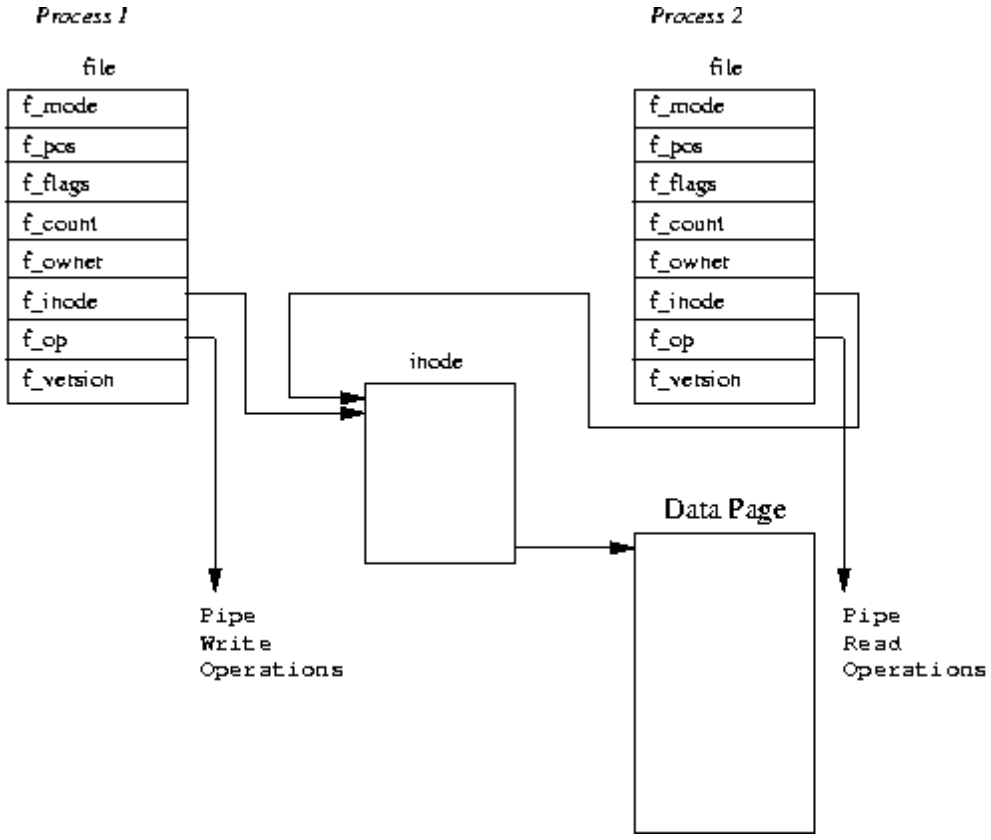


图 5.1 管道

在 Linux 中，管道是通过指向同一个临时 VFS inode 的两个 file 数据结构来实现的，此 VFS inode 指向内存中的一个物理页面。图 5.1 中每个 file 数据结构指向不同的文件操作例程向量，一个是实现对管道的写，另一个从管道中读。

这样就隐藏了读写管道和读写普通的文件时系统调用的差别。当写入进程对管道写时，字节被拷贝到共享数据页面中，当读取进程从管道中读时，字节从共享数据页面中拷贝出来。Linux 必须同步对管道的访问。它必须保证读者和写者以确定的步骤执行，为此需要使用锁、等待队列和信号等同步机制。

当写者想对管道写入时，它使用标准的写库函数。表示打开文件和打开管道的描叙符用来对进程的 file 数据 结构集合进行索引。Linux 系统调用使用由管道 file 数据结构指向的 write 过程。这个 write 过程用保存在表示管道的 VFS inode 中的信息来管理写请求。

如果没有足够的空间容纳对所有写入管道的数据，只要管道没有被读者加锁。则 Linux 为写者加锁，并把从写入进程地址空间中写入的字节拷贝到共享数据页面中去。如果管道被读者加锁或者没有足够空间存储数据，当前进程将在管道 inode 的等待队列中睡眠，同时调度管理器开始执行以选择其它进程来执行。如果写入进程是可中断的，则当有足够的空间或者管道被解锁时，它将被读者唤醒。当数据被写入时，管道的 VFS inode 被解锁，同时任何在此 inode 的等待队列上睡眠的读者进程都将被唤醒。

从管道中读出数据的过程和写入类似。

进程允许进行非阻塞读（这依赖于它们打开文件或者管道的方式），此时如果没有数据可读或者管道被加锁， 则返回错误信息表明进程可以继续执行。阻塞方式则使读者进程在

管道 inode 的等待队列上睡眠直到写者 进程结束。当两个进程对管道的使用结束时，管道 inode 和共享数据页面将同时被遗弃。

Linux 还支持命名管道(named pipe)，也就是 FIFO 管道，因为它总是按照先进先出的原则工作。第一个被写入 的数据将首先从管道中读出来。和其它管道不一样，FIFO 管道不是临时对象，它们是文件系统中的实体并且 可以通过 mkfifo 命令来创建。进程只要拥有适当的权限就可以自由使用 FIFO 管道。打开 FIFO 管道的方式稍有不同。其它管道需要先创建（它的两个 file 数据结构，VFS inode 和共享数据页面）而 FIFO 管道已经存在，只需要由使用者打开与关闭。在写者进程打开它之前，Linux 必须让读者进程先打开此 FIFO 管道；任何读者进程从中读取之前必须有写者进程向其写入数据。FIFO 管道的使用方法与普通管道基本相同，同时它们使用相同数据结构和操作。

5.3 套接口

5.3.1 系统 V IPC 机制

Linux 支持 Unix 系统 V（1983）版本中的三种进程间通讯机制。它们是消息队列、信号灯以及共享内存。这些系统 V IPC 机制使用共同的授权方法。只有通过系统调用将标志符传递给核心之后，进程才能存取这些资源。这些系统 V IPC 对象使用与文件系统非常类似的访问控制方式。对象的引用标志符被用来作为资源表中的索引。这个索引值需要一些处理后才能得到。

系统中所有系统 V IPC 对象的 Linux 数据结构包含一个 ipc_perm 结构，它含有进程拥有者和创建者及组标志符。另外还有对此对象（拥有者，组及其它）的存取模式以及 IPC 对象键。此键值被用来定位系统 V IPC 对象的引用标志符。这样的键值一共有两组：公有与私有。如果此键为公有，则系统中任何接受权限检查的进程都可以找到系统 V IPC 对象的引用标志符。系统 V IPC 对象绝不能用一个键值来引用，而只能使用引用标志符。

5.3.2 消息队列

消息队列允许一个或者多个进程向它写入与读取消息。Linux 维护着一个 msgque 消息队列链表，其中每个元素 指向一个描叙消息队列的 msqid_ds 结构。当创建新的消息队列时，系统将从系统内存中分配一个 msqid_ds 结构，同时将其插入到数组中。

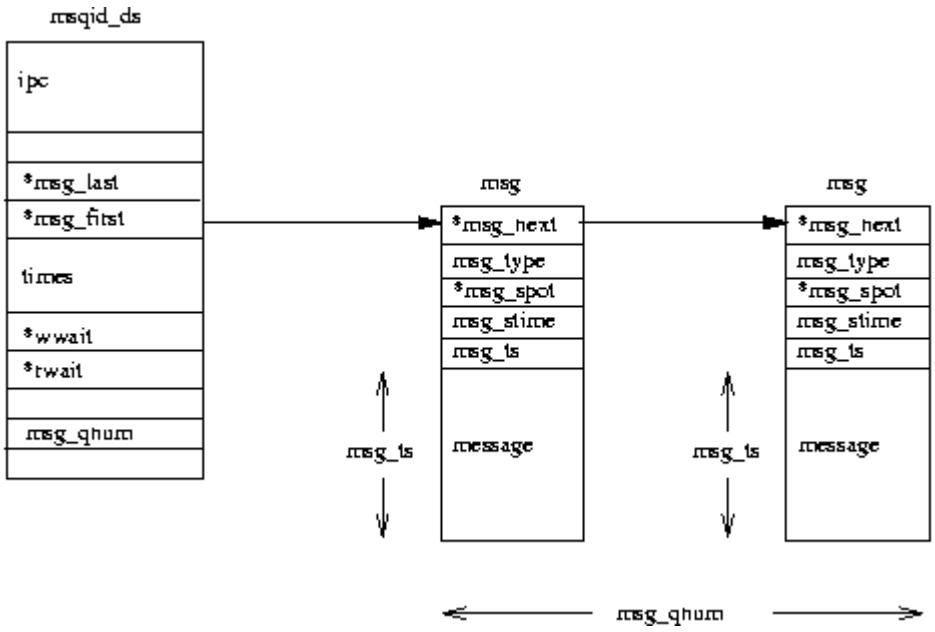


图 5.2 系统 V IPC 消息队列

每个 msqid_ds 结构包含一个 ipc_perm 结构和指向已经进入此队列消息的指针。另外，Linux 保留有关队列修改时间信息，如上次系统向队列中写入的时间等。msqid_ds 包含两个等待队列：一个为队列写入进程使用而另一个由队列读取进程使用。

每次进程试图向写入队列写入消息时，系统将把其有效用户和组标志符与此队列的 ipc_perm 结构中的模式进行比较。如果允许写入操作，则把此消息从此进程的地址空间拷贝到 msg 数据结构中，并放置到此消息队列尾部。由于 Linux 严格限制可写入消息的个数和长度，队列中可能容纳不下这个消息。此时，此写入进程将被添加到这个消息队列的等待队列中，同时调用调度管理器选择新进程运行。当由消息从此队列中释放时，该进程将被唤醒。

从队列中读的过程与之类似。进程对这个写入队列的访问权限将被再次检验。读取进程将选择队列中第一个消息（不管是什么类型）或者第一个某特定类型的消息。如果没有消息可以满足此要求，读取进程将被添加 到消息队列的读取等待队列中，然后系统运行调度管理器。当有新消息写入队列时，进程将被唤醒继续执行。

5.3.3 信号灯

信号灯最简单的形式是某个可以被多个进程检验和设置(test&set)的内存单元。这个检验与设置操作对每个进程而言是不可中断或者说是一个原子性操作；一旦启动谁也终止不了。检验与设置操作的结果是信号灯当前值加 1， 这个值可以是正数也可以是负数。根据这个操作的结果，进程可能可以一直睡眠到此信号灯的值被另一个进程更改为止。信号灯可用来实现临界区(critical region)：某一时刻在此区域内的代码只能被一个进程执行。

如果你有多个协作进程从一个数据文件中读取与写入记录。有时你可能需要这些文件访问遵循严格的访问次序。 那么可在文件操作代码上使用一个初始值为 1 的信号灯，它带有两个信号灯操作，一个检验并对信号灯 值减 1，而另一个检验并加 1。第一个访问文件

的进程将试图将信号灯值减 1，如果获得成功则信号灯值变成了 0。此进程于是开始使用这个数据文件，但是此时如果另一进程也想将信号灯值减 1，则信号灯值将为-1，这次操作将会失败。它将挂起执行直到第一个进程完成对此数据文件的使用。此时这个等待进程将被唤醒，这次它对信号灯的操作将成功。

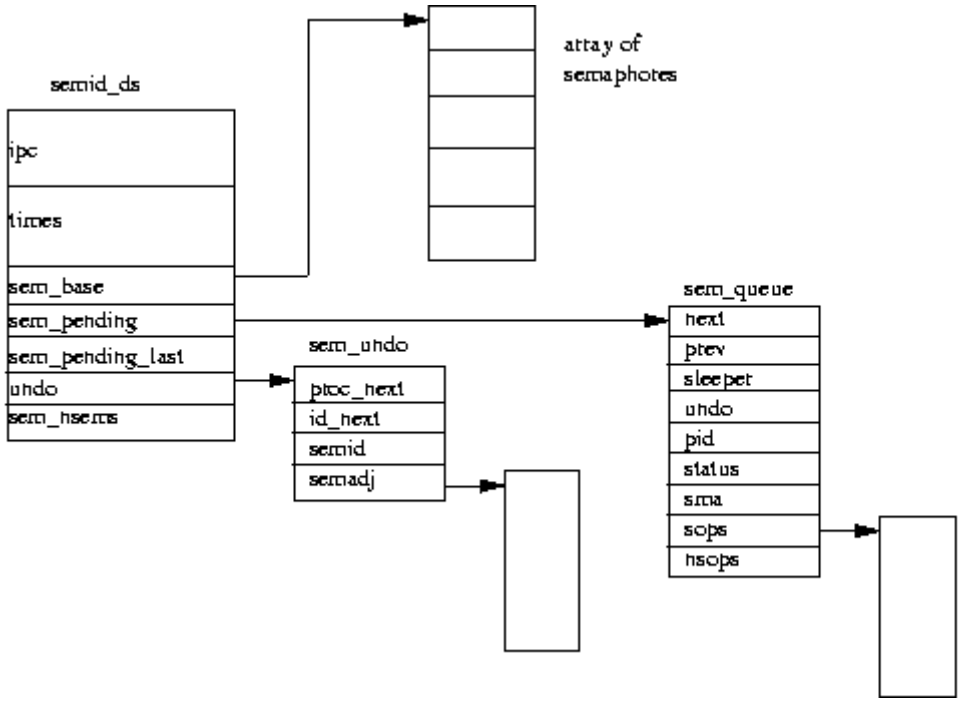


图 5.3 系统 V IPC 信号灯

每个系统 V IPC 信号灯对象对应一个信号灯数组，Linux 使用 semid_ds 结构来表示。系统中所有 semid_ds 结构由一组 semary 指针来指示。在每个信号灯数组中有一个 sem_nsems，它表示一个由 sem_base 指向的 sem 结构。授权的进程可以使用系统调用来操纵这些包含系统 V IPC 信号灯对象的信号灯数组。这个系统调用可以定义许多种操作，每个操作用三个输入来描叙：信号灯索引、操作值和一组标志。信号灯索引是一个信号灯数组的索引，而操作值是将被加到信号灯上的数值。首先 Linux 将检查是否所有操作已经成功。如果操作值与信号灯当前数值相加大于 0，或者操作值与信号灯当前值都是 0，操作将会成功。如果所有信号灯操作失败，Linux 仅仅会把那些操作标志没有要求系统调用为非阻塞类型的进程挂起。进程挂起后，Linux 必须保存信号灯操作的执行状态并将当前进程放入等待队列。系统还在堆栈上建立 sem_queue 结构并填充各个域。这个 sem_queue 结构将被放到此信号灯对象等待队列的尾部（使用 sem_pending 和 sem_pending_last 指针）。系统把当前进程置入 sem_queue 结构中的等待队列(sleeper)中，然后启动调度管理器选择其它进程运行。

如果所有这些信号灯操作都成功则无需挂起当前进程，Linux 将对信号灯数组中的其他成员进行相同操作，然后检查那些处于等待或者挂起状态的进程。首先，Linux 将依次检查挂起队列(sem_pending) 中的每个成员，看信号灯操作能否继续。如果可以则将其 sem_queue 结构从挂起链表中删除并对信号灯数组发出信号灯操作。Linux 还将唤醒处于睡眠状态的进程并使之成为下一个运行的进程。如果在对挂起队列的遍历过程中有的信号灯操作不能完成则 Linux 将一直重复此过程，直到所有信号灯操作完成且没有进程需要继续

睡眠。

但是信号灯的使用可能产生一个严重的问题：死锁。当一个进程进入临界区时它改变了信号灯的值而离开临界区时由于运行失败或者被 kill 而没有改回信号灯时，死锁将会发生。Linux 通过维护一组描叙信号灯数组变化的链表来防止该现象的发生。它的具体做法是让 Linux 将把此信号灯设置为进程对其进行操作前的状态。这些状态值被保存在使用该信号灯数组进程的 semid_ds 和 task_struct 结构的 sem_undo 结构中。

信号灯操作将迫使系统对它引起的状态变化进行维护。Linux 为每个进程维护至少一个对应于信号灯数组的 sem_undo 结构。如果请求进行信号灯操作的进程没有该结构，则必要时 Linux 会为其创建一个。这个 sem_undo 结构将同时放入此进程的 task_struct 结构和此信号灯数组的 semid_ds 结构中。当对信号灯进行操作时，信号灯变化值的负数被置入进程的 sem_undo 结构中该信号的入口中。所以当操作值为 2 时，则此信号灯的调整入口中将加入一个-2。

象正常退出一样，当进程被删除时，Linux 将遍历该进程的 sem_undo 集合对信号灯数组使用调整值。如果信号灯集合被删除而 sem_undo 数据结构还在进程的 task_struct 结构中则此信号灯数组标志符将被置为无效。此时 信号灯清除代码只需丢弃 sem_undo 结构即可。

5.3.4 共享内存

共享内存允许一个或多个进程通过同时出现在它们虚拟地址空间中的内存来通讯。此虚拟内存的页面出现在每个共享进程页表中。但此页面并不一定位于所有共享进程虚拟内存的相同位置。和其它系统 V IPC 对象的使用方法一样，对共享内存区域的访问是通过键和访问权限检验来控制的。一旦内存被共享，则再不会检验进程对对象的使用方式。它依赖于其它机制，如系统 V 信号灯，来同步对共享内存的访问。

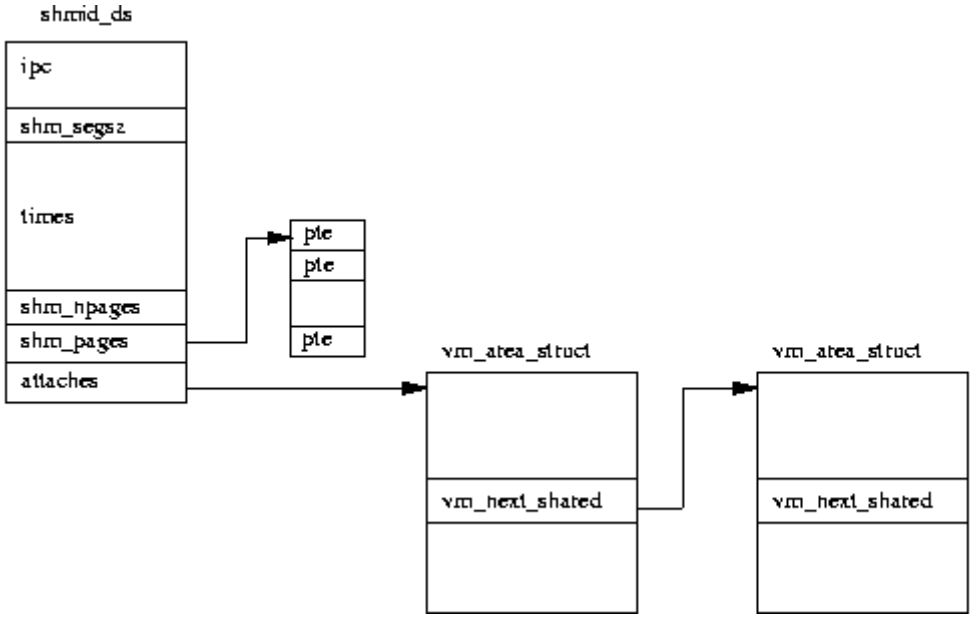


图 5.4 系统 V IPC 共享内存

每个新创建的共享内存区域由一个 shmid_ds 数据结构来表示。它们被保存再 shm_segs 数组中。 shmid_ds 数据结构描叙共享内存的大小，进程如何使用以及共享内存映射到其

各自地址空间的方式。由共享内存创建者控制对此内存的存取权限以及其键是公有还是私有。如果它由足够权限，它还可以将此共享内存加载到物理内存中。

每个使用此共享内存的进程必须通过系统调用将其连接到虚拟内存上。这时进程创建新的 vm_area_struct 来描叙此共享内存。进程可以决定此共享内存在其虚拟地址空间的位置，或者让 Linux 选择一块足够大的区域。 新的 vm_area_struct 结构将被放到由 shmid_ds 指向的 vm_area_struct 链表中。通过 vm_next_shared 和 vm_prev_shared 指针将它们连接起来。虚拟内存在连接时并没有创建；进程访问它时才创建。

当进程首次访问共享虚拟内存中的页面时将产生页面错。当取回此页面后，Linux 找到了描叙此页面的 vm_area_struct 数据结构。它包含指向使用此种类型虚拟内存的处理函数地址指针。共享内存页面错误处理 代码将在此 shmid_ds 对应的页表入口链表中寻找是否存在此共享虚拟内存页面。如果不存在，则它将分配物理页面并为其创建页表入口。同时还将它放入当前进程的页表中，此入口被保存在 shmid_ds 结构中。这意味着下个试图访问此内存的进程还会产生页面错误，共享内存错误处理函数将为此进程使用其新创建的物理页面。这样，第一个访问虚拟内存页面的进程创建这块内存，随后的进程把此页面加入到各自的虚拟地址空间中。

当进程不再共享此虚拟内存时，进程和共享内存的连接将被断开。如果其它进程还在使用这个内存，则此操作只影响当前进程。其对应的 vm_area_struct 结构将从 shmid_ds 结构中删除并回收。当前进程对应此共享内存地址的页表入口也将被更新并置为无效。当最后一个进程断开与共享内存的连接时，当前位于物理内存中的共享内存页面将被释放，同时还有此共享内存的 shmid_ds 结构。

当共享内存没有被锁入物理内存时，情况将更加复杂。此时共享内存页面可能会在内存使用高峰期，被交换到系统的交换磁盘上。共享内存如何被交换与调入物理内存将在 mm 一章中详细描叙。

第六章 PCI

外围设备互连(PCI)是一种将系统中外部设备以结构化与可控制方式连接到起来的总线标准，包括系统部件连接的电气特性及行为。本章将详细讨论 Linux 核心对系统中的 PCI 总线与设备的初始化过程。

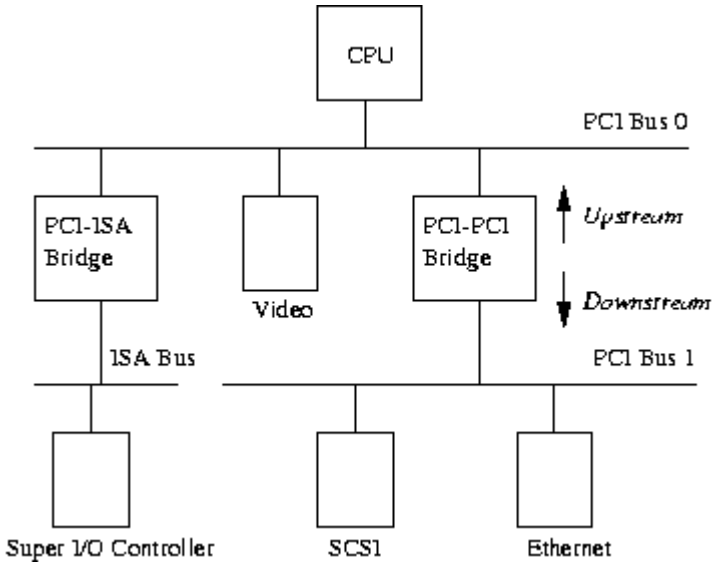


图 6.1 一个基于 PCI 的系统示意图

图 6.1 是一个基于 PCI 的系统示意图。PCI 总线和 PCI-PCI 桥接器在连接系统中设备到上起关键作用，在这个系统中 CPU 和视频设备被连到 PCI bus 0 上，它是系统中的主干 PCI 总线。而 PCI-PCI 桥接器这个特殊 PCI 设备将主干总线 PCI bus 0 与下级总线 PCI bus 1 连接到一起。PCI 标准术语中，PCI bus 1 是 PCI-PCI 桥接器的 downstream 而 PCI bus 0 是此桥接器的 up-stream。SCSI 和以太网设备通过二级 PCI 总线连接到这个系统中。而在物理实现上，桥接器和二级 PCI 总线被集成到一块 PCI 卡上。而 PCI-ISA 桥接器用来支持古老的 ISA 设备，图中有一个高级 I/O 控制芯片来控制键盘、鼠标及软盘设备。

6.1 PCI 地址空间

CPU 和 PCI 设备需要存取在它们之间共享的内存空间。这块内存区域被设备驱动用来控制 PCI 设备并在 CPU 与 PCI 设备之间传递信息。最典型的共享内存包括设备的控制与状态寄存器。这些寄存器用来控制设备并读取其 信息。例如 PCI SCSI 设备驱动可以通过读取其状态寄存器，找出已准备好将一块数据写入 SCSI 磁盘的 SCSI 设备。同时还可以在设备加电后，通过对控制寄存器写入信息来启动设备。

CPU 的系统内存可以被用作这种共享内存，但是如果采用这种方式，则每次 PCI 设备访问此内存块时，CPU 将被迫停止工作以等待 PCI 设备完成此操作。这种方式将共享内存限制成每次只允许一个系统设备访问。该策略会大大降低系统性能。但如果允许系统外设不受限制地访问主存也不是好办法。它的危险之处在于一个有恶意行为的设备将使整个系统置于不稳定状态。

外设有其自身的内存空间。CPU 可以自由存取此空间，但设备对系统主存的访问将处于 DMA（直接内存访问）通道的严格控制下。ISA 设备需要存取两个地址空间：ISA I/O（输入输出）和 ISA 内存。而 PCI 设备需要访问三种地址空间：PCI I/O、PCI 内存和 PCI 配置空间。CPU 则可以访问所有这些地址空间。PCI I/O 和 PCI 内存由设备驱动程序使用而 PCI 配置空间被 Linux 核心中的 PCI 初始化代码使用。

Alpha AXP 处理器并不能象访问系统地址空间那样随意访问这些地址空间，它只能通过辅助芯片组来存取这些 地址空间，如 PCI 配置空间。Alpha AXP 处理器使用稀疏地址映射策略来从系统巨大的虚拟内存中"窃取"一部分并将其映射到 PCI 地址空间。

6.2 PCI 配置头

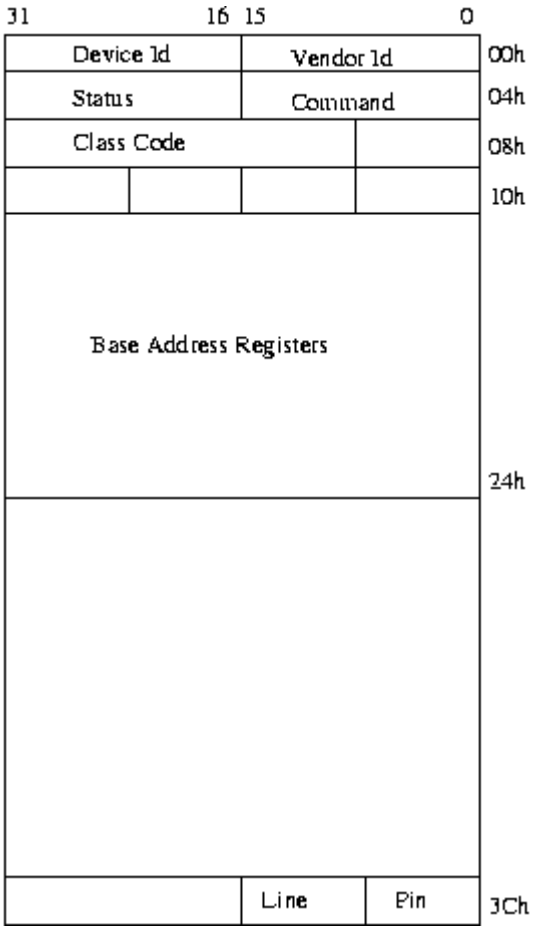


图 6.2 PCI 配置头

系统中每个 PCI 设备，包括 PCI-PCI 桥接器在内，都有一个配置数据结构，它通常位于 PCI 配置地址空间中。PCI 配置头允许系统来标识与控制设备。配置头在 PCI 配置空间的位置取决于系统中 PCI 设备的拓扑结构。例如将一个 PCI 视频卡插入不同的 PCI 槽，其配置头位置会变化。但对系统没什么影响，系统将找到每个 PCI 设备与桥接器并使用它们配置头中的信息来配置其寄存器。

典型的办法是用 PCI 槽相对主板的位置来决定其 PCI 配置头在配置空间中的偏移。比如主板中的第一个 PCI 槽的 PCI 配置头位于配置空间偏移 0 处，而第二个则位于偏移 256 处（所有 PCI 配置头长度都相等，为 256 字节），其它槽可以由此类推。系统还将提供一种硬件相关机制以便 PCI 设置代码能正确的辨认出对应 PCI 总线上所有存在的设备的 PCI 配置头。通过 PCI 配置头中的某些域来判断哪些设备存在及哪些设备不存在（这个域叫厂

商标志域：Vendor Identification field）。对空 PCI 槽中这个域的读操作将得到一个值为 0xFFFFFFFF 的错误信息。

图 6.2 给出了 256 字节 PCI 配置头的结构，它包含以下域：

厂商标识(Vendor Identification)

用来唯一标识 PCI 设备生产厂家的数值。Digital 的 PCI 厂商标识为 0x1011 而 Intel 的为 0x8086。

设备标识(Device Identification)

用来唯一标识设备的数值。Digital 21141 快速以太设备的设备标识为 0x0009。

状态(Status)

此域提供 PCI 标准定义中此设备的状态信息。

命令(Command)

通过对此域的写可以控制此设备，如允许设备访问 PCI I/O 内存。

分类代码(Class Code)

此域标识本设备的类型。对于每种类型的视频，SCSI 等设备都有标准的分类代码。如 SCSI 设备分类代码为 0x0100。

基地址寄存器(Base Address Registers)

这些寄存器用来决定和分配此设备可以使用的 PCI I/O 与 PCI 内存空间的类型，数量及位置。

中断引脚(Interrupt Pin)

PCI 卡上的四个物理引脚可以将中断信号从插卡上带到 PCI 总线上。这四个引脚标准的标记分别为 A、B、C 及 D。中断引脚域描叙此 PCI 设备使用的引脚号。通常特定设备都是采用硬连接方式。这也是系统启动时，设备总使用相同中断引脚的原因。中断处理子系统用它来管理来自该设备的中断。

中断连线(Interrupt Line)

本设备配置头中的中断连线域用来在 PCI 初始化代码、设备驱动以及 Linux 中断处理子系统间传递中断处理过程。虽然本域中记录的这个数值对于设备驱动毫无意义。但是它可以将中断处理过程从 PCI 卡上正确路由到 Linux 操作系统中相应的设备驱动中断处理代码中。在 interrupt 一章中将详细描叙 Linux 中断处理过程。

6.3 PCI I/O 和 PCI 内存地址

这两个地址空间用来实现 PCI 设备和 Linux 核心中设备驱动程序之间的通讯。例如 DEC21141 快速以太网设备的内部寄存器被映射到 PIC I/O 空间上时，其对应的 Linux 设备驱动可以通过对这些寄存器的读写来控制此设备。PCI 视频卡通常使用大量的 PCI 内存空间来存储视频信息。

在 PCI 系统建立并通过用 PCI 配置头中的命令域来打开这些地址空间前，系统决不允许对它们进行存取。值得注意的是只有 PCI 配置代码读取和写入 PCI 配置空间，Linux 设备驱动只读写 PCI I/O 和 PCI 内存地址。

6.4 PCI-ISA 桥接器

这种桥接器通过将 PCI I/O 和 PCI 内存空间的存取转换成对 ISA I/O 和 ISA 内存的存取来支持古老的 ISA 设备。市场上许多主板中同时包含几个 ISA 总线槽和 PCI 槽。但今后对 ISA 设备的向后兼容支持将逐渐减弱，最终主板上只会有 PCI 槽。早期的 Intel 8080 PC 就将 ISA 设备的 ISA 地址空间固定了下来。即使在价值 5000 美圆的 Alpha AXP 系统中其 ISA 软盘控制器地址也和最早 IBM PC 上的相同。PCI 标准将 PCI I/O 和 PCI 内存的低端部分保留给系统中的 ISA 外设，另外还使用 PCI-ISA 桥接器实现从 PCI 内存访问到 ISA 内存访问的转换。

6.5 PCI-PCI 桥接器

PCI-PCI 桥接器是一种将系统中所有 PCI 总线连接起来的特殊 PCI 设备。在简单系统中只存在一条 PCI 总线，由于受电气特性的限制，它所连接的 PCI 设备个数有限。引入 PCI-PCI 桥接器后系统可以使用更多的 PCI 设备。对于高性能服务器这是非常重要的。Linux 提供了对 PCI-PCI 桥接器的全面支持。

6.5.1 PCI-PCI 桥接器：PCI I/O 和 PCI 内存窗口

PCI-PCI 桥接器将 PCI I/O 和 PCI 内存读写请求中的一个子集向下传送。例如在图 6.1 中，如果来自 PCI 总线 0 请求是对 SCSI 或以太设备所拥有的 PCI I/O 或 PCI 内存的读写，则此 PCI-PCI 桥接器将只需把请求从总线 0 传递到 PCI 总线 1 上；所有其它 PCI I/O 和内存地址都将被它忽略。这个过滤使得这些地址信息不会在整个系统中扩散。为了实现这点，PCI-PCI 桥接器必须编程为有某个 PCI I/O 及 PCI 内存基址和上限，只有在这个地址范围内的 PCI 地址访问才能从主干总线传递到二级总线。一旦系统中的 PCI-PCI 桥接器被设置成这样，则只要当 Linux 设备驱动程序通过这个窗口访问 PCI I/O 和 PCI 内存空间时，此 PCI-PCI 桥接器就将变得透明。这样也给 Linux PCI 设备驱动编写者提供了方便。我们在稍后的讨论中将看到 Linux 对 PCI-PCI 桥接器非常巧妙的配置。

6.5.2 PCI-PCI 桥接器：PCI 配置循环及 PCI 总线编号方式

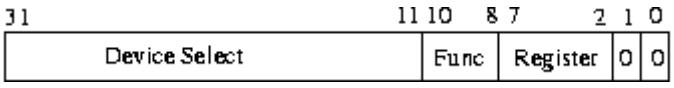


图 6.3 0 类型 PCI 配置循环

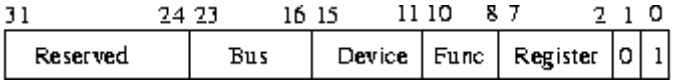


图 6.4 1 类型 PCI 配置循环

为了让 CPU 上运行的 PCI 初始化代码能访问位于分支 PCI 总线上的设备，必须为桥接器提供某种机制以便它可以决定是否将配置循环从主干接口传递到其二级接口。循环是出现在 PCI 总线上的一个地址。PCI 标准定义了两种 PCI 配置寻址格式；类型 0 和类型 1；它们分别如图 6.3 及 6.4 所示。类型 0 PCI 配置循环不包含总线序号,同时在此 PCI 总线上对应于这个 PCI 配置地址的所有 PCI 设备都会来对它们进行解释。类型 0 配置循环的 11 位到 31 位用来进行 PCI 设备选择。有种设计方式是让每位代表系统中一个不同的设备。这时 11 位对应 PCI 槽 0 中的 PCI 设备而 12 位标识槽 1 中的设备等等，如此类推。另外一种方式是直接将设备的槽号写入到位 31 到 11 中。系统使用哪种机制依赖于系统 PCI 内存控制器。

类型 1 PCI 配置循环包含一个 PCI 总线序号,同时这种配置循环将被除桥接器外的所有 PCI 设备所忽略。所有发现类型 1 配置循环的 PCI-PCI 桥接器把它们看到的地址传递到各自的下级 PCI 总线。至于 PCI-PCI 桥接器是否忽略类型 1 配置循环或将其传递到 PCI 总线则依赖于 PCI-PCI 桥接器的配置方式。每个 PCI-PCI 桥接器都拥有一个主干总线接口序号以及一个二级总线接口序号。主干总线是那个离 CPU 最近的 PCI 总线而二级总线是离它稍远的 PCI 总线。任何 PCI-PCI 桥接器还包含一个从属总线序号，这是所有二级总线接口所桥接的 PCI 总线中序号最大的那个。或者说这个从属总线序号是 PCI-PCI 桥接器向下连接中 PCI 总线的最大序号。当 PCI-PCI 桥接器看到类型 1 PCI 配置循环时它将进行如下操作：

如果此总线序号不在桥接器的二级总线序号和从属总线序号之间则忽略掉它。

如果此总线序号与桥接器的二级总线序号相同则将其转换成类型 0 配置命令。

如果此总线序号位于桥接器的二级总线序号与从属总线序号之间则将它不作改变的传递到二级总线接口中。

所以如果想寻址 PCI-PCI 配置例 4 中总线 3 上的设备 1，我们继续从 CPU 中产生一个类型 1 配置命令。桥接器 1 将其传递给总线 1。桥接器 2 虽然忽略它但会将其转换成一个类型 0 配置命令并送到总线 3 上，在那里设备 1 将作出相应反应。

PCI 配置中总线序号由操作系统来分配。但是序号分配策略必须遵循对系统中所有 PCI-PCI 桥接器都正确的描叙：

“位于 PCI-PCI 桥接器后所有的 PCI 总线必须位于二级总线序号和从属总线序号之间”。

如果这个规则被打破，则 PCI-PCI 桥接器将不能正确的传递与转换类型 1 PCI 配置循环,同时系统将找不到或者不能正确地初始化系统中的 PCI 设备。为了满足这个序号分配策略，Linux 以特殊的顺序配置这些特殊的设备。PCI-PCI 总线序号分配一节详细描叙了 Linux 的 PCI 桥接器与总线序号分配策略。

6.6 Linux PCI 初始化过程

Linux 中的 PCI 初始化代码逻辑上可分成三个部分：

PCI 设备驱动

这个伪设备驱动程序将从总线 0 开始搜索 PCI 系统并定位系统中所有的 PCI 设备与桥接器。它将建立起一个描叙系统拓扑结构的数据结构链表。另外它还为所有的桥接器进行编号。

PCI BIOS

这个软件层提供了在 bib-pci-bios 定义中描叙的服务。即使 Alpha AXP 没有 BIOS 服务，Linux 核心也将为它提供具有相同功能的代码。

PCI Fixup

系统相关补丁代码将整理 PCI 初始化最后阶段的一些系统相关事物。

6.6.1 Linux 核心 PCI 数据结构

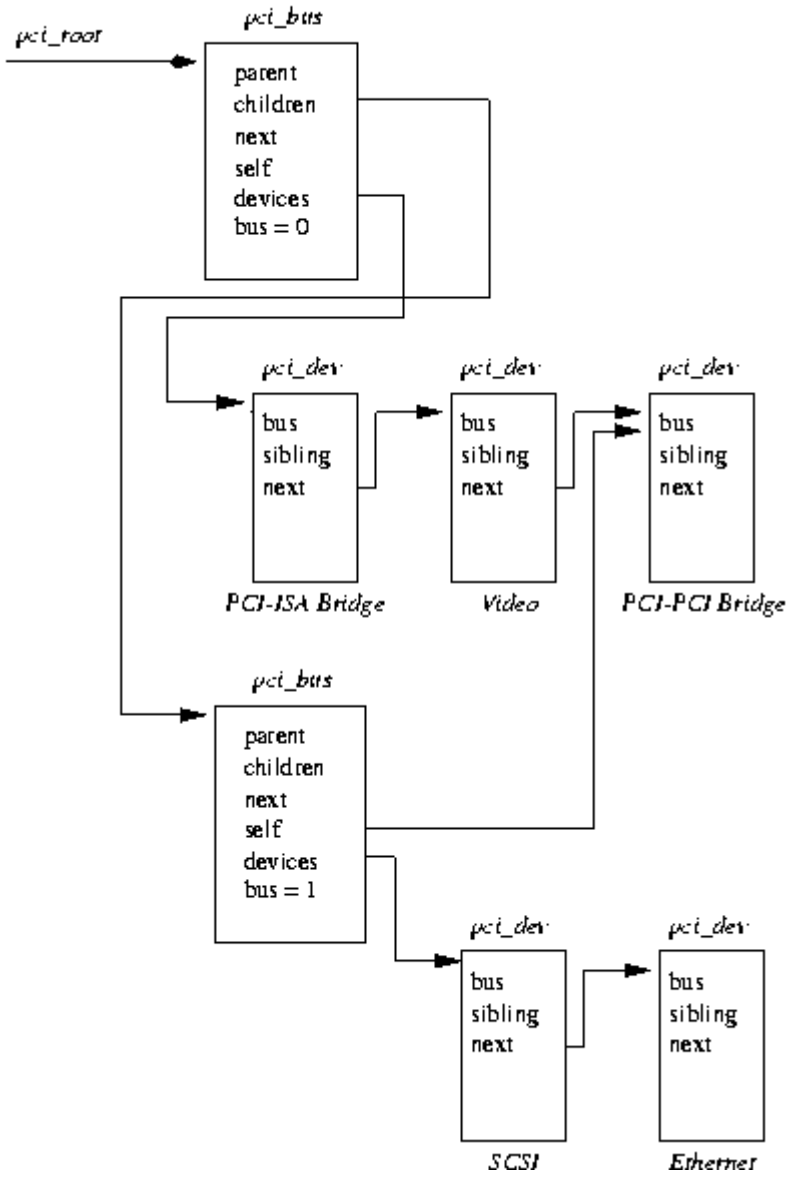


图 6.5 Linux 核心 PCI 数据结构

Linux 核心初始化 PCI 系统时同时也建立了反应系统中真实 PCI 拓扑的数据结构。 图 6.5 显示了图 6.1 所标识的 PCI 示例系统中数据结构间关系。每个 PCI 设备（包括 PCI-PCI 桥接器）用一个 pci_dev 数据结构来描叙。每个 PCI 总线用一个 pci_bus 数据结构来描叙。这样系统中形成了一个 PCI 总线树，每棵树上由一些子 PCI 设备组成。由于 PCI 总线仅能通过 PCI-PCI 桥接器（除了主干 PCI 总线 0）存取，所以 pci_bus 结构中包含一个指向 PCI-PCI 桥接器的指针。这个 PCI 设备是 PCI 总线的父 PCI 总线的子设备。

在图 6.5 中没有显示出来的是一个指向系统中所有 PCI 设备的指针，pci_devices。系统中所有的 PCI 设备将其各自的 pci_dev 数据结构加入此队列中。这个队列被 Linux 核心用来迅速查找系统中所有的 PCI 设备。

6.6.2 PCI 设备驱动

PCI 设备驱动根本不是真正的设备驱动，它仅是在系统初始化时由操作系统调用的一些函数。PCI 初始化代码将扫描系统中所有的 PCI 总线以找到系统中所有的 PCI 设备（包括 PCI-PCI 桥接器）。

它通过 PCI BIOS 代码来检查当前 PCI 总线的每个插槽是否已被占用。如果被占用则它建立一个 pci_dev 数据结构来描叙此设备并将其连接到已知 PCI 设备链表中（由 pci_devices 指向）。

首先 PCI 初始化代码扫描 PCI 总线 0。它将试图读取对每个 PCI 槽中可能的 PCI 设备厂商标志与设备标志域。当发现槽被占用后将建立一个 pci_dev 结构来描叙此设备。所有这些 PCI 初始化代码建立的 pci_dev 结构（包括 PCI-PCI 桥接器）将被连接到一个单向链表 pci_devices 中。

如果这个 PCI 设备是一个 PCI-PCI 桥接器则建立一个 pci_bus 结构并将其连接到由 pci_root 指向的 pci_dev 结构和 pci_bus 树中。PCI 初始化代码通过类别代码 0x060400 来判断此 PCI 设备是否是一个 PCI-PCI 桥接器。然后 Linux 核心代码将配置此 PCI-PCI 桥接器下方的 PCI 设备。如果有更多的桥接器被找到则进行同样的配置。显然这个过程使用了深度优先搜索算法; 系统中 PCI 拓扑将在进行广度映射前先进行深度优先映射。图 6.1 中 Linux 将在配置 PCI 总线 0 上的视频设备前先配置 PCI 设备 1 上的以太与 SCSI 设备。

由于 Linux 优先搜索从属的 PCI 总线,它必须处理 PCI-PCI 桥接器二级总线与从属总线序号。在下面的 pci-pci 总线序号分配中将进行详细讨论。

配置 PCI-PCI 桥接器 - 指定 PCI 总线序号

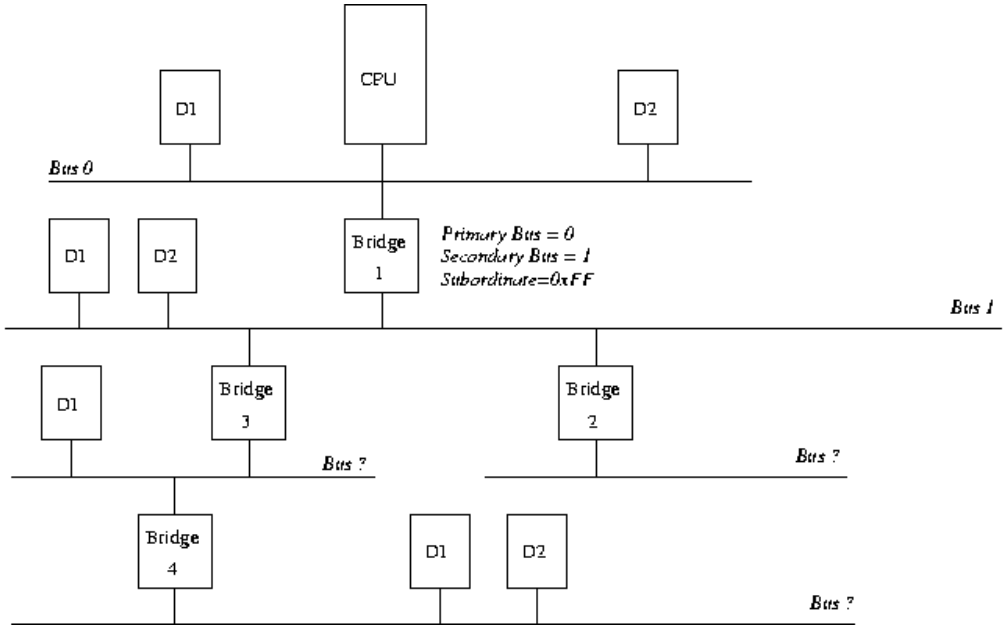


图 6.6 配置 PCI 系统：第一部分

为了让 PCI-PCI 桥接器可以传递 PCI I/O、PCI 内存或 PCI 配置地址空间，它们需要如下内容：

Primary Bus Number:主干总线序号
位于 PCI-PCI 桥接器上方的总线序号
Secondary Bus Number:二级总线序号
位于 PCI-PCI 桥接器下方的总线序号
Subordinate Bus Number:从属总线序号
在桥接器下方可达的最大总线序号
PCI I/O and PCI Memory Windows:PCI I/O 与 PCI 内存窗口
对于 PCI-PCI 桥接器下方所有 PCI I/O 地址空间与 PCI 内存地址空间的窗口基址和大小。

配置任一 PCI-PCI 桥接器时我们对此桥接器的从属总线序号一无所知。不知道是否还有下一级桥接器存在,同时也不知道指派给它们的序号是什么。但可以使用深度优先遍历算法来对扫描出指定 PCI-PCI 桥接器连接的每条总线，同时将它们编号。当找到一个 PCI-PCI 桥接器时，其二级总线被编号并且将临时从属序号 0xff 指派给它以便对其所有下属 PCI-PCI 桥接器进行扫描与指定序号。以上过程看起来十分复杂，下面将提供一个实例以帮助理解。

PCI-PCI 桥接器序号分配：步骤 1

考虑图 6.6 所显示的拓扑结构，第一个被扫描到的桥接器将是桥 1。所以桥 1 下方的总线将被编号成总线 1,同时桥 1 被设置为二级总线 1 且拥有临时总线序号 0xff。这意味着所有 PCI 总线序号为 1 或以上的类型 1 PCI 配置地址将被通过桥 1 传递到 PCI 总线 1 上。如果其总线序号为 1 则此配置循环将被转换成类型 0 配置循环，对于其它序号不作转换。这正是 Linux PCI 初始化代码所需要的按序访问及扫描 PCI 总线 1。

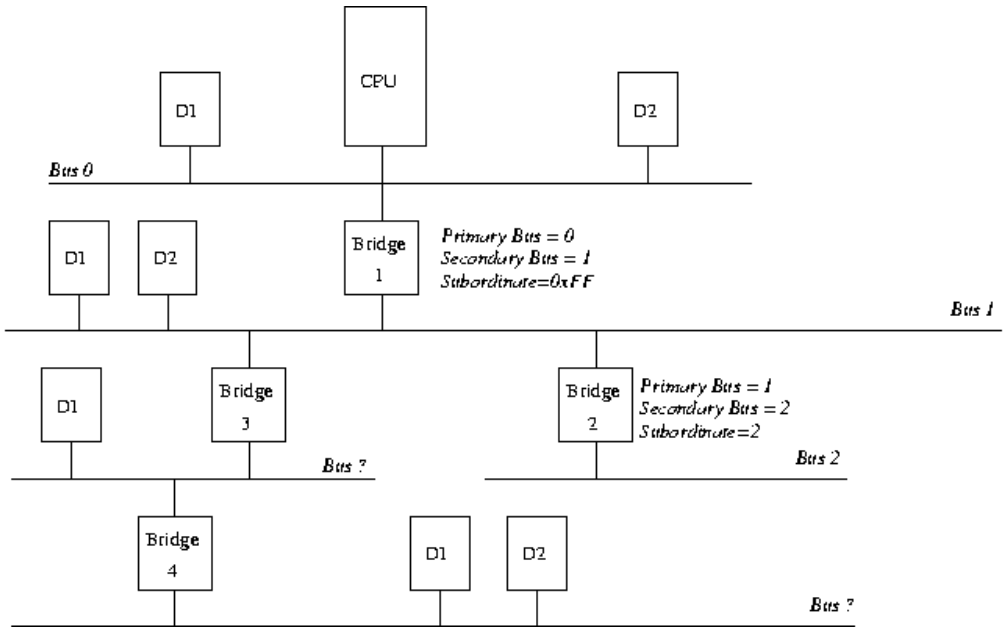


图 6.7 配置 PCI 系统：第二部分

PCI-PCI 桥接器序号分配：第二步

由于 Linux 使用深度优先算法,初始化代码将继续扫描 PCI 总线 1。在此处它将发现一个 PCI-PCI 桥接器 2。除此桥接器 2 外再没有其它桥接器存在，因此它被分配给从属总线序号 2，这正好和其二级接口序号相同。图 6.7 画出了此处的 PCI-PCI 桥接器与总线的编号情况。

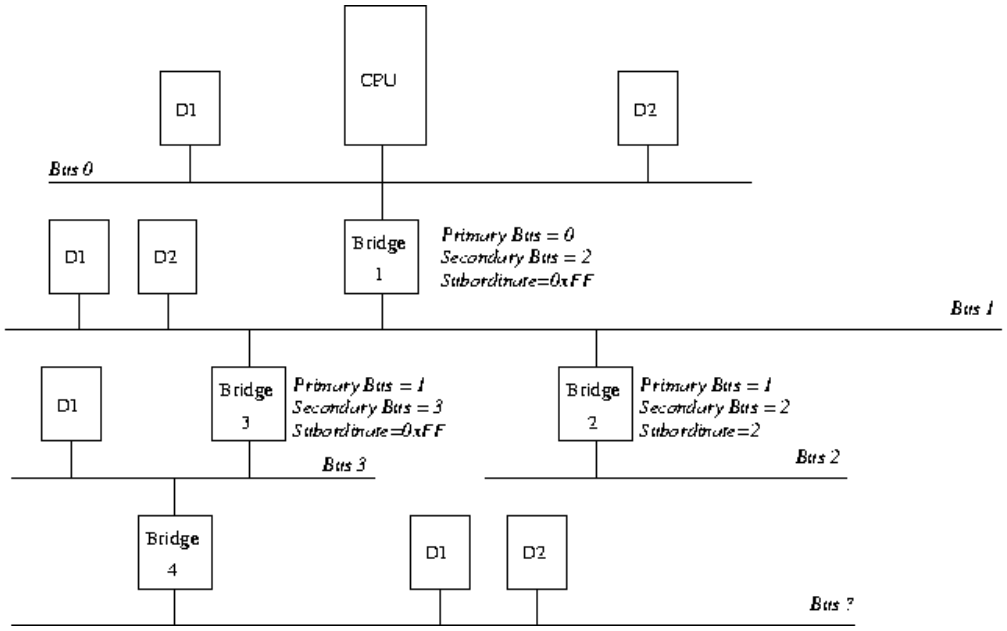


图 6.8 配置 PCI 系统：第三部分

PCI-PCI 桥接器序号分配：步骤三

PCI 初始化代码将继续扫描总线 1 并发现另外一个 PCI-PCI 桥接器，桥 3。桥 3 的主干总线接口序号被设置成 1，二级总线接口序号为 3,同时从属总线序号为 0xff。图 6.8 给出了系统现在的配置情况。 带总线序号 1、2 或者 3 的类型 1 PCI 配置循环将被发送到正确的 PCI 总线。

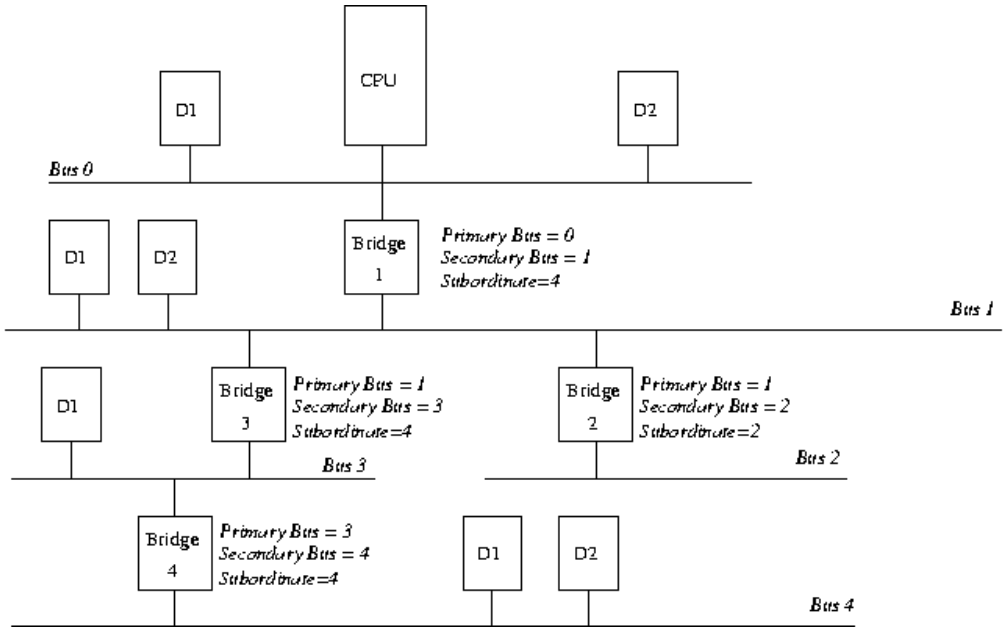


图 6.9 配置 PCI 系统：第四部分

PCI-PCI 桥接器序号分配：步骤四

Linux 开始沿 PCI 总线 3 向下扫描 PCI-PCI 桥接器。PCI 总线 3 上有另外一个 PCI-PCI 桥接器（桥 4)， 桥 4 的主干总线序号被设置成 3，二级总线序号为 4。由于它是此分支上最后一个桥接器所以它的从属总线接口序号为 4。初始化代码将重新从 PCI-PCI 桥接器 3 开始并将其从属总线序号设为 4。 最后 PCI 初始化代码将 PCI-PCI 桥接器 1 的从属总线序号设置为 4。图 6.9 给出了最后的总线序号分配情况。

6.6.3 PCI BIOS 函数

PCI BIOS 函数是一组适用于所有平台的标准过程。在 Intel 和 Alpha AXP 系统上没有区别。虽然在 CPU 控制下可以用它们对所有 PCI 地址空间进行访问。但只有 Linux 核心代码和设备驱动才能使用它们。

6.6.4 PCI 补丁代码

在 Alpha AXP 平台上的 PCI 补丁代码所作工作量要大于 Intel 平台。

基于 Intel 的系统在系统启动时就已经由系统 BIOS 完成了 PCI 系统的配置。Linux 只需要完成简单的映射配置. 非 Intel 系统将需要更多的配置：

为每个设备分配 PCI I/O 及 PCI 内存空间。

为系统中每个 PCI-PCI 桥接器配置 PCI I/O 和 PCI 内存地址窗口。

为这些设备产生中断连线值；用来控制设备的中断处理。

下一节将描叙这些代码的工作过程。

确定设备所需 PCI I/O 和 PCI 内存空间的大小

系统要查询每个 PCI 设备需要多少 PCI I/O 于 PCI 内存地址空间。为了完成这项工作，每个基地址寄存器将被写上全 1 并读取出来。设备将把不必要的地址位设为 0 从而有效的定义所需地址空间。

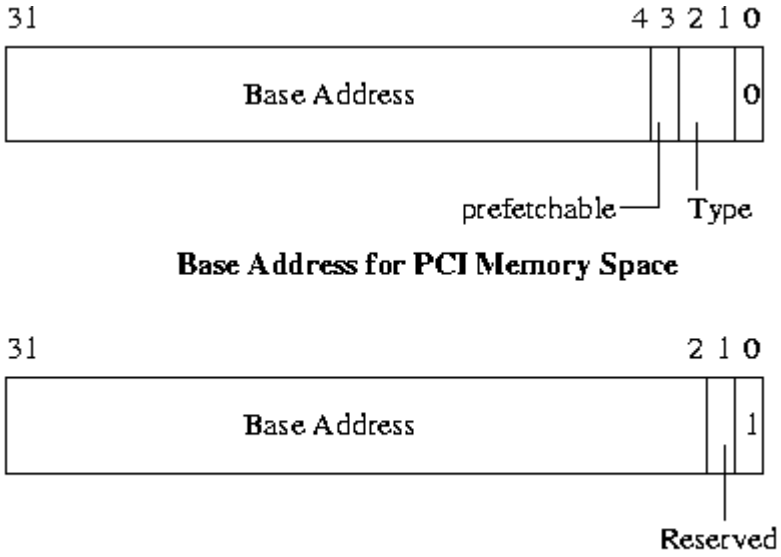


图 6.10 PCI 配置头：基地址寄存器

有两类基本的基地址寄存器，一类标识设备寄存器必须驻留的地址空间；另一类是 PCI I/O 或 PCI 内存空间。此寄存器的 0 位来进行类型的区分。图 6.10 给出了对应于 PCI 内存和 PCI I/O 两种不同类型的基地址寄存器。

确定某个基地址寄存器所需地址空间大小时,先向此寄存器写入全 1 再读取此寄存器,设备将在某些位填上 0 来形成一个二进制数表示所需有效地址空间。

以初始化 DEC 21142 PCI 快速以太设备为例，它将告诉系统需要 0x100 字节的 PCI I/O 空间或者 PCI 内存空间。于是初始化代码为其分配空间。空间分配完毕后，就可以在那些地址上看到 21142 的控制与状态寄存器。

为 PCI-PCI 桥接器与设备分配 PCI I/O 与 PCI 内存

象所有内存一样，PCI I/O 和 PCI 内存空间是非常有限甚至匮乏。非 Intel 系统的 PCI 补丁代码（或者 Intel 系统的 BIOS 代码）必须为每个设备分配其所要求的内存。PCI I/O 和 PCI 内存必须以自然对齐方式分配给每个设备。比如如果一个设备要求 0xB0 大小的 PCI I/O 空间则它必须和一个 0xB0 倍数的地址对齐。除此以外，对于任何指定桥接器，其 PCI I/O 和 PCI 内存基址必须以在 1M 字节边界上以 4K 字节方式对齐。所以在桥接器下方的设备的地址空间必须位于任意指定设备上方的 PCI-PCI 桥接器的内存范围内。进行有效的空间分配是一件比较困难的工作。

Linux 使用的算法依赖于由 PCI 设备驱动程序建立的描叙 PCI 设备的总线/设备树，每个设备的地址空间按照 PCI I/O 内存顺序的升序来分配。同时再次使用遍历算法来遍历由 PCI 初始化代码建立的 pci_bus 和 pci_dev 结构。从根 PCI 总线开始（由 pci_boot 指向）PCI 补丁代码将完成下列工作：

使当前全局 PCI I/O 和内存的基址在 4K，边界在 1M 上对齐。

对于当前总线上的每个设备（按照 PCI I/O 内存需要的升序排列）
在 PCI I/O 和 PCI 内存中为其分配空间

为全局 PCI I/O 和内存基址同时加上一个适当值

授予设备对 PCI I/O 和 PCI 内存的使用权

为对于当前总线下方的所有总线循环分配空间。注意这将改变全局 PCI I/O 和内存基址。

使当前全局 PCI I/O 和内存的基址和边界分别在 4K 和 1M 对齐，以便确定当前 PCI-PCI 桥接器所需的 PCI I/O 和 PCI 内存基址及大小。

对此 PCI-PCI 桥接器编程，将其 PCI I/O 和 PCI 内存基址及界限连接到总线上。

打开 PCI-PCI 桥接器上的 PCI I/O 和 PCI 内存访问桥接功能。这时在此桥接器主干 PCI 总线上位于此桥接器 PCI I/O 和 PCI 内存地址窗口中的任何 PCI I/O 或者 PCI 内存地址将被桥接到二级 PCI 总线上。

以图 6.1 中的 PCI 系统为例，PCI 补丁代码将以如下方式设置系统：

对齐 PCI 基址

PCI I/O 基址为 0x4000 而 PCI 内存基址为 0x100000。这样允许 PCI-ISA 桥接器将此地址以下的地址转换成 ISA 地址循环。

视频设备

我们按照它的请求从当前 PCI 内存基址开始分配 0x200000 字节给它，这样可以在边界上对齐。PCI 内存基址被移到 0x400000 同时 PCI I/O 基址保持在 0x4000。

PCI-PCI 桥接器

现在我们将穿过 PCI-PCI 桥接器来分配 PCI 内存，注意此时我们无需对齐这些基址，因为它们已经自然对齐。

以太网设备

它需要 0xB0 字节的 PCI I/O 和 PCI 内存空间。这些空间从 PCI I/O 地址 0x4000 和 PCI 内存地址 0x400000 处开始。PCI 内存基址被移动到 0x4000B0 同时 PCI I/O 基址移动到 0x40B0。

SCSI 设备

它需要 0x1000 字节 PCI 内存，所以它将在自然对齐后从从 0x401000 处开始分配空间。PCI I/O 基址仍然在 0x40B0 而 PCI 内存基址被移动到 0x402000。

PCI-PCI 桥接器的 PCI I/O 和内存窗口

现在我们重新回到桥接器并将其 PCI I/O 窗口设置成 0x4000 和 0x40B0 之间，同时其 PCI 内存窗口被设置到 0x400000 和 0x402000 之间。这样此 PCI-PCI 桥接器将忽略对视频设备的 PCI 内存访问但传递对以太网设备或者 SCSI 设备的访问。

第七章 中断及中断处理

本章主要描叙 Linux 核心的中断处理过程。尽管核心提供通用机制与接口来进行中断

处理，大多数中断处理细节都是 CPU 体系结构相关的。

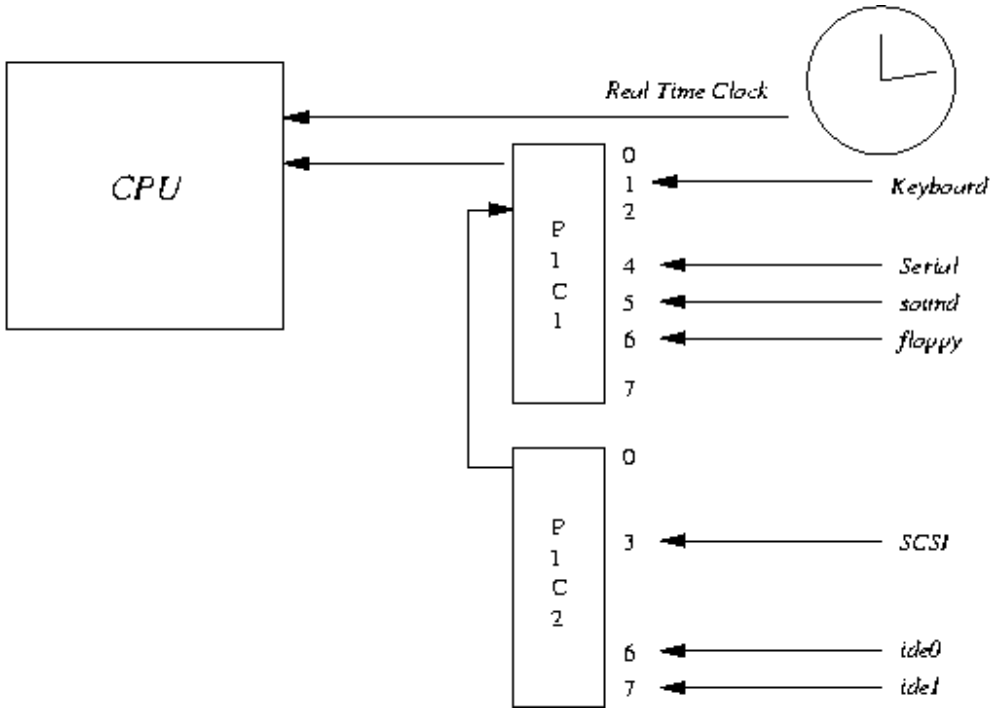


图 7.1 中断路由的逻辑图

Linux 通过使用多种不同硬件来执行许多不同任务。包括驱动显示器的视频设备、驱动硬盘的 IDE 设备等。 我们可以同步驱动这些设备，即我们可以发送一个请求执行一组操作（比如说将一块内存数据写入到磁盘）然后等待到执行完毕。这种方式虽然可以工作,但是效率很低，因为操作系统必须等待每个操作的完成,所以操作系统将花费大量时间在“忙等待”上。更为有效的方式是执行请求，然后转去执行其它任务。当设备完成请求时再通过中断通知操作系统。这样系统中可以同时存在多个未完成的任务。

不管 CPU 在作什么工作，为了让设备产生中断必须提供一些必要的硬件支持。几乎所有的通用处理器如 Alpha AXP 都使用近似的方法。CPU 的一些物理引脚被设计成可以改变电压（如从+5V 变成-5V）从而引起 CPU 停止当前工作并开始执行处理中断的特殊代码：中断处理程序。这些引脚之一被连接到一个周期性时钟上并每隔千分之一秒就接收一次中断，其它引脚则可连接到系统中其它设备如 SCSI 控制器上。

系统常使用中断控制器来在向 CPU 中断引脚发送信号之前将设备中断进行分组。这样可以节省 CPU 上中断引脚个数，同时增加了系统设计的灵活性。此中断控制器通过屏蔽与状态寄存器来控制中断。通过设置屏蔽寄存器中的某些位可以使能或者关闭中断，读取状态寄存器可得到系统当前处于活动状态的中断。

系统中有些中断是通过硬连线连接的，如实时时钟的周期性定时器可能被固定连接到中断控制器的引脚 3 上。而其它连接到控制器的引脚只能由插到特定 ISA 或 PCI 槽中的控制卡来决定。例如中断控制器中的引脚 4 可能被连接到 PCI 槽号 0，但可能某天此槽中插入一块以太网卡而过几天又会换成 SCSI 控制器。总之每个系统都有其自身的中断路由机制，同时操作系统还应该能灵活处理这些情况。

多数现代通用微处理器使用近似的方法来处理中断。硬件中断发生时，CPU 将停止执

行当前指令并将跳转到内存中包含中断处理代码或中断处理代码指令分支的位置继续执行。这些代码在一种特殊 CPU 模式： 中断模式下执行。通常在此模式下不会有其它中断发生。但是也有例外；有些 CPU 将中断的优先级进行分类，此时更高优先级的中断还可能发生。这样意味着必须认真编写第一级中断处理代码，同时中断处理过程应该拥有其自身的堆栈，以便存储转到中断处理过程前的 CPU 执行状态（所有 CPU 的普通寄存器和上下文）。一些 CPU 具有一组特殊的寄存器-它们仅存在于中断模式中，在中断模式下可以使用这些寄存器来保存执行所需要的执行上下文。

当中断处理完毕后 CPU 状态将被重储，同时中断也将被释放。CPU 将继续做那些中断发生前要做的工作。中断处理代码越精炼越好，这样将减少操作系统阻塞在中断上的时间与频率。

7.1 可编程中断控制器

系统设计者可以自由选择中断结构，一般的 IBM PC 兼容将使用 Intel 82C59A-2 CMOS 可编程中断控制器或其派生者。这种控制器在 PC 诞生之前便已经产生，它的可编程性体现在那些位于众所周知 ISA 内存位置中的寄存器上。非 Intel 系统如基于 Alpha AXP 的 PC 不受这些体系结构限制，它们经常使用各种不同的中断控制器。

图 7.1 给出了两个级连的 8 位控制器，每个控制器都有一个屏蔽与中断状态寄存器：PIC1 和 PIC2。这两个屏蔽寄存器分别位于 ISA I/O 空间 0x21 和 0xA1 处，状态寄存器则位于 0x20 和 0xA0。对此屏蔽寄存器某个特定位置位将使能某一中断，写入 0 则屏蔽它。但是不幸的是中断屏蔽寄存器是只写的，所以你无法读取你写入的值。这也意味着 Linux 必须保存一份对屏蔽寄存器写入值的局部拷贝。一般在中断使能和屏蔽例程中修改这些保存值，同时每次将这些全屏蔽码写入寄存器。

当有中断产生时，中断处理代码将读取这两个中断状态寄存器（ISR）。它将 0x20 中的 ISR 看成一个 16 位中断寄存器的低 8 位而将 0xA0 中的 ISR 看成其高 8 位。这样 0xA0 中 ISR 第 1 位上的中断将被视作系统中断 9。PIC1 上的第二位由于被用来级连 PIC2 所以不能作其它用处，PIC2 上的任何中断将导致 PIC1 的第二位被置位。

7.2 初始化中断处理数据结构

核心的中断处理数据结构在设备驱动请求系统中断控制时建立。为完成此项工作，设备驱动使用一组 Linux 核心函数来请求中断，使能中断和屏蔽中断。
每个设备驱动将调用这些过程来注册其中断处理例程地址。

有些中断由于传统的 PC 体系结构被固定下来，所以驱动仅需要在其初始化时请求它的中断。软盘设备驱动正是使用的这种方式;它的中断号总为 6。有时设备驱动也可能不知道设备使用的中断号。对 PCI 设备驱动来说这不是什么大问题，它们总是可以知道其中断号。但对于 ISA 设备驱动则没有取得中断号的方便方式。Linux 通过让设备驱动检测它们的中断号来解决这个问题。

设备驱动首先迫使设备引起一个中断。系统中所有未被分配的中断都被使能。此时设备引发的中断可以通过可编程中断控制器来发送出去。Linux 再读取中断状态寄存器并将其内容返回给设备驱动。非 0 结果则表示在此次检测中有一个或多个中断发生。设备驱动

然后将关闭检测并将所有未分配中断屏蔽掉。

如果 ISA 设备驱动成功找到了设备的 IRQ 号，就可以象平常一样请求对设备的控制。

基于 PCI 系统比基于 ISA 系统有更多的动态性。ISA 设备使用的中断引脚通常是通过硬件设备上的跳线来设置并固定在设备驱动中。PCI 设备在系统启动与初始化 PCI 时由 PCI BIOS 或 PCI 子系统来分配中断。每个 PCI 设备可以使用 A，B，C 或 D 之中的任意中断。这个中断在设备建立时确定且通常多数设备的缺省中断为 A。PCI 槽中的 PCI 中断连线 A，B，C 和 D 被正确路由到中断控制器中。所以 PCI 槽 4 上的引脚 A 可能被路由到中断控制器上的引脚 6，PCI 槽 7 上的引脚 B 被路由到中断控制器上的引脚 7 等等。

如何路由 PCI 中断完全取决于特定的系统，一般设置代码能理解 PCI 中断路由拓扑。在基于 Intel 的 PC 上由系统 BIOS 代码在启动时作这些设置而在不带 BIOS（如 Alpha AXP）系统中由 Linux 核心来完成这个任务。

PCI 设置代码将每个设备对应的中断控制器的引脚号写入 PCI 配置头中。通过得到 PCI 中断路由拓扑及设备的 PCI 槽号和 PCI 中断引脚设置代码可以确定其对应的中断引脚（或 IRQ）号。设备使用的中断引脚被保存在此设备的 PCI 配置头中为此目的保留的中断连线域中。当运行设备驱动时这些信息被读出并用来控制来自 Linux 核心的中断请求。

系统中可能存在许多 PCI 中断源，比如在使用 PCI-PCI 桥接器时。这些中断源的个数可能将超出系统可编程中断控制器的引脚数。此时 PCI 设备必须共享中断号-中断控制器上的一个引脚可能被多个 PCI 设备同时使用。Linux 让中断的第一个请求者申明此中断是否可以共享。中断的共享将导致 irq_action 数组中的一个入口同时指向几个 irqaction 数据结构。当共享中断发生时 Linux 将调用对应此中断源的所有中断处理过程。没有中断需要服务时，任何共享此中断（所有的 PCI 设备驱动）的设备驱动都要准备好其中断处理过程的调用。

7.3 中断处理

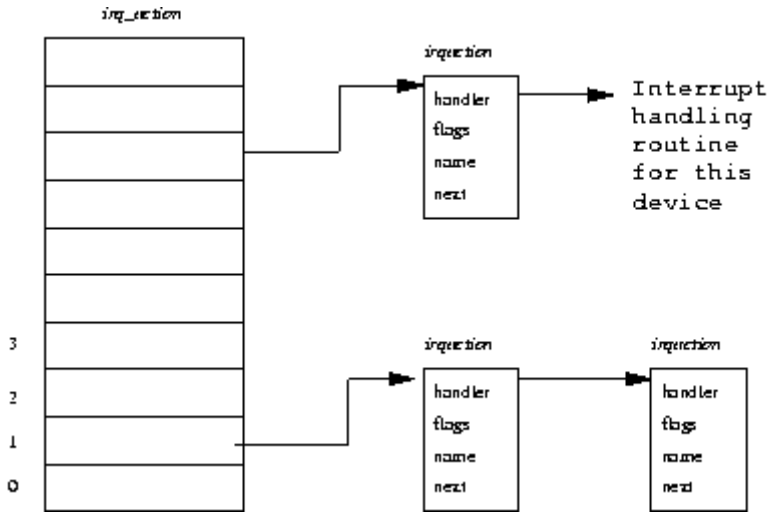


图 7.2 Linux 中断处理数据结构

Linux 中断处理子系统的一个基本任务是将中断正确路由到中断处理代码中的正确位

置。这些代码必须了解系统的中断拓扑结构。例如在中断控制器上引脚 6 上发生的软盘控制器中断必须被辨认出的确来自软盘并路由到系统的软盘设备驱动的中断处理代码中。Linux 使用一组指针来指向包含处理系统中断的例程的调用地址。这些例程属于对应于此设备的设备驱动，同时由它负责在设备初始化时为每个设备驱动申请其请求的中断。图 7.2 给出了一个指向一组 irqaction 的 irq_action 指针。每个 irqaction 数据结构中包含了对应于此中断处理的相关信息，包括中断处理例程的地址。而中断个数以及它们被如何处理则会根据体系结构及系统的变化而变化。Linux 中的中断处理代码就是和体系结构相关的。这也意味着 irq_action 数组的大小随于中断源的个数而变化。

中断发生时 Linux 首先读取系统可编程中断控制器中中断状态寄存器判断出中断源，将其转换成 irq_action 数组中偏移值。例如中断控制器引脚 6 来自软盘控制器的中断将被转换成对应于中断处理过程数组中的第 7 个指针。如果此中断没有对应的中断处理过程则 Linux 核心将记录这个错误，不然它将调用对应此中断源的所有 irqaction 数据结构中的中断处理例程。

当 Linux 核心调用设备驱动的中断处理过程时此过程必须找出中断产生的原因以及相应的解决办法。为了找到设备驱动的中断原因，设备驱动必须读取发生中断设备上的状态寄存器。设备可能会报告一个错误或者通知请求的处理已经完成。如软盘控制器可能将报告它已经完成软盘读取磁头对某个扇区的正确定位。一旦确定了中断产生的原因，设备驱动还要完成更多的工作。如果这样 Linux 核心将推迟这些操作。以避免了 CPU 在中断模式下花费太多时间。在设备驱动中断中我们将作详细讨论。

第八章 设备驱动

操作系统的目的之一就是将系统硬件设备细节从用户视线中隐藏起来。例如虚拟文件系统对各种类型已安装的文件系统提供了统一的视图而屏蔽了具体底层细节。本章将描叙 Linux 核心对系统中物理设备的管理。

CPU 并不是系统中唯一的智能设备，每个物理设备都拥有自己的控制器。键盘、鼠标和串行口由一个高级 I/O 芯片统一管理，IDE 控制器控制 IDE 硬盘而 SCSI 控制器控制 SCSI 硬盘等等。每个硬件控制器都有各自的控制和状态寄存器(CSR)并且各不相同。例如 Adaptec 2940 SCSI 控制器的 CSR 与 NCR 810 SCSI 控制器完全不一样。这些 CSR 被用来启动和停止，初始化设备及对设备进行诊断。在 Linux 中管理硬件设备控制器的代码并没有放置在每个应用程序中而是由内核统一管理。这些处理和管理硬件控制器的软件就是设备驱动。Linux 核心设备驱动是一组运行在特权级上的内存驻留底层硬件处理共享库。正是它们负责管理各个设备。

设备驱动的一个基本特征是设备处理的抽象概念。所有硬件设备都被看成普通文件；可以通过和操纵普通文件相同的标准系统调用来打开、关闭、读取和写入设备。系统中每个设备都用一种特殊的设备相关文件来表示(device special file)，例如系统中第一个 IDE 硬盘被表示成/dev/hda。块（磁盘）设备和字符设备的设备相关文件可以通过 mknod 命令来创建，并使用主从设备号来描叙此设备。网络设备也用设备相关文件来表示，但 Linux 寻找和初始化网络设备时才建立这种文件。由同一个设备驱动控制的所有设备具有相同的主设备号。从设备号则被用来区分具有相同主设备号且由相同设备驱动控制的不同设备。 例

如主 IDE 硬盘的每个分区的从设备号都不相同。如/dev/hda2 表示主 IDE 硬盘的主设备号为 3 而从设备号为 2。Linux 通过使用主从设备号将包含在系统调用中的（如将一个文件系统 mount 到一个块设备）设备相关文件映射到设备的设备驱动以及大量系统表格中，如字符设备表，chrdevs。

Linux 支持三类硬件设备：字符、块及网络设备。字符设备指那些无需缓冲直接读写的设备，如系统的串口设备/dev/cua0 和/dev/cua1。块设备则仅能以块为单位读写，典型的块大小为 512 或 1024 字节。块设备的存取是通过 buffer cache 来进行并且可以进行随机访问，即不管块位于设备中何处都可以对其进行读写。块设备可以通过其设备相关文件进行访问，但更为平常的访问方法是通过文件系统。只有块设备才能支持可安装文件系统。网络设备可以通过 BSD 套接口访问，我们将在网络一章中讨论网络子系统。

Linux 核心中虽存在许多不同的设备驱动但它们具有一些共性：

核心代码

设备驱动是核心的一部分，象核心中其它代码一样，出错将导致系统的严重损伤。一个编写奇差的设备驱动甚至能使系统崩溃并导致文件系统的破坏和数据丢失。

核心接口

设备驱动必须为 Linux 核心或者其从属子系统提供一个标准接口。例如终端驱动为 Linux 核心提供了一个文件 I/O 接口而 SCSI 设备驱动为 SCSI 子系统提供了一个 SCSI 设备接口，同时此子系统为核心提供了文件 I/O 和 buffer cache 接口。

核心机制与服务

设备驱动可以使用标准的核心服务如内存分配、中断发送和等待队列等等。

动态可加载

多数 Linux 设备驱动可以在核心模块发出加载请求时加载，同时在不再使用时卸载。这样核心能有效地利用系统资源。

可配置

Linux 设备驱动可以连接到核心中。当核心被编译时，哪些核心被连入核心是可配置的。

动态性

当系统启动及设备驱动初始化时将查找它所控制的硬件设备。如果某个设备的驱动为一个空过程并不会有什么问题。此时此设备驱动仅仅是一个冗余的程序，它除了会占用少量系统内存外不会对系统造成什么危害。

8.1 轮询与中断

设备被执行某个命令时，如“将读取磁头移动到软盘的第 42 扇区上”，设备驱动可以从轮询方式和中断方式中选择一种以判断设备是否已经完成此命令。

轮询方式意味着需要经常读取设备的状态，一直到设备状态表明请求已经完成为止。如果设备驱动被连接进入核心，这时使用轮询方式将会带来灾难性后果：核心将在此过程中无所事事，直到设备完成此请求。但是轮询设备驱动可以通过使用系统定时器，使核心周期性调用设备驱动中的某个例程来检查设备状态。 定时器过程可以检查命令状态及 Linux 软盘驱动的工作情况。使用定时器是轮询方式中最好的一种，但更有效的方法是使用中断。

基于中断的设备驱动会在它所控制的硬件设备需要服务时引发一个硬件中断。如以太网设备驱动从网络上接收到一个以太数据报时都将引起中断。Linux 核心需要将来自硬件

设备的中断传递到相应的设备驱动。这个过程由设备驱动向核心注册其使用的中断来协助完成。此中断处理例程的地址和中断号都将被记录下来。在/proc/interrupts 文件中你可以看到设备驱动所对应的中断号及类型：

```
 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
```

对中断资源的请求在驱动初始化时就已经完成。作为 IBM PC 体系结构的遗产，系统中有些中断已经固定。例如软盘控制器总是使用中断 6。其它中断，如 PCI 设备中断，在启动时进行动态分配。设备驱动必须在取得对此中断的所有权之前找到它所控制设备的中断号（IRQ）。Linux 通过支持标准的 PCI BIOS 回调函数来确定系统中 PCI 设备的中断信息，包括其 IRQ 号。

如何将中断发送给 CPU 本身取决于体系结构，但是在多数体系结构中，中断以一种特殊模式发送同时还将阻止系统中其它中断的产生。设备驱动在其中断处理过程中作的越少越好，这样 Linux 核心将能很快的处理完中断并返回中断前的状态中。为了在接收中断时完成大量工作，设备驱动必须能够使用核心的底层处理例程或者任务队列来对以后需要调用的那些例程进行排队。

8.2 直接内存访问 (DMA)

数据量比较少时，使用中断驱动设备驱动程序能顺利地在硬件设备和内存之间交换数据。例如波特率为 9600 的 modem 可以每毫秒传输一个字符。如果硬件设备引起中断和调用设备驱动中断所消耗的中断时延比较大（如 2 毫秒）则系统的综合数据传输率会很低。则 9600 波特率 modem 的数据传输只能利用 0.002%的 CPU 处理时间。高速设备如硬盘控制器或者以太网设备数据传输率将更高。SCSI 设备的数据传输率可达到每秒 40M 字节。

直接内存存取（DMA）是解决此类问题的有效方法。DMA 控制器可以在不受处理器干预的情况下在设备和系统内存之间高速传输数据。PC 机的 ISA DMA 控制器有 8 个 DMA 通道，其中七个可以由设备驱动使用。每个 DMA 通道具有一个 16 位的地址寄存器和一个 16 位的记数寄存器。为了初始化数据传输，设备驱动将设置 DMA 通道地址和记数寄存器以描叙数据传输方向以及读写类型。然后通知设备可以在任何时候启动 DMA 操作。传输结束时设备将中断 PC。在传输过程中 CPU 可以转去执行其他任务。

设备驱动使用 DMA 时必须十分小心。首先 DMA 控制器没有任何虚拟内存的概念，它只存取系统中的物理内存。同时用作 DMA 传输缓冲的内存空间必须是连续物理内存块。这意味着不能在进程虚拟地址空间内直接使用 DMA。但是你可以将进程的物理页面加锁以防止在 DMA 操作过程中被交换到交换设备上去。另外 DMA 控制器所存取物理内存有

限。DMA 通道地址寄存器代表 DMA 地址的高 16 位而页面寄存器记录的是其余 8 位。所以 DMA 请求被限制到内存最低 16M 字节中。

DMA 通道是非常珍贵的资源，一共才有 7 个并且还不能够在设备驱动间共享。与中断一样，设备驱动必须找到它应该使用那个 DMA 通道。有些设备使用固定的 DMA 通道。例如软盘设备总使用 DMA 通道 2。有时设备的 DMA 通道可以由跳线来设置，许多以太网设备使用这种技术。设计灵活的设备将告诉系统它将使用哪个 DMA 通道，此时设备驱动仅需要从 DMA 通道中选取即可。

Linux 通过 dma_chan（每个 DMA 通道一个）数组来跟踪 DMA 通道的使用情况。dma_chan 结构中包含有两个域，一个是指向此 DMA 通道拥有者的指针，另一个指示 DMA 通道是否已经被分配出去。当敲入 cat /proc/dma 打印出来的结果就是 dma_chan 结构数组。

8.3 内存

设备驱动必须谨慎使用内存。由于它属于核心,所以不能使用虚拟内存。系统接收到中断信号时或调度底层任务队列处理过程时，设备驱动将开始运行，而当前进程会发生改变。设备驱动不能依赖于任何运行的特定进程，即使当前是为该进程工作。与核心的其它部分一样，设备驱动使用数据结构来描叙它所控制的设备。这些结构被设备驱动代码以静态方式分配，但会增大核心而引起空间的浪费。多数设备驱动使用核心中非页面内存来存储数据。

Linux 为设备驱动提供了一组核心内存分配与回收过程。核心内存以 2 的次幂大小的块来分配。如 512 或 128 字节，此时即使设备驱动的需求小于这个数量也会分配这么多。所以设备驱动的内存分配请求可得到以块大小为边界的内存。这样核心进行空闲块组合更加容易。

请求分配核心内存时 Linux 需要完成许多额外的工作。如果系统中空闲内存数量较少，则可能需要丢弃些物理页面或将其写入交换设备。一般情况下 Linux 将挂起请求者并将此进程放置到等待队列中直到系统中有足够的物理内存为止。不是所有的设备驱动（或者真正的 Linux 核心代码）都会经历这个过程，所以如分配核心内存的请求不能立刻得到满足，则此请求可能会失败。如果设备驱动希望在此内存中进行 DMA，那么它必须将此内存设置为 DMA 使能的。这也是为什么是 Linux 核心而不是设备驱动需要了解系统中的 DMA 使能内存的原因。

8.4 设备驱动与核心的接口

Linux 核心与设备驱动之间必须有一个以标准方式进行互操作的接口。每一类设备驱动：字符设备、块设备 及网络设备都提供了通用接口以便在需要时为核心提供服务。这种通用接口使得核心可以以相同的方式来对待不同的设备及设备驱动。如 SCSI 和 IDE 硬盘的区别很大但 Linux 对它们使用相同的接口。

Linux 动态性很强。每次 Linux 核心启动时如遇到不同的物理设备将需要不同的物理设备驱动。Linux 允许通过配置脚本在核心重建时将设备驱动包含在内。设备驱动在启动初始化时可能会发现系统中根本没有任何硬件需要控制。其它设备驱动可以在必要时作为核心模块动态加载到。为了处理设备驱动的动态属性，设备驱动在初始化时将其注册到核心

中去。Linux 维护着已注册设备驱动表作为和设备驱动的接口。这些表中包含支持此类设备例程的指针和相关信息。

8.4.1 字符设备

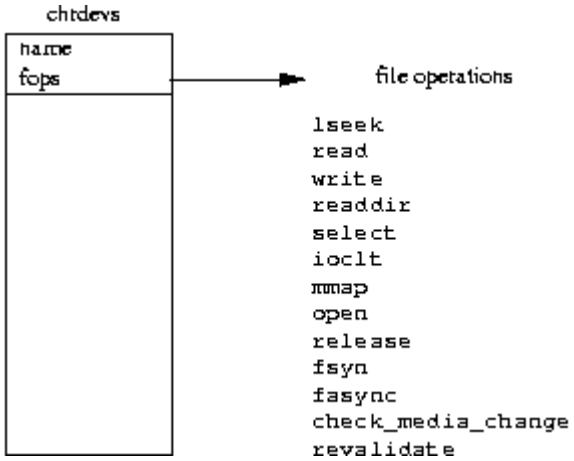


图 8.1 字符设备

字符设备是 Linux 设备中最简单的一种。应用程序可以和存取文件相同的系统调用来打开、读写及关闭它。即使此设备是将 Linux 系统连接到网络中的 PPP 后台进程的 modem 也是如此。字符设备初始化时，它的设备驱动通过在 device_struct 结构的 chrdevs 数组中添加一个入口来将其注册到 Linux 核 心上。设备的主设备标志符用来对此数组进行索引（如对 tty 设备的索引 4）。设备的主设备标志符是固定的。

chrdevs 数组每个入口中的 device_struct 数据结构包含两个元素；一个指向已注册的设备驱动名称，另一个则是指向一组文件操作指针。它们是位于此字符设备驱动内部的文件操作例程的地址指针，用来处理相关的文件操作如打开、读写与关闭。/proc/devices 中字符设备的内容来自 chrdevs 数组。

当打开代表字符设备的字符特殊文件时（如/dev/cua0），核心必须作好准备以便调用相应字符设备驱动的文件操作例程。与普通的目录和文件一样，每个字符特殊文件用一个 VFS 节点表示。每个字符特殊文件使用的 VFS inode 和所有设备特殊文件一样，包含着设备的主从标志符。这个 VFS inode 由底层的文件系统来建立（比如 EXT2），其信息来源于设备相关文件名称所在文件系统。

每个 VFS inode 和一组文件操作相关联,它们根据 inode 代表的文件系统对象变化而不同。当创建一个代表字符相关文件的 VFS inode 时，其文件操作被设置为缺省的字符设备操作。

字符设备只有一个文件操作:打开文件操作。当应用打开字符特殊文件时，通用文件打开操作使用设备的主标志符来索引此 chrdevs 数组，以便得到那些文件操作函数指针。同时建立起描叙此字符特殊文件的 file 结构,使其文件操作指针指向此设备驱动中的文件操作指针集合。这样所有应用对它进行的文件操作都被映射到此字符设备的文件操作集合上。

8.4.2 块设备

块设备也支持以文件方式访问。系统对块设备特殊文件提供了非常类似于字符特殊文件的文件操作机制。Linux 在 blkdevs 数组中维护所有已注册的块设备。象 chrdevs 数组一样，blkdevs 也使用设备的主设备号进行索引。其入口也是 device_struct 结构。和字符设备不同的是系统有几类块设备。SCSI 设备是一类而 IDE 设备则是另外一类。它们将以各自类别登记到 Linux 核心中并为核心提供文件操作功能。某类块设备的设备驱动为此类型设备提供了类别相关的接口。如 SCSI 设备驱动必须为 SCSI 子系统提供接口以便 SCSI 子系统能用它来为核心提供对此设备的文件操作。

和普通文件操作接口一样, 每个块设备驱动必须为 buffer cache 提供接口。每个块设备驱动将填充其在 blk_dev 数组中的 blk_dev_struct 结构入口。数组的索引值还是此设备的主设备号。这个 blk_dev_struct 结构包含请求过程的地址以及指向请求数据结构链表的指针，每个代表一个从 buffer cache 中来让设备进行数据读写的请求。

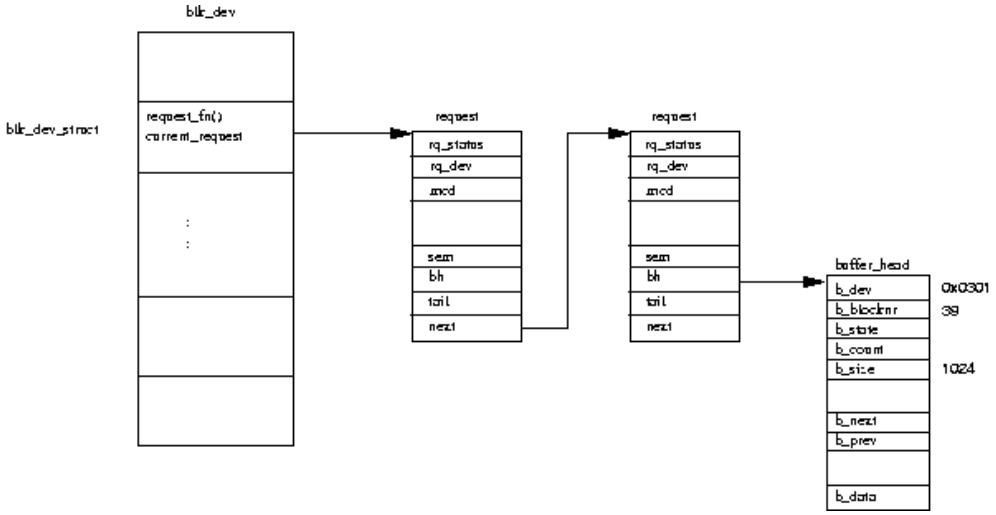


图 8.2 buffer cache 块设备请求

每当 buffer cache 希望从一个已注册设备中读写数据块时,它会将 request 结构添加到其 blk_dev_struct 中。图 8.2 表示每个请求有指向一个或多个 buffer_hear 结构的指针，每个请求读写一块数据。如 buffer cache 对 buffer_head 结构上锁, 则进程会等待到对此缓冲的块操作完成。每个 request 结构都从静态链表 all_requests 中分配。如果此请求被加入到空请求链表中,则将调用驱动请求函数以启动此请求队列的处理,否则该设备驱动将简单地处理请求链表上的 request 。

一旦设备驱动完成了请求则它必须将每个 buffer_heard 结构从 request 结构中清除，将它们标记成已更新状态并解锁之。对 buffer_head 的解锁将唤醒所有等待此块操作完成的睡眠进程。如解析文件名称时,EXT2 文件系统必须从包含此文件系统的设备中读取包含下个 EXT2 目录入口的数据块。在 buffer_head 上睡眠的进程在设备驱动被唤醒后将包含此目录入口。request 数据结构被标记成空闲以便被其它块请求使用。

8.5 硬盘

磁盘驱动器提供了一个永久性存储数据的方式，将数据保存在旋转的盘片上。写入数据时磁头将磁化盘片上的一个小微粒。这些盘片被连接到一个中轴上并以 3000 到 10，000RPM（每分钟多少转）的恒定速度旋转。而软盘的转速仅为 360RPM。磁盘的读/写磁头负责读写数据，每个盘片的两侧各有一个磁头。磁头读写时并不接触盘片表面而是浮在距表面非常近的空气垫中（百万分之一英寸）。磁头由一个马达驱动在盘片表面移动。所有的磁头被连在一起，它们同时穿过盘片的表面。

盘片的每个表面都被划分成为叫做磁道的狭窄同心圆。0 磁道位于最外面而最大磁道位于最靠近中央主轴。柱面指一组相同磁道号的磁道。所以每个盘片上的第五磁道组成了磁盘的第五柱面。由于柱面号与磁道号相等所以我们经常可以看到以柱面描叙的磁盘布局。每个磁道可进一步划分成扇区。它是硬盘数据读写的最小单元同时也是磁盘的块大小。一般的扇区大小为 512 字节并且这个大小可以磁盘制造出来后格式化时设置。

一个磁盘经常被描绘成有多少各柱面、磁头以及扇区。例如系统启动时 Linux 将这样描叙一个 IDE 硬盘：

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

这表示此磁盘有 1050 各柱面（磁道），16 个磁头（8 个盘片）且每磁道包含 63 个扇区。这样我们可以通过扇区数、块数以及 512 字节扇区大小计算出磁盘的存储容量为 529200 字节。这个容量和磁盘自身声称的 516M 字节并不相同，这是因为有些扇区被用来存放磁盘分区信息。有些磁盘还能自动寻找坏扇区并重新索引磁盘以正常使用。

物理硬盘可进一步划分成分区。一个分区是一大组为特殊目的而分配的扇区。对磁盘进行分区使得磁盘可以同时被几个操作系统或不同目的使用。许多 Linux 系统具有三个分区：DOS 文件系统分区，EXT2 文件系统分区和交换分区。硬盘分区用分区表来描叙；表中每个入口用磁头、扇区及柱面号来表示分区的起始与结束。对于用 DOS 格式化的硬盘有 4 个主分区表。但不一定所有的四个入口都被使用。fdisk 支持 3 中分区类型：主分区、扩展分区及逻辑分区。扩展分区并不是真正的分区，它只不过包含了几个逻辑分区。扩展和逻辑分区用来打破四个主分区的限制。以下是一个包含两个主分区的 fdisk 命令的输出：

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

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

Expert command (m for help): p

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

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

4 00 0 0 0 0 0 0 0 0 00

这些内容表明第一个分区从柱面（或者磁道）0，头 1 和扇区 1 开始一直到柱面 477，扇区 22 和头 63 结束。 由于每磁道有 32 个扇区且有 64 个读写磁头则此分区在大小上等于柱面数。fdisk 使分区在柱面边界上对齐。 它从最外面的柱面 0 开始并向中间扩展 478 个柱面。第二个分区：交换分区从 478 号柱面开始并扩展到磁盘的最内圈。

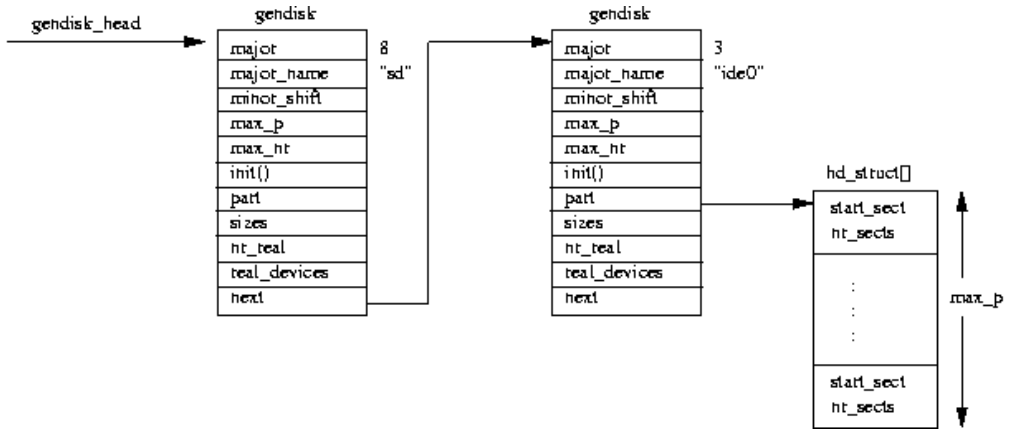


图 8.3 磁盘链表

在初始化过程中 Linux 取得系统中硬盘的拓扑结构映射。它找出有多少中硬盘以及是什么类型。另外 Linux 还要找到每个硬盘的分区方式。所有这些都用由 gendisk_head 链指针指向的 gendisk 结构链表来表示。每个磁盘子系统如 IDE 在初始化时产生表示磁盘结构的 gendisk 结构。同时它将注册其文件操作例程并将此入口添加到 blk_dev 数据结构中。每个 gendisk 结构包含唯一的主设备号，它与块相关设备的主设备号相同。例如 SCSI 磁盘子系统创建了一个主设备号为 8 的 gendisk 入口（"sd"），这也是所有 SCSI 硬盘设备的主设备号。图 8.3 给出了两个 gendisk 入口，一个表示 SCSI 磁盘子系统而另一个表示 IDE 磁盘控制器。ide0 表示主 IDE 控制器。

尽管磁盘子系统在其初始化过程中就建立了 gendisk 入口, 但是只有 Linux 作分区检查时才使用。每个磁盘子系统通过维护一组数据结构将物理硬盘上的分区与某个特殊主从特殊设备互相映射。无论何时通过 buffer cache 或文件操作对块设备的读写都将被核心定向到对具有某个特定主设备号的设备文件上（如 /dev/sda2）。而从设备号的定位由各自设备驱动或子系统来映射。

8.5.1 IDE 硬盘

Linux 系统上使用得最广泛的硬盘是集成电子磁盘或者 IDE 硬盘。IDE 是一个硬盘接口而不是类似 SCSI 的 I/O 总线接口。每个 IDE 控制器支持两个硬盘，一个为主另一个为从。主从硬盘可以通过盘上的跳线来设置。系统中的第一个 IDE 控制器成为主 IDE 控制器而另一个为从属控制器。IDE 可以以每秒 3.3M 字节的传输率传输数据且最大容量为 538M 字节。EIDE 或增强式 IDE 可以将磁盘容量扩展到 8.6G 字节而数据传输率为 16.6M 字节/秒。由于 IDE 和 EIDE 都比 SCSI 硬盘便宜, 所以大多现代 PC 机在包含一个或几个板上 IDE 控制器。

Linux 以其发现控制器的顺序来对 IDE 硬盘进行命名。在主控制器中的主盘为/dev/hda

而从盘为/dev/hdb。/dev/hdc 用来表示从属 IDE 控制器中的主盘。IDE 子系统将向 Linux 核心注册 IDE 控制器而不是 IDE 硬盘。主 IDE 控制器的主标志符为 3 而从属 IDE 控制器的主标志符为 22。如果系统中包含两个 IDE 控制器则 IDE 子系统的入口在 blk_dev 和 blkdevs 数组的第 2 和第 22 处。IDE 的块设备文件反应了这种编号方式，硬盘 /dev/hda 和/dev/hdb 都连接到主 IDE 控制器上，其主标志符为 3。对 IDE 子系统上这些块相关文件的文件或者 buffer cache 的操作都通过核心使用主设备标志符作为索引定向到 IDE 子系统上。当发出请求时，此请求由哪个 IDE 硬盘来完成取决于 IDE 子系统。为了作到这一点 IDE 子系统使用从设备编号对应的设备特殊标志符，由它包含的信息来将请求发送到正确的硬盘上。位于主 IDE 控制器上的 IDE 从盘/dev/hdb 的设备标志符为（3，64）。而此盘中第一个分区（/dev/hdb1）的设备标志符为(3，65)。

8.5.2 初始化 IDE 子系统

IDE 磁盘与 IBM PC 关系非常密切。在这么多年中这些设备的接口发生了变化。这使得 IDE 子系统的初始化过程比看上去要复杂得多。

Linux 可以支持的最多 IDE 控制器个数为 4。每个控制器用 ide_hwifs 数组中的 ide_hwif_t 结构来表示。每个 ide_hwif_t 结构包含两个 ide_drive_t 结构以支持主从 IDE 驱动器。在 IDE 子系统的初始化过程中 Linux 通过访问系统 CMOS 来判断是否有关于硬盘的信息。这种 CMOS 由电池供电所以系统断电时也不会遗失其中的内容。它 位于永不停止的系统实时时钟设备中。此 CMOS 内存的位置由系统 BIOS 来设置，它将通知 Linux 系统中有多少个 IDE 控制器与驱动器。Linux 使用这些从 BIOS 中发现的磁盘数据来建立对应此驱动器的 ide_hwif_t 结构。 许多现代 PC 系统使用 PCI 芯片组如 Intel 82430 VX 芯片组将 PCI EIDE 控制器封装在内。IDE 子系统使用 PCI BIOS 回调函数来定位系统中 PCI （E）IDE 控制器。然后对这些芯片组调用 PCI 特定查询例程。

每次找到一个 IDE 接口或控制器就有建立一个 ide_hwif_t 结构来表示控制器和与之相连的硬盘。在操作过程中 IDE 驱动器对 I/O 内存空间中的 IDE 命令寄存器写入命令。主 IDE 控制器的缺省控制和状态寄存器是 0x1F0 - 0x1F7。这个地址由早期的 IBM PC 规范设定。IDE 驱动器为每个控制器向 Linux 注册块缓冲 cache 和 VFS 节点并将其加入到 blk_dev 和 blkdevs 数组中。IDE 驱动器需要申请某个中断。一般主 IDE 控制器中断号为 14 而从属 IDE 控制器为 15。然而这些都可以通过命令行选项由核心来重载。IDE 驱动器同时还将 gendisk 入口加入到启动时发现的每个 IDE 控制器的 gendisk 链表中去。分区检查代码知道每个 IDE 控制器可能包含两个 IDE 硬盘。

8.5.3 SCSI 硬盘

SCSI（小型计算机系统接口）总线是一种高效的点对点数据总线，它最多可以支持 8 个设备，其中包括多个主设备。每个设备有唯一的标志符并可以通过盘上的跳线来设置。在总线上的两个设备间数据可以以同步或异步方式，在 32 位数据宽度下传输率为 40M 字节来交换数据。SCSI 总线上可以在设备间同时传输数据与状态信息。initiator 设备和 target 设备间的执行步骤最多可以包括 8 个不同的阶段。你可以从总线上 5 个信号来分辨 SCSI 总线的当前阶段。这 8 个阶段是：

BUS FREE

当前没有设备在控制总线且总线上无事务发生。

ARBITRATION

一个 SCSI 设备试图取得 SCSI 总线的控制权，这时它将其 SCSI 标志符放置到地址引脚上。具有最高 SCSI 标志符编号的设备将获得总线控制权。

SELECTION

当设备通过仲裁成功地取得了对 SCSI 总线的控制权后它必须向它准备发送命令的那个 SCSI 设备发出信号。具体做法是将目标设备的 SCSI 标志符放置在地址引脚上进行声明。

RESELECTION

在一个请求的处理过程中 SCSI 设备可能会断开连接。目标（target）设备将再次选择启动设备 （initiator)。不是所有的 SCSI 设备都支持此阶段。

COMMAND

此阶段中 initiator 设备将向 target 设备发送 6、10 或 12 字节命令。

DATA IN, DATA OUT

此阶段中数据将在 initiator 设备和 target 设备间传输。

STATUS

所有命令完毕后将进入此阶段，此时允许 target 设备向 initiator 设备发送状态信息以指示操作成功与否。

MESSAGE IN, MESSAGE OUT

此阶段附加信息将在 initiator 设备和 target 设备间传输。

Linux SCSI 子系统由两个基本部分组成，每个由一个数据结构来表示。

host

一个 SCSI host 即一个硬件设备：SCSI 控制权。NCR 810 PCI SCSI 控制权即一种 SCSI host。在 Linux 系统中可以存在相同类型的多个 SCSI 控制权，每个由一个单独的 SCSI host 来表示。这意味着一个 SCSI 设备驱动可以控制多个控制权实例。SCSI host 总是 SCSI 命令的 initiator 设备。

Device

虽然 SCSI 支持多种类型设备如磁带机、CD-ROM 等等，但最常见的 SCSI 设备是 SCSI 磁盘。SCSI 设备总是 SCSI 命令的 target。这些设备必须区别对待，例如象 CD-ROM 或者磁带机这种可移动设备，Linux 必须检测介质是否已经移动。不同的磁盘类型有不同的主设备号，这样 Linux 可以将块设备请求发送到正确的 SCSI 设备。

初始化 SCSI 子系统

SCSI 子系统的初始化非常复杂，它必须反映处 SCSI 总线及其设备的动态性。Linux 在启动时初始化 SCSI 子系统。 如果它找到一个 SCSI 控制器（即 SCSI hosts）则会扫描此 SCSI 总线来找出总线上的所有设备。然后初始化这些设备并通过普通文件和 buffer cache 块设备操作使 Linux 核心的其它部分能使用这些设备。初始化过程分成四个阶段：

首先 Linux 将找出在系统核心连接时被连入核心的哪种类型的 SCSI 主机适配器或控制器有硬件需要控制。每个 核心中的 SCSI host 在 builtin_scsi_hosts 数组中有一个 Scsi_Host_Template 入口。而 Scsi_Host_Template 结构中包含执行特定 SCSI host 操作, 如检测连到此 SCSI host 的 SCSI 设备的例程的入口指针。这些例程在 SCSI 子系统进行自我配置时使用同时它们还是支持此 host 类型的 SCSI 设备驱动的一部分。每个被检测的 SCSI host，即与真正 SCSI 设备连接的 host 将其自身的 Scsi_Host_Template 结构添加到活动 SCSI

hosts 的 scsi_hosts 结构链表中去。每个被检测 host 类型的实例用一个 scsi_hostlist 链表中的 Scsi_Host 结构来表示。例如一个包含两个 NCR810 PCI SCSI 控制器的系统的链表中将有两个 Scsi_Host 入口，每个控制器对应一个。每个 Scsi_Host 指向一个代表器设备驱动的 Scsi_Host_Template。

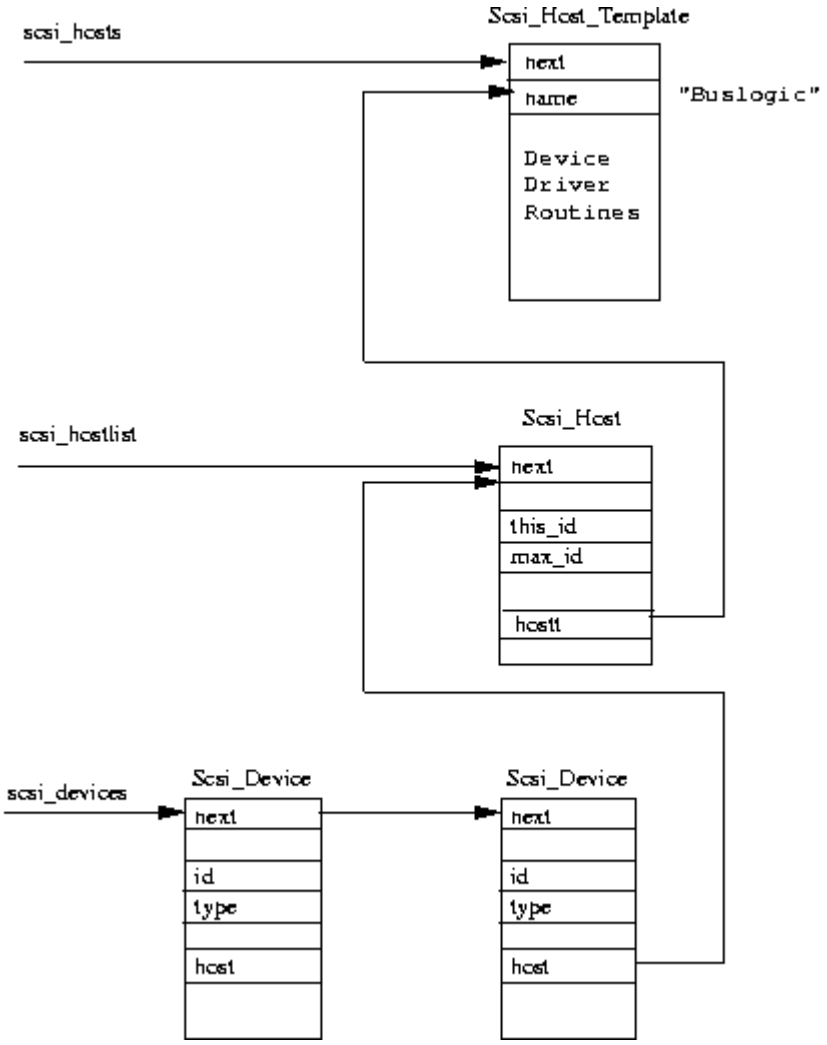


图 8.4 SCSI 数据结构

现在每个 SCSI host 已经找到，SCSI 子系统必须找出哪些 SCSI 设备连接哪个 host 的总线。SCSI 设备的编号是 从 0 到 7，对于一条 SCSI 总线上连接的各个设备，其设备编号或 SCSI 标志符是唯一的。SCSI 标志符可以通过设 备上的跳线来设置。SCSI 初始化代码通过在 SCSI 总线上发送一个 TEST_UNIT_READY 命令来找出每个 SCSI 设备。 当设备作出相应时其标志符通过一个 ENQUIRY 命令来读取。Linux 将从中得到生产厂商的名称和设备模式以及 修订版本号。SCSI 命令由一个 Scsi_Cmnd 结构来表示同时这些命令通过调用 Scsi_Host_Template 结构中的设备 驱动例程传递到此 SCSI host 的设备驱动中。被找到的每个 SCSI 设备用一个 Scsi_Device 结构来表示，每个指向 其父 Scsi_Host 结构。所有这些 Scsi_Device 结构被添加到 scsi_device 链表中。图 8.4 给出了这些主要数据结构 间的关系。

一共有四种 SCSI 设备类型：磁盘，磁带机，CD-ROM 和普通 SCSI 设备。每种类型的 SCSI 设备以不同的主块设备 类型单独登记到核心中。如果有多个类型的 SCSI 设备存在则它们只登记自身。每个 SCSI 设备类型，如 SCSI 磁盘 维护着其自身的设备列表。它使用这些表将核心块操作（file 或者 buffer cache）定向到正确的设备驱动或 SCSI host 上。每种 SCSI 设备类型用一个 Scsi_Device_Template 结构来表示。此结构中包含此类型 SCSI 设备的 信息以及执行各种任务的例程的入口地址。换句话说，如果 SCSI 子系统希望连接一个 SCSI 磁盘设备它将调用 SCSI 磁盘类型连接例程。如果有多个该种类型的 SCSI 设备被检测到则此 Scsi_Type_Template 结构将被添加到 scsi_devicelist 链表中。

SCSI 子系统的最后一个阶段是为每个已登记的 Scsi_Device_Template 结构调用 finish 函数。对于 SCSI 磁盘类型 设备它将驱动所有 SCSI 磁盘并记录其磁盘布局。同时还将添加一个表示所有连接在一起的 SCSI 磁盘的 gendisk 结构，如图 8.3。

发送块设备请求

一旦 SCSI 子系统初始化完成这些 SCSI 设备就可以使用了。每个活动的 SCSI 设备类型将其自身登记到核心以便 Linux 正确定向块设备请求。这些请求可以是通过 blk_dev 的 buffer cache 请求也可以是通过 blkdevs 的文件 操作。以一个包含多个 EXT2 文件系统分区的 SCSI 磁盘驱动器为例，当安装其中一个 EXT2 分区时系统是怎样将 核心缓冲请求定向到正确的 SCSI 磁盘的呢？

每个对 SCSI 磁盘分区的块读写请求将导致一个新的 request 结构被添加到对应此 SCSI 磁盘的 blk_dev 数组中的 current_request 链表中。如果此 request 正在被处理则 buffer cache 无需作任何工作；否则它必须通知 SCSI 磁盘子系统去处理它的请求队列。系统中每个 SCSI 磁盘用一个 Scsi_Disk 结构来表示。例如/dev/sdb1 的主设备 号为 8 而从设备号为 17；这样产生一个索引值 1。每个 Scsi_Disk 结构包含一个指向表示此设备的 Scsi_Device 结构。这样反过来又指向拥有它的 Scsi_Host 结果。这个来自 buffer cache 的 request 结构将被转换成一个描 叙 SCSI 命令的 Scsi_Cmd 结构，这个 SCSI 命令将发送到此 SCSI 设备同时被排入表示此设备的 Scsi_Host 结构。一 旦有适当的数据块需要读写，这些请求将被独立的 SCSI 设备驱动来处理。

8.6 网络设备

网络设备，即 Linux 的网络子系统，是一个发送与接收数据包的实体。它一般是一个象以太网卡的物理设备。 有些网络设备如 loopback 设备仅仅是一个用来向自身发送数据的软件。每个网络设备都用一个 device 结构来 表示。网络设备驱动在核心启动初始化网络时将这些受控设备登记到 Linux 中。device 数据结构中包含有有关 设备的信息以及用来支持各种网络协议的函数地址指针。这些函数主要用来使用网络设备传输数据。设备使用 标准网络支持机制来将接收到的数据传递到适当的协议层。所有传输与接收到的网络数据用一个 sk_buff 结构 来表示，这些灵活的数据结构使得网络协议头可以更容易的添加与删除。网络协议层如何使用网络设备以及 如何使用 sk_buff 来交换数据将在网络一章中详细描叙。本章只讨论 device 数据结构及如何发现与初始化网络。

device 数据结构包含以下有关网络设备的信息：

Name

与使用 mknod 命令创建的块设备特殊文件与字符设备特殊文件不同，网络设备特殊文件仅在于系统 网络设备发现与初始化时建立。它们使用标准的命名方法，每个名字代表一种类型的设备。多个 相同类型设备将从 0 开始记数。这样以太网设备被命名为/dev/eth0，/dev/eth1,/dev/eth2 等等。 一些常见的网络设备如下：

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

Bus Information

这些信息被设备驱动用来控制设备。irq 号表示设备使用的中断号。base address 指任何设备在 I/O 内存中的控制与状态寄存器地址。DMA 通道指此网络设备使用的 DMA 通道号。所有这些信息在设备初 始化时设置。

Interface Flags

它们描叙了网络设备的属性与功能：

IFF_UP 接口已经建立并运行
IFF_BROADCAST 设备中的广播地址有效
IFF_DEBUG 设备调试被使能
IFF_LOOPBACK 这是一个 loopback 设备
IFF_POINTTOPOINT 这是点到点连接（SLIP 和 PPP）
IFF_NOTRAILERS 无网络追踪者
IFF_RUNNING 资源已被分配
IFF_NOARP 不支持 ARP 协议
IFF_PROMISC 设备处于混乱的接收模式，无论包地址怎样它都将接收

IFF_ALLMULTI 接收所有的 IP 多播帧
IFF_MULTICAST 可以接收 IP 多播帧

Protocol Information

每个设备描叙它可以被网络协议层如何使用：

mtu

指不包括任何链路层头在内的，网络可传送的最大包大小。这个值被协议层用来选择适当 大小的包进行发送。

Family

这个 family 域表示设备支持的协议族。所有 Linux 网络设备的族是 AF_INET，互联网地址族。

Type

这个硬件接口类型描叙网络设备连接的介质类型。Linux 网络设备可以支持多种不同类型的 介质。包括以太网、X.25，令牌环，Slip，PPP 和 Apple Localtalk。

Addresses

结构中包含大量网络设备相关的地址，包括 IP 地址。

Packet Queue

指网络设备上等待传输的 sk_buff 包队列。

Support Functions

每个设备支持一组标准的例程，它们被协议层作为设备链路层的接口而调用。如传输建立和帧传输 例程以及添加标准帧头以及收集统计数据的例程。这些统计数据可以使用 ifconfig 命令来观察。

8.6.1 初始化网络设备

网络设备驱动可以象其它 Linux 设备驱动一样建立到 Linux 核心中来。每个潜在的网络设备由一个被 dev_base 链表指针指向的网络设备链表内部的 device 结构表示。当网络层需要某个特定工作执行时。它将调用大量网 络服务例程中的一个，这些例程的地址被保存在 device 结构内部。初始化时每个 device 结构仅包含一个初始 化或者检测例程的地址。

对于网络设备驱动有两个问题需要解决。首先是不是每个连接到核心中的网络设备驱动都有设备要控制。其 次虽然底层的设备驱动迥然不同，但系统中的以太网设备总是命名为/dev/eth0 和/dev/eth1。混淆网络设备 这个问题很容易解决。当每个网络设备的初始化例程被调用时，将得到一个指示是否存在当前控制器实例的 状态信息。如果驱动找不到任何设备，它那个由 dev_base 指向的 device 链表将被删除。如果驱动找到了设备 则它将用设备相关信息以及网络设备驱动中支撑函数的地址指针来填充此 device 数据结构。

第二个问题，即为以太网设备动态分配标准名称/dev/ethN 设备特殊文件的工作的解决方法十分巧妙。在设备 链表中有 8 个标准入口；从 eth0 到 eth7。它们使用相同的初始化例程，此初始化过程将依次尝试这些被建立到 核心中的以太网设备驱动直到找到一个设备。当驱动找到其以太网设备时它将填充对应的 ethN 设备结构。同时 此网络设备驱动初始化其控制的物理硬件并找出使用的 IRQ 号以及 DMA 通道等信息。如果驱动找到了此网络设备 的多个实例它将建立多个/dev/ethN device 数据结构。一旦所有 8 个标准/dev/ethN 被分配完毕则不会在检测 其它的以太网设备。

第九章 文件系统

本章主要描叙 Linux 核心对文件系统的支持，虚拟文件系统（VFS）以及 Linux 核心对实际文件系统的支持。

Linux 的最重要特征之一就是支持多种文件系统。这样它更加灵活并可以和许多其它种操作系统共存。在本文写作时 Linux 已经支持 15 种文件系统：ext,ext2,xia,minix,umsdos,msdos,vfat,proc,smb,ncp,iso9660,sysv,hpfs,affs 以及 ufs。毫无疑问，今后支持的文件系统类型还将增加。

Linux 和 Unix 并不使用设备标志符（如设备号或驱动器名称）来访问独立文件系统，而是通过一个将整个文件系统表示成单一实体的层次树结构来访问它。Linux 每安装(mount)一个文件系统时都会其加入到文件系统层次树中。不管是文件系统属于什么类型，都被连接到一个目录上且此文件系统上的文件将取代此目录中已存在的文件。这个目录被称为安装点或者安装目录。当卸载此文件系统时这个安装目录中原有的文件将再次出现。

当磁盘初始化时（使用 fdisk），磁盘中将添加一个描叙物理磁盘逻辑构成的分区结构。每个分区可以拥有一个独立文件系统如 EXT2。文件系统将文件组织成包含目录，软连接

等存在于物理块设备中的逻辑层次结构。包含文件系统的设备叫块设备。Linux 文件系统认为这些块设备是简单的线性块集合，它并不关心或理解底层的物理磁盘结构。这个工作由块设备驱动来完成，由它将对某个特定块的请求映射到正确的设备上去；此块所在硬盘的对应磁道、扇区及柱面数都被保存起来。不管哪个设备持有这个块，文件系统都必须使用相同的方式来寻找并操纵此块。Linux 文件系统不管（至少对系统用户来说）系统中有哪些不同的控制器控制着哪些不同的物理介质且这些物理介质上有几个不同的文件系统。文件系统甚至还可以不在本地系统而在通过网络连接的远程硬盘上。设有一个根目录内容如下的 SCSI 硬盘：

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

此时不管是用户还是程序都无需知道他们现在操纵的这些文件中的/C 实际上是位于系统第一个 IDE 硬盘上并已安装 VFAT 文件系统。在此例中/E 表示系统中第二个 IDE 控制器上的主 IDE 硬盘。至于第一个 IDE 控制器是 PCI 控制器和第二个则是控制 IDE CDROM 的 ISA 控制器无关紧要。当使用 modem 通过 PPP 网络协议来拨入网络时,可以将 Alpha AXP Linux 文件系统安装到/mnt/remote 目录下。

文件系统中的文件是数据的集合；包含本章内容的文件是一个名叫 filesystems.tex 的 ASCII 文件。文件系统不仅包含着文件中的数据而且还有文件系统的结构。所有 Linux 用户和程序看到的文件、目录、软连接及文件保护信息等都存储在其中。此外文件系统中必须包含安全信息以便保持操作系统的基本完整性。没人愿意使用一个动不动就丢失数据和文件的操作系统。

Linux 最早的文件系统是 Minix，它受限甚大且性能低下。其文件名最长不能超过 14 个字符（虽然比 8.3 文件名要好）且最大文件大小为 64M 字节。64M 字节看上去很大,但实际上一个中等的数据库将超过这个尺寸。 第一个专门为 Linux 设计的文件系统被称为扩展文件系统（Extended File System）或 EXT。它出现于 1992 年四月，虽然能够解决一些问题但性能依旧不好。1993 年扩展文件系统第二版或 EXT2 被设计出来并添加到 Linux 中。它是本章将详细讨论的文件系统。

将 EXT 文件系统添加入 Linux 产生了重大影响。每个实际文件系统从操作系统和系统服务中分离出来，它们之间通过一个接口层：虚拟文件系统或 VFS 来通讯。

VFS 使得 Linux 可以支持多个不同的文件系统，每个表示一个 VFS 的通用接口。由于软件将 Linux 文件系统的所有细节进行了转换, 所以 Linux 核心的其它部分及系统中运行的程序将看到统一的文件系统。 Linux 的虚拟文件系统允许用户同时能透明地安装许多不同的文件系统。

虚拟文件系统的设计目标是为 Linux 用户提供快速且高效的文件访问服务。同时它必须保证文件及其数据的正确性。这两个目标相互间可能存在冲突。当安装一个文件系统并使用时, Linux VFS 为其缓存相关信息。此缓存中数据在创建、写入和删除文件与目录时如果被修改，则必须谨慎地更新文件系统中对应内容。 如果能够在运行核心内看到文件系统的数据结构, 那么就可以看到那些正被文件系统读写的数据块。描叙文件与目录的数据结构被不断的创建与删除而设备驱动将不停地读取与写入数据。这些缓存中最重要的是 Buffer Cache，它被集成到独立文件系统访问底层块设备的例程中。当进行块存取时数据块首先将被放入 Buffer Cache 里并根据其状态保存在各个队列中。此 Buffer Cache 不仅缓存数据而且帮助管理块设备驱动中的异步接口。

9.1 第二代扩展文件系统（EXT2）

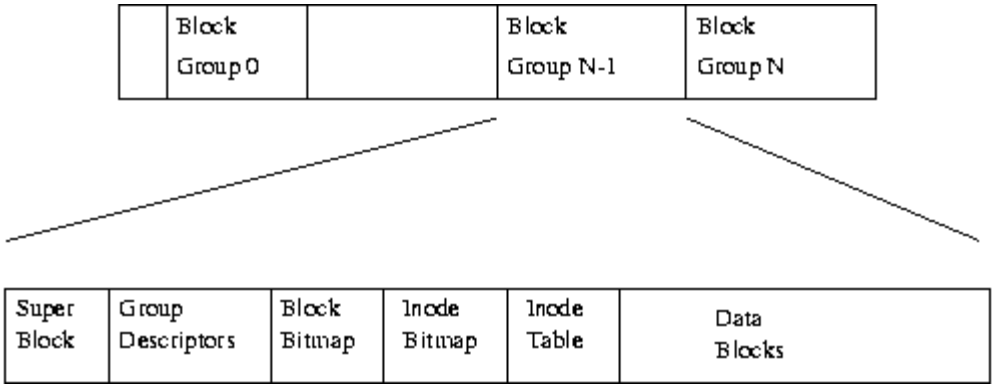


图 9.1 EXT2 文件系统的物理分布

第二代扩展文件系统由 Rey Card 设计，其目标是为 Linux 提供一个强大的可扩展文件系统。它同时也是 Linux 界中设计最成功的文件系统。

象很多文件系统一样, EXT2 建立在数据被保存在数据块中的文件内这个前提下。这些数据块长度相等且这个长度可以变化，某个 EXT2 文件系统的块大小在创建（使用 mke2fs）时设置。 每个文件的大小和刚好大于它的块大小正数倍相等。如果块大小为 1024 字节而一个 1025 字节长的文件将占据两个 1024 字节大小的块。这样你不得不浪费差不多一般的空间。我们通常需要在 CPU 的内存利用率和磁盘空间使用上进行折中。而大多数操作系统，包括 Linux 在内，为了减少 CPU 的工作负载而被迫选择相对较低的磁盘空间利用率。并不是文件中每个块都包含数据，其中有些块被用来包含描叙此文件系统结构的信息。EXT2 通过一个 inode 结构来描叙文件系统中文件并确定此文件系统的拓扑结构。 inode 结构描叙文件中数据占据哪个块以及文件的存取权限、文件修改时间及文件类型。EXT2 文件系统中的每个文件用一个 inode 来表示且每个 inode 有唯一的编号。文件系统中所有的 inode 都被保存在 inode 表中。 EXT2 目录仅是一个包含指向其目录入口指针的特殊文件（也用 inode 表示）。

图 9.1 给出了占用一系列数据块的 EXT2 文件系统的布局。对文件系统而言文件仅是一系列可读写的数据块。文件系统并不需要了解数据块应该放置到物理介质上什么位置，这些都是设备驱动的任务。无论何时只要文件系统需要从包含它的块设备中读取信息或数据，它将请求底层的设备驱动读取一个基本块大小整数倍的数据块。EXT2 文件系统将它所使用的逻辑分区划分成数据块组。每个数据块组将那些对文件系统完整性最重要的信息复制出来, 同时将实际文件和目录看作信息与数据块。为了发生灾难性事件时文件系统的修复，这些复制非常有必要。以下一节将着重描叙每个数据块组的内容。

9.1.1 The EXT2 Inode

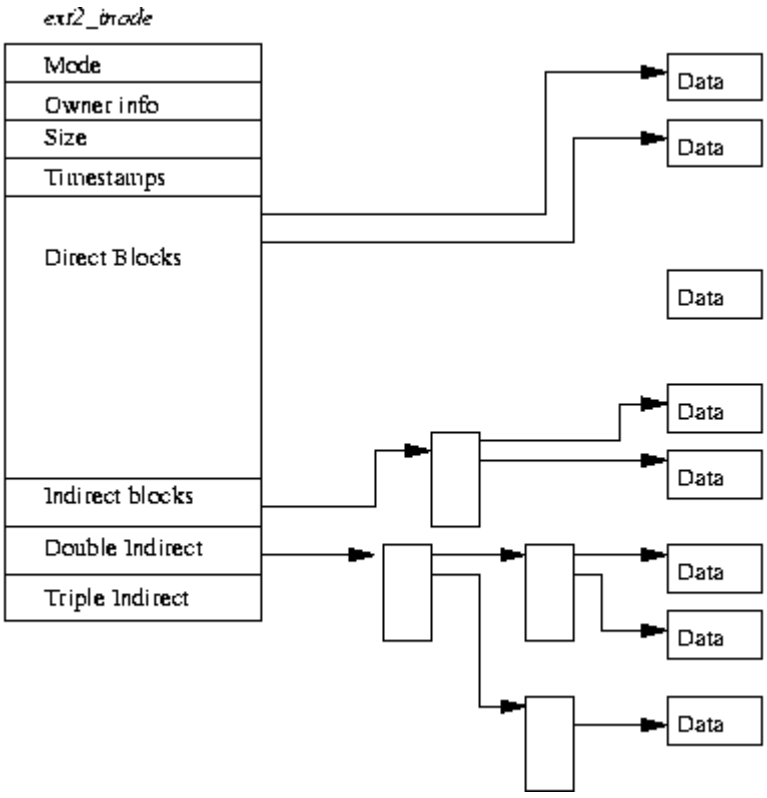


图 9.2 EXT2 Inode

在 EXT2 文件系统中 inode 是基本块；文件系统中的每个文件与目录由唯一的 inode 来描叙。每个数据块组的 EXT2 inode 被保存在 inode 表中，同时还有一个位图被系统用来跟踪已分配和未分配的 inode。图 9.2 给出了 EXT2 inode 的格式，它包含以下几个域：

mode

它包含两类信息；inode 描叙的内容以及用户使用权限。EXT2 中的 inode 可以表示一个文件、目录、符号连接、块设备、字符设备或 FIFO。

Owner Information

表示此文件或目录所有者的用户和组标志符。文件系统根据它可以进行正确的存取。

Size

以字节计算的文件尺寸。

Timestamps

inode 创建及最后一次被修改的时间。

Datablocks

指向此 inode 描叙的包含数据的块指针。前 12 个指针指向包含由 inode 描叙的物理块，最后三个指针包含多级间接指针。例如两级间接指针指向一块指针，而这些指针又指向一些数据块。这意味着访问文件尺寸小于或等于 12 个数据块的文件将比访问大文件快得多。

EXT2 inode 还可以描叙特殊设备文件。虽然它们不是真正的文件，但可以通过它们访问设备。所有那些位于/dev 中的设备文件可用来存取 Linux 设备。例如 mount 程序可把设备文件作为参数。

9.1.2 EXT2 超块

超块中包含了描叙文件系统基本尺寸和形态的信息。文件系统管理器利用它们来使用和维护文件系统。 通常安装文件系统时只读取数据块组 0 中的超块，但是为了防止文件系统被破坏，每个数据块组都包含了复制拷贝。超块包含如下信息：

Magic Number

文件系统安装软件用来检验是否是一个真正的 EXT2 文件系统超块。当前 EXT2 版本中为 0xEF53。

Revision Level

这个主从修订版本号让安装代码能判断此文件系统是否支持只存在于某个特定版本文件系统中的属性。同时它还是特性兼容标志以帮助安装代码判断此文件系统的新特性是否可以安全使用。

Mount Count and Maximum Mount Count

系统使用它们来决定是否应对此文件系统进行全面检查。每次文件系统安装时此安装记数将递增，当它等于最大安装记数时系统将显示一条警告信息“maxumal mount count reached, running e2fsck is recommended”。

Block Group Number

超块的拷贝。

Block Size

以字节记数的文件系统块大小，如 1024 字节。

Blocks per Group

每个组中块数目。当文件系统创建时此块大小被固定下来。

Free Blocks

文件系统中空闲块数。

Free Inodes

文件系统中空闲 Inode 数。

First Inode

文件系统中第一个 inode 号。EXT2 根文件系统中第一个 inode 将是指向'/'目录的目录入口。

9.1.3 EXT2 组标志符

每个数据块组都拥有一个描叙它结构。象超块一样，所有数据块组中的组描叙符被复制到每个数据块组中以防文件系统崩溃。每个组描叙符包含以下信息：

Blocks Bitmap

对应此数据块组的块分配位图的块号。在块分配和回收时使用。

Inode Bitmap

对应此数据块组的 inode 分配位图的块号。在 inode 分配和回收时使用。

Inode Table
对应数据块组的 inode 表的起始块号。每个 inode 用下面的 EXT2 inode 结构来表示。
Free blocks count, Free Inodes count, Used directory count
组描叙符放置在一起形成了组描叙符表。每个数据块组在超块拷贝后包含整个组描叙符表。EXT2 文件系统仅使用第一个拷贝（在数据块组 0 中）。其它拷贝都象超块拷贝一样用来防止主拷贝被破坏。

9.1.4 EXT2 目录

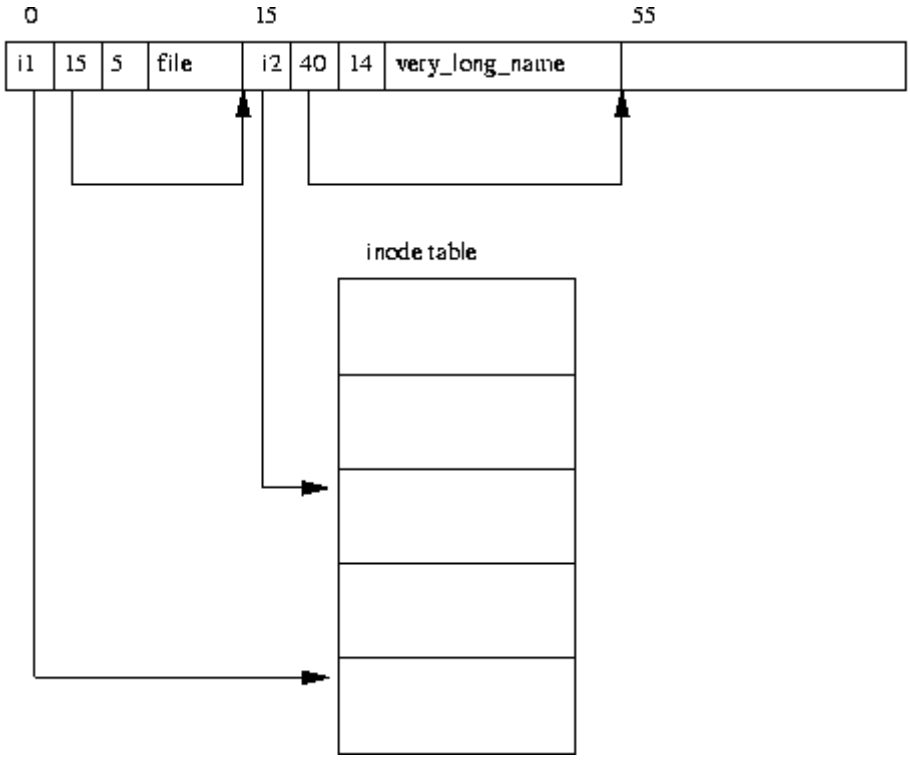


图 9.3 EXT2 目录

在 EXT2 文件系统中目录是用来创建和包含文件系统中文件存取路径的特殊文件。图 9.3 给出了内存中的目录入口布局。

目录文件是一组目录入口的链表，它们包含以下信息：
inode
对应每个目录入口的 inode。它被用来索引储存在数据块组的 Inode 表中的 inode 数组。 在图 9.3 中 file 文件的目录入口中有一个对 inode 号 11 的引用。
name length
以字节记数的目录入口长度。
name
目录入口的名称
每个目录的前两个入口总是"."和".."。它们分别表示当前目录和父目录。

9.1.5 在 EXT2 文件系统中搜寻文件

Linux 文件名的格式与 Unix 类似,是一系列以"/"隔开的目录名并以文件名结尾。/home/rusling/.cshrc 中/home 和/rusling 都是目录名而文件名为.cshrc。象 Unix 系统一样，Linux 并不关心文件名格式本身，它可以由任意可打印字符组成。为了寻找 EXT2 文件系统中表示此文件的 inode，系统必须将文件名从目录名中分离出来。
我们所需要的第一个 inode 是根文件系统的 inode，它被存放在文件系统的超块中。为读取某个 EXT2 inode, 我们必须在适当数据块组的 inode 表中进行搜寻。如果根 inode 号为 42 则我们需要数据块组 0 inode 表的第 42 个 inode。此根 inode 对应于一个 EXT2 目录，即根 inode 的 mode 域将它描叙成目录且其数据块包含 EXT2 目录入口。home 目录是许多目录的入口同时此目录给我们提供了大量描叙/home 目录的 inode。我们必须读取此目录以找到 rusling 目录入口，此入口又提供了许多描叙/home/rusling 目录的 inode。最后读取由/home/rusling 目录描叙的 inode 指向的目录入口以找出.cshrc 文件的 inode 号并从中取得包含在文件中信息的数据块。

9.1.6 改变 EXT2 文件系统中文件的大小

文件系统普遍存在的一个问题是碎块化。一个文件所包含的数据块遍布整个文件系统, 这使得对文件数据块的顺序访问越来越慢。EXT2 文件系统试图通过分配一个和当前文件数据块在物理位置上邻接或者至少位于同一个数据块组中的新块来解决这个问题。只有在这种分配策略失败时才在其它数据块组中分配空间。

当进程准备写某文件时, Linux 文件系统首先检查数据是否已经超出了文件最后一个被分配的块空间。如果是则必须为此文件分配一个新数据块。进程将一直等待到此分配完成; 然后将其余数据写入此文件。EXT2 块分配例程所作的第一件事是对此文件系统的 EXT2 超块加锁。这是因为块分配和回收将导致超块中某些域的改变，Linux 文件系统不能在同一时刻为多个进程进行此类服务。如果另外一个进程需要分配更多的数据块时它必须等到此进程完成分配操作为止。 在超块上等待的进程将被挂起直到超块的控制权被其当前使用者释放。对超块的访问遵循先来先服务原则，一旦进程取得了超块的控制则它必须保持到操作结束为止。如果系统中空闲块不多则此分配的将失败，进程会释放对文件系统超块的控制。

如果 EXT2 文件系统被设成预先分配数据块则我们可以从中取得一个。预先分配块实际上并不存在，它们仅仅包含在已分配块的位图中。我们试图为之分配新数据块文件所对应的 VFS inode 包含两个 EXT2 特殊域：prealloc_block 和 prealloc_count，它们分别代表第一个预先分配数据块的块号以及各自的数目。如果没有使用预先分配块或块预先分配数据块策略，则 EXT2 文件系统必须分配一个新块。它首先检查此文件最后一个块后的数据块是否空闲。从逻辑上来说这是让其顺序访问更快的最有效块分配策略。如果此块已被使用则它会在理想块周围 64 个块中选择一个。这个块虽然不是最理想但和此文件的其它数据块都位于同一个数据块组中。

如果此块还是不空闲则进程将在所有其它数据块组中搜寻，直到找到一空闲块。块分配代码将在某个数据块组中寻找一个由 8 个空闲数据块组成的簇。如果找不到那么它将取更小的尺寸。如果使用了块预先分配则它将更新相应的 prealloc_block 和 prealloc_count。

找到空闲块后块分配代码将更新数据块组中的位图并在 buffer cache 中为它分配一个数据缓存。这个数据缓存由文件系统支撑设备的标志符以及已分配块的块号来标志。缓存中

的数据被置 0 且缓存被标记成 dirty 以显示其内容还没有写入物理磁盘。最后超块也被标记为 dirty 以表示它已被更新并解锁了。如果有进程在等待这个超块则队列中的第一个进程将得到运行并取得对超块的独占控制。如果数据块被填满则进程的数据被写入新数据块中，以上的整个过程将重复且另一个数据块被分配。

9.2 虚拟文件系统(VFS)

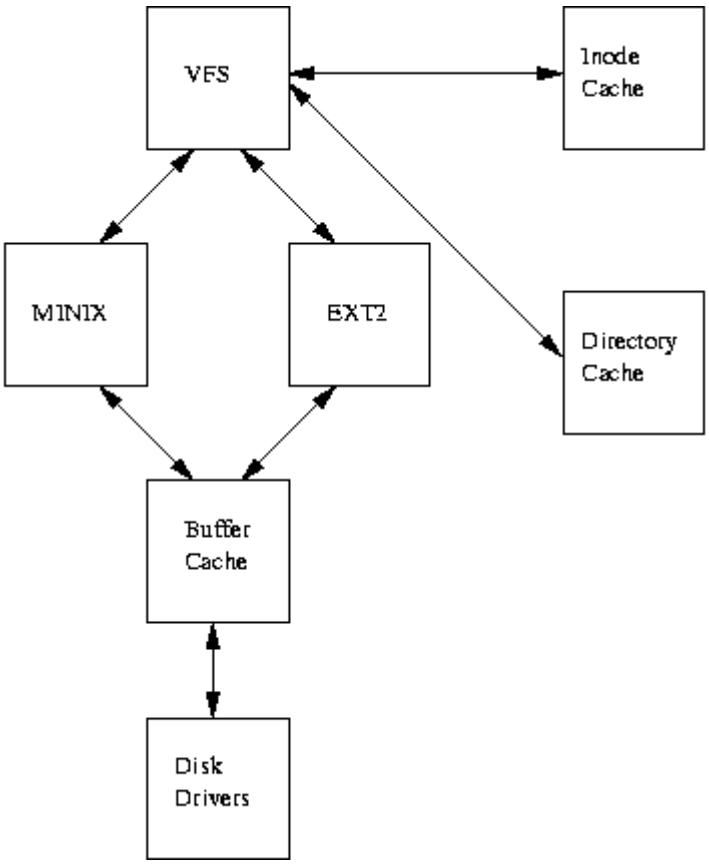


图 9.4 虚拟文件系统的逻辑示意图

图 9.4 给出了 Linux 核心中虚拟文件系统和实际文件系统间的关系。此虚拟文件系统必须能够管理在任何时刻 mount 到系统的不同文件系统。它通过维护一个描叙整个虚拟文件系统和实际已安装文件系统的结构来完成这个工作。

容易让人混淆的是 VFS 使用了和 EXT2 文件系统类似的方式：超块和 inode 来描叙文件系统。象 EXT2 inode 一样 VFS inode 描叙系统中的文件和目录以及 VFS 中的内容和拓扑结构。从现在开始我将用 VFS inode 和 VFS 超块来将它们和 EXT2 inode 和超块进行区分。

文件系统初始化时将其自身注册到 VFS 中。它发生在系统启动和操作系统初始化时。这些实际文件系统可以构造到核心中也可以设计成可加载模块。文件系统模块可以在系统需要时进行加载，例如 VFAT 就被实现成一个核心模块，当 mount VFAT 文件系统时它将

被加载。mount 一个基于块设备且包含根文件系统的文件系统时，VFS 必须读取其超块。每个文件系统类型的超块读取例程必须了解文件系统的拓扑结构并将这些信息映射到 VFS 超块结构中。VFS 在系统中保存着一组已安装文件系统的链表及其 VFS 超块。每个 VFS 超块包含一些信息以及一个执行特定功能的函数指针。例如表示一个已安装 EXT2 文件系统的超块包含一个指向 EXT2 相关 inode 读例程的指针。这个 EXT2 inode 读例程象所有文件系统相关读例程一样填充了 VFS inode 中的域。每个 VFS 超块包含此文件系统中第一个 VFS inode 的指针。对于根文件系统此 inode 表示的是"/"目录。这种信息映射方式对 EXT2 文件系统非常有效但是对其它文件系统要稍差。

系统中进程访问目录和文件时将使用系统调用遍历系统的 VFS inode。

例如键入 ls 或 cat 命令则会引起虚拟文件系统对表示此文件系统的 VFS inode 的搜寻。由于系统中每个文件与目录都使用一个 VFS inode 来表示，所以许多 inode 会被重复访问。这些 inode 被保存在 inode cache 中以加快访问速度。如果某个 inode 不在 inode cache 中则必须调用一个文件系统相关例程来读取此 inode。对这个 inode 的读将把此它放到 inode cache 中以备下一次访问。不经常使用的 VFS inode 将会从 cache 中移出。

所有 Linux 文件系统使用一个通用 buffer cache 来缓冲来自底层设备的数据以便加速对包含此文件系统的物理 设备的存取。

这个 buffer cache 与文件系统无关并被集成到 Linux 核心分配与读写数据缓存的机制中。让 Linux 文件系统独立于底层介质和设备驱动好处很多。所有的块结构设备将其自身注册到 Linux 核心中并提供基于块的一致性异步接口。象 SCSI 设备这种相对复杂的块设备也是如此。当实际文件系统从底层物理磁盘读取数据时，块设备驱动将从它们所控制的设备中读取物理块。buffer cache 也被集成到了块设备接口中。 当文件系统读取数据块时它们将被保存在由所有文件系统和 Linux 核心共享的全局 buffer cache 中。这些 buffer 由其块号和读取设备的设备号来表示。所以当某个数据块被频繁使用则它很可能能从 buffer cache 而不是磁盘中读取出来，后者显然将花费更长的时间。有些设备支持通过预测将下一次可能使用的数据提前读取出来。

VFS 还支持一种目录 cache 以便对经常使用的目录对应的 inode 进行快速查找。我们可以做一个这样的实验，首先我们对一个最近没有执行过列目录操作的目录进行列目录操作。第一次列目录时你可能发现会有较短的停顿但第二次操作时结果会立刻出现。目录 cache 不存储目录本身的 inode；这些应该在 inode cache 中，目录 cache 仅仅保存全目录名和其 inode 号之间的映射关系。

9.2.1 VFS 超块

每个已安装的文件系统由一个 VFS 超块表示；它包含如下信息：

Device

表示文件系统所在块设备的设备标志符。例如系统中第一个 IDE 硬盘的设备标志符为 0x301。

Inode pointers

这个 mounted inode 指针指向文件系统中第一个 inode。而 covered inode 指针指向此文件系统安装目录的 inode。根文件系统的 VFS 超块不包含 covered 指针。

Blocksize

以字节记数的文件系统块大小，如 1024 字节。

Superblock operations

指向此文件系统一组超块操纵例程的指针。这些例程被 VFS 用来读写 inode 和超块。

File System type
这是一个指向已安装文件系统的 file_system_type 结构的指针。
File System specific
指向文件系统所需信息的指针。

9.2.2 The VFS Inode

和 EXT2 文件系统相同，VFS 中的每个文件、目录等都用且只用一个 VFS inode 表示。每个 VFS inode 中的信息通过文件系统相关例程从底层文件系统中得到。VFS inode 仅存在于核心内存并且保存只要对系统有用，它们就会被保存在在 VFS inode cache 中。每个 VFS inode 包含下列域：
device
包含此文件或此 VFS inode 代表的任何东西的设备的设备标志符。
inode number
文件系统中唯一的 inode 号。在虚拟文件系统中 device 和 inode 号的组合是唯一的。
mode
和 EXT2 中的相同, 表示此 VFS inode 的存取权限。
user ids
所有者的标志符。
times
VFS inode 创建、修改和写入时间。
block size
以字节计算的文件块大小，如 1024 字节。
inode operations
指向一组例程地址的指针。这些例程和文件系统相关且对此 inode 执行操作，如截断此 inode 表示的文件。
count
使用此 VFS inode 的系统部件数。一个 count 为 0 的 inode 可以被自由的丢弃或重新使用。
lock
用来对某个 VFS inode 加锁，如用于读取文件系统时。
dirty
表示这个 VFS inode 是否已经被写过，如果是则底层文件系统需要更新。
file system specific information

9.2.3 注册文件系统

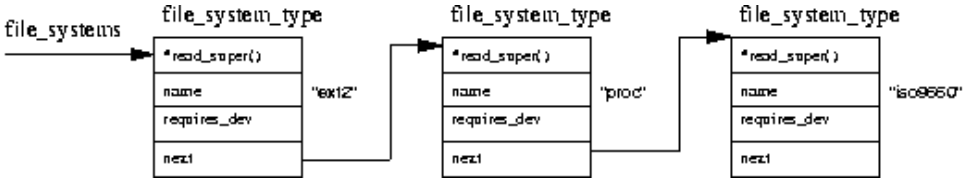


图 9.5 已注册文件系统

当重新建立 Linux 核心时安装程序会询问是否需要所有可支持的文件系统。核心重建时文件系统启动代码包含了所有那些编入核心的文件系统的初始化例程。

Linux 文件系统可构造成模块，此时它们会仅在需要时加载或者使用 insmod 来载入。当文件系统模块被加载时，它将向核心注册并在卸载时撤除注册。每个文件系统的初始化例程还将向虚拟文件系统注册，它用一个包含文件系统名称和指向其 VFS 超块读例程的指针的 file_system_type 结构表示。每个 file_system_type 结构包含下列信息：

Superblock read routine
此例程载文件系统的一个实例被安装时由 VFS 调用。
File System name
文件系统的名称如 ext2。
Device needed
文件系统是否需要设备支持。并不是所有的文件系统都需要设备来保存它。例如/proc 文件系统不需要块设备支持。
你可以通过查阅/proc/filesystems 可找出已注册的文件系统，如：

```
        ext2
nodev proc
        iso9660
```

9.2.4 安装文件系统

当超级用户试图安装一个文件系统时，Linux 核心首先使系统调用中的参数有效化。尽管 mount 程序会做一些基本的检查，但是它并不知道核心构造时已经支持那些文件系统，同时那些建议的安装点的确存在。看如下的一个 mount 命令：

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

mount 命令将传递三个参数给核心：文件系统名，包含文件系统的物理块设备以及此新文件系统要安装到的已存在的目录名。

虚拟文件系统首先必须做的是找到此文件系统。它将通过由链指针 file_systems 指向的 file_system_type 结 构来在所有已知文件系统中搜寻。

如果找到了一个相匹配的文件系统名,那么它就知道核心支持此文件系统并可得到读取此文件系统超块相关例程的指针。如果找不到,但文件系统使用了可动态加载核心模块,则操作仍可继续。此时核心将请求核心后台进程加载相应的文件系统模块。

接下来如果由 mount 传递的物理设备还没有安装, 则必须找到新文件系统将要安装到的那个目录的 VFS inode。 这个 VFS inode 可能在 inode cache 中也可能在支撑这个安装点所在文件系统的块设备中。一旦找到这个 inode 则将对它进行检查以确定在此目录中是否已经安装了其它类型的文件系统。多个文件系统不能使用相同目录作为安装点。

此时 VFS 安装代码必须分配一个 VFS 超块并将安装信息传递到此文件系统的超块读例程中。系统中所有的 VFS 超块都被保存在由 super_block 结构构成的 super_blocks 数组中, 并且对应此安装应有一个这种结构。超块读 例程将基于这些从物理设备中读取的信息来填充这些 VFS 超块域。对于 EXT2 文件系统此信息的转化过程十分 简便，仅需要读取 EXT2 超块并填充 VFS 超块。但其它文件系统如 MS-DOS 文件系统就不那么容易了。不管哪种文件系统，对 VFS 超块的填充意味着文件系统必须从支持它的块设备中读取描叙它的所有信息。如果块设备驱动不能从中读取或不包含这种类型文件系统则 mount 命令会失败。

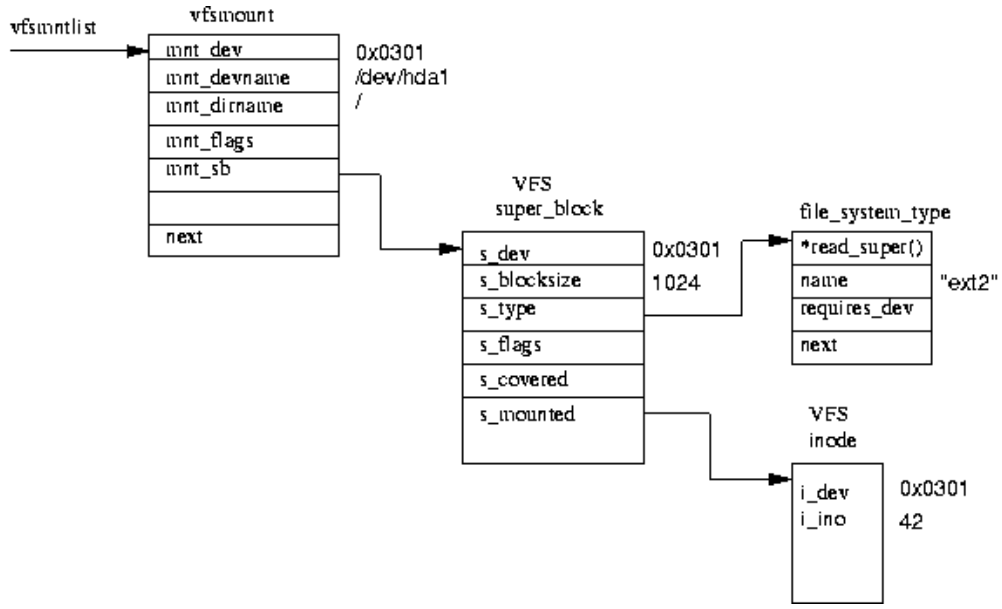


图 9.6 一个已安装的文件系统

每个文件系统用一个 vfsmount 结构来描叙。如图 9.6 所示。它们被排入由 vfsmntlist 指向的的链表中。

另外一个指针：vfsmnttail 指向链表的最后一个入口, 同时 mru_vfsmnt 指针指向最近使用最多的文件系统。 每个 vfsmount 结构中由以下部分组成：包含此文件系统的块设备的设备号，此文件系统安装的目录以及文件 系统安装时分配的 VFS 超块指针。VFS 超块指向这种类型文件系统和此文件系统根 inode 的 file_system_type 结构。一旦此文件系统被加载, 这个 inode 将一直驻留在 VFS inod cache 中。

9.2.5 在虚拟文件系统中搜寻文件

为了在虚拟文件系统中找到某个文件的 VFS inode，VFS 必须依次解析此文件名字中

的间接目录直到找到此 VFS inode。每次目录查找包括一个对包含在表示父目录 VFS inode 中的查找函数的调用。由于我们总是让每个文件系统的根可用并且由此系统的 VFS 超块指向它，所以这是一个可行方案。每次在实际文件系统中寻找 inode 时，文件系统将在目录 cache 中寻找相应目录。如果在目录 cache 中无相应入口则文件系统必须从底层文件系统或 inode cache 中取得此 VFS inode。

9.2.6 Creating a File in the Virtual File System

9.2.7 卸载文件系统

如果已安装文件系统中有些文件还在被系统使用则不能卸载此文件系统。例如有进程使用/mnt/cdrom 或其子目录时将不能卸载此文件系统。如果将要卸载的文件系统中有些文件还在被使用，那么在 VFS inode cache 中有与其对应的 VFS inode。通过在 inode 链表中查找此文件系统占用设备的 inode 来完成此工作。对应此已安装文件系统的 VFS 超块为 dirty，表示它已被修改过所以必须写回到磁盘的文件系统中。一旦写入磁盘,VFS 超块占用的内存将归还到核心的空闲内存池中。最后对应的 vfsmount 结构将从 vfsmntlist 中释放。

9.2.8 The VFS Inode Cache

操纵已安装文件系统时，它们的 VFS inode 将被连续读写。虚拟文件系统通过维护一个 inode cache 来加速对所有已安装文件系统的访问。每次 VFS inode 都可从 inode cache 中读取出来以加速对物理设备的访问。

VFS inode cache 以散列表形式实现，其入口时指向具有相同散列值的 VFS inode 链表。每个 inode 的散列值可通过包含此文件系统的底层物理设备标志符和 inode 号计算出来。每当虚拟文件系统访问一个 inode 时,系统将首先在 VFS inode cache 中查找。为了在 cache 中寻找 inode，系统先计算出其散列值然后将其作为 inode 散列表的索引。这样将得到指向一系列相同散列值的 inode 链表。然后依次读取每个 inode 直到找到那个具有相同 inode 号以及设备标志符的 inode 为止。

如果在 cache 中找到了此 inode 则它的 count 值递增以表示用户增加了一个,同时文件操作将继续进行。否则必须找到一个空闲 VFS inode 以便文件系统能从内存中读取此 inode。VFS 有许多种选择来取得空闲 inode。如果系统可以分配多个 VFS inode 则它将按如下步骤进行：首先分配核心页面并将其打碎成新的空闲 inode 并将其放入 inode 链表中。系统所有的 VFS inode 都被放到由 first_inode 指向的链表和 inode 散列表中。如果系统已经拥有所有 inode, 则它必须找到便于重新使用的 inode。那些 inode 最好 count 记数为 0; 因为这种 inode 没有谁在使用。很重要的 VFS inode，如文件系统的根 inode，其 count 域总是大于 0，所以它所使用的 inode 是不能被重新使用的。一旦找到可重用 inode 则应清除之: 其 VFS inode 可能为 dirty,必须要写入到文件系统中或者需要加锁，此时系统必须等到解锁时才能继续运行。

找到新的 VFS inode 后必须调用文件系统相关例程使用从底层实际文件系统中读出的内容填充它。在填充过程 中，此新 VFS inode 的 count 记数为 1 并被加锁以排斥其它进程对它的使用直到此 inode 包含有效信息为止。

为了取得真正需要的 VFS inode，文件系统可能需要存取几类其它 inode。我们读取一个目录时虽然只需要最后一级目录但是所有的中间目录也被读了出来。由于使用了 VFS inode cache，较少使用的 inode 将被丢弃而较多使用的 inode 将保存在 cache 中。

9.2.9 目录 Cache

为了加速对常用目录的访问，VFS 维护着一个目录入口 cache。

当在实际文件系统寻找目录时，有关此目录的细节将被存入目录 cache 中。当再次寻找此目录时，例如在此目录中列文件名或打开文件，则这些信息就可以在目录 cache 中找到。在实际实现中只有短目录入口（最多 15 个字 符）被缓存，这是因为那些较短目录名的目录正是使用最频繁的。例如/usr/X11R6/bin 这个短目录经常被 X server 所使用。

目录 cache 也由散列表组成，每个入口指向具有相同散列值的目录 cache 人口链表。散列函数使用包含此文件系统的设备号以及目录名称来计算在此散列表中的偏移值或者索引值，这样能很快找到被缓存的目录。 如果在 cache 中的搜寻消耗的时间太长或者甚至没有找到则使用此 cache 用处不大。

为了保证 cache 的有效性和及时更新，VFS 保存着一个最近最少使用（LRU）的目录 cache 人口链表。当首次查找此目录时其目录入口被首次放入 cache 中并添加到第一级 LRU 链表的尾部。在已经充满的 cache 中它代替位于 LRU 链表最前端的现存入口。此目录入口被再次使用时它将被放到第二级 LRU cache 链表的最后。此时需要将位于第二级 LRU cache 链表的最前端的那个替换掉。入口在链表前端的唯一原因是它们已经很久没被访问过了。如果被访问过那么它们将位于此链表的尾部附近。位于第二级 LRU cache 链表中的入口要比位于第一级 LRU cache 链表中的安全一些。

9.3 The Buffer Cache

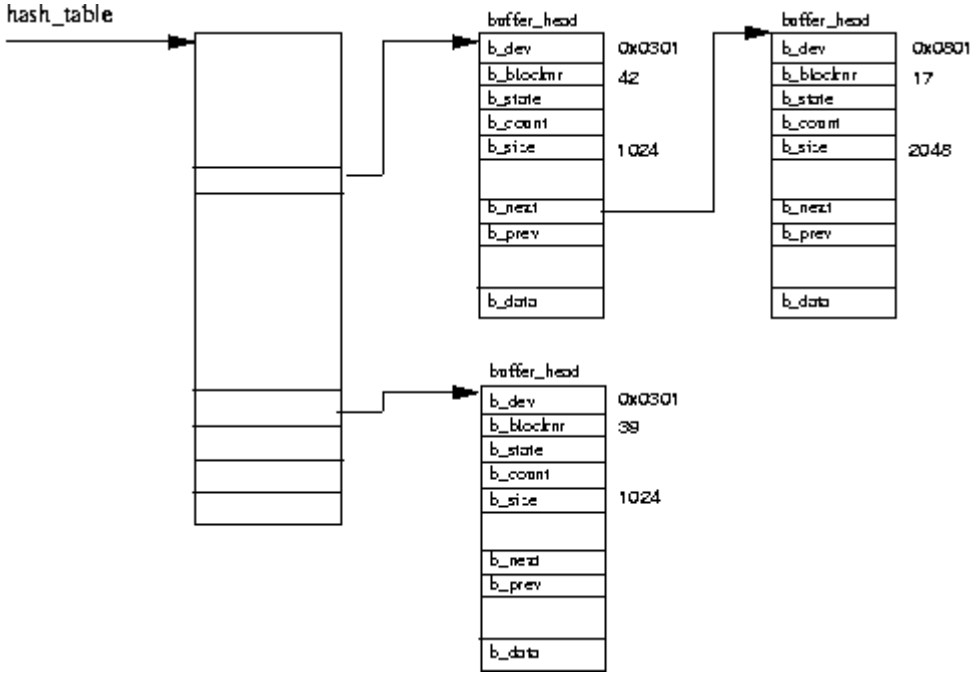


图 9.7 Buffer Cache 示意图

操纵已安装文件系统将产生大量对此块设备的读写请求。这些块读写请求都是通过标准核心例程调用以 buffer_head 结构形式传递到设备驱动中。它们提供了设备驱动所需的所有信息：表示设备的设备标志符以及请求的块号。所有块设备都被看成相同块大小的线性块集合。为了加速对物理块设备的访问，Linux 使用了一个块 buffer cache。系统中全部的块缓冲，包括那些没使用过的新缓冲都保存在此 buffer cache 中。这个 cache 被多个物理块设备共享；任何时刻此 cache 中都有许多属于不同系统块设备且状态不同的块缓冲。如果有效数据可以从 buffer cache 中找到则将节省大量访问物理设备的时间。任何对块设备读写的块缓冲都被放入此 cache 中。随时间的变化有些块缓冲可能将会被此 cache 中删除以为更需要它的缓冲腾出空间，如果它被频繁使用则可以一直保存在此 cache 中。

此 cache 中的块缓冲由设备标志符以及缓冲对应的块号来唯一的表示。它由两个功能部分组成。其一是空闲块缓冲链表。它为每个可支持的块大小提供了一个链表并且系统中的空闲块缓冲在创建或者被丢弃时都被排入此链表中。当前可支持的块大小为 512、1024、2048、4096 与 8192 字节。其二是 cache 自身。它是用一组指向具有相同散列索引值的缓冲链的散列表。这个散列索引值通过其自身的设备标志符与数据块设备的块号来产生。图 9.7 给出了一个带有一些入口的散列表。块缓冲要么在空闲链表中要么在此 buffer cache 中。如果在 buffer cache 中则它们按照最近最少使用（LRU）链表来排列。 对于每种缓冲类型都有一个 LRU 链表，系统使用它们来对某种缓冲进行操作，如将带新数据的缓冲写入到磁盘上。缓冲的类型表示其当前状态，Linux 现在支持以下集中类型：

clean

未使用的新缓冲

locked

等待写入且加锁的缓冲

dirty

dirty 缓冲。它们包含新的有效数据，但目前没被调度执行写操作。

shared

共享缓冲

unshared

以前被共享但现在没有被共享的缓冲

当文件系统需要从其底层物理设备读取一个缓冲块时，它将首先在 buffer cache 里寻找。如果在此 buffer cache 中找不到则它将从适当大小的空闲链表中取得一个 clean 状态的节点, 同时将新缓冲添加到 buffer cache 中去。如果所需的缓冲位于 buffer cache 中，那么它可能已经或没有更新。如果没有被更新或者它为新块则文件系统必须请求相应的数据驱动从磁盘中读取该数据块。

为了让此 buffer cache 运行更加有效并且在使用此 buffer cache 的块设备之间合理的分配 cache 入口，系统必须对其进行维护。Linux 使用 bdflush 核心后台进行来对此 cache 执行许多琐碎工作,但有时作为使用 cache 的结构自动进行。

9.3.1 bdflush 核心后台进程

bdflush 是对过多的 dirty 缓冲系统提供动态响应的简单核心后台进程；这些缓冲块中包含必须被写入到硬盘上的数据。它在系统启动时作为一个核心线程运行，其名字叫"kflushd"。你可以使用 ps 命令看到此系统进程。通常情况下此进程一直在睡眠直到系统中的 dirty 缓冲数目增大到一定数目。当分配与丢弃缓冲时,系统中 dirty 缓冲的数目将做一个统计。如果其数目超过某个数值则唤醒 bdflush 进程。缺省的阀值为 60%， 但是如果系统急需缓冲则任何时刻都可能唤醒 bdflush。使用 update 命令可以看到和改变这个数值。

```
# 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).
```

但有数据写入缓冲使之变成 dirty 时，所有的 dirty 缓冲被连接到一个 BUF_DIRTY LRU 链表中，bdflush 会将适当数目的缓冲块写到磁盘上。这个数值的缺省值为 500。

9.3.2 update 进程

update 命令不仅仅是一个命令；它还是一个后台进程。当作为超级用户运行时（在系统初始化时）它将周期性调用系统服务例程将老的 dirty 缓冲冲刷到磁盘上去。它所完成的这个工作与 bdflush 类似。当一个 dirty 缓冲完成此操作后，它将把本应写入到各自磁盘上的时间标记到其中。update 每次运行时它将在系统的所有 dirty 缓冲中查找那些冲刷时间已过期的。这些过期缓冲都被写入到磁盘。

9.4 /proc 文件系统

/proc 文件系统真正显示了 Linux 虚拟文件系统的能力。事实上它并不存在-不管时/proc 目录还是其子目录和文件都不真正的存在。但是我们是如何能够执行 cat /proc/devices 命令的？/proc 文件系统象一个真正的文 件系统一样将向虚拟文件系统注册。然而当有对/proc 中的文件和目录的请求发生时，VFS 系统将从核心中的数据中临时构造这些文件和目录。例如核心的/proc/devices 文件是从描叙其设备的内核数据结构中产生出来。/proc 文件系统提供给用户一个核心内部工作的可读窗口。几个 Linux 子系统，如在 modules 一章描叙的 Linux 核心模块都在/proc 文件系统中创建入口。

9.5 设备特殊文件

和所有 Unix 版本一样 Linux 将硬件设备看成特殊的文件。如/dev/null 表示一个空设备。设备文件不使用文件 系统中的任何数据空间，它仅仅是对设备驱动的访问入口点。EXT2 文件系统和 Linux VFS 都将设备文件实现成特殊的 inode 类型。有两种类型的设备文件：字符与块设备特殊文件。在核心内部设备驱动实现了类似文件的操作过程：我们可以对它执行打开、关闭等工作。字符设备允许以字符模式进行 I/O 操作而块设备的 I/O 操作需要通过 buffer cache。当对一个设备文件发出的 I/O 请求将被传递到相应的设备驱动。常常这种设备文件并不是一个真正的设备驱动而仅仅是一个伪设备驱动，如 SCSI 设备驱动层。设备文件通过表示设备类型的主类型标志符和表示单元或主类型实例的从类型来引用。例如在系统中第一个 IDE 控制器上的 IDE 硬盘的主设备号为 3 而其第一个分区的从标志符为 1。所以执行 ls -l /dev/hda1 将有如下结果：

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

在核心内部每个设备由唯一的 kdev_t 结构来表示，其长度为两字节，首字节包含从设备号而尾字节包含主设备号。 上例中的核心 IDE 设备为 0x0301。表示块或者字符设备的 EXT2 inode 在其第一个直接块指针包含了设备的主从设备号。当 VFS 读取它时，表示它的 VFS inode 结构的 i_rdev 域被设置成相应的设备标志符。

第十章 网络

网络和 Linux 是密切相关的。从某种意义来说 Linux 是一个针对 Internet 和 WWW 的产品。它的开发者和用户用 Web 来交换信息思想、程序代码，而 Linux 自身常常被用来支持各种组织机构的网络需求。这一章讲的是 Linux 如何支持如 TCP/IP 等网络协议的。

TCP/IP 协议最初是为支持 ARPANET（一个美国政府资助的研究性网络）上计算机通讯而设计的。ARPANET 提出了一些网络概念如包交换和协议分层（一个协议使用另一个协议提供的服务）。ARPANET 于 1988 年隐退，但是它的继承人(NSF1 NET 和 Internet）却变得更大了。现在我们所熟知的万维网 World Wide Web 就是从 ARPANET 演变过来的，它自身支持 TCP/IP 协议。Unix TM 被广泛应用于 ARPANET，它的第一个网络版本是 4.3 BSD。Linux 的网络实现是以 4.3 BSD 为模型的，它支持 BSD sockets(及一些扩展)和所有的 TCP/IP 网络。选这 个编程接口是因为它很流行，并且有助于应用程序从 Linux 平台移植到其它 Unix TM 平台。

10.1 TCP/IP 网络简介

这一部分简单介绍一下 TCP/IP 网络的主要原理，而不是进行详细地讲述。在 IP 网络中，每台机器都有一个 IP 地址，一个 32 位的数字，它唯一地标识这台机器。WWW 是一个非常巨大并且迅速增长的网络，每台连在上面的机器都必须有一个独立的 IP 地址。IP 地址由四个用点分开的数字表示，如 16.42.0.9。这个 IP 地址实际上分成两个部分：网络地址和主机地址，每部分的长度是可以变化的（有好几类 IP 地址）。以 16.42.0.9 为例，网络地址是 16.42，主机地址是 0.9。主机地址又进一步分为子网地址和主机地址。还是以 16.42.0.9 为例，子网地址是 16.42.0，主机地址是 16.42.0.9。这样的子划分可以允许某部门划分他们自己的子网络。例如，如果 16.42 是 ACME 计算机公司的网络地址，则 16.42.0 可能是子网 0，16.42.1 可能是子网 1。 这些子网可以是分别建立的，可能租用电话线或用微波进行相互间通讯。IP 地址由网络管理员分配，用 IP 子网可以很好地管理网络。IP 子网的管理员可以自由分配子网内的 IP 地址。

通常，IP 地址是比较难记的，而名称则容易多了，象 linux.acme.com 就比 16.42.0.9 要好记一些。但是必须有一些机器来将网络名称转变为 IP 地址。这些名称被静态地定义在 /etc/hosts 文件中或者 Linux 能请求域名服务器(DNS)来解析它。这种情况下，本地主机必须知道一个或一个以上的 DNS 服务器并且这些服务器要将其名称指定到 /etc/resolv.conf 中。

当你想要与另一台计算机连接时，比如说你想阅读一个 Web 页，你的 IP 地址就会被用来与那台机器交换数据。这些数据被包含在一些 IP 包中，每个 IP 包都有一个 IP 头用来包含源机器的 IP 地址和目的机器的 IP 地址，校验和以及其它的有用信息。IP 包的校验和用来让 IP 包的接收端判断 IP 包是否在传输过程中发生错误，譬如说由于电话线路的问题而引起的错误。应用程序想要传输的数据可能被分成很多个容易处理的小包。IP 数据包的大小是根据传输媒体的变化而不同的；以太网包通常比 PPP 包要大一些。目的主机在将数据送给接收端应用程序前需要将这些包重新拚装起来。如果你从一个比较慢的站点访问一个有大量图象的 Web 页，就会看到数据的分割与重组。

同一子网内的主机之间可以直接发送 IP 包，而其它的 IP 包将被送到一个特定的主机：

网关。网关（或路由器）是用来连接多个 IP 子网的，它们会转发送从子网内来的 IP 包。例如，如果子网 16.42.1.0 和 16.42.0.0 之间通过一个网关相连，那么任何从子网 0 发往子网 1 的包必须由网关指引，网关可以帮这些包找到正确的路线。本地主机建立路由表用以为 IP 包找到正确的机器。每一个目的 IP 都有一个条目在路由表中，用以告诉 Linux 将 IP 包送到哪一台主机。这些路由表是随网络的拓扑结构变化而动态变化的。

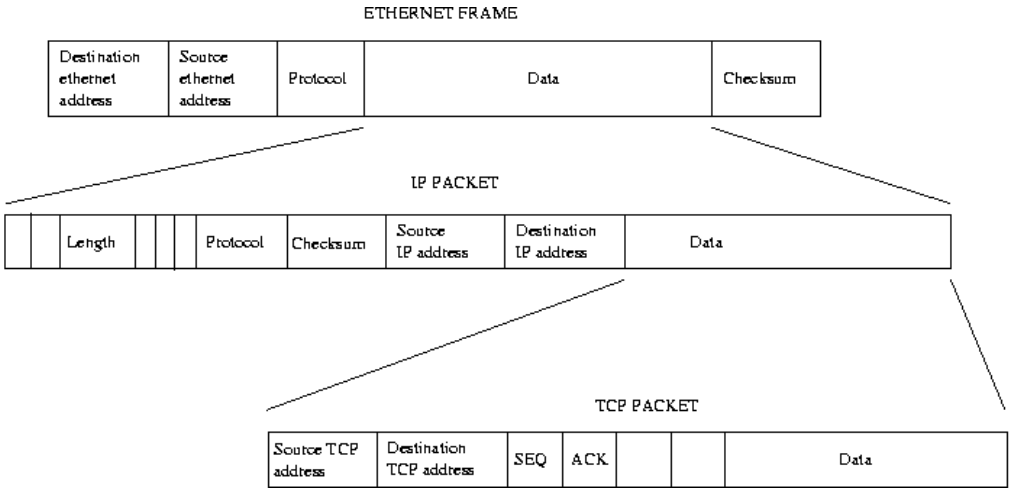


图 10.1: TCP/IP 协议层

IP 协议是一个传输层的协议，其它协议可以用它来传输数据。传输控制协议（TCP）是一个可靠的端对端的协议，它用 IP 来传送和接收它自己的包。正如 IP 包有它自己的头一样，TCP 也有它自己的头。TCP 是一个面向连接的协议，两个网络应用程序通过一个虚连接相连，即使它们之间可能隔着很多子网、网关、路由器。TCP 可靠地传送和接收两应用程序间的数据，并保证数据不会丢失。当用 IP 来传输 TCP 包时，IP 包的数据段就是 TCP 包。每一个通讯主机的 IP 层负责传送和接收 IP 包。用户数据报协议（UDP）也用 IP 层来传输它的包，不象 TCP，UDP 不是一个可靠的协议，但它提供了一种数据报服务。有多个协议可以使用 IP 层，接收 IP 包的时候必需知道该 IP 包中的数据是哪个上层协议的，因此 IP 包头中有个一字节包含着协议标识符。例如，当 TCP 请求 IP 层传输一个 IP 包时，IP 包的包头中用标识符指明该包包含一个 TCP 包，IP 接收层用该标识符决定由哪一协议来 接收数据，这个例子中是 TCP 层。当应用程序通过 TCP/IP 进行通讯时，它们不仅要指定目标的 IP 地址，而且还 要指定应用的端口地址。一个端口地址唯一地标识一个应用，标准的网络应用使用标准的端口地址；如，Web 服务使用 80 端口。这些已登记的端口地址可在 /etc/services 中看到。

这一层的协议不仅仅是 TCP、UDP 和 IP。IP 协议层本身用很多种物理媒介将 IP 包从一个主机传到其它主机。这些媒介可以加入它们自己的协议头。以太网层就是一个例子，但 PPP 和 SLIP 不是这样。一个以太网络允许很个主机同时连接到同一根物理电缆。传输中的每一个以太网帧可以被所有主机看见，因此每一以太网设备有个唯一的地址。任何传送给该地址的以太网帧被有该地址的以太网设备接收，而其它主机则忽略该帧。这个唯一的地址内置于每一以太网设备中，通常是在网卡出厂时就写在 SROM2 中了。以太网地址有 6 个字节长，如： 08-00-2b-00-49-A4 。一些以太网地址是保留给多点传送用的，送往这些地址的以太网帧将被网上所有的主机接收。以太网帧可以携带很多种协议（作为数据），如 IP 包，并且也包括它们头中的协议标识符。这使得以太网层能正确地接收 IP 包并将它

们传给 IP 层。

为了能通过象以太网这样的多连接协议传送 IP 包，IP 层必须找到每一 IP 主机的以太网地址。IP 地址仅仅是一个地址概念，以太网设备有它们自身的物理地址。从另一方面说，IP 地址是可以被网络管理员根据需要来分配和再分配的，而网络硬件只对含有它们自己的物理地址或多点传送地址的以太网帧作出响应。Linux 用地址 解析协议（ARP）来允许机器将 IP 地址转变成真正的硬件地址，如以太网地址。如果一个主机想知道某一 IP 地址对应的硬件地址，它就用一个多点传送地址将一个包含了该 IP 地址的 ARP 请求包发给网上所有节点，拥有该 IP 地址的的目标主机则响应一个包含物理硬件地址的 ARP 应答。ARP 不仅仅局限于以太网设备，它能够用来在其它一些物理媒介上解析 IP 地址，如 FDDI。那些不支持 ARP 的网络设备会被标记出来，Linux 将不会用 ARP。还有一个提供相反功能的反向地址解析协议（RARP），用来将物理网络地址转变为 IP 地址。这一协议常常被网关用来响应包含远程网络 IP 地址的 ARP 请求。

10.2 Linux TCP/IP 网络层

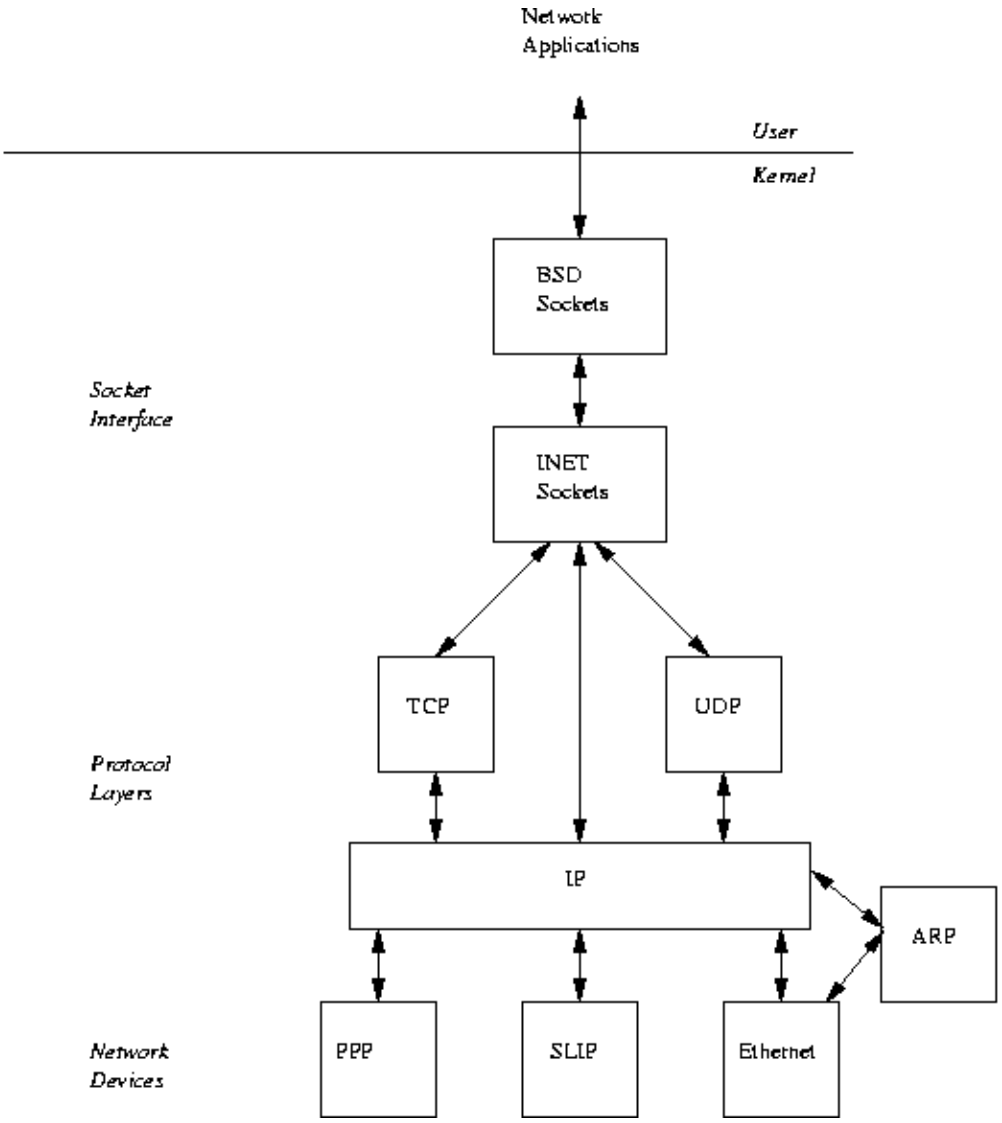


图 10.2: Linux 网络层

正如网络协议本身，图 10.2 显示出 Linux 用一系列相互连接层的软件实现 Internet 协议地址族。BSD 套接字（BSD sockets）由专门处理 BSD sockets 通用套接字管理软件处理。它由 INET sockets 层来支持，这一层为基于 IP 的协议 TCP 和 UDP 管理传输端点。UDP（用户数据报协议）是一个无连接协议而 TCP（传输控制协议）是个可靠的端对端协议。传输 UDP 包时，Linux 不知道也不关心是否它们安全到达目的地。TCP 包则被 TCP 连接两端编号以保证传输的数据被正确接收。IP 层包含了实现 Internet 协议的代码。这些代码给要传输的数据加上 IP 头，并知道如何把传入的 IP 包送给 TCP 或 UDP。在 IP 层以下，是网络设备来支持所有 Linux 网络工作，如 PPP 和以太网。网络设备不总是物理设备；一些象 loopback 这样的设备是纯软件设备。标准的 Linux 设备用 mknod 命令建立，网络设备要用

底层软件发现并初始化它。建立一个有适当的以太网设备驱动在内的内核后，你就可以看到 /dev/eth0 。ARP 协议位于 IP 层与支持 ARP 的协议之间。

10.3 BSD Socket 接口

这是一个通用的接口，它不仅支持各种网络工作形式，而且还是一个交互式通讯机制。一个套接字描述一个通讯连接的一端，两个通讯程序中各自有一个套接字来描述它们自己那一端。套接字可以被看成一个专门的管道，但又不象管道，套接字对它们能容纳的数据量没有限制。Linux 支持多种类型的套接字。这是因为每一类型的套接字有它自己的通信寻址方法。Linux 支持下列套接字地址族或域：

UNIX Unix 域套接字
INET Internet 地址族支持通过 TCP/IP 协议的通信
AX25 Amateur radio X25
IPX Novell IPX
APPLETALK Appletalk DDP
X25 X25

有一些套接字类型支持面向连接的服务类型。并非所有的地址族能支持所有的服务类型。Linux BSD 套接字支持下列套接字类型：

Stream
这些套接字提供可靠的双工顺序数据流，能保证传送过程中数据不丢失，不被弄混和复制。Internet 地址中的 TCP 协议支持流套接字。
Datagram
这些套接字提供双工数据传送，但与流套接字不同，这里不保证信息的到达。即使它们到达了，也不能保其到达的顺序，甚至不能保证被复制和弄混。这类套接字由 Internet 地址族中的 UDP 协议支持。
Raw
允许直接处理下层协议(所以叫“Raw”)。例如，有可能打开一个 raw 套接字到以太网设备，看 raw IP 数据传输。
Reliable Delivered Messages
与数据报很象，但它能保证数据的到达。
Sequenced Packets
与流套接字相似，但的数据包大小是固定的。
Packet
这不是一个标准的 BSD 套接字类型，而是一个 Linux 特定的扩展，它允许在设备级上直接处理包。

客户服务器模式下使用套接字进行通信。服务器提供一种服务，客户使用这种服务。Web 服务器就是一个例子，它提供网页，而客户端，或者说浏览器，来读这些网页。服务器要使用套接字，首先要建立套接字并将它与一个名称绑定。名称的格式由套接字的地址族来定，是服务器的本地有效地址。套接字的名称或地址用结构 sockaddr 来指定。一个 INET 套接字还与一个端口地址绑定。已注册的端口号可在 /etc/services 中找到；例如，Web 服务的端口号是 80。将套接字与地址绑定以后，服务器不可以监听指定的绑定了的地址上的

引入连接请求。请求的发起者，客户端，建立一个套接字并通过它来发出一个连接请求到指定的目标服务器地址。对于一个 INET 套接字，服务器地址是它的 IP 地址和它的端口号。这些引入请求必须通过各种协议层找到目的地址，然后等待服务器的监听套接字。服务器收到引入请求后可以接收或拒绝它。如果决定接收，服务器必需建立一个新一套接字来接收请求。当一个套接字被用来监听引入连接请求时，它就不能用来支持连接了。连接建立后两端就可以自由地发送和接收数据了。最后，当不再需要连接时，就将之关闭。要注意保证在传输过程正确处理数据包。

对 BSD socket 进行准确操作要依赖于它下面的地址族。设置 TCP/IP 连接与设置 amateur radio X.25 连接有很大不同。象虚拟文件系统一样，Linux 从 BSD socket 层抽象出 socket 接口，应用程序和 BSD socket 由每个地址族的特定软件来支持。内核初始化时，地址族被置入内核中并将自己注册到 BSD socket 接口。之后，当应用程序建立用使用 BDS sockets 时，在 BSD socket 与它支持的地址族之间将产生一个联接。这一联接是由交叉链接数据结构和地址族表特定支持程序产生。例如，每当应用程序建立一个新的 socket，就会有一个 BSD socket 接口用的地址族特定 socket 建立程序。

构造内核时，一些地址族和协议被置入 protocols 向量。每个由它的名称来表征，例如，“INET”和它的初始程序地址。当套接口启动时被初始化时，要调用每一协议和初始程序。对 socket 地址族来说，这会导致它们注册一套协议操作。这是一套例程，其中的每一例程执行一个特定的针对那一地址族的操作。已注册的协议操作被存在 pops 向量，一个指向 proto_ops 数据结构的向量中。

proto_ops 结构由地址族类型和一系列指向与特定地址族对应的 socket 操作例程的指针组成。pops 向量通过地址族标识符来索引，如 Internet 地址族标识符（AF_INET 是 2）。

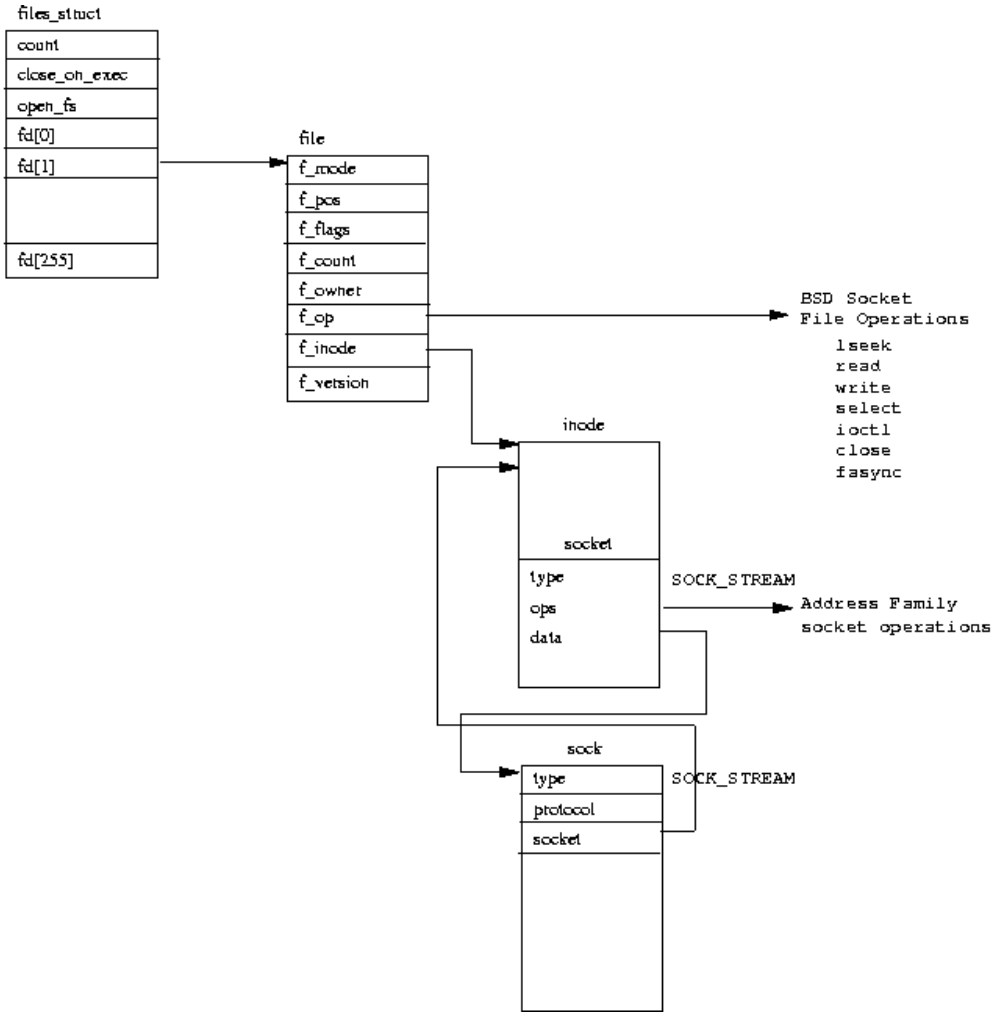


图 10.3: Linux BSD Socket 数据结构

10.4 INET Socket 层

INET socket 层支持包括 TCP/IP 协议在内的 internet 地址族。如前所述，这些协议是分层的，一个协议使用另一个协议的服务。Linux 的 TCP/IP 代码和数据结构反映了这一分层模型。它与 BSD socket 层的接口要通过一系列 Internet 地址族 socket 操作，这一操作是在网络初始化时就已经注册到 BSD socket 层的。这些都与其它已注册的地址族一起保存在 pops 向量中。BSD socket 层从已注册的 INET proto_ops 数据结构中调用 INET 层 socket 支持例程来为它执行工作。例如，一个地址族为 INET 的 BSD socket 建立请求，将用到下层的 INET socket 的建立函数。在这些操作中，BSD socket 层把用来描述 BSD socket 的 socket 结构传构到 INET 层。为了不把 BSD socket 与 TCP/IP 的特定信息搞混，INET socket 层使用它自己的数据结构，sock ，它与 BSD socket 结构相连。这一联接关系可以从图 10.3 中看出。它用 BSD socket 的 data 指针来连接 sock 结构与 BSD socket 结构。这意味着后来的 INET socket 调用能够很容易地重新找到 sock 结构。 sock 结构的协议操作指针也在初始化时建立，它依赖与被请求的协议。如果请求的是 TCP，那么 sock 结构的协议操作指

针将指向 TCP 连接所必需的 TCP 协议操作集。

10.4.1 建立 BSD socket

系统建立一个新的 socket 时，通过标识符来确定它的地址族，socket 类型和协议。

首先，从 pops 向量中搜索与被请求的地址族相匹配的地址族。它可能是一个作为核心模块来实现的一个特定的地址族，这样，在其能继续工作前， kerneld 守护进程必须加载这一模块。分配一个新的 socket 结构来代表 BSD socket。实际上 socket 结构是 VFS inode 结构的一部分，分配一个 socket 实际上就是分配一个 VFS inode 。除非你认为 socket 操作能和普通的文件操作一样，否则会觉得这好象很奇怪。所有的文件用 VFS inode 结构来表示，为了支持文件操作，BSD socket 必须也用 VFS inode 来表示。

最新建立的 BSD socket 结构包含一个指向地址族特定 socket 例程的指针，可以用来从 pops 向量中找到 proto_ops 结构。它的类型被设置成被请求的 socket 类型：SOCK_STREAM，SOCK_DGRAM 等等之一。调用地址族特定创建例程使用保存在 proto_ops 结构中的地址。

从当前过程 fd 向量中分配一个自由的文件描述符，对 file 结构所指向的进行初始化。包括将文件操作指针设置为指向由 BSD socket 接口支持的 BSD socket 文件操作集。任何操作将被引到 socket 接口，通过调用它的地址族操作例程将它们传到支持的地址族。

10.4.2 将地址与 INET BSD socket 绑定

为了能监听输入的 internet 连接请求，每个服务器必须建立一个 INET BSD socket，并将地址与其绑定。绑定操作主要在 INET socket 层内处理，下面的 TCP 和 UDP 协议层提供一些支持。与一个地址绑定了的 socket 不能用来进行任何其它的通讯工作，也就是说：socket 的状态必须是 TCP_CLOSE 。 sockaddr 结构包含了与一个任意的端口号绑定的 IP 地址。通常绑定的 IP 地址已经分配给了一个网络设备，该设备支持 INET 地址族且其接口是可用的。可以在系统中用 ifconfig 命令来查看哪一个网络接口是当前激活的。IP 地址也可以是广播地址，全 1 或全 0。这是些特定的地址，用以表示发送给任何人 3。如果机器充当一个透明的代理或防火墙，则 IP 地址可被指定为任一个 IP 地址，但只有有超级用户权限的进程能绑定到任何一个 IP 地址。绑定的 IP 地址被存在 sock 结构中的 recv_addr 和 saddr 字段。端口号是可选的，如果没有指定，将任意指定一个。按惯例，小于 1024 的端口号不能被没有超级用户权限的进程使用。如果下层网络没有分配端口号，则分配一个大于 1024 的端口号。

下层网络设备接收的包必须由经正确的 INET 和 BSD socket 才能被处理。因此，UDP 和 TCP 维护了一些 hash 表用来在输入 IP 消息内查找地址并将它们导向正确的 socket/sock 对。TCP 是一个面向连接的协议，因而涉及处理 TCP 包的信息比用于处理 UDP 包的信息多。

UDP 维护着一张已分配 UDP 端口表， udp_hash 表。由指向 sock 数据结构的指针组成，通过一个基于端口号的 hash 函数来索引。UDPhash 表比允许的端口号的数目小得多（udp_hash 为 128 或者说是 UDP_HTABLE_SIZE ）表中的一些项指向一个 sock 结构链，该链用每个 sock 结构中的 next 指针来将每个 sock 连接起来。

TCP 是十分复杂的，它包括几个 hash 表。但实际上 TCP 在绑定操作时没有将 sock 结构与其 hash 表绑定，它仅仅检查被请求的端口号当前没被使用。 sock 结构是在 listen 操

作时被加入 TCP 的 hash 表的。

复习提要: What about the route entered?

10.4.3 在 INET BSD Socket 上建立连接

建立一个 socket，如果没有用它来监听连入请求，那么就能用它来发连出请求。对于面向无连接的协议如 UDP 来说，这一 socket 操作并不做许多事，但对于面向连接的协议如 TCP 来说，这一操作包括了在两个应用间建立一个虚连接。

一个连出连接操作只能由一个在正确状态下的 INET BSD socket 来完成；换句话说，socket 不能是已建立连接的，并且有被用来监听连入连接。这意味着 BSD socket 结构必须是 SS_UNCONNECTED 状态。UDP 协议没有在两个应用间建立虚连接，任何发出的消息都是数据报，这些消息可能到达也可能不到达目的地。但它不支持 BSD socket 的 connect 操作。建立在 UDP 的 INET BSD socket 上的连接操作简单地设置远程应用的地址：IP 地址和 IP 端口号。另外，它还设置路由表入口的 cache 以便这一 BSD socket 在发用 UDP 包时不用再次查询路由数据库（除非这一路由已经无效）。INET sock 结构中的 ip_route_cache 指针指向路由缓存信息。如果没有给出地址信息，缓存的路由和 IP 地址信息将自动地被用来发送消息。UDP 将 sock 的状态改为 TCP_ESTABLISHED 。

对于基于 TCP BSD socket 的连接操作，TCP 必须建立一个包括连接信息的 TCP 消息，并将它送到目的 IP。TCP 消息包含与连接有关的信息，一个唯一标识的消息开始顺序号，通过初始化主机来管理的消息大小的最大值，及发送与接收窗口大小等等。在 TCP 内，所有的消息都是编号的，初始的顺序号被用来作为第一消息号。Linux 选用一个合理的随机值来避免恶意协议冲突。每一从 TCP 连接的一端成功地传到另一端的消息要确认其已经正确到达。未确认的消息将被重传。发送与接收窗口的大小是第一个确认到达之前消息的个数。消息尺寸的最大值与网络设备有关，它们在初始化请求的最后时刻确定下来。如果接收端的网络设备的消息尺寸最大值更小，则连接将以小的一端为准。应用程序发出连接请求后必须等待目标应用程序的接受或拒绝连接的响应。TCP sock 期望着一个输入消息，它被加入 tcp_listening_hash 以便输入 TCP 消息能被指向这一 sock 结构。TCP 同时也开始计时，当目标应用没有响应请求，则连出连接请求超时。

10.4.4 监听 INET BSD Socket

socket 与地址绑定后，能监听指定地址的连入连接请求。一个网络应用程序能监听 socket 而不用先将地址 与之绑定；在这个例子中，INET socket 层找到一个未用的端口号（对这一协议）并自动将它与 socket 绑定。 监听 socket 函数将 socket 状态设成 TCP_LISTEN ，并做其它连入连接所需要的工作。

对于 UDP sockets，改变 socket 的状态就足够了，而 TCP 现在加了 socket 的 sock 数据结构到两个 hash 表中并激活， tcp_bound_hash 表和 tcp_listening_hash 表。这两个表都通过一个基于 IP 端口号的 hash 函数来索引。

无论何时，一个激活的监听 socket 接收一个连入的 TCP 连接请求，TCP 都要建立一个新的 sock 结构来描述它。最终接收时，这个 sock 结构将成为 TCP 连接的底层。它也复制包含连接请求的 sk_buff ，并将它放到监听 sock 结构的 receive_queue 中排队。复制的 sk_buff 包含一个指向新建立的 sock 结构的指针。

10.4.5 接收连接请求

UDP 不支持连接的概念，接收 INET socket 连接请求只适用于 TCP 协议，一个监听 socket 接收操作从原始的监听 socket 中复制新的 socket 结构。接收操作透过支持的协议层，本例是 INET，来接收任何连入连接请求。如果下层协议，如 UDP，不支持连接，INET 协议层接收操作将失败。否则接收操作透过真实协议层，本例是 TCP。接收操作可以是阻塞或非阻塞。在非阻塞情况下，如果没有连入连接可接收，则接收操作失败，新建的 socket 结构被废弃。在阻塞情况下，网络应用程序执行接收操作将加上一个等待队列并将之挂起，直到接收到 TCP 连接请求。当接收一个连接请求后，包含请求的 sk_buff 被废弃，并且 sock 数据结构返回到 INET socket 层，在那与一个新的更早建立的 socket 结构连接。新 socket 文件描述符(fd)号返回给网络应用程序，然后，应用程序就能在 socket 操作中将这一文件描述符用于新建立的 INET BSD socket。

10.5 IP 层

10.5.1 Socket 缓存

每一层协议用另外层提供的服务，这样使用多层网络协议会有一个问题：每个协议都要在传送数据时都要 加上协议头和协议尾，而数据到达时又要将之去掉。这样，在不同的协议间要有数据缓存，每一层需要知道特 定协议的头和尾放在哪个位置。一个解决办法就是在每一层中都拷贝缓存，但这样做效率就很低。Linux 用 socket 缓存或者说 sk_buffs 来在协议层与网络设备驱动之间交换数据。 sk_buffs 包括指针和字段长度，这样每 个协议层就可以通过标准的函数或“方法”来操作应用程序数据。

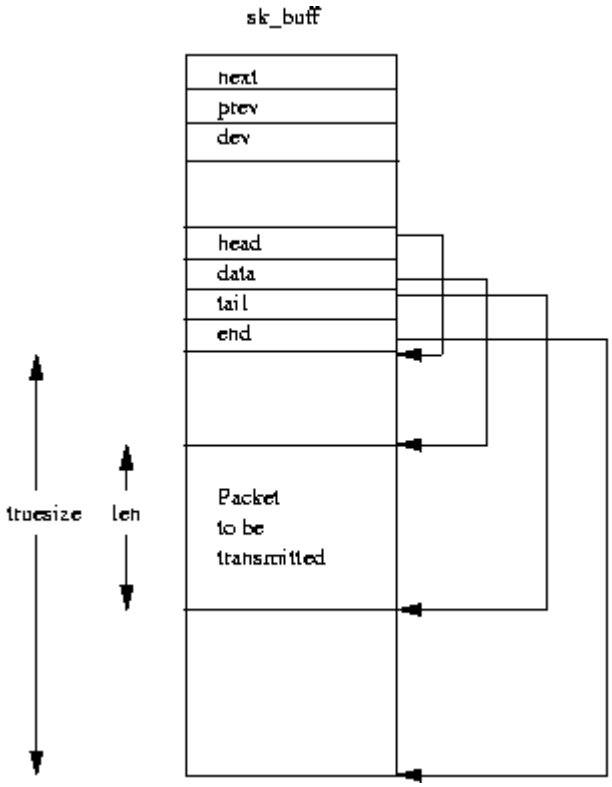


图 10.4: Socket 缓存 (sk_buff)

图 10.4 显示了 sk_buff 数据结构；每个 sk_buff 有一个数据块与之相连。sk_buff 有四个指针，这些指针 用来操作和管理 socket 缓存的数据：

head
指向内存中数据区的开头。这一指针在 sk_buff 和其相关的数据块分配时就固定了。
data
指向当前协议数据的开头。这一指针是随当前拥有 sk_buff 的是哪个协议层而变化的。
tail
指向当前协议数据的结尾。同样，这一指针也是随当前拥有 sk_buff 的是哪个协议层而变化的。
end
指向内存中数据区的结尾。这一指针在 sk_buff 和其相关的数据块分配时固定。

len 和 truesize 这两个字段分别用来描述当前协议包长度和数据缓存总体长度。sk_buff 处理代码提供标准的操作来向应用程序增加和移除协议头和协议尾。这就可以安全地操作 sk_buff 中的 data , tail 和 len 字段。
push
它把 data 指针指向数据区的开始并增加 len 。用于在要传输的数据开始处增加协议头。
pull
它把 data 指针从数据区的开始处移到数据区的结尾处，并减小 len 。用于在已接收的数据开始处移除协议头。
put

它把 tail 指针指向数据区的结尾处，并增加 len 。用于在要传输的数据结尾处增加数据或协议信息。

trim

它把 tail 指针指向数据区的开始处，并减小 len 。用于在已接收的数据尾移除数据或协议信息。

sk_buff 结构还包含了用于一些指针，用于在处理过程中存入 sk_buff 的双连接环路列表。通用 sk_buff 例 程可以将 sk_buff 加入到这些列表的前面或后面，也可以删除它们。

10.5.2 接收 IP 包

第 dd-chapter 章描述了 Linux 的网络设备是如何置入内核并初始化的。一系列 device 数据结构在 dev_base 表中相互连接起来。每个 device 结构描述了它的设备并提供回调例程，当需要网络驱动来执行工作时，网络协议层调用这些例程。这些函数与传输的数据及网络设备地址紧密相关。当一个网络设备从网上接收包时，它必须将接收的数据转换成 sk_buff 结构。这些 sk_buff 则被网络驱动加入到了 backlog 队列中。

如果 backlog 队列太长，则丢弃接收的 sk_buff。准备好要运行时，网络底层将被设置标志。

当网络底层按计划开始运行后，处理 backlog 队列之前，任何等待着被传输的网络包都由它来处理。 sk_buff 决定哪些层处理被接收的包。

Linux 网络层初始化时，每一协议通过将 packet_type 结构加入到 ptype_all 列表或 ptype_base hash 表中来 注册它自己。packet_type 结构包含了协议类型，一个指向网络设备的指针，一个指向协议的接收数据处理例程的指针，最后还包括一个指向列表链或 hash 链中下一个 packet_type 结构的指针。ptype_all 链用于监听从网络设备上接收的所有包，通常不使用它。ptype_base hash 表是被协议标识符弄乱的，用于决定哪个协议将接收传入的网络包。网络底层通过两个表中的一个或多个 packet_type 项来匹配传入 sk_buff 的协议类型。协议可以和多于一个的项相匹配，如在监听网上所有的传输时要复制多个 sk_buff 。 sk_buff 将通过被匹配协议处理例程。

10.5.3 发送 IP 包

应用程序交换数据时要传输包，否则由网络协议在建立连接或支持一个已建立的连接时来生成。无论数据 是由哪种方法生成的，都要建立一个 sk_buff 来包含数据，当通过协议层时，这些协议层会加上各种头。

sk_buff 需要通过网络设备传输。首先协议，如 IP，需要确定是哪个网络设备在用。这有赖于包的最佳路 由。对于通过 modem 连入一个简单网络，如通过 PPP 协议，的计算机来说，路由的选择是很简单的。包应该通过 本地环路设备发送给本地主机，或发送给 PPP modem 连接的网关。对于连在以在网上的计算机来说，连接在网 络上的计算机越多，路由越复杂。

对于每一个被传输的 IP 包，IP 用路由表来为目的 IP 地址解析路由。从路由表中成功地找到目的 IP 时将返回一个描述了要使用的路由的 rtable 结构。这包括要用到的源 IP 地址，网络 device 结构的地址，有时还有预建立的硬件头。这些硬件头是网络设备特定的，包含了源和目的的物理地址和其它的特定媒体信息。如果网络设备是一个以太网设备，硬

件头则应如图 10.1 所示，并且源和目的地址应是物理的以太网地址。硬件头在路由的时候会缓存起来，因为必须将它加到每一个要传输的 IP 包中。硬件头包含的物理地址要用 ARP 协议来解析。传出的包在地址被解析后才会发出。解析了地址后，硬件头被缓存起来以便以后的 IP 包在使用这一接口时不需要再使用 ARP。

10.5.4 数据分块

每个网络设备都有一个包大小的最大值，发送或接收数据包不能比这一值大。IP 协议允许将数据分成更小单元以便网络设备能处理。IP 协议头有分块字段，它里面包含了一个标志和分割偏移量。

当 IP 包准备要传输时，IP 找到网络设备来将 IP 包发送出去。这个设备是从 IP 路由表中找到的。每一 device 结构中有一项 mtu ，用来描述最大传输单元（以字节为单位）。如果设备的 mtu 比要传输的 IP 包的包大小要小，则 IP 包必须被分割成更小的单元。每一单元用一个 sk_buff 结构来表征；它的 IP 头会被做上标记以标识它是一个分块了的包，其中还包含分割偏移量。最后一个包被标识为最后 IP 单元。如果在分块过程中，IP 不能分配 sk_buff ，则传输失败。

接收 IP 分块单元要比发送它们要麻烦一些，因为这些 IP 单元可能以任何顺序到达，必须所有的单元都接收 到了以后才能重新将它们组装起来。每接收一个 IP 包都要检查其是否是 IP 分割单元。在第一个 IP 分割单元到达 时，IP 会建立一个新的 ipq 结构，这一结构与用于 IP 单元重组的 ipqueue 列表相连。当接收到更多的 IP 单元时，先找到正确的 ipq 结构，并为每个单元新建立一个 ipfrag 结构。每个 ipq 结构唯一地描述一个接收 IP 分割单元的，包括它的源和目的 IP 地址，上层协议标识和本 IP 帧的标识。当接收到所有的 IP 分割单元后，将它们重新组成一个 sk_buff ，然后交给上层协议处理。每个 ipq 中包含一个定时器，它在每接收到一个合法的单元后重新时。如果定时器到时， ipq 结构和它的一些 ipfrag 结构将被丢弃，传送的信息则被假定为丢失。然后提交给层协议来重传该信息。

10.6 地址解析协议 (ARP)

地址解析协议担当了一个把 IP 地址翻译成物理硬件地址如以太网地址的角色。IP 在将数据（以 sk_buff 的形式)通过设备驱动传送时需要这一转换。

它执行各种检查，来看是否这一设备需要硬件头，是否需要重建包的硬件头。Linux 缓存了硬件头，这样可以避频繁重建。如果需要重建硬件头，则调用设备指定的硬件头重建例程。所有的以太网设备使用相同的头重例程，这些例程将目的 IP 地址转换成物理地址。

ARP 协议本身是很简单的，它包括两个消息类型，ARP 请求与 ARP 应答。ARP 请求包含了需要解析的 IP 地址，ARP 应答（希望它）包含被解析的 IP 地址，硬件地址。ARP 请求向连接在网络上的所有主机广播，因此，对于以网，所有连在网上的机器都能看到 ARP 请求。拥有 ARP 请求中的 IP 地址的机器将发出包含了它自己的物理地址 ARP 应答。

ARP 协议在 Linux 中是围绕 arp_table 结构表来建立的，每个结构描述一个 IP 到物理地址的转换。这些表项 在需要进行 IP 地址解析时生成，在随时间变旧时被删除。每个 arp_table 结构有如下字段:

last used	本 ARP 项最近一次使用的时间
last updated	本 ARP 项最近一次更新的时间

flags	描述本项的状态，如是否完成等
IP address	本项描述的 IP 地址
hardware address	要解析的硬件地址
hardware header	指向缓存硬件头的指针
timer	是个 timer_list 项，用于 ARP 请求没有响应时的超时
retries	ARP 请求重试的次数
sk_buff queue	等待 IP 地址解析的 sk_buff 项列表

ARP 表包括了指向 arp_table 链的指针（ arp_table 向量)。缓存这些表项可以加速对它们的访问，每个表项用 IP 地址的最后两个字节来生成索引，然后就可以查找表链以找到正确的表项。Linux 也以 hh_cache 结构的形式来缓存 arp_table 项的预建的硬件头。

请求一个 IP 地址解析并且没有相应的 arp_table 项时，ARP 必须发送一个 ARP 请求。它在表和 sk_buff 队列中 生成一个新的 arp_table 项， sk_buff 包含了需要进行地址解析的网络包。发送 ARP 请求时运行 ARP 定时器。如果没有响应，ARP 将重试几次，如果仍然没有响应，ARP 将删除该 arp_table 项。同时会通知队列中等待 IP 地址解析的 sk_buff 结构，传送它们的上层协议将处理这一失败。UDP 不关心丢包，而 TCP 则会建立 TCP 连接进行重传。如果 IP 地址的所有者返回了它的硬件地址，则 arp_table 项被标记为完成，队列中的 sk_buff 将被删除，传输动作继续。硬件地址被写到每个 sk_buff 的硬件头中。

ARP 协议层必须响应 ARP 请求。它注册它的协议类型(ETH_P_ARP)，生成一个 packet_type 结构。这表示它将检查网络设备收到的所有 ARP 包。与 ARP 应答一样，这包括 ARP 请求。用保存在接收设备的 device 结构中的硬件地址来生成 ARP 应答。

网络拓扑结构会随时间改变，IP 地址会被重新分配不同的硬件地址。例如，一些拨号服务为每一次新建的连接分配一个 IP 地址。为了使 ARP 表包含这些数据项，ARP 运行一个周期性的定时器，用来查看所有的 arp_table 项中哪一个超时。要注意不要移除包含一个或多个缓存硬件头的项。移除这些项是很危险的，因为其它的数据结构要用到它们。一些 arp_table 项被标记为永久的，它们不会被释放。ARP 表不能太大；每个 arp_table 项会消耗一些核心内存。要分配一个新的表项而 ARP 表的大小已经到达它的最大值时，就要查找并删除最老的表项。

10.7 IP 路由

IP 路由函数决定了将预定的有指定 IP 地址的 IP 包送到哪。在传送 IP 包时有很多种选择。能最终到达目标吗？如果能，要用到哪个网络设备呢？如果有多于一个的网络设备可被使用，哪一个是较好的呢？IP 路由数据库里存的信息给出了这些问题的答案。有两个数据库，最重要的一个是 Forwarding Information Database。这是一个有关已知的目的 IP 和它们的最佳路由的祥细列表，route cache 则用来快速找到目的 IP 的路由。和其它的缓一样，它包含的只是常用的路由；它的内容来自 Forwarding Information Database。

通过 IOCTL 请求可将路由加入到 BSD socket 接口或从中删除。这些是通过协议来实现的。INET 协议层只允许 处理有超级用户权限的 IP 路由的添加与删除。这些路由可以是固定的，也可以随时间而动态改变。大多数系统使用固定路由。路由器运行路由协议，路由协议持续地检查所有已知目的 IP 的可得到的路由。没有路由器的系统是端系统。路由协议是作为一个守护进程来实现的，如 GATED，它们也用 IOCTL 来向 BSD socket 接口添加和删除路由。

10.7.1 路由缓存

无论什么时候查找 IP 路由，首先都要在路由缓存中检查是否有匹配的路由。如果路由缓存里没有匹配的路由，则要从 Forwarding Information Database 中查找路由。如果那里也没有找到路由，则 IP 包发送失败并通知应用程序。如果在路由缓存中没有而在 Forwarding Information Database 中找到路由，则会为些路由生成一个新项，并添加到路由缓存中。路由缓存是一个表(ip_rt_hash_table)，它包括指向 rtable 数据结构链的指针。hash 函数利用 IP 地址中最小最重要的两个字节来从路由表中进行索引。这两个字节是在目的与提供的最佳 hash 值间是不同的。第个 rtable 项包含路由信息，目的 IP 地址，用于到达那个 IP 地址的网络设备，信息大小的最大值等等。它还有一个 reference count，一个 usage count 和一个最近一次被用的时间信息（在 jiffies 里）。reference count 在每次路由后增加，用于显示该次路由的网络连接数。它在应用程序停止使用路由时减小。useage count 在每次查找路由时增加，用于将 rtable 项在它的 hash 链中排序。路由缓存中的对于所有项的最后被用时间信息将被周期性地检查，以确定是否 rtable 已经旧了。如果某一路由最近没有被使用，则从路由缓存中将之丢弃。由于路由缓存中的路由在有序的，所以常用的路由会排在 hash 链的前面。这意味着能更快地找到这些路由。

10.7.2 The Forwarding Information Database

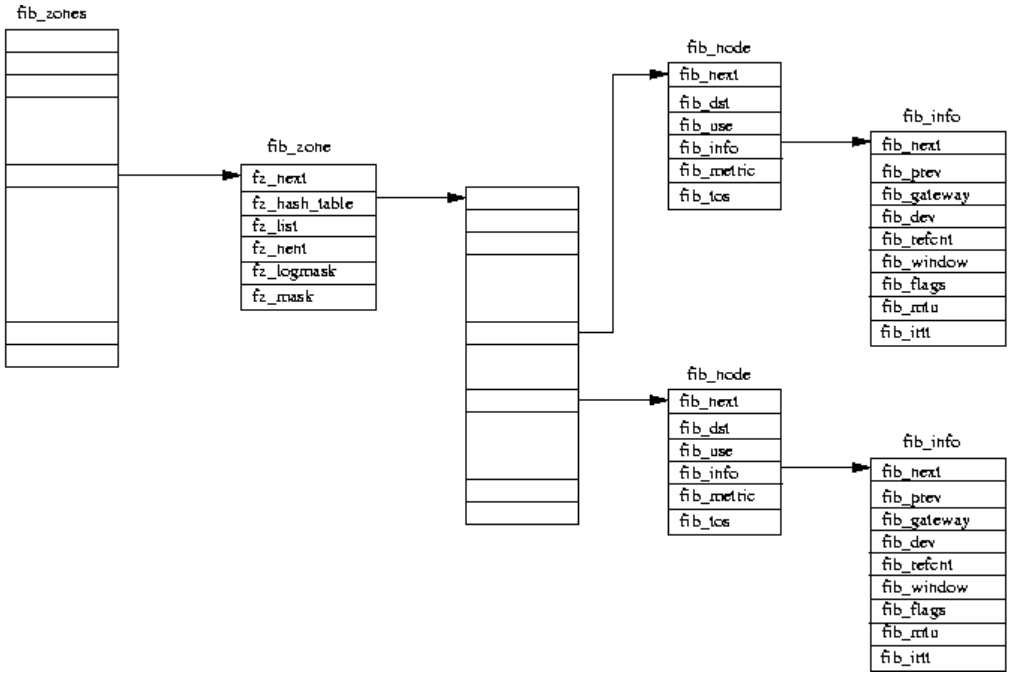


图 10.5: The Forwarding Information Database

Forwarding Information Database（如图 10.5 所示）包含对当前系统当前时间可得到的 IP 路由。它是一个 很复杂的数据结构，尽管进行了合理有效的安排，它仍然不是一个快速的数据库。特别是要在这个数据库中为每一要传送的 IP 包查找目的地时将会非常慢。

这就是要用路由缓存的原因：可以用已知的好的路由来加速 IP 包的传送。路由缓存中的路由来源于 Forwarding Information Database。

每个 IP 子网用一个 fib_zone 结构来描述。 fib_zone hash 表指向着这些结构。hash 索引来源于 IP 子网掩码。所有通向同一子网的路由由 fib_node 和 fib_info 结构来描述，这两结构在每个 fib_zone 结构的 fz_list 中排队。如果这个子网中的路由数增大，则生成一个 hash 表，以使查找 fib_node 结构更加容易。

通向同一子网可以有多个路由，这些路由可能通过多个网关中的一个。IP 路由层不允许用同一个网关对一个子网有多于一个的路由。换言之，如果通向同一子网有多个路由，则每个路由要保证使用一个不同的网关。与每个路由相关的有一个 metric 结构。它用来测量该路由有多优。一个路由的 metric 实质上是它在到达目的子网前所经过的 IP 子网数。metric 越大，路由越差。

脚注:

1 National Science Foundation

2 Synchronous Read Only Memory

3 duh? What used for?

第十一章 核心机制

本章主要描叙 Linux 核心为使核心其他部分能有效工作而提供的几个常用任务与机制。

11.1 底层部分处理机制

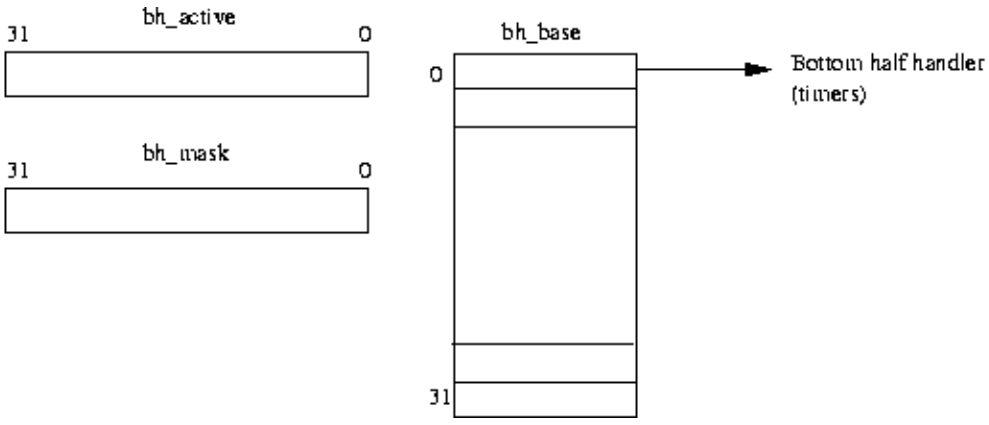


图 11.1 底层部分处理机制数据结构

某些特殊时刻我们并不愿意在核心中执行一些操作。例如中断处理过程中。当中断发生时处理器将停止当前的工作, 操作系统将中断发送到相应的设备驱动上去。由于此时系统中其他程序都不能运行, 所以设备驱动中的中断处理过程不宜过长。有些任务最好稍后执行。Linux 底层部分处理机制可以让设备驱动和 Linux 核心其他部分将这些工作进行排序以延迟执行。图 11.1 给出了一个与底层部分处理相关的核心数据结构。

系统中最多可以有 32 个不同的底层处理过程；bh_base 是指向这些过程入口的指针数组。而 bh_active 和 bh_mask 用来表示那些处理过程已经安装以及那些处于活动状态。如果 bh_mask 的第 N 位置位则表示 bh_base 的 第 N 个元素包含底层部分处理例程。如果 bh_active 的第 N 位置位则表示第 N 个底层处理过程例程可在调度器认 为合适的时刻调用。这些索引被定义成静态的；定时器底层部分处理例程具有最高优先级（索引值为 0）， 控制台底层部分处理例程其次（索引值为 1）。典型的底层部分处理例程包含与之相连的任务链表。例如 immediate 底层部分处理例程通过那些需要被立刻执行的任务的立即任务队列（tq_immediate）来执行。

有些核心底层部分处理过程是设备相关的，但有些更加具有通用性：

TIMER
每次系统的周期性时钟中断发生时此过程被标记为活动，它被用来驱动核心的定时器队列机制。
CONSOLE
此过程被用来处理进程控制台消息。
TQUEUE
此过程被用来处理进程 tty 消息。
NET
此过程被用来做通用网络处理。
IMMEDIATE
这是被几个设备驱动用来将任务排队成稍后执行的通用过程。

当设备驱动或者核心中其他部分需要调度某些工作延迟完成时，它们将把这些任务加入到相应的系统队列中去，如定时器队列，然后对核心发出信号通知它需要调用某个底层处理过程。具体方式是设置 bh_active 中的某些位。如果设备驱动将某个任务加入到了 immediate 队列并希望底层处理过程运行和处理它,可将第 8 位置 1。每次系统调用结束返回调用进程前都要检查 bh_active。如果有位被置 1 则调用处于活动状态的底层处理过程。检查的顺序是从 0 位开始直到第 31 位。

每次调用底层处理过程时 bh_active 中的对应位将被清除。bh_active 是一个瞬态变量，它仅仅在调用调度管理器时有意义；同时它还可以在空闲状态时避免对底层处理过程的调用。

11.2 任务队列

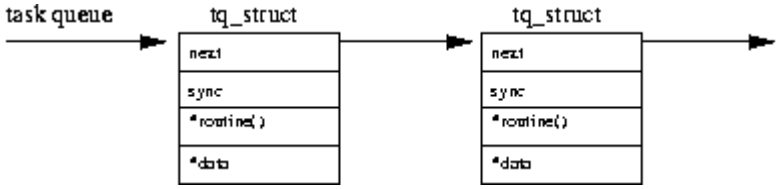


图 11.2: 一个任务队列

任务队列是核心延迟任务启动的主要手段。 Linux 提供了对队列上任务排队以及处理它们的通用机制。

任务队列通常和底层处理过程一起使用；底层的定时器队列处理过程运行时对定时器队列进行处理。任务队列的结构很简单，如图 11.2 所示，它由一个 tq_struct 结构链表构成，每个节点中包含处理过程的地址指针以及指向数据的指针。

处理任务队列上元素时将用到这些过程，同时此过程还将用到指向这些数据的指针。

核心的所有部分，如设备驱动, 都可以创建与使用任务队列。但是核心自己创建与管理的任务队列只有以下三个:

timer

此队列用来对下一个时钟滴答时要求尽快运行的任务进行排队。每个时钟滴答时都要检查此队列看是否为空，如果不为空则定时器底层处理过程将激活此任务。当调度管理器下次运行时定时器队列底层处理过程将和其他底层处理过程一道对任务队列进行处理。这个队列不能和系统定时器相混淆。

immediate

immediate 底层处理过程的优先级低于定时器底层处理过程，所以此类型任务将延迟运行。

scheduler

此任务队列直接由调度管理器来处理。它被用来支撑系统中其他任务队列，此时可以运行的任务是一个处理任务队列的过程，如设备驱动。

当处理任务队列时，处于队列头部的元素将从队列中删除同时以空指针代替它。这个删除操作是一个不可中断的原子操作。队列中每个元素的处理过程将被依次调用。这个队列中的元素通常使用静态分配数据。 然而并没有一个固有机制来丢弃已分配内存。任务队列处理例程简单的指向链表中下一个元素。这个任务才真正清除任何已分配的核心内存。

11.3 定时器（TIMER）

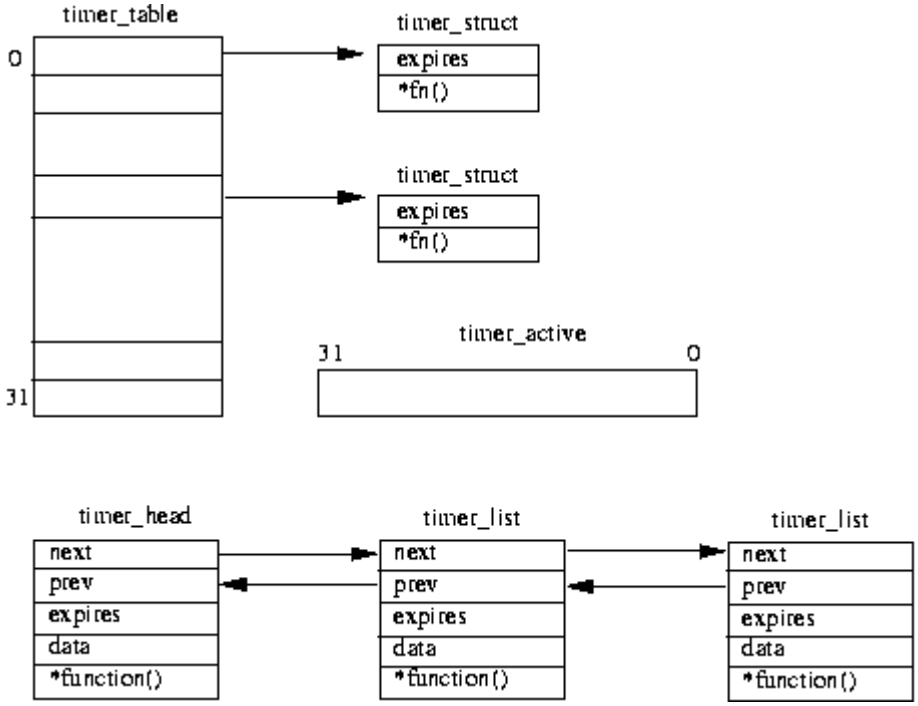


图 11.3 系统定时器

操作系统应该能够在将来某个时刻准时调度某个任务。所以需要一种能保证任务较准时调度运行的机制。希望支持每种操作系统的微处理器必须包含一个可周期性中断它的可编程间隔定时器。这个周期性中断被称为系统时钟滴答，它象节拍器一样来组织系统任务。

Linux 的时钟观念很简单：它表示系统启动后的以时钟滴答记数的时间。所有的系统时钟都基于这种量度，在 系统中的名称和一个全局变量相同-jiffies。

Linux 包含两种类型的系统定时器，它们都可以在某个系统时间上被队列例程使用，但是它们的实现稍有区别。 图 11.3 画出了这两种机制。

第一个是老的定时器机制，它包含指向 timer_struct 结构的 32 位指针的静态数组以及当前活动定时器的屏蔽码 ：time_active。

此定时器表中的位置是静态定义的（类似底层部分处理表 bh_base）。其入口在系统初始化时被加入到表中。 第二种是相对较新的定时器，它使用一个到期时间以升序排列的 timer_list 结构链表。

这两种方法都使用 jiffies 作为终结时间，这样希望运行 5 秒的定时器将不得不将 5 秒时间转换成 jiffies 的单位并且将它和以 jiffies 记数的当前系统时间相加从而得到定时器的终结时间。在每个系统时钟滴答时,定时器的底层部分处理过程被标记成活动状态以便调度管理器下次运行时能进行定时器队列的处理。定时器底层 部分处理过程包含两种类型的系统定时器。老的系统定时器将检查 timer_active 位是否置位。

如果活动定时器已经到期则其定时器例程将被调用同时它的活动位也被清除。新定时器位于 timer_list 结构链表中的入口也将受到检查。每个过期定时器将从链表中清除,同时

它的例程将被调用。新定时器机制的优点之一是能传递一个参数给定时器例程。

11.4 等待队列

进程经常要等待某个系统资源。例如某个进程可能需要描叙文件系统中某个目录的 VFS inode 但是此 inode 可 能不在 buffer cache 中。此时这个进程必须等到该 inode 从包含此文件系统的物理介质中取出来才可以继续 运行。

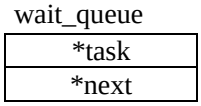


图 11.4 等待队列

Linux 核心使用一个非常简单的队列：等待队列（见图 11.4）。它包含一个指向进程 task_struct 结构的指针以及等待队列中下一个元素的指针。加入到等待队列中的进程既可以是可中断也可以是不可中断的。可中断 进程能够被如定时器到期或者信号等时间中断。此等待进程的状态必须说明成是 INTERRUPTIBLE 还是 UNINTERRUPTIBLE。由于进程现在不能继续运行则调度管理器将接过系统控制权并选择一个新进程运行而等待进程将被挂起。处理等待进程时，每个处于等待队列中的进程都被置为 RUNNING 状态。如果此进程已经从运行队列中删除则它将被重新放入运行队列。下次调度管理器运行时，由于这些进程不再等待所以它们都将是运行候选者。 等待队列可以用来同步对系统资源的访问，同时它们还被 Linux 用于信号灯的实现中。

11.5 Buzz 锁

它使用更频繁的名字叫自旋锁。这是一种保护数据结构或代码片段的原始方式。在某个时刻只允许一个进程访问临界区内的代码。Linux 还同时将一个整数域作为锁来限制对数据结构中某些域的存取。每个希望进入 此区域的进程都试图将此锁的初始值从 0 改成 1。如果当前值是 1 则进程将再次尝试，此时进程好象在一段紧循环代码中自旋。对包含此锁的内存区域的存取必须是原子性的，即检验值是否为 0 并将其改变成 1 的过程不能被任何进程中断。多数 CPU 结构通过特殊指令提供对此方式的支持，同时我们可以在一个非缓冲主存中实现这个流言锁。

当控制进程离开临界区时它将递减此 Buzz 锁。任何处于自旋状态的进程都可以读取它，它们中最快的那个将递增此值并进入临界区。

11.6 信号灯

信号灯被用来保护临界区中的代码和数据。请记住每次对临界区数据，如描叙某个目录 VFS inode 的访问，是通过代表进程的核心代码来进行的。允许某个进程擅自修改由其他进程使用的临界区数据是非常危险的。防止此问题发生的一种方式是在被存取临界区周围使用 buzz 锁，但这种简单的方式将降低系统性能。Linux 使用信号灯来迫使某个时刻只

有唯一进程访问临界区代码和数据，其他进程都必须等待资源被释放才可使用。这些等待进程将被挂起而系统中其他进程可以继续运行。

一个 Linux semaphore 结构包含了以下信息：

count
此域用来保存希望访问此资源的文件个数。当它为正数时表示资源可用。负数和 0 表示进程必须等待。当它初始值为 1 时表示一次仅允许一个进程来访问此资源。当进程需要此资源时它们必须将此 count 值减 1 并且在使用完后将其加 1。
waking
这是等待此资源的进程个数，同时也是当资源可利用时等待被唤醒的进程个数。
wait queue
当进程等待此资源时，它们被放入此等待队列。
lock
访问 waking 域时使用的 buzz 锁。

假设此信号灯的初始值为 1，第一个使用它的进程看到此记数为正数，然后它将其减去 1 而得到 0。现在此进程 拥有了这些被信号灯保护的段代码和资源。当此进程离开临界区时它将增加此信号灯的记数值，最好的情况 是没有其他进程与之争夺临界区的控制权。Linux 将信号灯设计成能在多数情况下有效工作。

如果此时另外一个进程希望进入此已被别的进程占据的临界区时，它也将此记数减 1。当它看到此记数值为-1 则它知道现在不能进入临界区，必须等待到此进程退出使用临界区为止。在这个过程中 Linux 将让这个等待 进程睡眠。等待进程将其自身添加到信号灯的等待队列中然后系统在一个循环中检验 waking 域的值并当 waking 非 0 时调用调度管理器。

临界区的所有者将信号灯记数值加 1，但是如果此值仍然小于等于 0 则表示还有等待此资源的进程在睡眠。在 理想情况下此信号灯的记数将返回到初始值 1 而无需做其他工作。所有者进程将递增 waking 记数并唤醒在此 信号灯等待队列上睡眠的进程。当等待进程醒来时，它发现 waking 记数值已为 1，那么它知道现在可以进入临界区了。然后它将递减 waking 记数，将其变成 0 并继续。所有对信号灯 waking 域的访问都将受到使用信号灯 的 buzz 锁的保护。

第十二章 模块

本章主要描叙 Linux 核心动态加载功能模块（如文件系统）的工作原理。

Linux 核心是一种 monolithic 类型的内核，即单一的大程序，核心中所有的功能部件都可以对其全部内部数据结构和例程进行访问。核心的另外一种形式是微内核结构，此时核心的所有功能部件都被拆成独立部分， 这些部分之间通过严格的通讯机制进行联系。这样通过配置进程将新部件加入核心的方式非常耗时。比如说我们想为一个 NCR 810 SCSI 卡配置 SCSI 驱动，但是核心中没有这个部分。那么我们必须重新配置并重构核心。 Linux 可以让我们可以随意动态的加载与卸载操作系统部件。Linux 模块就是这样一种可在系统启动后的任何时候动态连入核心的代码块。当我们不再需要它时又可以将它从核心中卸载

并删除。Linux 模块多指设备驱动、伪设备驱动, 如网络设备和文件系统。

Linux 为我们提供了两个命令：使用 insmod 来显式加载核心模块，使用 rmmod 来卸载模块。同时核心自身也可以请求核心后台进程 kerneld 来加载与卸载模块。

动态可加载代码的好处在于可以让核心保持很小的尺寸同时非常灵活。在我的 Intel 系统中由于使用了模块，整个核心仅为 406K 字节长。由于我只是偶尔使用 VFAT 文件系统, 所以我将 Linux 核心构造成当 mount VFAT 分区时自动加载 VFAT 文件系统模块。当我卸载 VFAT 分区时系统将检测到我不再需要 VFAT 文件系统模块，将把它从系统中卸载。模块同时还可以让我们无需重构核心并频繁重新启动来尝试运行新核心代码。尽管使用模块很自由，但是也有可能同时带来与核心模块相关的性能与内存损失。可加载模块的代码一般有些长并且额外的数据结构可能会占据一些内存。同时对核心资源的间接使用可能带来一些效率问题。

一旦 Linux 模块被加载则它和普通核心代码一样都是核心的一部分。它们具有与其他核心代码相同的权限与职 责；换句话说 Linux 核心模块可以象所有核心代码和设备驱动一样使核心崩溃。

模块为了使用所需核心资源所以必须能够找到它们。例如模块需要调用核心内存分配例程 kmalloc()来分配 内存。模块在构造时并不知道 kmalloc()在内存中何处，这样核心必须在使用这些模块前修改模块中对 kmalloc()的引用地址。核心在其核心符号表中维护着一个核心资源链表这样当加载模块时它能够解析出模块 中对核心资源的引用。Linux 还允许存在模块堆栈，它在模块之间相互调用时使用。例如 VFAT 文件系统模块 可能需要 FAT 文件系统模块的服务，因为 VFAT 文件系统多少是从 FAT 文件系统中扩展而来。某个模块对其他模 块的服务或资源的需求类似于模块对核心本身资源或服务的请求。不过此时所请求的服务是来自另外一个事先 已加载的模块。每当加载模块时，核心将把新近加载模块输出的所有资源和符号添加到核心符号表中。

当试图卸载某个模块时，核心需要知道此模块是否已经没有被使用，同时它需要有种方法来通知此将卸载模块。 模块必须能够在从核心种删除之前释放其分配的所有系统资源，如核心内存或中断。当模块被卸载时，核心将从核心符号表中删除所有与之对应的符号。

可加载模块具有使操作系统崩溃的能力，而编写较差的模块会带来另外一种问题。当你在一个或早或迟构造的核心而不是当前你运行的核心上加载模块时将会出现什么结果？一种可能的情况是模块将调用具有错误参数的核心例程。核心应该使用严格的版本控制来对加载模块进行检查以防止这种这些情况的发生。

12.1 模块的加载

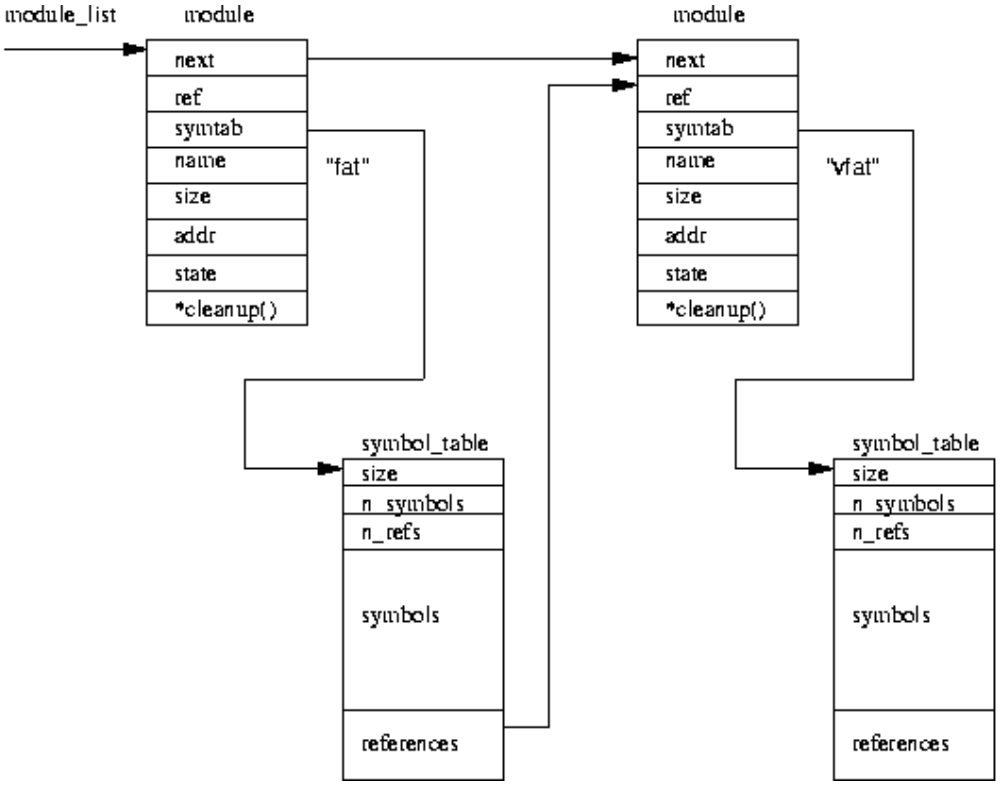


图 12.1 核心模块链表 s

核心模块的加载方式有两种。首先一种是使用 insmod 命令手工加载模块。另外一种则是在需要时加载模块；我们称它为请求加载。当核心发现有必要加载某个模块时，如用户安装了核心中不存在的文件系统时，核心将请求核心后台进程（kerneld）准备加载适当的模块。这个核心后台进程仅仅是一个带有超级用户权限的普通用户进程。当系统启动时它也被启动并为核心打开了一个进程间通讯（IPC）通道。核心需要执行各种任务时用它来向 kerneld 发送消息。

kerneld 的主要功能是加载和卸载核心模块，但是它还可以执行其他任务，如通过串行线路建立 PPP 连接并在适当时候关闭它。kerneld 自身并不执行这些任务，它通过某些程序如 insmod 来做此工作。它只是核心的代理，为核心进行调度。

insmod 程序必须找到要求加载的核心模块。请求加载核心模块一般被保存在 /lib/modules/kernel-version 中。这些核心模块和系统中其他程序一样是已连接的目标文件，但是它们被连接成可重定位映象。即映象没有被连接到在特定地址上运行。这些核心模块可以是 a.out 或 ELF 文件格式。insmod 将执行一个特权级系统调用来找到核心的输出符号。这些都以符号名以及数值形式，如地址值成对保存。核心输出符号表被保存在核心维护的模块链表的第一个 module 结构中，同时 module_list 指针指向此结构。只有特殊符号被添加到此表中，它们在核心编译与连接时确定，不是核心每个符号都被输出到其模块中。例如设备驱动为了控制某个特定系统中断而由核心例程调用的"request_irq"符号。在我的系统

中，其值为 0x0010cd30。我们可以通过使用 ksyms 工具或者查看/proc/ksyms 来观看当前核心输出符号。ksyms 工具既可以显示所有核心输出符号也可以只显示那些已加载模块的符号。insmod 将模块读入虚拟内存并通过使用来自核心输出符号来修改其未解析的核心例程和资源的引用地址。这些修改工作采取由 insmod 程序直接将符号的地址写入模块中相应地址来修改内存中的模块映象。

当 insmod 修改完模块对核心输出符号的引用后，它将再次使用特权级系统调用来申请足够的空间来容纳新核 心。核心将为其分配一个新的 module 结构以及足够的核心内存来保存新模块，并将它放到核心模块链表的尾部。 然后将其新模块标志为 UNINITIALIZED。

图 12.1 给出了一个加载两个模块：VFAT 和 FAT 后的核心链表示意图。不过图中没有画出链表中的第一个模块： 用来存放核心输出符号表的一个伪模块。lsmod 可以帮助我们列出系统中所有已加载的核心模块以及相互间 依赖关系。它是通过重新格式化从核心 module 结构中建立的/proc/modules 来进行这项工作的。核心为其分配的内存被映射到 insmod 的地址空间，这样它就能访问核心空间。insmod 将模块拷贝到已分配空间中，如果为它分配的核心内存已用完，则它将再次申请。不过不要指望多次将加载模块到相同地址，更不用说在两个不同 Linux 系统的相同位置。另外此重定位工作包括使用适当地址来修改模块映象。

这个新模块也希望将其符号输出到核心中，insmod 将为其构造输出符号映象表。每个核心模块必须包含模块 初始化和模块清除例程，它们的符号被设计成故意不输出，但是 insmod 必须知道这些地址，这样它可以将它们传递给核心。所有这些工作做完之后，insmod 将调用初始化代码并执行一个特权级系统调用将模块的初始化与清除例程地址传递给核心。

当将一个新模块加载到核心中间时，核心必须更新其符号表并修改那些被新模块使用的老模块。那些依赖于其他模块的模块必须维护在其符号表尾部维护一个引用链表并在其 module 数据结构中指向它。图 12.1 中 VFAT 依赖于 FAT 文件系统模块。所以 FAT 模块包含一个对 VFAT 模块的引用；这个引用在加载 VFAT 模块时添加。核心调用模块的初始化例程，如果成功它将安装此模块。模块的清除例程地址被存储在其 module 结构中，它将在 模块卸载时由核心调用。最后模块的状态被设置成 RUNNING。

12.2 模块的卸载

模块可以通过使用 rmmod 命令来删除，但是请求加载模块将被 kerneld 在其使用记数为 0 时自动从系统中删除。 kerneld 在其每次 idle 定时器到期时都执行一个系统调用以将系统中所有不再使用的请求加载模块从系统中 删除。这个定时器的值在启动 kerneld 时设置；我系统上的值为 180 秒。这样如果你安装一个 iso9660 CDROM 并且你的 iso9660 文件系统是一个可加载模块，则在卸载 CD ROM 后的很短时间内此 iso9660 模块将从核心中删除。

如果核心中的其他部分还在使用某个模块，则此模块不能被卸载。例如如果你的系统中安装了多个 VFAT 文件系统则你将不能卸载 VFAT 模块。执行 lsmod 我们将看到每个模块的引用记数。如：

```
Module:         #pages:   Used by:
msdos             5              1
vfat              4              1 (autoclean)
```

```
fat                     6       [vfat msdos]       2 (autoclean)
```

此记数表示依赖此模块的核心实体个数。在上例中 VFAT 和 msdos 模块都依赖于 fat 模块, 所以 fat 模块的引用记数为 2。vfat 和 msdos 模块的引用记数都为 1，表示各有一个已安装文件系统。如果我们安装另一个 VFAT 文件系统则 vfat 模块的引用记数将为 2。模块的引用记数被保存在其映象的第一个长字中。这个字同时还包含 AUTOCLEAN 和 VISITED 标志。请求加载模块使用这两个标志域。如果模块被标记成 AUTOCLEAN 则核心知道此模 块可以自动卸载。VISITED 标志表示此模块正被一个或多个文件系统部分使用；只要有其他部分使用此模块则这个标志被置位。每次系统被 kerneld 要求将没有谁使用的请求模块删除时，核心将在所有模块中扫描可能的候选者。但是一般只查看那些被标志成 AUTOCLEAN 并处于 RUNNING 状态的模块。如果某模块的 VISITED 标记被清除则它将被删除出去。如果某模块可以卸载，则可以调用其清除例程来释放掉分配给它的核心资源。它所对应的 module 结构将被标记成 DELETED 并从核心模块链表中断开。其他依赖于它的模块将修改它们各自的引用域来表示它们间的依赖关系不复存在。此模块需要的核心内存都将被回收。

第十三章 处理器

Linux 可以运行在许多类型的处理器上，本章将给出对它们的简单描叙。

13.1 X86

省略

13.2 ARM

ARM 处理器是一种低功耗高性能的 32 位 RISC 处理器。它在嵌入式设备如移动电话和 PDA 中广泛使用。共有 31 个 32 位寄存器而其中 16 个可以在任何模式下看到。它的指令为简单的加载与存储指令（从内存中加载某个值， 执行完操作后再将其放回内存）。ARM 一个有趣的特点是它所有的指令都带有条件。例如你可以测试某个 寄存器的值但是直到下次你使用同一条件时进行测试时，你才能有条件的执行这些指令。另一个特征是可以 在加载数值的同时进行算术和移位操作。它可以在几种模式下操作，包括通过使用 SWI（软件中断）指令从 用户模式进入的系统模式。

ARM 处理器是一个综合体，ARM 公司自身并不制造微处理器。它们是有 ARM 的合作伙伴（Intel 或 LSI）制造。 ARM 还允许将其他处理器通过协处理器接口进行紧耦合。它还包括几种内存管理单元的变种，包括简单的 内存保护到复杂的页面层次。

13.3 Alpha AXP 处理器

Alpha AXP 是一种 64 位的 load/store 类型的 RISC 处理器，其设计目标就是高速度。它所有的寄存器都是 64 位； 还拥有 32 个整数寄存器和 32 个浮点数寄存器。第 31 个整数与浮点数寄存器被用来进行空操作。对它们读将 得到 0，对它们的写没有什么影响。所有的指令都是 32 位并且内存操作不是写就是读。这种结构允许不同的 实现。

不能对内存中数值的操作，所有的数据操作都是在寄存器中完成。所以如果你试图递增一个内存中的计数器 则必须先读入寄存器，修改后再写回。指令之间的相互操作仅仅通过其中一个对寄存器和内存位置的写入而 另一个从寄存器或内存位置读出而进行。Alpha AXP 处理器的一个有趣的特征是包含可产生标志位的指令。 如测试两个寄存器中的值是否相等，其结果没有存放在处理器状态寄存器中而是放在第 3 个寄存器里。初 看起来好象很奇怪，但是删除对状态寄存器的依赖关系将更加容易构造一个超标量多发射 CPU 体系结构。在 不相关寄存器中的指令将不必为从单一状态寄存器等待而浪费执行时间。缺少对内存的直接操作以及大量 寄存器对多发射结构也有帮助。

Alpha AXP 结构使用叫做特权体系库代码（PALcode）的一组子程序。此 PALcode 依赖于特定的操作系统、 Alpha AXP 体系的 CPU 实现以及系统硬件。这些子程序为操作系统提供了上下文切换、中断、异常和内存 管理原语。它们可以由硬件或者通过 CALL_PAL 指令来调用。PALcode 使用标准的 Alpha AXP 汇编代码写成并 做了一些扩展以提供对底层硬件指令的直接访问，如内部处理器寄存器。PALcode 在一种叫 PALmode 的特权 模式下执行，此时它将停止一些系统事件的发生并允许 PALcode 对物理系统硬件进行完全的控制。

第十四章 Linux 核心资源

本章主要描叙寻找某个特殊核心函数时用到的 Linux 核心资源。

本书并不要求读者具有 C 编程语言的能力或者拥有 Linux 核心源代码来理解 Linux 核心工作原理。但是如果 对核心源代码进行阅读将加深对 Linux 操作系统的理解。本章提供了一个核心源代码的综述。

从哪里得到 Linux 核心源码

所有主要 Linux 分发版本（如 Craftworks，Debian，Slackware，Redhat）都包含了源码在内。通常安装在 你的 Linux 系统核心就是从这些源码中构造出来的。由于一些显然的因素，这些源码都或多或少有点过期。 你可以在 www-appendix 一章中的那些 WEB 站点中得到最新的版本。这些站点包括 ftp://ftp.cs.helsinki.fi 以及所有其他镜象站点中。helsinki 的这个 WEB 站点上的 Linux 源码显然是最新的但是 MIT 和 Sunsite 中的也 不会差太远。如果你无法访问这些 WEB 站点，有许多 CD ROM 厂商以非常合理的价格提供了这些 WEB 站点的镜象光盘。有些 厂商还提供每季度甚至每个月更新的订购服务。另外你所在的本地 Linux 用户组也是一个很好的资源。

Linux 核心代码的版本编号很简单。任何偶数编号的核心（如 2.0.30）都是稳定的发行版而记数编号的核心 （如 2.1.42）都是正在开发的核心。本书基于稳定的 2.0.30 版本。开发版的核心具有所有最新的特征并支持 最新的设备。尽管它们不是你所希望的那样稳定，但是对于 Linux 用户团体来说试用新核心是非常重要的。 因为他们将完成这些评测工作。当试用非发行版本核心时备份系统总是有好处的。

核心的修改以 patch 文件来分发。而 patch 实用程序被用来对一些核心源码进行编辑。例如如果现在你已经 有了 2.0.39 的核心代码但是你想升级到 2.0.30，那么你在取得 2.0.30 补丁文件后可以实用以下命令来修改 现存核心：

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

一个收集核心补丁的站点是 http://www.linuxhq.com。

核心源码的组织

核心源码的顶层是/usr/src/linux 目录，在此目录下你可以看到大量子目录：

arch

这个子目录包含了所有体系结构相关的核心代码。它还包含每种支持的体系结构的子目录，如 i386。

include

这个目录包括了用来重构核心的大多数 include 文件。对于每种支持的体系结构分别有一个子目录。 此目录中的 asm 子目录中是对应某种处理器的符号连接，如 include/asm-i386。要修改处理器结构 则只需编辑核心的 makefile 并重新运行 Linux 核心配置程序。

init

此目录包含核心启动代码。

mm

此目录包含了所有的内存管理代码。与具体体系结构相关的内存管理代码位于 arch/*/mm 目录下， 如 arch/i386/mm/fault.c 。

drivers

系统中所有的设备驱动都位于此目录中。它又进一步划分成几类设备驱动，如 block。

ipc

此目录包含了核心的进程间通讯代码。

modules

此目录仅仅包含已建好的模块。

fs

所有的文件系统代码。它也被划分成对应不同文件系统的子目录，如 vfat 和 ext2。

kernel

主要核心代码。同时与处理器结构相关代码都放在 arch/*/kernel 目录下。

net

核心的网络部分代码。

lib

此目录包含了核心的库代码。与处理器结构相关库代码被放在 arch/*/lib/目录下。

scripts
此目录包含用于配置核心的脚本文件（如 awk 和 tk 脚本）。

从哪里入手

阅读象 Linux 核心代码这样的复杂程序令人望而生畏。它象一个越滚越大的雪球。阅读核心某个部分经常要 用到好几个其他的相关文件，不久你将会忘记你原来在干什么。本小节将给出一些提示。

系统启动与初始化

在基于 intel 的系统上，Linux 可以通过 loadlin.exe 或者 LILO 将核心载入内存并将控制传递给它。这部分程序 位于 arch/i386/kerneld/head.S。此文件完成一些处理器相关操作并跳转到 init/main.c 中的 main()例程。

内存管理

这部分代码主要位于 mm 目录中但其处理器结构相关部分被放在 arch/*/mm 中。页面出错处理代码位于 mm 下的 memory.c 文件中而内存映射与页面 cache 代码位于 filemap.c 中。buffer cache 则在 mm/buffer.c 中实现，swap cache 位于 mm/swap_state.c 和 mm/swapfile.c 中。

核心

大多数通用代码位于 kernel 目录下而处理器相关代码被放在 arch/*/kernel 中。调度器位于 kernel/sched.c 而 fork 代码位于 kernel/fork.c 中。底层部分处理代码位于 include/linux/interrupt.h 中。task_struct 的 描叙则在/linux/sched.h 中可以找到。

PCI

PCI 伪设备驱动位于 drivers/pci/pci.c 且其系统通用定义放在 include/linux/pci.h 中。每个处理器结构 具有特殊的 PCI BIOS 代码，Alpha AXP 的位于 arch/alpha/kernel/bios32.c 中。

进程间通讯

所有这些代码都在 ipc 目录中。系统 V IPC 对象都包含一个 ipc_perm 结构，它在 include/linux/ipc.h 中描叙。 系统 V 消息在 ipc/msg.c 中实现，共享内存在 ipc/shm.c 而信号灯位于 ipc/sem.c 中。管道在 ipc/pipe.c 中实现。

中断处理

核心的中断处理代码总是与微处理器结构相关。Intel 系统的中断处理代码位于 arch/i386/kernel/irq.c 中， 其定义位于 include/asm-i386/irq.h 中。

设备驱动

Linux 核心源码的大多数都是设备驱动。所有 Linux 的设备驱动源码都放在 drivers 目录中并分成以下几类：

/block

块设备驱动包括 IDE（在 ide.c 中）驱动。如果你想寻找这些可包含文件系统的设备的初始化过程 则应该在 drivers/block/genhd.c 中的 device_setup()。当安装一个 nfs 文件系统时

不但要初始化 硬盘还需初始化网络。块设备包括 IDE 与 SCSI 设备。

/char

此目录包含字符设备的驱动，如 ttys，串行口以及鼠标。

/cdrom

包含所有 Linux CDROM 代码。在这里可以找到某些特殊的 CDROM 设备（如 Soundblaster CDROM）。 IDE 接口的 CD 驱动位于 drivers/block/ide-cd.c 中而 SCSI CD 驱动位于 drivers/scsi/scsi.c 中。

/pci

它包含了 PCI 伪设备驱动源码。这里可以找到关于 PCI 子系统映射与初始化的代码。另外位于 arch/alpha/kernel/bios32.c 中的 Alpha AXP PCI 补丁代码也值得一读。

/scsi

这里可以找到所有的 SCSI 代码以及 Linux 支持的 SCSI 设备的设备驱动。

/net

包含网络驱动源码，如 tulip.c 中的 DECChip 21040 PCI 以太网驱动。

/sound

所有的声卡驱动源码。

文件系统

EXT2 文件系统的源码位于 fs/ext2 中，其数据结构定义位于 include/linux/ext2_fs.h, ext2_fs_i.h 以及 ext2_fs_sb.h 中。虚拟文件系统数据结构在 include/linux/fs.h 中描叙且其代码在 fs/*中。buffer cache 和 update 核心后台进程在 fs/buffer.c 中实现。

网络

网络代码位于 net 目录而大多数包含文件位于 include/net 中。BSD 套接口代码位于 net/socket.c 中。IPV4 的 INET 套接口代码位于 net/ipv4/af_inet.c 中。通用协议支撑代码（包括 sk_buff 处理过程）位于 net/core 中， 同时 TCP/IP 网络代码位于 net/ipv4 中。网络设备驱动位于 drivers/net 中。

模块

核心模块代码部分位于核心中部分位于 modules 包中。核心代码位于 kernel/modules.c 且其数据结构与核心 后台进程 kerneld 消息位于 include/linux/module.h 和 include/linux/kerneld.h 目录中。同时必要时需查阅 include/linux/elf.h 中的 ELF 文件格式。

第十五章 Linux 核心数据结构

本章列出了 Linux 实用的主要数据结构。

15.1 block_dev_struct

此结构用于向核心登记块设备，它还被 buffer cache 实用。所有此类结构都位于 blk_dev 数组中。

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

15.2 buffer_head

此结构包含关于 buffer cache 中一块缓存的信息。

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

struct buffer_head {
   /* First cache line: */
   unsigned long        b_blocknr;      /* block number                 */
   kdev_t               b_dev;          /* device (B_FREE = free)       */
   kdev_t               b_rdev;         /* Real device                  */
   unsigned long        b_rsector;      /* Real buffer location on disk */
```

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

15.3 device

系统中每个网络设备都用一个设备数据结构来表示。

```
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);
};
```

15.4 device_struct

此结构被块设备和字符设备用来向核心登记（包含设备名称以及可对此设备进行的文件操作）。chrdevs 和 blkdevs 中的每个有效分别表示一个字符设备和块设备。

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

15.5 file

每个打开的文件、套接口都用此结构表示。

```
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 */
};
```

15.6 files_struct

描叙被某进程打开的所有文件。

```
struct files_struct {
    int             count;
```

```
    fd_set          close_on_exec;
    fd_set          open_fds;
    struct file     * fd[NR_OPEN];
};
```

15.7 fs_struct

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

15.8 gendisk

包含关于某个硬盘的信息。用于磁盘初始化与分区检查时。

```
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;
};
```

15.9 inode

此 VFS inode 结构描叙磁盘上一个文件或目录的信息。

```
struct inode {
        kdev_t                  i_dev;
        unsigned long           i_ino;
        umode_t                 i_mode;
        nlink_t                 i_nlink;
        uid_t                   i_uid;
        gid_t                   i_gid;
        kdev_t                  i_rdev;
        off_t                   i_size;
        time_t                  i_atime;
        time_t                  i_mtime;
        time_t                  i_ctime;
        unsigned long           i_blksize;
        unsigned long           i_blocks;
        unsigned long           i_version;
        unsigned long           i_nrpages;
        struct semaphore        i_sem;
        struct inode_operations *i_op;
        struct super_block      *i_sb;
        struct wait_queue       *i_wait;
        struct file_lock        *i_flock;
        struct vm_area_struct   *i_mmap;
        struct page             *i_pages;
        struct dquot            *i_dquot[MAXQUOTAS];
        struct inode            *i_next, *i_prev;
        struct inode            *i_hash_next, *i_hash_prev;
        struct inode            *i_bound_to, *i_bound_by;
        struct inode            *i_mount;
        unsigned short          i_count;
        unsigned short          i_flags;
        unsigned char           i_lock;
        unsigned char           i_dirt;
        unsigned char           i_pipe;
        unsigned char           i_sock;
        unsigned char           i_seek;
        unsigned char           i_update;
        unsigned short          i_writecount;
        union {
                struct pipe_inode_info  pipe_i;
                struct minix_inode_info minix_i;
```

```
        struct ext_inode_info       ext_i;
        struct ext2_inode_info      ext2_i;
        struct hpfs_inode_info      hpfs_i;
        struct msdos_inode_info     msdos_i;
        struct umsdos_inode_info umsdos_i;
        struct iso_inode_info       isofs_i;
        struct nfs_inode_info       nfs_i;
        struct xiafs_inode_info     xiafs_i;
        struct sysv_inode_info      sysv_i;
        struct affs_inode_info      affs_i;
        struct ufs_inode_info       ufs_i;
        struct socket                 socket_i;
        void                            *generic_ip;
    } u;
};
```

15.10 ipc_perm

此结构描叙对一个系统 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 */
};
```

15.11 irqaction

用来描叙系统的中断处理过程。

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

```
    void *dev_id;
    struct irqaction *next;
};
```

15.12 linux_binfmt

用来表示可被 Linux 理解的二进制文件格式。

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

15.13 mem_map_t

用来保存每个物理页面的信息。

```
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;
```

15.14 mm_struct

用来描叙某任务或进程的虚拟内存。

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

15.15 pci_bus

表示系统中的一个 PCI 总线。

```
struct pci_bus {
   struct pci_bus    *parent;         /* parent bus this bridge is on */
   struct pci_bus    *children;      /* chain of P2P bridges on this bus */
   struct pci_bus    *next;           /* chain of all PCI buses */
   struct pci_dev    *self;           /* bridge device as seen by parent */
   struct pci_dev    *devices;       /* devices behind this bridge */
   void        *sysdata;                 /* hook for sys-specific extension */
   unsigned char    number;            /* bus number */
   unsigned char    primary;          /* number of primary bridge */
   unsigned char    secondary;       /* number of secondary bridge */
   unsigned char    subordinate;    /* max number of subordinate buses */
};
```

15.16 pci_dev

表示系统中的每个 PCI 设备，包括 PCI-PCI 和 PCI-PCI 桥接器。

```
/*
 * 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 */
};
```

15.17 request

被用来向系统的块设备发送请求。它总是向 buffer cache 读出或写入数据块。

```
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;
};
```

15.18 rtable

用来描叙向某个 IP 主机发送包的路由信息。此结构在 IP 路由 cache 内部实用。

```
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;
};
```

15.19 semaphore

保护临界区数据结构和代码信号灯。

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

15.20 sk_buff

用来描叙在协议层之间交换的网络数据。

```
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                    */
};
```

15.21 sock

包含 BSD 套接口的协议相关信息。例如对于一个 INET（Internet Address Domain）套接口此数据结构 包含 TCP/IP 和 UDP/IP 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);

};
```

15.22 socket

包含 BSD 套接口的信息。它不独立存在，一般位于一个 VFS inode 结构中。

```
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         */
};
```

15.23 task_struct

用来描叙系统中的进程或任务。

```
struct task_struct {
/* these are hardcoded - don't touch */
    volatile long               state;              /* -1 unrunnable, 0 runnable, >0 stopped */
    long                        counter;
    long                        priority;
    unsigned                    long signal;
    unsigned                    long blocked;      /* bitmap of masked signals */
    unsigned                    long flags;         /* per process flags, defined below */
    int errno;
    long                        debugreg[8];        /* Hardware debugging registers */
    struct exec_domain      *exec_domain;
/* various fields */
    struct linux_binfmt     *binfmt;
    struct task_struct      *next_task, *prev_task;
    struct task_struct      *next_run,    *prev_run;
    unsigned long               saved_kernel_stack;
    unsigned long               kernel_stack_page;
    int                         exit_code, exit_signal;
    /* ??? */
    unsigned long               personality;
    int                         dumpable:1;
    int                         did_exec:1;
    int                         pid;
    int                         pgrp;
    int                         tty_old_pgrp;
    int                         session;
    /* boolean value for session group leader */
    int                         leader;
    int                         groups[NGROUPS];
    /*
     * pointers to (original) parent process, youngest child, younger sibling,
     * older sibling, respectively.    (p->father can be replaced with
```

```
     * p->p_pptr->pid)
     */
    struct task_struct      *p_opptr, *p_pptr, *p_cptr,
                                *p_ysptr, *p_osptr;
    struct wait_queue       *wait_chldexit;
    unsigned short          uid,euid,suid,fsuid;
    unsigned short          gid,egid,sgid,fsgid;
    unsigned long           timeout, policy, rt_priority;
    unsigned long           it_real_value, it_prof_value, it_virt_value;
    unsigned long           it_real_incr, it_prof_incr, it_virt_incr;
    struct timer_list       real_timer;
    long                        utime, stime, cutime, cstime, start_time;
/* mm fault and swap info: this can arguably be seen as either
      mm-specific or thread-specific */
    unsigned long           min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
    int swappable:1;
    unsigned long           swap_address;
    unsigned long           old_maj_flt;       /* old value of maj_flt */
    unsigned long           dec_flt;           /* page fault count of the last time */
    unsigned long           swap_cnt;          /* number of pages to swap on next pass */
/* limits */
    struct rlimit           rlim[RLIM_NLIMITS];
    unsigned short          used_math;
    char                        comm[16];
/* file system info */
    int                         link_count;
    struct tty_struct       *tty;                   /* NULL if no tty */
/* ipc stuff */
    struct sem_undo         *semundo;
    struct sem_queue        *semsleeping;
/* ldt for this task - used by Wine.    If NULL, default_ldt is used */
    struct desc_struct *ldt;
/* tss for this task */
    struct thread_struct tss;
/* filesystem information */
    struct fs_struct        *fs;
/* open file information */
    struct files_struct    *files;
/* memory management info */
    struct mm_struct        *mm;
/* signal handlers */
    struct signal_struct *sig;
#ifdef __SMP__
    int                         processor;
```

```
    int                 last_processor;
    int                 lock_depth;     /* Lock depth.
                                           We can context switch in and out
                                           of holding a syscall kernel lock... */
#endif
};
```

15.24 timer_list

用来为进程实现实时时钟。

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

15.25 tq_struct

每个任务队列结构（tq_struct）包含着已经排队的任务信息。它被设备驱动用来描叙那些无需立刻 执行的任务。

```
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 */
};
```

15.26 vm_area_struct

表示某进程的一个虚拟内存区域。

```
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 */
};
```

第十六章 Linux 相关 Web 和 FTP 站点

http://www.azstarnet.com/ [\tilde]axplinux

This is David Mosberger-Tang's Alpha AXP Linux web site and it is the place to go for all of the Alpha AXP HOWTOs. It also has a large number of pointers to Linux and Alpha AXP specific information such as CPU data sheets.

http://www.redhat.com/

Red Hat's web site. This has a lot of useful pointers.

ftp://sunsite.unc.edu

This is the major site for a lot of free software. The Linux specific software is held in pub/Linux.

http://www.intel.com

Intel's web site and a good place to look for Intel chip information.

http://www.ssc.com/lj/index.html

The Linux Journal is a very good Linux magazine and well worth the yearly subscription for its excellent articles.

http://www.blackdown.org/java-linux.html

This is the primary site for information on Java on Linux.

ftp://tsx-11.mit.edu/ [\tilde]ftp/pub/linux

MIT's Linux ftp site.

ftp://ftp.cs.helsinki.fi/pub/Software/Linux/Kernel

Linus's kernel sources.

http://www.linux.org.uk

The UK Linux User Group.

http://sunsite.unc.edu/mdw/linux.html

Home page for the Linux Documentation Project,

http://www.digital.com

Digital Equipment Corporation's main web page.

http://altavista.digital.com

DIGITAL's Altavista search engine. A very good place to search for information within the web and news groups.

http://www.linuxhq.com

The Linux HQ web site holds up to date official and unoffical patches as well as advice and web pointers that help you get the best set of kernel sources possible for your system.

http://www.amd.com

The AMD web site.

http://www.cyrix.com

Cyrix's web site.

http://www.arm.com

ARM's web site.

附录 A 作者简介

David A Rusling 出生于 1957 年英格兰北部。在大学时由于老师开办的操作系统核心讲座对 Unix 发生兴趣。在大学毕业时用最新的 PDP-11 进行项目开发。1982 年以优异成绩从计算机专业毕业后在普利茅斯的 Prime 公司工作，2 年后加入 DEC。 在 DEC 的最初 5 年期间先后参加多个项目，后来调入致力于 Alpha 芯片和 StrongARM 评测板的半导体部门。1998 年来到 ARM，主要工作是编写底层固件程序以及操作系统移植。

David A Rusling
3 Foxglove Close,
Wokingham,
Berkshire RG41 3NF,
United Kingdom

附录 B The GNU General Public License

Printed below is the GNU General Public License (the GPL or copyleft), under which Linux is licensed. It is reproduced here to clear up some of the confusion about Linux's copyright status-Linux is not shareware, and it is not in the public domain. The bulk of the Linux kernel is copyright © 1993 by Linus Torvalds, and other software and parts of the kernel are copyrighted by their authors. Thus, Linux is copyrighted, however, you may redistribute it under the terms of the GPL printed below.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Preamble
The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software-to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software

(and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions for Copying, Distribution, and Modification

[0.] This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The ``Program", below, refers to any such program or work, and a ``work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term ``modification".) Each licensee is addressed as ``you".
Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

[1.] You may copy and distribute verbatim copies of the Program's source code as you receive it,

in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.
You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

[2.] You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

[a.] You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

[b.] You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

[c.] If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)
These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

[3.] You may copy and distribute the Program (or a work based on it, under Section 2) in object

code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

[a.] Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

[b.] Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

[c.] Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)
The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

[4.] You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

[5.] You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

[6.] Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the

Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

[7.] If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.
If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

[8.] If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

[9.] The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.
Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and ``any later version'', you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

[10.] If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

[11.] BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM ``AS IS'' WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

[12.] IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the ``copyright'' line and a pointer to where the full notice is found.

Copyright © 19yy This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19yy name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items-whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a ``copyright disclaimer'' for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker.
, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.