第1章　硬件基础与软件基础

1.1　硬件基础

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

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

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

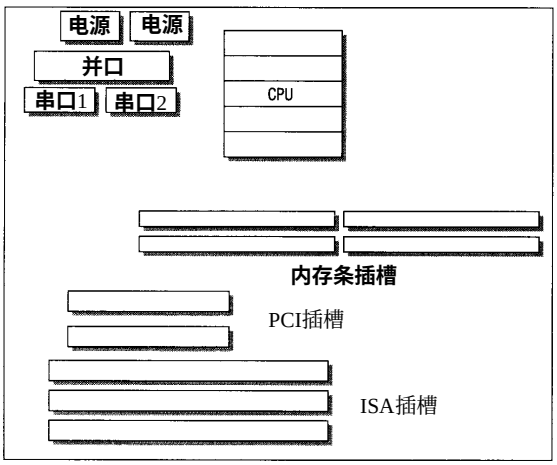

图1-1-1　典型的PC主板

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

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

1.1.1 CPU

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

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

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

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

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

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

理器有以下的专用寄存器：

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

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

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

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

1.1.2 存储器

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

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

1.1.3 总线

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

1.1.4 控制器和外设

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

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

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

1.1.5 地址空间

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

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

1.1.6 时钟

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

1.2 软件基础

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

1.2.1 计算机语言

1. 汇编语言

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

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

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

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

2. C语言和编译器

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

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

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

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

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

C允许你将相关的变量捆在一起成为数据结构。例如：

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

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

3. 连接器

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

1.2.2 什么是操作系统

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

为了开始理解什么是操作系统，考虑一下你敲入一个简单的命令后发生了什么：

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

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

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

1. 存储器管理

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

2. 进程

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

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

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

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

3. 设备驱动程序

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

4. 文件系统

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

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

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

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

1.2.3 内核数据结构

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

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

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

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

1. 链表

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

2. 散列表

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

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

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

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

3. 抽象接口

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

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