# 第1章 设备驱动程序简介

以Linux为代表的自由操作系统的众多优点之一，就是它们的内部源代码可供所有人查看。操作系统曾经是一个黑暗而神秘的领域，其代码仅局限于少数程序员，现在具备必要技能的人员随时对其都可以进行检查，理解和修改。Linux已经帮助操作系统实现了“平民化”。Linux内核仍然是一个庞大而复杂的代码体，然而，那些想要成为内核“黑客”的需要一个入口点，以此为突破口逐渐理解代码，而不会被复杂性所淹没。通常，设备驱动程序提供了该方法。

设备驱动程序在Linux内核中扮演着特殊的角色。它们是一个个不同的“黑盒子”，它提供了良好定义的内部编程接口来响应某一种特定的硬件，完全隐藏了设备工作的细节。用户活动通过一组独立于具体驱动程序的标准化调用来执行；然后将这些调用映射到作用于真实硬件的设备具体操作上，就是设备驱动程序的职责了。这种编程接口使得驱动程序可以与内核的其余部分分开构建，并在需要的时候，运行时“插入”。这种模块化的方式，使Linux驱动程序易于编写，到目前为止，有多达数百种可用。

有太多的理由对编写Linux设备驱动程序产生兴趣。在可预见的未来，新硬件更新换代的速率必然导致驱动程序开发者会非常忙。个人也可能为了访问某一个特定的设备而了解驱动程序。硬件厂商通过为他们的产品提供Linux驱动程序，可以将庞大而不断增长的Linux用户群添加到他们的潜在市场中。Linux系统的开源性意味着，如果驱动程序开发者愿意，驱动程序的源代码可以迅速传播给数百万用户。

本书教你如何编写自己的驱动程序，以及如何在内核的相关部分畅游。我们采取了独立于设备的方法；尽可能的展示编程技术和接口，而不是依赖于具体的设备。每个驱动程序都是不同，作为驱动开发者，你需要深入理解你的特定设备。但大多数驱动的原理和基本技术都是一样的。本书不讲解具体的设备信息，而是告诉你，要想设备工作而需要的背景知识。

当你学习编写驱动程序时，你会发现很多关于Linux内核的知识；这可以帮助了解机器是如何工作的，以及为什么事情并不总是像你期望的那样快，或者没有做到你想做的事。我们逐渐引入新的想法，以非常简单的驱动程序开始并在其上进行构建；每一个新概念都伴随着不需要特殊硬件进行测试的示例代码。

本章并没有涉及编写代码。但是，我们会介绍一些关于Linux内核的概念。理解这些概念对于后面编程学习是很有帮助的。

## 1.1 设备驱动的角色

作为一名程序员，您可以自行选择您的驱动程序，并在所需的编程时间和结果的灵活性之间选择一个可接受的平衡点。虽然说驱动程序是“灵活的”可能很奇怪，但我们喜欢这个词，因为它强调设备驱动程序的角色是提供机制而不是策略。

机制和策略之间的区别是Unix设计背后的最佳想法之一。大多数编程问题确实可以分为两部分：“要提供什么功能”（机制）和“如何使用这些功能”（策略）。如果这两个问题由程序的不同部分或者甚至完全由不同的程序解决，则软件包更容易开发且更容易适应特定需求。

例如，Unix图形显示管理分为X服务器，及窗口和会话管理器；X服务器负责感知硬件，提供给用户程序一个统一的接口；窗口和会话管理器在不知道硬件的情况下，实现了一种特定的策略。用户可以在不同的硬件上使用相同的窗口管理器，不同的用户可以在相同的工作站上运行不同的配置。即使完全不同的桌面环境，诸如KDE和GNOME，也可以在同一个系统上共存。另一个例子是TCP/IP网络的分层结构：操作系统提供套接字抽象，它不实现有关要传输的数据的策略，而不同的服务器负责服务（及其相关的策略）。进一步讲，像ftpd这样的服务器提供了文件传输机制，而用户可以使用他们喜欢的任何客户端;即使存在命令行和图形客户端，任何人也都可以编写新的用户界面来传输文件。

涉及驱动程序的地方，采取同样的机制和策略分离的思想。软盘驱动器（floppy）就是无策略的，它的作用只是将软盘显示为连续的数据块。系统的更高级别提供策略，例如谁可以访问软盘驱动器，直接访问驱动器还是通过文件系统访问，以及用户是否可以在驱动器上装入文件系统。由于不同的环境通常需要以不同的方式使用硬件，因此尽可能无策略是很重要的。

编写驱动程序时，程序员应该特别注意这个基本概念：编写内核代码来访问硬件，但不要强制用户的特定策略，因为不同的用户有不同的需求。驱动程序应该使硬件可用，关于如何使用硬件的所有问题留给应用程序。如果驱动程序在不增加约束的情况下提供对硬件功能的访问，则它是灵活的。然而有时候，一些策略还是必须的。例如，数字I/O驱动程序可能只提供字节宽度的硬件访问权限，以避免处理单个位所需的额外代码。

您也可以从不同的角度来看待您的驱动程序：它是位于应用程序和实际设备之间的软件层。驱动程序的这种特权角色允许驱动程序员准确选择设备的显示方式：不同的驱动程序可以提供不同的功能，即使是相同的设备。实际的驱动程序设计应该是许多不同考虑之间的平衡。例如，一个设备可以被不同的程序并发使用，并且驱动程序员可以完全自由地决定如何处理并发。您可以实现设备的内存映射而不依赖其硬件功能，或者您可以提供一个用户库来帮助应用程序员在可用基元之上实施新策略等等。其中一个主要考虑因素是想向用户提供尽可能多的选项，和编写驱动程序所需的时间及保持简单以避免错误发生之间的平衡。

无策略驱动程序具有许多典型特征。其中包括对同步和异步操作的支持，多次打开的能力，利用硬件全部功能的能力以及缺乏软件层来“简化事物”或提供与策略相关的操作。这种驱动程序不仅对最终用户更好，而且写起来也更容易。无策略是软件设计者的通用目标。

实际上，许多设备驱动程序与用户程序一起发布，以帮助配置和访问目标设备。这些程序可以从简单程序到完整的图形应用程序。示例包括tunelp程序，它调整并行端口打印机驱动程序的操作方式，以及作为PCMCIA驱动程序包一部分的显卡控制程序。通常也提供客户端库，它提供的功能是没有作为驱动程序本身的一部分实现的功能。

本书的范围是内核，所以我们尽量不处理策略问题或应用程序或支持库。有时我们会谈论不同的策略以及如何支持这些策略，但我们不会详细介绍使用该设备的程序或实施的策略。但是，您应该了解，用户程序是软件包不可或缺的一部分，即使是无需使用策略的软件包也可以使用将默认行为应用于基础机制的配置文件进行分发。

## 1.2 划分内核

在Unix系统中，几个并发进程参与不同的任务。每个进程都会要求系统资源，无论是计算能力，内存，网络连接还是其它资源。内核是负责处理所有这些请求的可执行代码的整体。虽然不同的内核任务之间的区别并不总是清楚地标出，但是内核的职责可以被拆分（如图1-1所示）为以下几部分：

* 进程管理  
  内核负责创建和销毁进程并处理与外部的连接（输入和输出）。不同进程之间的通信（通过信号，管道或进程间通信原语）是整个系统功能的基础，也是由内核处理的。另外，调度程序控制进程如何共享CPU，它是进程管理的一部分。更一般地说，内核的进程管理活动在单个CPU或几个CPU之上实现了几个进程的抽象。
* 内存管理  
  计算机的内存是主要资源，用于处理它的策略对系统性能至关重要。内核为所有进程在有限的可用资源之上构建虚拟地址空间。内核的不同部分通过一组函数调用与内存管理子系统进行交互，从简单的malloc/free，到更复杂的功能。
* 文件系统  
  Unix主要基于文件系统概念；几乎Unix中的所有东西都可以看作是一个文件。内核在非结构化硬件的基础上构建了一个结构化的文件系统，并且生成的文件抽象在整个系统中大量使用。另外，Linux支持多种文件系统类型，即在物理介质上组织数据的不同方式。例如，可以使用Linux标准的ext3文件系统，常用的FAT文件系统或其它几种格式化磁盘。
* 设备控制  
  几乎每个系统操作最终都映射到一个物理设备。除了处理器，存储器和其它几个实体之外，所有设备控制操作都是通过特定于被寻址设备的代码执行的。该代码被称为设备驱动程序。内核必须为系统上的每个外设嵌入一个设备驱动程序，从硬盘驱动器到键盘和磁带机。内核的这个功能是本书中我们主要感兴趣的地方。
* 联网  
  网络必须由操作系统管理，因为大多数网络操作不是特定于进程：接收数据包是异步事件。数据包必须在进程处理之前收集，识别和分派。系统负责在程序和网络接口之间传送数据包，并且必须根据网络活动来控制程序的执行。另外，所有的路由和寻址策略问题都在内核中实现。

### 1.2.1可加载模块

Linux一个很好的功能是能够在运行时扩展内核提供的功能。 这意味着您可以在系统启动并运行时向内核添加功能（或删除功能）。

可以在运行时添加到内核的代码称为module。Linux内核支持多种不同类型（或类别）的模块，包括但不限于设备驱动程序。 每个模块由目标代码组成（未链接成完整的可执行文件），可以通过insmod动态链接到正在运行的内核，并且可以由rmmod程序取消链接。

图1-1表示了负责特定任务的不同类别的模块-模块根据其提供的功能属于某一特定类别。 图1-1中的模块放置覆盖了最重要的类，但是远远没有完成，因为Linux中越来越多的功能正在模块化。

## 1.3设备和模块类

根据Linux查看设备的方式可以将设备划分为3种基本设备类型，字符模块、块模块、网络模块。

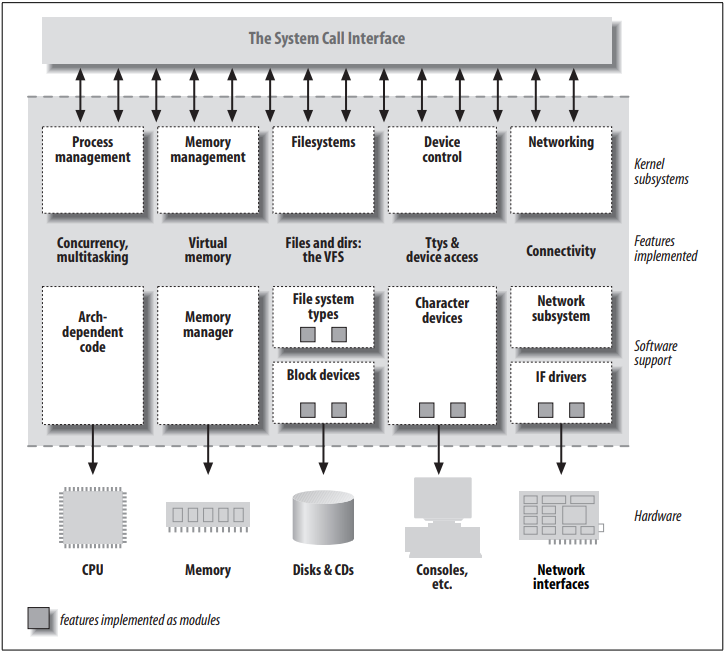


图1-1 简要内核框图

* 字符设备  
  字符（char）设备是可以作为字节流访问的设备（如文件）；字符设备驱动程序就是实现这种行为的程序。通常，至少实现open，close，read和write等系统调用。文本控制台（/dev/console）和串行端口（/dev/ttyS0）是典型的字符设备，因为它们能够很好地用数据流表示。字符设备通过文件系统节点访问，如/dev/tty1和/dev/lp0。字符设备和普通文件之间唯一的区别在于，常规文件中可以来回移动，而大多数字符设备只是数据通道，只能依次访问。然而，也有字符设备看起来像数据区域，你可以在它们之间来回移动；例如，图像采集卡，应用程序可以使用mmap或lseek访问整个采集的图像。
* 块设备  
  像字符设备一样，块设备可以通过/dev目录中的文件系统节点访问。块设备是可以托管文件系统的设备（例如，磁盘）。在大多数Unix系统中，块设备只能处理传输一个或多个块的I/O操作，这些块通常是512字节（或更大）。与此相反的是，Linux允许应用程序像字符设备那样读写块设备-它允许一次传输任意数量的字节。因此，块设备和字符设备的区别仅在于内核管理数据的方式，也就是内核/驱动程序软件接口。像字符设备一样，每个块设备都通过文件系统节点访问，它们之间的区别对用户来说是透明的。块驱动程序与字符设备驱动相比，与内核的接口完全不同。
* 网络接口  
  任何网络事务都是通过一个接口完成的，也就是能够与其它主机交换数据的设备。通常，接口是硬件设备，但它也可能是纯粹的软件设备，如loopback接口。网络接口负责发送和接收由内核网络子系统驱动的数据包，而不知道单个事务如何映射到正在传输的实际数据包。许多网络连接（特别是使用TCP的网络连接）都是面向流的，但网络设备通常是围绕数据包的传输和接收而设计的。网络驱动程序对个人连接一无所知;它只处理数据包。  
  不是面向流的设备，网络接口不容易映射到文件系统中的节点，如/dev/tty1所示。提供对接口访问的Unix方式仍然是给它们分配一个唯一的名字（比如eth0），但是这个名字在文件系统中没有相应的条目。内核和网络设备驱动程序之间的通信与使用字符和块驱动程序完全不同。内核调用与数据包传输相关的函数，而不用读写。

还有其它方法对上述设备驱动程序模块进行分类。一般来说，某些类型的驱动程序可以为给定类型的设备使用额外的内核支持功能层。例如，可以讨论通用串行总线（USB）模块，串行模块，SCSI模块等。每个USB设备都由可与USB子系统配合使用的USB模块驱动，但设备本身在系统中显示为字符设备（例如USB串行端口），块设备（USB存储卡读取器）或 一个网络设备（一个USB以太网接口）。

Other classes of device drivers have been added to the kernel in recent times, including FireWire drivers and I2O drivers. In the same way that they handled USB and SCSI drivers, kernel developers collected class-wide features and exported them to driver implementers to avoid duplicating work and bugs, thus simplifying and strengthening the process of writing such drivers.

其他类别的设备驱动程序最近已添加到内核中，包括FireWire驱动程序和I2O驱动程序。 与他们处理USB和SCSI驱动程序的方式相同，内核开发人员收集了整个级别的功能，并将它们导出到驱动程序实施者，以避免重复工作和错误，从而简化和加强编写此类驱动程序的过程。

## 1.4安全问题

## 1.5版本编号

## 1.6许可条款

## 1.7加入内核开发社区

## 1.8本书概述

# 第2章 构建和运行模块

差不多是时候开始编程了。本章介绍了关于模块和内核编程的一些基本概念。寥寥几页中，通过构建并运行一个完整的模块，查看所有模块共有的一些基本代码。为了避免一次引出太多的概念，本章只讲解模块，而不涉及任何特定的设备类。

## 2.1 建立测试系统

从本章开始，开始使用例子验证一些编程概念。（所有的例子，在O’Reilly’s FTP网站上都能找到）构建，加载，和修改这些例子，都是增加对驱动程序的工作原理及如何与内核交互的理解。

示例模块应该可以与任何一种2.6.x内核一起使用，包括由发行商提供的发行版内核。但是，我们建议您直接从kernel.org镜像网络获取“主线”内核，并将其安装到您的系统上。发行商提供的内核与“主线” 内核有着很大差异，因为其往往打了大量的补丁文件；有时，发行商的补丁程序可能更改设备驱动程序所看到的内核API。如果你正在编写一个必须适用于特定发行版的驱动程序，那么你一定会想要针对相关的内核进行构建和测试。但是，为了学习驱动程序写作的目的，标准内核是最好的。

无论您的内核的来源如何，为2.6.x构建模块都要求您在系统上配置并构建内核树。这个要求与以前的内核版本发生改变，其中当前的头文件集是足够的。将2.6内核的模块链接到内核源代码树中找到的目标文件；结果是一个更强大的模块加载器，但也要求这些目标文件可用。因此，您的第一步是创建一个内核源代码树（来自kernel.org网络或您的经销商的内核源代码包），构建一个新的内核并将其安装到您的系统上。由于我们稍后会看到的原因，如果在构建模块时实际运行目标内核，通常是最简单的，但这不是必需的。

***注意：你也应该考虑一下你在哪里做模块实验，开发和测试。我们尽最大努力使我们的示例模块安全并正确，但存在错误的可能性始终存在。内核代码中的错误可能会导致用户进程崩溃，偶尔会导致整个系统崩溃。它们通常不会产生更严重的问题，例如磁盘损坏。尽管如此，最好还是在不包含无法承受损失的数据的系统上进行内核实验，而这些数据不会执行基础服务。 为了测试新代码，内核黑客通常会保留一个“牺牲”系统。***

## 2.2 Hello World

无处不在的“Hello World”。代码内容如下：

|  |
| --- |
| #include <linux/init.h>  #include <linux/module.h>  MODULE\_LICENSE("Dual BSD/GPL");  static int hello\_init(void)  {  printk(KERN\_ALERT "Hello, world\n");  return 0;  }  static void hello\_exit(void)  {  printk(KERN\_ALERT "Goodbye, cruel world\n");  }  module\_init(hello\_init);  module\_exit(hello\_exit); |

这个模块定义了2个函数，一个是当模块被加载时调用的函数（hello\_init）；还有一个是卸载模块时调用的函数(hello\_exit)。module\_init和module\_exit使用特定的内核宏表明这两个函数的职责。另外一个宏MODULE\_LICENSE用来告诉内核，该模块使用的许可证。

printk函数类似于C标准库中的printf函数。为什么模块能够调用内核的printk函数？那是因为模块被使用insmod命令加载进入内核后，模块就会被连接到内核，可以访问内核的公共符号（函数和变量）。KERN\_ALERT是消息的优先级。

你可以使用insmod和rmmod命令加载、卸载模块程序，注意的是，只有root用户具有加载、卸载的权限。

|  |
| --- |
| % make  make[1]: Entering directory `/usr/src/linux-2.6.10'  CC [M] /home/ldd3/src/misc-modules/hello.o  Building modules, stage 2.  MODPOST  CC /home/ldd3/src/misc-modules/hello.mod.o  LD [M] /home/ldd3/src/misc-modules/hello.ko  make[1]: Leaving directory `/usr/src/linux-2.6.10'  % su  root# insmod ./hello.ko  Hello, world  root# rmmod hello  Goodby cruel world  root# |

请再次注意，上述步骤能够正常执行，您必须在makefile能够找到它的地方有正确配置和构建的内核树（在本例中为/usr/src/linux-2.6.10）。 我们将在“[2.4编译和加载](#_2.4_编译和加载)”一节中详细介绍如何构建模块。

根据系统用于传递消息行的机制，输出可能会有所不同。特别是，之前的屏幕dump是从文本控制台获取的；如果您正在窗口系统下运行的终端仿真程序运行insmod和rmmod，则屏幕上不会显示任何内容。消息发送到其中一个系统日志文件，例如/var/log/messages（实际文件的名称因Linux发行版而异）。第4章介绍了用于传递内核消息的机制。

如你所见，编写一个模块并不像你期望的那么困难-只要模块不需要做任何有价值的事情。难点在于了解设备以及如何最大限度地提高性能。在本章中，我们将详细探讨模块化，将针对具体设备的问题留到后面的章节。

## 2.3 内核模块与应用程序

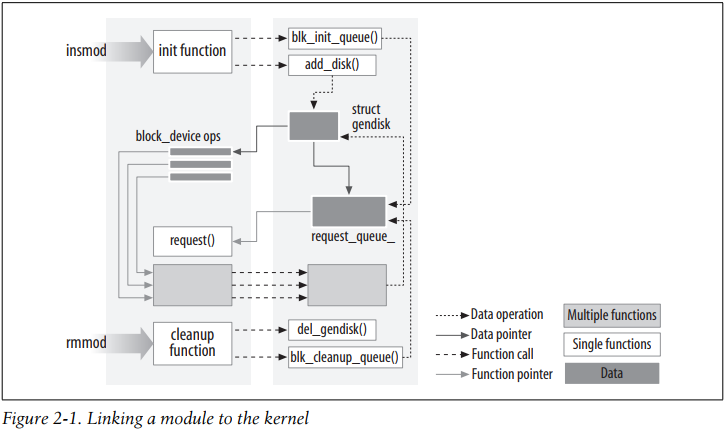
在我们继续深入之前，必须强调一下内核模块和应用程序之间的各种差异。

大部分的中小型应用程序从头到尾都在执行一个简单的任务，而内核模块仅仅是为了后面的请求而注册自己，初始化函数在注册完成后立即终止。换句话说，模块初始化函数的任务是为稍后调用模块的函数做准备;就好像这个模块在说“我在这里，而这正是我所能做到的。”模块的exit函数（示例中为hello\_exit）在模块卸载之前被调用。它应该告诉内核，“我不在那里了，不要再要求我做其它任何事情。”这种编程方法与事件驱动编程类似，但并非所有的应用程序都是事件驱动的，但每个内核模块都是。事件驱动的应用程序和内核代码之间的另一个主要区别在于exit函数：即将终止的应用程序可能懒惰地释放资源或者避免完全清理，模块的exit函数必须小心地撤消init函数所构建的所有内容，或者系统重启前一直存在的碎片。

顺便提一句，卸载模块是最应该欣赏的模块化特性之一，因为它有助于缩短开发时间；你可以测试新驱动程序的后续版本，而无需每次都经历漫长的关机/重启周期。

作为程序员，你知道应用程序可以调用它没有定义的函数：链接阶段使用适当的函数库来解析外部引用。printf是这些可调用函数之一，并在libc中定义。另一方面，模块只与内核链接，它可以调用的唯一函数是内核导出的函数；没有可链接的库。例如，早先在hello.c中使用的printk函数是在内核中定义并导出到模块的printf版本。它的行为与原始函数类似，但有一些细微差别，主要原因是缺少浮点支持。

图2-1显示了在模块中如何使用函数调用和函数指针向正在运行的内核添加新功能。



为没有库链接到模块，所以源文件不应该包含通用的头文件，唯一的例外是，<stdarg.h>和非常特殊的情况。内核模块中只能使用实际上是内核本身的一部分的函数。任何与内核相关的东西都在你已经设置和配置的内核源代码树中找到的头文件中声明;大多数相关头文件都包含在include/linux和include/asm中，但include的其它子目录已添加到与特定内核子系统关联的主机素材中了。

在本书中介绍了各个内核头文件的作用，因为它们都是需要的。

内核编程和应用程序编程之间的另一个重要区别在于如何处理故障：在应用程序开发过程中segmentation fault是无害的，而调试器总是可以用来跟踪源代码中的问题的错误，内核故障即使不能杀死整个系统，至少会杀死当前进程。我们在第4章看到如何跟踪内核错误。

### 2.3.1 用户空间和内核空间

一个模块在内核空间运行，而应用程序在用户空间运行。这个概念是操作系统理论的基础。

实际上，操作系统的作用是提供与计算机硬件一致的程序。 另外，操作系统必须考虑程序的独立操作并防止未经授权访问资源。只有当CPU强制从应用程序中保护系统软件时，才可能执行这项重要的任务。

每个现代处理器都能够执行这种行为。所选择的方法是在CPU本身中实现不同的操作模式（或等级）。各等级具有不同的角色，并且一些操作在较低级别上被禁止；程序代码只能通过有限数量的通道来从一个层切换到另一个层。Unix系统旨在利用这种硬件功能，使用两个这样的等级。目前所有的处理器至少有2个保护级别，而一些像x86系列的处理器有更多的级别;当存在多个级别时，使用最高级别和最低级别。在Unix下，内核以最高级别（也称为超级用户模式）执行，其中允许所有内容，而应用程序以最低级别（所谓的用户模式）执行，其中处理器管理对硬件的直接访问和对内存的非授权访问。

我们通常将执行模式称为内核空间和用户空间。这些术语不仅包含两种模式固有的不同特权级别，还包含每种模式都可以有自己的内存映射-它自己的地址空间的事实。

每当应用程序发出系统调用或被硬件中断挂起时，Unix都会将执行从用户空间转移到内核空间。执行系统调用的内核代码在进程的上下文中工作，它代表调用进程运行，并能够访问进程地址空间中的数据。另一方面，处理中断的代码与进程不同步，并且与任何特定进程无关。

模块的作用是扩展内核功能；模块化代码在内核空间中运行。通常，驱动程序会执行前面概述的任务：模块中的某些功能是作为系统调用被执行，而有些功能负责中断处理。

### 2.3.2 内核中的并发处理

内核编程与传统应用程序编程大不相同的一种方法就是并发性问题的处理。大多数应用程序（显然，多线程应用程序除外）一般来说，从头到尾都是顺序执行，无需担心可能会发生什么情况改变其执行环境。内核代码不能假想这样简单的环境中运行，即使最简单的模块代码也必须考虑多种情况可能会发生。

内核编程中有几个并发的来源。当然，Linux系统运行多个进程，其中同时不止一个可以尝试使用你的驱动程序。大多数设备都能够中断处理器；中断处理程序异步运行，并可在驱动程序尝试执行其它操作的同时调用。一些软件抽象（如第7章介绍的内核定时器）也是异步运行的。此外，Linux也可以在对称多处理器（SMP）系统上运行，结果就是驱动程序可以在多个CPU上同时执行。最后，在Linux 2.6内核代码是可抢占式的；这种改变甚至会导致单处理器系统与多处理器系统有许多相同的并发问题。

因此，Linux内核代码（包括驱动程序代码）必须是可重入的 - 它必须能够同时在多个环境中运行。必须仔细设计数据结构以保证多个执行线程是分离的，并且代码必须小心访问共享数据，防止数据损坏。编写处理并发性和避免竞争条件的代码（不确定的执行顺序可能导致不想要的行为况）需要思考并且可能非常棘手。正确的并发管理需要编写正确的内核代码；出于这个原因，本书中的每个示例驱动程序都是以并发的思想编写的。所使用的技术在我们遇到时会解释；第5章也专门讨论了这个问题以及可用于并发管理的内核原语。

驱动程序员犯的一个常见错误是假设只要某一代码段没有进入休眠状态（或“阻塞”状态），并发就不是问题。即使在以前的内核中（非抢先式的），这种假设在多处理器系统上是无效的。在2.6内核代码中（几乎）不会假定它可以给某一固定的代码段保持处理器。如果没有并发的思维编写内核相关代码，将会面临极其难以调试的灾难性故障。

### 2.3.3 当前进程

尽管内核模块并不像应用程序那样按顺序执行，但内核执行的大多数操作都是代表特定进程完成的。内核代码可以通过访问<asm/current.h>中定义的全局量current来引用当前进程，这会产生一个指向struct task\_struct 的指针，在<linux/sched.h>中定义。current指针指的是当前正在执行的进程。在执行系统调用期间，例如open或read，当前进程是调用该系统调用的进程。如果需要的话，内核代码可以使用current访问具体进程的信息。第6章介绍了使用这种技术的一个例子。

实际上，current并不是真正的全局变量。支持SMP系统的需求迫使内核开发人员开发一种在相关的CPU上查找当前进程的机制。这种机制必须很快，因为对current的引用经常发生。其结果就是依赖于特定架构的机制，在内核的堆栈上隐藏一个结构体task\_struct的指针。实现的细节对其它内核子系统仍然保持隐藏，设备驱动程序只需包含头文件<linux/sched.h>就可以引用current进程。例如，下面的语句通过访问struct task\_struct中的某些字段来打印当前进程的进程标识和命令名称：

|  |
| --- |
| printk(KERN\_INFO "The process is \"%s\" (pid %i)\n", current->comm, current->pid); |

存储在current->comm中的命令名称是当前进程正在执行的程序文件的基本名称（如果需要，可以修剪为15个字符）。

### 2.3.4 其它细节

内核编程在很多方面与用户空间编程不同。在本书的过程中，随着逐渐接近它们，我们会尽量的指出它们，但是有一些基本问题值得一提，尽管这些问题不属于它们的一部分。因此，在深入研究内核时，应牢记以下问题。

应用程序运行在虚拟内存，具有很大的堆栈空间。堆栈用于保存函数调用历史记录和由当前活动函数创建的所有自动变量。相反，内核有一个非常小的堆栈；它小到只有4096个字节。所以，你的函数必须与整个内核空间调用链共享该堆栈。因此，声明大的自动变量绝不是一个好主意；如果你需要更大的结构，应该在调用时动态分配它们。

通常，当您查看内核API时，您将遇到以双下划线（\_\_）开头的函数名称。如此标记的功能通常是较低层面的接口函数，应谨慎使用。本质上，就好像，双下划线对程序员说：“如果你调用这个函数，确保知道你在做什么。”

内核代码不能做浮点运算。启用浮点需要要求内核保存并恢复浮点处理器的每个入口的状态，并退出内核空间-至少在某些体系结构上是这样的。鉴于内核代码中实际上不需要浮点数，额外的开销并不值得。

## 2.4 编译和加载

本章开头的“hello world”示例包括构建模块并将其加载到系统中的简要演示。当然了，编译和加载模块的过程比我们之前的示例要复杂的多。本节详细介绍了模块作者如何将源代码转换为内核中可执行的代码。

### 2.4.1编译模块

首先，我们需要了解模块的构建方式。模块的构建过程与用户空间应用程序的构建过程明显不同；内核是一个庞大的独立程序，对于如何将它们放在一起，有着详细而明确的要求。构建过程也不同于之前版本的内核所做的工作；新的构建系统使用起来更简单，并产生更多正确的结果，但它与以前看起来非常不同。内核构建系统是一个复杂的野兽，我们只是看它的一小块。对于想要深入了解内核构建系统的人，可以查看内核源代码中Documentation/kbuild目录下的文件。

在构建内核模块之前，有一些前提条件必须解决。首先是确保你有足够的当前版本的编译器，模块实用程序和其它必要的工具。文件Documentation/Changes文件总是会列出所需的工具版本；在继续之前，你应该查看它。试图用错误的工具版本构建内核（及其模块）可能会导致无法解决的棘手问题。请注意，有时候，太新的编译器版本可能与过时的版本一样有问题；内核源代码对编译器做出了很多假设，新版本有时可能会不合适。

如果手头上还没有一个内核源码树，又或者仍然没有配置和构建内核，那么是时候去构建一个了。如果在你的主机上没有Linux2.6内核的源码树你不可能构建可加载的模块。实际运行正在构建的内核也是很有帮助的。

一切准备就绪，为你的模块编写一个Makefile就是非常简单的事情了。事实上，之前展示的“hello world”的示例，单行代码就足够了：

|  |
| --- |
| obj-m := hello.o |

熟悉make，但是不熟悉Linux 2.6内核构建系统的读者，很可能想知道这个Makefile是怎样工作的。毕竟，上面的代码看起来不像我们常见的Makefile那样。当然了，答案就是构建系统替你完成了剩余的工作。上面的赋值语句（使用了GNU make提供的扩展语法）表明，将会有一个模块使用目标文件hello.o构建。构建后，生成的模块名为hello.ko。

如果有两个源文件（称为file1.c和file2.c），生成module.ko的模块，那正确的写法是：

|  |
| --- |
| obj-m := module.o  module-objs := file1.o file2.o |

上面列出的这些Makefile文件，要想工作，必须放在构建系统里。比如说，你的内核源代码位于~/kernel-2.6目录下，构建你的模块所需要的make命令就是：

|  |
| --- |
| make -C ~/kernel-2.6 M=`pwd` modules |

使用-C选项，指定你的内核源代码目录；在那里它会发现内核的顶层Makefile。M选项会在尝试构建modules目标之前移回到模块源码目录中。这个modules目标指的是obj-m变量中指定的模块列表，在我们的例子中，我们已经设置为module.o。

这样的make命令会使得在内核源代码树之外构建模块变得更为容易。编写Makefile的诀窍如下：

|  |
| --- |
| # 如果KERNELRELEASE被定义，我们将会被kernel构建系统调用，使用它的语言  ifneq ($(KERNELRELEASE),)  obj-m := hello.o  # 否则，我们直接被从命令行调用；调用内核构建系统  else  KERNELDIR ?= /lib/modules/$(shell uname -r)/build  PWD := $(shell pwd)  default:  $(MAKE) -C $(KERNELDIR) M=$(PWD) modules  endif |

我们再一次看到了GNU make的扩展语法。这个Makefile在典型的构建中被读取两次。当从命令行调用Makefile时，它注意到KERNELRELEASE变量尚未设置。它通过利用已安装模块目录中的符号链接指向内核构建树的事实来定位内核源代码目录。如果您实际上运行的不是正在构建的内核，则可以在命令行上提供KERNELDIR =选项，设置KERNELDIR环境变量，或者重写在Makefile中设置KERNELDIR的行。 一旦找到内核源代码树，Makefile会调用default：目标，它将运行第2个make命令（在Makefile中参数化为$（MAKE））来调用内核构建系统，如前所述。在第2次读取时，Makefile设置obj-m，内核Makefile负责实际构建模块。

这种构建模块的机制可能会让你感到有点笨拙和模糊。然而，一旦你习惯了它，你可能会喜欢已经编入内核编译系统的功能。请注意，以上不是完整的Makefile；一个真正的Makefile包含了用于清理不需要的文件，安装模块等常用的目标。查看源代码目录中的makefile以获取完整示例。

### 2.4.2 加载和卸载模块

### 2.4.3 版本依赖

### 2.4.4 平台依赖

## 2.5 内核符号表

2.6 预备知识

2.7 初始化和关机

2.7.1 清理函数

2.7.2 初始化中的错误

2.7.3 模块加载竞争

2.8 模块参数

2.9 在用户空间做

2.10 快速参考

# 第14章 Linux设备模型

2.5开发周期的目标之一是为内核创建统一的设备模型。以前的内核没有单一的数据结构，用来获取有关系统如何把设备放在一起的信息。尽管缺乏这方面的信息，但一段时间以来，内核仍然运行良好。但是，具有更复杂拓扑结构，并需要支持电源管理等功能的新系统，为了使其更清晰简洁，就需要一种系统结构的抽象描述。

2.6内核设备模型提供了这种抽象描述。在内核中，被用来支持各种任务，如下：

* 电源管理和系统关机  
  这需要了解系统的结构。例如，在处理连接到USB主机适配器的所有设备之前，适配器无法关闭。设备模型确保能够以正确的顺序遍历系统的硬件。
* 和用户空间进行通信  
  sysfs虚拟文件系统的实现与设备模型紧密相关，并揭示了它所代表的结构。将系统信息提供给用户空间和用于改变操作参数的“旋钮”越来越多地通过sysfs完成，因此也是通过设备模型完成。
* 可热插拔设备  
  计算机硬件越来越多的是动态设备，用户可以随意插拔外设。内核里用来处理热插拔设备，尤其是和其通信的hotplug机制，是通过设备模型管理的。
* 设备类  
  系统的许多部分对如何连接设备几乎没有兴趣，但他们需要知道哪些设备可用。设备模型包括一个用于将设备分配给类的机制，这些类可以在更高的功能级别上描述这些设备，并允许从用户空间中发现它们。
* 对象生命周期  
  上面描述的许多功能，包括热插拔支持和sysfs，都会使内核中对象的创建和操作复杂化。设备模型的实现需要创建一组处理对象生命周期的机制，它们之间的关系以及它们在用户空间中的表示。

Linux设备模型是一个复杂的数据结构。例如，如图14-1所示，该图简化显示了有关USB鼠标的设备模型结构的一小部分。沿着图的中心，可以看到，“设备”树展示了鼠标是怎样连接到系统的。“总线”树跟踪每条总线的连接情况，而“类”的子树则关心设备提供的功能，而不管他们是如何连接的。即使在一个简单的系统，设备树模型也包含上百个上述这样的节点，所以这是一个非常难以整体可视化的数据结构。

图14-1，设备模型的一小部分

对于大多数情况下，Linux设备模型考虑了所有应该考虑的事情，而不是把这些交给驱动程序的开发者去完成。它在后台运行，与设备模型的直接交互也是通过总线或者其它内核子模块完成。结果，许多驱动程序的开发者能够忽略设备模型，并相信它会自行处理。

但是，对设备模型的理解总是一件好事情的。有时候设备模型会从其它层“泄漏出去”；例如，使用struct device的通用DMA代码（我们在第15章中遇到）。您可能需要使用设备模型提供的某些功能，例如kobjects提供的引用计数和相关功能。通过sysfs与用户空间进行通信也是一种设备模型功能；本章解释了该通讯的工作原理。

我们对设备模型实行自下而上演示。设备模型的复杂性使得从顶层视图理解变得非常困难。我们希望，通过展示底层设备组件的工作方式，我们可以帮助您准备应对这些组件如何用于构建更大结构的挑战。

对于很多读者来说，本章可以被看作是第一次不需要阅读的高级资料。但是，我们鼓励那些对Linux设备模型如何工作感兴趣的同学们，随我们一起进入更低层次的细节。

## 14.1 Kobjects，Ksets和Subsystems

kobject是将设备模型整合在一起的基本结构。它最初被设想为一个简单的引用计数器，但它的职责随着时间的推移而增长，其内容也是。现在由struct kobject及其支持代码处理的任务包括：

* 对象的引用计数  
  通常，在创建内核对象时，无法知道它将存在多长时间。跟踪这些对象生命周期的一种方法是通过引用计数。当内核中没有代码持有给定对象的引用时，该对象已经完成了它的使用寿命并且可以被删除。
* Sysfs表示  
  每个在sysfs中显示的对象都有一个kobject，它与内核交互以创建其可见的表示形式。
* 数据结构胶水  
  整个设备模型是一个非常复杂的数据结构，由多个层次组成，并且它们之间有很多链接。 kobject实现了这个结构并且把它保存在一起。
* 热插拔事件处理  
  kobject子系统负责有关系统硬件插拔时，通知用户空间的事件的产生。

从前面的内容中可以看出：kobject是一个复杂的结构。但是，我们可以通过每次一小块的方式，尽可能地理解这种数据结构和它的工作原理。

### 14.1.1 Kobject基础知识

一个kobject对象的类型是struct kobject；它在<linux/kobject.h>中定义。该文件还包含许多与kobjects相关的其它结构的声明，当然还包括一系列用于操作它们的函数。

**1. 嵌入式内核对象Kobject**

在详细讨论细节之前，花点时间了解如何使用Kobjects。如果回顾一下由kobjects处理的函数列表，您会发现它们都是代表其他对象执行的服务。换句话说，kobject对象对它本身并不感兴趣。它只存在于将更高级别的对象绑定到设备模型中。

因此，内核代码很少（甚至不知道）创建独立的kobject;相反，kobject对象被用来控制对更大的，特定领域对象的访问。为此，kobject对象被发现总是嵌入在其它结构中。如果您习惯于以面向对象的方式来思考事物，那么kobjects可以被看作是其它类派生的顶级抽象类。 kobject实现了一组本身并不特别有用的功能，但在其它对象中很有用。C语言不允许直接表示继承，所以必须使用其它技术 - 例如将一个结构嵌入另一个结构中。

作为一个例子，让我们回头看看在第3章中遇到的struct cdev。

这个结构，在2.6.10内核中，是这样的：