# 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。

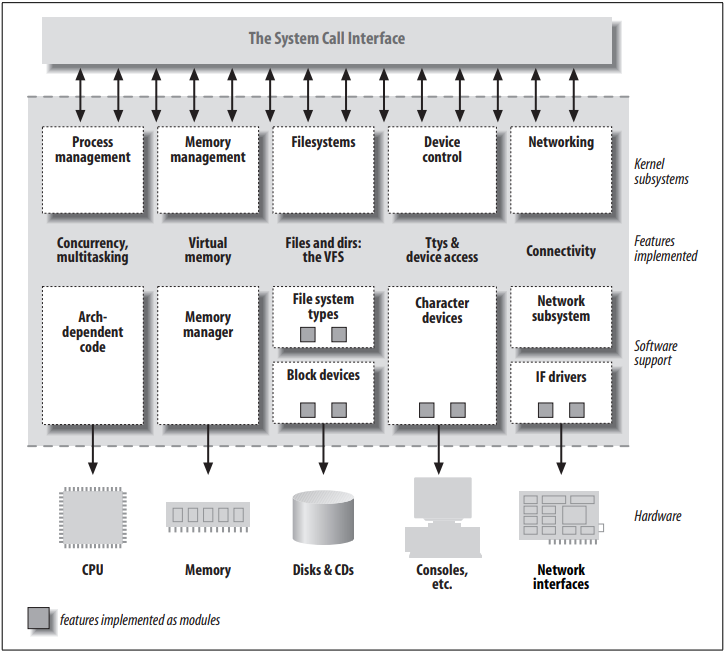


图1-1 简要内核框图

* 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

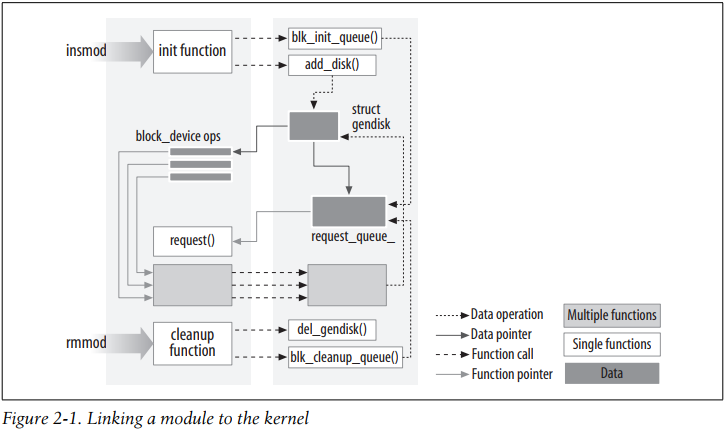
在我们继续深入之前，必须强调一下内核模块和应用程序之间的各种差异。

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

请记住，模块代码必须为每个它所连接的内核版本重新编译。模块与特定内核版本中定义的数据结构和函数原型紧密相关；模块所使用的内核提供的接口可能会因为内核的不同而有着显著变化。这对开发内核尤为关键。

内核不仅假定给定的模块是根据根据正确的内核版本构建。编译过程的一个步骤就是，将编译的模块与内核源码树中的文件（vermagic.o）相链接。这个文件包含编译模块所基于的Linux内核版本的信息，包括目标内核版本，编译器版本，许多重要的配置变量的设置。当尝试载入一个模块时，就会验证这些信息是否与正在运行的内核相同。如果发生不匹配，内核不会载入；相反，你还能看到下面的错误信息：

|  |
| --- |
| # insmod hello.ko  Error inserting './hello.ko': -1 Invalid module format |

查看系统log文件（/var/log/messages或者你的系统中其它类似的配置文件）能够看到造成模块加载失败的原因。

如果您需要编译特定内核版本的模块，则需要使用该特定版本的编译系统和源代码树。之前的示例Makefile中的KERNELDIR变量的就是很好的一个方法。

If you are writing a module that is intended to work with multiple versions of the kernel (especially if it must work across major releases), you likely have to make use of macros and #ifdef constructs to make your code build properly. This edition of this book only concerns itself with one major version of the kernel, so you do not often see version tests in our example code. But the need for them does occasionally arise. In such cases, you want to make use of the definitions found in linux/version.h. This header file, automatically included by linux/module.h, defines the following macros:

内核接口经常伴随着版本发生变化。如果您正在编写一个旨在与多个内核版本一起工作的模块（特别是如果它必须在主要版本中工作），则可能必须使用宏和#ifdef结构来正确构建代码。本书的这个版本只关注内核的一个主要版本，所以你不会在我们的示例代码中看到版本测试。但是他们的需求偶尔会出现。在这种情况下，您想要使用linux/version.h中的定义。这个头文件自动包含了linux/module.h，且定义了以下宏：

### 2.4.3 版本依赖

UTS\_RELEASE

描述内核版本的字符串

LINUX\_VERSION\_CODE

内核版本的二进制表示，版本号的每一个部分都有一个字节，例如，2.6.10的代码是132618（即0x02060a）

KERNEL\_VERSION(major,minor,release)

这是用于从构建版本号的单个数字构建整数版本代码的宏。 例如，KERNEL\_VERSION（2,6,10）扩展到132618.当需要比较当前版本和已知检查点时，此宏非常有用。

使用KERNEL\_VERSION和LINUX\_VERSION\_CODE，大多数基于内核版本的依赖可以通过预处理条件来解决。但是版本依赖性不应该使用#ifdef条件变量和驱动代码混在一起。最好的方式就是把这些不兼容的东西单独放到一个头文件里。常用规则就是，明显的版本依赖隐藏在低层次的宏和函数里。高层的代码只需调用，而无需关心底层细节。这样的代码才会更易读更健壮。

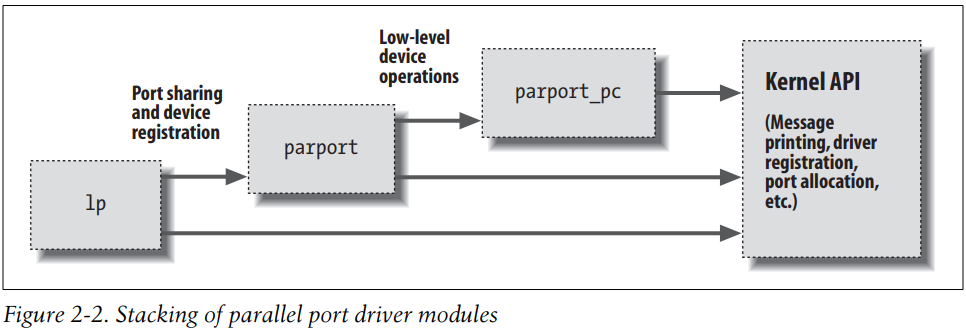
### 2.4.4 平台依赖

## 2.5 内核符号表

我们已经看了insmod命令是如何根据公共内核符号表解析未定义符号。该表包含全局内核函数和变量的地址，这些都是实现模块化驱动程序所需要的。当一个模块被加载时，模块中导出的符号就成为了内核符号表的一部分。在正常情况下，模块可以实现自己的功能，而不要任何导出任何符号。但是如果别的模块想要使用它就需要导出这些符号。

新模块可以使用你的模块导出的符号，你也可以在其它模块之上堆叠新的模块。模块的堆叠也可以在主流内核源代码中实现：msdos文件系统依赖于fat模块导出的符号，每个输入USB设备模块都是建立在usbcore和input模块基础之上。

模块堆叠在复杂的项目中非常有用。如果一个新的抽象是以设备驱动程序的形式实现的，它可能会为硬件特定的实现提供一个插件。例如，video-for-linux驱动程序集被分成一个通用模块，该模块导出用于特定硬件的底层设备驱动程序使用的符号。根据您的设置，您将加载通用video模块和已经安装的硬件具体模块。并行端口支持和各种可连接设备的处理方式与USB内核子系统相同。在并行端口子系统中结构图如图2-2所示; 箭头表示模块和内核编程接口之间的通信。



当使用有依赖的模块时，modprobe命令是很有帮助的。如前面所示，modprobe与insmod在功能上非常相似，但它也会加载模块所需的任何其它模块。

使用模块堆叠将模块拆分为多个层可以通过简化每个层来缩短开发时间。这与我们在第1章中讨论过的机制和策略的分离思想相似。

Linux内核头文件提供了一种方便的方式来管理符号的可见性，从而减少命名空间污染（有可能会和内核命名空间中已有的名称冲突），并促进正确的信息隐藏。如果模块需要导出符号供其它模块使用，则应使用以下宏。

|  |
| --- |
| EXPORT\_SYMBOL(name);  EXPORT\_SYMBOL\_GPL(name); |

上面两个宏都可以使符号在模块外部可用。\_GPL版本只允许符合GPL许可的模块使用符号。这些符号必须被导出到该模块文件的全局部分，任何函数之外，因为这些宏是被期望用来全局访问的一些特殊变量。这些变量被存储在模块可执行文件（“ELF”段）的特定部分，被内核用在加载的时候查找模块导出的符号。（感兴趣的读者可以查看<linux/module.h>了解详细信息）

## 2.6 预备知识

虽然我们越来越接近真实的模块代码，但是，在此之前，我们还需要看一些其它的东西，这些东西也是内核模块代码中需要的。内核是一个独一无二的环境，所以它也对接口代码做了强制性的要求。

大部分的内核代码包含大量的头文件，来获取函数的定义，数据类型和变量。但是也有一些对于模块来说是特殊的，是必须出现在可加载模块中的。几乎所有的模块代码都具有以下内容：

|  |
| --- |
| #include <linux/module.h>  #include <linux/init.h> |

头文件module.h包含了许多可加载模块所需要的符号和函数的定义。头文件init.h是定义模块的初始化和退出函数时必须的。还有许多模块包含头文件<moduleparam.h>使能在加载的时候传递参数给模块的功能。我们很快就会看到。

虽然不是必须的，但是你还是尽量指明模块遵循的许可，使用宏MODULE\_LICENSE实现：

|  |
| --- |
| MODULE\_LICENSE("GPL"); |

内核认可的具体许可协议是“GPL”（适用于任何版本的GNU通用公共许可证），“GPL v2”，“GPL和附加权利”，“Dual BSD/GPL”，“ Dual MPL/GPL”和“私有”。如果模块没有被标记为内核识别的自由协议，那么就会被认定为是私有的。当加载模块时，内核被污染。而且正如在第1章“许可条款”中提到的，内核开发人员不会热情地帮助私有模块解决遇到的问题。

其它的还有：

MODULE\_AUTHOR-模块作者

MODULE\_DESCRIPTION-模块描述

MODULE\_VERSION-模块版本

MODULE\_ALIAS-模块别名-第11章

MODULE\_DEVICE\_TABLE-告诉用户空间模块支持哪些设备（第12章）

上述这些以MODULE\_开头的宏可以出现在源代码中的任何位置（函数外）。作为约定，一般放在文件的最后。

## 2.7 初始化和关机

如前所述，模块初始化功能会注册模块提供的任何设施。 通过这些设施，新功能，无论是整个驱动程序还是新的软件抽象，都可以被应用程序访问。初始化函数的实际定义总是如下所示：

|  |
| --- |
| static int \_\_init initialization\_function(void)  {  /\* 初始化代码 \*/  }  module\_init(initialization\_function); |

初始化函数应该声明为static，这是因为一般情况下，不需要对外可见，除非有特殊要求。定义中\_\_init声明是用来告知内核，该函数只有在初始化的时候使用。模块加载程序在模块加载后会删除初始化函数，使其内存可用于其它用途。还有一个类似的数据标签(\_\_initdata)。\_\_init和\_\_initdata是可选的，虽然有点麻烦但是很有用处。只要确保不要在初始化完成后，再作用于函数或者数据就可以了。你还有可能在内核源代码中遇到\_\_devinit和\_\_devinitdata；只要内核没有被配置为热插拔设备就会被翻译为\_\_init和\_\_initdata。在第14章我们再详细讨论。

module\_init是强制性的。没有这个定义，你的初始化函数永远不会被调用。

Modules can register many different types of facilities, including different kinds of devices, filesystems, cryptographic transforms, and more. For each facility, there is a specific kernel function that accomplishes this registration. The arguments passed to the kernel registration functions are usually pointers to data structures describing the new facility and the name of the facility being registered. The data structure usually contains pointers to module functions, which is how functions in the module body get called.

模块可以注册许多不同类型的设施，包括不同类型的设备，文件系统，密码转换等等。对于每个设施，都有一个特定的内核函数来完成这个注册。传递给内核注册函数的参数通常是指向描述新设施和被注册设施名称的数据结构的指针。数据结构通常包含指向模块函数的指针，这就是模块体中函数的调用方式。

The items that can be registered go beyond the list of device types mentioned in Chapter 1. They include, among others, serial ports, miscellaneous devices, sysfs entries, /proc files, executable domains, and line disciplines. Many of those registrable items support functions that aren’t directly related to hardware but remain in the “software abstractions” field. Those items can be registered, because they are integrated into the driver’s functionality anyway (like /proc files and line disciplines for example).

可以注册的项目超出了第1章提到的设备类型列表。其中包括串行端口，其他设备，sysfs条目，/ proc文件，可执行域和行规范。许多可注册的项目支持与硬件不直接相关的功能，但仍保留在“软件抽象”字段中。这些项目可以被注册，因为它们无论如何都集成到了驱动程序的功能中（例如/ proc文件和行规范）。

### 2.7.1 清理函数

每个正式的模块都要求一个清理函数，它的作用是，在模块移除之前，取消注册并返回所有使用的资源。这个函数的定义如下：

|  |
| --- |
| static void \_\_exit cleanup\_function(void)  {  /\*代码 \*/  }  module\_exit(cleanup\_function); |

不需要返回值，所以定义为void类型。修饰符\_\_exit标记代码只为卸载时使用(让编译器将其放入特定的ELF段中)。如果模块直接编译进内核，又或者内核被配置为不允许卸载模块，那么\_\_exit标记的函数就会被放弃。基于这个原因，\_\_exit标记的函数只能在模块卸载或系统关机时使用，其它的使用都是错误的。再有，\_\_exit声明就是确保内核能够找到你的cleanup函数。

如果模块没有定义cleanup函数，内核不允许卸载该模块。

### 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3章 字符驱动

本章的目的就是编写一个完整的字符设备驱动程序。我们选择字符设备驱动程序，是因为大多数简单的硬件设备都属于这一类。字符设备驱动程序相比块设备和网络设备要更容易理解（后面的章节我们再讨论其它的设备）。最终我们会编写一个模块化的字符驱动程序，但是并没有讨论模块性的问题。

纵观本章，我们从实际的设备抽象出了这段代码：scull（简单的字符设备工具）。Scull是一个作用于内存区域上的字符设备驱动程序。我们可以这样理解，scull所使用的内存区域就是一个设备。

Scull的优点就是不依赖硬件，它只需要占用内核分配的一些内存即可。任何人都可以编译、运行在linux系统之上。除了演示内核和字符驱动程序之间的接口并允许用户做一些测试之外，该设备不会执行任何其它有用的操作。

## 3.1 scull设计

驱动程序编写的第一步是定义驱动程序为用户程序提供的功能（机制）。由于我们的“设备”是计算机内存的一部分，因此我们可以自由地按照自己的意愿去做。它可以是顺序或随机访问设备，也可以是一个或多个设备，等等。

为了让scull成为编写真实设备真实驱动程序的模板，我们将向您展示如何在计算机内存上实现多个设备抽象，每个设备抽象具有不同的个性。

scull源实现以下设备。 模块实现的每种设备称为一种类型。

* scull0到scull3  
  四个设备，每个设备都包含一个既全局又持久的内存区域。全局意味着如果设备被多次打开，设备中包含的数据将被打开它的所有文件描述符共享。持久意味着如果设备关闭并重新打开，数据不会丢失。此设备可以很有趣，因为它可以使用常规命令（如cp，cat和shell I/O重定向）进行访问和测试。
* scullpipe0到scullpipe3  
  四个FIFO（先进先出）设备，就像管道一样。一个进程读取另一个进程写入的内容。如果多个进程读取相同的设备，会产生数据竞争。scullpipe的内部将展示如何实现阻塞和非阻塞读写，而不必求助于中断。 虽然真正的驱动程序使用硬件中断与其设备同步，但是阻塞和非阻塞操作的主题非常重要，与中断处理（第10章中介绍）是分开的。
* scullsingle  
  scullpriv  
  sculluid  
  scullwuid  
  这些设备与scull0类似，但在打开时有一些限制。第一个（scullsingle）每次只允许一个进程使用驱动程序，而scullpriv对于每个虚拟控制台（或X终端会话）是私有的，因为每个控制台/终端上的进程获得不同的内存区域。sculluid和scullwuid可以多次打开，但一次只能由一个用户打开;如果另一个用户正在锁定设备，则前者返回“设备忙”的错误，而后者实现阻止打开。这些变种的scull似乎是混淆策略和机制，但值得关注，因为一些现实生活中的设备需要这种管理。

每个scull设备都展示了驱动程序的不同特征，表现出了不同的难度。本章涵盖了scull0到scull3的内部结构；关于更高级的设备在第6章中介绍。有关scullpipe的信息，请参见“一个阻塞I/O的例子”一节，其它的在“设备文件的访问控制”中进行了介绍。

## 3.2 主次设备号

字符设备通过文件系统中的名称进行访问。这些名称被称为特殊文件或设备文件或简称为文件系统树的节点;它们通常位于/dev目录中。字符驱动程序的特殊文件由ls -l输出的第一列中的“c”标识。块设备也出现在/dev中，但是它们由“b”标识。本章的焦点在字符设备上，但以下大多数信息也适用于块设备。

如果您发出ls -l命令，则会在上次修改日期之前的设备文件条目中看到两个数字（用逗号分隔），其中文件长度通常出现在该处。这些数字是特定设备的主要和次要设备号。以下列表显示了一些典型系统上显示的设备。它们的主要设备号是1,4,7和10，而次要设备号是1,3,5,64,65和129。

|  |
| --- |
| crw-rw-rw- 1 root root 1, 3 Apr 11 2002 null  crw------- 1 root root 10, 1 Apr 11 2002 psaux  crw------- 1 root root 4, 1 Oct 28 03:04 tty1  crw-rw-rw- 1 root tty 4, 64 Apr 11 2002 ttys0  crw-rw---- 1 root uucp 4, 65 Apr 11 2002 ttyS1  crw--w---- 1 vcsa tty 7, 1 Apr 11 2002 vcs1  crw--w---- 1 vcsa tty 7, 129 Apr 11 2002 vcsa1  crw-rw-rw- 1 root root 1, 5 Apr 11 2002 zero |

传统上，主设备号标识与设备关联的驱动程序。例如，/dev/null和/dev/zero都由驱动程序1管理，而虚拟控制台和串行终端由驱动程序4管理；类似地，vcs1和vcsa1设备都由驱动程序7管理。现代Linux内核允许多个驱动程序共享主设备号，但您将看到的大多数设备仍然按照一主一驱动程序原则组织。

内核使用次设备号来确定究竟哪个设备被引用。根据驱动程序的写法（如下面所示），您可以从内核获取指向设备的直接指针，也可以使用次要编号作为本地设备阵列的索引。无论哪种方式，内核本身除了知道它们引用的是驱动程序实现的设备之外，对其它一无所知。

### 3.2.1 设备号的内部表示

在内核中，dev\_t类型（在<linux/types.h>中定义）用于保存设备号-主要和次要部分。从内核版本2.6.0开始，dev\_t是一个32位数，主设备号为12位，次设备号为20。当然，您的代码不应该对设备编号的内部组织做任何假设；它应该使用<linux/kdev\_t.h>中的一组宏。要获取dev\_t的主要或次要部分，请使用：

|  |
| --- |
| MAJOR(dev\_t dev);  MINOR(dev\_t dev); |

相反，如果您有主要和次要设备号并需要将它们变成dev\_t，请使用：

|  |
| --- |
| MKDEV(int major, int minor); |

请注意，2.6内核可容纳大量设备，而以前的内核版本仅限于255个主要数字和255个次要数字。人们假设在一段时间内这样的范围已经足够了，但是计算领域充斥着对这种本质的错误假设。所以你应该期待dev\_t的格式在未来可能会再次发生变化；但是，如果您仔细编写驱动程序，这些更改不会成为问题。

### 3.2.2 分配和释放设备号

在注册字符设备时，驱动程序需要做的第一件事就是获取一个或多个要使用的设备编号。这项任务所需要的函数是register\_chrdev\_region，它在<linux/fs.h>中声明：

|  |
| --- |
| int register\_chrdev\_region(dev\_t first, unsigned int count, char \*name); |

这里，first是想要分配的范围的起始设备号。first的次要设备号通常是0，但没有要求。 count是请求的连续设备号的总数。请注意，如果count很大，您请求的范围可能会溢出到下一个主要设备号；但只要请求的号码范围可用，所有内容仍将正常工作。最后，name是应该与此号码范围关联的设备的名称；它会出现在/proc/ devices和sysfs中。

与大多数内核函数一样，如果分配成功执行，register\_chrdev\_region的返回值将为0。 如果发生错误，将返回一个负数的错误代码，并且您将无法访问请求的区域。

如果您事先知道所需的设备号，register\_chrdev\_region可以很好地工作。但是，通常情况下，您不知道设备将使用哪些主设备号；Linux内核开发社区一直在努力转向使用动态分配的设备编号。内核会快速地为设备分配一个主设备号，但必须使用不同的函数来请求：

|  |
| --- |
| int alloc\_chrdev\_region(dev\_t \*dev, unsigned int firstminor, unsigned int count, char \*name); |

使用此函数，dev是一个只输出的参数，在成功完成时，将保留分配范围中的第一个数字。 firstminor应该是请求的第一个次设备号；它通常是0。count和name参数与request\_chrdev\_region的参数一样。

无论您如何分配设备号，不再使用时都应该在释放它们。设备号释放函数：

|  |
| --- |
| void unregister\_chrdev\_region(dev\_t first, unsigned int count); |

调用unregister\_chrdev\_region的通常位置是在模块的清理函数中。

上述函数为驱动程序分配设备号，但它们不会告诉内核使用这些设备号做什么。在用户空间程序访问其中一个设备号之前，驱动程序需要将它们连接到实现设备操作的内部函数。我们将尽快描述这种连接是如何完成的，但首先需要处理几个其它的问题。

### 3.2.3 主设备号的动态分配

一些主设备号被静态地分配给最常用的设备了。这些设备的列表可以在内核源码树中的Documentation/devices.txt中找到。但是，已经分配过的静态设备号再给新驱动程序使用的可能性很小。所以，作为一名驱动开发者，你有一个选择：可以简单地选择一个似乎未被使用的设备号，或者可以动态地分配主设备号。只要驱动程序的唯一用户是自己，选择一个号码就可以工作；一旦驱动程序被广泛地部署，随机挑选的主设备号将导致冲突和麻烦。

因此，对于新的驱动程序，我们强烈建议使用动态分配来获取主设备号，而不是随机选择一个号。换句话说，驱动程序尽量使用alloc\_chrdev\_region而不是register\_chrdev\_region。

动态分配的缺点是无法预先创建设备节点，因为分配给模块的主设备号会有所不同。对于驱动程序的正常使用，这不是一个问题，因为一旦分配了编号，就可以从/proc/devices中读取它。

因此，使用动态主设备号加载驱动程序，insmod的调用可以用一个简单的脚本替换，该脚本在调用insmod后读取/proc/devices以创建设备文件。

典型的/proc/devices文件如下所示：

|  |
| --- |
| Character devices:  1 mem  2 pty  3 ttyp  4 ttyS  6 lp  7 vcs  10 misc  13 input  14 sound  21 sg  180 usb  Block devices:  2 fd  8 sd  11 sr  65 sd  66 sd |

因此，加载已分配动态设备号的模块的脚本可以使用awk等工具编写，以便从/proc/devices中检索信息以便在/dev中创建文件。

下面的脚本scull\_load是scull的一部分。以模块形式发布驱动程序的用户可以从系统的rc.local文件中调用此脚本，或者在需要模块时手动调用它。

|  |
| --- |
| #!/bin/sh  module="scull"  device="scull"  mode="664"  # invoke insmod with all arguments we got  # and use a pathname, as newer modutils don't look in . by default  /sbin/insmod ./$module.ko $\* || exit 1  # remove stale nodes  rm -f /dev/${device}[0-3]  major=$(awk "\\$2= =\"$module\" {print \\$1}" /proc/devices)  mknod /dev/${device}0 c $major 0  mknod /dev/${device}1 c $major 1  mknod /dev/${device}2 c $major 2  mknod /dev/${device}3 c $major 3  # give appropriate group/permissions, and change the group.  # Not all distributions have staff, some have "wheel" instead.  group="staff"  grep -q '^staff:' /etc/group || group="wheel"  chgrp $group /dev/${device}[0-3]  chmod $mode /dev/${device}[0-3] |

该脚本可以通过重新定义变量并调整mknod行，就可以应用于其它驱动程序。 上面的脚本创建了四个设备，因为默认就是4个scull设备。

脚本的最后几行可能看起来有些难以理解：为什么改变设备的组和模式？原因是脚本必须由超级用户运行，所以新创建的设备文件由root拥有。权限位默认为只有root拥有写权限，任何人都有读权限。通常，设备节点需要不同的访问策略，因此必须以某种方式更改访问权限。脚本中的默认设置是为一组用户提供访问权限，但您的需求可能会有所不同。在第6章“设备文件的访问控制”一节中，sculluid的代码演示了驱动程序如何执行自己的设备访问授权。

scull\_unload脚本也可用于清理/ dev目录并删除模块。

为了替换上面加载和卸载的脚本，可以编写一个init脚本，放在使用这些脚本的目录中。作为scull源码的一部分，我们提供了一个相当完整和可配置的初始化脚本示例，称为scull.init；它接受常规参数- start，stop和restart-并执行scull\_load和scull\_unload的角色。

如果反复创建和销毁/dev节点听起来像是过分的行为，那么有一个有用的解决方法。 如果只加载和卸载单个驱动程序，则可以在第一次使用脚本创建设备文件后使用rmmod和insmod：动态数字不是随机的，并且可以指望每次选择相同的数字，如果不加载任何其他（动态）模块。避免冗长的脚本在开发过程中很有用。 但是，这个只适用于一次一个驱动程序的时候。

在我们看来，分配主设备号的最好方法是默认动态分配，同时让你自己可以选择在加载时或编译时指定主设备号。scull实现以这种方式工作；它使用一个全局变量scull\_major来保存选定的主设备号（对于次设备号也有一个scull\_minor）。该变量在scull.h中定义，初始化为SCULL\_MAJOR。源代码中SCULL\_MAJOR的缺省值为0，这意味着“使用动态分配”。用户可以接受缺省值或选择特定的主设备号，可以通过在编译前修改宏或在insmod中指定scull\_major的值。最后，通过使用scull\_load脚本，用户可以在scull\_load的命令行上将参数传递给insmod。

以下是我们在scull源代码中使用的代码，以获得一个主设备号：

|  |
| --- |
| if (scull\_major) {  dev = MKDEV(scull\_major, scull\_minor);  result = register\_chrdev\_region(dev, scull\_nr\_devs, "scull");  } else {  result = alloc\_chrdev\_region(&dev, scull\_minor, scull\_nr\_devs, "scull");  scull\_major = MAJOR(dev);  }  if (result < 0) {  printk(KERN\_WARNING "scull: can't get major %d\n", scull\_major);  return result;  } |

本书中几乎所有使用的示例驱动程序都使用类似的代码来进行主设备号的分配。

## 3.3 重要数据结构

如你所想，设备号注册只是驱动程序代码必须执行的众多任务中的一项。我们很快会看到其它重要的驱动程序组件，但首先需要考虑程序使用的数据结构。大部分基本的驱动程序操作都涉及三个重要的内核数据结构，称为file\_operations，file和inode。进行后面有趣的驱动程序实现之前，对这些数据结构的熟悉是必须的，所以我们现在将在详细介绍如何实现基本的驱动程序操作之前，先快速浏览它们一遍。

### 3.3.1 文件操作

到目前为止，我们已经预留了一些设备号供我们使用，但是我们还没有将我们的任何驱动程序的操作连接到这些设备号。file\_operations结构是字符驱动程序建立这种连接的关键。 该结构在<linux/fs.h>中定义，是一堆函数指针的集合。每个打开的文件（由一个文件结构内部表示，我们将很快检查）与它自己的一组函数相关联（包含一个名为f\_op的字段指向file\_operations结构）。这些操作主要负责实现系统调用，因此被命名为open，read等。我们可以将文件视为一个“对象”，并将操作它的函数作为它的“方法”，使用面向对象的编程术语来表示由对象声明的动作作用于它自身。这是我们在Linux内核中看到的第一个面向对象编程的地方，我们将在后面的章节中看到更多。

通常，file\_operations结构或指向它的指针被称为fops（或一些变体）。结构中的每个字段必须指向实现具体操作的驱动程序中的函数，或者对于不支持的操作将其设置为NULL。当指定NULL指针时，内核的确切行为对于每个函数都是不同的，本节后面的列表会列出。

以下列表介绍了应用程序可以在设备上调用的所有操作。我们试图保持列表简洁，以便它可以用作参考，只是在使用NULL指针时，总结每个操作和缺省的内核行为。

As you read through the list of file\_operations methods, you will note that a number of parameters include the string \_\_user. This annotation is a form of documentation, noting that a pointer is a user-space address that cannot be directly dereferenced. For normal compilation, \_\_user has no effect, but it can be used by external checking software to find misuse of user-space addresses.

在通读file\_operations方法列表时，会注意到一些参数包含字符串\_\_user。此注释是一种文档形式，注意指针是不能直接解除引用的用户空间地址。 对于正常的编译，\_\_user没有任何作用，但是它可以被外部检查软件用来发现用户空间地址的滥用。

The rest of the chapter, after describing some other important data structures, explains the role of the most important operations and offers hints, caveats, and real code examples. We defer discussion of the more complex operations to later chapters, because we aren’t ready to dig into topics such as memory management, blocking operations, and asynchronous notification quite yet.

本章的其余部分在描述了一些其他重要的数据结构之后，解释了最重要的操作的作用，并提供了提示，注意事项和真实代码示例。 我们将更复杂的操作讨论推迟到后面的章节，因为我们还没有准备好深入讨论内存管理，阻塞操作和异步通知等主题。

struct module \*owner

file\_operations的第一个字段根本不是一个操作；它是指向“拥有”结构的模块的指针。该字段用于防止模块在其操作使用期间被卸载。通常情况下，它都被初始化为THIS\_MODULE，它是一个在<linux/module.h>中定义的宏。

loff\_t (\*llseek) (struct file \*, loff\_t, int);

llseek方法用于更改文件中当前的读/写位置，并将新位置作为（正）返回值返回。loff\_t参数是一个“long offset”，即使在32位平台上也至少为64位。错误的话，返回负值。如果此函数指针为NULL，则seek调用将以不可预知的方式修改文件结构中的位置计数器（如“文件结构”一节中所述）。

ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);

用来从设备中读取数据。如果被赋为NULL空指针，造成read系统调用出错并返回-EINVAL（“Invalid argument”）。函数返回非负值表示成功读取的字节数（返回值为“signed size”数据类型，通常就是目标平台固有的整数类型）。

ssize\_t (\*aio\_read)(struct kiocb \*, char \_\_user \*, size\_t, loff\_t);

ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);

向设备发送数据。如果是NULL，write系统调用会返回-EINVAL。如果返回值非负，表示成功写入的字节数。

ssize\_t (\*aio\_write)(struct kiocb \*, const char \_\_user \*, size\_t, loff\_t \*);

int (\*readdir) (struct file \*, void \*, filldir\_t);

unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);

int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);

系统调用ioctl提供了一种发出特定设备命令的方法（例如，格式化软盘的某个磁道，其即不读也不写）。另外，内核还能识别一些ioctl命令，而不必调用fops表中的ioctl。如果设备未提供ioctl方法，则对于任何未预定义的请求，系统调用都会返回错误（-ENOTTY，“No such ioctl for device”（该设备没有ioctl方法））

int (\*mmap) (struct file \*, struct vm\_area\_struct \*);

int (\*open) (struct inode \*, struct file \*);

尽管这始终是对设备文件进行的第一个操作，然而却不要求驱动程序一定要声明一个相应的方法。如果设置为NULL，设备永远打开成功，但是系统不会通知驱动程序。

int (\*flush) (struct file \*);

int (\*release) (struct inode \*, struct file \*);

当file结构被释放时，将调用这个操作。与open相仿，也可以将这个release设置为NULL。

int (\*fsync) (struct file \*, struct dentry \*, int);

int (\*aio\_fsync)(struct kiocb \*, int);

int (\*fasync) (int, struct file \*, int);

int (\*lock) (struct file \*, int, struct file\_lock \*);

ssize\_t (\*readv) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);

ssize\_t (\*writev) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);

ssize\_t (\*sendfile)(struct file \*, loff\_t \*, size\_t, read\_actor\_t, void \*);

ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);

unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);

int (\*check\_flags)(int);

int (\*dir\_notify)(struct file \*, unsigned long);scull设备驱动程序只实现最重要的设备方法。 其file\_operations结构初始化如下：

|  |
| --- |
| struct file\_operations scull\_fops = {  .owner = THIS\_MODULE,  .llseek = scull\_llseek,  .read = scull\_read,  .write = scull\_write,  .ioctl = scull\_ioctl,  .open = scull\_open,  .release = scull\_release,  }; |

此声明使用标准C的标记结构初始化语法。采用这种方法，因为它使得驱动程序在结构定义的变化中更具可移植性，并且可以使代码更加紧凑和可读。标记初始化方法允许对结构成员重新排列。在某些场合下，将频繁被访问的成员放在相同的硬件缓存行上，将大大提高性能。

### 3.3.2 file结构

在<linux/fs.h>中定义的结构体file是设备驱动程序中使用的第二个最重要的数据结构。请注意，file结构体和用户空间程序的FILE没有任何关联。FILE在C库中定义，永远不会出现在内核代码中。另一方面，struct file是内核结构，永远不会出现在用户程序中。

该file结构表示一个打开的文件。（它不是特定于设备驱动程序;系统中的每个打开的文件在内核空间中都有一个关联的file结构。）它由内核在open时创建，并传递给操作文件的任何函数，直到最后一次关闭。文件的所有实例关闭后，内核会释放数据结构。

在内核源代码中，指向file结构的指针通常称为file或filp（“文件指针”）。为了防止和file结构本身相混淆，我们一致称为filp指针。这样，file指的是结构，filp则是指向该结构的指针。

下面解释了file结构中最重要的成员。与之前的段落一样，下面罗列的成员在初次阅读的时候可以跳过。在本章的后面，当我们查看一些真正的C代码的时候，我们会详细讨论其中的成员。

mode\_t f\_mode;

The file mode identifies the file as either readable or writable (or both), by means of the bits FMODE\_READ and FMODE\_WRITE. You might want to check this field for read/write permission in your open or ioctl function, but you don’t need to check permissions for read and write, because the kernel checks before invoking your method. An attempt to read or write when the file has not been opened for that type of access is rejected without the driver even knowing about it.

文件模式通过FMODE\_READ和FMODE\_WRITE位将文件标识为可读或可写（或两者）。 您可能希望在open或ioctl函数中检查此字段的读/写权限，但不需要检查读写权限，因为内核在调用方法之前进行检查。 没有驱动程序甚至不知道该文件时，如果文件未被打开以便进行读取或写入操作，则会被拒绝。

loff\_t f\_pos;

The current reading or writing position. loff\_t is a 64-bit value on all platforms (long long in gcc terminology). The driver can read this value if it needs to know the current position in the file but should not normally change it; read and write should update a position using the pointer they receive as the last argument instead of acting on filp->f\_pos directly. The one exception to this rule is in the llseek method, the purpose of which is to change the file position.

当前的阅读或写作位置。 loff\_t是所有平台上的64位值（gcc术语中的long long）。 如果需要知道文件中的当前位置，驱动程序可以读取该值，但通常不应更改该值; 读写应该使用它们接收的指针作为最后一个参数来更新一个位置，而不是直接对filp-> f\_pos进行操作。 这个规则的一个例外是llseek方法，其目的是改变文件的位置。

unsigned int f\_flags;

These are the file flags, such as O\_RDONLY, O\_NONBLOCK, and O\_SYNC. A driver should check the O\_NONBLOCK flag to see if nonblocking operation has been requested (we discuss nonblocking I/O in the section “Blocking and Nonblocking Operations” in Chapter 1); the other flags are seldom used. In particular, read/write permission should be checked using f\_mode rather than f\_flags. All the flags are defined in the header <linux/fcntl.h>.

这些是文件标志，例如O\_RDONLY，O\_NONBLOCK和O\_SYNC。 驱动程序应检查O\_NONBLOCK标志以查看是否请求了非阻塞操作（我们在第1章中的“阻塞和非阻塞操作”一节中讨论了非阻塞I / O）。 其他标志很少使用。 特别是，应该使用f\_mode而不是f\_flags来检查读/写权限。 所有标志都在头文件<linux / fcntl.h>中定义。

struct file\_operations \*f\_op;

The operations associated with the file. The kernel assigns the pointer as part of its implementation of open and then reads it when it needs to dispatch any operations. The value in filp->f\_op is never saved by the kernel for later reference; this means that you can change the file operations associated with your file, and the new methods will be effective after you return to the caller. For example, the code for open associated with major number 1 (/dev/null, /dev/zero, and so on) substitutes the operations in filp->f\_op depending on the minor number being opened. This practice allows the implementation of several behaviors under the same major number without introducing overhead at each system call. The ability to replace the file operations is the kernel equivalent of “method overriding” in object-oriented programming.

与文件关联的操作。 内核将指针指定为open的实现的一部分，然后在需要分派任何操作时读取它。 filp-> f\_op中的值永远不会被内核保存供以后参考; 这意味着您可以更改与文件关联的文件操作，并且在返回给调用者后新方法将生效。 例如，与主编号1（/ dev / null，/ dev / zero等）关联的打开代码取决于正在打开的次编号在filp-> f\_op中的操作。 这种做法允许在相同的主号码下执行多个行为，而不会在每次系统调用时引入开销。 替换文件操作的能力是面向对象编程中“方法重写”的内核等价物。

void \*private\_data;

The open system call sets this pointer to NULL before calling the open method for the driver. You are free to make its own use of the field or to ignore it; you can use the field to point to allocated data, but then you must remember to free that memory in the release method before the file structure is destroyed by the kernel. private\_data is a useful resource for preserving state information across system calls and is used by most of our sample modules.

在调用驱动程序的打开方法之前，开放系统调用将此指针设置为NULL。 你可以自由使用该领域或忽略它; 您可以使用该字段指向已分配的数据，但是在文件结构被内核销毁之前，您必须记住在释放方法中释放该内存。 private\_data是跨系统调用保存状态信息的有用资源，并且被大多数示例模块使用。

struct dentry \*f\_dentry;

The directory entry (dentry) structure associated with the file. Device driver writers normally need not concern themselves with dentry structures, other than to access the inode structure as filp->f\_dentry->d\_inode.

与文件关联的目录条目（dentry）结构。 设备驱动程序编写者通常不需要关心dentry结构，只需要以filp-> f\_dentry-> d\_inode的形式访问inode结构。

真正的file结构有更多的成员，但它们对设备驱动程序无用。 我们可以放心地忽略这些成员，因为驱动程序从不创建file结构，他们只能通过在别处创建的结构引用。

### 3.3.3 inode结构

内核使用inode结构来表示文件。因此，它与file结构不同，后者表示打开的文件描述符。对单个文件，可能会有许多个表示打开的文件描述符的file结构存在，但是它们都指向单个inode结构。

inode结构包含大量关于有关文件的信息。一般情况下，此结构中只有两个成员对编写驱动程序代码很有用：

* dev\_t i\_rdev;  
  对于表示设备文件的inode，此成员包含实际的设备编号。
* struct cdev \*i\_cdev;  
  struct cdev是表示字符设备的内核的内部结构；当inode指向一个字符设备文件时，该成员包含指向该结构的指针。

在2.5系列开发过程中，i\_rdev的类型发生了变化，这破坏了大量驱动程序的兼容性。为了鼓励编写可移植性更强的代码，内核开发人员添加了两个宏，可用于从inode获取主和次设备编号：

|  |
| --- |
| unsigned int iminor(struct inode \*inode);  unsigned int imajor(struct inode \*inode); |

为了防止因为类似的改变而出现问题，应该使用这些宏而不是直接操作i\_rdev。

## 3.4 字符设备注册

正如我们所提到的，内核在内部使用struct cdev类型的结构表示字符设备。在内核调用设备的操作之前，必须分配并注册一个或多个上述结构。为此，我们的代码应该包含<linux/cdev.h>，该文件定义了这个结构及其相关的辅助函数。

分配和初始化上述结构的方法有两种。如果读者希望在运行时获得独立的cdev结构，则应该使用如下代码来实现：

|  |
| --- |
| struct cdev \*my\_cdev = cdev\_alloc( );  my\_cdev->ops = &my\_fops; |

但是，你可能希望将cdev结构嵌入到自己的特定设备结构中；scull就是这么做的。在这种情况下，需要使用下面的代码来初始化已经分配的结构：

|  |
| --- |
| void cdev\_init(struct cdev \*cdev, struct file\_operations \*fops); |

无论哪种方式，还有一个struct cdev的成员需要初始化。和file\_operations结构一样，struct cdev的owner字段应该设置为THIS\_MODULE。

在cdev结构设置好后，最后一步就是通过下面的调用告知内核该结构的信息：

|  |
| --- |
| int cdev\_add(struct cdev \*dev, dev\_t num, unsigned int count); |

这里，dev是cdev结构，num是该设备对应的第一个设备编号，count是与设备关联的设备编号的数量。通常count取1，但在某些情况下，有多个设备编号对应于特定设备是有意义的。 例如，考虑SCSI磁带驱动程序，它允许用户空间通过为每个物理设备分配多个次设备编号来选择操作模式（如密度）。

使用cdev\_add时需要牢记几件重要的事情。首先，这个调用可能会失败。如果它返回负的错误码，则说明设备尚未添加到系统中。然而，它几乎总是成功的，这就又提出了另一个问题：只要cdev\_add返回，我们的设备就是“活着的”，它的操作可以被内核调用。所以，在驱动程序完全准备好处理设备上的操作之前，不应该调用cdev\_add。

要从系统中移除一个字符设备，请调用：

|  |
| --- |
| void cdev\_del(struct cdev \*dev); |

显然，在将cdev结构传递给cdev\_del之后，就不应该再访问cdev结构了。

### 3.4.1 scull中的设备注册

在scull内部，它通过struct scull\_dev结构来表示每个设备。这个结构的定义如下：

|  |
| --- |
| struct scull\_dev {  struct scull\_qset \*data; /\* 指向第一个量子集的指针 \*/  int quantum; /\* 当前量子的大小 \*/  int qset; /\* 当前数组的大小 \*/  unsigned long size; /\* 保存在其中的数据总量 \*/  unsigned int access\_key; /\* sculluid 和scullpriv使用 \*/  struct semaphore sem; /\* 互斥信号量 \*/  struct cdev cdev; /\* 字符设备结构 \*/  }; |

当我们遇到这个结构体中的成员的时候再展开讨论，但是现在，我们先将注意力放到cdev身上，结构体cdev是我们的设备和内核之间的接口。这个结构体必须如上所述地被初始化并被添加到系统中，scull中的处理这一部分的代码如下：

|  |
| --- |
| static void scull\_setup\_cdev(struct scull\_dev \*dev, int index)  {  int err, devno = MKDEV(scull\_major, scull\_minor + index);  cdev\_init(&dev->cdev, &scull\_fops);  dev->cdev.owner = THIS\_MODULE;  dev->cdev.ops = &scull\_fops;  err = cdev\_add (&dev->cdev, devno, 1);  /\* Fail gracefully if need be \*/  if (err)  printk(KERN\_NOTICE "Error %d adding scull%d", err, index);  } |

由于cdev结构嵌入在struct scull\_dev中，因此必须调用cdev\_init才能执行该结构的初始化。

### 3.4.2 早期的方法

如果读者阅读2.6内核中的其它驱动程序代码，也许会注意到相当数量的字符驱动程序不使用我们前面描述的cdev接口。其实，读者看到的是还没有升级到2.6接口的老代码。因为这些代码也可以工作，因此在较长的时间内升级可能不会发生。为了完整起见，我们会描述早期的字符设备注册接口，但是新代码不应该使用它，因为这种机制会在未来的内核中消失。

注册字符设备驱动程序的经典方法是：

|  |
| --- |
| int register\_chrdev(unsigned int major, const char \*name, struct file\_operations \*fops); |

在这里，major是设备的主设备号，name是驱动程序的名称（它出现在/proc/devices中），而fops是默认的file\_operations结构。对register\_chrdev的调用会为给定的主设备号注册0-255作为次设备号，并为每个设备建立一个默认的cdev结构。使用此接口的驱动程序必须准备好处理所有256个次设备号（无论它们是否与真实设备相对应）的open调用，并且它们不能使用大于255的主或次设备号。

对应的移除字符设备的函数是：

|  |
| --- |
| int unregister\_chrdev(unsigned int major, const char \*name); |

major和name必须与传递给register\_chrdev的相同，否则调用将失败。

## 3.5 open和release

现在，我们已经简单浏览了这些字段，下面将在实际的scull函数中使用这些字段。

### 3.5.1 open方法

open方法提供给驱动程序进行初始化的能力，为以后的操作做准备。在大多数驱动程序中，open应该执行以下任务：

* 检查设备特定错误（像设备未就绪或类似的硬件问题）；
* 如果设备是首次打开，初始化设备；
* 如有必要，更新f\_op指针；
* 分配并填充放置于filp->private\_data中的任何数据结构；

然而，首先要做的就是确定要打开的设备。注意，open方法的原型如下：

|  |
| --- |
| int (\*open)(struct inode \*inode, struct file \*filp); |

参数inode以i\_cdev字段的形式提供我们需要的信息，其中包含我们之前设置的cdev结构。 唯一的问题是我们通常不需要cdev结构本身，我们希望包含该cdev结构的scull\_dev结构。 C语言让程序员可以玩各种各样的技巧来进行这种转换；然而，使用这样的技巧是容易出错的，并且导致其他人难以阅读和理解代码。幸运的是，在这种情况下，内核hacker已经为我们做了棘手的事情，它通过定义在<linux / kernel.h>中的container\_of宏的形式实现：

|  |
| --- |
| container\_of(pointer, container\_type, container\_field); |

这个宏在类型为container\_type的结构中接受一个类型为container\_field的字段的指针，并返回一个指向包含该字段的结构的指针。在scull\_open中，该宏用于查找适当的设备结构：

|  |
| --- |
| struct scull\_dev \*dev; /\* 设备信息 \*/  dev = container\_of(inode->i\_cdev, struct scull\_dev, cdev);  filp->private\_data = dev; /\* for other methods \*/ |

一旦找到了scull\_dev结构，scull就会在file结构的private\_data字段中存储一个指向它的指针，以便将来访问。

另外一个确定要打开的设备的方法是查看inode结构中存储的次设备号。如果使用register\_chrdev注册设备，则必须使用此技术。 请务必使用iminor宏从inode结构中获取次设备号，并确保它对应于驱动程序实际准备打开的设备。

经过简化的scull\_open的代码如下：

|  |
| --- |
| int scull\_open(struct inode \*inode, struct file \*filp)  {  struct scull\_dev \*dev; /\* 设备信息 \*/  dev = container\_of(inode->i\_cdev, struct scull\_dev, cdev);  filp->private\_data = dev; /\* 其它方法 \*/  /\* now trim to 0 the length of the device if open was write-only \*/  if ( (filp->f\_flags & O\_ACCMODE) = = O\_WRONLY) {  scull\_trim(dev); /\* ignore errors \*/  }  return 0; /\* success \*/  } |

这段代码看起来相当短小，因为在调用open时不执行任何特定的设备处理。因为scull设备被设计为全局且持久的，这段代码无需做什么工作。特别是，我们不需要维护scull设备的打开计数，而只维护模块的使用计数，因此也就没有“首次打开时初始化设备”这类动作。

对设备唯一的实际操作就是，当设备以写的方式打开时，它的长度被截为0.出现这种特性的原因在于，在设计上，当用更短的文件覆盖一个scull设备时，设备数据区应相应缩小。这与用写方式打开普通文件时将长度截短为0的方式很相似。如果设备以读取方式打开，则什么也不做。

### 3.5.2 release方法

release方法的作用正好和open相反。有时候，读者会发现这个方法的实现被称为device\_close。无论是哪种形式，都应实现下面的功能：

释放由open分配的、保存在filp->private\_data中的所有内容；

在最后一次关闭操作时关闭设备；

最基本的scull没有硬件可以关闭，因此所需的代码是最小的：

|  |
| --- |
| int scull\_release(struct inode \*inode, struct file \*filp)  {  return 0;  } |

当关闭一个设备文件的次数比打开它的次数多时，系统中会发生什么情况呢？毕竟，dup和fork都会在不调用open的情况下创建已打开文件的副本；但是在程序终止时每一个副本都会被关闭。例如，大多数程序从来不会打开它们的stdin文件（或设备），但是它们都会在终止时关闭。驱动程序如何才能知道已打开设备文件要被真正关闭呢？

答案很简单：并不是每个close系统调用都会导致release方法被调用。只有实际释放设备数据结构的close调用才会调用该方法–这就是它名称的由来。内核保存一个file结构使用次数的计数器。fork和dup都不会创建一个新的file结构（只能打开）。他们只是增加现有结构中的计数器。只有当file结构的计数器下降到0时，close系统调用才会执行release方法，这发生在file结构被销毁时。release方法和close系统调用之间的这种关系可以保证对于每次open调用，驱动程序只会看到一个release调用。

请注意，flush方法在应用程序每次调用close时都会调用。然而，很少有驱动程序实现flush，因为除非release被调用，否则通常没有什么可以在close时去做。

正如你想象的那样，即使应用程序终止时，没有显式地关闭它打开的文件，前面的讨论也是适用的：内核会在进程退出时自动调用close系统调用关闭所有相关的文件。

## 3.6 scull内存使用

在介绍读写操作之前，我们最好先看看scull如何并为何执行内存分配。“如何分配”有助于彻底理解代码，“为何分配”则表明了驱动程序作者需要做出的选择，尽管scull作为一种设备来说，肯定还不具备代表性。

本节仅涉及scull中的内存分配策略，并未涉及编写实际驱动程序所需的硬件管理技能。这些技巧在第9章和第10章中介绍。因此，如果读者对面向内存操作的scull驱动程序的内部工作原理不感兴趣的话，可以跳过本节。

scull使用的内存区域也称为设备，长度可变。写的越多，它就越长; 通过用较短的文件重写设备来执行修剪。

scull驱动程序引入了在Linux内核中管理内存的两个主要函数。定义在<linux/slab.h>中：

|  |
| --- |
| void \*kmalloc(size\_t size, int flags);  void kfree(void \*ptr); |

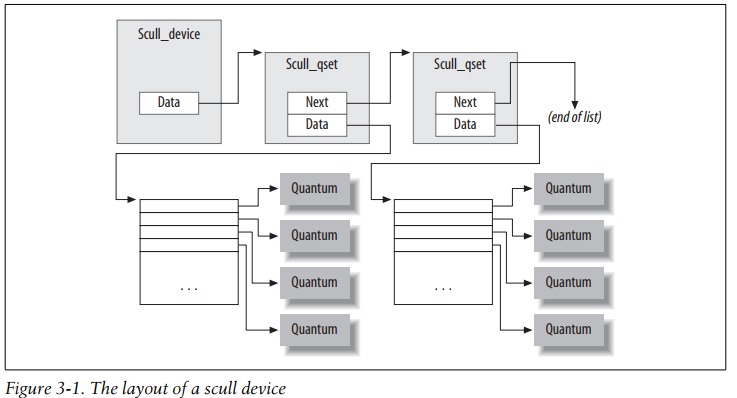
对kmalloc的调用会尝试分配size个大小的内存；返回值是指向该内存的指针，如果分配失败，则返回NULL。flags参数用于描述如何分配内存；我们在第8章详细讨论这些标志。现在，我们一直使用GFP\_KERNEL。分配的内存应该用kfree释放。我们不应该把非kmalloc返回的指针传递给kfree。但是，将NULL指针传递给kfree是合法的。

kmalloc并不是分配大块内存的最有效方式（参见第8章），所以为scull选择的实现并不是特别巧妙。更巧妙的实现的源代码将更难以阅读，但本节的目标是演示读写，而不是内存管理。这就是为什么虽然按整页分配更高效，而代码只使用了kmalloc和kfree而没有采取分配整个页面的操作的原因。

另一方面，从理论和实际的角度，我们不想限制“设备”区域的大小。从理论角度来看，对被管理的数据项任意施加限制总是一个坏主意。从实际角度来看，为了在较小内存的情况下进行测试，scull可以用来暂时吞噬您的系统内存。运行这些测试可能会帮助你了解系统的内部。可以使用命令cp /dev/zero /dev/scull0用光所有系统RAM，也可以使用dd工具选择将多少数据复制到scull设备中。

In scull, each device is a linked list of pointers, each of which points to a scull\_dev structure. Each such structure can refer, by default, to at most four million bytes, through an array of intermediate pointers. The released source uses an array of 1000 pointers to areas of 4000 bytes. We call each memory area a quantum and the array (or its length) a quantum set. A scull device and its memory areas are shown in Figure 3-1.

在scull中，每个设备都是一个指针链表，每个指针指向一个scull\_dev结构。默认情况下，每个这样的结构可以通过一组中间指针数组引用至多400万个字节。我们发布的源代码使用了一个有1000个指针的数组，每个指针指向一个4000个字节的区域。我们称每个存储区为一个量子，而指针数组（或其长度）称为量子集。图3-1显示了scull设备及其存储区。



这样，选择的参数使得向sucll写入一个字节，就会消耗8000或12000个字节的内存：每个量子占用4000个字节，而一个量子集占用4000到8000个字节（取决于目标平台的指针本身占用32位还是64位）。然而，如果向scull写入大量的数据，链表的开销不会太大。每4M字节对应一个链表元素，而设备的最大尺寸受计算机内存大小的限制。

为量子和量子集合选择适当的值是一个策略问题，而不是机制，最佳大小取决于设备的使用方式。因此，scull驱动程序不应该强制使用任何特定的量子和量子集尺寸数值。在scull中，用户可以通过以下几种方式更改该数值：通过在编译时更改scull.h中的宏SCULL\_QUANTUM和SCULL\_QSET；在模块加载时设置整数值scull\_quantum和scull\_qset；或者在运行时，使用ioctl更改当前值和默认值。

使用宏和整数值来允许编译时和加载时进行配置，这种方法和前面选择主设备号的方法类似。 对于驱动程序中任何不确定的或与策略相关的数值我们都可以使用这种技巧。

剩下的唯一问题是如何选择默认数值。在这种特殊情况下，量子和量子集未充分填满会造成内存浪费，而如果量子和量子集过小则会在进行内存分配、释放和指针链接等操作时增加系统开销，默认数值的选择就是在这两者之间找到最佳平衡点。 此外，还应考虑kmalloc的内部设计。（不过我们现在不会去追求这一点；第8章将探讨kmalloc的内部特征）。现在，默认数值的选择基于在测试时可能会写入大量的数据的假设，尽管正常情况下，可能只传输几K字节的数据。

我们已经看到了代表设备内部的scull\_dev结构。该结构的quantum和qset字段分别保存设备的量子和量子集大小。但是，实际的数据由另外一个结构处理，对于该结构，我们称之为struct scull\_qset：

|  |
| --- |
| struct scull\_qset {  void \*\*data;  struct scull\_qset \*next;  }; |

下面的代码段说明了如何利用struct scull\_dev和struct scull\_qset来保存数据。函数scull\_trim负责释放整个数据区域，并且在文件以写入方式打开时由scull\_open调用。它简单地遍历链表，并释放所有找到的量子和量子集。

|  |
| --- |
| int scull\_trim(struct scull\_dev \*dev)  {  struct scull\_qset \*next, \*dptr;  int qset = dev->qset; /\* "dev" is not-null \*/  int i;  for (dptr = dev->data; dptr; dptr = next) { /\* all the list items \*/  if (dptr->data) {  for (i = 0; i < qset; i++)  kfree(dptr->data[i]);  kfree(dptr->data);  dptr->data = NULL;  }  next = dptr->next;  kfree(dptr);  }  dev->size = 0;  dev->quantum = scull\_quantum;  dev->qset = scull\_qset;  dev->data = NULL;  return 0;  } |

scull\_trim也在模块清理函数中用于将scull使用的内存返回给系统。

## 3.7 write和read

read和write方法完成的任务相似，亦即拷贝数据到应用程序空间，或反过来从应用程序空间拷贝数据。因此，它们的原型十分相似，不妨同时介绍它们：

|  |
| --- |
| ssize\_t read(struct file \*filp, char \_\_user \*buff, size\_t count, loff\_t \*offp);  ssize\_t write(struct file \*filp, const char \_\_user \*buff, size\_t count, loff\_t \*offp); |

对于这两个方法，参数flip是文件指针，参数count是请求传输的数据长度。参数buff是指向用户空间的缓冲区，这个缓冲区或者保存要写入的数据，或者是一个存放新读入数据的空缓冲区。最后的offp是一个指向“long offset type”对象的指针，这个对象指明用户在文件中存取操作的位置。返回值是“signed size”类型，后面会谈到它的用法。

需要再次指出的是，buff参数是用户空间的指针。因此，内核代码不能直接应用其中的内容。出现这种限制的原因有如下几个：

* 随着驱动程序所运行的架构不同或者内核配置不同，在内核模式中运行时，用户空间的指针可能是无效的。该地址可能根本无法映射到内核空间，或者可能指向某些随机的数据。
* 即使该指针在内核空间代表相同的东西，但用户空间的内存是分页的，而在调用系统调用时，涉及的内存可能根本就不在RAM中。对用户空间内存的直接引用将导致页错误，而这对内核代码来说是不允许发生的事情。其结果可能就是一个“oop”，它将导致调用该系统调用的进程死亡。
* 我们讨论的指针可能由用户程序提供，而该程序可能存在缺陷或者是恶意程序。如果我们的驱动程序盲目引用用户提供的指针，将导致系统出现打开的后门，从而允许用户空间程序随意访问或覆盖系统中的内存。如果读者不打算因为自己的驱动程序而危及用户系统的安全性，则永远不应直接应用用户空间的指针。

很显然，驱动程序必须访问用户空间的缓冲区以便完成自己的工作。为了确保安全，这种访问应该始终通过内核的专有函数来完成。这里，我们将介绍几个类似的函数（定义在<asm/uaccess.h>），其它的函数将在第6章介绍。这些函数使用了一些特殊地与架构相关的方法来确保在内核和用户空间之间安全正确地交换数据。

scull的read和write要做的工作就是在内核地址空间和用户地址空间之间进行整段数据的拷贝。这种能力是由下面的内核函数完成的，

|  |
| --- |
| unsigned long copy\_to\_user(void \_\_user \*to, const void \*from, unsigned long count);  unsigned long copy\_from\_user(void \*to, const void \_\_user \*from, unsigned long count); |

3.7.1 read方法

3.7.2 write方法

3.7.3 readv和writev

## 3.8 使用新设备

一旦准备好了上面的四个方法就可以编译和测试驱动程序了。它保留写入的数据，直到用新数据覆盖。这个设备有点像长度只受物理RAM限制的数据缓冲区。

### 3.8.1 测试环境

测试平台：qemu+mini2440

构建内核工具：buildroot-2012.05

Linux内核版本：Linux 3.3.7

### 3.8.2 测试过程

1. 安装

编译完成后，使用命令modprobe或insmod安装scull.ko模块。查看安装成功的方法可以使用下面的命令：

|  |
| --- |
| cat /proc/devices | grep scull |

结果，

|  |
| --- |
| 251 scull |

如果有类似上面的信息（数字可能不一样），则说明一切顺利。否则说明模块初始化有错误，可查看/var/log/kern.log 了解具体情况。

或者使用lsmod查看。会看到scull模块已经安装。

2. 建立设备文件

|  |
| --- |
| sudo mknod /dev/scull c 251 0 // 按照查看的设备号更改为251 |

3. 测试

|  |
| --- |
| echo 123 > /dev/scull  cat /dev/scull |

如果看到输出123证明设备正常工作。

4. 设备移除

|  |
| --- |
| rm /dev/scull  sudo rmmod scull |

# 第4章 调试技术

内核编程由于其特殊性，带来了调试方面的挑战。内核代码不能在调试器下运行，也不容易被追踪，因为它是一组与特定进程无关的函数组成的。内核代码错误难以再现，并且容易造成整个系统的崩溃，从而破坏了追踪它们的证据。

本文介绍了在这种及其困难的环境下监视内核代码和追踪错误的技术。

## 4.1内核中的调试支持

在第二章中，我们推荐读者编译并运行自己的内核，而不是运行宿主机上发布版本的备份内核。一个重要的原因就是，内核开发者为内核提供了一个调试功能。这些调试功能能够产生额外的输出信息但也降低了系统性能，所以，一般在正式的发布版本内核里，这些功能都不会被使能。但是，我们现在的侧重点不一样，能够接受因为额外的调试功能而带来的开销。

在这里，我们列出开发内核代码时，常用的配置选项。除非另有说明，否则在你使用的任何内核配置工具的“kernel hacking”菜单下找到这些配置项。值得注意的是，这些选项中并不是所有的架构都支持。

CONFIG\_DEBUG\_KERNEL

DEBUG配置“总开关”，打开它，其它功能才能使用。其本身没有任何调试功能。

CONFIG\_DEBUG\_SLAB

这个关键选项打开内核内存分配函数中的几种类型检查；随着这些检查被使能，就能够检测内存溢出和忘记初始化等错误。分配的每一个字节都会被设为0xA5，释放时，每个字节被设置为0x6B。如果你看到上面的字节数据重复性地出现在驱动程序的输出信息（或经常出现在oop列表中），很容易能够定位问题所在。还有，如果debug功能被使能，内核会在分配的内存对象的前后设置守卫值，如果内存发生溢出，内核就会知道。同时，更多隐含的错误检查也会被使能。

|  |
| --- |
| CONFIG\_DEBUG\_PAGEALLOC  CONFIG\_DEBUG\_SPINLOCK  CONFIG\_DEBUG\_SPINLOCK\_SLEEP  CONFIG\_INIT\_DEBUG  CONFIG\_DEBUG\_INFO  CONFIG\_MAGIC\_SYSRQ  CONFIG\_DEBUG\_STACKOVERFLOW  CONFIG\_DEBUG\_STACK\_USAGE  CONFIG\_KALLSYMS  CONFIG\_IKCONFIG  CONFIG\_IKCONFIG\_PROC  CONFIG\_ACPI\_DEBUG  CONFIG\_DEBUG\_DRIVER  CONFIG\_SCSI\_CONSTANTS  CONFIG\_INPUT\_EVBUG  CONFIG\_PROFILING |

后面我们在查看各种追踪内核问题技术的时候，还会重新浏览上面的配置项。首先，让我们来看看最为经典的调试技术：prink语句。

## 4.2 使用打印调试

最常用的调试技术之一就是监视，在应用程序编程中，通过在合适的地方添加printf来实现。同样，在内核编程中，使用printk也可以实现。

### 4.2.1 printk

在前面的章节中使用printk函数时，作了简单的假设，其工作方式和printf一样。现在，是时候介绍一些它们的不同了。

不同之一就是，可以根据消息的严重性将其分类，比如将日志级别、优先级等与消息关联起来。可以使用宏定义指明日志级别。例如，在前面的print语句中使用的KERN\_INFO，就是表示日志级别的宏定义。日志级别宏定义展开为一个字符串，在编译时链接到消息文本信息。这也是为什么在下面的例子中，优先级和格式化字符串之间没有“，”的原因。下面是打印调试信息和关键信息的2条printk语句。

|  |
| --- |
| printk(KERN\_DEBUG "Here I am: %s:%i\n", \_\_FILE\_\_, \_\_LINE\_\_);  printk(KERN\_CRIT "I'm trashed; giving up on %p\n", ptr); |

在头文件<linux/printk.h>（内核版本不同而不同）中定义了8个可能的日志级别字符串；

|  |
| --- |
| #define KERN\_EMERG "<0>" /\* system is unusable \*/  #define KERN\_ALERT "<1>" /\* action must be taken immediately \*/  #define KERN\_CRIT "<2>" /\* critical conditions \*/  #define KERN\_ERR "<3>" /\* error conditions \*/  #define KERN\_WARNING "<4>" /\* warning conditions \*/  #define KERN\_NOTICE "<5>" /\* normal but significant condition \*/  #define KERN\_INFO "<6>" /\* informational \*/  #define KERN\_DEBUG "<7>" /\* debug-level messages \*/ |

其数值分别为0~7，数值越小，优先级越高。

printk语句的默认优先级是DEFAULT\_MESSAGE\_LOGLEVEL，在kernel/printk.c函数中定义，是一个整数。3.3.7内核中，该值为KERN\_WARNING，版本不同，其值可能会发生变化。

根据日志级别，内核可以打印消息到现在的控制台，文本模式终端，串行端口或者并行打印机。如果优先级小于整数变量console\_loglevel，则消息一次发送到控制台一行（除非后面跟随换行符，否则不发送任何消息）。如果klogd和syslogd都在运行，那么内核消息会被添加到/var/log/messages（或者依赖于syslogd配置作其它处理），而与console\_loglevel无关。如果klogd没有正在运行，消息不会被送到用户空间，除非从/proc/kmsg中读取（可以通过命令dmesg命令实现）。当使用klogd时，应该记住，其不会保存连续相同的行，只会保存一行，然后记录重复的次数。

变量console\_loglevel被初始化为DEFAULT\_CONSOLE\_LOGLEVEL，并且可以通过sys\_syslog系统调用进行修改。改变它的一种方法是在调用klogd时指定-c开关，如klogd手册页中指定的那样。请注意，要更改当前值，必须先杀掉klogd，然后使用-c选项重新启动它。或者，可以编写一个程序来更改控制台日志级别。您可以在O'Reilly的FTP站点上提供的源文件中的misc-progs/setlevel.c中找到此类程序的一个版本。新的级别被指定为介于1和8之间的整数值。如果它设置为1，则只有级别0（KERN\_EMERG）的消息到达控制台；如果它设置为8，则显示所有消息，包括调试消息。

也可以使用文件/proc/sys/kernel/printk来读取和修改控制台日志级别。可以使用下面的命令读取其内容，

|  |
| --- |
| # cat /proc/sys/kernel/printk  7 4 1 7 |

该文件拥有4个整数值：当前的日志级别，缺省时默认日志级别，允许的最低日志级别以及启动时默认的日志级别。向该文件写入某个值可以更改当前的日志级别。例如，只需输入以下内容即可使控制台上显示所有内核消息：

|  |
| --- |
| # echo 8 > /proc/sys/kernel/printk |

现在应该明白为什么hello.c示例中使用KERN\_ALERT日志级别；是确保消息出现在控制台上。

### 4.2.2 重定向控制台信息

可以在任何一个控制台上调用ioctl（TIOCLINUX）命令来指定接收消息的其它虚拟终端。必须是在超级用户权限运行，在misc-progs目录下能够找到它。

以下是该程序的全部内容。

|  |
| --- |
| /\*  \* 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., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.  \*/    #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <errno.h>  #include <unistd.h>  #include <sys/ioctl.h>  int main(int argc, char \*\*argv)  {  char bytes[2] = {11,0}; /\* 11 is the TIOCLINUX cmd number \*/  if (argc= =2)  bytes[1] = atoi(argv[1]); /\* 选中的控制台 \*/  else {  fprintf(stderr, "%s: need a single arg\n",argv[0]);  exit(1);  }  if (ioctl(STDIN\_FILENO, TIOCLINUX, bytes)<0) { /\* use stdin \*/  fprintf(stderr,"%s: ioctl(stdin, TIOCLINUX): %s\n", argv[0], strerror(errno));  exit(1);  }  exit(0);  } |

setconsole使用了特殊的ioctl命令：TIOCLINUX，这个命令可以实现一些特定的linux功能。从代码中可以看出，要使用TIOCLINUX，必须传递给一个参数，该参数就是要指定的控制台。在setconsole中，使用子命令11，并且下一个字节（以byte[1]存储）标识虚拟控制台。关于TIOCLINUX的完整描述可以在drivers/tty/tty\_io.c中的内核源代码中找到。

### 4.2.3 消息是如何记录的

printk函数将消息写入\_\_LOG\_BUF\_LEN字节长的循环缓冲区中：\_\_LOG\_BUF\_LEN的大小在配置内核时可以选择为4 KB至1 MB。然后，该函数唤醒正在等待消息的任何进程，比如，在syslog系统调用中休眠的进程或读取/proc/kmsg文件的进程。

对/proc/kmsg进行读取操作，日志缓冲区被读取的数据就不再保留。

syslog系统调用可以选择返回日志数据并保留数据给其它进程使用。

dmesg命令可在不刷新缓冲区数据的情况下获得缓冲区的内容。

如果循环缓冲区填满了，printk就绕回缓冲区的开始出填写新的数据，将覆盖最早的数据。

### 4.2.4 打开和关闭消息

通过预处理指令开启和关闭调试信息。因为我们在调试驱动程序的过程中不可能每次添加删除打印调试信息。共有2种方法：

* 通过删除或添加宏，启用或禁用打印语句。
* 在编译之前，通过更改CFLAGS变量的值，可以一次禁用所有消息。

代码宏定义实现，查看头文件scull.h的代码：

|  |
| --- |
| #undef PDEBUG /\* undef it, just in case \*/  #ifdef SCULL\_DEBUG  #ifdef \_\_KERNEL\_\_  /\* This one if debugging is on, and kernel space \*/  # define PDEBUG(fmt, args...) printk( KERN\_DEBUG "scull: " fmt, ## args)  # else  /\* This one for user space \*/  # define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)  # endif  #else  #define PDEBUG(fmt, args...) /\* not debugging: nothing \*/  #endif  #undef PDEBUGG  #define PDEBUGG(fmt, args...) /\* nothing: it's a placeholder \*/ |

为了进一步简化该过程，请将以下行添加到Makefile中：

|  |
| --- |
| # Comment/uncomment the following line to disable/enable debugging  DEBUG = y  # Add your debugging flag (or not) to CFLAGS  ifeq ($(DEBUG),y)  DEBFLAGS = -O -g -DSCULL\_DEBUG # "-O" is needed to expand inlines  else  DEBFLAGS = -O2  endif  CFLAGS += $(DEBFLAGS) |

### 4.2.5 速率限制

为了避免printk重复输出过快而阻塞系统，最好就是设置一个标志，说明已经报告过这个消息了，不要再打印相同的信息了。内核提供了以下函数：

|  |
| --- |
| int printk\_ratelimit(void); |

在打印经常重复的消息时，调用次函数。如果函数返回非0值，执行打印消息；否则，不执行打印。如下所示：

|  |
| --- |
| if (printk\_ratelimit( ))  printk(KERN\_NOTICE "The printer is still on fire\n"); |

printk\_ratelimit的工作原理就是，追踪发送到控制台的消息的数量，如果达到一定的阈值，那么printk\_ratelimit返回0，导致消息不再打印。

可以通过修改/proc/sys/kernel/printk\_ratelimit（重新使能消息发送之前等待的秒数）和/proc/sys/kernel/printk\_ratelimit\_burst（修改其阈值）来定制printk\_ratelimit的行为。

### 4.2.6 打印设备编号

有时候，可能需要打印感兴趣的硬件设备编号。打印主、次设备编号很简单，但是为了保持一致性，内核为此提供了2个宏，定义在<linux/kdev\_t.h>文件中。

|  |
| --- |
| #define print\_dev\_t(buffer, dev);  #define format\_dev\_t(buffer, dev); |

两个宏都是将设备编号打印到给定的缓冲区中；唯一的区别就是print\_dev\_t返回打印的字符数，而format\_dev\_t返回缓冲区。

## 4.3 使用查询调试

驱动程序开发者可以使用一些技术来查询系统：在/proc文件系统中创建文件，使用驱动程序的ioctl方法以及通过sysfs导出属性等。使用sysfs需要有相当的驱动程序模型的背景知识。这将在第14章讨论。

### 4.3.1 使用/proc文件系统

/proc文件系统是一个特殊的，由软件创建的文件系统，内核用它将信息导出到外界。/proc下的每个文件都与内核函数绑定在一起，当用户读取其中的文件时，该函数动态地生成文件的“内容”。例如，/proc/modules总是返回当前已加载模块的列表。

Linux系统中，大量使用/proc。许多工具，比如ps，top，uptime，都是通过/proc获取它们所需的信息。设备驱动程序当然也可以从/proc导出信息。/proc文件系统是动态的，因此模块程序可以随时添加、删除文件。

具有完整功能的/proc文件是非常复杂的，但/proc文件可以读也可以写。但是大部分时候，/proc中的文件是只读的。本段假设都是只读的。如果想要实现更为复杂的/proc文件，请参阅linux内核源代码。

但是，/proc下添加文件是不鼓励的。推荐使用方式是sysfs。但是，使用sysfs需要了解Linux设备模型，这个到第14章，我们才会讨论。同时，在/proc中创建文件又是非常简易的，且符合调试的目的，因此，我们先讨论它。

#### 4.3.1.1 /proc实现文件

所有使用/proc的模块，必须包含头文件<linux/proc\_fs.h>，用以定义正确的函数。

为了创建一个只读的/proc文件，驱动程序必须实现一个函数，当该文件被读时，产生数据。当某个进程读这个文件是（调用read系统调用），请求通过该函数送达驱动模块。那么，我们首先来看一下这个函数，稍后再看其注册接口。

当一个进程从你的/ proc文件中读取时，内核会分配一页内存（即PAGE\_SIZE字节），驱动程序可以在其中写入要返回给用户空间的数据。该缓冲区被传递给名为read\_proc的函数：

|  |
| --- |
| int (\*read\_proc)(char \*page, char \*\*start, off\_t offset, int count, int \*eof, void \*data); |

指针page是要写入数据的缓冲区；函数使用start参数来说明感兴趣的数据在页面中写入的位置（稍后会详细介绍）；offset和count的含义与read方法相同。eof参数是指向整数的指针，驱动程序必须设置该整数以表明它没有更多数据要返回，而data是指向驱动程序特定数据的指针，可用于用户记录。

这个函数返回写入到page缓冲区内的数据的字节数，就像read方法一样。eof是一个简单的标志，但是start的使用在某种程度上更为复杂；它的目的就是帮助实现更大的/proc文件（大于一页）。

参数start有一些非常规的用途。它的目的是指出要在哪里（页面内）找到要返回给用户的数据。当你的proc\_read方法被调用时，\*start设为NULL。如果将其设为NULL，则内核假定数据从偏移量为0的地址开始写入，就像参数offset等于0一样；换句话说，就是忽略掉参数offset，不论其值为多少。相反，如果将\*start设为非NULL值，则内核假定\*start+offset为开始地址，然后将数据返回给用户。一般来说，只有少量的数据的话，直接忽略掉参数start就好。更复杂的方法考虑start和offset参数配合使用。

There has long been another major issue with /proc files, which start is meant to solve as well. Sometimes the ASCII representation of kernel data structures changes between successive calls to read, so the reader process could find inconsistent data from one call to the next. If \*start is set to a small integer value, the caller uses it to increment filp->f\_pos independently of the amount of data you return, thus making f\_pos an internal record number of your read\_proc procedure. If, for example, your read\_proc function is returning information from a big array of structures, and five of those structures were returned in the first call, \*start could be set to 5. The next call provides that same value as the offset; the driver then knows to start returning data from the sixth structure in the array. This is acknowledged as a “hack” by its authors and can be seen in fs/proc/generic.c.

/ proc文件长期以来一直是另一个主要问题，这也意味着要解决这个问题。 有时内核数据结构的ASCII表示会在连续的读取调用之间发生变化，因此读者进程可能会发现从一次调用到另一次调用的不一致数据。 如果\* start被设置为一个小整数值，调用者使用它来增加filp-> f\_pos，而与您返回的数据量无关，从而使f\_pos成为read\_proc过程的内部记录号。 例如，如果您的read\_proc函数正在从大型结构数组中返回信息，并且在第一次调用中返回了其中五个结构，则可以将\* start设置为5.下一次调用将提供与偏移量相同的值; 然后驱动程序知道开始从数组中的第六个结构返回数据。 这被它的作者公认为“黑客”，可以在fs / proc / generic.c中看到。

Note that there is a better way to implement large /proc files; it’s called seq\_file, and we’ll discuss it shortly. First, though, it is time for an example. Here is a simple (if somewhat ugly) read\_proc implementation for the scull device:

请注意，有更好的方法来实现大型/ proc文件; 它被称为seq\_file，我们很快就会讨论它。 首先，这是一个例子。 这是一个简单的（如果有点难看的）scull设备的read\_proc实现：

|  |
| --- |
| int scull\_read\_procmem(char \*buf, char \*\*start, off\_t offset, int count,  int \*eof, void \*data)  {  int i,j,len=0;  int limit = count - 80; /\* 打印不能超过这个大小 \*/  for (i = 0; i < scull\_nr\_devs && len <= limit; i++)  {  struct scull\_dev \*d = &scull\_devices[i];  struct scull\_qset \*qs = d->data;  if (down\_interruptible(&d->sem))  {  return -ERESTARTSYS;  }  len += sprintf(buf+len,"\nDevice %i: qset %i, q %i, sz %li\n",  i, d->qset, d->quantum, d->size);  for (; qs && len <= limit; qs = qs->next) /\* scan the list \*/  {  len += sprintf(buf + len, " item at %p, qset at %p\n",  qs, qs->data);  if (qs->data && !qs->next) /\* dump only the last item \*/  {  for (j = 0; j < d->qset; j++)  {  if (qs->data[j])  {  len += sprintf(buf + len," % 4i: %8p\n",  j, qs->data[j]);  }  }  }  }  up(&scull\_devices[i].sem);  }  \*eof = 1;  return len;  } |

This is a fairly typical read\_proc implementation. It assumes that there will never be a need to generate more than one page of data and so ignores the start and offset values. It is, however, careful not to overrun its buffer, just in case.

这是一个相当典型的read\_proc实现。 它假定永远不会需要生成多个页面的数据，因此忽略开始和偏移值。 但是，小心不要超出缓冲区，以防万一。

#### 4.3.1.2 旧接口

如果您通读内核源代码，则可能会遇到使用较旧接口实现/proc文件的代码：

|  |
| --- |
| int (\*get\_info)(char \*page, char \*\*start, off\_t offset, int count); |

所有参数与read\_proc具有相同的含义，但缺少eof和data参数。新代码应该使用read\_proc接口。

#### 4.3.1.3 创建自己的/proc文件

Once you have a read\_proc function defined, you need to connect it to an entry in the /proc hierarchy. This is done with a call to create\_proc\_read\_entry:

一旦定义了read\_proc函数，就需要将其连接到/ proc层次结构中的条目。 这是通过调用create\_proc\_read\_entry完成的：

|  |
| --- |
| struct proc\_dir\_entry \*create\_proc\_read\_entry(const char \*name, mode\_t mode,  struct proc\_dir\_entry \*base, read\_proc\_t \*read\_proc, void \*data); |

Here, name is the name of the file to create, mode is the protection mask for the file (it can be passed as 0 for a system-wide default), base indicates the directory in which the file should be created (if base is NULL, the file is created in the /proc root), read\_proc is the read\_proc function that implements the file, and data is ignored by the kernel (but passed to read\_proc). Here is the call used by scull to make its /proc function available as /proc/scullmem:

这里，name是要创建的文件的名称，mode是文件的保护掩码（对于系统范围的默认值，它可以作为0传递），base指示文件应该创建的目录（如果base是 NULL，该文件在/ proc root中创建），read\_proc是实现该文件的read\_proc函数，数据被内核忽略（但传递给read\_proc）。 下面是scull使用它的/ proc函数作为/ proc / scullmem的调用：

|  |
| --- |
| create\_proc\_read\_entry("scullmem", 0 /\* default mode \*/, NULL /\* parent dir \*/, scull\_read\_procmem,  NULL /\* client data \*/); |

Here, we create a file called scullmem directly under /proc, with the default, world readable protections.

在这里，我们直接在/ proc下创建一个名为scullmem的文件，其默认的世界可读保护。

The directory entry pointer can be used to create entire directory hierarchies under /proc. Note, however, that an entry may be more easily placed in a subdirectory of /proc simply by giving the directory name as part of the name of the entry—as long as the directory itself already exists. For example, an (often ignored) convention says that /proc entries associated with device drivers should go in the subdirectory driver/; scull could place its entry there simply by giving its name as driver/scullmem.

目录项指针可用于在/ proc下创建整个目录层次结构。 但是，请注意，只要目录本身已经存在，仅通过将目录名称作为条目名称的一部分，就可以将条目更容易地放置在/ proc的子目录中。 例如，一个（经常被忽略的）约定说与设备驱动程序相关的/ proc条目应该放在子目录driver /中; scull可以通过简单的命名为driver / scullmem来进入。

Entries in /proc, of course, should be removed when the module is unloaded. remove\_proc\_entry is the function that undoes what create\_proc\_read\_entry already did:

当模块卸载时，/ proc中的条目当然应该被删除。 remove\_proc\_entry是取消create\_proc\_read\_entry已经做了什么的函数：

|  |
| --- |
| remove\_proc\_entry("scullmem", NULL /\* parent dir \*/); |

Failure to remove entries can result in calls at unwanted times, or, if your module has been unloaded, kernel crashes.

When using /proc files as shown, you must remember a few nuisances of the implementation—no surprise its use is discouraged nowadays.

如果未能删除条目，可能会导致在不需要的时间进行调用，或者如果您的模块已被卸载，则会导致内核崩溃。

如图所示使用/ proc文件时，您必须记住一些实施的麻烦 - 现在并不让人惊讶它的使用受到阻碍。

The most important problem is with removal of /proc entries. Such removal may well happen while the file is in use, as there is no owner associated to /proc entries, so using them doesn’t act on the module’s reference count. This problem is simply triggered by running sleep 100 < /proc/myfile just before removing the module, for example.

最重要的问题是删除/ proc条目。 当文件被使用时，这种移除可能会发生，因为没有所有者与/ proc条目相关联，所以使用它们不会影响模块的引用计数。 例如，仅在移除模块之前运行睡眠100 </ proc / myfile>即可触发此问题。

Another issue is about registering two entries with the same name. The kernel trusts the driver and doesn’t check if the name is already registered, so if you are not careful you might end up with two or more entries with the same name. This is a problem known to happen in classrooms, and such entries are indistinguishable, both when you access them and when you call remove\_proc\_entry.

另一个问题是关于注册两个具有相同名称的条目。 内核信任驱动程序，并且不检查名称是否已经注册，所以如果你不小心，可能会得到两个或多个具有相同名称的条目。 这是已知的在教室中发生的问题，并且这些条目在您访问它们和调用remove\_proc\_entry时无法区分。

4.3.1.4 The seq\_file interface

|  |
| --- |
| Device 0: qset 1000, q 4000, sz 6906  item at c3ba3ce0, qset at c3bc0000  0: c3bc1000  1: c3bc2000  Device 1: qset 1000, q 4000, sz 0  Device 2: qset 1000, q 4000, sz 0  Device 3: qset 1000, q 4000, sz 0  scull:  dptr:c3b82060  scull:  dptr->data:c3b88000  scull:  dptr->data[s\_pos]:c3b89000  scull:  dptr:c3b82060  scull:  dptr->data[s\_pos]:c3b8a000  scull:  dptr:c3b82060  Device 0: qset 1000, q 4000, sz 7626  item at c3b82060, qset at c3b88000  0: c3b89000  1: c3b8a000  Device 1: qset 1000, q 4000, sz 0  Device 2: qset 1000, q 4000, sz 0  Device 3: qset 1000, q 4000, sz 0 |

### 4.3.2 使用ioctl方法

ioctl是作用域文件描述符之上的一个系统调用，我们将会在第六章介绍它的用法。ioctl接受一个命令及一个（可选的）参数，通常是一个指针。作为替代/proc文件系统的方法，我们可以专为调试设计若干ioctl命令。这些命令从驱动程序复制相关的数据到用户空间中，然后可在用户空间中检验这些数据。

使用ioctl这种方式获取信息比使用/proc更难，因为您需要另一个程序来调用ioctl并显示结果。 这个程序必须被编写，编译，并保持与你正在测试的模块同步。 另一方面，驱动程序代码可能比实现/ proc文件所需的代码更容易。

使用ioctl获取信息比使用/proc更难，因为需要有程序调用ioctl并显式结果。我们必须编写和编译这个程序，且要与测试的模块保持同步。另一方面，驱动程序侧的代码要比/proc文件实现起来容易一些。

ioctl方法的另一个有趣的优点是，即使在调试被禁用时，获取信息的命令也可以留在驱动程序中。不像/proc文件对人任何查看该文件的人都可见，未使用文档说明的ioctl命令很可能不被注意。此外，万一驱动程序有什么异常，这些命令仍然可以用来调试。唯一的缺点是模块程序会稍大一些。

## 4.4 使用strace

strace命令是一个强大的工具，可显示用户空间程序发出的所有系统调用。它不仅显示调用，还可以以符号形式显示调用的参数及其返回值。当系统调用失败时，将显示错误的符号值（例如ENOMEM）和相应的字符串（out of memory）。strace有许多命令行选项；其中最有用的是-t显示执行调用的时间，-T显示在调用中执行的时间，-e限制追踪的调用类型，-o将输出重定向到一个文件。默认情况下，strace在stderr上打印追踪信息。

strace 从内核自身获取信息。这意味着可以追踪一个程序，不管它是否带有调试支持编译(对 gcc是-g选项)以及不管它是否被strip过。此外，你也可以追踪一个正在运行中的进程，这类似于调试器连接到一个运行中的进程并控制它。

跟踪信息常用来支持发给应用程序开发者的故障报告，但是对内核程序员也是很有价值的。我们已经看到驱动代码运行如何响应系统调用，strace 允许我们检查每个调用的输入和输出数据的一致性。

例如，运行命令strace ls /dev > /dev/scull0的最后几行：

|  |
| --- |
| open("/dev", O\_RDONLY|O\_NONBLOCK|O\_LARGEFILE|O\_DIRECTORY) = 3  fstat64(3, {st\_mode=S\_IFDIR|0755, st\_size=24576, ...}) = 0  fcntl64(3, F\_SETFD, FD\_CLOEXEC) = 0  getdents64(3, /\* 141 entries \*/, 4096) = 4088  [...]  getdents64(3, /\* 0 entries \*/, 4096) = 0  close(3) = 0  [...]  fstat64(1, {st\_mode=S\_IFCHR|0664, st\_rdev=makedev(254, 0), ...}) = 0  write(1, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 4096) = 4000  write(1, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 96) = 96  write(1, "b\nptyxc\nptyxd\nptyxe\nptyxf\nptyy0\n"..., 4096) = 3904  write(1, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 192) = 192  write(1, "\nvcs47\nvcs48\nvcs49\nvcs5\nvcs50\nvc"..., 673) = 673  close(1) = 0  exit\_group(0) = ? |

从第一个 write 调用看，明显地，在 ls 结束查看目标目录后，它试图写 4KB。但奇怪的是，只有 4000 字节被成功写入, 并且操作被重复。但当我们查看scull 中的写实现，发现它一次最多只允许写一个quantum（共4000字节），可见驱动本来就是期望部分写。几步之后, 所有东西清空, 程序成功退出。正是通过strace的输出，使我们确信驱动的部分写功能运行正确。

作为另一个例子，让我们读一下scull设备（使用wc命令）：

|  |
| --- |
| [...]  open("/dev/scull0", O\_RDONLY|O\_LARGEFILE) = 3  fstat64(3, {st\_mode=S\_IFCHR|0664, st\_rdev=makedev(254, 0), ...}) = 0  read(3, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 16384) = 4000  read(3, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 16384) = 4000  read(3, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 16384) = 865  read(3, "", 16384) = 0  fstat64(1, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(136, 1), ...}) = 0  write(1, "8865 /dev/scull0\n", 17) = 17  close(3) = 0  exit\_group(0) = ? |

正如所料，读取一次只能检索到4000个字节，但总数据量与前面示例中写入的数据量相同。 有趣的是要注意在这个例子中如何组织重试，与之前的跟踪相反。 wc针对快速读取进行了优化，因此绕过了标准库，试图通过一次系统调用来读取更多数据。 您可以从跟踪中的读取行中看到wc如何尝试一次读取16 KB。

如同期望的，read一次只能获取4000字节，但是数据总量等同于前个例子写入的。这个例子，意外的收获是：可以肯定，wc为快速读进行了优化，它因此绕过了标准库（没有使用fscanf），而是直接一个系统调用以读取更多数据。这一点，可从跟踪到的读的行里看到wc一次试图读取16 KB的数据而确认。

## 4.5 调试系统故障

即使使用了所有的监控和调试手段，驱动程序中也难免会存在bug，导致在运行时系统崩溃。那么发生这种情况时，尽可能多的收集有用的信息去解决问题就至关重要了。

当然了linux代码足够健壮，能够从容地应付大多数的错误。通常，故障只会导致当前进程的销毁，而系统继续运行。如果故障发生在进程的上下文之外，或者，如果系统的关键部分收到损害，那么系统可能会发生混乱。但是当驱动发生错误的时候，一般就是调用该驱动的进程突然死亡。当进程被销毁时，唯一不可恢复的损害就是分配给进程上下文的动态内存可能丢失。但是，因为当进程“死掉”的时候，内核会为每个打开的设备调用close操作，所以驱动还是会释放掉由open方法分配的内存。

即使oop不会搞垮整个系统，但是有时候，你会发现不得不重启系统才能继续工作。有问题的驱动程序有可能会让硬件保留在未使用的状态，导致内核资源处于不一致的状态，随机损毁内存等。通常，发生oop时，可以通过卸载驱动，重新来过就好了。但是，如果发生系统不好的情况时，最好的办法就是重启。

我们已经说过，当内核代码行为异常时，会往控制台上打印一条告知性的消息。下面的内容就是怎样理解和使用这些消息。尽管对于新手来说，它们看上去毫无头绪，但是内核dump了所有的有趣的信息，通常情况下，能够查明程序错误而无需额外的测试。

### 4.5.1 查看oop消息

很多bug就是引用了NULL指針或者其它不正确的指针。它们通常的结果就是发生oop消息。

处理器所使用的地址几乎都是虚拟地址，通过复杂的页表结构映射到真实的物理地址上（例外就是内存管理子系统本身使用的物理地址）。适用错误的指针，分页机制无法将指针映射到物理地址上，就会向操作系统发送一个“页错误”。如果地址无效，内核不能“导入”丢失的地址；如果在处理器处于管理员模式时，发生这种情况，它（通常）会生成一个oop消息。

一条oop消息展示了故障发生时处理器的状态，包括CPU寄存器的内容和其它似乎很难理解的信息。该条消息是通过fault处理函数的printk语句打印出来的，代码位于arch/\*/kernel/traps.c文件中。

让我们来看这样一条消息。运行在linux2.6内核上的NULL指针的引用导致的错误。最相关的信息就是指令指针（EIP），错误指令的地址。

|  |
| --- |
| Unable to handle kernel NULL pointer dereference at virtual address 00000000  printing eip:  d083a064  Oops: 0002 [#1]  SMP  CPU: 0  EIP: 0060:[<d083a064>] Not tainted  EFLAGS: 00010246 (2.6.6)  EIP is at faulty\_write+0x4/0x10 [faulty]  eax: 00000000 ebx: 00000000 ecx: 00000000 edx: 00000000  esi: cf8b2460 edi: cf8b2480 ebp: 00000005 esp: c31c5f74  ds: 007b es: 007b ss: 0068  Process bash (pid: 2086, threadinfo=c31c4000 task=cfa0a6c0)  Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460  fffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480  00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005  Call Trace:  [<c0150558>] vfs\_write+0xb8/0x130  [<c0150682>] sys\_write+0x42/0x70  [<c0103f8f>] syscall\_call+0x7/0xb  Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0 |

该消息是写入faulty模块所拥有的设备时产生的。该模块就是专门验证oop消息的，而且faulty.c文件中的write方法实现也很简短：

|  |
| --- |
| ssize\_t faulty\_write (struct file \*filp, const char \_\_user \*buf, size\_t count,  loff\_t \*pos)  {  /\* make a simple fault by dereferencing a NULL pointer \*/  \*(int \*)0 = 0;  return 0;  } |

正如你所看到的，我们所做的就是解引用一个NULL指针。因为0不是一个合法的指针值，故障就会发生，内核就会产生前面所显示的oop消息。然后，调用进程被杀死。

模块fault在其read方法中具有不同的发生故障的条件：

|  |
| --- |
| ssize\_t faulty\_read(struct file \*filp, char \_\_user \*buf, size\_t count, loff\_t \*pos)  {  int ret;  char stack\_buf[4];  memset(stack\_buf, 0xff, 20); /\* 尝试一下数组溢出 \*/  if (count > 4)  count = 4; /\* copy 4 bytes to the user \*/  ret = copy\_to\_user(buf, stack\_buf, count);  if (!ret)  return count;  return ret;  } |

该方法把一个字符串拷贝到局部变量中；不幸的是，字符串比数组长。当函数调用返回时，数组溢出就会产生oop消息。因为return返回时，指令指针不知会指向哪里，所以这类故障很难追踪，你可以得到如下的信息：

|  |
| --- |
| EIP: 0010:[<00000000>]  Unable to handle kernel paging request at virtual address ffffffff  printing eip:  ffffffff  Oops: 0000 [#5]  SMP  CPU: 0  EIP: 0060:[<ffffffff>] Not tainted  EFLAGS: 00010296 (2.6.6)  EIP is at 0xffffffff  eax: 0000000c ebx: ffffffff ecx: 00000000 edx: bfffda7c  esi: cf434f00 edi: ffffffff ebp: 00002000 esp: c27fff78  ds: 007b es: 007b ss: 0068  Process head (pid: 2331, threadinfo=c27fe000 task=c3226150)  Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286 cf434f00 fffffff7  bfffda70 c27fe000 c0150612 cf434f00 bfffda70 00002000 cf434f20 00000000  00000003 00002000 c0103f8f 00000003 bfffda70 00002000 00002000 bfffda70  Call Trace:  [<c0150612>] sys\_read+0x42/0x70  [<c0103f8f>] syscall\_call+0x7/0xb  Code: Bad EIP value. |

在这个例子中，我们只看到调用堆栈的一部分（vfs\_read和faulty\_read丢失），并且内核发出“Bad EIP value”的错误。这个错误信息和开始处违规的地址都暗示着内核堆栈已经被破坏。

一般来说，当你遇到一个oop消息时，首先要做的是看问题发生的位置，在上面显示的第一个oops中，相关行是：

|  |
| --- |
| EIP is at faulty\_write+0x4/0x10 [faulty] |

在这儿，我们看到位于模块faulty的函数faulty\_write中（模块名称faulty使用方括号[]括起来了）。用16进制表明指令指针在函数里占用的字节数（比如这里就是4个字节）和指令总数（这儿，16/4，就是4。所以，从函数faulty\_write开始处，执行4条指令的地方就是发生错误的地方）。通常这就足够找到问题所在了。

如果需要更多信息，调用的堆栈会告诉程序分崩离析的原因。堆栈内容是以Hex形式打印的。通过一些工作，可以从堆栈列表中确定局部变量和函数参数的值。有经验的内核开发人员可以从中受益；例如，查看faulty\_read的oop消息中的堆栈列表：

|  |
| --- |
| Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286 cf434f00 fffffff7  bfffda70 c27fe000 c0150612 cf434f00 bfffda70 00002000 cf434f20 00000000  00000003 00002000 c0103f8f 00000003 bfffda70 00002000 00002000 bfffda70 |

栈顶的0xffffffff就是那个导致程序故障的字符串的一部分。默认情况下，x86架构上，用户空间的堆栈是从0xc0000000开始的；因而，重复出现的0xbfffda70正好是用户空间的堆栈地址；事实上，它是传递给read系统调用的buffer地址，在内核调用链中，每次传递都会被拷贝一次。同样，x86架构上，内核空间是从0xc0000000开始，所以上面的很多地址是内核地址。

最后，在查看oops列表时，始终要注意本章开头讨论的分配内存时的初始值“0xa5”。因此，举个例子，如果得到一个内核oop消息，其中有问题的地址是0xa5a5a5a5，几乎肯定是在某处忘记初始化动态内存。

值得注意的是，上面的符号化调用堆栈只有当你打开CONFIG\_KALLSYMS内核选项编译内核时才会看到。否则，将会看到一个原始的16进制列表，除非有其它的解码方式，否则没啥太大用处。

### 4.5.2 系统挂起

尽管内核中的大多数bug都会报告oop消息，但有时候也会挂起系统，没有任何消息打印。比如说，如果代码中有一个死循环，内核停止调度，系统不会相应任何动作，包括神奇的Ctrl-Alt-Del组合按键。处理系统挂起有两种选择，要么事前避免，要么事后能够debug代码。

可以通过在关键点插入schedule调用来阻止死循环。schedule调用能够调用调度器，所以允许其它进程从现在的进程中抢占CPU时间。如果因为驱动的bug，导致某一个进程在内核空间中循环，schedule调用保证能够终止进程。

必须意识到，任何对schedule的调用都会给驱动程序带来可重入的问题，因为允许了其它进程运行。如果在驱动中使用了恰当的锁定的话，可重入就不会带来问题。但是，如果驱动程序持有自旋锁的话，不论什么时候，都不要调用schedule。

如果驱动程序确实挂起了系统，但并不知道在哪插入schedule调用，最好的方法就是添加一些print语句，输出信息到控制台上。

有时候系统看上去像是挂起了，但实际上并没有。显示器上的时钟或系统负荷表就是一个很好的状态监视器；只要其继续更新，schedule就在工作。

### 4.5.3 ARM9架构下查看oop消息

1. 硬件环境参考3.8.1节

2. 编写fault.c文件，文件内容如下：

|  |
| --- |
| //#include <linux/config.h>  #include <linux/module.h>  #include <linux/init.h>  #include <linux/kernel.h> /\* printk() \*/  #include <linux/fs.h> /\* everything... \*/  #include <linux/types.h> /\* size\_t \*/  #include <asm/uaccess.h>  MODULE\_LICENSE("Dual BSD/GPL");  int faulty\_major = 0;  ssize\_t faulty\_read(struct file \*filp, char \_\_user \*buf,  size\_t count, loff\_t \*pos)  {  int ret;  char stack\_buf[4];  /\* Let's try a buffer overflow \*/  memset(stack\_buf, 0xff, 20);  if (count > 4)  count = 4; /\* copy 4 bytes to the user \*/  ret = copy\_to\_user(buf, stack\_buf, count);  if (!ret)  return count;  return ret;  }  ssize\_t faulty\_write (struct file \*filp, const char \_\_user \*buf, size\_t count,  loff\_t \*pos)  {  /\* make a simple fault by dereferencing a NULL pointer \*/  \*(int \*)0 = 0;  return 0;  }  struct file\_operations faulty\_fops = {  .read = faulty\_read,  .write = faulty\_write,  .owner = THIS\_MODULE  };  int faulty\_init(void)  {  int result;  /\*  \* Register your major, and accept a dynamic number  \*/  result = register\_chrdev(faulty\_major, "faulty", &faulty\_fops);  if (result < 0)  return result;  if (faulty\_major == 0)  faulty\_major = result; /\* dynamic \*/  return 0;  }  void faulty\_cleanup(void)  {  unregister\_chrdev(faulty\_major, "faulty");  }  module\_init(faulty\_init);  module\_exit(faulty\_cleanup); |

编译内核模块faulty.ko。将其使用命令insmod或modprobe加载进内核。使用下面的命令

# cat /proc/devices

使用下面生成设备文件，

|  |
| --- |
| mknod /dev/faulty c 251 0 #生成设备文件 |

然后操作，

# echo 1 > /dev/faulty

就会发生下面这样的oop消息

|  |
| --- |
| 1 Unable to handle kernel NULL pointer dereference at virtual address 00000000  2 pgd = c3bdc000  3 [00000000] \*pgd=33bf2831, \*pte=00000000, \*ppte=00000000  4 Internal error: Oops: 817 [#1]  5 Modules linked in: faulty [last unloaded: faulty]  6 CPU: 0 Not tainted (3.3.7 #3)  7 PC is at faulty\_write+0x4/0xc [faulty]  8 LR is at vfs\_write+0xb8/0x14c  9 pc : [<bf004004>] lr : [<c00824e4>] psr: a0000013  10 sp : c3be1f68 ip : bf004000 fp : 0075b2a0  11 r10: 00000000 r9 : c3be0000 r8 : c0009344  12 r7 : c3be1f88 r6 : 0075cf38 r5 : 00000002 r4 : c3bd3f80  13 r3 : c3be1f88 r2 : 00000002 r1 : 0075cf38 r0 : 00000000  14 Flags: NzCv IRQs on FIQs on Mode SVC\_32 ISA ARM Segment user  15 Control: c0007177 Table: 33bdc000 DAC: 00000015  16 Process sh (pid: 802, stack limit = 0xc3be0270)  17 Stack: (0xc3be1f68 to 0xc3be2000)  18 1f60: 00000000 00000000 c3bd3f80 00000000 00000000 00000004  19 1f80: c0009344 c0082774 00000000 00000000 0075cf6d 00000000 000a4a70 00000002  20 1fa0: 0075cf38 c00091c0 000a4a70 00000002 00000001 0075cf38 00000002 00000000  21 1fc0: 000a4a70 00000002 0075cf38 00000004 00000001 bec1f76c 00000020 0075b2a0  22 1fe0: 00000001 bec1f738 0000e228 b6fd363c 60000010 00000001 00000000 00000000  23 [<bf004004>] (faulty\_write+0x4/0xc [faulty]) from [<c00824e4>] (vfs\_write+0xb8/0x14c)  24 [<c00824e4>] (vfs\_write+0xb8/0x14c) from [<c0082774>] (sys\_write+0x40/0x74)  25 [<c0082774>] (sys\_write+0x40/0x74) from [<c00091c0>] (ret\_fast\_syscall+0x0/0x2c)  26 Code: bad PC value  27 ---[ end trace 49756d2b10a033ac ]---  28 |

我们现在逐行分析上面的这条oop消息：

* 第1行，报告出错的原因；
* 第2-4行，oop信息序号；
* 第5行，出错时内核已加载模块；
* 第6行，发生错误的CPU序号；
* 第7-15行，发生错误的位置，以及当时CPU各个寄存器的值，这最有利于我们找出问题所在；
* 第16行，当前进程的名字及进程ID；
* 第17-22行，出错时栈内的内容；
* 第23-25行，栈回溯信息，出错时的函数递进调用关系（确保CONFIG\_FRAME\_POINTER被定义）

反汇编faulty.ko（~/qemu/buildroot-2012.05/output/host/usr/bin/arm-linux-objdump -D faulty.ko > faulty.dis），查看faulty.dis文件内容（cat faulty.dis），如下：

|  |
| --- |
| 00000000 <faulty\_write>:  0: e3a00000 mov r0, #0  4: e5800000 str r0, [r0]  8: e12fff1e bx lr |

pc寄存器和lr寄存器的内容：

|  |
| --- |
| pc : [<bf004004>] lr : [<c00824e4>] psr: a0000013 |

堆栈的回溯过程数据：

|  |
| --- |
| 23 [<bf004004>] (faulty\_write+0x4/0xc [faulty]) from [<c00824e4>] (vfs\_write+0xb8/0x14c)  24 [<c00824e4>] (vfs\_write+0xb8/0x14c) from [<c0082774>] (sys\_write+0x40/0x74)  25 [<c0082774>] (sys\_write+0x40/0x74) from [<c00091c0>] (ret\_fast\_syscall+0x0/0x2c) |

出错代码是faulty\_write函数中的第1条指令（(0xbf004004-0xbf004004)/4 + 1），该函数的首地址是0xbf004004，该函数总共3条指令（0xc/0x4=3）。该函数由位于0xc00824e4地址处的函数vfs\_write调用，所以返回的地址是0xc00824e4，这一点可以从寄存器lr的值等于0xc00824e4看出。调用该函数的指令是vfs\_write的第49条（0xb8/4=46）指令。

所以到达出错地方的函数调用流程是：write（用户空间的系统调用）-> sys\_write -> vfs\_write -> faulty\_write。

## 4.6 调试器和相关工具

### 4.6.1 使用gdb

### 4.6.2 kdb内核调试器（仅IA-32（x86））

### 4.6.3 kgdb补丁

### 4.6.4 用户模式的Linux虚拟机

### 4.6.5 Linux跟踪工具包

### 4.6.6 动态探测

# 第5章 并发和竞态

并发问题是操作系统编程中的核心问题之一。并发时相关bug易发生，难解决。所以，驱动程序开发者必须在开始就把并发问题考虑在内。现在让我们看一下，之前我们简单编写的scull驱动程序是否存在潜在的问题。如果有，想出解决办法。

## 5.1 scull的缺陷

让我们看一下scull模块中内存管理相关代码。下面这段代码是write函数实现中的一段代码。Scull代码需要判断它所申请的内存是否被分配。

|  |
| --- |
| if (!dptr->data[s\_pos]) {  dptr->data[s\_pos] = kmalloc(quantum, GFP\_KERNEL);  if (!dptr->data[s\_pos])  goto out;  } |

这段代码，咋看上没有什么问题。但是，假设我们的系统中存在2个彼此独立的进程（假设分别为进程A和B），在某一个时刻，同时访问scull设备中的同一偏移的区域，会发生什么呢？假设2个进程都同时访问到第一个if条件语句处，而此时判断的结果恰好是Null，那么它们都会调用kmalloc()函数，申请分配内存，然后将返回的内存指针赋给dptr->data[s\_pos]，这时，如果A先赋值，那么B赋值时，就会覆盖掉A之前的赋值。那么A申请的内存，就会发生内存泄漏，系统无法找到了。

这就是竞态条件的一个例证。竞态条件就是对共享数据的访问不加控制的结果。竞态条件可能会造成系统崩溃，数据损坏，以及安全问题等。大家可能认为，竞态的发生是一个概率非常非常低的事情，但是，一旦发生，后果就是极其严重的。

让我们再消除scull中的竞态问题前，让我们更全面地对并发做一下了解吧。

## 5.2 并发及其管理

在现代操作系统中，并发的来源有很多种。多个运行中的用户进行，可能以不可思议的组合方式访问你的代码；SMP系统可以在不同的CPU上同时执行你的代码；内核代码是可抢占式的，你的驱动程序可能会在任何时候被打断，而替代的进程可能正在运行你的驱动程序；设备硬件中断是异步发生的，可能会造成你的代码同时执行。内核提供了多种延时代码执行的机制、诸如工作队列、tasklet、和定时器，它们可以让你的代码在任何时候运行，而不关注当前的进程正在做什么。另外，现在大多数系统都支持“热插拔”机制，也就是说，在你使用设备期间，它随时有可能消失。

避免竞态条件的出现是一个艰难的任务。实践证明，竞态条件可以通过多思考、使用内核并发控制原语、和一些基本原则的应用进行避免。接下来，我们首先介绍这些原则，而后再研究如何应用它们。

**原则一：尽量避免使用共享资源**。

竞态条件就是因对资源的共享访问而产生的。所以，第一条经验准则就是牢记避免在你的驱动程序中使用共享资源。没有并发访问，也就没有竞态条件。如果非得在程序中使用共享资源，必须有“不得不这样做”的理由。

但是，实际情况就是，共享又经常需要。当遇上了，我们就得面对：

**原则二：任何时间，发生对硬件或软件资源的访问，而有可能造成竞态条件时，我们必须显式地控制对这些资源的访问。**

在scull的示例中，我们必须控制对scull的数据结构的访问。我们需要让代码知道，是否已经分配内存，还要让代码知道，分配的内存是“其它人”的。这种访问管理的常用方法就是加锁和互斥-保证同时只有一个线程在控制共享的资源。

下面让我们专注于分析锁的使用。在此之前，还有一个重要的原则，我们必须作一简单的介绍：内核代码创建一个与其它代码共享对象，在对该对象所有的引用解除之前，其必须一直存在。这条规则带来了2个要求：内核的对象在内核没有处于可运行状态时，必须是不可用的；对该对象的引用必须被追踪。在大部分时候，引用计数的处理都是由内核完成，但是，万事无绝对。所以，我们必须非常注重细节的实现。

## 5.3 信号量和互斥

So let us look at how we can add locking to scull. Our goal is to make our operations on the scull data structure atomic, meaning that the entire operation happens at once as far as other threads of execution are concerned. For our memory leak example, we need to ensure that if one thread finds that a particular chunk of memory must be allocated, it has the opportunity to perform that allocation before any other thread can make that test. To this end, we must set up critical sections: code that can be executed by only one thread at any given time.

Not all critical sections are the same, so the kernel provides different primitives for different needs. In this case, every access to the scull data structure happens in process context as a result of a direct user request; no accesses will be made from interrupt handlers or other asynchronous contexts. There are no particular latency (response time) requirements; application programmers understand that I/O requests are not usually satisfied immediately. Furthermore, the scull is not holding any other critical system resource while it is accessing its own data structures. What all this means is that if the scull driver goes to sleep while waiting for its turn to access the data structure, nobody is going to mind.

“Go to sleep” is a well-defined term in this context. When a Linux process reaches a point where it cannot make any further processes, it goes to sleep (or “blocks”), yielding the processor to somebody else until some future time when it can get work done again. Processes often sleep when waiting for I/O to complete. As we get deeper into the kernel, we will encounter a number of situations where we cannot sleep. The write method in scull is not one of those situations, however. So we can use a locking mechanism that might cause the process to sleep while waiting for access to the critical section.

Just as importantly, we will be performing an operation (memory allocation with kmalloc) that could sleep—so sleeps are a possibility in any case. If our critical sections are to work properly, we must use a locking primitive that works when a thread that owns the lock sleeps. Not all locking mechanisms can be used where sleeping is a possibility (we’ll see some that don’t later in this chapter). For our present needs,however, the mechanism that fits best is a semaphore.

Semaphores are a well-understood concept in computer science. At its core, a semaphore is a single integer value combined with a pair of functions that are typically called P and V. A process wishing to enter a critical section will call P on the relevant semaphore; if the semaphore’s value is greater than zero, that value is decremented by one and the process continues. If, instead, the semaphore’s value is 0 (or less),the process must wait until somebody else releases the semaphore. Unlocking a semaphore is accomplished by calling V; this function increments the value of the semaphore and, if necessary, wakes up processes that are waiting.

When semaphores are used for mutual exclusion—keeping multiple processes from running within a critical section simultaneously—their value will be initially set to 1. Such a semaphore can be held only by a single process or thread at any given time. A semaphore used in this mode is sometimes called a mutex, which is, of course, an abbreviation for “mutual exclusion.” Almost all semaphores found in the Linux kernel are used for mutual exclusion.

### 5.3.1 linux信号量实现

Linux内核提供了符合上述语义的信号量实现，尽管术语略有不同。为了使用信号量，必须包含头文件<asm/semaphore.h>。相关的数据结构是struct semaphore；实际的信号量可以通过几种方式进行声明和初始化。其中一种就是直接创建信号量，然后使用sema\_init()函数进行设置：

|  |
| --- |
| void sema\_init(struct semaphore \*sem, int val); |

在这儿，val是sem的初始值。

但是，通常情况下，信号量都是以互斥的方式使用。为了方便，内核提供了一组辅助函数和宏。因此，可使用下面宏声明一个互斥量并初始化：

|  |
| --- |
| DECLARE\_MUTEX(name);  DECLARE\_MUTEX\_LOCKED(name); |

这儿，产生了一个称为name的信号量，其初始值为0（使用DECLARE\_MUTEX\_LOCKED）或1（使用DECLARE\_MUTEX）。其值为0时，互斥量以一个被锁的状态开始；任何线程想要访问之前，必须被显式地解锁。

在运行时动态初始化互斥量时，调用下面的函数：

|  |
| --- |
| void init\_MUTEX(struct semaphore \*sem);  void init\_MUTEX\_LOCKED(struct semaphore \*sem); |

注意：在新版本的Linux内核（2.6.37之后）中，上面的函数已经不存在。

Linux中调用P函数-被称为down或者其它变体。在这儿，down指的是函数递减信号量，甚至将调用者休眠一会，直到信号量变得可用，并授予对受保护资源的访问权。这儿有3个版本的down函数：

|  |
| --- |
| void down(struct semaphore \*sem);  int down\_interruptible(struct semaphore \*sem);  int down\_trylock(struct semaphore \*sem); |

down减少信号量的值，并在需要时等待。

down\_interruptible 做相同工作，但是可被中断。我们大部分时候想要的都是可中断版本，它允许等待信号量的用户空间进程可被用户中断。作为通用规则，你不想使用非中断版本，除非没有其它选择。不可中断进程是创建不可杀进程的一种好方法（可怕的“D状态”-ps命令的结果中可以看到），但是会令用户烦恼。特别值得注意的是，使用down\_interruptible时，如果操作被中断，此时返回值为非零值，而调用者其实已经不再拥有信号量。所以调用down\_interruptible时，必须检查返回值，作出相应的处理。

down\_trylock不会休眠；如果信号量不可用，down\_trylock会立即返回，并返回一个非零值。

一旦，线程调用了上面的某一个down函数，就会说它拥有信号量。该线程现在有权访问受保护的临界区代码。

Linux中的V函数就是up函数：

|  |
| --- |
| void up(struct semaphore \*sem); |

一旦调用up，调用者将不再拥有该信号量。

正如你所料，任何线程都要通过一次调用释放掉信号量。但是，在错误发生时，必须特别小心，如果在拥有信号量的同时发生了错误，必须在返回错误状态给调用者之前释放掉信号量。未能释放信号量是一个容易犯的错误；结果可能很难重现并追踪。

### 5.3.2 在scull中使用信号量

信号量机制给了scull代码一种避免竞态条件的方法。但是对于我们编程者来说，正确地使用信号量才是关键。指定那些资源被保护，确保使用正确的加锁方式对这些资源进行访问，这些都是正确使用信号量的关键。

让我们再看一次scull\_dev结构体：

|  |
| --- |
| struct scull\_dev {  struct scull\_qset \*data; /\* 指向第一个量子集的指针 \*/  int quantum; /\* 当前量子的大小 \*/  int qset; /\* 当前数组的大小 \*/  unsigned long size; /\* 数据的大小 \*/  unsigned int access\_key; /\* 被sculluid和scullpriv使用 \*/  struct semaphore sem; /\* 互斥信号量 \*/  struct cdev cdev; /\* 字符设备结构 \*/  }; |

上面结构体的倒数第2个成员，sem，互斥信号量。我们选择为每一个虚拟的scull设备提供一个独立的互斥信号量。其实呢，使用一个全局的互斥信号量也是可以的。但是，这几个虚拟的scull设备之间没有共同的资源，而且使用全局互斥信号量的话，一个虚拟设备在使用，其它设备也会无法使用，这样的效率有点低。所以，我们选择为每个设备提供一个独立的互斥信号量。

信号量在使用之前必须被初始化。看下面的代码：

|  |
| --- |
| for (i = 0; i < scull\_nr\_devs; i++) {  scull\_devices[i].quantum = scull\_quantum;  scull\_devices[i].qset = scull\_qset;  sema\_init(&scull\_devices[i].sem, 1);  scull\_setup\_cdev(&scull\_devices[i], i);  } |

在系统可以使用scull设备之前，互斥信号量sem必须被初始化。因此，sema\_init必须在scull\_setup\_cdev之前进行调用。

接下来，我们必须遍历所有的代码，确保没有在没有持有互斥信号量的情况下访问scull\_dev结构体。所以，scull\_write开头部分的代码如下所示：

|  |
| --- |
| if (down\_interruptible(&dev->sem))  return -ERESTARTSYS; |

注意：检查down\_interruptible的返回值，如果返回非0，说明操作被中断。在这种情况下，通常返回-ERESTARTSYS。基于这个返回码，内核上层代码决定重启调用还是将其错误返回给用户。如果返回-ERESTARTSYS，则必须首先撤消可能已进行的任何用户可见更改，以便在重试系统调用时发生正确的操作。 如果您无法以这种方式撤消操作，则应返回-EINTR。

不论scull\_write中的其它代码能否正常执行，都必须释放其占用的信号量。如果执行正常，最后会执行下面的代码：

|  |
| --- |
| out:  up(&dev->sem);  return retval; |

### 5.3.3 读写信号量

## 5.4 completion机制

## 5.5 自旋锁

### 5.5.1 自旋锁API介绍

### 5.5.2 自旋锁和原子上下文

### 5.5.3 自旋锁函数

### 5.5.4 读写自旋锁

## 5.6 锁陷阱

### 5.6.1 不明确的规则

### 5.6.2 加锁的顺序规则

### 5.6.3 细粒度锁和粗粒度锁的对比

## 5.7 除锁之外的选择

### 5.7.1 免锁算法

### 5.7.2 原子变量

### 5.7.3 位操作

### 5.7.4 seqlock锁

### 5.7.5 读取-复制-更新

# 第6章 高级字符驱动操作

在第3章中，我们已经编写了一个可以同步读写的完整字符驱动程序。但是，真实的设备往往提供了更多的功能。现在，我们已经掌握了驱动程序的调试方法，对设备的并发问题也有了坚实的理解，这些都有助于我们编写更高级的驱动程序了。

那么本章我们要学习的内容重点就是，编写功能完整的驱动所需的一些概念。从实现ioctl系统调用开始，它是用于设备控制的常用接口。然后介绍和用户空间同步的几种方式。学完本章，读者应该掌握如何使进程休眠（并唤醒），如何实现非阻塞I/O，以及在设备可读写时如何通知用户空间等。最后，我们介绍了如何在驱动程序中实现几种不同的访问设备的策略。

这些实现都是通过scull驱动的几个变体演示的。真实的硬件驱动，直到第9章才能看到。

## 6.1 ioctl接口

除了读写设备之外，多数驱动程序还要求有对设备的访问控制能力。大部分设备不光光是简单的数据传输，它们还要实现一些其它的操作，比如，设备加锁、弹出介质、报告错误信息、改变波特率或者自破坏等。这些操作都是通过ioctl方法实现的。

在用户空间，系统调用ioctl具有如下的原型：

|  |
| --- |
| int ioctl(int fd, unsigned long cmd, ...); |

### 6.1.1 选择ioctl命令

在编写ioctl代码之前，需要选择与ioctl命令对应的数字。许多程序员的第一直觉是选择一组从0或1开始的小数字。但是，有充分的理由不这样做。ioctl命令编号在整个系统中应该是唯一的，以防止由于向错误的设备发出正确的命令而导致的错误。这种向错误设备发送正确命令很有可能发生，程序可能会发现自己试图改变非串行端口输入流的波特率，例如FIFO或音频设备。如果每个ioctl命令编号都是唯一的，则应用程序会收到EINVAL错误，而不是成功执行意外的操作。

为了帮助程序员创建唯一的ioctl命令码，这些编码被划分成几个位字段。第一个Linux版本中使用16位数字：前八位是与设备相关的“幻数”，而后八位是序列号，在设备中是唯一的。据linus所说，这是随便设计的，因为当时他也是“毫无头绪”（他自己的话）；更好的位域划分只是后来才构想出来的。不幸的是，不少驱动程序仍然使用旧的约定。他们必须这么做的原因就是：改变ioctl命令编码将可能破坏二进制程序的执行，而这不是内核开发人员愿意做的事情。

要根据Linux内核约定为驱动程序选择ioctl编号，首先应检查include/asm/ioctl.h和Documentation/ioctl/ioctl-number.txt。该头文件定义了将要使用的位字段：类型（幻数），序数，传输方向和参数大小。ioctl-number.txt文件列出了整个内核使用的幻数，因此您能够选择自己的幻数并避免重叠。该文件还描述了使用该约定的原因。

定义编号的新方法使用了4个位字段，其含义如下所示。下面所要介绍的新符号都定义在<linux/ioctl.h>中。

* type

幻数。只需选择一个数字（在看过ioctl-number.txt之后）并在整个驱动程序中使用它。该字段为8位宽（\_IOC\_TYPEBITS）。

* number

序数（顺序编号）。它也是8位（\_IOC\_NRBITS）宽。

* direction

如果特定命令涉及数据传输，则该字段代表数据传输的方向。可能的取值是\_IOC\_NONE（没有数据传输），\_IOC\_READ和\_IOC\_READ|\_IOC\_WRITE（数据双向传输）。数据传输方向的判定是从用户空间应用程序的视角判断的。

* size

所涉及的用户数据的大小。该字段的宽度取决于体系结构，但通常为13或14位。可通过宏\_IOC\_SIZEBITS中找到特定体系结构的具体数值。使用size字段并不是强制性的-内核不会检查它-但这是一个好主意。正确使用此字段有助于检测用户空间编程错误，并且如果您需要更改相关数据项的大小，则可以实现向后兼容性。但是，如果需要更大的数据结构，则可以忽略size字段。我们将很快看到这个字段是如何使用的。

<linux/ioctl.h>中包含的<asm/ioctl.h>头文件定义了一些有助于设置命令编号的宏，如下所示：\_IO（type，nr）（对于没有参数的命令），\_IOR（type，nr，datatype）（用于从驱动程序读取数据），\_IOW（type，nr，datatype）（用于写入数据）和\_IOWR（type，nr，datatype）（用于双向传输）。type和number字段作为参数传递，size字段通过将sizeof应用于数据类型参数来获得。

头文件还定义了解除这些编号的宏：\_IOC\_DIR(nr)，\_IOC\_TYPE(nr)，\_IOC\_NR(nr)，和 \_IOC\_SIZE(nr)。我们在这里不作介绍，后面代码中会有提及。

以下是在scull中定义的一些ioctl命令。特别是，这些命令可以设置和获取驱动程序的配置参数。

|  |
| --- |
| /\* Use 'k' as magic number \*/  #define SCULL\_IOC\_MAGIC 'k'  /\* Please use a different 8-bit number in your code \*/  #define SCULL\_IOCRESET \_IO(SCULL\_IOC\_MAGIC, 0)  /\*  \* S means "Set" through a ptr,  \* T means "Tell" directly with the argument value  \* G means "Get": reply by setting through a pointer  \* Q means "Query": response is on the return value  \* X means "eXchange": switch G and S atomically  \* H means "sHift": switch T and Q atomically  \*/  #define SCULL\_IOCSQUANTUM \_IOW(SCULL\_IOC\_MAGIC, 1, int)  #define SCULL\_IOCSQSET \_IOW(SCULL\_IOC\_MAGIC, 2, int)  #define SCULL\_IOCTQUANTUM \_IO(SCULL\_IOC\_MAGIC, 3)  #define SCULL\_IOCTQSET \_IO(SCULL\_IOC\_MAGIC, 4)  #define SCULL\_IOCGQUANTUM \_IOR(SCULL\_IOC\_MAGIC, 5, int)  #define SCULL\_IOCGQSET \_IOR(SCULL\_IOC\_MAGIC, 6, int)  #define SCULL\_IOCQQUANTUM \_IO(SCULL\_IOC\_MAGIC, 7)  #define SCULL\_IOCQQSET \_IO(SCULL\_IOC\_MAGIC, 8)  #define SCULL\_IOCXQUANTUM \_IOWR(SCULL\_IOC\_MAGIC, 9, int)  #define SCULL\_IOCXQSET \_IOWR(SCULL\_IOC\_MAGIC,10, int)  #define SCULL\_IOCHQUANTUM \_IO(SCULL\_IOC\_MAGIC, 11)  #define SCULL\_IOCHQSET \_IO(SCULL\_IOC\_MAGIC, 12)  #define SCULL\_IOC\_MAXNR 14 |

### 6.1.2 返回值

ioctl通常是通过switch语句实现的，但是当命令编号不匹配任何合法的操作时，默认的选择是什么？对于这个问题颇有争议。有些内核函数返回-EINVAL（“Invalid argument”，非法参数），这是合理的，因为命令参数确实不是合法的。但是，POSIX标准规定，如果使用了不合适的ioctl命令参数，则返回-ENOTTY。C库将这个错误码解释为“inappropriate ioctl for device”。这看起来更贴切些。返回-EINVAL仍然是常用的方法。

### 6.1.3 预定义命令

尽管大部分的ioctl系统调用用于操作设备，但是还是有一些命令是可以被内核识别的。所以，要注意这部分预定义的命令，它们会在被设备文件操作调用之前被内核解析。所以，如果驱动程序选用了与这部分预定义命令相同的编号，就永远不会收到该命令的请求，而且由于编号冲突，应用程序的行为将无法预测。

预定义命令被分为3组：

* 可以作用于任何文件的命令（普通，设备，FIFO或socket等文件）
* 只作用于普通文件
* 特定于文件系统类型的命令

最后一组命令只能在宿主文件系统上运行（见chattr命令）。设备驱动程序开发者只关注第一组命令，它们的幻数都是“T”。

### 6.1.4 使用ioctl参数

ioctl函数原型中还有一个可选参数，如果是整数的话直接使用就好了，如果是指针的话，有些问题需要注意的。

当指针指向用户空间时，必须确保用户空间地址是合法的。否则，如果不加验证的话，可能会造成内核oops、系统崩溃或者安全问题。驱动程序应该负责对每个用到的用户空间地址作适当的检查。如果是非法地址就返回一个错误。

在第3章中，我们了解了copy\_from\_user和copy\_to\_user两个函数，它们负责在内核空间和用户空间之间安全的搬运数据。但是，ioctl调用一般都是很小的数据，使用其它方法操作。所以，要验证用户空间地址是否合法需要调用一个函数access\_ok，其定义位于<asm/uaccess.h>中其原型如下：

|  |
| --- |
| int access\_ok(int type, const void \*addr, unsigned long size); |

第一个参数，type应该是VERIFY\_READ或VERIFY\_WRITE，取决于要执行的动作是对用户空间的读写操作方式。第二个参数，addr是用户空间地址。第三个参数，size是字节数。例如，如果ioctl需要从用户空间读取一个整数，size就是sizeof(int)。如果需要对指定地址进行读写操作，那么使用VERIFY\_WRITE，因为它包含VERIFY\_READ。

与大多数内核函数不同的是，access\_ok返回bool值：1，成功；0，失败。如果返回false，驱动程序应该返回-EFAULT，告知调用方。

对于access\_ok函数，我们需要关注的两点就是：首先，它没有完成内存检查的所有工作，仅仅检查了所访问的内存是进程可以访问的区域。尤其是，保证其没有指向内核空间的内存区域。其次，大多数驱动程序不需真的调用access\_ok。后面描述的内存管理程序会处理它。尽管如此，我们还是要演示一下它的使用。

下面的scull源代码展示了参数的检查：

|  |
| --- |
| int err = 0, tmp;  int retval = 0;  /\*  \* extract the type and number bitfields, and don't decode  \* wrong cmds: return ENOTTY (inappropriate ioctl) before access\_ok( )  \*/  if (\_IOC\_TYPE(cmd) != SCULL\_IOC\_MAGIC) return -ENOTTY;  if (\_IOC\_NR(cmd) > SCULL\_IOC\_MAXNR) return -ENOTTY;  /\*  \* the direction is a bitmask, and VERIFY\_WRITE catches R/W  \* transfers. `Type' is user-oriented, while  \* access\_ok is kernel-oriented, so the concept of "read" and  \* "write" is reversed  \*/  if (\_IOC\_DIR(cmd) & \_IOC\_READ)  err = !access\_ok(VERIFY\_WRITE, (void \_\_user \*)arg, \_IOC\_SIZE(cmd));  else if (\_IOC\_DIR(cmd) & \_IOC\_WRITE)  err = !access\_ok(VERIFY\_READ, (void \_\_user \*)arg, \_IOC\_SIZE(cmd));  if (err) return -EFAULT; |

调用access\_ok之后，驱动程序就能够安全地执行数据传输了。除了copy\_from\_user和copy\_to\_user函数之外，编程人员还可以使用一组专门为最常用数据大小（1,2,4,和8字节）优化过的函数。这些函数定义在文件<asm/uaccess.h>中，描述如下：

put\_user(datum, ptr)和\_\_put\_ user(datum, ptr)

这些宏把datum写到用户空间。他们相对快些，当传递单个数据时应该使用它代替copy\_to\_user。允许传递任何类型的指针给put\_user，只要是用户空间地址就行。传递数据的大小依赖于指针参数的类型，编译时由编译器内建指令sizeof和typeof确定。因此，如果ptr是字符指针，传输一个字节，2、4、8字节情况类似。

put\_user检查确保进程可以写入指定的内存地址。成功返回0，失败返回-EFAULT。\_\_put\_ user执行的检查少一些（不调用access\_ok）。建议尽量使用put\_user。

get\_user(local, ptr)和\_\_get\_user(local, ptr)

从用户空间获取一个数据，行为与put\_user和\_\_put\_ user类似，只是传输数据的方向相反。获取的值保存在局部变量local中，返回值表明操作是否成功。同样，\_\_get\_user没有使用access\_ok验证内存地址。

如果尝试使用上面两类函数传输不是上面指定大小的数据时，编译器常常给出奇怪的信息：“conversion to non-scalar type requested.”。这种情况下，必须使用copy\_from\_user和copy\_to\_user。

### 6.1.5 兼容性和受限操作

### 6.1.6 ioctl命令的实现

### 6.1.7 非ioctl的设备实现

## 6.2 阻塞I/O

在第3章中，我们已经编写了scull设备read和write方法，但是我们忽略了一个重要的问题，如果当我们read时，scull设备还没有数据可用怎么办？当我们write时，scull设备还没准备好接收数据怎么办？在这些情况下，驱动程序默认情况下应该阻塞进程，让其休眠直至发出的请求可以被响应。

本章展示了怎样使一个进程休眠和何时唤醒。首先，先来了解一些新概念。

### 6.2.1 休眠的介绍

什么是休眠？休眠就是进程被标记为特殊状态并从调度器管理的运行队列中删除。除非外界修改这种状态，否则进程就不会被任何CPU调度。

在linux设备驱动中使进程休眠是一件很简单的事情。但是，要想使代码安全的休眠，必须注意两条规则：

第一条规则：原子操作的上下文中不能休眠。在第5章我们已经了解了，原子操作是阻止并发访问的一种手段。那么也就意味着你的驱动程序持有自旋锁，seqlock，或RCU lock时，不能进行休眠。当然了，如果中断被失能，也不能进行休眠。拥有信号量可以休眠，但是必须小心。如果拥有信号量的代码休眠，那么其它等待信号量的代码也会休眠。因此，拥有信号量的代码休眠时间必须短，且你应该确信，不会阻塞那些最终会唤醒你的那些进程。

关于休眠，另一条规则就是：当你唤醒时，肯定不会知道已经失去CPU控制权多长时间；同时，在这段时间内发生了什么。另外，你通常也不会知道是否有另一个进程也在等待同一事件而在休眠；那个进程可能在你的进程之前被唤醒同时掠夺你的进程可能在等待的资源。最后的结论就是，你不能在进程唤醒之后假设系统的状态，必须确保你所等待的条件是真实存在的。

关于休眠，另一个观点就是，如果没有其它代码唤醒它，就不要休眠。当唤醒事件发生时，通知wait队列（等待某一事件的进程列表）。

在linux中，使用“等待队列头”来管理wait队列，其类型为wait\_queue\_head\_t结构体，定义在<linux/wait.h>中。一个等待队列头初始化如下：

DECLARE\_WAIT\_QUEUE\_HEAD(name);

或者动态初始化：

|  |
| --- |
| wait\_queue\_head\_t my\_queue;  init\_waitqueue\_head(&my\_queue); |

### 6.2.2 简单休眠

Linux内核中，唤醒休眠的进程最简单的方法就是使用宏定义wait\_event及一些变体，它的形式是：

|  |
| --- |
| wait\_event(queue, condition)  wait\_event\_interruptible(queue, condition)  wait\_event\_timeout(queue, condition, timeout)  wait\_event\_interruptible\_timeout(queue, condition, timeout) |

在这里，queue就是使用的等待队列头。注意是值传递的方式。condition就是休眠前后通过宏计算的任意bool表达式；除非condition称为true，否则一直休眠。值得注意的是，condition是次数的任意数字，所以不能有任何副作用。

如果使用wait\_event，进程将会进入不可中断的休眠中，这应该不是你想要的结果。优先选择的替代方案就是，wait\_event\_interruptible，它可被信号中断，返回整数值。非0，说明休眠被某一类信号中断，驱动程序就应该返回-ERESTARTSYS。wait\_event\_timeout和wait\_event\_interruptible\_timeout定周期等待，如果时间到，无论condition是否满足条件，都返回0。

唤醒休眠的进程的语法是：

|  |
| --- |
| void wake\_up(wait\_queue\_head\_t \*queue);  void wake\_up\_interruptible(wait\_queue\_head\_t \*queue); |

wake\_up唤醒给定队列中所有正在等待进程（当然了实际情况非常复杂，我们稍后会看到）。wake\_up\_interruptible只唤醒可中断休眠。通常情况下，与等待被唤醒的语句配套使用。比如wake\_up和wait\_event配合使用。

我们下面看一个关于休眠和唤醒的简单例子。模块我们称之为sleepy。表现行为：任何试图读取设备的进程都会进入休眠；无论什么时候有一个进程写设备，都会唤醒所有的进程。读写方法如下，所示：

|  |
| --- |
| static DECLARE\_WAIT\_QUEUE\_HEAD(wq);  static int flag = 0;  ssize\_t sleepy\_read (struct file \*filp, char \_\_user \*buf, size\_t count, loff\_t \*pos)  {  printk(KERN\_DEBUG "process %i (%s) going to sleep\n", current->pid,  current->comm);  wait\_event\_interruptible(wq, flag != 0);  flag = 0;  printk(KERN\_DEBUG "awoken %i (%s)\n", current->pid, current->comm);  return 0; /\* EOF \*/  }  ssize\_t sleepy\_write (struct file \*filp, const char \_\_user \*buf, size\_t count,  loff\_t \*pos)  {  printk(KERN\_DEBUG "process %i (%s) awakening the readers...\n",  current->pid, current->comm);  flag = 1;  wake\_up\_interruptible(&wq);  return count; /\* succeed, to avoid retrial \*/  } |

### 6.2.3 阻塞和非阻塞操作

### 6.2.4 一个阻塞I/O例子

我们来看一个真实的阻塞I/O的设备驱动，它是scull一个特殊形式，类似于pipe的设备。

驱动中，当有数据到达的时候就会唤醒被read调用阻塞的进程。通常情况下，硬件会发送一个中断信号作为事件，驱动程序唤醒写进程。但是scullpipe设备驱动程序的工作方式和真实的硬件是不同的，所以，我们使用另一个进程去唤醒被阻塞的进程。

设备驱动的device结构体，其包含两个等待队列和一个buffer。Buffer的大小是可配置的。

|  |
| --- |
| struct scull\_pipe {  wait\_queue\_head\_t inq, outq; /\* read 和 write队列 \*/  char \*buffer, \*end; /\* 缓冲区的头尾 \*/  int buffersize; /\* 缓冲区大小 \*/  char \*rp, \*wp; /\* 读写位置指针 \*/  int nreaders, nwriters; /\* r/w操作的数量 \*/  struct fasync\_struct \*async\_queue; /\* 异步读取者 \*/  struct semaphore sem; /\* 互斥信号量 \*/  struct cdev cdev; /\* 字符设备结构 \*/  }; |

read负责管理阻塞型和非阻塞型输入，如下所示：

|  |
| --- |
| static ssize\_t scull\_p\_read(struct file \*filp, char \_\_user \*buf, size\_t count,  loff\_t \*f\_pos)  {  struct scull\_pipe \*dev = filp->private\_data;  if(down\_interruptible(&dev->sem))  return -ERESTARTSYS;  while (dev->rp == dev->wp) { /\* 什么也没读到 \*/  up(&dev->sem); /\* 释放锁 \*/  if (filp->f\_flags & O\_NONBLOCK)  return -EAGAIN;  PDEBUG("\"%s\" reading: going to sleep\n", current->comm);  if (wait\_event\_interruptible(dev->inq, (dev->rp != dev->wp)))  return -ERESTARTSYS; /\* 信号: 通知fs层进行处理 \*/    /\* 否则循环，但是首先获取锁 \*/  if (down\_interruptible(&dev->sem))  return -ERESTARTSYS;  }  /\* 数据已就绪，返回 \*/  if (dev->wp > dev->rp)  count = min(count, (size\_t)(dev->wp - dev->rp));  else /\* 写入指针回卷，返回数据直到dev->end \*/  count = min(count, (size\_t)(dev->end - dev->rp));  if (copy\_to\_user(buf, dev->rp, count)) {  up(&dev->sem);  return -EFAULT;  }    dev->rp += count;  if (dev->rp == dev->end)  dev->rp = dev->buffer; /\* 回卷 \*/  up(&dev->sem);  /\* 最后，唤醒所有写入者并返回 \*/  wake\_up\_interruptible(&dev->outq);  PDEBUG("\"%s\" did read %li bytes\n",current->comm, (long)count);  return count;  } |

PDBUG行代码用于调试。

分析scull\_p\_read是如何处理等待数据的。while循环在拥有设备信号量时，测试缓冲区。如果其中有数据，则可以立即将数据返回给用户而不需要休眠，这样，循环内的内容就被跳过了。如果缓冲区为空，那么休眠。但在休眠之前必须释放设备信号量。因为如果在拥有信号量的时候休眠，那么任何写入者都没有机会来唤醒。在释放信号量之后，快速检查用户请求的是否是非阻塞I/O，如果是，则返回，否则调用wake\_up\_interruptible。

### 6.2.5 高级休眠

到目前为止，对于大多数驱动程序来说，已经介绍的函数足够满足休眠操作的需要了。但是，有些场合，可能需要对linux等待队列工作机理进行深入的了解。复杂的锁或性能方面的要求，都有可能强迫驱动程序使用更底层的函数去实现休眠。本节中，我们将会看到更底层的函数是如何影响进程休眠的。

#### 6.2.5.1 进程如何休眠

如果你查看<linux/wait.h>，你会发现wait\_ queue\_head\_t类型背后的数据结构非常简单; 它由一个自旋锁和一个链表组成。该链表的内容是等待队列条目，使用wait\_queue\_t类型声明。该结构包含有关休眠进程的信息以及它想要被唤醒的确切方式。

将进程置于休眠状态的第一步通常是wait\_queue\_t结构的分配和初始化，然后将其添加到正确的等待队列。当一切都到位后，任何负责唤醒的任务都能找到正确的进程。

下一步是设置进程的状态以将其标记为处于休眠状态。在<linux/sched.h>中定义了几个任务状态。TASK\_RUNNING意味着该进程能够运行，尽管在任何特定时刻，它不一定在处理器中正在执行。有两种状态表明进程处于休眠状态：TASK\_INTERRUPTIBLE和TASK\_UNINTERRUPTIBLE；当然，它们对应于两种类型的休眠。驱动程序作者通常不需要关心其它状态。

在2.6内核中，驱动程序代码通常不需要直接操作进程状态。但是，如果您需要这样做，则使用的调用是：

|  |
| --- |
| void set\_current\_state(int new\_state); |

#### 6.2.5.2 手动休眠

In previous versions of the Linux kernel, nontrivial sleeps required the programmer to handle all of the above steps manually. It was a tedious process involving a fair amount of error-prone boilerplate code. Programmers can still code a manual sleep in that manner if they want to; <linux/sched.h> contains all the requisite definitions, and the kernel source abounds with examples. There is an easier way, however.

在以前版本的Linux内核中，重要的休眠需要程序员手动处理上述所有步骤。这是一个繁琐的过程，涉及相当多的容易出错的样板代码。程序员仍然可以按照这种方式编写手动睡眠代码;<linux/sched.h>包含所有必需的定义，内核源代码中包含大量示例。然而，有一种更简单的方法。

第一步就是wait队列条目的创建和初始化，使用宏：

|  |
| --- |
| DEFINE\_WAIT(my\_wait); |

还可以这样写，

|  |
| --- |
| wait\_queue\_t my\_wait; init\_wait(&my\_wait); |

通常来说，使用宏更好一些。

第二步就是添加my\_wait等待队列项到队列中，设置进程状态。可以使用下面的函数处理：

|  |
| --- |
| void prepare\_to\_wait(wait\_queue\_head\_t \*queue, wait\_queue\_t \*wait, int state); |

在这里，queue和wait分别是等待队列头和进程添加到等待队列中的项，state就是进程的最新状态，TASK\_INTERRUPTIBLE或TASK\_UNINTERRUPTIBLE。

调用prepare\_to\_wait之后，进程由schedule管理-检查是否还需要继续等待。一旦从schedule返回，就会调用下面这个函数，进行善后处理，

|  |
| --- |
| void finish\_wait(wait\_queue\_head\_t \*queue, wait\_queue\_t \*wait); |

在之前，我们已经查看了scullpipe设备的read方法，它使用了wait\_event方法。但是write却是用prepare\_to\_wait和finish\_wait实现的。正常情况下，不要混合使用这些方法，但是，这里我们为了演示处理休眠的两种方法，特意如此而已。

下面我们看一下write方法的代码：

|  |
| --- |
| /\*  \* 判断缓冲区剩余  \*/  static int space\_free(struct scull\_pipe \*dev)  {  if (dev->rp == dev->wp)  {  return dev->buffersize - 1;  }  return ((dev->rp + dev->buffersize - dev->wp) % dev->buffersize) - 1;  }  /\*  \* 等待有可写入空间；调用者必须拥有设备信号量；  \* 发生错误时，信号量必须在返回之前被释放。  \*/  static int scull\_get\_write\_space(struct scull\_pipe \*dev, struct file \*filp)  {  while (space\_free(dev) == 0) /\* 满 \*/  {  DEFINE\_WAIT(wait);    up(&dev->sem);  if (filp->f\_flags & O\_NONBLOCK)  return -EAGAIN;  PDEBUG("\"%s\" writing: going to sleep\n", current->comm);  prepare\_to\_wait(&dev->outq, &wait, TASK\_INTERRUPTIBLE);  if (space\_free(dev) = = 0)  schedule();  finish\_wait(&dev->outq, &wait);  if (signal\_pending(current))  return -ERESTARTSYS; /\* 发送信号：通知fs层处理 \*/  if (down\_interruptible(&dev->sem))  return -ERESTARTSYS;  }  return 0;  }  /\*  \* scullpipe设备的write方法  \*/  static ssize\_t scull\_p\_write(struct file \*filp, const char \_\_user \*buf,  size\_t count, loff\_t \*f\_pos)  {  struct scull\_pipe \*dev = filp->private\_data;  int result;  if (down\_interruptible(&dev->sem))  return -ERESTARTSYS;    /\* 是否有足够的写空间 \*/  result = scull\_get\_write\_space(dev, filp)  if(result)  {  return result; /\* scull\_get\_write\_space 调用 up(&dev->sem) \*/  }  /\* 空间足够，写入 \*/  count = min(count, (size\_t) space\_free(dev));  if(dev->wp >= dev->rp)  {  count = min(count, (size\_t)(dev->end - dev->wp)); /\* 到缓冲区尾 \*/  }else /\* 写指针回卷，最大可以写到读指针-1 \*/  {  count = min(count, (size\_t)(dev->rp - dev->wp - 1));  }  PDEBUG("Going to accept %li bytes to %p from %p\n",  (long)count, dev->wp, buf);  if (copy\_from\_user(dev->wp, buf, count))  {  up(&dev->sem);  return -EFAULT;  }  dev->wp += count;  if (dev->wp == dev->end)  {  dev->wp = dev->buffer; /\* 回卷 \*/  }  up(&dev->sem);    /\* 最后，唤醒所有的读进程 \*/  wake\_up\_interruptible(&dev->inq); /\* blocked in read( ) and select( ) \*/  /\* and signal asynchronous readers, explained late in chapter 5 \*/  if (dev->async\_queue)  {  kill\_fasync(&dev->async\_queue, SIGIO, POLL\_IN);  }  PDEBUG("\"%s\" did write %li bytes\n",current->comm, (long)count);  return count;  } |

这段代码中，将休眠操作部分写为一个单独的函数scull\_get\_write\_space，其余内容与read方法类似。它的工作就是确保缓冲区有空间存储新数据，如果有需要的话进行休眠。一旦，有足够的空间，scull\_p\_write就会复制用户数据到缓冲区，调整指针后，唤醒所有等待读取数据的进程。

函数scull\_get\_write\_space的说明：

如果还有空间就返回。否则，释放设备的信号量且等待，直到再有空间。代码中使用DEFINE\_WAIT建立等待队条目并且调用prepare\_to\_wait准备好休眠。然后对缓冲区强制检查；我们必须处理，进入循环且释放信号量之后，添加进等待队列之前，缓冲区又变得可用的情况。如果不做检查，在那个时间段内读进程清空了缓冲区，我们就会错过唯一一次唤醒写进程的机会而永远休眠下去。如果有空间就调用schedule()函数。

那么我们来考虑另外一种情况，如果在if语句和schedule语句之间，发生唤醒事件怎么办？这种情况还好，唤醒事件会将进程状态重置为TASK\_RUNNING且schedule返回，尽管这不需要立即执行。只要对缓冲区的检查发生在进程把自己添加到等待队列中并改变其状态之后，就会正常工作。

然后，调用finish\_wait。对signal\_pending的调用告诉我们是否被信号唤醒了；如果是这样，我们需要返回给用户，让他们稍后再试。否则，我们重新获取信号量，再测试一次可用的缓冲区是否存在。

#### 6.2.5.3 独占等待

我们已经看到，当一个进程调用wake\_up，以作用于等待队列时，等待队列上所有的进程都会变成可运行态。在许多情况下，这是没有问题的。但是，在其它情况下，很可能提前知道被唤醒的进程只有一个能够获得所需资源，其余的将再次休眠。然而，这些进程中的每一个都将占有处理器的使用权，竞争所需资源（任何控制锁），然后明确地回到休眠状态。如果等待队列中的进程数量很大，这种“thundering herd”行为会严重地降低系统的性能。

为了应对这种竞争性问题（thundering herd）问题，内核开发者给内核添加了“独占等待”这个选项。这种等待跟正常的休眠非常类似，只有两点重要的不同：

* 当等待队列条目设置了WQ\_FLAG\_EXCLUSIVE标志时，它将被添加到等待队列的末尾。相反，没有该标志的条目被添加到开头。
* 在等待队列上调用wake\_up时，它会在唤醒设置了WQ\_FLAG\_EXCLUSIVE标志的第一个进程后停止。

最终的结果就是，以有序的方式，一次只能唤醒一个执行独占等待的进程，所以不会产生“thundering herds”问题。但与此同时，内核会唤醒所有没有执行独占等待的进程。

如果满足两个条件，则在驱动程序中使用独占等待是值得考虑的：对一个资源的使用存在大量竞争，只唤醒一个进程有效地使用资源。举例说，对于Apache web服务器，独占等待就工作的很好；当新连接到来时，只有系统中Apache 进程中一个可以唤醒并去处理。在scullpipe设备中，我们将不会使用独占式等待，因为很难产生数据竞争，另外，我们也不知道一旦读取进程被唤醒后，是否会消耗所有的可用数据。

将进程置于可中断的等待中是一个简单的问题，即调用prepare\_to\_wait\_exclusive：

|  |
| --- |
| void prepare\_to\_wait\_exclusive(wait\_queue\_head\_t \*queue, wait\_queue\_t \*wait,int state); |

当使用此调用代替prepare\_to\_wait时，在等待队列条目中设置“exclusive”标志，并将该进程添加到等待队列的末尾。请注意，无法使用wait\_event及其变体执行独占等待。

#### 6.2.5.4 唤醒的细节

我们上面表述的唤醒进程的细节相比内核中真实地发生的事情要简单的多。在wait队列中有个函数专门控制当进程唤醒时的行为。默认唤醒函数（default\_wake\_function）将进程设置为运行态，如果该进程具有更高优先级，那么还要进行上下文内容切换。设备驱动程序根本没有必要提供一个不同的唤醒函数。如果你认为有必要，可参阅<linux/wait.h>获得关于如何设置的信息。

我们还没有见过所有的wake\_up的变体。大多数驱动程序作者也不需要实现这些，但是，为了完整性，这里全部列出了：

|  |
| --- |
| wake\_up(wait\_queue\_head\_t \*queue);  wake\_up\_interruptible(wait\_queue\_head\_t \*queue);  wake\_up\_nr(wait\_queue\_head\_t \*queue, int nr);  wake\_up\_interruptible\_nr(wait\_queue\_head\_t \*queue, int nr);  wake\_up\_all(wait\_queue\_head\_t \*queue);  wake\_up\_interruptible\_all(wait\_queue\_head\_t \*queue);  wake\_up\_interruptible\_sync(wait\_queue\_head\_t \*queue); |

如果不能记住也没关系，除了wake\_up\_interruptible之外，其它的驱动程序很少调用。

#### 6.2.5.5 旧接口

如果深入研读linux内核代码，你会发现我们到为止忽略讨论的两个函数：

|  |
| --- |
| void sleep\_on(wait\_queue\_head\_t \*queue);  void interruptible\_sleep\_on(wait\_queue\_head\_t \*queue); |

目前的计划就是在不远的将来，sleep\_on及其变体我们还没有显示几种超时形式）将会被删除。所以，不做过多的介绍。

### 6.2.6 测试scullpipe驱动

1. 测试环境

测试平台：qemu+mini2440

构建内核工具：buildroot-2012.05

Linux内核版本：Linux 3.3.7

2. 测试方法

我们已经看到scullpipe驱动程序如何实现阻塞I/O。如果你想尝试一下，可以在本书其余部分的示例中找到此驱动程序的源代码。打开2个窗口，第一个可以运行cat /dev/scullpipe命令；在另一个窗口中将文件复制到 /dev/scullpipe，您应该会看到该文件的内容出现在第一个窗口中。

测试非阻塞功能比较棘手，因为shell可用的传统程序不执行非阻塞操作。misc-progs源目录包含以下的简单程序，称为nbtest，用于测试非阻塞操作。它所做的只是将其输入复制到其输出，使用非阻塞I/O并在重试之间延迟。延迟时间在命令行上传递，默认为1秒。

|  |
| --- |
| int main(int argc, char \*\*argv)  {  int delay = 1, n, m = 0;    if (argc > 1)  delay = atoi(argv[1]);    fcntl(0, F\_SETFL, fcntl(0,F\_GETFL) | O\_NONBLOCK); /\* stdin \*/  fcntl(1, F\_SETFL, fcntl(1,F\_GETFL) | O\_NONBLOCK); /\* stdout \*/    while (1)  {  n = read(0, buffer, 4096);  if (n >= 0)  m = write(1, buffer, n);  if ((n < 0 || m < 0) && (errno != EAGAIN))  break;  sleep(delay);  }  perror(n < 0 ? "stdin" : "stdout");  exit(1);  } |

如果在进程跟踪工具（如strace）下运行此程序，则可以看到每个操作的成功或失败，具体取决于尝试操作时数据是否可用。

## 6.3 poll和select

### 6.3.1 与read和write的交互

### 6.3.2 底层的数据结构

## 6.4 异步通知

### 6.4.1 从驱动程序的角度考虑

## 6.5 定位设备

### 6.5.1 llseek的实现

## 6.6 设备文件的访问控制

### 6.6.1 独享设备

### 6.6.2 限制每次只由一个用户访问

### 6.6.3 替代EBUSY的阻塞型open

### 6.6.4 在open时复制设备

# 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：