第3章 内核体系结构概述

本章从较高层次上对内核进行说明。从顺序上来说，本章首先介绍内核设计目标，接下来介绍内核体系结构，最后介绍内核源程序目录结构。

内核设计目标

Linux 的内核展现出了几个相互关联的设计目标，它们依次是：清晰性（clarity），兼容性（compatibility），可移植性（portability），健壮性（robustness），安全性（security）和速度（speed）。这些目标有时是互补的，有时则是矛盾的。但是它们被尽可能的保持在相互一致的状态，内核设计和实现的特性通常都要回归到这些问题上来。本节接下来的部分将分别讨论这些设计目标，同时还将对它们之间的取舍与平衡进行简要的说明。

清晰性

稍微过于简化的说，内核目标是在保证速度和健壮性的前提下尽量清晰。这和现在的大多数应用程序的开发有所区别，后者的目标通常是在保证清晰性和健壮性的基础上尽量提高速度。因而在内核内部，速度和清晰性经常是一对矛盾。

在某种程度上，清晰性是健壮性的必要补充：一个很容易理解的实现方法比较容易证明是正确的；或者即使不正确，也能比较容易的找出其问题所在。从而这两个目标很少会发生冲突。

但是清晰性和速度通常却是一对矛盾。经过仔细手工优化的算法通常都使用了编译器生成代码的类似技术，很少可能是最清晰的解决方案。当内核中清晰性和速度要求不一致时，通常都是以牺牲清晰性来保证速度的。即便如此，程序员仍然清楚的知道清晰性的重要性，而且他们也做了大量完美的工作以使用最清晰的方法保证速度。

兼容性

正如第 1 章中所述，Linux 最初的编写目的是为了实现一个完整的、与 Unix 兼容的操作系统内核。随着开发过程的展开，它也开始以符合 POSIX 标准为目标。就内核而言，兼容 Unix（至少是同某一现代的 Unix 实现相兼容）和符合 POSIX 标准并没有什么区别，因此我们也不会在这个问题上详细追究。

内核提供了另外一种类型的兼容性。基于 Linux 的系统能够提供可选择的对 Java.class 文件的本地运行支持。（据说 Linux 是第一个提供这种支持的操作系统。）尽管实际负责 Java 程序解释执行的是另外一个 Java 虚拟机进程，该虚拟机并没有内置到内核中。但是内核提供的这种机制可以使得这种支持对用户是透明的。通过内核本身提供的程度不同的支持（这并不代表大部分工作像 Java 的解决方式一样能够通过外部进程实现），对其它可执行文件格式的支持也能够以同样的方式插入内核中。这方面的内容将在第 7 章中详细介绍。

另外需要说明的是，GNU/Linux 系统作为一个整体通过 DOSEMU 仿真机器提供了对 DOS 可执行程序的支持，而且也通过 WINE 设计提供了对 Windows 可执行程序的部分支持。系统还以同样的方式通过 SAMBA 提供了对 Windows 兼容文件和打印服务的支持。但是这些都不是同内核密切相关的问题，因此在本书中我们不再对它们进行讨论。

兼容性的另外一个方面是兼容异种文件系统，本章中稍后会有更为详细的介绍，但是大

部分内容已经超出了本书的范围。Linux 能够支持很多文件系统，例如 ext2（“本地”文件系统），ISO-9660（CD-ROM 使用的文件系统），MS-DOS，网络文件系统（NFS）等许多其它文件系统。如果你有使用其它操作系统格式的磁盘或者一个网络磁盘服务器，那么 Linux 将能够和这些不同的文件系统进行交互。

兼容性的另外一个问题是网络，这在当今 Internet 流行的时代尤为重要。作为 Unix 的一个变种，Linux 自然从很早就开始提供对 TCP/IP 的支持。内核还支持其它许多网络协议，它们包括 AppleTalk 协议的代码，这使得 Linux 单元（box）可以和 Macintosh 机自由通讯；Novell 的网络协议，也就是网络报文交换（IPX），分组报文交换（SPX），和 NetWare 核心协议（NCP）；IP 协议的新版本 IPv6；以及其它一些不太出名的协议。

兼容性考虑的最后一个方面是硬件兼容性。似乎每个不常见的显卡，市场份额小的网卡，非标准的 CD-ROM 接口和专用磁带设备都有 Linux 的驱动程序。（只要它不是专为特定操作系统设计的专用硬件。）而且只要越来越多的厂商也逐渐认识到 Linux 的优势，并能够为更容易地实现向 Linux 上移植而开放相应的源程序代码，Linux 对硬件支持会越来越好。

这些兼容性必须通过一个重要的子目标：模块度（Modularity）来实现。在可能的情况下，内核只定义子系统的抽象接口，这种抽象接口可以通过任何方法来实现。例如，内核对于新文件系统的支持将简化为对虚拟文件系统（VFS）接口的代码实现。第 7 章中介绍的是另外一个例子，内核对二进制句柄的抽象支持是实现对诸如 Java 之类的新可执行格式的支持的方法。增加新的可执行格式的支持将转变为对相应的二进制句柄接口的实现。

可移植性

与硬件兼容性相关的设计目标是可移植性（portability），也就是在不同硬件平台上运行 Linux 的能力。系统最初是为运行在标准 IBM 兼容机上的 Intel x86 CPU 而设计的，当时根本没有考虑到可移植性的问题。但是情况从那以后已经发生了很大的变化。现在正式的内核移植包括向基于 Alpha，ARM，Motorola 69x0，MIPS，PowerPC，SPARC 以及 SPARC-64 CPU 系统的移植。因而，Linux 可以在 Amigas，旧版或新版的 Macintosh，Sun 和 SGI 工作站以及 NeXT 机等机器上运行。而且这些还只是标准内核发行版本的移植范围。从老的 DEC VAX 到 3Com 掌上系列个人数字助理（例如 Palm III）的非正式的移植工作也在不断进行中。成功的非正式移植版本后来通常都会变成正式的移植版本，因此这些非正式的移植版本很多最终都会出现在主开发树中。

广泛平台支持之所以能够成功的部分原因在于内核把源程序代码清晰地划分为体系结构无关部分和体系结构相关部分。在本章的后续部分将对这个问题进行更深入的讨论。

健壮性和安全性

Linux 必须健壮、稳定。系统自身应该没有任何缺陷，并它还应该可以保护进程（用户）以防止互相干扰，这就像把整个系统从其它系统中隔离开来加以保护一样。后一种考虑很大程度上是受信任的用户空间应用程序领域的问题，但是内核至少也应该提供支撑安全体系的原语（primitive）。健壮性和安全性比任何别的目标都要重要，包括速度。（系统崩溃的速度很快又有什么好处呢？）

保证 Linux 健壮性和安全性的唯一一个最重要的因素是其开放的开发过程，它可以被看作是一种广泛而严格的检查。内核中的每一行代码、每一个改变都会很快由世界上数不清的程序员检验。还有一些程序员专门负责寻找和报告潜在的缺陷——他们这样做完全是出于自己的个人爱好，因为他们也希望自己的 Linux 系统能够健壮安全。以前检查中所没有发现的缺陷可以通过这类人的努力来定位、修复，而这种修复又合并进主开发树以使所有的人都能

够受益。安全警告和缺陷报告通常在几天甚至几个小时内就能够得到处理和修复。

Linux 可能并不一定是现有的最安全的操作系统（很多人认为这项桂冠应该属于 OpenBSD，它是一个以安全性为主要目标的 Unix 变种），但是它是一个有力的竞争者。而且 Linux 健壮性远没有发展到尽头。

速度

这个术语经常自己就可以说明问题。速度几乎是最重要的衡量标准，虽然其等级比健壮性、安全性和（在有些时候）兼容性的等级要低。然而它却是代码最直观的几个方面之一。Linux 内核代码经过了彻底的优化，而最经常使用的部分——例如调度——则是优化工作的重点。几乎在任何时候都有一些不可思议的代码，这是由于这种方式的执行速度比较快。（这并不总是很明显，但是你经常不得不通过自己的试验来对这种优化代码进行确认。）虽然有时一些更直接的实现方法速度也很快，但是我所见过的这种情况屈指可数。

在某些情况下，本书推荐用可读性更好的代码来替代那些以速度的名义而被故意扭曲了的代码。虽然速度是一个设计目标，但我基本上只在以下两种情况时才会这样做：a) 在所考虑的问题中，速度明显不是关键问题 b) 没有其它的办法。

内核体系结构初始

图 3.1 是一种类 Unix 操作系统的相当标准的视图，实际上，更细致的来说，该图能够说明所有期望具有平台无关特性的操作系统。它着重强调了内核的下面两个特性：

- 内核将应用程序和硬件分离开来。
- 部分内核是体系结构和硬件特有的，而部分内核则是可移植的。

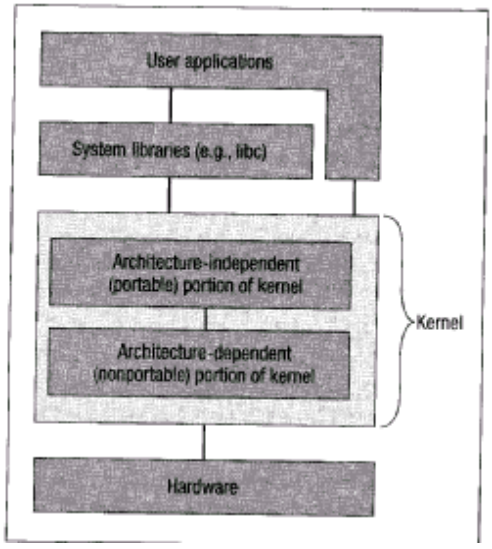

图 3.1 内核体系结构基本结构图

第一点我们在前面章节中已经讨论清楚了，在这里没有必要重复说明。第二点，也就是与体系结构无关和与体系结构相关代码的内容对于我们的讨论比较有意义。内核通过使用与处理用户应用程序相同的技巧来实现部分可移植性。这也就是说，如同内核把用户应用程序和硬件分离一样，部分内核将会因为与硬件的联系而同其它内核分离开来。通过这种分离，

用户应用程序和部分内核都成为可移植的。

虽然这通常并不能够使得内核本身更清楚，但是源程序代码的体系结构无关部分通常定义了与低层，也就是体系结构相关部分（或假定）的接口。作为一个简单的例子，内存管理代码中的体系结构无关部分假定只要包含特定的头文件就可以获得合适的 **PAGE_SIZE** 宏（参看 10791 行）的定义，该宏定义了系统的内存管理硬件用于分割系统地址空间的内存块的大小（参看第 8 章）。体系结构无关代码并不关心宏的确切定义，而把这些问题都留给体系结构相关代码去处理。（顺便一提，这比到处使用**#ifdef/#endif** 程序块来定义平台相关代码要清晰易懂得多。）

这样，内核向新的体系结构的移植就转变成为确认这些特性以及在新内核上实现它们的问题。

另外，用户应用程序的可移植性还可以通过它和内核的中间层次——标准 C 库（libc）——的协助来实现。应用程序实际上从不和内核直接通讯，而只通过 libc 来实现。图 3.1 中显示应用程序和内核直接通讯的唯一原因在于它们能够和内核通讯。虽然在实际上应用程序并不同内核直接通讯——这样做是毫无意义的。通过直接和内核通讯所能处理的问题都可以通过使用 libc 实现，而且更容易。

Libc 和内核通讯的方式是体系结构相关的（这和图中有一点矛盾），libc 负责将用户代码从实现细节中解放出来。有趣的是，甚至大部分 libc 都不了解这些细节。大部分的 libc，例如 **atoi** 和 **rand** 的实现，都根本不需要和内核进行通讯。剩余部分的大部分 libc，例如 **printf** 函数，在涉及到内核之前或之后就已经处理大量的工作。（**printf** 必需首先解释格式化字符串，分析相应参数，设定打印方法，在临时内部缓冲器中记录预期输出。直到此时它才调用底层系统调用 **write** 来实际打印该缓冲区。）其它部分的 libc 则只是相应系统调用的简单代理。因而一旦发生函数调用时，它们会立即调用内核相应函数以完成主要工作。在最低层次上，大部分 libc 通过单通道同内核进行交流，而它们所使用的机制将第 5 章中进行详细介绍。

由于这种设计，所有的用户应用程序，甚至大部分的 C 库，都是通过体系结构无关的方式和内核通讯的。

内核体系结构的深入了解

图 3.2 显示了内核概念化的一种可能方式。该图和区分内核的体系结构无关和体系结构相关的方法有所不同，它是一种更具有普遍性的结构视图。在“ Kernel ”框内的本书中有所涉及的内核部分都用括号注明了相应的章节编号。虽然有关对称多处理（SMP）的支持也属于本书的范围，但是在这里我们却没有标明章号。部分原因在于相当多的 SMP 代码广泛地分布于整个内核中，因此很难将它与某一个模块联系起来。同样的道理，对于内核初始化的支持也属于本书的范围，但是也没有标明章号。这样做仅仅是因为从设计的观点上看，该问题并不重要。最后，虽然在图中我们将第 6 章和“ 进程间通讯 ”框联系在一起，但是该章只涉及一部分进程间通讯的内容。

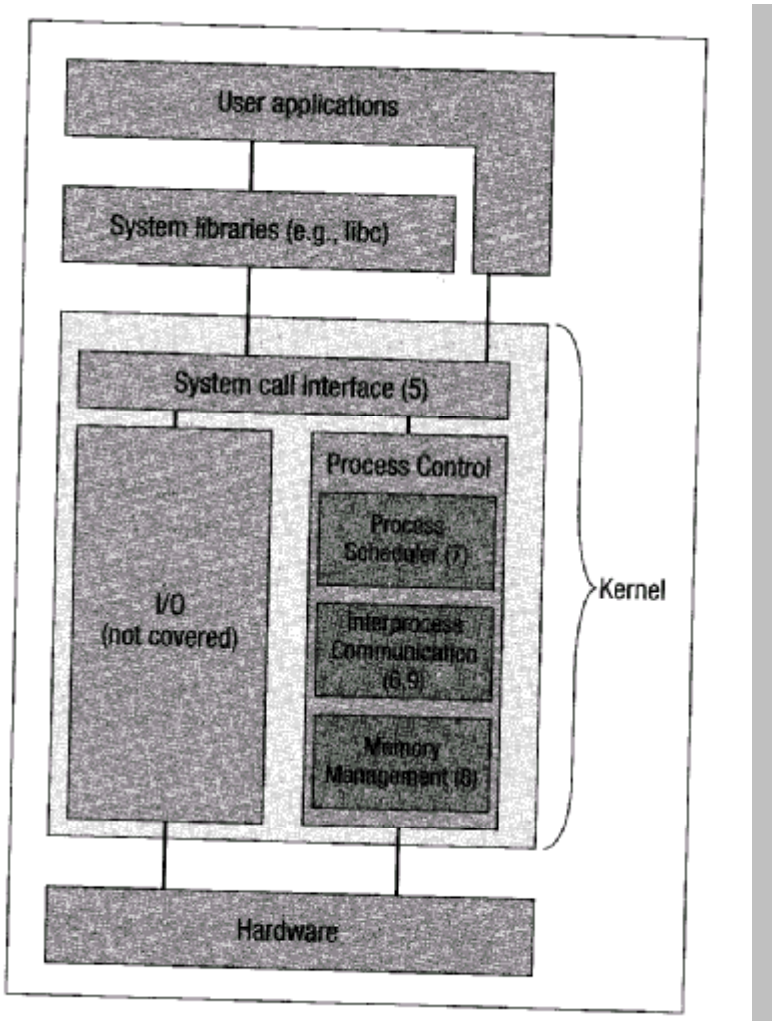

图 3.2 详细的内核体系结构图

进程和内核的交互通常需要通过如下步骤：

1. 用户应用程序调用系统调用，通常是使用 libc。
2. 该调用被内核的 **system_call** 函数截获（第 5 章，171 行），此后该函数会将调用请求转发给另外的执行请求的内核函数。
3. 该函数随即和相关内部代码模块建立通讯，而这些模块还可能需要和其它的代码模块或者底层硬件通讯。
4. 结果按照同样的路径依次返回。

然而，并不是所有内核和进程间的交互都是由进程发起的。内核有时也会自行决定同哪个进程交互，例如通过释放信号量或者简单的采用直接杀死进程的方法终止该进程的执行（如当进程用完所有可用的 CPU 时间片），以便使其它进程有机会运行。这些交互过程在该图中并没有表示，主要是因为它们通常都只是内核对自己的内部数据结构的修改（信号量传递对于这种规则来说是一个例外）。

是层次化（Layered），模块化（Modular）还是其它？

解决复杂性的所有方法都基于一个基本原理：问题分解和各个击破。也就是说，都是把

大型的、难以解决的问题（或系统）分解成一定数量的复杂度较低的子问题（或子系统），再根据需要重复这一过程直到每一部分都小到可以解决为止，而各种方法只是这种原理的一些不同运用而已。

计算机科学中有三种经典的方法比较适合于构建大型系统（我首先必须说明的是，这些定义都是经过我深思熟虑的讨论对象）。

- 层次（Layer）——将解决方案分解成若干部分，在这些部分中存在一个问题域的最底层，它为上层的抽象层次较高的工作提供基础。较高层建立在其低层基础之上。OSI 和 TCP/IP 协议堆栈是众所周知的层次化软件设计的成功的例子。操作系统设计的层次化解决方案可能会包含一个可以直接和硬件通讯的层次、然后在其上提供为更高层提供抽象支持的层次。这样更高层就可以对磁盘、网卡等硬件进行访问，而并不需要了解这些设备的具体细节。

  层次化设计的一个特征是要逐步构建符号集（vocabulary）。随着层次的升高，符号集的功能将越来越强大。层次化设计的另外一个特征是完全可以在对其上下层透明的条件下替换某一层次。在最理想的情况下，移植层次化的操作系统只需要重写最低层的代码。纯层次化模型实现的执行速度可能会很慢，因为高层必须（间接的）通过调用一系列连续的低层才能处理完自己的任务——N 层调用 N-1 层，N-1 层调用 N-2 层，等等，直到实际的工作在 0 层被处理完成。接着，结果当然是通过同样的路径反向传递回来。因此，层次化设计通常会包含对某些高层直接和某些低层通讯的支持；这样虽然提高了速度，但是却使得各个层次的替换工作更加困难（因为不止一个高层会直接依赖于这个你所希望进行替换的层次）。
- 模块（Module）——模块将具体的一部分功能块隐藏在抽象的接口背后。模块的最大特点是将接口和其实现分离开来，这样就能够保证一个模块可以在不影响其它模块的情况下进行改变。这样也将模块之间的依赖关系仅仅限定于接口。模块的范围是试图反映求解域内一些方面的自然的概念性界限。纯模块化的操作系统因而就可能有一个磁盘子系统模块，一个内存管理子系统模块，等等。纯模块化和纯层次化的操作系统之间的主要区别是一个可以由其它模块自由调用，模块间没有上层和下层的概念。（从这个意义上来说，模块是广义的层次。按照纯粹的观点，层次是最多可供一个其它模块调用的模块，这个模块也就是它的直接上层模块。）
- 对象（Object）——对象和模块不同，因为对于初学者来说它们具有不同的问题考虑方式，实现的方法也可能各自独立。但是，就我们当前的目的来说，对象不过是结构化使用模块的方法。组件（Component）作为对象思想的进一步改进目前还没有在操作系统设计中广泛使用。即便如此（按照我们的观点），我们也没有足够的理由将其和模块划分在不同的范畴中。

图 3.1 强调了内核的层次化的视图，而且是体系结构无关层次位于体系结构相关层次之上。（更为精确的视图是在顶层增加一个附加的体系结构相关的层次。这是因为系统调用接口位于应用程序和内核之间，而且是体系结构相关的。）图 3.2 着重强调了更加模块化的内核视图。

从合理的表述层次上看，这两种观点都是正确的。但也可以说这两种观点都是错误的。我可以用大量的图片向你证明内核是遵从所有你所能够指出的设计原则集合的，因为它就是从众多思想中抽取出来的。简单说来，事实是 Linux 内核既不是严格层次化的，也不是严格模块化的，也不是严格意义上的任何类型，而是以实用为主要依据的。（实际上，如果要用一个词来概括 Linux 从设计到实现的所有特点，那么实用就是最确切的。）也许最保守的观点是内核的实现是模块化的，虽然这些模块有时会为了追求速度而有意跨越模块的界限。

这样，Linux 的设计同时兼顾了理论和实际。Linux 并没有忽视设计方法；相反，在 Linux

的开发基本思想中，设计方法的作用就像是编译器：它是完成工作的有力工具。选择一个基本的设计原则（例如对象）并完全使用这种原则，不允许有任何例外，这对于测试该原则的限制，或者构建以说明这些方法为目的的教学系统来说都是一个不错的方法。但是如果要用它来达到 Linux 的设计目标则会引起许多问题。而且 Linux 的设计目标中也并不包括要使内核成为一个完全纯化的系统。Linux 开发者为了达到设计目标宁愿违背妨碍目标实现的原则。

实际上，如果对于 Linux 来说是正确的，那么它们对于所有最成功的设计来说都是正确的。最成功、应用最广泛的实际系统必然是实用的系统。有些开发人员试图寻找功能强大的可以解决所有问题的特殊方法。他们一旦找到了这种方法，所有的问题就都迎刃而解了。像 Linux 内核一样的成功设计通常需要为系统的不同部分和描述上的不同层次使用不同的方法。这样做的结果可能不是很清晰，也不是很纯粹，但是这种混合产物比同等功能的纯粹系统要强大而且优秀得多。

Linux 大部分都是单内核的

操作系统内核可能是微内核，也可能是单内核（后者有时称之为宏内核 Macrokernel）。按照类似封装的形式，这些术语定义如下：

- 微内核（Microkernel kernel）——在微内核中，大部分内核都作为独立的进程在特权状态下运行，它们通过消息传递进行通讯。在典型情况下，每个概念模块都有一个进程。因此，如果在设计中有一个系统调用模块，那么就必然有一个相应的进程来接收系统调用，并和能够执行系统调用的其它进程（或模块）通讯以完成所需任务。

  在这些设计中，微内核部分经常只不过是一个消息转发站：当系统调用模块要给文件系统模块发送消息时，消息直接通过内核转发。这种方式有助于实现模块间的隔离。（某些时候，模块也可以直接给其它模块传递消息。）在一些微内核的设计中，更多的功能，如 I/O 等，也都被封装在内核中了。但是最根本的思想还是要保持微内核尽量小，这样只需要把微内核本身进行移植就可以完成将整个内核移植到新的平台上。其它模块都只依赖于微内核或其它模块，并不直接直接依赖硬件。

  微内核设计的一个优点是在不影响系统其它部分的情况下，用更高效的实现代替现有文件系统模块的工作将会更加容易。我们甚至可以在系统运行时将开发出的新系统模块或者需要替换现有模块的模块直接而且迅速的加入系统。另外一个优点是不需要的模块将不会被加载到内存中，因此微内核就可以更有效的利用内存。
- 单内核（Monolithic kernel）——单内核是一个很大的进程。它的内部又可以被分为若干模块（或者是层次或其它）。但是在运行的时候，它是一个独立的二进制大映象。其模块间的通讯是通过直接调用其它模块中的函数实现的，而不是消息传递。

单内核的支持者声称微内核的消息传递开销引起了效率的损失。微内核的支持者则认为因此而增加的内核设计的灵活性和可维护性可以弥补任何损失。

我并不想讨论这些问题，但必须说明非常有趣的一点是，这种争论经常会令人想到前几年 CPU 领域中 RISC 和 CISC 的斗争。现代的成功 CPU 设计中包含了所有这两种技术，就像 Linux 内核是微内核和单一内核的混合产物一样。Linux 内核基本上是单一的，但是它并不是一个纯粹的集成内核。前面一章所介绍的内核模块系统将微内核的许多优点引入到 Linux 的单内核设计中。（顺便提一下，我考虑过一种有趣的情况，就是 Linux 的内核模块系统可以将系统内核转化成为简单的不传递消息的微内核设计。虽然我并不赞成，但是它仍然是一个有趣的想法。）

为什么 Linux 必然是单内核的呢？一个方面是历史的原因：在 Linus 的观点看来，通过把内核以单一的方式进行组织并在最初始的空间中运行是相当容易的事情。这种决策避免了有关消息传递体系结构，计算模块装载方式等方面的相关工作。（内核模块系统在随后的几

年中又进行了不断地改进。）

另外一个原因是充足的开发时间的结果。Linux 既没有开发时间的限制，也没有深受市场压力的发行进度。 所有的限制只有并不过分的对内核的修改与扩充。内核的单一设计在内部实现了充分的模块化，在这种条件下的修改或增加都并不怎么困难。而且问题还在于没有必要为了追求尚未证实的可维护性的微小增长而重写 Linux 的内核。（Linus 曾多次特别强调了如下的观点：为了这点利益而损耗速度是不值得的。）后面章节中的部分内容将详细的重新考虑充足开发时间的效果。

如果 Linux 是纯微内核设计，那么向其它体系结构上的移植将会比较容易。实际上，有一些微内核，如 Mach 微内核，就已经成功的证明了这种可移植性的优点。实际的情况是，Linux 内核的移植虽然不是很简单，但也绝不是不可能的：大约的数字是，向一个全新的体系结构上的典型的移植工作需要 30,000 到 60,000 行代码，再加上不到 20,000 行的驱动程序代码。（并不是所有的移植都需要新的驱动程序代码。）粗略的计算一下，我估计一个典型的移植平均需要 50,000 行代码。这对于一个程序员或者最多一个程序小组来说是力所能及的，可以在一年之内完成。虽然这比微内核的移植需要更多的代码，但是 Linux 的支持者将会提出，这样的 Linux 内核移植版本比微内核更能够有效的利用底层硬件，因而移植过程中的额外工作是能够从系统性能的提高上得到补偿的。

这种特殊设计的权衡也不是很轻松就可以达到的，单内核的实现策略公然违背了传统的看法，后者认为微内核是未来发展的趋势。但是由于单一模式（大部分情况下）在 Linux 中运行状态良好，而且内核移植相对来说比较困难，但没有明显地阻碍程序员团体的工作，他们已经热情高涨地把内核成功的移植到了现存的大部分实际系统中，更不用说类似掌上型电脑的一些看起来很不实际的目标了。只要 Linux 的众多特点仍然值得移植，新的移植版本就会不断涌现。

设计和实现的关系

接下来的部分将介绍一些内核设计和实现之间的关系。本部分最重要的内容是对于内核源程序目录结构的概述，这一点随后就会提到。本章最后以实现中体系结构无关代码和体系结构相关代码的相对大小的估算作为总结。

内核源程序目录结构

按照惯例，内核源程序代码安装在/usr/src/linux 目录下。在该目录下还有几个其它目录，每一个都代表一个特定的内核功能性子集（或者非常粗略的说是高层代码模块）。

Documentation

这个目录下面没有内核代码，只有一套有用的文档。但是这些文档的质量不一。有一部分内核文档，例如文件系统，在该目录下有相当优秀而且相当完整的文档；而另外一部分内核，例如进程调度，则根本就没有文档。但是在这里你可以不时的发现自己所最需要的东西。

arch

arch 目录下的所有子目录中都是体系结构相关的代码。每个体系结构特有的子目录下都又至少包含三个子目录：kernel，存放支持体系结构特有的诸如信号量处理和 SMP 之类特征的实现；lib，存放高速的体系结构特有的诸如 **strlen** 和 **memcpy** 之类的通用函数的实现；以及 mm，存放体系结构特有的内存管理程序的实现。

除了这三个子目录以外，大多数体系结构在必要的情况下还都有一个 boot 子目录，该目录中包含有在这种平台上启动内核所使用的部分或全部平台特有代码。这些启动代码中的部分或全部也可以在平台特有的内核目录下找到。

最后，大部分体系结构所特有的目录还可以根据需要包含了供附加特性或改进的组织使用的其它子目录。例如，i386 目录包含一个 math-emu 子目录，其中包括了在缺少数学协处理器（FPU）的 CPU 上运行模拟 FPU 的代码。作为另外一个例子，m68k 移植版本中为每一个该移植版本所支持的基于 680x0 的机器建立了一个子目录，从而这些机器所特有的代码都有一个自然的根目录。

下面几个是 arch 目录下的子目录：

- arch/alpha/——Linux 内核到基于 DEC Alphs CPU 工作站的移植。
- arch/arm/——Linux 到 ARM 系列 CPU 的移植，该类 CPU 主要用于诸如 Corel 的 NetWinder 和 Acorn RiscPC 之类的机器。
- arch/i386/——最接近于 Linux 内核原始平台或标准平台。这是为 Intel 的 80386 结构使用的，当然包括对同一系列后来的 CPU（80486，Pentium 等等）的支持。它还包括了对 AMD，Cyrix 和 IDT 等公司的一些兼容产品的支持。

  本书基本上将这种体系结构称为“ x86 ”。即使这样，严格说来“ x86 ”对于我们的目标来说还是要求得过于宽泛。早期的 Intel CPU，例如 80286，并没有包括 Linux 运行所需的所有特性。对于这些机器，Linux 也没有正式的支持版本。（顺便提一下，Linux 对这种 CPU 的独立移植版本是存在的，不过它在功能上有部分损失。）当本书中提到“ x86 平台 ”时，通常是指 80386 或更新的 CPU。
- arch/m68k/——到 Motorola 的 680x0 CPU 系列的移植。该版本可以提供对基于从 68020（只要它同内存管理单元（MMU）68851 一起使用）到 68060 的一切机器的支持。很多公司在他们的产品中使用 680x0 系列芯片，例如 Commodore（现在是 Gateway）的 Amiga，Apple 的 Macintosh，Atari ST，等等。这些老机器中的很多现在正充当可靠的 Linux 工作站。另外，到 NeXT 工作站和 SUN 3 工作站的移植也正在进行中。
- arch/mips/——到 MIPS 的 CPU 系列的移植。虽然有其它几个厂商也使用 MIPS 开发了一些系统，但是基于这种 CPU 的最出名的机器是 Silicon Graphics（SGI）工作站。
- arch/ppc/——到 Motorola/IBM 的 PowerPC 系列 CPU 的移植。这包括对基于 PowerPC 的 Macintosh 和 Amiga 以及 BeBox、IBM 的 RS/6000 等其它一些机器的支持。
- arch/sparc/——到 32 位 SPARC CPU 的移植。这包括对从 Sun SPARC 1 到 SPARC 20 的全部支持。
- arch/sparc64/——到基于 64 位 SPARC CPU（UltraSPARC 系）系统的移植。这里所能够支持的机器包括 Sun 的 Ultra 1，Ultra 2 和更高配置的机器，直到 Sun 的最新产品 Enterprise 10000。注意 32 位和 64 位的 SPARC 的移植版本正在合并中。

不幸的是，本书必须将注意力集中在 x86 上，因此只应用到了 arch/i386/目录下的代码，而其它体系结构所特有的代码将不再涉及了。

drivers

这个目录是内核中非常大的一块。实际上，drivers 目录下包含的代码占整个内核发行版本代码的一半以上。它包括显卡、网卡、SCSI 适配器、软盘驱动器，PCI 设备和其它任何你可以说出的 Linux 支持的外围设备的软件驱动程序。

Drivers 目录下的一些子目录是平台特有的，例如，zorro 子目录中包含有和 Zorro 总线通讯的代码。而 Zorro 总线只在 Amiga 中使用过，因此这些代码必然是 Amiga 特有的。而其它一些子目录，例如 pci 子目录，则至少是部分平台无关的。

fs

Linux 支持的所有文件系统在 fs 目录下面都有一个对应的子目录。一个文件系统（file system）是存储设备和需要访问存储设备的进程之间的媒介。

文件系统可能是本地的物理上可访问的存储设备，例如硬盘或 CD-ROM 驱动器；在这两种情况下将分别使用 ext2 和 isofs 文件系统。文件系统也可能是可以通过网络访问的存储设备；这种情况下使用的文件系统是 NFS。

还有一些伪文件系统，例如 proc 文件系统，可以以伪文件的形式提供其它信息（例如，在 proc 的情况下是提供内核的内部变量和数据结构）。虽然在底层并没有实际的存储设备与这些文件系统相对应，但是进程可以像有实际存储设备一样处理（NFS 也可以作为伪文件系统来使用）。

include

这个目录包含了 Linux 源程序树中大部分的包含（.h）文件。这些文件按照下面的子目录进行分组：

- include/asm-*/——这样的子目录有多个，每一个都对应着一个 arch 的子目录，例如 include/asm-alpha，include/asm-arm，include/asm-i386 等等。每个目录下的文件中包含了支持给定体系结构所必须的预处理器宏和短小的内联函数。这些内联函数很多都是全部或部分地使用汇编语言实现的，而且在 C 或者汇编代码中都会应用到这些文件。
  当编译内核时，系统将建立一个从 include/asm 到目标体系结构特有的目录的符号链接。结果是体系结构无关内核源程序代码可以使用如下形式的代码来实现所需功能：
  #include <asm/some-file>
  这样就能够将适当地体系结构特有的文件包含（**#include**）进来。
- include/linux/——内核和用户应用程序请求特定内核服务时所使用的常量和数据结构在头文件中定义，而该目录中就包含了这些头文件。这些文件大都是平台独立的。这个目录被全部复制（更多的情况是链接）到/usr/inlude/linux 下。这样用户应用程序就可以使用#include 包含这些头文件，而且能够保证所包含进来的头文件的内容和内核中的定义一致。第 9 章将会给出有关的一个样例。
- 对这些文件的移植只有对于内核来说才是必须的，对用户应用程序则没有必要。移植工作可以按照如下的方式封装处理：
  ```
  /* …   Stuff for user apps and kernel … */
  #ifdef   __KERNEL__

  /* … Stuff for kernel only … */
  #endif   /* __KERNEL__ */
  ```
- include/net/——这个目录供与网络子系统有关的头文件使用。
- include/scsi/——这个目录供与 SCSI 控制器和 SCSI 设备有关的头文件使用。
- include/video/——这个目录供与显卡和帧显示缓存有关的头文件使用。

init

这个目录下面的两个文件中比较重要的一个是 main.c，它包含了大部分协调内核初始化的代码。第 4 章将详细介绍这部分代码。

ipc

这个目录下的文件实现了 System V 的进程间通讯（IPC）。在第 9 章中将会对它们进行详细介绍。

kernel

这个目录中包含了 Linux 中最重要的部分：实现平台独立的基本功能。这部分内容包括进程调度（kernel/sched.c）以及创建和撤销进程的代码（kernel/fork.c 和 kernel/exit.c）；以上所有的以及其它部分内容将在第 7 章中有所涉及。但是我并不想给你留下这样的印象：需要了解的内容都在这个目录下。实际上在其它目录下也有很多重要的内容。但是，不管怎样说，最重要部分的代码是在这个目录下的。

lib

lib 目录包含两部分的内容。lib/inflate.c 中的函数能够在系统启动时展开经过压缩的内核（请参看第 4 章）。lib 目录下剩余的其它文件实现一个标准 C 库的有用子集。这些实现的焦点集中在字符串和内存操作的函数（**strlen**，**mmcpy** 和其它类似的函数）以及有关 **sprintf** 和 **atoi** 的系列函数上。

这些文件都是使用 C 语言编写的，因此在新的内核移植版本中可以立即使用这些文件。正如本章前面部分说明的那样，一些移植提供了它们独有的高速的函数版本，这些函数通常是经过手工调整过的汇编程序，在移植后的系统使用这些函数来代替原来的通用函数。

mm

该目录包含了体系结构无关的内存管理代码。正如我们前面说明的那样，为每个平台实现最低层的原语的体系结构特有的内存管理程序是存储在 arch/platform/mm 中的。大部分平台独立和 x86 特有的内存管理代码将在第 8 章中介绍。

net

这个目录包含了 Linux 应用的网络协议代码，例如 AppleTalk，TCP/IP，IPX 等等。

scripts

该目录下没有内核代码，它包含了用来配置内核的脚本。当运行 **make menuconfig** 或者 **make xconfig** 之类的命令配置内核时，用户就是和位于这个目录下的脚本进行交互的。

体系结构相关和体系结构无关的代码

现在我们来估计一下体系结构相关和体系结构无关代码的相对大小。我们首先给出一些数字。完整的 2.2.5 的内核总共有 1,725,645 行代码。（顺便一提，请注意本书只包含了 39,000 行代码，但是我们仍然努力涵盖了相当部分的核心函数。）其中一共有 392,884 行代码在体系结构特有的目录之内，也就是 arch/*和 include/asm-*下面。我估计还有超过 64,000 行的代码是仅供一种体系结构专用的驱动程序。这意味着大约 26%的代码是专用于特定体系结构的。

但是，对于单一一种体系结构，体系结构相关代码比例相对较小。不妨理想一点，如果某种体系结构所需要的特有代码约有 50,000 行，而体系结构无关代码则大约有 1,250,000 行，那么体系结构相关代码大概只占到 4%。当然，在特定的一个内核中，并不是所有这些体系结构无关代码都会被用到，因此体系结构相关代码在特定内核中所占的比重与内核的配置有关。但是不管怎样，很显然大部分内核代码是平台独立的。