Linux 设备驱动 Edition 3

下一页

Linux 设备驱动 Edition 3

By Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

由 quickwhale 翻译的简体中文版本 V0.0.1

遵循原版的版权声明. 还在完善中. 欢迎任何意见, 请给我邮件. 请发信至 quickwhale 的邮箱 <quickwhale@hotmail.com>

Printed in the United States of America. Published by O ’ Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. O ’ Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (safari.oreilly.com). For more information, contact our corporate/insti-tutional sales department: (800) 998-9938 or corporate@oreilly.com.

感谢

感谢本书原版的作者 Jonathan Corbet, Alessandro Rubini 和 Greg Kroah-Hartman

感谢我的家人 爸爸^2, 妈妈^2, $PIGY_{nuonuo}$

目录

下一页

第 1 章 第一章 设备驱动简介

第 1 章 第一章 设备驱动简介

上一页 下一页

第 1 章 第一章 设备驱动简介

目录

以 Linux 为代表的自由操作系统的很多优点之一, 是它们的内部是开放给所有人看的. 操作系统, 曾经是一个隐藏的神秘的地方, 它的代码只局限于少数的程序员, 现在已准备好让任何具备必要技能的人来检查, 理解以及修改. Linux 已经帮助使操作系统民主化. Linux 内核保留有大量的复杂的代码, 但是, 那些想要成为内核 hacker 的人需要一个入口点, 这样他们可以进入代码中, 不会被代码的复杂性压倒. 通常, 设备驱动提供了这样的门路.

驱动程序在 Linux 内核里扮演着特殊的角色. 它们是截然不同的"黑盒子", 使硬件的特殊的一部分响应定义好的内部编程接口. 它们完全隐藏了设备工作的细节. 用户的活动通过一套标准化的调用来进行, 这些调用与特别的驱动是独立的; 设备驱动的角色就是将这些调用映射到作用于实际硬件的和设备相关的操作上. 这个编程接口是这样, 驱动可以与内核的其他部分分开建立, 并在需要的时候在运行时"插入". 这种模块化使得 Linux 驱动易写, 以致于目前有几百个驱动可用.

编写 Linux 设备驱动有许多理由让人感兴趣. 可用的新硬件出现的速率(以及陈旧的速率)就确保了驱动编写者在可见的将来内是忙碌的. 个别人可能需要了解驱动以便存取一个他们感兴趣的特殊设备. 硬件供应商, 通过为他们的产品开发 Linux 驱动, 可以给他们的潜在市场增加大量的正在扩张的 Linux 用户基数. 还有 Linux 系统的开放源码性质意味着如果驱动编写者愿意, 驱动源码能够快速地散布到几百万用户.

本书指导你如何编写你自己的驱动, 以及如何利用内核相关的部分. 我们采用一种设备-独立的方法; 编程技术和接口, 在任何可能的时候, 不会捆绑到任何特定的设备. 每一个驱动都是不同的; 作为一个驱动编写者, 你需要深入理解你的特定设备. 但是大部分的原则和基本技术对所有驱动都是一样的. 本书无法教你关于你的设备的东西, 但是它给予你所需要的使你的设备运行起来的背景知识

的指导.

在你学习编写驱动时, 你通常会发现大量有关 Linux 内核的东西. 这也许会帮助你理解你的机器是如何工作的, 以及为什么事情不是如你所愿的快, 或者不是如你所要的进行. 我们会逐步介绍新概念, 由非常简单的驱动开始并建立它们; 每一个新概念都伴有例子代码, 这样的代码不需要特别的硬件来测试.

本章不会真正进入编写代码. 但是, 我们介绍一些 Linux 内核的背景概念, 这样在以后我们动手编程时, 你会感到乐于知道这些.

1.1. 驱动程序的角色

作为一个程序员, 你能够对你的驱动作出你自己的选择, 并且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. 尽管说一个驱动是"灵活"的, 听起来有些奇怪, 但是我们喜欢这个字眼, 因为它强调了一个驱动程序的角色是提供机制, 而不是策略.

机制和策略的区分是其中一个在 Unix 设计背后的最好观念. 大部分的编程问题其实可以划分为 2 部分:" 提供什么能力"(机制) 和 "如何使用这些能力"(策略). 如果这两方面由程序的不同部分来表达, 或者甚至由不同的程序共同表达, 软件包是非常容易开发和适应特殊的需求.

例如, 图形显示的 Unix 管理划分为 X 服务器, 它理解硬件以及提供了统一的接口给用户程序, 还有窗口和会话管理器, 它实现了一个特别的策略, 而对硬件一无所知. 人们可以在不同的硬件上使用相同的窗口管理器, 而且不同的用户可以在同一台工作站上运行不同的配置. 甚至完全不同的桌面环境, 例如 KDE 和 GNOME, 可以在同一系统中共存. 另一个例子是 TCP/IP 网络的分层结构: 操作系统提供 socket 抽象层, 它对要传送的数据而言不实现策略, 而不同的服务器负责各种服务(以及它们的相关策略). 而且, 一个服务器, 例如 ftpd 提供文件传输机制, 同时用户可以使用任何他们喜欢的客户端; 无论命令行还是图形客户端都存在, 并且任何人都能编写一个新的用户接口来传输文件.

在驱动相关的地方, 机制和策略之间的同样的区分都适用. 软驱驱动是不含策略的--它的角色仅仅是将磁盘表现为一个数据块的连续阵列. 系统的更高级部分提供了策略, 例如谁可以存取软驱驱动, 这个软驱是直接存取还是要通过一个文件系统, 以及用户是否可以加载文件系统到这个软驱. 因为不同的环境常常需要不同的使用硬件的方式, 尽可能对策略透明是非常重要的.

在编写驱动时, 程序员应当特别注意这个基础的概念: 编写内核代码来存取硬件, 但是不能强加特别的策略给用户, 因为不同的用户有不同的需求. 驱动应当做到使硬件可用, 将所有关于如何使用硬件的事情留给应用程序. 一个驱动, 这样, 就是灵活的, 如果它提供了对硬件能力的存取, 没有增加约束. 然而, 有时必须作出一些策略的决定. 例如, 一个数字 I/O 驱动也许只提供对硬件的字符存取, 以便避免额外的代码处理单个位.

你也可以从不同的角度看你的驱动: 它是一个存在于应用程序和实际设备间的软件层. 驱动的这种

特权的角色允许驱动程序员y严密地选择设备应该如何表现: 不同的驱动可以提供不同的能力, 甚至是同一个设备. 实际的驱动设计应当是在许多不同考虑中的平衡. 例如, 一个单个设备可能由不同的程序并发使用, 驱动程序员有完全的自由来决定如何处理并发性. 你能在设备上实现内存映射而不依赖它的硬件能力, 或者你能提供一个用户库来帮助应用程序员在可用的原语之上实现新策略, 等等. 一个主要的考虑是在展现给用户尽可能多的选项, 和你不得不花费的编写驱动的时间之间做出平衡, 还有需要保持事情简单以避免错误潜入.

对策略透明的驱动有一些典型的特征. 这些包括支持同步和异步操作, 可以多次打开的能力, 利用硬件全部能力, 没有软件层来"简化事情"或者提供策略相关的操作. 这样的驱动不但对他们的最终用户好用, 而且证明也是易写易维护的. 成为策略透明的实际是一个共同的目标, 对软件设计者来说.

许多设备驱动, 确实, 是与用户程序一起发行的, 以便帮助配置和存取目标设备. 这些程序包括简单的工具到完全的图形应用. 例子包括 tunelp 程序, 它调整并口打印机驱动如何操作, 还有图形的 cardctl 工具, 它是 PCMCIA 驱动包的一部分. 经常会提供一个客户库, 它提供了不需要驱动自身实现的功能.

本书的范围是内核, 因此我们尽力不涉及策略问题, 应用程序, 以及支持库. 有时我们谈论不同的策略以及如何支持他们, 但是我们不会进入太多有关使用设备的程序的细节, 或者是他们强加的策略的细节. 但是, 你应当理解, 用户程序是一个软件包的构成部分, 并且就算是对策略透明的软件包在发行时也会带有配置文件, 来对底层的机制应用缺省的动作.

1.2. 划分内核

在 Unix 系统中, 几个并发的进程专注于不同的任务. 每个进程请求系统资源, 象计算能力, 内存, 网络连接, 或者一些别的资源. 内核是个大块的可执行文件, 负责处理所有这样的请求. 尽管不同内核任务间的区别常常不是能清楚划分, 内核的角色可以划分(如同图内核的划分)成下列几个部分:

进程管理

内核负责创建和销毁进程, 并处理它们与外部世界的联系(输入和输出). 不同进程间通讯(通过信号, 管道, 或者进程间通讯原语)对整个系统功能来说是基本的, 也由内核处理. 另外, 调度器, 控制进程如何共享 CPU, 是进程管理的一部分. 更通常地, 内核的进程管理活动实现了多个进程在一个单个或者几个 CPU 之上的抽象.

内存管理

计算机的内存是主要的资源, 处理它所用的策略对系统性能是至关重要的. 内核为所有进程的每一个都在有限的可用资源上建立了一个虚拟地址空间. 内核的不同部分与内存管理子系统通过一套函数调用交互, 从简单的 malloc/free 对到更多更复杂的功能.

文件系统

Unix 在很大程度上基于文件系统的概念; 几乎 Unix 中的任何东西都可看作一个文件. 内核在非结构化的硬件之上建立了一个结构化的文件系统, 结果是文件的抽象非常多地在整个系统中应用. 另外, Linux 支持多个文件系统类型, 就是说, 物理介质上不同的数据组织方式. 例如, 磁盘可被格式化成标准 Linux 的 ext3 文件系统, 普遍使用的 FAT 文件系统, 或者其他几个文件系统.

设备控制

几乎每个系统操作最终都映射到一个物理设备上. 除了处理器, 内存和非常少的别的实体之外, 全部中的任何设备控制操作都由特定于要寻址的设备相关的代码来进行. 这些代码称为设备驱动. 内核中必须嵌入系统中出现的每个外设的驱动, 从硬盘驱动到键盘和磁带驱动器. 内核功能的这个方面是本书中的我们主要感兴趣的地方.

网络

网络必须由操作系统来管理, 因为大部分网络操作不是特定于某一个进程: 进入系统的报文是异步事件. 报文在某一个进程接手之前必须被收集, 识别, 分发. 系统负责在程序和网络接口之间递送数据报文, 它必须根据程序的网络活动来控制程序的执行. 另外, 所有的路由和地址解析问题都在内核中实现.

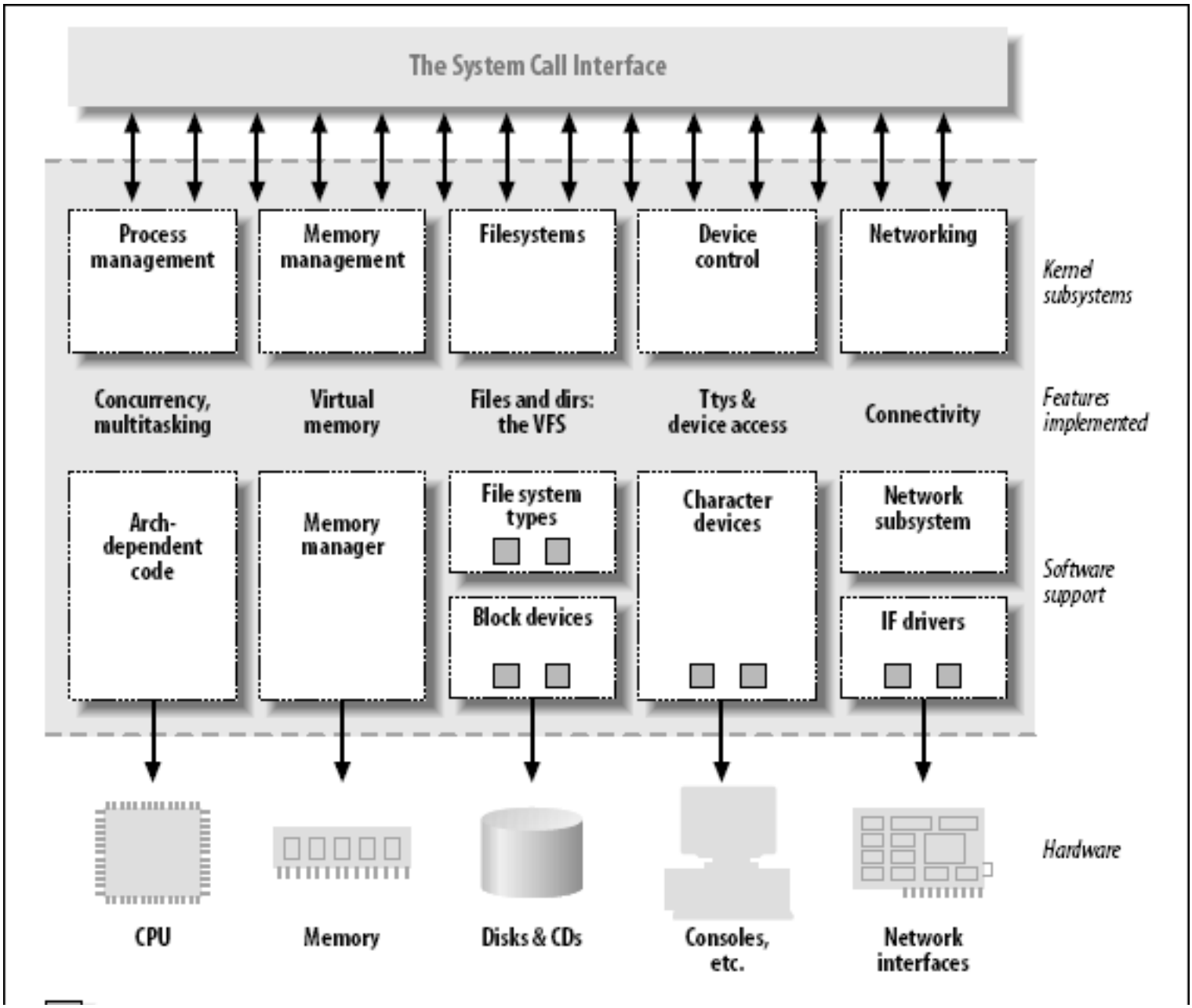
1.2.1. 可加载模块

Linux 的众多优良特性之一就是可以在运行时扩展由内核提供的特性的能力. 这意味着你可以在系统正在运行着的时候增加内核的功能(也可以去除).

每块可以在运行时添加到内核的代码, 被称为一个模块. Linux 内核提供了对许多模块类型的支持, 包括但不限于, 设备驱动. 每个模块由目标代码组成(没有连接成一个完整可执行文件), 可以动态连接到运行中的内核中, 通过 insmod 程序, 以及通过 rmmod 程序去连接.

图 内核的划分 表示了负责特定任务的不同类别的模块, 一个模块是根据它提供的功能来说它属于一个特别类别的. 图 内核的划分 中模块的安排涵盖了最重要的类别, 但是远未完整, 因为在 Linux 中越来越多的功能被模块化了.

图 1.1. 内核的划分



上一页 上一级 下一页
第 1 章 第一章 设备驱动简介 起始页 1.3. 设备和模块的分类

1.3. 设备和模块的分类

以 LInux 的方式看待设备可区分为 3 种基本设备类型. 每个模块常常实现 3 种类型中的 1 种, 因此可分类成字符模块, 块模块, 或者一个网络模块. 这种将模块分成不同类型或类别的方法并非是固定不变的; 程序员可以选择建立在一个大块代码中实现了不同驱动的巨大模块. 但是, 好的程序员, 常常创建一个不同的模块给每个它们实现的新功能, 因为分解是可伸缩性和可扩张性的关键因素.

3 类驱动如下:

既然不是一个面向流的设备, 一个网络接口就不象 /dev/tty1 那么容易映射到文件系统的一个结点上. Unix 的提供对接口的存取的方式仍然是通过分配一个名子给它们(例如 eth0), 但是这个名子在文件系统中没有对应的入口. 内核与网络设备驱动间的通讯与字符和块设备驱动所用的完全不同. 不用 read 和 write, 内核调用和报文传递相关的函数.

字符设备

> 一个字符(char) 设备是一种可以当作一个字节流来存取的设备(如同一个文件); 一个字符驱动负责实现这种行为. 这样的驱动常常至少实现 open, close, read, 和 write 系统调用. 文本控制台(/dev/console)和串口(/dev/ttyS0 及其友)是字符设备的例子, 因为它们很好地展现了流的抽象. 字符设备通过文件系统结点来存取, 例如 /dev/tty1 和 /dev/lp0. 在一个字符设备和一个普通文件之间唯一有关的不同就是, 你经常可以在普通文件中移来移去, 但是大部分字符设备仅仅是数据通道, 你只能顺序存取.然而, 存在看起来象数据区的字符设备, 你可以在里面移来移去. 例如, frame grabber 经常这样, 应用程序可以使用 mmap 或者 lseek 存取整个要求的图像.

块设备

> 如同字符设备, 块设备通过位于 /dev 目录的文件系统结点来存取. 一个块设备(例如一个磁盘)应该是可以驻有一个文件系统的. 在大部分的 Unix 系统, 一个块设备只能处理这样的 I/O 操作, 传送一个或多个长度经常是 512 字节(或一个更大的 2 的幂的数)的整块. Linux, 相反, 允许应用程序读写一个块设备象一个字符设备一样 -- 它允许一次传送任意数目的字节. 结果就是, 块和字符设备的区别仅仅在内核在内部管理数据的方式上, 并且因此在内核/驱动的软件接口上不同. 如同一个字符设备, 每个块设备都通过一个文件系统结点被存取的, 它们之间的区别对用户是透明的. 块驱动和字符驱动相比, 与内核的接口完全不同.

网络接口

> 任何网络事务都通过一个接口来进行, 就是说, 一个能够与其他主机交换数据的设备. 通常, 一个接口是一个硬件设备, 但是它也可能是一个纯粹的软件设备, 比如环回接口. 一个网络接

口负责发送和接收数据报文, 在内核网络子系统的驱动下, 不必知道单个事务是如何映射到实际的被发送的报文上的. 很多网络连接(特别那些使用 TCP 的)是面向流的, 但是网络设备却常常设计成处理报文的发送和接收. 一个网络驱动对单个连接一无所知; 它只处理报文.

有其他的划分驱动模块的方式, 与上面的设备类型是正交的. 通常, 某些类型的驱动与给定类型设备的其他层的内核支持函数一起工作. 例如, 你可以说 USB 模块, 串口模块, SCSI 模块, 等等. 每个 USB 设备由一个 USB 模块驱动, 与 USB 子系统一起工作, 但是设备自身在系统中表现为一个字符设备(比如一个 USB 串口), 一个块设备(一个 USB 内存读卡器), 或者一个网络设备(一个 USB 以太网接口).

另外的设备驱动类别近来已经添加到内核中, 包括 FireWire 驱动和 I2O 驱动. 以它们处理 USB 和 SCSI 驱动相同的方式, 内核开发者集合了类别范围内的特性, 并把它们输出给驱动实现者, 以避免重复工作和 bug, 因此简化和加强了编写类似驱动的过程.

在设备驱动之外, 别的功能, 不论硬件和软件, 在内核中都是模块化的. 一个普通的例子是文件系统. 一个文件系统类型决定了在块设备上信息是如何组织的, 以便能表示一棵目录与文件的树. 这样的实体不是设备驱动, 因为没有明确的设备与信息摆放方式相联系; 文件系统类型却是一种软件驱动, 因为它将低级数据结构映射为高级的数据结构. 文件系统决定一个文件名多长, 以及在一个目录入口中存储每个文件的什么信息. 文件系统模块必须实现最低级的系统调用, 来存取目录和文件, 通过映射文件名和路径(以及其他信息, 例如存取模式)到保存在数据块中的数据结构. 这样的一个接口是完全与数据被传送来去磁盘(或其他介质)相互独立, 这个传送是由一个块设备驱动完成的.

如果你考虑一个 Unix 系统是多么依赖下面的文件系统, 你会认识到这样的一个软件概念对系统操作是至关重要的. 解码文件系统信息的能力处于内核层级中最低级, 并且是最重要的; 甚至如果你为你的新 CD-ROM 编写块驱动, 如果你对上面的数据不能运行 ls 或者 cp 就毫无用处. Linux 支持一个文件系统模块的概念, 其软件接口声明了不同操作, 可以在一个文件系统节点, 目录, 文件和超级块上进行操作. 对一个程序员来说, 居然需要编写一个文件系统模块是非常不常见的, 因为官方内核已经包含了大部分重要的文件系统类型的代码.

1.4. 安全问题

安全是当今重要性不断增长的关注点. 我们将讨论安全相关的问题, 在它们在本书中出现时. 有几个通用的概念, 却值得现在提一下.

系统中任何安全检查都由内核代码强加上去. 如果内核有安全漏洞, 系统作为一个整体就有漏洞. 在官方的内核发布里, 只有一个有授权的用户可以加载模块; 系统调用 init_module 检查调用进程是否是有权加载模块到内核里. 因此, 当运行一个官方内核时, 只有超级用户[1]或者一个成功获得特权的入侵者, 才可以利用特权代码的能力.

在可能时, 驱动编写者应当避免将安全策略编到他们的代码中. 安全是一个策略问题, 最好在内核高层来处理, 在系统管理员的控制下. 但是, 常有例外.

作为一个设备驱动编写者, 你应当知道在什么情形下, 某些类型的设备存取可能反面地影响系统作为一个整体, 并且应当提供足够地控制. 例如, 会影响全局资源的设备操作(例如设置一条中断线), 可能会损坏硬件(例如, 加载固件), 或者它可能会影响其他用户(例如设置一个磁带驱动的缺省的块大小), 常常是只对有足够授权的用户, 并且这种检查必须由驱动自身进行.

驱动编写者也必须要小心, 当然, 来避免引入安全 bug. C 编程语言使得易于犯下几类的错误. 例如, 许多现今的安全问题是由于缓冲区覆盖引起, 它是由于程序员忘记检查有多少数据写入缓冲区, 数据在缓冲区结尾之外结束, 因此覆盖了无关的数据. 这样的错误可能会危及整个系统的安全, 必须避免. 幸运的是, 在设备驱动上下文中避免这样的错误经常是相对容易的, 这里对用户的接口经过精细定义并被高度地控制.

一些其他的通用的安全观念也值得牢记. 任何从用户进程接收的输入应当以极大的怀疑态度来对待; 除非你能核实它, 否则不要信任它. 小心对待未初始化的内存; 从内核获取的任何内存应当清零或者在其对用户进程或设备可用之前进行初始化. 否则, 可能发生信息泄漏(数据, 密码的暴露等等). 如果你的设备解析发送给它的数据, 要确保用户不能发送任何能危及系统的东西. 最后, 考虑一下设备操作的可能后果; 如果有特定的操作(例如, 加载一个适配卡的固件或者格式化一个磁盘), 能影响到系统的, 这些操作应该完全确定地要限制在授权的用户中.

也要小心, 当从第三方接收软件时, 特别是与内核有关: 因为每个人都可以接触到源码, 每个人都可以分拆和重组东西. 尽管你能够信任在你的发布中的预编译的内核, 你应当避免运行一个由不能信任的朋友编译的内核 -- 如果你不能作为 root 运行预编译的二进制文件, 那么你最好不要运行一个预编译的内核. 例如, 一个经过了恶意修改的内核可能会允许任何人加载模块, 这样就通过 init_module 开启了一个不想要的后门.

注意, Linux 内核可以编译成不支持任何属于模块的东西, 因此关闭了任何模块相关的安全漏洞. 在这种情况下, 当然, 所有需要的驱动必须直接建立到内核自身内部. 在 2.2 和以后的内核, 也可以在系统启动之后, 通过 capability 机制来禁止内核模块的加载.

---

[1] 从技术上讲, 只有具有 CAP_SYS_MODULE 权利的人才可以进行这个操作. 我们第 6 章讨论 capabilities .

上一页 1.3. 设备和模块的分类 | 上一级 起始页 | 下一页 1.5. 版本编号

1.5. 版本编号

在深入编程之前, 我们应当对 Linux 使用的版本编号方法和本书涉及的版本做些说明.

首先, 注意的是在 Linux 系统中使用的每一个软件包有自己的发行版本号, 它们之间存在相互依赖性: 你需要一个包的特别的版本来运行另外一个包的特别版本. Linux 发布的创建者常常要处理匹配软件包的繁琐问题, 这样用户从一个已打包好的发布中安装就不需要处理版本号的问题了. 另外, 那些替换和更新系统软件的人, 就要自己处理这个问题了. 幸运的是, 几乎所有的现代发布支持单个软件包的更新, 通过检查软件包之间的依赖性; 发布的软件包管理器通常不允许更新, 直到满足了依赖性.

为了运行我们在讨论过程中介绍的例子, 你除了 2.6 内核要求的之外不需要任何工具的特别版本; 任何近期的 Linux 发布都可以用来运行我们的例子. 我们不详述特别的要求, 因为你内核源码中的文件 Document/Changes 是这种信息的最好的来源, 如果你遇到任何问题.

至于说内核, 偶数的内核版本(就是说, 2.6.x)是稳定的, 用来做通用的发布. 奇数版本(例如 2.7.x), 相反, 是开发快照并且是非常短暂的; 它们的最新版本代表了开发的当前状态, 但是会在几天内就过时了.

本书涵盖内核 2.6 版本. 我们的目标是为设备驱动编写者展示 2.6.10 内核的所有可用的特性, 这是我们在编写本书时的内核版本. 本书的这一版不涉及内核的其他版本. 你们有人感兴趣的话, 本书第 2 版详细涵盖 2.0 到 2.4 版本. 那个版本依然在 htt://lwn.net/Kernel/LDD2 在线获取到.

内核程序员应当明白到 2.6 内核的开发过程的改变. 2.6 系列现在接受之前可能认为对一个"稳定"的内核太大的更改. 在其他的方面, 这意味着内核内部编程接口可能改变, 因此潜在地会使本书某些部分过时; 基于这个原因, 伴随着文本的例子代码已知可以在 2.6.10 上运行, 但是某些模块没有在之前的版本上编译. 想紧跟内核编程变化的程序员最好加入邮件列表, 并且利用列在参考书目中的网站. 也有一个网页在 http://lwn.net/Articls/2.6-kernel-api 上维护, 它包含自本书出版以来的 API 改变的信息.

本文不特别地谈论奇数内核版本. 普通用户不会有理由运行开发中的内核. 试验新特性的开发者, 但是, 想运行最新的开发版本. 他们常常不停更新到最新的版本, 来收集 bug 的修正和新的特性实现. 但是注意, 试验中的内核没有任何保障[2], 如果你由于一个非当前的奇数版本内核的一个 bug 而引起的问题, 没人可以帮你. 那些运行奇数版本内核的人常常是足够熟练的深入到代码中, 不需要一本教科书, 这也是我们为什么不谈论开发中的内核的另一个原因.

Linux 的另一个特性是它是平台独立的操作系统, 并非仅仅是" PC 克隆体的一种 Unix 克隆 ", 更多

的: 它当前支持大约 20 种体系. 本书是尽可能地平台独立, 所有的代码例子至少是在 x86 和 x86-64 平台上测试过. 因为代码已经在 32-bit 和 64-bit 处理器上测试过, 它应当能够在所有其他平台上编译和运行. 如同你可能期望地, 依赖特殊硬件的代码例子不会在所有支持的平台上运行, 但是这个通常在源码里说明了.

---

[2] 注意, 对于偶数版本的内核也不存在保证, 除非你依靠一个同意提供它自己的担保的商业供应商.

上一页 1.4. 安全问题 | 上一级 起始页 | 下一页 1.6. 版权条款

1.6. 版权条款

Linux 是以 GNU 通用公共版权(GPL)的版本 2 作为许可的, 它来自自由软件基金的 GNU 项目. GPL 允许任何人重发布, 甚至是销售, GPL 涵盖的产品, 只要接收方对源码能存取并且能够行使同样的权力. 另外, 任何源自使用 GPL 产品的软件产品, 如果它是完全的重新发布, 必须置于 GPL 之下发行.

这样一个许可的主要目的是允许知识的增长, 通过同意每个人去任意修改程序; 同时, 销售软件给公众的人仍然可以做他们的工作. 尽管这是一个简单的目标, 关于 GPL 和它的使用存在着从未结束的讨论. 如果你想阅读这个许可证, 你能够在你的系统中几个地方发现它, 包括你的内核源码树的目录中的 COPYING 文件.

供应商常常询问他们是否可以只发布二进制形式的内核模块. 对这个问题的答案已是有意让它模糊不清. 二进制模块的发布 -- 只要它们依附预已公布的内核接口 -- 至今已是被接受了. 但是内核的版权由许多开发者持有, 并且他们不是全都同意内核模块不是衍生产品. 如果你或者你的雇主想在非自由的许可下发布内核模块, 你真正需要的是和你的法律顾问讨论. 请注意内核开发者不会对于在内核发行之间破坏二进制模块有任何疑虑, 甚至在一个稳定的内核系列之间. 如果它根本上是可能的, 你和你的用户最好以自由软件的方式发行你的模块.

如果你想你的代码进入主流内核, 或者如果你的代码需要对内核的补丁, 你在发行代码时, 必须立刻使用一个 GPL 兼容的许可. 尽管个人使用你的改变不需要强加 GPL, 如果你发布你的代码, 你必须包含你的代码到发布里面 -- 要求你的软件包的人必须被允许任意重建二进制的内容.

至于本书, 大部分的代码是可自由地重新发布, 要么是源码形式, 要么是二进制形式, 我们和 O'Reilly 都不保留任何权利对任何的衍生的工作. 所有的程序都可从 ftp://ftp.ora.com/pub/examples/linux/drivers/ 得到, 详尽的版权条款在相同目录中的 LICENSE 文件里阐述.

1.7. 加入内核开发社团

在你开始为 Linux 内核编写模块时, 你就成为一个开发者大社团的一部分. 在这个社团中, 你不仅会发现有人忙碌于类似工作, 还有一群特别投入的工程师努力使 Linux 成为更好的系统. 这些人可以是帮助, 理念, 以及关键的审查的来源, 以及他们将是你愿意求助的第一类人, 当你在寻找一个新驱动的测试者.

对于 Linux 内核开发者, 中心的汇聚点是 Linux 内核邮件列表. 所有主要的内核开发者, 从 Linus Torvalds 到其他人, 都订阅这个列表. 请注意这个列表不适合心力衰弱的人: 每天或者几天内的书写流量可能多至 200 条消息. 但是, 随这个列表之后的是对那些感兴趣于内核开发的人重要的东西; 它也是一个最高品质的资源, 对那些需要内核开发帮助的人.

为加入 Linux 内核列表, 遵照在 Linux 内核邮件列表 FAQ: http://www.tux.org/lkml 中的指示. 阅读这个 FAQ 的剩余部分, 当你熟悉它时; 那里有大量的有用的信息. Linux 内核开发者都是忙碌的人, 他们更多地愿意帮助那些已经清楚地首先完成了属于自己的那部分工作的人.

1.8. 本书的内容

从这里开始, 我们进入内核编程的世界. 第 2 章介绍了模块化, 解释了内部的秘密以及展示了运行模块的代码. 第 2 章谈论字符驱动以及展示一个基于内存的设备驱动的代码, 出于乐趣对它读写. 使用内存作为设备的硬件基础使得任何人可以不用要求特殊的硬件来运行代码.

调试技术对程序员是必备的工具, 第 4 章介绍它. 对那些想分析当前内核的人同样重要的是并发的管理和竞争情况. 第 5 章关注的是由于并发存取资源而导致的问题, 并且介绍控制并发的 Linux 机制.

在具备了调试和并发管理的能力下, 我们转向字符驱动的高级特性, 例如阻塞操作, selet 的使用, 以及重要的 ioctl 调用; 这是第 6 章的主题.

在处理硬件管理之前, 我们研究多一点内核软件接口: 第 7 章展示了内核中是如何管理时间的, 第 8 章讲解了内存分配.

接下来我们集中到硬件. 第 9 章描述了 I/O 口的管理和设备上的内存缓存; 随后是中断处理, 在 第 10 章. 不幸的是, 不是每个人都能运行这些章节中的例子代码, 因为确实需要某些硬件来测试软件接口中断. 我们尽力保持需要的硬件支持到最小程度, 但是你仍然需要某些硬件, 例如标准并口, 来使用这些章节的例子代码.

第 11 章涉及内核数据类型的使用, 以及编写可移植代码.

本书的第 2 半专注于更高级的主题. 我们从深入硬件内部开始, 特别的, 是特殊外设总线功能. 第 12 章涉及编写 PCI 设备驱动, 第 13 章检验使用 USB 设备的 API.

具有了对外设总线的理解, 我们详细看一下 Linux 设备模型, 这是内核使用的抽象层来描述它管理的硬件和软件资源. 第 14 章是一个自底向上的设备模型框架的考察, 从 kobject 类型开始以及从那里进一步进行. 它涉及设备模型与真实设备的集成; 接下来是利用这些知识来接触如热插拔设备和电源管理等主题.

在第 15 章, 我们转移到 Linux 的内存管理. 这一章显示如何映射系统内存到用户空间(mmap 系统调用), 映射用户内存到内核空间(使用 get_user_pages), 以及如何映射任何一种内存到设备空间(进行 直接内存存取 [DMA] 操作).

我们对内存的理解将对下面两章是有用的, 它们涉及到其他主要的驱动类型. 第 16 章介绍了块驱动, 并展示了与我们到现在为止已遇到过的字符驱动的区别. 第 17 章进入网络驱动的编写. 我们最

后是讨论串行驱动(第 18 章)和一个参考书目.

上一页 上一级 下一页
1.7. 加入内核开发社团 起始页 第 2 章 建立和运行模块

第 2 章 建立和运行模块

上一页 下一页

第 2 章 建立和运行模块

目录

时间差不多该开始编程了. 本章介绍所有的关于模块和内核编程的关键概念. 在这几页里, 我们建立并运行一个完整(但是相对地没有什么用处)的模块, 并且查看一些被所有模块共用的基本代码. 开发这样的专门技术对任何类型的模块化的驱动都是重要的基础. 为避免一次抛出太多的概念, 本章只论及模块, 不涉及任何特别的设备类型.

在这里介绍的所有的内核项 (函数, 变量, 头文件, 和宏)在本章的结尾的参考一节里有说明.

2.1. 设置你的测试系统

在本章开始, 我们提供例子模块来演示编程概念. (所有的例子都可从 O' Reilly' s 的 FTP 网站上得到, 如第 1 章解释的那样)建立, 加载, 和修改这些例子, 是提高你对驱动如何工作以及如何与内核交互的理解的好方法.

例子模块应该可以在大部分的 2.6.x 内核上运行, 包括那些由发布供应商提供的. 但是, 我们建议你获得一个主流内核, 直接从 kernel.org 的镜像网络, 并把它安装到你的系统中. 供应商的内核可能是主流内核被重重地打了补丁并且和主流内核有分歧; 偶尔, 供应商的补丁可能改变了设备驱动可见的内核 API. 如果你在编写一个必须在特别的发布上运行的驱动, 你当然要在相应的内核上建立和测试. 但是, 处于学习驱动编写的目的, 一个标准内核是最好的.

不管你的内核来源, 建立 2.6.x 的模块需要你有一个配置好并建立好的内核树在你的系统中. 这个要求是从之前内核版本的改变, 之前只要有一套当前版本的头文件就足够了. 2.6 模块针对内核源码树里找到的目标文件连接; 结果是一个更加健壮的模块加载器, 还要求那些目标文件也是可用的. 因此你的第一个商业订单是具备一个内核源码树(或者从 krenel.org 网络或者你的发布者的内核源码包), 建立一个新内核, 并且安装到你的系统. 因为我们稍后会见到的原因, 生活通常是最容易的如果当你建立模块时真正运行目标内核, 尽管这不是需要的.

注意

你应当也考虑一下在哪里进行你的模块试验, 开发和测试. 我们已经尽力使我们的例子模块安全和正确, 但是 bug 的可能性是经常会有的. 内核代码中的错误可能会引起一个用户进程的死亡, 或者偶尔, 瘫痪整个系统. 它们正常地不会导致更严重地后果, 例如磁盘损伤. 然而, 还是建议你进行你的内核试验在一个没有包含你负担不起丢失的数据的系统, 并且没有进行重要的服务. 内核开发者典型地会保留一台"牺牲"系统来测试新的代码.

因此, 如果你还没有一个合适的系统, 带有一个配置好并建立好的源码树在磁盘上, 现在是时候建立了. 我们将等待. 一旦这个任务完成, 你就准备好开始摆布内核模块了.

2.2. Hello World 模块

许多编程书籍从一个 "hello world" 例子开始, 作为一个展示可能的最简单的程序的方法. 本书涉及的是内核模块而不是程序; 因此, 对无耐心的读者, 下面的代码是一个完整的 "hello world"模块:

```
#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);
```

这个模块定义了两个函数, 一个在模块加载到内核时被调用(hello_init)以及一个在模块被去除时被调用(hello_exit). moudle_init 和 module_exit 这几行使用了特别的内核宏来指出这两个函数的角色. 另一个特别的宏 (MODULE_LICENSE) 是用来告知内核, 该模块带有一个自由的许可证; 没有这样的说明, 在模块加载时内核会抱怨.

printk 函数在 Linux 内核中定义并且对模块可用; 它与标准 C 库函数 printf 的行为相似. 内核需要它自己的打印函数, 因为它靠自己运行, 没有 C 库的帮助. 模块能够调用 printk 是因为, 在 insmod 加载了它之后, 模块被连接到内核并且可存取内核的公用符号 (函数和变量, 下一节详述). 字串 KERN_ALERT 是消息的优先级. [3]

我们在此模块中指定了一个高优先级, 因为使用缺省优先级的消息可能不会在任何有用的地方显示, 这依赖于你运行的内核版本, klogd 守护进程的版本, 以及你的配置. 现在你可以忽略这个因素; 我们在第 4 章讲解它.

你可以用 insmod 和 rmmod 工具来测试这个模块. 注意只有超级用户可以加载和卸载模块.

```
% 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
Goodbye cruel world
root#
```

请再一次注意, 为使上面的操作命令顺序工作, 你必须在某个地方有正确配置和建立的内核树, 在那里可以找到 makefile (/usr/src/linux-2.6.10, 在展示的例子里面). 我们在 "编译和加载" 这一节深入模块建立的细节.

依据你的系统用来递交消息行的机制, 你的输出可能不同. 特别地, 前面的屏幕输出是来自一个字符控制台; 如果你从一个终端模拟器或者在窗口系统中运行 insmod 和 rmmod, 你不会在你的屏幕上看到任何东西. 消息进入了其中一个系统日志文件中, 例如 /var/log/messages (实际文件名子随 Linux 发布而变化). 内核递交消息的机制在第 4 章描述.

如你能见到的, 编写一个模块不是如你想象的困难 -- 至少, 在模块没有要求做任何有用的事情时. 困难的部分是理解你的设备, 以及如何获得最高性能. 通过本章我们深入模块化内部并且将设备相关的问题留到后续章节.

---

[3] 优先级只是一个字串, 例如 <1>, 前缀于 printk 格式串之前. 注意在 KERN_ALERT 之后缺少一个逗号; 添加一个逗号在那里是一个普通的讨厌的错误 (幸运的是, 编译器会捕捉到).

2.3. 内核模块相比于应用程序

在我们深入之前, 有必要强调一下内核模块和应用程序之间的各种不同.

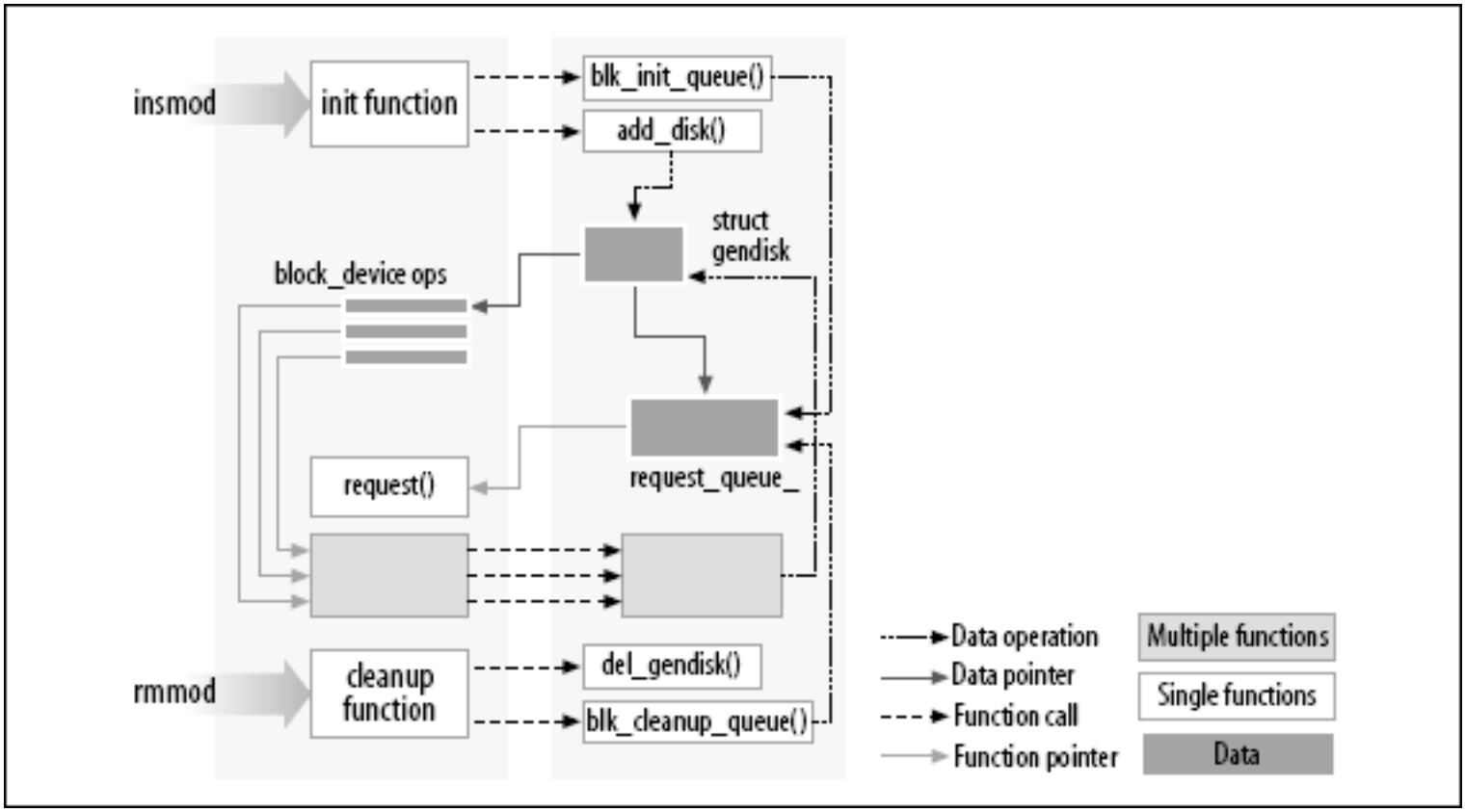
不同于大部分的小的和中型的应用程序从头至尾处理一个单个任务, 每个内核模块只注册自己以便来服务将来的请求, 并且它的初始化函数立刻终止. 换句话说, 模块初始化函数的任务是为以后调用模块的函数做准备; 好像是模块说, " 我在这里, 这是我能做的."模块的退出函数(例子里是 hello_exit)就在模块被卸载时调用. 它好像告诉内核, "我不再在那里了, 不要要求我做任何事了."这种编程的方法类似于事件驱动的编程, 但是虽然不是所有的应用程序都是事件驱动的, 每个内核模块都是. 另外一个主要的不同, 在事件驱动的应用程序和内核代码之间, 是退出函数: 一个终止的应用程序可以在释放资源方面懒惰, 或者完全不做清理工作, 但是模块的退出函数必须小心恢复每个由初始化函数建立的东西, 否则会保留一些东西直到系统重启.

偶然地, 卸载模块的能力是你将最欣赏的模块化的其中一个特色, 因为它有助于减少开发时间; 你可测试你的新驱动的连续的版本, 而不用每次经历漫长的关机/重启周期.

作为一个程序员, 你知道一个应用程序可以调用它没有定义的函数: 连接阶段使用合适的函数库解决了外部引用. printf 是一个这种可调用的函数并且在 libc 里面定义. 一个模块, 在另一方面, 只连接到内核, 它能够调用的唯一的函数是内核输出的那些; 没有库来连接.在 hello.c 中使用的 printk 函数, 例如, 是在内核中定义的 printf 版本并且输出给模块. 它表现类似于原始的函数, 只有几个小的不同, 首要的一个是缺乏浮点的支持.

图 连接一个模块到内核 展示了函数调用和函数指针在模块中如何使用来增加新功能到一个运行中的内核.

图 2.1. 连接一个模块到内核



因为没有库连接到模块中, 源文件不应当包含通常的头文件, <stdarg.h>和非常特殊的情况是仅有的例外. 只有实际上是内核的一部分的函数才可以在内核模块里使用. 内核相关的任何东西都在头文件里声明, 这些头文件在你已建立和配置的内核源码树里; 大部分相关的头文件位于 include/linux 和 include/asm, 但是别的 include 的子目录已经添加到关联特定内核子系统的材料里了.

单个内核头文件的作用在书中需要它们的时候进行介绍.

另外一个在内核编程和应用程序编程之间的重要不同是每一个环境是如何处理错误: 在应用程序开发中段错误是无害的, 一个调试器常常用来追踪错误到源码中的问题, 而一个内核错误至少会杀掉当前进程, 如果不终止整个系统. 我们会在第 4 章看到如何跟踪内核错误.

2.3.1. 用户空间和内核空间

A module runs in kernel space, whereas applications run in user space. This concept is at the base of operating systems theory. 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.

操作系统的角色, 实际上, 是给程序提供一个一致的计算机硬件的视角. 另外, 操作系统必须承担程序的独立操作和保护对于非授权的资源存取. 这一不平凡的任务只有 CPU 增强系统软件对应用程序的保护才有可能.

每种现代处理器都能够加强这种行为. 选中的方法是 CPU 自己实现不同的操作形态(或者级别). 这

些级别有不同的角色, 一些操作在低些级别中不允许; 程序代码只能通过有限的几个门从一种级别切换到另一个. Unix 系统设计成利用了这种硬件特性, 使用了两个这样的级别. 所有当今的处理器至少有两个保护级别, 并且某些, 例如 x86 家族, 有更多级别; 当几个级别存在时, 使用最高和最低级别. 在 Unix 下, 内核在最高级运行(也称之为超级模式), 这里任何事情都允许, 而应用程序在最低级运行(所谓的用户模式), 这里处理器控制了对硬件的直接存取以及对内存的非法存取.

我们常常提到运行模式作为内核空间和用户空间. 这些术语不仅包含存在于这两个模式中不同特权级别, 还包含有这样的事实, 即每个模式有它自己的内存映射 -- 它自己的地址空间.

Unix 从用户空间转换执行到内核空间, 无论何时一个应用程序发出一个系统调用或者被硬件中断挂起时. 执行系统调用的内核代码在进程的上下文中工作 -- 它代表调用进程并且可以存取该进程的地址空间. 换句话说, 处理中断的代码对进程来说是异步的, 不和任何特别的进程有关.

模块的角色是扩展内核的功能; 模块化的代码在内核空间运行. 经常地一个驱动进行之前提到的两种任务: 模块中一些的函数作为系统调用的一部分执行, 一些负责中断处理.

2.3.2. 内核的并发

内核编程与传统应用程序编程方式很大不同的是并发问题. 大部分应用程序, 多线程的应用程序是一个明显的例外, 典型地是顺序运行的, 从头至尾, 不必要担心其他事情会发生而改变它们的环境. 内核代码没有运行在这样的简单世界中, 即便最简单的内核模块必须在这样的概念下编写, 很多事情可能马上发生.

内核编程中有几个并发的来源. 自然的, Linux 系统运行多个进程, 在同一时间, 不止一个进程能够试图使用你的驱动. 大部分设备能够中断处理器; 中断处理异步运行, 并且可能在你的驱动试图做其他事情的同一时间被调用. 几个软件抽象(例如内核定时器, 第 7 章介绍)也异步运行. 而且, 当然, Linux 可以在对称多处理器系统(SMP)上运行, 结果是你的驱动可能在多个 CPU 上并发执行. 最后, 在 2.6, 内核代码已经是可抢占的了; 这个变化使得即便是单处理器会有许多与多处理器系统同样的并发问题.

结果, Linux 内核代码, 包括驱动代码, 必须是可重入的 -- 它必须能够同时在多个上下文中运行. 数据结构必须小心设计以保持多个执行线程分开, 并且代码必须小心存取共享数据, 避免数据的破坏. 编写处理并发和避免竞争情况(一个不幸的执行顺序导致不希望的行为的情形)的代码需要仔细考虑并可能是微妙的. 正确的并发管理在编写正确的内核代码时是必须的; 由于这个理由, 本书的每一个例子驱动都是考虑了并发下编写的. 用到的技术在我们遇到它们时再讲解; 第 5 章也专门讲述这个问题, 以及并发管理的可用的内核原语.

驱动程序员的一个通常的错误是假定并发不是一个问题, 只要一段特别的代码没有进入睡眠(或者 "阻塞"). 即便在之前的内核(不可抢占), 这种假设在多处理器系统中也不成立. 在 2.6, 内核代码不能(极少)假定它能在一段给定代码上持有处理器. 如果你不考虑并发来编写你的代码, 就极有可能导致严重失效, 以至于非常难于调试.

2.3.3. 当前进程

尽管内核模块不象应用程序一样顺序执行, 内核做的大部分动作是代表一个特定进程的. 内核代码可以引用当前进程, 通过存取全局项 current, 它在 <asm/current.h> 中定义, 它产生一个指针指向结构 task_struct, 在 <linux/sched.h> 定义. current 指针指向当前在运行的进程. 在一个系统调用执行期间, 例如 open 或者 read, 当前进程是发出调用的进程. 内核代码可以通过使用 current 来使用进程特定的信息, 如果它需要这样. 这种技术的一个例子在第 6 章展示.

实际上, current 不真正地是一个全局变量. 支持 SMP 系统的需要强迫内核开发者去开发一种机制, 在相关的 CPU 上来找到当前进程. 这种机制也必须快速, 因为对 current 的引用非常频繁地发生. 结果就是一个依赖体系的机制, 常常, 隐藏了一个指向 task_struct 的指针在内核堆栈内. 实现的细节对别的内核子系统保持隐藏, 一个设备驱动可以只包含 <linux/sched.h> 并且引用当前进程. 例如, 下面的语句打印了当前进程的进程 ID 和命令名称, 通过存取结构 task_struct 中的某些字段.

```
printk(KERN_INFO "The process is \"%s\" (pid %i)\n", current->comm, current->pid);
```

存于 current->comm 的命令名称是由当前进程执行的程序文件的基本名称(截短到 15 个字符, 如果需要).

2.3.4. 几个别的细节

内核编程与用户空间编程在许多方面不同. 我们将在本书的过程中指出它们, 但是有几个基础性的问题, 尽管没有保证它们自己有一节内容, 也值得一提. 因此, 当你深入内核时, 下面的事项应当牢记.

应用程序存在于虚拟内存中, 有一个非常大的堆栈区. 堆栈, 当然, 是用来保存函数调用历史以及所有的由当前活跃的函数创建的自动变量. 内核, 相反, 有一个非常小的堆栈; 它可能小到一个, 4096 字节的页. 你的函数必须与这个内核空间调用链共享这个堆栈. 因此, 声明一个巨大的自动变量从来就不是一个好主意; 如果你需要大的结构, 你应当在调用时间内动态分配.

常常, 当你查看内核 API 时, 你会遇到以双下划线(__)开始的函数名. 这样标志的函数名通常是一个低层的接口组件, 应当小心使用. 本质上讲, 双下划线告诉程序员:" 如果你调用这个函数, 确信你知道你在做什么."

内核代码不能做浮点算术. 使能浮点将要求内核在每次进出内核空间的时候保存和恢复浮点处理器的状态 -- 至少, 在某些体系上. 在这种情况下, 内核代码真的没有必要包含浮点, 额外的负担不值得.

2.4. 编译和加载

本章开头的 "hello world" 例子包含了一个简短的建立并加载模块到系统中去的演示. 当然, 整个过程比我们目前看到的多. 本节提供了更多细节关于一个模块作者如何将源码转换成内核中的运行的子系统.

2.4.1. 编译模块

第一步, 我们需要看一下模块如何必须被建立. 模块的建立过程与用户空间的应用程序的建立过程有显著不同; 内核是一个大的, 独立的程序, 对于它的各个部分如何组合在一起有详细的明确的要求. 建立过程也与以前版本的内核的过程不同; 新的建立系统用起来更简单并且产生更正确的结果, 但是它看起来与以前非常不同. 内核建立系统是一头负责的野兽, 我们就看它一小部分. 在内核源码的 Document/kbuild 目录下发现的文件, 任何想理解表面之下的真实情况的人都要阅读一下.

有几个前提, 你必须在能建立内核模块前解决. 第一个是保证你有版本足够新的编译器, 模块工具, 以及其他必要工具. 在内核文档目录下的文件 Documentation/Changes 一直列出了需要的工具版本; 你应当在向前走之前参考一下它. 试图建立一个内核(包括它的模块), 用错误的工具版本, 可能导致不尽的奇怪的难题. 注意, 偶尔地, 编译器的版本太新可能会引起和太老的版本引起的一样的问题. 内核源码对于编译器做了很大的假设, 新的发行版本有时会一时地破坏东西.

如果你仍然没有一个内核树在手边, 或者还没有配置和建立内核, 现在是时间去做了. 没有源码树在你的文件系统上, 你无法为 2.6 内核建立可加载的模块. 实际运行为其而建立的内核也是有帮助的(尽管不是必要的).

一旦你已建立起所有东西, 给你的模块创建一个 makefile 就是直截了当的. 实际上, 对于本章前面展示的" hello world" 例子, 单行就够了:

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

熟悉 make , 但是对 2.6 内核建立系统不熟悉的读者, 可能奇怪这个 makefile 如何工作. 毕竟上面的这一行不是一个传统的 makefile 的样子. 答案, 当然, 是内核建立系统处理了余下的工作. 上面的安排(它利用了由 GNU make 提供的扩展语法)表明有一个模块要从目标文件 hello.o 建立. 在从目标文件建立后结果模块命名为 hello.ko.

反之, 如果你有一个模块名为 module.ko, 是来自 2 个源文件(姑且称之为, file1.c 和 file2.c), 正确的书写应当是:

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

对于一个象上面展示的要工作的 makefile, 它必须在更大的内核建立系统的上下文被调用. 如果你的内核源码数位于, 假设, 你的 ~/kernel-2.6 目录, 用来建立你的模块的 make 命令(在包含模块源码和 makefile 的目录下键入)会是:

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

这个命令开始是改变它的目录到用 -C 选项提供的目录下(就是说, 你的内核源码目录). 它在那里会发现内核的顶层 makefile. 这个 M= 选项使 makefile 在试图建立模块目标前, 回到你的模块源码目录. 这个目标, 依次地, 是指在 obj-m 变量中发现的模块列表, 在我们的例子里设成了 module.o.

键入前面的 make 命令一会儿之后就会感觉烦, 所以内核开发者就开发了一种 makefile 方式, 使得生活容易些对于那些在内核树之外建立模块的人. 这个窍门是如下书写你的 makefile:

```
# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)

 obj-m := hello.o
# Otherwise we were called directly from the command
# line; invoke the kernel build system.
else

 KERNELDIR ?= /lib/modules/$(shell uname -r)/build
 PWD := $(shell pwd)
default:
 $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif
```

再一次, 我们看到了扩展的 GNU make 语法在起作用. 这个 makefile 在一次典型的建立中要被读 2 次. 当从命令行中调用这个 makefile , 它注意到 KERNELRELEASE 变量没有设置. 它利用这样一个事实来定位内核源码目录, 即已安装模块目录中的符号连接指回内核建立树. 如果你实际上没有运行你在为其而建立的内核, 你可以在命令行提供一个 KERNELDIR= 选项, 设置 KERNELDIR 环境变量, 或者重写 makefile 中设置 KERNELDIR 的那一行. 一旦发现内核源码树, makefile 调用 default: 目标, 来运行第 2 个 make 命令(在 makefile 里参数化成 $(MAKE))象前面描述过的一样来调用内核建立系统. 在第 2 次读, makefile 设置 obj-m, 并且内核的 makefile 文件完成实际的建立模块工作.

这种建立模块的机制你可能感觉笨拙模糊. 一旦你习惯了它, 但是, 你很可能会欣赏这种已经编排进内核建立系统的能力. 注意, 上面的不是一个完整的 makefile; 一个真正的 makefile 包含通常的目

标类型来清除不要的文件, 安装模块等等. 一个完整的例子可以参考例子代码目录的 makefile.

2.4.2. 加载和卸载模块

模块建立之后, 下一步是加载到内核. 如我们已指出的, insmod 为你完成这个工作. 这个程序加载模块的代码段和数据段到内核, 接着, 执行一个类似 ld 的函数, 它连接模块中任何未解决的符号连接到内核的符号表上. 但是不象连接器, 内核不修改模块的磁盘文件, 而是内存内的拷贝. insmod 接收许多命令行选项(详情见 manpage), 它能够安排值给你模块中的参数, 在连接到当前内核之前. 因此, 如果一个模块正确设计了, 它能够在加载时配置; 加载时配置比编译时配置给了用户更多的灵活性, 有时仍然在用. 加载时配置在本章后面的 "模块参数" 一节讲解.

感兴趣的读者可能想看看内核如何支持 insmod: 它依赖一个在 kernel/module.c 中定义的系统调用. 函数 sys_init_module 分配内核内存来存放模块 (这个内存用 vmalloc 分配; 看第 8 章的 "vmalloc 和其友"); 它接着拷贝模块的代码段到这块内存区, 借助内核符号表解决模块中的内核引用, 并且调用模块的初始化函数来启动所有东西.

如果你真正看了内核代码, 你会发现系统调用的名子以 sys_ 为前缀. 这对所有系统调用都是成立的, 并且没有别的函数. 记住这个有助于在源码中查找系统调用.

modprobe 工具值得快速提及一下. modprobe, 如同 insmod, 加载一个模块到内核. 它的不同在于它会查看要加载的模块, 看是否它引用了当前内核没有定义的符号. 如果发现有, modprobe 在定义相关符号的当前模块搜索路径中寻找其他模块. 当 modprobe 找到这些模块(要加载模块需要的), 它也把它们加载到内核. 如果你在这种情况下代替以使用 insmod , 命令会失败, 在系统日志文件中留下一条 " unresolved symbols "消息.

如前面提到, 模块可以用 rmmod 工具从内核去除. 注意, 如果内核认为模块还在用(就是说, 一个程序仍然有一个打开文件对应模块输出的设备), 或者内核被配置成不允许模块去除, 模块去除会失败. 可以配置内核允许"强行"去除模块, 甚至在它们看来是忙的. 如果你到了需要这选项的地步, 但是, 事情可能已经错的太严重以至于最好的动作就是重启了.

lsmod 程序生成一个内核中当前加载的模块的列表. 一些其他信息, 例如使用了一个特定模块的其他模块, 也提供了. lsmod 通过读取 /proc/modules 虚拟文件工作. 当前加载的模块的信息也可在位于 /sys/module 的 sysfs 虚拟文件系统找到.

2.4.3. 版本依赖

记住, 你的模块代码一定要为每个它要连接的内核版本重新编译 -- 至少, 在缺乏 modversions 时, 这里不涉及因为它们更多的是给内核发布制作者, 而不是开发者. 模块是紧密结合到一个特殊内核版本的数据结构和函数原型上的; 模块见到的接口可能一个内核版本与另一个有很大差别. 当然, 在开发中的内核更加是这样.

内核不只是认为一个给定模块是针对一个正确的内核版本建立的. 建立过程的其中一步是对一个

当前内核树中的文件(称为 vermagic.o)连接你的模块; 这个东东含有相当多的有关要为其建立模块的内核的信息, 包括目标内核版本, 编译器版本, 以及许多重要配置变量的设置. 当尝试加载一个模块, 这些信息被检查与运行内核的兼容性. 如果不匹配, 模块不会加载; 代之的是你见到如下内容:

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

看一下系统日志文件(/var/log/message 或者任何你的系统被配置来用的)将发现导致模块无法加载特定的问题.

如果你需要编译一个模块给一个特定的内核版本, 你将需要使用这个特定版本的建立系统和源码树. 前面展示过的在例子 makefile 中简单修改 KERNELDIR 变量, 就完成这个动作.

内核接口在各个发行之间常常变化. 如果你编写一个模块想用来在多个内核版本上工作(特别地是如果它必须跨大的发行版本), 你可能只能使用宏定义和 #ifdef 来使你的代码正确建立. 本书的这个版本只关心内核的一个主要版本, 因此不会在我们的例子代码中经常见到版本检查. 但是这种需要确实有时会有. 在这样情况下, 你要利用在 linux/version.h 中发现的定义. 这个头文件, 自动包含在 linux/module.h, 定义了下面的宏定义:

UTS_RELEASE

: 这个宏定义扩展成字符串, 描述了这个内核树的版本. 例如, "2.6.10".

LINUX_VERSION_CODE

: 这个宏定义扩展成内核版本的二进制形式, 版本号发行号的每个部分用一个字节表示. 例如, 2.6.10 的编码是 132618 (就是, 0x02060a). [4]有了这个信息, 你可以(几乎是)容易地决定你在处理的内核版本.

KERNEL_VERSION(major,minor,release)

: 这个宏定义用来建立一个整型版本编码, 从组成一个版本号的单个数字. 例如, KERNEL_VERSION(2.6.10) 扩展成 132618. 这个宏定义非常有用, 当你需要比较当前版本和一个已知的检查点.

大部分的基于内核版本的依赖性可以使用预处理器条件解决, 通过利用 KERNEL_VERSION 和 LINUX_VERSION_VODE. 版本依赖不应当, 但是, 用繁多的 #ifdef 条件来搞乱驱动的代码; 处理不兼容的最好的方式是把它们限制到特定的头文件. 作为一个通用的原则, 明显版本(或者平台)依赖的代码应当隐藏在一个低级的宏定义或者函数后面. 高层的代码就可以只调用这些函数, 而不必关心低层的细节. 这样书写的代码易读并且更健壮.

2.4.4. 平台依赖性

每个电脑平台有其自己的特点, 内核设计者可以自由使用所有的特性来获得更好的性能. in the target object file ???

不象应用程序开发者, 他们必须和预编译的库一起连接他们的代码, 依附在参数传递的规定上, 内核开发者可以专用某些处理器寄存器给特别的用途, 他们确实这样做了. 更多的, 内核代码可以为一个 CPU 族里的特定处理器优化, 以最好地利用目标平台; 不象应用程序那样常常以二进制格式发布, 一个定制的内核编译可以为一个特定的计算机系列优化.

例如, IA32 (x86) 结构分为几个不同的处理器类型. 老式的 80386 处理器仍然被支持(到现在), 尽管它的指令集, 以现代的标准看, 非常有限. 这个体系中更加现代的处理器已经引入了许多新特性, 包括进入内核的快速指令, 处理器间的加锁, 拷贝数据, 等等. 更新的处理器也可采用 36 位(或者更大) 的物理地址, 当在适当的模式下, 以允许他们寻址超过 4 GB 的物理内存. 其他的处理器家族也有类似的改进. 内核, 依赖不同的配置选项, 可以被建立来使用这些附加的特性.

清楚地, 如果一个模块与一个给定内核工作, 它必须以与内核相同的对目标处理器的理解来建立. 再一次, vermagic.o 目标文件登场. 当加载一个模块, 内核为模块检查特定处理器的配置选项, 确认它们匹配运行的内核. 如果模块用不同选项编译, 它不会加载.

如果你计划为通用的发布编写驱动, 你可能很奇怪你怎么可能支持所有这些不同的变体. 最好的答案, 当然, 是发行你的驱动在 GPL 兼容的许可之下, 并且贡献它给主流内核. 如果没有那样, 以源码形式和一套脚本发布你的驱动, 以便在用户系统上编译可能是最好的答案. 一些供应商已发行了工具来简化这个工作. 如果你必须发布你的驱动以二进制形式, 你需要查看由你的目标发布所提供的不同的内核, 并且为每个提供一个模块版本. 要确认考虑到了任何在产生发布后可能发行的勘误内核. 接着, 要考虑许可权的问题, 如同我们在第 1 章的" 许可条款" 一节中讨论的. 作为一个通用的规则, 以源码形式发布东西是你行于世的易途.

---

[4] 这允许在稳定版本之间多达 256 个开发版本.

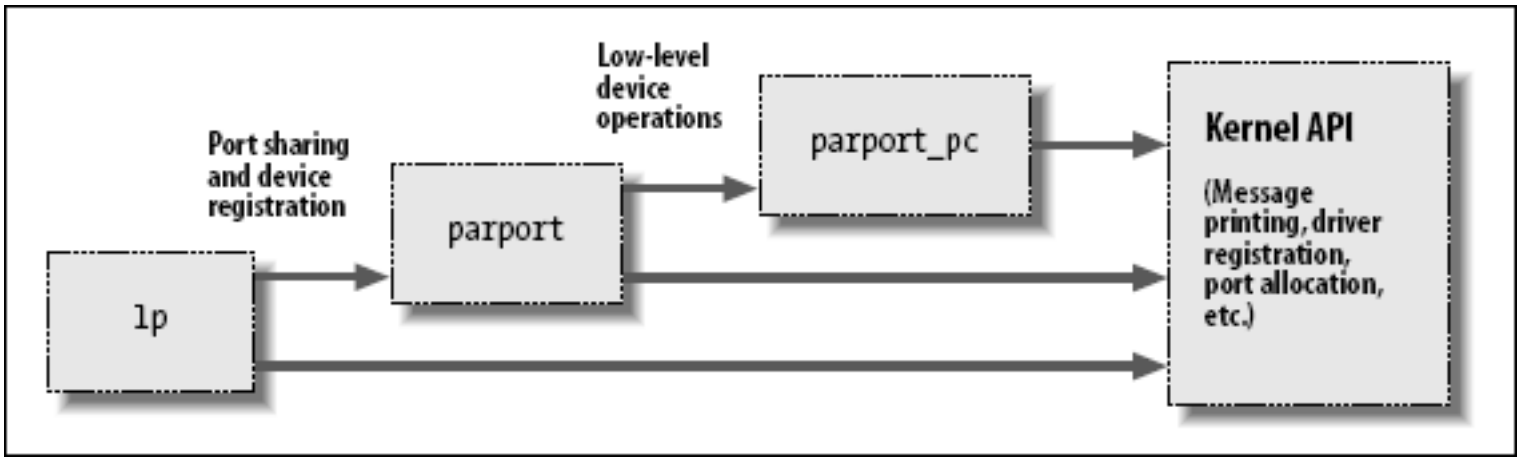
2.5. 内核符号表

我们已经看到 insmod 如何对应共用的内核符号来解决未定义的符号. 表中包含了全局内核项的地址 -- 函数和变量 -- 需要来完成模块化的驱动. 当加载一个模块, 如何由模块输出的符号成为内核符号表的一部分. 通常情况下, 一个模块完成它自己的功能不需要输出如何符号. 你需要输出符号, 但是, 在任何别的模块能得益于使用它们的时候.

新的模块可以用你的模块输出的符号, 你可以堆叠新的模块在其他模块之上. 模块堆叠在主流内核源码中也实现了: msdos 文件系统依赖 fat 模块输出的符号, 某一个输入 USB 设备模块堆叠在 usbcore 和输入模块之上.

模块堆叠在复杂的工程中有用处. 如果一个新的抽象以驱动程序的形式实现, 它可能提供一个特定硬件实现的插入点. 例如, video-for-linux 系列驱动分成一个通用模块, 输出了由特定硬件的低层设备驱动使用的符号. 根据你的设置, 你加载通用的视频模块和你的已安装硬件对应的特定模块. 对并口的支持和众多可连接设备以同样的方式处理, 如同 USB 内核子系统. 在并口子系统的堆叠在图 并口驱动模块的堆叠 中显示; 箭头显示了模块和内核编程接口间的通讯.

图 2.2. 并口驱动模块的堆叠



当使用堆叠的模块时, 熟悉 modprobe 工具是有帮助的. 如我们前面讲的, modprobe 函数很多地方与 insmod 相同, 但是它也加载任何你要加载的模块需要的其他模块. 所以, 一个 modprobe 命令有时可能代替几次使用 insmod(尽管你从当前目录下加载你自己模块仍将需要 insmod, 因为 modprobe 只查找标准的已安装模块目录).

使用堆叠来划分模块成不同层, 这有助于通过简化每一层来缩短开发时间. 这同我们在第 1 章讨论的区分机制和策略是类似的.

linux 内核头文件提供了方便来管理你的符号的可见性, 因此减少了命名空间的污染(将与在内核别处已定义的符号冲突的名子填入命名空间), 并促使了正确的信息隐藏. 如果你的模块需要输出符号给其他模块使用, 应当使用下面的宏定义:

```
EXPORT_SYMBOL(name);
EXPORT_SYMBOL_GPL(name);
```

上面宏定义的任一个使得给定的符号在模块外可用. _GPL 版本的宏定义只能使符号对 GPL 许可的模块可用. 符号必须在模块文件的全局部分输出, 在任何函数之外, 因为宏定义扩展成一个特殊用途的并被期望是全局存取的变量的声明. 这个变量存储于模块的一个特殊的可执行部分(一个 "ELF 段"), 内核用这个部分在加载时找到模块输出的变量. (感兴趣的读者可以看 <linux/module.h> 获知详情, 尽管并不需要这些细节使东西动起来.)

上一页 2.4. 编译和加载 | 上一级 起始页 | 下一页 2.6. 预备知识

2.6. 预备知识

我们正在接近去看一些实际的模块代码. 但是首先, 我们需要看一些需要出现在你的模块源码文件中的东西. 内核是一个独特的环境, 它将它的要求强加于要和它接口的代码上.

大部分内核代码包含了许多数量的头文件来获得函数, 数据结构和变量的定义. 我们将在碰到它们时检查这些文件, 但是有几个文件对模块是特殊的, 必须出现在每一个可加载模块中. 因此, 几乎所有模块代码都有下面内容:

```
#include <linux/module.h>
#include <linux/init.h>
```

moudle.h 包含了大量加载模块需要的函数和符号的定义. 你需要 init.h 来指定你的初始化和清理函数, 如我们在上面的 "hello world" 例子里见到的, 这个我们在下一节中再讲. 大部分模块还包含 moudleparam.h, 使得可以在模块加载时传递参数给模块. 我们将很快遇到.

不是严格要求的, 但是你的模块确实应当指定它的代码使用哪个许可. 做到这一点只需包含一行 MODULE_LICENSE:

```
MODULE_LICENSE("GPL");
```

内核认识的特定许可有, "GPL"(适用 GNU 通用公共许可的任何版本), "GPL v2"(只适用 GPL 版本 2), "GPL and additional rights", "Dual BSD/GPL", "Dual MPL/GPL", 和 "Proprietary". 除非你的模块明确标识是在内核认识的一个自由许可下, 否则就假定它是私有的, 内核在模块加载时被"弄污浊"了. 象我们在第 1 章"许可条款"中提到的, 内核开发者不会热心帮助在加载了私有模块后遇到问题的用户.

可以在模块中包含的其他描述性定义有 MODULE_AUTHOR (声明谁编写了模块), MODULE_DESCRIPION(一个人可读的关于模块做什么的声明), MODULE_VERSION (一个代码修订版本号; 看 <linux/module.h> 的注释以便知道创建版本字串使用的惯例), MODULE_ALIAS (模块为人所知的另一个名子), 以及 MODULE_DEVICE_TABLE (来告知用户空间, 模块支持那些设备). 我们会讨论 MODULE_ALIAS 在第 11 章以及 MUDULE_DEVICE_TABLE 在第 12 章.

各种 MODULE_ 声明可以出现在你的源码文件的任何函数之外的地方. 但是, 一个内核代码中相对近期的惯例是把这些声明放在文件末尾.

上一页 上一级 下一页
2.5. 内核符号表 起始页 2.7. 初始化和关停

2.7. 初始化和关停

如已提到的, 模块初始化函数注册模块提供的任何功能. 这些功能, 我们指的是新功能, 可以由应用程序存取的或者一整个驱动或者一个新软件抽象. 实际的初始化函数定义常常如:

```
static int __init initialization_function(void)
{

 /* Initialization code here */
}
module_init(initialization_function);
```

初始化函数应当声明成静态的, 因为它们不会在特定文件之外可见; 没有硬性规定这个, 然而, 因为没有函数能输出给内核其他部分, 除非明确请求. 声明中的 __init 标志可能看起来有点怪; 它是一个给内核的暗示, 给定的函数只是在初始化使用. 模块加载者在模块加载后会丢掉这个初始化函数, 使它的内存可做其他用途. 一个类似的标签 (__initdata) 给只在初始化时用的数据. 使用 __init 和 __initdata 是可选的, 但是它带来的麻烦是值得的. 只是要确认不要用在那些在初始化完成后还使用的函数(或者数据结构)上. 你可能还会遇到 __devinit 和 __devinitdata 在内核源码里; 这些只在内核没有配置支持 hotplug 设备时转换成 __init 和 _initdata. 我们会在 14 章谈论 hotplug 支持.

使用 moudle_init 是强制的. 这个宏定义增加了特别的段到模块目标代码中, 表明在哪里找到模块的初始化函数. 没有这个定义, 你的初始化函数不会被调用.

模块可以注册许多的不同设施, 包括不同类型的设备, 文件系统, 加密转换, 以及更多. 对每一个设施, 有一个特定的内核函数来完成这个注册. 传给内核注册函数的参数常常是一些数据结构的指针, 描述新设施以及要注册的新设施的名子. 数据结构常常包含模块函数指针, 模块中的函数就是这样被调用的.

能够注册的项目远远超出第 1 章中提到的设备类型列表. 它们包括, 其他的, 串口, 多样设备, sysfs 入口, /proc 文件, 执行域, 链路规程. 这些可注册项的大部分都支持不直接和硬件相关的函数, 但是处于"软件抽象"区域里. 这些项可以注册, 是因为它们以各种方式(例如象 /proc 文件和链路规程)集成在驱动的功能中.

对某些驱动有其他的设施可以注册作为补充, 但它们的使用太特别, 所以不值得讨论它们. 它们使用堆叠技术, 在"内核符号表"一节中讲过. 如果你想深入探求, 你可以在内核源码里查找 EXPORT_SYMBOL , 找到由不同驱动提供的入口点. 大部分注册函数以 register_ 做前缀, 因此找到它们的另外一个方法是在内核源码里查找 register_ .

2.7.1. 清理函数

每个非试验性的模块也要求有一个清理函数, 它注销接口, 在模块被去除之前返回所有资源给系统. 这个函数定义为:

```
static void __exit cleanup_function(void)
{
 /* Cleanup code here */
}

module_exit(cleanup_function);
```

清理函数没有返回值, 因此它被声明为 void. __exit 修饰符标识这个代码是只用于模块卸载(通过使编译器把它放在特殊的 ELF 段). 如果你的模块直接建立在内核里, 或者如果你的内核配置成不允许模块卸载, 标识为 __exit 的函数被简单地丢弃. 因为这个原因, 一个标识 __exit 的函数只在模块卸载或者系统停止时调用; 任何别的使用是错的. 再一次, moudle_exit 声明对于使得内核能够找到你的清理函数是必要的.

如果你的模块没有定义一个清理函数, 内核不会允许它被卸载.

2.7.2. 初始化中的错误处理

你必须记住一件事, 在注册内核设施时, 注册可能失败. 即便最简单的动作常常需要内存分配, 分配的内存可能不可用. 因此模块代码必须一直检查返回值, 并且确认要求的操作实际上已经成功.

如果在你注册工具时发生任何错误, 首先第一的事情是决定模块是否能够无论如何继续初始化它自己. 常常, 在一个注册失败后模块可以继续操作, 如果需要可以功能降级. 在任何可能的时候, 你的模块应当尽力向前, 并提供事情失败后具备的能力.

如果证实你的模块在一个特别类型的失败后完全不能加载, 你必须取消任何在失败前注册的动作. 内核不保留已经注册的设施的每模块注册, 因此如果初始化在某个点失败, 模块必须能自己退回所有东西. 如果你无法注销你获取的东西, 内核就被置于一个不稳定状态; 它包含了不存在的代码的内部指针. 这种情况下, 经常地, 唯一的方法就是重启系统. 在初始化错误发生时, 你确实要小心地将事情做正确.

错误恢复有时用 goto 语句处理是最好的. 我们通常不愿使用 goto, 但是在我们的观念里, 这是一个它有用的地方. 在错误情形下小心使用 goto 可以去掉大量的复杂, 过度对齐的, "结构形" 的逻辑. 因此, 在内核里, goto 是处理错误经常用到, 如这里显示的.

下面例子代码(使用设施注册和注销函数)在初始化在任何点失败时做得正确:

```
int __init my_init_function(void)
{
      int err;
      /* registration takes a pointer and a name */
      err = register_this(ptr1, "skull");
      if (err)
            goto fail_this;
      err = register_that(ptr2, "skull");
      if (err)
            goto fail_that;
      err = register_those(ptr3, "skull");
      if (err)
            goto fail_those;
      return 0; /* success */
fail_those:
      unregister_that(ptr2, "skull");
fail_that:
      unregister_this(ptr1, "skull");
fail_this:
      return err; /* propagate the error */

}
```

这段代码试图注册 3 个(虚构的)设施. goto 语句在失败情况下使用, 在事情变坏之前只对之前已经成功注册的设施进行注销.

另一个选项, 不需要繁多的 goto 语句, 是跟踪已经成功注册的, 并且在任何出错情况下调用你的模块的清理函数. 清理函数只回卷那些已经成功完成的步骤. 然而这种选择, 需要更多代码和更多 CPU 时间, 因此在快速途径下, 你仍然依赖于 goto 作为最好的错误恢复工具.

my_init_function 的返回值, err, 是一个错误码. 在 Linux 内核里, 错误码是负数, 属于定义于 <linux/errno.h> 的集合. 如果你需要产生你自己的错误码代替你从其他函数得到的返回值, 你应当包含 <linux/errno.h> 以便使用符号式的返回值, 例如 -ENODEV, -ENOMEM, 等等. 返回适当的错误码总是一个好做法, 因为用户程序能够把它们转变为有意义的字串, 使用 perror 或者类似的方法.

显然, 模块清理函数必须撤销任何由初始化函数进行的注册, 并且惯例(但常常不是要求的)是按照注册时相反的顺序注销设施.

```
void __exit my_cleanup_function(void)
{
 unregister_those(ptr3, "skull");
 unregister_that(ptr2, "skull");
```

```
 unregister_this(ptr1, "skull");
 return;

}
```

如果你的初始化和清理比处理几项复杂, goto 方法可能变得难于管理, 因为所有的清理代码必须在初始化函数里重复, 包括几个混合的标号. 有时, 因此, 一种不同的代码排布证明更成功.

使代码重复最小和所有东西流线化, 你应当做的是无论何时发生错误都从初始化里调用清理函数. 清理函数接着必须在撤销它的注册前检查每一项的状态. 以最简单的形式, 代码看起来象这样:

```
struct something *item1;
struct somethingelse *item2;
int stuff_ok;

void my_cleanup(void)
{
	if (item1)
		release_thing(item1);
	if (item2)
		release_thing2(item2);
	if (stuff_ok)
		unregister_stuff();
	return;
}

int __init my_init(void)
{
	int err = -ENOMEM;

	item1 = allocate_thing(arguments);
	item2 = allocate_thing2(arguments2);
	if (!item2 || !item2)
		goto fail;

	err = register_stuff(item1, item2);
	if (!err)
		stuff_ok = 1;
	else
		goto fail;
	return 0; /* success */

fail:
```

```
    my_cleanup();
    return err;
}
```

如这段代码所示, 你也许需要, 也许不要外部的标志来标识初始化步骤的成功, 要依赖你调用的注册/分配函数的语义. 不管要不要标志, 这种初始化会变得包含大量的项, 常常比之前展示的技术要好. 注意, 但是, 清理函数当由非退出代码调用时不能标志为 __exit, 如同前面的例子.

2.7.3. 模块加载竞争

到目前, 我们的讨论已来到一个模块加载的重要方面: 竞争情况. 如果你在如何编写你的初始化函数上不小心, 你可能造成威胁到整个系统的稳定的情形. 我们将在本书稍后讨论竞争情况; 现在, 快速提几点就足够了:

首先时你应该一直记住, 内核的某些别的部分会在注册完成之后马上使用任何你注册的设施. 这是完全可能的, 换句话说, 内核将调用进你的模块, 在你的初始化函数仍然在运行时. 所以你的代码必须准备好被调用, 一旦它完成了它的第一个注册. 不要注册任何设施, 直到所有的需要支持那个设施的你的内部初始化已经完成.

你也必须考虑到如果你的初始化函数决定失败会发生什么, 但是内核的一部分已经在使用你的模块已注册的设施. 如果这种情况对你的模块是可能的, 你应当认真考虑根本不要使初始化失败. 毕竟, 模块已清楚地成功输出一些有用的东西. 如果初始化必须失败, 必须小心地处理任何可能的在内核别处发生的操作, 直到这些操作已完成.

2.8. 模块参数

驱动需要知道的几个参数因不同的系统而不同. 从使用的设备号(如我们在下一章见到的)到驱动应当任何操作的几个方面. 例如, SCSI 适配器的驱动常常有选项控制标记命令队列的使用, IDE 驱动允许用户控制 DMA 操作. 如果你的驱动控制老的硬件, 还需要被明确告知哪里去找硬件的 I/O 端口或者 I/O 内存地址. 内核通过在加载驱动的模块时指定可变参数的值, 支持这些要求.

这些参数的值可由 insmod 或者 modprobe 在加载时指定; 后者也可以从它的配置文件(/etc/modprobe.conf)读取参数的值. 这些命令在命令行里接受几类规格的值. 作为演示这种能力的一种方法, 想象一个特别需要的对本章开始的"hello world"模块(称为 hellop)的改进. 我们增加 2 个参数: 一个整型值, 称为 howmany, 一个字符串称为 whom. 我们的特别多功能的模块就在加载时, 欢迎 whom 不止一次, 而是 howmany 次. 这样一个模块可以用这样的命令行加载:

insmod hellop howmany=10 whom="Mom"

一旦以那样的方式加载, hellop 会说 "hello, Mom" 10 次.

但是, 在 insmod 可以修改模块参数前, 模块必须使它们可用. 参数用 moudle_param 宏定义来声明, 它定义在 moduleparam.h. module_param 使用了 3 个参数: 变量名, 它的类型, 以及一个权限掩码用来做一个辅助的 sysfs 入口. 这个宏定义应当放在任何函数之外, 典型地是出现在源文件的前面. 因此 hellop 将声明它的参数, 并如下使得对 insmod 可用:

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

模块参数支持许多类型:

bool
invbool

一个布尔型(true 或者 false)值(相关的变量应当是 int 类型). invbool 类型颠倒了值, 所以真值变成 false, 反之亦然.

charp

一个字符指针值. 内存为用户提供的字串分配, 指针因此设置.

int
long
short
uint
ulong
ushort

基本的变长整型值. 以 u 开头的是无符号值.

数组参数, 用逗号间隔的列表提供的值, 模块加载者也支持. 声明一个数组参数, 使用:

module_param_array(name,type,num,perm);

这里 name 是你的数组的名子(也是参数名), type 是数组元素的类型, num 是一个整型变量, perm 是通常的权限值. 如果数组参数在加载时设置, num 被设置成提供的数的个数. 模块加载者拒绝比数组能放下的多的值.

如果你确实需要一个没有出现在上面列表中的类型, 在模块代码里有钩子会允许你来定义它们; 任何使用它们的细节见 moduleparam.h. 所有的模块参数应当给定一个缺省值; insmod 只在用户明确告知它的时候才改变这些值. 模块可检查明显的参数, 通过对应它们的缺省值检查这些参数.

最后的 module_param 字段是一个权限值; 你应当使用 <linux/stat.h> 中定义的值. 这个值控制谁可以存取这些模块参数在 sysfs 中的表示. 如果 perm 被设为 0, 就根本没有 sysfs 项. 否则, 它出现在 /sys/module[5] 下面, 带有给定的权限. 使用 S_IRUGO 作为参数可以被所有人读取, 但是不能改变; S_IRUGO|S_IWUSR 允许 root 来改变参数. 注意, 如果一个参数被 sysfs 修改, 你的模块看到的参数值也改变了, 但是你的模块没有任何其他的通知. 你应当不要使模块参数可写, 除非你准备好检测这个改变并且因而作出反应.

---

[5] 然而, 在本书写作时, 有讨论将参数移出 sysfs.

2.9. 在用户空间做

一个第一次涉及内核问题的 Unix 程序员, 可能会紧张写一个模块. 编写一个用户程序来直接读写设备端口可能容易些.

确实, 有几个论据倾向于用户空间编程, 有时编写一个所谓的用户空间设备驱动对比钻研内核是一个明智的选择. 在本节, 我们讨论几个理由, 为什么你可能在用户空间编写驱动. 本书是关于内核空间驱动的, 但是, 所以我们不超越这个介绍性的讨论.

用户空间驱动的好处在于:

- 完整的 C 库可以连接. 驱动可以进行许多奇怪的任务, 不用依靠外面的程序(实现使用策略的工具程序, 常常随着驱动自身发布).
- 程序员可以在驱动代码上运行常用的调试器, 而不必走调试一个运行中的内核的弯路.
- 如果一个用户空间驱动挂起了, 你可简单地杀掉它. 驱动的问题不可能挂起整个系统, 除非被控制的硬件真的疯掉了.
- 用户内存是可交换的, 不象内核内存. 一个不常使用的却有很大一个驱动的设备不会占据别的程序可以用到的 RAM, 除了在它实际在用时.
- 一个精心设计的驱动程序仍然可以, 如同内核空间驱动, 允许对设备的并行存取.
- 如果你必须编写一个封闭源码的驱动, 用户空间的选项使你容易避免不明朗的许可的情况和改变的内核接口带来的问题.

例如, USB 驱动能够在用户空间编写; 看(仍然年幼) libusb 项目, 在 libusb.sourceforge.net 和 "gadgetfs" 在内核源码里. 另一个例子是 X 服务器: 它确切地知道它能处理哪些硬件, 哪些不能, 并且它提供图形资源给所有的 X 客户. 注意, 然而, 有一个缓慢但是固定的漂移向着基于 frame-buffer 的图形环境, X 服务器只是作为一个服务器, 基于一个内核空间的真实的设备驱动, 这个驱动负责真正的图形操作.

常常, 用户空间驱动的编写者完成一个服务器进程, 从内核接管作为单个代理的负责硬件控制的任务. 客户应用程序就可以连接到服务器来进行实际的操作; 因此, 一个聪明的驱动经常可以允许对设备的并行存取. 这就是 X 服务器如何工作的.

但是用户空间的设备驱动的方法有几个缺点. 最重要的是:

- 中断在用户空间无法用. 在某些平台上有对这个限制的解决方法, 例如在 IA32 体系上的 vm86 系统调用.
- 只可能通过内存映射 /dev/mem 来使用 DMA, 而且只有特权用户可以这样做.

- 存取 I/O 端口只能在调用 ioperm 或者 iopl 之后. 此外, 不是所有的平台支持这些系统调用, 而存取/dev/port可能太慢而无效率. 这些系统调用和设备文件都要求特权用户.
- 响应时间慢, 因为需要上下文切换在客户和硬件之间传递信息或动作.
- 更不好的是, 如果驱动已被交换到硬盘, 响应时间会长到不可接受. 使用 mlock 系统调用可能会有帮助, 但是常常的你将需要锁住许多内存页, 因为一个用户空间程序依赖大量的库代码. mlock, 也, 限制在授权用户上.
- 最重要的设备不能在用户空间处理, 包括但不限于, 网络接口和块设备.

如你所见, 用户空间驱动不能做的事情毕竟太多. 感兴趣的应用程序还是存在: 例如, 对 SCSI 扫描器设备的支持(由 SANE 包实现)和 CD 刻录器 (由 cdrecord 和别的工具实现). 在两种情况下, 用户级别的设备情况依赖 "SCSI gneric" 内核驱动, 它输出了低层的 SCSI 功能给用户程序, 因此它们可以驱动它们自己的硬件.

一种在用户空间工作的情况可能是有意义的, 当你开始处理新的没有用过的硬件时. 这样你可以学习去管理你的硬件, 不必担心挂起整个系统. 一旦你完成了, 在一个内核模块中封装软件就会是一个简单操作了.

上一页 2.8. 模块参数 | 上一级 起始页 | 下一页 2.10. 快速参考

2.10. 快速参考

本节总结了我们在本章接触到的内核函数, 变量, 宏定义, 和 /proc 文件. 它的用意是作为一个参考. 每一项列都在相关头文件的后面, 如果有. 从这里开始, 在几乎每章的结尾会有类似一节, 总结一章中介绍的新符号. 本节中的项通常以在本章中出现的顺序排列:

insmod
modprobe
rmmod

用户空间工具, 加载模块到运行中的内核以及去除它们.

```
#include <linux/init.h>
module_init(init_function);
module_exit(cleanup_function);
```

指定模块的初始化和清理函数的宏定义.

__init
__initdata
__exit
__exitdata

函数(__init 和 __exit)和数据 (__initdata 和 __exitdata)的标记, 只用在模块初始化或者清理时间. 为初始化所标识的项可能会在初始化完成后丢弃; 退出的项可能被丢弃如果内核没有配置模块卸载. 这些标记通过使相关的目标在可执行文件的特定的 ELF 节里被替换来工作.

```
#include <linux/sched.h>
```

最重要的头文件中的一个. 这个文件包含很多驱动使用的内核 API 的定义, 包括睡眠函数和许多变量声明.

```
struct task_struct *current;
```

当前进程.

current->pid
current->comm

进程 ID 和 当前进程的命令名.

obj-m

一个 makefile 符号, 内核建立系统用来决定当前目录下的哪个模块应当被建立.

/sys/module
/proc/modules

/sys/module 是一个 sysfs 目录层次, 包含当前加载模块的信息. /proc/moudles 是旧式的, 那种信息的单个文件版本. 其中的条目包含了模块名, 每个模块占用的内存数量, 以及使用计数. 另外的字串追加到每行的末尾来指定标志, 对这个模块当前是活动的.

vermagic.o

来自内核源码目录的目标文件, 描述一个模块为之建立的环境.

```
#include <linux/module.h>
```

必需的头文件. 它必须在一个模块源码中包含.

```
#include <linux/version.h>
```

头文件, 包含在建立的内核版本信息.

LINUX_VERSION_CODE

整型宏定义, 对 #ifdef 版本依赖有用.

```
EXPORT_SYMBOL (symbol);
EXPORT_SYMBOL_GPL (symbol);
```

宏定义, 用来输出一个符号给内核. 第 2 种形式输出没有版本信息, 第 3 种限制输出给 GPL 许可的模块.

```
MODULE_AUTHOR(author);
MODULE_DESCRIPTION(description);
MODULE_VERSION(version_string);
MODULE_DEVICE_TABLE(table_info);
MODULE_ALIAS(alternate_name);
```

放置文档在目标文件的模块中.

```
module_init(init_function);
module_exit(exit_function);
```

宏定义, 声明一个模块的初始化和清理函数.

```
#include <linux/moduleparam.h>
module_param(variable, type, perm);
```

宏定义, 创建模块参数, 可以被用户在模块加载时调整(或者在启动时间, 对于内嵌代码). 类型可以是 bool, charp, int, invbool, short, ushort, uint, ulong, 或者 intarray.

```
#include <linux/kernel.h>
int printk(const char * fmt, ...);
```

内核代码的 printf 类似物.

上一页 2.9. 在用户空间做
上一级 起始页
下一页 第 3 章 字符驱动

第 3 章 字符驱动

第 3 章 字符驱动

目录

本章的目的是编写一个完整的字符设备驱动. 我们开发一个字符驱动是因为这一类适合大部分简单硬件设备. 字符驱动也比块驱动易于理解(我们在后续章节接触). 我们的最终目的是编写一个模块化的字符驱动, 但是我们不会在本章讨论模块化的事情.

贯串本章, 我们展示从一个真实设备驱动提取的代码片段: scull(Simple Character Utility for Loading Localities). scull 是一个字符驱动, 操作一块内存区域好像它是一个设备. 在本章, 因为 scull 的这个怪特性, 我们可互换地使用设备这个词和"scull使用的内存区".

scull 的优势在于它不依赖硬件. scull 只是操作一些从内核分配的内存. 任何人都可以编译和运行 scull, 并且 scull 在 Linux 运行的体系结构中可移植. 另一方面, 这个设备除了演示内核和字符驱动的接口和允许用户运行一些测试之外, 不做任何有用的事情.

3.1. scull 的设计

编写驱动的第一步是定义驱动将要提供给用户程序的能力(机制).因为我们的"设备"是计算机内存的一部分, 我们可自由做我们想做的事情. 它可以是一个顺序的或者随机存取的设备, 一个或多个设备, 等等.

为使 scull 作为一个模板来编写真实设备的真实驱动, 我们将展示给你如何在计算机内存上实现几个设备抽象, 每个有不同的个性.

scull 源码实现下面的设备. 模块实现的每种设备都被引用做一种类型.

scull0 到 scull3

> 4 个设备, 每个由一个全局永久的内存区组成. 全局意味着如果设备被多次打开, 设备中含有的数据由所有打开它的文件描述符共享. 永久意味着如果设备关闭又重新打开, 数据不会丢失. 这个设备用起来有意思, 因为它可以用惯常的命令来存取和测试, 例如 cp, cat, 以及 I/O 重定向.

scullpipe0 到 scullpipe3

> 4 个 FIFO (先入先出) 设备, 行为象管道. 一个进程读的内容来自另一个进程所写的. 如果多个进程读同一个设备, 它们竞争数据. scullpipe 的内部将展示阻塞读写和非阻塞读写如何实现, 而不必采取中断. 尽管真实的驱动使用硬件中断来同步它们的设备, 阻塞和非阻塞操作的主题是重要的并且与中断处理是分开的.(在第 10 章涉及).

scullsingle
scullpriv
sculluid
scullwuid

> 这些设备与 scull0 相似, 但是在什么时候允许打开上有一些限制. 第一个(snullsingle) 只允许一次一个进程使用驱动, 而 scullpriv 对每个虚拟终端(或者 X 终端会话)是私有的, 因为每个控制台/终端上的进程有不同的内存区. sculluid 和 scullwuid 可以多次打开, 但是一次只能是一个用户; 前者返回一个"设备忙"错误, 如果另一个用户锁着设备, 而后者实现阻塞打开. 这些 scull 的变体可能看来混淆了策略和机制, 但是它们值得看看, 因为一些实际设备需要这类管理.

每个 scull 设备演示了驱动的不同特色, 并且呈现了不同的难度. 本章涉及 scull0 到 scull3 的内部; 更

高级的设备在第 6 章涉及. scullpipe 在"一个阻塞 I/O 例子"一节中描述, 其他的在"设备文件上的存取控制"中描述.

上一页 下一页
2.10. 快速参考 起始页 3.2. 主次编号

3.2. 主次编号

字符设备通过文件系统中的名子来存取. 那些名子称为文件系统的特殊文件, 或者设备文件, 或者文件系统的简单结点; 惯例上它们位于 /dev 目录. 字符驱动的特殊文件由使用 ls -l 的输出的第一列的"c"标识. 块设备也出现在 /dev 中, 但是它们由"b"标识. 本章集中在字符设备, 但是下面的很多信息也适用于块设备.

如果你发出 ls -l 命令, 你会看到在设备文件项中有 2 个数(由一个逗号分隔)在最后修改日期前面, 这里通常是文件长度出现的地方. 这些数字是给特殊设备的主次设备编号. 下面的列表显示了一个典型系统上出现的几个设备. 它们的主编号是 1, 4, 7, 和 10, 而次编号是 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, 主编号标识设备相连的驱动. 例如, /dev/null 和 /dev/zero 都由驱动 1 来管理, 而虚拟控制台和串口终端都由驱动 4 管理; 同样, vcs1 和 vcsa1 设备都由驱动 7 管理. 现代 Linux 内核允许多个驱动共享主编号, 但是你看到的大部分设备仍然按照一个主编号一个驱动的原则来组织.

次编号被内核用来决定引用哪个设备. 依据你的驱动是如何编写的(如同我们下面见到的), 你可以从内核得到一个你的设备的直接指针, 或者可以自己使用次编号作为本地设备数组的索引. 不论哪个方法, 内核自己几乎不知道次编号的任何事情, 除了它们指向你的驱动实现的设备.

3.2.1. 设备编号的内部表示

在内核中, dev_t 类型(在 <linux/types.h>中定义)用来持有设备编号 -- 主次部分都包括. 对于 2.6.0 内核, dev_t 是 32 位的量, 12 位用作主编号, 20 位用作次编号. 你的代码应当, 当然, 对于设备编号的内部组织从不做任何假设; 相反, 应当利用在 <linux/kdev_t.h>中的一套宏定义. 为获得一个 dev_t 的主或者次编号, 使用:

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

相反, 如果你有主次编号, 需要将其转换为一个 dev_t, 使用:

MKDEV(int major, int minor);

注意, 2.6 内核能容纳有大量设备, 而以前的内核版本限制在 255 个主编号和 255 个次编号. 有人认为这么宽的范围在很长时间内是足够的, 但是计算领域被这个特性的错误假设搞乱了. 因此你应当希望 dev_t 的格式将来可能再次改变; 但是, 如果你仔细编写你的驱动, 这些变化不会是一个问题.

3.2.2. 分配和释放设备编号

在建立一个字符驱动时你的驱动需要做的第一件事是获取一个或多个设备编号来使用. 为此目的的必要的函数是 register_chrdev_region, 在 <linux/fs.h>中声明:

int register_chrdev_region(dev_t first, unsigned int count, char *name);

这里, first 是你要分配的起始设备编号. first 的次编号部分常常是 0, 但是没有要求是那个效果. count 是你请求的连续设备编号的总数. 注意, 如果 count 太大, 你要求的范围可能溢出到下一个次编号; 但是只要你要求的编号范围可用, 一切都仍然会正确工作. 最后, name 是应当连接到这个编号范围的设备的名子; 它会出现在 /proc/devices 和 sysfs 中.

如同大部分内核函数, 如果分配成功进行, register_chrdev_region 的返回值是 0. 出错的情况下, 返回一个负的错误码, 你不能存取请求的区域.

如果你确实事先知道你需要哪个设备编号, register_chrdev_region 工作得好. 然而, 你常常不会知道你的设备使用哪个主编号; 在 Linux 内核开发社团中一直努力使用动态分配设备编号. 内核会乐于动态为你分配一个主编号, 但是你必须使用一个不同的函数来请求这个分配.

int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);

使用这个函数, dev 是一个只输出的参数, 它在函数成功完成时持有你的分配范围的第一个数. fisetminor 应当是请求的第一个要用的次编号; 它常常是 0. count 和 name 参数如同给 request_chrdev_region 的一样.

不管你任何分配你的设备编号, 你应当在不再使用它们时释放它. 设备编号的释放使用:

void unregister_chrdev_region(dev_t first, unsigned int count);

调用 unregister_chrdev_region 的地方常常是你的模块的 cleanup 函数.

上面的函数分配设备编号给你的驱动使用, 但是它们不告诉内核你实际上会对这些编号做什么. 在

用户空间程序能够存取这些设备号中一个之前, 你的驱动需要连接它们到它的实现设备操作的内部函数上. 我们将描述如何简短完成这个连接, 但首先顾及一些必要的枝节问题.

3.2.3. 主编号的动态分配

一些主设备编号是静态分派给最普通的设备的. 一个这些设备的列表在内核源码树的 Documentation/devices.txt 中. 分配给你的新驱动使用一个已经分配的静态编号的机会很小, 但是, 并且新编号没在分配. 因此, 作为一个驱动编写者, 你有一个选择: 你可以简单地捡一个看来没有用的编号, 或者你以动态方式分配主编号. 只要你是你的驱动的唯一用户就可以捡一个编号用; 一旦你的驱动更广泛的被使用了, 一个随机捡来的主编号将导致冲突和麻烦.

因此, 对于新驱动, 我们强烈建议你使用动态分配来获取你的主设备编号, 而不是随机选取一个当前空闲的编号. 换句话说, 你的驱动应当几乎肯定地使用 alloc_chrdev_region, 不是 register_chrdev_region.

动态分配的缺点是你无法提前创建设备节点, 因为分配给你的模块的主编号会变化. 对于驱动的正常使用, 这不是问题, 因为一旦编号分配了, 你可从 /proc/devices 中读取它.[6]

为使用动态主编号来加载一个驱动, 因此, 可使用一个简单的脚本来代替调用 insmod, 在调用 insmod 后, 读取 /proc/devices 来创建特殊文件.

一个典型的 /proc/devices 文件看来如下:

```
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
```

因此加载一个已经安排了一个动态编号的模块的脚本, 可以使用一个工具来编写, 如 awk , 来从 /proc/devices 获取信息以创建 /dev 中的文件.

下面的脚本, snull_load, 是 scull 发布的一部分. 以模块发布的驱动的用户可以从系统的 rc.local 文件中调用这样一个脚本, 或者在需要模块时手工调用它.

```
#!/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]
```

这个脚本可以通过重定义变量和调整 mknod 行来适用于另外的驱动. 这个脚本仅仅展示了创建 4 个设备, 因为 4 是 scull 源码中缺省的.

脚本的最后几行可能有些模糊:为什么改变设备的组和模式? 理由是这个脚本必须由超级用户运行, 因此新建的特殊文件由 root 拥有. 许可位缺省的是只有 root 有写权限, 而任何人可以读. 通常, 一个设备节点需要一个不同的存取策略, 因此在某些方面别人的存取权限必须改变. 我们的脚本缺省是给一个用户组存取, 但是你的需求可能不同. 在第 6 章的"设备文件的存取控制"一节中, sculluid 的代码演示了驱动如何能够强制它自己的对设备存取的授权.

还有一个 scull_unload 脚本来清理 /dev 目录并去除模块.

作为对使用一对脚本来加载和卸载的另外选择, 你可以编写一个 init 脚本, 准备好放在你的发布使用这些脚本的目录中. [7]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 init 脚本例子, 称为 scull.init; 它接受传统的参数 -- start, stop, 和 restart -- 并且完成 scull_load 和 scull_unload 的角色.

如果反复创建和销毁 /dev 节点, 听来过分了, 有一个有用的办法. 如果你在加载和卸载单个驱动, 你可以在你第一次使用你的脚本创建特殊文件之后, 只使用 rmmod 和 insmod: 这样动态编号不是随机的. [8]并且你每次都可以使用所选的同一个编号, 如果你不加载任何别的动态模块. 在开发中避免长脚本是有用的. 但是这个技巧, 显然不能扩展到一次多于一个驱动.

安排主编号最好的方式, 我们认为, 是缺省使用动态分配, 而留给自己在加载时指定主编号的选项权, 或者甚至在编译时. scull 实现以这种方式工作; 它使用一个全局变量, scull_major, 来持有选定的编号(还有一个 scull_minor 给次编号). 这个变量初始化为 SCULL_MAJOR, 定义在 scull.h. 发布的源码中的 SCULL_MAJOR 的缺省值是 0, 意思是"使用动态分配". 用户可以接受缺省值或者选择一个特殊主编号, 或者在编译前修改宏定义或者在 insmod 命令行指定一个值给 scull_major. 最后, 通过使用 scull_load 脚本, 用户可以在 scull_load 的命令行传递参数给 insmod.[9]

这是我们用在 scull 的源码中获取主编号的代码:

```
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;
}
```

本书使用的几乎所有例子驱动使用类似的代码来分配它们的主编号.

---

[6] 从 sysfs 中能获取更好的设备信息, 在基于 2.6 的系统通常加载于 /sys. 但是使 scull 通过 sysfs 输出信息超出了本章的范围; 我们在 14 章中回到这个主题.

[7] Linux Standard Base 指出 init 脚本应当放在 /etc/init.d, 但是一些发布仍然放在别处. 另外, 如果你的脚本在启动时运行, 你需要从合适的运行级别目录做一个连接给它(也就是, .../rc3.d).

[8] 尽管某些内核开发者已警告说将来就会这样做.

[9] init 脚本 scull.init 不在命令行中接受驱动选项, 但是它支持一个配置文件, 因为它被设计来在启动和关机时自动使用.

3.3. 一些重要数据结构

如同你想象的, 注册设备编号仅仅是驱动代码必须进行的诸多任务中的第一个. 我们将很快看到其他重要的驱动组件, 但首先需要涉及一个别的. 大部分的基础性的驱动操作包括 3 个重要的内核数据结构, 称为 file_operations, file, 和 inode. 需要对这些结构的基本了解才能够做大量感兴趣的事情, 因此我们现在在进入如何实现基础性驱动操作的细节之前, 会快速查看每一个.

3.3.1. 文件操作

到现在, 我们已经保留了一些设备编号给我们使用, 但是我们还没有连接任何我们设备操作到这些编号上. file_operation 结构是一个字符驱动如何建立这个连接. 这个结构, 定义在 <linux/fs.h>, 是一个函数指针的集合. 每个打开文件(内部用一个 file 结构来代表, 稍后我们会查看)与它自身的函数集合相关连(通过包含一个称为 f_op 的成员, 它指向一个 file_operations 结构). 这些操作大部分负责实现系统调用, 因此, 命名为 open, read, 等等. 我们可以认为文件是一个"对象"并且其上的函数操作称为它的"方法", 使用面向对象编程的术语来表示一个对象声明的用来操作对象的动作. 这是我们在 Linux 内核中看到的第一个面向对象编程的现象, 后续章中我们会看到更多.

传统上, 一个 file_operation 结构或者其一个指针称为 fops(或者它的一些变体). 结构中的每个成员必须指向驱动中的函数, 这些函数实现一个特别的操作, 或者对于不支持的操作留置为 NULL. 当指定为 NULL 指针时内核的确切的行为是每个函数不同的, 如同本节后面的列表所示.

下面的列表介绍了一个应用程序能够在设备上调用的所有操作. 我们已经试图保持列表简短, 这样它可作为一个参考, 只是总结每个操作和在 NULL 指针使用时的缺省内核行为.

在你通读 file_operations 方法的列表时, 你会注意到不少参数包含字串 __user. 这种注解是一种文档形式, 注意, 一个指针是一个不能被直接解引用的用户空间地址. 对于正常的编译, __user 没有效果, 但是它可被外部检查软件使用来找出对用户空间地址的错误使用.

本章剩下的部分, 在描述一些其他重要数据结构后, 解释了最重要操作的角色并且给了提示, 告诫和真实代码例子. 我们推迟讨论更复杂的操作到后面章节, 因为我们还不准备深入如内存管理, 阻塞操作, 和异步通知.

struct module *owner

> 第一个 file_operations 成员根本不是一个操作; 它是一个指向拥有这个结构的模块的指针. 这个成员用来在它的操作还在被使用时阻止模块被卸载. 几乎所有时间中, 它被简单初始化为 THIS_MODULE, 一个在 <linux/module.h> 中定义的宏.

loff_t (*llseek) (struct file *, loff_t, int);

llseek 方法用作改变文件中的当前读/写位置, 并且新位置作为(正的)返回值. loff_t 参数是一个"long offset", 并且就算在 32位平台上也至少 64 位宽. 错误由一个负返回值指示. 如果这个函数指针是 NULL, seek 调用会以潜在地无法预知的方式修改 file 结构中的位置计数器 (在"file 结构" 一节中描述).

ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);

用来从设备中获取数据. 在这个位置的一个空指针导致 read 系统调用以 -EINVAL("Invalid argument") 失败. 一个非负返回值代表了成功读取的字节数(返回值是一个 "signed size" 类型, 常常是目标平台本地的整数类型).

ssize_t (*aio_read)(struct kiocb *, char __user *, size_t, loff_t);

初始化一个异步读 -- 可能在函数返回前不结束的读操作. 如果这个方法是 NULL, 所有的操作会由 read 代替进行(同步地).

ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);

发送数据给设备. 如果 NULL, -EINVAL 返回给调用 write 系统调用的程序. 如果非负, 返回值代表成功写的字节数.

ssize_t (*aio_write)(struct kiocb *, const char __user *, size_t, loff_t *);

初始化设备上的一个异步写.

int (*readdir) (struct file *, void *, filldir_t);

对于设备文件这个成员应当为 NULL; 它用来读取目录, 并且仅对文件系统有用.

unsigned int (*poll) (struct file *, struct poll_table_struct *);

poll 方法是 3 个系统调用的后端: poll, epoll, 和 select, 都用作查询对一个或多个文件描述符的读或写是否会阻塞. poll 方法应当返回一个位掩码指示是否非阻塞的读或写是可能的, 并且, 可能地, 提供给内核信息用来使调用进程睡眠直到 I/O 变为可能. 如果一个驱动的 poll 方法为 NULL, 设备假定为不阻塞地可读可写.

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

ioctl 系统调用提供了发出设备特定命令的方法(例如格式化软盘的一个磁道, 这不是读也不是写). 另外, 几个 ioctl 命令被内核识别而不必引用 fops 表. 如果设备不提供 ioctl 方法, 对于任何未事先定义的请求(-ENOTTY, "设备无这样的 ioctl"), 系统调用返回一个错误.

int (*mmap) (struct file *, struct vm_area_struct *);

mmap 用来请求将设备内存映射到进程的地址空间. 如果这个方法是 NULL, mmap 系统调用返回 -ENODEV.

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

尽管这常常是对设备文件进行的第一个操作, 不要求驱动声明一个对应的方法. 如果这个项是 NULL, 设备打开一直成功, 但是你的驱动不会得到通知.

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

flush 操作在进程关闭它的设备文件描述符的拷贝时调用; 它应当执行(并且等待)设备的任何未完成的操作. 这个必须不要和用户查询请求的 fsync 操作混淆了. 当前, flush 在很少驱动中使用; SCSI 磁带驱动使用它, 例如, 为确保所有写的数据在设备关闭前写到磁带上. 如果 flush 为 NULL, 内核简单地忽略用户应用程序的请求.

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

在文件结构被释放时引用这个操作. 如同 open, release 可以为 NULL.

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

这个方法是 fsync 系统调用的后端, 用户调用来刷新任何挂着的数据. 如果这个指针是 NULL, 系统调用返回 -EINVAL.

```
int (*aio_fsync)(struct kiocb *, int);
```

这是 fsync 方法的异步版本.

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

这个操作用来通知设备它的 FASYNC 标志的改变. 异步通知是一个高级的主题, 在第 6 章中描述. 这个成员可以是NULL 如果驱动不支持异步通知.

```
int (*lock) (struct file *, int, struct file_lock *);
```

lock 方法用来实现文件加锁; 加锁对常规文件是必不可少的特性, 但是设备驱动几乎从不实现它.

```
ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
```

这些方法实现发散/汇聚读和写操作. 应用程序偶尔需要做一个包含多个内存区的单个读或写操作; 这些系统调用允许它们这样做而不必对数据进行额外拷贝. 如果这些函数指针为 NULL, read 和 write 方法被调用(可能多于一次).

```
ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t, void *);
```

这个方法实现 sendfile 系统调用的读, 使用最少的拷贝从一个文件描述符搬移数据到另一个. 例如, 它被一个需要发送文件内容到一个网络连接的 web 服务器使用. 设备驱动常常使 sendfile 为 NULL.

```
ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
```

sendpage 是 sendfile 的另一半; 它由内核调用来发送数据, 一次一页, 到对应的文件. 设备驱动实际上不实现 sendpage.

```
unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned
```

long);

这个方法的目的是在进程的地址空间找一个合适的位置来映射在底层设备上的内存段中. 这个任务通常由内存管理代码进行; 这个方法存在为了使驱动能强制特殊设备可能有的任何的对齐请求. 大部分驱动可以置这个方法为 NULL.[10]

int (*check_flags)(int)

这个方法允许模块检查传递给 fnctl(F_SETFL...) 调用的标志.

int (*dir_notify)(struct file *, unsigned long);

这个方法在应用程序使用 fcntl 来请求目录改变通知时调用. 只对文件系统有用; 驱动不需要实现 dir_notify.

scull 设备驱动只实现最重要的设备方法. 它的 file_operations 结构是如下初始化的:

```
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,
};
```

这个声明使用标准的 C 标记式结构初始化语法. 这个语法是首选的, 因为它使驱动在结构定义的改变之间更加可移植, 并且, 有争议地, 使代码更加紧凑和可读. 标记式初始化允许结构成员重新排序; 在某种情况下, 真实的性能提高已经实现, 通过安放经常使用的成员的指针在相同硬件高速存储行中.

3.3.2. 文件结构

struct file, 定义于 <linux/fs.h>, 是设备驱动中第二个最重要的数据结构. 注意 file 与用户空间程序的 FILE 指针没有任何关系. 一个 FILE 定义在 C 库中, 从不出现在内核代码中. 一个 struct file, 另一方面, 是一个内核结构, 从不出现在用户程序中.

文件结构代表一个打开的文件. (它不特定给设备驱动; 系统中每个打开的文件有一个关联的 struct file 在内核空间). 它由内核在 open 时创建, 并传递给在文件上操作的任何函数, 直到最后的关闭. 在文件的所有实例都关闭后, 内核释放这个数据结构.

在内核源码中, struct file 的指针常常称为 file 或者 filp("file pointer"). 我们将一直称这个指针为 filp 以

避免和结构自身混淆. 因此, file 指的是结构, 而 filp 是结构指针.

struct file 的最重要成员在这展示. 如同在前一节, 第一次阅读可以跳过这个列表. 但是, 在本章后面, 当我们面对一些真实 C 代码时, 我们将更详细讨论这些成员.

mode_t f_mode;

文件模式确定文件是可读的或者是可写的(或者都是), 通过位 FMODE_READ 和 FMODE_WRITE. 你可能想在你的 open 或者 ioctl 函数中检查这个成员的读写许可, 但是你不需要检查读写许可, 因为内核在调用你的方法之前检查. 当文件还没有为那种存取而打开时读或写的企图被拒绝, 驱动甚至不知道这个情况.

loff_t f_pos;

当前读写位置. loff_t 在所有平台都是 64 位(在 gcc 术语里是 long long). 驱动可以读这个值, 如果它需要知道文件中的当前位置, 但是正常地不应该改变它; 读和写应当使用它们作为最后参数而收到的指针来更新一个位置, 代替直接作用于 filp->f_pos. 这个规则的一个例外是在 llseek 方法中, 它的目的就是改变文件位置.

unsigned int f_flags;

这些是文件标志, 例如 O_RDONLY, O_NONBLOCK, 和 O_SYNC. 驱动应当检查 O_NONBLOCK 标志来看是否是请求非阻塞操作(我们在第一章的"阻塞和非阻塞操作"一节中讨论非阻塞 I/O); 其他标志很少使用. 特别地, 应当检查读/写许可, 使用 f_mode 而不是 f_flags. 所有的标志在头文件 <linux/fcntl.h> 中定义.

struct file_operations *f_op;

和文件关联的操作. 内核安排指针作为它的 open 实现的一部分, 接着读取它当它需要分派任何的操作时. filp->f_op 中的值从不由内核保存为后面的引用; 这意味着你可改变你的文件关联的文件操作, 在你返回调用者之后新方法会起作用. 例如, 关联到主编号 1 (/dev/null, /dev/zero, 等等)的 open 代码根据打开的次编号来替代 filp->f_op 中的操作. 这个做法允许实现几种行为, 在同一个主编号下而不必在每个系统调用中引入开销. 替换文件操作的能力是面向对象编程的"方法重载"的内核对等体.

void *private_data;

open 系统调用设置这个指针为 NULL, 在为驱动调用 open 方法之前. 你可自由使用这个成员或者忽略它; 你可以使用这个成员来指向分配的数据, 但是接着你必须记住在内核销毁文件结构之前, 在 release 方法中释放那个内存. private_data 是一个有用的资源, 在系统调用间保留状态信息, 我们大部分例子模块都使用它.

struct dentry *f_dentry;

关联到文件的目录入口(dentry)结构. 设备驱动编写者正常地不需要关心 dentry 结构, 除了作为 filp->f_dentry->d_inode 存取 inode 结构.

真实结构有多几个成员, 但是它们对设备驱动没有用处. 我们可以安全地忽略这些成员, 因为驱动从不创建文件结构; 它们真实存取别处创建的结构.

3.3.3. inode 结构

inode 结构由内核在内部用来表示文件. 因此, 它和代表打开文件描述符的文件结构是不同的. 可能有代表单个文件的多个打开描述符的许多文件结构, 但是它们都指向一个单个 inode 结构.

inode 结构包含大量关于文件的信息. 作为一个通用的规则, 这个结构只有 2 个成员对于编写驱动代码有用:

dev_t i_rdev;

对于代表设备文件的节点, 这个成员包含实际的设备编号.

struct cdev *i_cdev;

struct cdev 是内核的内部结构, 代表字符设备; 这个成员包含一个指针, 指向这个结构, 当节点指的是一个字符设备文件时.

i_rdev 类型在 2.5 开发系列中改变了, 破坏了大量的驱动. 作为一个鼓励更可移植编程的方法, 内核开发者已经增加了 2 个宏, 可用来从一个 inode 中获取主次编号:

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

为了不要被下一次改动抓住, 应当使用这些宏代替直接操作 i_rdev.

---

[10] 注意, release 不是每次进程调用 close 时都被调用. 无论何时共享一个文件结构(例如, 在一个 fork 或 dup 之后), release 不会调用直到所有的拷贝都关闭了. 如果你需要在任一拷贝关闭时刷新挂着的数据, 你应当实现 flush 方法.

3.4. 字符设备注册

如我们提过的, 内核在内部使用类型 struct cdev 的结构来代表字符设备. 在内核调用你的设备操作前, 你编写分配并注册一个或几个这些结构. [11]为此, 你的代码应当包含 <linux/cdev.h>, 这个结构和它的关联帮助函数定义在这里.

有 2 种方法来分配和初始化一个这些结构. 如果你想在运行时获得一个独立的 cdev 结构, 你可以为此使用这样的代码:

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
```

但是, 偶尔你会想将 cdev 结构嵌入一个你自己的设备特定的结构; scull 这样做了. 在这种情况下, 你应当初始化你已经分配的结构, 使用:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

任一方法, 有一个其他的 struct cdev 成员你需要初始化. 象 file_operations 结构, struct cdev 有一个拥有者成员, 应当设置为 THIS_MODULE. 一旦 cdev 结构建立, 最后的步骤是把它告诉内核, 调用:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

这里, dev 是 cdev 结构, num 是这个设备响应的第一个设备号, count 是应当关联到设备的设备号的数目. 常常 count 是 1, 但是有多个设备号对应于一个特定的设备的情形. 例如, 设想 SCSI 磁带驱动, 它允许用户空间来选择操作模式(例如密度), 通过安排多个次编号给每一个物理设备.

在使用 cdev_add 是有几个重要事情要记住. 第一个是这个调用可能失败. 如果它返回一个负的错误码, 你的设备没有增加到系统中. 它几乎会一直成功, 但是, 并且带起了其他的点: cdev_add 一返回, 你的设备就是"活的"并且内核可以调用它的操作. 除非你的驱动完全准备好处理设备上的操作, 你不应当调用 cdev_add.

为从系统去除一个字符设备, 调用:

```
void cdev_del(struct cdev *dev);
```

显然, 你不应当在传递给 cdev_del 后存取 cdev 结构.

3.4.1. scull 中的设备注册

在内部, scull 使用一个 struct scull_dev 类型的结构表示每个设备. 这个结构定义为:

```
struct scull_dev {
 struct scull_qset *data;  /* Pointer to first quantum set */
 int quantum;  /* the current quantum size */
 int qset;  /* the current array size */
 unsigned long size;  /* amount of data stored here */
 unsigned int access_key;  /* used by sculluid and scullpriv */
 struct semaphore sem;  /* mutual exclusion semaphore  */

 struct cdev cdev; /* Char device structure */
};
```

我们在遇到它们时讨论结构中的各个成员, 但是现在, 我们关注于 cdev, 我们的设备与内核接口的 struct cdev. 这个结构必须初始化并且如上所述添加到系统中; 处理这个任务的 scull 代码是:

```
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);
}
```

因为 cdev 结构嵌在 struct scull_dev 里面, cdev_init 必须调用来进行那个结构的初始化.

3.4.2. 老方法

如果你深入浏览 2.6 内核的大量驱动代码, 你可能注意到有许多字符驱动不使用我们刚刚描述过的 cdev 接口. 你见到的是还没有更新到 2.6 内核接口的老代码. 因为那个代码实际上能用, 这个更新可能很长时间不会发生. 为完整, 我们描述老的字符设备注册接口, 但是新代码不应当使用它; 这个机制在将来内核中可能会消失.

注册一个字符设备的经典方法是使用:

int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);

这里, major 是感兴趣的主编号, name 是驱动的名子(出现在 /proc/devices), fops 是缺省的 file_operations 结构. 一个对 register_chrdev 的调用为给定的主编号注册 0 - 255 的次编号, 并且为每一个建立一个缺省的 cdev 结构. 使用这个接口的驱动必须准备好处理对所有 256 个次编号的 open 调用(不管它们是否对应真实设备), 它们不能使用大于 255 的主或次编号.

如果你使用 register_chrdev, 从系统中去除你的设备的正确的函数是:

int unregister_chrdev(unsigned int major, const char *name);

major 和 name 必须和传递给 register_chrdev 的相同, 否则调用会失败.

---

[11] 有一个早些的机制以避免使用 cdev 结构(我们在"老方法"一节中讨论).但是, 新代码应当使用新技术.

上一页 3.3. 一些重要数据结构 | 上一级 起始页 | 下一页 3.5. open 和 release

3.5. open 和 release

到此我们已经快速浏览了这些成员, 我们开始在真实的 scull 函数中使用它们.

3.5.1. open 方法

open 方法提供给驱动来做任何的初始化来准备后续的操作. 在大部分驱动中, open 应当进行下面的工作:

- 检查设备特定的错误(例如设备没准备好, 或者类似的硬件错误
- 如果它第一次打开, 初始化设备
- 如果需要, 更新 f_op 指针.
- 分配并填充要放进 filp->private_data 的任何数据结构

但是, 事情的第一步常常是确定打开哪个设备. 记住 open 方法的原型是:

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

inode 参数有我们需要的信息,以它的 i_cdev 成员的形式, 里面包含我们之前建立的 cdev 结构. 唯一的问题是通常我们不想要 cdev 结构本身, 我们需要的是包含 cdev 结构的 scull_dev 结构. C 语言使程序员玩弄各种技巧来做这种转换; 但是, 这种技巧编程是易出错的, 并且导致别人难于阅读和理解代码. 幸运的是, 在这种情况下, 内核 hacker 已经为我们实现了这个技巧, 以 container_of 宏的形式, 在 <linux/kernel.h> 中定义:

container_of(pointer, container_type, container_field);

这个宏使用一个指向 container_field 类型的成员的指针, 它在一个 container_type 类型的结构中, 并且返回一个指针指向包含结构. 在 scull_open, 这个宏用来找到适当的设备结构:

```
struct scull_dev *dev; /* device information */
dev = container_of(inode->i_cdev, struct scull_dev, cdev);
filp->private_data = dev; /* for other methods */
```

一旦它找到 scull_dev 结构, scull 在文件结构的 private_data 成员中存储一个它的指针, 为以后更易存取.

识别打开的设备的另外的方法是查看存储在 inode 结构的次编号. 如果你使用 register_chrdev 注册你的设备, 你必须使用这个技术. 确认使用 iminor 从 inode 结构中获取次编号, 并且确定它对应一个你的驱动真正准备好处理的设备.

scull_open 的代码(稍微简化过)是:

```
int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* device information */
    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
    filp->private_data = dev; /* for other methods */

    /* 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 */
}
```

代码看来相当稀疏, 因为在调用 open 时它没有做任何特别的设备处理. 它不需要, 因为 scull 设备设计为全局的和永久的. 特别地, 没有如"在第一次打开时初始化设备"等动作, 因为我们不为 scull 保持打开计数.

唯一在设备上的真实操作是当设备为写而打开时将它截取为长度为 0. 这样做是因为, 在设计上, 用一个短的文件覆盖一个 scull 设备导致一个短的设备数据区. 这类似于为写而打开一个常规文件, 将其截短为 0. 如果设备为读而打开, 这个操作什么都不做.

在我们查看其他 scull 特性的代码时将看到一个真实的初始化如何起作用的.

3.5.2. release 方法

release 方法的角色是 open 的反面. 有时你会发现方法的实现称为 device_close, 而不是 device_release. 任一方式, 设备方法应当进行下面的任务:

- 释放 open 分配在 filp->private_data 中的任何东西
- 在最后的 close 关闭设备

scull 的基本形式没有硬件去关闭, 因此需要的代码是最少的:[12]

```
int scull_release(struct inode *inode, struct file *filp)
```

```
{
 return 0;
}
```

你可能想知道当一个设备文件关闭次数超过它被打开的次数会发生什么. 毕竟, dup 和 fork 系统调用不调用 open 来创建打开文件的拷贝; 每个拷贝接着在程序终止时被关闭. 例如, 大部分程序不打开它们的 stdin 文件(或设备), 但是它们都以关闭它结束. 当一个打开的设备文件已经真正被关闭时驱动如何知道?

答案简单: 不是每个 close 系统调用引起调用 release 方法. 只有真正释放设备数据结构的调用会调用这个方法 -- 因此得名. 内核维持一个文件结构被使用多少次的计数. fork 和 dup 都不创建新文件(只有 open 这样); 它们只递增正存在的结构中的计数. close 系统调用仅在文件结构计数掉到 0 时执行 release 方法, 这在结构被销毁时发生. release 方法和 close 系统调用之间的这种关系保证了你的驱动一次 open 只看到一次 release.

注意, flush 方法在每次应用程序调用 close 时都被调用. 但是, 很少驱动实现 flush, 因为常常在 close 时没有什么要做, 除非调用 release.

如你会想到的, 前面的讨论即便是应用程序没有明显地关闭它打开的文件也适用: 内核在进程 exit 时自动关闭了任何文件, 通过在内部使用 close 系统调用.

---

[12] 其他风味的设备由不同的函数关闭, 因为 scull_open 为每个设备替换了不同的 filp->f_op. 我们在介绍每种风味时再讨论它们.

3.6. scull 的内存使用

在介绍读写操作前, 我们最好看看如何以及为什么 scull 进行内存分配. "如何"是需要全面理解代码, "为什么"演示了驱动编写者需要做的选择, 尽管 scull 明确地不是典型设备.

本节只处理 scull 中的内存分配策略, 不展示给你编写真正驱动需要的硬件管理技能. 这些技能在第 9 章和第 10 章介绍. 因此, 你可跳过本章, 如果你不感兴趣于理解面向内存的 scull 驱动的内部工作.

scull 使用的内存区, 也称为一个设备, 长度可变. 你写的越多, 它增长越多; 通过使用一个短文件覆盖设备来进行修整.

scull 驱动引入 2 个核心函数来管理 Linux 内核中的内存. 这些函数, 定义在 <linux/slab.h>, 是:

```
void *kmalloc(size_t size, int flags);
void kfree(void *ptr);
```

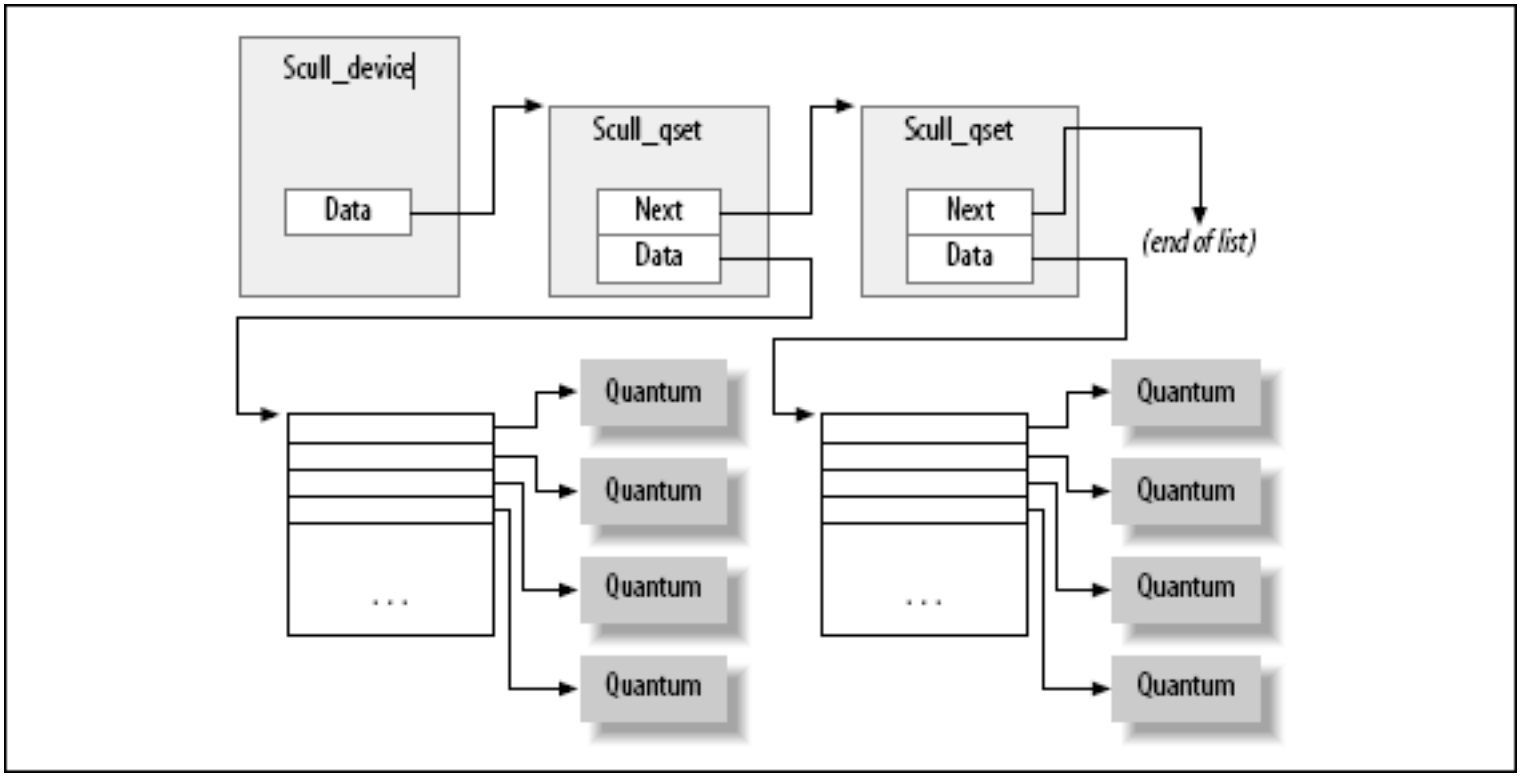
对 kmalloc 的调用试图分配 size 字节的内存; 返回值是指向那个内存的指针或者如果分配失败为 NULL. flags 参数用来描述内存应当如何分配; 我们在第 8 章详细查看这些标志. 对于现在, 我们一直使用 GFP_KERNEL. 分配的内存应当用 kfree 来释放. 你应当从不传递任何不是从 kmalloc 获得的东西给 kfree. 但是, 传递一个 NULL 指针给 kfree 是合法的.

kmalloc 不是最有效的分配大内存区的方法(见第 8 章), 所以挑选给 scull 的实现不是一个特别巧妙的. 一个巧妙的源码实现可能更难阅读, 而本节的目标是展示读和写, 不是内存管理. 这是为什么代码只是使用 kmalloc 和 kfree 而不依靠整页的分配, 尽管这个方法会更有效.

在 flip 一边, 我们不想限制"设备"区的大小, 由于理论上的和实践上的理由. 理论上, 给在被管理的数据项施加武断的限制总是个坏想法. 实践上, scull 可用来暂时地吃光你系统中的内存, 以便运行在低内存条件下的测试. 运行这样的测试可能会帮助你理解系统的内部. 你可以使用命令 cp /dev/zero /dev/scull0 来用 scull 吃掉所有的真实 RAM, 并且你可以使用 dd 工具来选择贝多少数据给 scull 设备.

在 scull, 每个设备是一个指针链表, 每个都指向一个 scull_dev 结构. 每个这样的结构, 缺省地, 指向最多 4 兆字节, 通过一个中间指针数组. 发行代码使用一个 1000 个指针的数组指向每个 4000 字节的区域. 我们称每个内存区域为一个量子, 数组(或者它的长度) 为一个量子集. 一个 scull 设备和它的内存区如图一个 scull 设备的布局所示.

图 3.1. 一个 scull 设备的布局



选定的数字是这样, 在 scull 中写单个一个字节消耗 8000 或 12,000 KB 内存: 4000 是量子, 4000 或者 8000 是量子集(根据指针在目标平台上是用 32位还是 64位表示). 相反, 如果你写入大量数据, 链表的开销不是太坏. 每 4 MB 数据只有一个链表元素, 设备的最大尺寸受限于计算机的内存大小.

为量子和量子集选择合适的值是一个策略问题, 而不是机制, 并且优化的值依赖于设备如何使用. 因此, scull 驱动不应当强制给量子和量子集使用任何特别的值. 在 scull 中, 用户可以掌管改变这些值, 有几个途径:编译时间通过改变 scull.h 中的宏 SCULL_QUANTUM 和 SCULL_QSET, 在模块加载时设定整数值 scull_quantum 和 scull_qset, 或者使用 ioctl 在运行时改变当前值和缺省值.

使用宏定义和一个整数值来进行编译时和加载时配置, 是对于如何选择主编号的回忆. 我们在驱动中任何与策略相关或专断的值上运用这个技术.

余下的唯一问题是如果选择缺省值. 在这个特殊情况下, 问题是找到最好的平衡, 由填充了一半的量子和量子集导致内存浪费, 如果量子和量子集小的情况下分配释放和指针连接引起开销. 另外, kmalloc 的内部设计应当考虑进去. (现在我们不追求这点, 不过; kmalloc 的内部在第 8 章探索.) 缺省值的选择来自假设测试时可能有大量数据写进 scull, 尽管设备的正常使用最可能只传送几 KB 数据.

我们已经见过内部代表我们设备的 scull_dev 结构. 结构的 quantum 和 qset 分别代表设备的量子和量子集大小. 实际数据, 但是, 是由一个不同的结构跟踪, 我们称为 struct scull_qset:

```
struct scull_qset {
 void **data;
 struct scull_qset *next;
};
```

下一个代码片段展示了实际中 struct scull_dev 和 struct scull_qset 是如何被用来持有数据的. sucll_trim 函数负责释放整个数据区, 由 scull_open 在文件为写而打开时调用. 它简单地遍历列表并且释放它发现的任何量子和量子集.

```
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 也用在模块清理函数中, 来归还 scull 使用的内存给系统.

3.7. 读和写

读和写方法都进行类似的任务, 就是, 从和到应用程序代码拷贝数据. 因此, 它们的原型相当相似, 可以同时介绍它们:

```
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 2 个方法, filp 是文件指针, count 是请求的传输数据大小. buff 参数指向持有被写入数据的缓存, 或者放入新数据的空缓存. 最后, offp 是一个指针指向一个"long offset type"对象, 它指出用户正在存取的文件位置. 返回值是一个"signed size type"; 它的使用在后面讨论.

让我们重复一下, read 和 write 方法的 buff 参数是用户空间指针. 因此, 它不能被内核代码直接解引用. 这个限制有几个理由:

- 依赖于你的驱动运行的体系, 以及内核被如何配置的, 用户空间指针当运行于内核模式可能根本是无效的. 可能没有那个地址的映射, 或者它可能指向一些其他的随机数据.
- 就算这个指针在内核空间是同样的东西, 用户空间内存是分页的, 在做系统调用时这个内存可能没有在 RAM 中. 试图直接引用用户空间内存可能产生一个页面错, 这是内核代码不允许做的事情. 结果可能是一个"oops", 导致进行系统调用的进程死亡.
- 置疑中的指针由一个用户程序提供, 它可能是错误的或者恶意的. 如果你的驱动盲目地解引用一个用户提供的指针, 它提供了一个打开的门路使用户空间程序存取或覆盖系统任何地方的内存. 如果你不想负责你的用户的系统的安全危险, 你就不能直接解引用用户空间指针.

显然, 你的驱动必须能够存取用户空间缓存以完成它的工作. 但是, 为安全起见这个存取必须使用特殊的, 内核提供的函数. 我们介绍几个这样的函数(定义于 <asm/uaccess.h>), 剩下的在第一章"使用 ioctl 参数"一节中. 它们使用一些特殊的, 依赖体系的技巧来确保内核和用户空间的数据传输安全和正确.

scull 中的读写代码需要拷贝一整段数据到或者从用户地址空间. 这个能力由下列内核函数提供, 它们拷贝一个任意的字节数组, 并且位于大部分读写实现的核心中.

```
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);
```

尽管这些函数表现象正常的 memcpy 函数, 必须加一点小心在从内核代码中存取用户空间. 寻址的用户也当前可能不在内存, 虚拟内存子系统会使进程睡眠在这个页被传送到位时. 例如, 这发生在

必须从交换空间获取页的时候. 对于驱动编写者来说, 最终结果是任何存取用户空间的函数必须是可重入的, 必须能够和其他驱动函数并行执行, 并且, 特别的, 必须在一个它能够合法地睡眠的位置. 我们在第 5 章再回到这个主题.

这 2 个函数的角色不限于拷贝数据到和从用户空间: 它们还检查用户空间指针是否有效. 如果指针无效, 不进行拷贝; 如果在拷贝中遇到一个无效地址, 另一方面, 只拷贝部分数据. 在 2 种情况下, 返回值是还要拷贝的数据量. scull 代码查看这个错误返回, 并且如果它不是 0 就返回 -EFAULT 给用户.

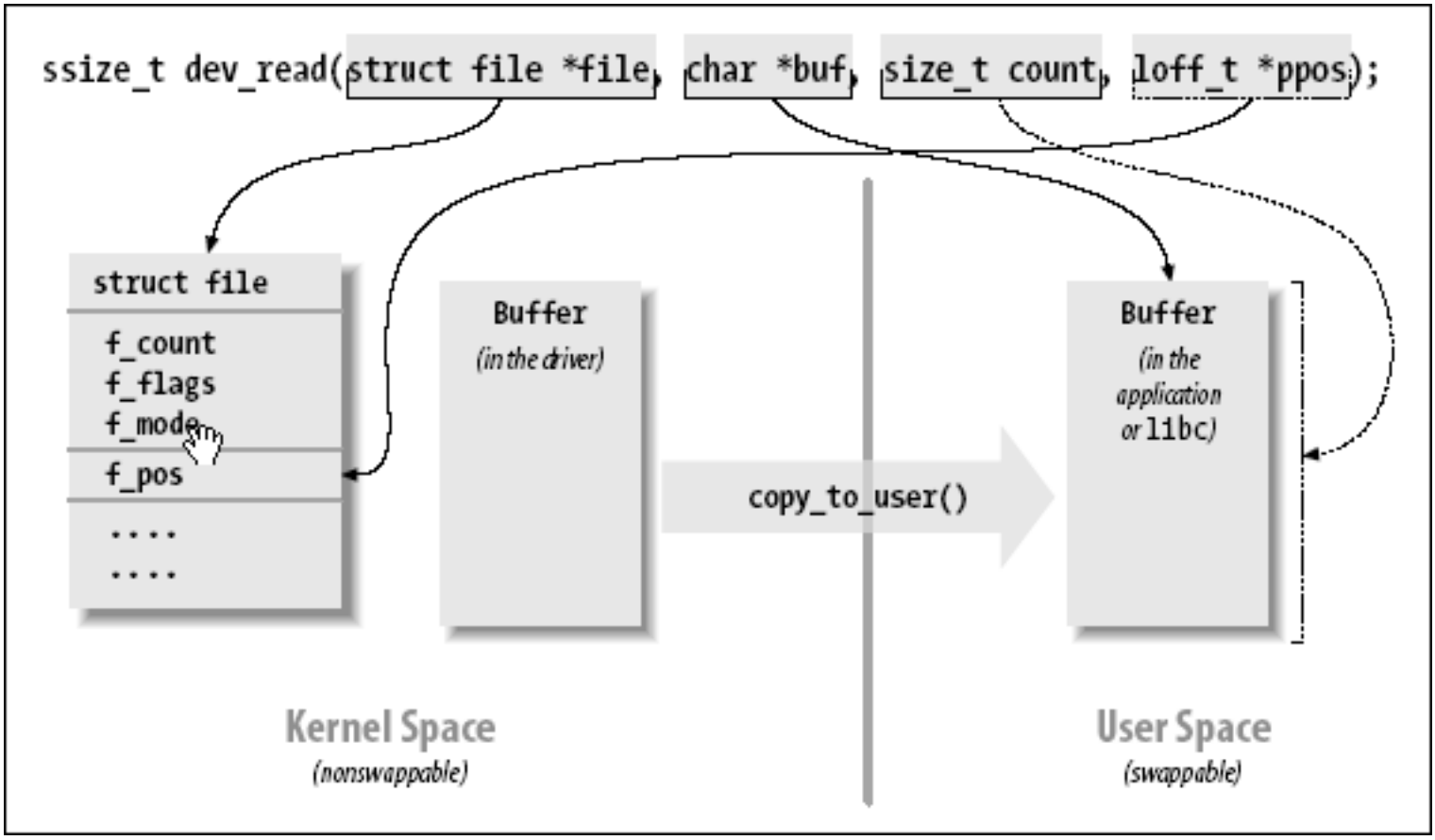
用户空间存取和无效用户空间指针的主题有些高级, 在第 6 章讨论. 然而, 值得注意的是如果你不需要检查用户空间指针, 你可以调用 __copy_to_user 和 __copy_from_user 来代替. 这是有用处的, 例如, 如果你知道你已经检查了这些参数. 但是, 要小心; 事实上, 如果你不检查你传递给这些函数的用户空间指针, 那么你可能造成内核崩溃和/或安全漏洞.

至于实际的设备方法, read 方法的任务是从设备拷贝数据到用户空间(使用 copy_to_user), 而 write 方法必须从用户空间拷贝数据到设备(使用 copy_from_user). 每个 read 或 write 系统调用请求一个特定数目字节的传送, 但是驱动可自由传送较少数据 -- 对读和写这确切的规则稍微不同, 在本章后面描述.

不管这些方法传送多少数据, 它们通常应当更新 *offp 中的文件位置来表示在系统调用成功完成后当前的文件位置. 内核接着在适当时候传播文件位置的改变到文件结构. pread 和 pwrite 系统调用有不同的语义; 它们从一个给定的文件偏移操作, 并且不改变其他的系统调用看到的文件位置. 这些调用传递一个指向用户提供的位置的指针, 并且放弃你的驱动所做的改变.

图给 read 的参数表示了一个典型读实现是如何使用它的参数.

图 3.2. 给 read 的参数



read 和 write 方法都在发生错误时返回一个负值. 相反, 大于或等于 0 的返回值告知调用程序有多少字节已经成功传送. 如果一些数据成功传送接着发生错误, 返回值必须是成功传送的字节数, 错误不报告直到函数下一次调用. 实现这个传统, 当然, 要求你的驱动记住错误已经发生, 以便它们可以在以后返回错误状态.

尽管内核函数返回一个负数指示一个错误, 这个数的值指出所发生的错误类型(如第 2 章介绍), 用户空间运行的程序常常看到 -1 作为错误返回值. 它们需要存取 errno 变量来找出发生了什么. 用户空间的行为由 POSIX 标准来规定, 但是这个标准没有规定内核内部如何操作.

3.7.1. read 方法

read 的返回值由调用的应用程序解释:

- 如果这个值等于传递给 read 系统调用的 count 参数, 请求的字节数已经被传送. 这是最好的情况.
- 如果是正数, 但是小于 count, 只有部分数据被传送. 这可能由于几个原因, 依赖于设备. 常常, 应用程序重新试着读取. 例如, 如果你使用 fread 函数来读取, 库函数重新发出系统调用直到请求的数据传送完成.
- 如果值为 0, 到达了文件末尾(没有读取数据).
- 一个负值表示有一个错误. 这个值指出了什么错误, 根据 <linux/errno.h>. 出错的典型返回值包括 -EINTR(被打断的系统调用) 或者 -EFAULT(坏地址).

前面列表中漏掉的是这种情况"没有数据, 但是可能后来到达". 在这种情况下, read 系统调用应当阻塞. 我们将在第 6 章涉及阻塞.

scull 代码利用了这些规则. 特别地, 它利用了部分读规则. 每个 scull_read 调用只处理单个数据量子, 不实现一个循环来收集所有的数据; 这使得代码更短更易读. 如果读程序确实需要更多数据, 它重新调用. 如果标准 I/O 库(例如, fread)用来读取设备, 应用程序甚至不会注意到数据传送的量子化.

如果当前读取位置大于设备大小, scull 的 read 方法返回 0 来表示没有可用的数据(换句话说, 我们在文件尾). 这个情况发生在如果进程 A 在读设备, 同时进程 B 打开它写, 这样将设备截短为 0. 进程 A 突然发现自己过了文件尾, 下一个读调用返回 0.

这是 read 的代码(忽略对 down_interruptible 的调用并且现在为 up; 我们在下一章中讨论它们):

```
ssize_t scull_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr; /* the first listitem */
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset; /* how many bytes in the listitem */
    int item, s_pos, q_pos, rest;
    ssize_t retval = 0;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    if (*f_pos >= dev->size)
        goto out;
    if (*f_pos + count > dev->size)
        count = dev->size - *f_pos;

    /* find listitem, qset index, and offset in the quantum */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum;
    q_pos = rest % quantum;

    /* follow the list up to the right position (defined elsewhere) */
    dptr = scull_follow(dev, item);
    if (dptr == NULL || !dptr->data || ! dptr->data[s_pos])
        goto out; /* don't fill holes */

    /* read only up to the end of this quantum */
    if (count > quantum - q_pos)
```

```
        count = quantum - q_pos;

    if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count))
    {
        retval = -EFAULT;
        goto out;

    }
    *f_pos += count;
    retval = count;

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

3.7.2. write 方法

write, 象 read, 可以传送少于要求的数据, 根据返回值的下列规则:

- 如果值等于 count, 要求的字节数已被传送.
- 如果正值, 但是小于 count, 只有部分数据被传送. 程序最可能重试写入剩下的数据.
- 如果值为 0, 什么没有写. 这个结果不是一个错误, 没有理由返回一个错误码. 再一次, 标准库重试写调用. 我们将在第 6 章查看这种情况的确切含义, 那里介绍了阻塞.
- 一个负值表示发生一个错误; 如同对于读, 有效的错误值是定义于 <linux/errno.h>中.

不幸的是, 仍然可能有发出错误消息的不当行为程序, 它在进行了部分传送时终止. 这是因为一些程序员习惯看写调用要么完全失败要么完全成功, 这实际上是大部分时间的情况, 应当也被设备支持. scull 实现的这个限制可以修改, 但是我们不想使代码不必要地复杂.

write 的 scull 代码一次处理单个量子, 如 read 方法做的:

```
ssize_t scull_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr;
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset;
    int item, s_pos, q_pos, rest;
    ssize_t retval = -ENOMEM; /* value used in "goto out" statements */
    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
```

```
    /* find listitem, qset index and offset in the quantum */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum;
    q_pos = rest % quantum;
    /* follow the list up to the right position */
    dptr = scull_follow(dev, item);
    if (dptr == NULL)
        goto out;
    if (!dptr->data)
    {
        dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
        if (!dptr->data)
            goto out;
        memset(dptr->data, 0, qset * sizeof(char *));
    }
    if (!dptr->data[s_pos])
    {
        dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
        if (!dptr->data[s_pos])

            goto out;
    }
    /* write only up to the end of this quantum */
    if (count > quantum - q_pos)

        count = quantum - q_pos;
    if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count))
    {
        retval = -EFAULT;
        goto out;

    }
    *f_pos += count;
    retval = count;

    /* update the size */
    if (dev->size < *f_pos)
        dev->size = *f_pos;

out:
    up(&dev->sem);
```

```
    return retval;

}
```

3.7.3. readv 和 writev

Unix 系统已经长时间支持名为 readv 和 writev 的 2 个系统调用. 这些 read 和 write 的"矢量"版本使用一个结构数组, 每个包含一个缓存的指针和一个长度值. 一个 readv 调用被期望来轮流读取指示的数量到每个缓存. 相反, writev 要收集每个缓存的内容到一起并且作为单个写操作送出它们.

如果你的驱动不提供方法来处理矢量操作, readv 和 writev 由多次调用你的 read 和 write 方法来实现. 在许多情况, 但是, 直接实现 readv 和 writev 能获得更大的效率.

矢量操作的原型是:

```
ssize_t (*readv) (struct file *filp, const struct iovec *iov, unsigned long count, loff_t *ppos);
ssize_t (*writev) (struct file *filp, const struct iovec *iov, unsigned long count, loff_t *ppos);
```

这里, filp 和 ppos 参数与 read 和 write 的相同. iovec 结构, 定义于 <linux/uio.h>, 如同:

```
struct iovec
{
   void __user *iov_base; __kernel_size_t iov_len;
};
```

每个 iovec 描述了一块要传送的数据; 它开始于 iov_base (在用户空间)并且有 iov_len 字节长. count 参数告诉有多少 iovec 结构. 这些结构由应用程序创建, 但是内核在调用驱动之前拷贝它们到内核空间.

矢量操作的最简单实现是一个直接的循环, 只是传递出去每个 iovec 的地址和长度给驱动的 read 和 write 函数. 然而, 有效的和正确的行为常常需要驱动更聪明. 例如, 一个磁带驱动上的 writev 应当将全部 iovec 结构中的内容作为磁带上的单个记录.

很多驱动, 但是, 没有从自己实现这些方法中获益. 因此, scull 省略它们. 内核使用 read 和 write 来模拟它们, 最终结果是相同的.

3.8. 使用新设备

一旦你装备好刚刚描述的 4 个方法, 驱动可以编译并测试了; 它保留了你写给它的任何数据, 直到你用新数据覆盖它. 这个设备表现如一个数据缓存器, 它的长度仅仅受限于可用的真实 RAM 的数量. 你可试着使用 cp, dd, 以及 输入/输出重定向来测试这个驱动.

free 命令可用来看空闲内存的数量如何缩短和扩张的, 依据有多少数据写入 scull.

为对一次读写一个量子有更多信心, 你可增加一个 printk 在驱动的适当位置, 并且观察当应用程序读写大块数据中发生了什么. 可选地, 使用 strace 工具来监视程序发出的系统调用以及它们的返回值. 跟踪一个 cp 或者一个 ls -l > /dev/scull0 展示了量子化的读和写. 监视(以及调试)技术在第 4 章详细介绍.

3.9. 快速参考

本章介绍了下面符号和头文件. struct file_operations 和 struct file 中的成员的列表这里不重复了.

```
#include <linux/types.h>
dev_t
```

dev_t 是用来在内核里代表设备号的类型.

```
int MAJOR(dev_t dev);
int MINOR(dev_t dev);
```

从设备编号中抽取主次编号的宏.

```
dev_t MKDEV(unsigned int major, unsigned int minor);
```

从主次编号来建立 dev_t 数据项的宏定义.

```
#include <linux/fs.h>
```

"文件系统"头文件是编写设备驱动需要的头文件. 许多重要的函数和数据结构在此定义.

```
int register_chrdev_region(dev_t first, unsigned int count, char *name)
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name)
void unregister_chrdev_region(dev_t first, unsigned int count);
```

允许驱动分配和释放设备编号的范围的函数. register_chrdev_region 应当用在事先知道需要的主编号时; 对于动态分配, 使用 alloc_chrdev_region 代替.

```
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);
```

老的(2.6 之前) 字符设备注册函数. 它在 2.6 内核中被模拟, 但是不应当给新代码使用. 如果主编号不是 0, 可以不变地用它; 否则一个动态编号被分配给这个设备.

```
int unregister_chrdev(unsigned int major, const char *name);
```

恢复一个由 register_chrdev 所作的注册的函数. major 和 name 字符串必须包含之前用来注册设备时同样的值.

```
struct file_operations;
struct file;
struct inode;
```

大部分设备驱动使用的 3 个重要数据结构. file_operations 结构持有一个字符驱动的方法;

struct file 代表一个打开的文件, struct inode 代表磁盘上的一个文件.

```
#include <linux/cdev.h>
struct cdev *cdev_alloc(void);
void cdev_init(struct cdev *dev, struct file_operations *fops);
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
void cdev_del(struct cdev *dev);
```

cdev 结构管理的函数, 它代表内核中的字符设备.

```
#include <linux/kernel.h>
container_of(pointer, type, field);
```

一个传统宏定义, 可用来获取一个结构指针, 从它里面包含的某个其他结构的指针.

```
#include <asm/uaccess.h>
```

这个包含文件声明内核代码使用的函数来移动数据到和从用户空间.

```
unsigned long copy_from_user (void *to, const void *from, unsigned long count);
unsigned long copy_to_user (void *to, const void *from, unsigned long count);
```

在用户空间和内核空间拷贝数据.

上一页 3.8. 使用新设备 | 上一级 起始页 | 下一页 第 4 章 调试技术

第 4 章 调试技术

上一页 下一页

第 4 章 调试技术

目录

内核编程带有它自己的, 独特的调试挑战性. 内核代码无法轻易地在一个调试器下运行, 也无法轻易的被跟踪, 因为它是一套没有与特定进程相关连的功能的集合. 内核代码错误也特别难以重现, 它们会牵连整个系统与它们一起失效, 从而破坏了大量的能用来追踪错误的证据.

本章介绍了在如此艰难情况下能够用以监视内核代码和跟踪错误的技术.

4.1. 内核中的调试支持

在第 2 章, 我们建议你建立并安装你自己的内核, 而不是运行来自你的发布商的现成的内核. 运行你自己的内核的最充分的理由之一是内核开发者已经在内核自身中构建了多个调试特性. 这些特性能产生额外的输出并降低性能, 因此发布商的产品内核中往往不会使能它们. 作为一个内核开发者, 但是, 你有不同的优先权并且会乐于接收这些格外的内核调试支持带来的开销.

这里, 我们列出用来开发的内核应当激活的配置选项. 除了另外指出的, 所有的这些选项都在 "kernel hacking" 菜单, 不管什么样的你喜欢的内核配置工具. 注意有些选项不是所有体系都支持.

CONFIG_DEBUG_KERNEL

这个选项只是使其他调试选项可用; 它应当打开, 但是它自己不激活任何的特性.

CONFIG_DEBUG_SLAB

这个重要的选项打开了内核内存分配函数的几类检查; 激活这些检查, 就可能探测到一些内存覆盖和遗漏初始化的错误. 被分配的每一个字节在递交给调用者之前都设成 0xa5, 随后在释放时被设成 0x6b. 你在任何时候如果见到任一个这种"坏"模式重复出现在你的驱动输出 (或者常常在一个 oops 的列表), 你会确切知道去找什么类型的错误. 当激活调试, 内核还会在每个分配的内存对象的前后放置特别的守护值; 如果这些值曾被改动, 内核知道有人已覆盖了一个内存分配区, 它大声抱怨. 各种的对更模糊的问题的检查也给激活了.

CONFIG_DEBUG_PAGEALLOC

满的页在释放时被从内核地址空间去除. 这个选项会显著拖慢系统, 但是它也能快速指出某些类型的内存损坏错误.

CONFIG_DEBUG_SPINLOCK

激活这个选项, 内核捕捉对未初始化的自旋锁的操作, 以及各种其他的错误(例如 2 次解锁同一个锁).

CONFIG_DEBUG_SPINLOCK_SLEEP

这个选项激活对持有自旋锁时进入睡眠的检查. 实际上, 如果你调用一个可能会睡眠的函数, 它就抱怨, 即便这个有疑问的调用没有睡眠.

CONFIG_INIT_DEBUG

用__init (或者 __initdata) 标志的项在系统初始化或者模块加载后都被丢弃. 这个选项激活了对代码的检查, 这些代码试图在初始化完成后存取初始化时内存.

CONFIG_DEBUG_INFO

这个选项使得内核在建立时包含完整的调试信息. 如果你想使用 gdb 调试内核, 你将需要这些信息. 如果你打算使用 gdb, 你还要激活 CONFIG_FRAME_POINTER.

CONFIG_MAGIC_SYSRQ

激活"魔术 SysRq"键. 我们在本章后面的"系统挂起"一节查看这个键.

CONFIG_DEBUG_STACKOVERFLOW
CONFIG_DEBUG_STACK_USAGE

这些选项能帮助跟踪内核堆栈溢出. 堆栈溢出的确证是一个 oops 输出, 但是没有任何形式的合理的回溯. 第一个选项给内核增加了明确的溢出检查; 第 2 个使得内核监测堆栈使用并作一些统计, 这些统计可以用魔术 SysRq 键得到.

CONFIG_KALLSYMS

这个选项(在"Generl setup/Standard features"下)使得内核符号信息建在内核中; 缺省是激活的. 符号选项用在调试上下文中; 没有它, 一个 oops 列表只能以 16 进制格式给你一个内核回溯, 这不是很有用.

CONFIG_IKCONFIG
CONFIG_IKCONFIG_PROC

这些选项(在"Generl setup"菜单)使得完整的内核配置状态被建立到内核中, 可以通过 /proc 来使其可用. 大部分内核开发者知道他们使用的哪个配置, 并不需要这些选项(会使得内核更大). 但是如果你试着调试由其他人建立的内核中的问题, 它们可能有用.

CONFIG_ACPI_DEBUG

在"Power management/ACPI"下. 这个选项打开详细的 ACPI (Advanced Configuration and Power Interface) 调试信息, 它可能有用如果你怀疑一个问题和 ACPI 相关.

CONFIG_DEBUG_DRIVER

在"Device drivers"下. 打开了驱动核心的调试信息, 可用以追踪低层支持代码的问题. 我们在第 14 章查看驱动核心.

CONFIG_SCSI_CONSTANTS

这个选项, 在"Device drivers/SCSI device support"下, 建立详细的 SCSI 错误消息的信息. 如果你在使用 SCSI 驱动, 你可能需要这个选项.

CONFIG_INPUT_EVBUG

这个选项(在"Device drivers/Input device support"下)打开输入事件的详细日志. 如果你使用一个输入设备的驱动, 这个选项可能会有用. 然而要小心这个选项的安全性的隐含意义: 它记录了你键入的任何东西, 包括你的密码.

CONFIG_PROFILING

这个选项位于"Profiling support"之下. 剖析通常用在系统性能调整, 但是在追踪一些内核挂起和相关问题上也有用.

我们会再次遇到一些上面的选项, 当我们查看各种方法来追踪内核问题时. 但是首先, 我们要看一下经典的调试技术: print 语句.

上一页 下一页
3.9. 快速参考 起始页 4.2. 用打印调试

4.2. 用打印调试

最常用的调试技术是监视, 在应用程序编程当中是通过在合适的地方调用 printf 来实现. 在你调试内核代码时, 你可以通过 printk 来达到这个目的.

4.2.1. printk

我们在前面几章中使用 printk 函数, 简单地假设它如同 printf 一样使用. 现在到时候介绍一些不同的地方了.

一个不同是 printk 允许你根据消息的严重程度对其分类, 通过附加不同的记录级别或者优先级在消息上. 你常常用一个宏定义来指示记录级别. 例如, KERN_INFO, 我们之前曾在一些打印语句的前面看到过, 是消息记录级别的一种可能值. 记录宏定义扩展成一个字串, 在编译时与消息文本连接在一起; 这就是为什么下面的在优先级和格式串之间没有逗号的原因. 这里有 2 个 printk 命令的例子, 一个调试消息, 一个紧急消息:

```
printk(KERN_DEBUG "Here I am: %s:%i\n", __FILE__, __LINE__);
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

有 8 种可能的记录字串, 在头文件 <linux/kernel.h> 里定义; 我们按照严重性递减的顺序列出它们:

KERN_EMERG

用于紧急消息, 常常是那些崩溃前的消息.

KERN_ALERT

需要立刻动作的情形.

KERN_CRIT

严重情况, 常常与严重的硬件或者软件失效有关.

KERN_ERR

用来报告错误情况; 设备驱动常常使用 KERN_ERR 来报告硬件故障.

KERN_WARNING

有问题的情况的警告, 这些情况自己不会引起系统的严重问题.

KERN_NOTICE

正常情况, 但是仍然值得注意. 在这个级别一些安全相关的情况会报告.

KERN_INFO

信息型消息. 在这个级别, 很多驱动在启动时打印它们发现的硬件的信息.

KERN_DEBUG

用作调试消息.

每个字串(在宏定义扩展里)代表一个在角括号中的整数. 整数的范围从 0 到 7, 越小的数表示越大的优先级.

一条没有指定优先级的 printk 语句缺省是 DEFAULT_MESSAGE_LOGLEVEL, 在 kernel/printk.c 里指定作为一个整数. 在 2.6.10 内核中, DEFAULT_MESSAGE_LOGLEVEL 是 KERN_WARNING, 但是在过去已知是改变的.

基于记录级别, 内核可能打印消息到当前控制台, 可能是一个文本模式终端, 串口, 或者是一台并口打印机. 如果优先级小于整型值 console_loglevel, 消息被递交给控制台, 一次一行(除非提供一个新行结尾, 否则什么都不发送). 如果 klogd 和 syslogd 都在系统中运行, 内核消息被追加到 /var/log/messages (或者另外根据你的 syslogd 配置处理), 独立于 console_loglevel. 如果 klogd 没有运行, 你只有读 /proc/kmsg (用 dmsg 命令最易做到)将消息取到用户空间. 当使用 klogd 时, 你应当记住, 它不会保存连续的同样的行; 它只保留第一个这样的行, 随后是, 它收到的重复行数.

变量 console_loglevel 初始化成 DEFAULT_CONSOLE_LOGLEVEL, 并且可通过 sys_syslog 系统调用修改. 一种修改它的方法是在调用 klogd 时指定 -c 开关, 在 klogd 的 manpage 里有指定. 注意要改变当前值, 你必须先杀掉 klogd, 接着使用 -c 选项重启它. 另外, 你可写一个程序来改变控制台记录级别. 你会发现这样一个程序的版本在由 O' Reilly 提供的 FTP 站点上的 miscprogs/setlevel.c. 新的级别指定未一个整数, 在 1 和 8 之前, 包含 1 和 8. 如果它设为 1, 只有 0 级消息 (KERN_EMERG)到达控制台; 如果它设为 8, 所有消息, 包括调试消息, 都显示.

也可以通过文本文件 /proc/sys/kernel/printk 读写控制台记录级别. 这个文件有 4 个整型值: 当前记录级别, 适用没有明确记录级别的消息的缺省级别, 允许的最小记录级别, 以及启动时缺省记录级别. 写一个单个值到这个文件就改变当前记录级别成这个值; 因此, 例如, 你可以使所有内核消息出现在控制台, 通过简单地输入:

```
# echo 8 > /proc/sys/kernel/printk
```

现在应当清楚了为什么 hello.c 例子使用 KERN_ALERT 标志; 它们是要确保消息会出现在控制台上.

4.2.2. 重定向控制台消息

Linux 在控制台记录策略上允许一些灵活性, 它允许你发送消息到一个指定的虚拟控制台(如果你的控制台使用的是文本屏幕). 缺省地, 这个"控制台"是当前虚拟终端. 为了选择一个不同地虚拟终端来接收消息, 你可对任何控制台设备调用 ioctl(TIOCLINUX). 下面的程序, setconsole, 可以用来选择哪个控制台接收内核消息; 它必须由超级用户运行, 可以从 misc-progs 目录得到.

下面是全部程序. 应当使用一个参数来指定用以接收消息的控制台的编号.

```
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]); /* the chosen console */
  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 使用特殊的 ioctl 命令 TIOCLINUX, 来实现特定于 linux 的功能. 为使用 TIOCLINUX, 你传递它一个指向字节数组的指针作为参数. 数组的第一个字节是一个数, 指定需要的子命令, 下面的字节是特对于子命令的. 在 setconsole

里, 使用子命令 11, 下一个字节(存于 bytes[1])指定虚拟控制台. TIOCLINUX 的完整描述在内核源码的 drivers/char/tty_io.c 里.

4.2.3. 消息是如何记录的

printk 函数将消息写入一个 __LOG_BUF_LEN 字节长的环形缓存, 长度值从 4 KB 到 1 MB, 由配置内核时选择. 这个函数接着唤醒任何在等待消息的进程, 就是说, 任何在系统调用中睡眠或者在读取 /proc/kmsg 的进程. 这 2 个日志引擎的接口几乎是等同的, 但是注意, 从 /proc/kmsg 中读取是从日志缓存中消费数据, 然而 syslog 系统调用能够选择地在返回日志数据地同时保留它给其他进程. 通常, 读取 /proc 文件容易些并且是 klogd 的缺省做法. dmesg 命令可用来查看缓存的内容, 不会冲掉它; 实际上, 这个命令将缓存区的整个内容返回给 stdout, 不管它是否已经被读过.

在停止 klogd 后, 如果你偶尔手工读取内核消息, 你会发现 /proc 看起来象一个 FIFO, 读者阻塞在里面, 等待更多数据. 显然, 你无法以这种方式读消息, 如果 klogd 或者其他进程已经在读同样的数据, 因为你要竞争它.

如果环形缓存填满, printk 绕回并在缓存的开头增加新数据, 覆盖掉最老的数据. 因此, 这个记录过程会丢失最老的数据. 这个问题相比于使用这样一个环形缓存的优点是可以忽略的. 例如, 环形缓存允许系统即便没有一个日志进程也可运行, 在没有人读它的时候可以通过覆盖旧数据浪费最少的内存. Linux 对于消息的解决方法的另一个特性是, printk 可以从任何地方调用, 甚至从一个中断处理里面, 没有限制能打印多少数据. 唯一的缺点是可能丢失一些数据.

如果 klogd 进程在运行, 它获取内核消息并分发给 syslogd, syslogd 接着检查 /etc/syslog.conf 来找出如何处理它们. syslogd 根据一个设施和一个优先级来区分消息; 这个设施和优先级的允许值在 <sys/syslog.h> 中定义. 内核消息由 LOG_KERN 设施来记录, 在一个对应于 printk 使用的优先级上(例如, LOG_ERR 用于 KERN_ERR 消息). 如果 klogd 没有运行, 数据保留在环形缓存中直到有人读它或者缓存被覆盖.

如果你要避免你的系统被来自你的驱动的监视消息击垮, 你或者给 klogd 指定一个 -f (文件) 选项来指示它保存消息到一个特定的文件, 或者定制 /etc/syslog.conf 来适应你的要求. 但是另外一种可能性是采用粗暴的方式: 杀掉 klogd 和详细地打印消息在一个没有用到的虚拟终端上,[13] 或者从一个没有用到的 xterm 上发出命令 cat /proc/kmsg.

4.2.4. 打开和关闭消息

在驱动开发的早期, printk 非常有助于调试和测试新代码. 当你正式发行驱动时, 换句话说, 你应当去掉, 或者至少关闭, 这些打印语句. 不幸的是, 你很可能会发现, 就在你认为你不再需要这些消息并去掉它们时, 你要在驱动中实现一个新特性(或者有人发现了一个 bug), 你想要至少再打开一个消息. 有几个方法来解决这 2 个问题, 全局性地打开或关闭你地调试消息和打开或关闭单个消息.

这里我们展示一种编码 printk 调用的方法, 你可以单独或全局地打开或关闭它们; 这个技术依靠定义一个宏, 在你想使用它时就转变成一个 printk (或者 printf)调用.

- 每个 printk 语句可以打开或关闭, 通过去除或添加单个字符到宏定义的名子.
- 所有消息可以马上关闭, 通过在编译前改变 CFLAGS 变量的值.
- 同一个 print 语句可以在内核代码和用户级代码中使用, 因此对于格外的消息, 驱动和测试程序能以同样的方式被管理.

下面的代码片断实现了这些特性, 直接来自头文件 scull.h:

```
#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 */
```

符号 PDEBUG 定义和去定义, 取决于 SCULL_DEBUG 是否定义, 和以何种方式显示消息适合代码运行的环境: 当它在内核中就使用内核调用 printk, 在用户空间运行就使用 libc 调用 fprintf 到标准错误输出. PDEBUGG 符号, 换句话说, 什么不作; 他可用来轻易地"注释" print 语句, 而不用完全去掉它们.

为进一步简化过程, 添加下面的行到你的 makfile 里:

```
# 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)
```

本节中出现的宏定义依赖 gcc 对 ANSI C 预处理器的扩展, 支持带可变个数参数的宏定义. 这个 gcc 依赖不应该是个问题, 因为无论如何内核固有的非常依赖于 gcc 特性. 另外, makefile 依赖 GNU 版本的 make; 再一次, 内核也依赖 GNU make, 所以这个依赖不是问题.

如果你熟悉 C 预处理器, 你可以扩展给定的定义来实现一个"调试级别"的概念, 定义不同的级别, 安排一个整数(或者位掩码)值给每个级别, 以便决定它应当多么详细.

但是每个驱动有它自己的特性和监视需求. 好的编程技巧是在灵活性和效率之间选择最好的平衡, 我们无法告诉你什么是最好的. 记住, 预处理器条件(连同代码中的常数表达式)在编译时执行, 因此你必须重新编译来打开或改变消息. 一个可能的选择是使用 C 条件句, 它在运行时执行, 因而, 能允许你在出现执行时打开或改变消息机制. 这是一个好的特性, 但是它在每次代码执行时需要额外的处理, 这样即便消息给关闭了也会影响效率. 有时这个效率损失无法接受.

本节出现的宏定义已经证明在多种情况下是有用的, 唯一的缺点是要求在任何对它的消息改变后重新编译.

4.2.5. 速率限制

如果你不小心, 你会发现自己用 printk 产生了上千条消息, 压倒了控制台并且, 可能地, 使系统日志文件溢出. 当使用一个慢速控制台设备(例如, 一个串口), 过量的消息速率也能拖慢系统或者只是使它不反应了. 非常难于着手于系统出错的地方, 当控制台不停地输出数据. 因此, 你应当非常注意你打印什么, 特别在驱动的产品版本以及特别在初始化完成

后. 通常, 产品代码在正常操作时不应当打印任何东西; 打印的输出应当是指示需要注意的异常情况.

另一方面, 你可能想发出一个日志消息, 如果你驱动的设备停止工作. 但是你应当小心不要做过了头. 一个面对失败永远继续的傻瓜进程能产生每秒上千次的尝试; 如果你的驱动每次都打印"my device is broken", 它可能产生大量的输出, 如果控制台设备慢就有可能霸占 CPU -- 没有中断用来驱动控制台, 就算是一个串口或者一个行打印机.

在很多情况下, 最好的做法是设置一个标志说, "我已经抱怨过这个了", 并不打印任何后来的消息只要这个标志设置着. 然而, 有几个理由偶尔发出一个"设备还是坏的"的提示. 内核已经提供了一个函数帮助这个情况:

```
int printk_ratelimit(void);
```

这个函数应当在你认为打印一个可能会常常重复的消息之前调用. 如果这个函数返回非零值, 继续打印你的消息, 否则跳过它. 这样, 典型的调用如这样:

```
if (printk_ratelimit())
   printk(KERN_NOTICE "The printer is still on fire\n");
```

printk_ratelimit 通过跟踪多少消息发向控制台而工作. 当输出级别超过一个限度, printk_ratelimit 开始返回 0 并使消息被扔掉.

printk_ratelimit 的行为可以通过修改 /proc/sys/kern/printk_ratelimit(在重新使能消息前等待的秒数) 和 /proc/sys/kernel/printk_ratelimit_burst(限速前可接收的消息数)来定制.

4.2.6. 打印设备编号

偶尔地, 当从一个驱动打印消息, 你会想打印与感兴趣的硬件相关联的设备号. 打印主次编号不是特别难, 但是, 为一致性考虑, 内核提供了一些实用的宏定义(在 <linux/kdev_t.h> 中定义)用于这个目的:

```
int print_dev_t(char *buffer, dev_t dev);
char *format_dev_t(char *buffer, dev_t dev);
```

两个宏定义都将设备号编码进给定的缓冲区; 唯一的区别是 print_dev_t 返回打印的字符数, 而 format_dev_t 返回缓存区; 因此, 它可以直接用作 printk 调用的参数, 但是必须记住 printk 只有提供一个结尾的新行才会刷行. 缓冲区应当足够大以存放一个设备号; 如果 64 位编号在以后的内核发行中明显可能, 这个缓冲区应当可能至少是 20 字节长.

---

[13] * 例如, 使用 setlevel 8; setconsole 10 来配置终端 10 来显示消息.

4.3. 用查询来调试

前面一节描述了 printk 是任何工作的以及怎样使用它. 没有谈到的是它的缺点.

大量使用 printk 能够显著地拖慢系统, 即便你降低 cosole_loglevel 来避免加载控制台设备, 因为 syslogd 会不停地同步它的输出文件; 因此, 要打印的每一行都引起一次磁盘操作. 从 syslogd 的角度这是正确的实现. 它试图将所有东西写到磁盘上, 防止系统刚好在打印消息后崩溃; 然而, 你不想只是为了调试信息的原因而拖慢你的系统. 可以在出现于 /etc/syslogd.conf 中的你的日志文件名前加一个连字号来解决这个问题[14]. 改变配置文件带来的问题是, 这个改变可能在你结束调试后保留在那里, 即便在正常系统操作中你确实想尽快刷新消息到磁盘. 这样永久改变的另外的选择是运行一个非 klogd 程序(例如 cat /proc/kmsg, 如之前建议的), 但是这可能不会提供一个合适的环境给正常的系统操作.

经常地, 最好的获得相关信息的方法是查询系统, 在你需要消息时, 不是连续地产生数据. 实际上, 每个 Unix 系统提供许多工具来获取系统消息: ps, netstat, vmstat, 等等.

有几个技术给驱动开发者来查询系统: 创建一个文件在 /proc 文件系统下, 使用 ioctl 驱动方法, 借助 sysfs 输出属性. 使用 sysfs 需要不少关于驱动模型的背景知识. 在 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软件创建的文件系统, 内核用来输出消息到外界. /proc 下的每个文件都绑到一个内核函数上, 当文件被读的时候即时产生文件内容. 我们已经见到一些这样的文件起作用; 例如, /proc/modules, 常常返回当前已加载的模块列表.

/proc 在 Linux 系统中非常多地应用. 很多现代 Linux 发布中的工具, 例如 ps, top, 以及 uptime, 从 /proc 中获取它们的信息. 一些设备驱动也通过 /proc 输出信息, 你的也可以这样做. /proc 文件系统是动态的, 因此你的模块可以在任何时候添加或去除条目.

完全特性的 /proc 条目可能是复杂的野兽; 另外, 它们可写也可读, 但是, 大部分时间, /proc 条目是只读的文件. 本节只涉及简单的只读情况. 那些感兴趣于实现更复杂的东西的人可以从这里获取基本知识; 接下来可参考内核源码来获知完整的信息.

在我们继续之前, 我们应当提及在 /proc 下添加文件是不鼓励的. /proc 文件系统在内核开发者看作是有点无法控制的混乱, 它已经远离它的本来目的了(是提供关于系统中运行的进程的信息). 建议新代码中使信息可获取的方法是利用 sysfs. 如同建议的, 使用 sysfs 需要对 Linux 设备模型的理解, 然而, 我们直到 14 章才接触它. 同时, /proc 下的文件稍稍容易创建, 并且它们完全适合调试目的, 所以我们在这里包含它们.

4.3.1.1. 在 /proc 里实现文件

所有使用 /proc 的模块应当包含 <linux/proc_fs.h> 来定义正确的函数.

要创建一个只读 /proc 文件, 你的驱动必须实现一个函数来在文件被读时产生数据. 当某个进程读文件时(使用 read 系统调用), 这个请求通过这个函数到达你的模块. 我们先看看这个函数并在本章后面讨论注册接口.

当一个进程读你的 /proc 文件, 内核分配了一页内存(就是说, PAGE_SIZE 字节), 驱动可以写入数据来返回给用户空间. 那个缓存区传递给你的函数, 是一个称为 read_proc 的方法:

```
int (*read_proc)(char *page, char **start, off_t offset, int count, int *eof, void *data);
```

page 指针是你写你的数据的缓存区; start 是这个函数用来说有关的数据写在页中哪里(下面更多关于这个); offset 和 count 对于 read 方法有同样的含义. eof 参数指向一个整数, 必须由驱动设置来指示它不再有数据返回, data 是驱动特定的数据指针, 你可以用做内部用途.

这个函数应当返回实际摆放于 page 缓存区的数据的字节数, 就象 read 方法对别的文件所作一样. 别的输出值是 *eof 和 *start. eof 是一个简单的标志, 但是 start 值的使用有些复杂; 它的目的是帮助实现大的(超过一页) /proc 文件.

start 参数有些非传统的用法. 它的目的是指示哪里(哪一页)找到返回给用户的数据. 当调用你的 proc_read 方法, *start 将会是 NULL. 如果你保持它为 NULL, 内核假定数据已放进 page 偏移是 0; 换句话说, 它假定一个头脑简单的 proc_read 版本, 它安放虚拟文件的整个内容到 page, 没有注意 offset 参数. 如果, 相反, 你设置 *start 为一个 非 NULL 值, 内核认为由 *start 指向的数据考虑了 offset, 并且准备好直接返回给用户. 通常, 返回少量数据的简单 proc_read 方法只是忽略 start. 更复杂的方法设置 *start 为 page 并且只从请求的 offset 那里开始安放数据.

还有一段距离到 /proc 文件的另一个主要问题, 它也打算解答 start. 有时内核数据结构的 ASCII 表示在连续的 read 调用中改变, 因此读进程可能发现从一个调用到下一个有不一致的数据. 如果 *start 设成一个小的整数值, 调用者用它来递增 filp-<f_pos 不依赖你返回的数据量, 因此使 f_pos 成为你的 read_proc 过程的一个内部记录数. 如果, 例如, 如果你的 read_proc 函数从一个大结构数组返回信息并且第一次调用返回了 5 个结构, *start可设成5. 下一个调用提供同一个数作为 offset; 驱动就知道从数组中第 6 个结构返回数据. 这是被它的作者承认的一个" hack ", 可以在 fs/proc/generic.c 见到.

注意, 有更好的方法实现大的 /proc 文件; 它称为 seq_file, 我们很快会讨论它. 首先, 然而, 是时间举个例子了. 下面是一个简单的(有点丑陋) read_proc 实现, 为 scull 设备:

```
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; /* Don't print more than this */

  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;

}
```

这是一个相当典型的 read_proc 实现. 它假定不会有必要产生超过一页数据并且因此忽略了 start 和 offset 值. 它是, 但是, 小心地不覆盖它的缓存, 只是以防万一.

4.3.1.2. 老接口

如果你阅览内核源码, 你会遇到使用老接口实现 /proc 的代码:

```
int (*get_info)(char *page, char **start, off_t offset, int count);
```

所有的参数的含义同 read_proc 的相同, 但是没有 eof 和 data 参数. 这个接口仍然支持, 但是将来会消失; 新代码应当使用 read_proc 接口来代替.

4.3.1.3. 创建你的 /proc 文件

一旦你有一个定义好的 read_proc 函数, 你应当连接它到 /proc 层次中的一个入口项. 使用一个 creat_proc_read_entry 调用:

```
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);
```

这里, name 是要创建的文件名子, mod 是文件的保护掩码(缺省系统范围时可以作为 0 传递), base 指出要创建的文件的目录(如果 base 是 NULL, 文件在 /proc 根下创建), read_proc 是实现文件的 read_proc 函数, data 被内核忽略(但是传递给 read_proc). 这就是 scull 使用的调用, 来使它的 /proc 函数可用做 /proc/scullmem:

```
create_proc_read_entry("scullmem", 0 /* default mode */,
                       NULL /* parent dir */, scull_read_procmem,
                       NULL /* client data */);
```

这里, 我们创建了一个名为 scullmem 的文件, 直接在 /proc 下, 带有缺省的, 全局可读的保护.

目录入口指针可用来在 /proc 下创建整个目录层次. 但是, 注意, 一个入口放在 /proc 的子目录下会更容易, 通过简单地给出目录名子作为这个入口名子的一部分 -- 只要这个目录自身已经存在. 例如, 一个(常常被忽略)传统的是 /proc 中与设备驱动相连的入口应当在 driver/ 子目录下; scull 能够安放它的入口在那里, 简单地通过指定它为名子 driver/scullmem.

/proc 中的入口, 当然, 应当在模块卸载后去除. remove_proc_entry 是恢复 create_proc_read_entry 所做的事情的函数:

```
remove_proc_entry("scullmem", NULL /* parent dir */);
```

去除入口失败会导致在不希望的时间调用, 或者, 如果你的模块已被卸载, 内核崩掉.

当如展示的使用 /proc 文件, 你必须记住几个实现的麻烦事 -- 不要奇怪现在不鼓励使用它.

最重要的问题是关于去除 /proc 入口. 这样的去除很可能在文件使用时发生, 因为没有所有者关联到 /proc 入口, 因此使用它们不会作用到模块的引用计数. 这个问题可以简单的触发, 例如通过运行 sleep 100 < /proc/myfile, 刚好在去除模块之前.

另外一个问题时关于用同样的名子注册两个入口. 内核信任驱动, 不会检查名子是否已经注册了, 因此如果你不小心, 你可能会使用同样的名子注册两个或多个入口. 这是一个已知发生在教室中的问题, 这样的入口是不能区分的, 不但在你存取它们时, 而且在你调用 remove_proc_entry 时.

4.3.1.4. seq_file 接口

如我们上面提到的, 在 /proc 下的大文件的实现有点麻烦. 一直以来, /proc 方法因为当输出数量变大时的错误实现变得声名狼藉. 作为一种清理 /proc 代码以及使内核开发者活得轻松些的方法, 添加了 seq_file 接口. 这个接口提供了简单的一套函数来实现大内核虚拟文件.

set_file 接口假定你在创建一个虚拟文件, 它涉及一系列的必须返回给用户空间的项. 为使用 seq_file, 你必须创建一个简单的 "iterator" 对象, 它能在序列里建立一个位置, 向前进, 并且输出序列里的一个项. 它可能听起来复杂, 但是, 实际上, 过程非常简单. 我们一步步来创建 /proc 文件在 scull 驱动里, 来展示它是如何做的.

第一步, 不可避免地, 是包含 <linux/seq_file.h>. 接着你必须创建 4 个 iterator 方法, 称为 start, next, stop, 和 show.

start 方法一直是首先调用. 这个函数的原型是:

```
void *start(struct seq_file *sfile, loff_t *pos);
```

sfile 参数可以几乎是一直被忽略. pos 是一个整型位置值, 指示应当从哪里读. 位置的解释完全取决于实现; 在结果文件里不需要是一个字节位置. 因为 seq_file 实现典型地步进一系列感兴趣的项, position 常常被解释为指向序列中下一个项的指针. scull 驱动解释每个设备作为系列中的一项, 因此进入的 pos 简单地是一个 scull_device 数组的索引. 因此, scull 使用的 start 方法是:

```
static void *scull_seq_start(struct seq_file *s, loff_t *pos)
{
  if (*pos >= scull_nr_devs)
    return NULL;  /* No more to read */
  return scull_devices + *pos;
}
```

返回值, 如果非NULL, 是一个可以被 iterator 实现使用的私有值.

next 函数应当移动 iterator 到下一个位置, 如果序列里什么都没有剩下就返回 NULL. 这个方法的原型是:

```
void *next(struct seq_file *sfile, void *v, loff_t *pos);
```

这里, v 是从前一个对 start 或者 next 的调用返回的 iterator, pos 是文件的当前位置. next 应当递增有 pos 指向的值; 根据你的 iterator 是如何工作的, 你可能(尽管可能不会)需要递增 pos 不止是 1. 这是 scull 所做的:

```
static void *scull_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
   (*pos)++;
   if (*pos >= scull_nr_devs)
      return NULL;
   return scull_devices + *pos;
}
```

当内核处理完 iterator, 它调用 stop 来清理:

```
void stop(struct seq_file *sfile, void *v);
```

scull 实现没有清理工作要做, 所以它的 stop 方法是空的.

设计上, 值得注意 seq_file 代码在调用 start 和 stop 之间不睡眠或者进行其他非原子性任务. 你也肯定会看到在调用 start 后马上有一个 stop 调用. 因此, 对你的 start 方法来说请求信号量或自旋锁是安全的. 只要你的其他 seq_file 方法是原子的, 调用的整个序列是原子的. (如果这一段对你没有意义, 在你读了下一章后再回到这.)

在这些调用中, 内核调用 show 方法来真正输出有用的东西给用户空间. 这个方法的原型是:

```
int show(struct seq_file *sfile, void *v);
```

这个方法应当创建序列中由 iterator v 指示的项的输出. 不应当使用 printk, 但是; 有一套特殊的用作 seq_file 输出的函数:

```
int seq_printf(struct seq_file *sfile, const char *fmt, ...);
```

这是给 seq_file 实现的 printf 对等体; 它采用常用的格式串和附加值参数. 你必须也将给 show 函数的 set_file 结构传递给它, 然而. 如果seq_printf 返回非零值, 意思是缓存区已填充, 输出被丢弃. 大部分实现忽略了返回值, 但是.

```
int seq_putc(struct seq_file *sfile, char c);
int seq_puts(struct seq_file *sfile, const char *s);
```

它们是用户空间 putc 和 puts 函数的对等体.

```
int seq_escape(struct seq_file *m, const char *s, const char *esc);
```

这个函数是 seq_puts 的对等体, 除了 s 中的任何也在 esc 中出现的字符以八进制格式打印. esc 的一个通用值是"\t\n\\", 它使内嵌的空格不会搞乱输出和可能搞乱 shell 脚本.

```
int seq_path(struct seq_file *sfile, struct vfsmount *m, struct dentry *dentry, char *esc);
```

这个函数能够用来输出和给定命令项关联的文件名子. 它在设备驱动中不可能有用; 我们是为了完整在此包含它.

回到我们的例子; 在 scull 使用的 show 方法是:

```
static int scull_seq_show(struct seq_file *s, void *v)
{
   struct scull_dev *dev = (struct scull_dev *) v;
   struct scull_qset *d;
   int i;

   if (down_interruptible (&dev->sem))
      return -ERESTARTSYS;

   seq_printf(s, "\nDevice %i: qset %i, q %i, sz %li\n",
            (int) (dev - scull_devices), dev->qset,
            dev->quantum, dev->size);

   for (d = dev->data; d; d = d->next) { /* scan the list */
      seq_printf(s, " item at %p, qset at %p\n", d, d->data);
      if (d->data && !d->next) /* dump only the last item */

         for (i = 0; i < dev->qset; i++) {
            if (d->data[i])
               seq_printf(s, " % 4i: %8p\n",
                        i, d->data[i]);
         }
   }
   up(&dev->sem);
   return 0;
}
```

这里, 我们最终解释我们的" iterator" 值, 简单地是一个 scull_dev 结构指针.

现在已有了一个完整的 iterator 操作的集合, scull 必须包装起它们, 并且连接它们到 /proc 中的一个文件. 第一步是填充一个 seq_operations 结构:

```
static struct seq_operations scull_seq_ops = {
 .start = scull_seq_start,
 .next = scull_seq_next,
 .stop = scull_seq_stop,
 .show = scull_seq_show
};
```

有那个结构在, 我们必须创建一个内核理解的文件实现. 我们不使用前面描述过的 read_proc 方法; 在使用 seq_file 时, 最好在一个稍低的级别上连接到 /proc. 那意味着创建一个 file_operations 结构(是的, 和字符驱动使用的同样结构) 来实现所有内核需要的操作, 来处理文件上的读和移动. 幸运的是, 这个任务是简单的. 第一步是创建一个 open 方法连接文件到 seq_file 操作:

```
static int scull_proc_open(struct inode *inode, struct file *file)
{
   return seq_open(file, &scull_seq_ops);
}
```

调用 seq_open 连接文件结构和我们上面定义的序列操作. 事实证明, open 是我们必须自己实现的唯一文件操作, 因此我们现在可以建立我们的 file_operations 结构:

```
static struct file_operations scull_proc_ops = {
 .owner = THIS_MODULE,
 .open = scull_proc_open,
 .read = seq_read,
 .llseek = seq_lseek,
 .release = seq_release
};
```

这里我们指定我们自己的 open 方法, 但是使用预装好的方法 seq_read, seq_lseek, 和 seq_release 给其他.

最后的步骤是创建 /proc 中的实际文件:

```
entry = create_proc_entry("scullseq", 0, NULL);
if (entry)
   entry->proc_fops = &scull_proc_ops;
```

不是使用 create_proc_read_entry, 我们调用低层的 create_proc_entry, 我们有这个原型:

```
struct proc_dir_entry *create_proc_entry(const char *name,mode_t mode,struct proc_dir_entry *parent);
```

参数和它们的在 create_proc_read_entry 中的对等体相同: 文件名子, 它的位置, 以及父目录.

有了上面代码, scull 有一个新的 /proc 入口, 看来很象前面的一个. 但是, 它是高级的, 因为它不管它的输出有多么大, 它正确处理移动, 并且通常它是易读和易维护的. 我们建议使用 seq_file , 来实现包含多个非常小数目的输出行数的文件.

4.3.2. ioctl 方法

ioctl, 我们在第 1 章展示给你如何使用, 是一个系统调用, 作用于一个文件描述符; 它接收一个确定要进行的命令的数字和(可选地)另一个参数, 常常是一个指针. 作为一个使用 /proc 文件系统的替代, 你可以实现几个用来调试用的 ioctl 命令. 这些命令可以从驱动拷贝相关的数据结构到用户空间, 这里你可以检查它们.

这种方式使用 ioctl 来获取信息有些比使用 /proc 困难, 因为你需要另一个程序来发出 ioctl 并且显示结果. 必须编写这个程序, 编译, 并且与你在测试的模块保持同步. 另一方面, 驱动侧代码可能容易过需要实现一个 /proc 文件的代码.

有时候 ioctl 是获取信息最好的方法, 因为它运行比读取 /proc 快. 如果在数据写到屏幕之前必须做一些事情, 获取二进制形式的数据比读取一个文本文件要更有效. 另外, ioctl 不要求划分数据为小于一页的片段.

ioctl 方法的另一个有趣的优点是信息获取命令可留在驱动中, 当调试被禁止时. 不象对任何查看目录的人(并且太多人可能奇怪"这个怪文件是什么")都可见的 /proc 文件, 不记入文档的 ioctl 命令可能保持不为人知. 另外, 如果驱动发生了怪异的事情, 它们仍将在那里. 唯一的缺点是模块可能会稍微大些.

---

[14] 连字号, 或者减号, 是一个"魔术"标识以阻止 syslogd 刷新文件到磁盘在每个新消息, 有关文档在 syslog.conf (5), 一个值得一读的 manpage.

上一页 4.2. 用打印调试 | 上一级 起始页 | 下一页 4.4. 使用观察来调试

4.4. 使用观察来调试

有时小问题可以通过观察用户空间的应用程序的行为来追踪. 监视程序也有助于建立对驱动正确工作的信心. 例如, 我们能够对 scull 感到有信心, 在看了它的读实现如何响应不同数量数据的读请求之后.

有几个方法来监视用户空间程序运行. 你可以运行一个调试器来单步过它的函数, 增加打印语句, 或者在 strace 下运行程序. 这里, 我们将讨论最后一个技术, 当真正目的是检查内核代码时它是最有趣的.

strace 命令时一个有力工具, 显示所有的用户空间程序发出的系统调用. 它不仅显示调用, 还以符号形式显示调用的参数和返回值. 当一个系统调用失败, 错误的符号值(例如, ENOMEM)和对应的字串(Out of memory) 都显示. strace 有很多命令行选项; 其中最有用的是 -t 来显示每个调用执行的时间, -T 来显示调用中花费的时间, -e 来限制被跟踪调用的类型, 以及-o 来重定向输出到一个文件. 缺省地, strace 打印调用信息到 stderr.

strace 从内核自身获取信息. 这意味着可以跟踪一个程序, 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 strip 过. 你也可以连接追踪到一个运行中的进程, 类似于一个调试器的方式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, 下面的屏幕输出显示(大部分)运行命令 strace ls /dev > /dev/scull0 的最后的行:

```
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. 奇怪地(对ls), 只有 4000 字节写入, 并且操作被重复. 但是, 我们知道 scull 中的写实现一次写一个单个量子, 因此我们本来就期望部分写. 几步之后, 所有东西清空, 程序成功退出.

作为另一个例子, 让我们读取 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) = ?
```

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. 注意在这个例子里读取是如何组织的, 同前面跟踪的相反. wc 为快速读被优化过, 因此绕过了标准库, 试图一个系统调用读取更多数据. 你可从跟踪的读的行里看到 wc 是如何试图一次读取 16 KB.

Linux 专家能够从 strace 的输出中发现更多有用信息. 如果你不想看到所有的符号, 你可使用 efile 标志来限制你自己仅查看文件方法是如何工作的.

就个人而言, 我们发现 strace 对于查明系统调用的运行时错误是非常有用. 常常是应用程序或演示程序中的 perror 调用不足够详细, 并且能够确切说出哪个系统调用的哪个参数触发了错误是非常有帮助的.

4.5. 调试系统故障

即便你已使用了所有的监视和调试技术, 有时故障还留在驱动里, 当驱动执行时系统出错. 当发生这个时, 能够收集尽可能多的信息来解决问题是重要的.

注意"故障"不意味着"崩溃". Linux 代码是足够健壮地优雅地响应大部分错误:一个故障常常导致当前进程的破坏而系统继续工作. 系统可能崩溃, 如果一个故障发生在一个进程的上下文之外, 或者如果系统的一些至关重要的部分毁坏了. 但是当是一个驱动错误导致的问题, 它常常只会导致不幸使用驱动的进程的突然死掉. 当进程被销毁时唯一无法恢复的破坏是分配给进程上下文的一些内存丢失了; 例如, 驱动通过 kmalloc 分配的动态列表可能丢失. 但是, 因为内核为任何一个打开的设备在进程死亡时调用关闭操作, 你的驱动可以释放由 open 方法分配的东西.

尽管一个 oops 常常都不会关闭整个系统, 你很有可能发现在发生一次后需要重启系统. 一个满是错误的驱动能使硬件处于不能使用的状态, 使内核资源处于不一致的状态, 或者, 最坏的情况, 在随机的地方破坏内核内存. 常常你可简单地卸载你的破驱动并且在一次 oops 后重试. 然而, 如果你看到任何东西建议说系统作为一个整体不太好了, 你最好立刻重启.

我们已经说过, 当内核代码出错, 一个提示性的消息打印在控制台上. 下一节解释如何解释并利用这样的消息. 尽管它们对新手看来相当模糊, 处理器转储是很有趣的信息, 常常足够来查明一个程序错误而不需要附加的测试.

4.5.1. oops 消息

大部分 bug 以解引用 NULL 指针或者使用其他不正确指针值来表现自己的. 此类 bug 通常的输出是一个 oops 消息.

处理器使用的任何地址几乎都是一个虚拟地址, 通过一个复杂的页表结构映射为物理地址(例外是内存管理子系统自己使用的物理地址). 当解引用一个无效的指针, 分页机制无法映射指针到一个物理地址, 处理器发出一个页错误给操作系统. 如果地址无效, 内核无法"页入"缺失的地址; 它(常常)产生一个 oops 如果在处理器处于管理模式时发生这个情况.

一个 oops 显示了出错时的处理器状态, 包括CPU 寄存器内容和其他看来不可理解的信息. 消息由错误处理的 printk 语句产生(arch/*/kernel/traps.c)并且如同前面 "printk" 一节中描述的被分派.

我们看一个这样的消息. 这是来自在运行 2.6 内核的 PC 上一个 NULL 指针导致的结果. 这里最相关的信息是指令指针(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
```

写入一个由坏模块拥有的设备而产生的消息, 一个故意用来演示失效的模块. faulty.c 的 write 方法的实现是琐细的:

```
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;
}
```

如你能见, 我们这里做的是解引用一个 NULL 指针. 因为 0 一直是一个无效的指针值, 一个错误发生, 由内核转变为前面展示的 oops 消息. 调用进程接着被杀掉.

错误模块有不同的错误情况在它的读实现中:

```
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;
}
```

这个方法拷贝一个字串到一个本地变量; 不幸的是, 字串长于目的数组. 当函数返回时导致的缓存区溢出引起一次 oops . 因为返回指令使指令指针到不知何处, 这类的错误很难跟踪, 并且你得到如下的:

```
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.
```

这个情况, 我们只看到部分的调用堆栈(vfs_read 和 faulty_read 丢失), 内核抱怨一个"坏 EIP 值". 这个抱怨和在开头列出的犯错的地址 (ffffffff) 都暗示内核堆栈已被破坏.

通常, 当你面对一个 oops, 第一件事是查看发生问题的位置, 常常与调用堆栈分开列出. 在上面展示的第一个 oops, 相关的行是:

```
EIP is at faulty_write+0x4/0x10 [faulty]
```

这里我们看到, 我们曾在函数 faulty_write, 它位于 faulty 模块(在方括号中列出的). 16 进制数指示指令指针是函数内 4 字节, 函数看来是 10 (16 进制)字节长. 常常这就足够来知道问题是什么.

如果你需要更多信息, 调用堆栈展示给你如何得知在哪里坏事的. 堆栈自己是 16 机制形式打印的; 做一点工作, 你经常可以从堆栈的列表中决定本地变量的值和函数参数. 有经验的内核开发者可以从这里的某些模式识别中获益; 例如, 如果你看来自 faulty_read oops 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
```

堆栈顶部的 ffffffff 是我们坏事的字串的一部分. 在 x86 体系, 缺省地, 用户空间堆栈开始于 0xc0000000; 因此, 循环值 0xbfffda70 可能是一个用户堆栈地址; 实际上, 它是传递给 read 系统调用的缓存地址, 每次下传过系统调用链时都被复制. 在 x86 (又一次, 缺省地), 内核空间开始于 0xc0000000, 因此这个之上的值几乎肯定是内核空间的地址, 等等.

最后, 当看一个 oops 列表, 一直监视本章开始讨论的"slab 毒害"值. 例如,如果你得到一个内核 oops, 里面的犯错地址时 0xa5a5a5a5, 你几乎肯定 - 某个地方在初始化动态内存.

请注意, 只在你的内核是打开 CONFIG_KALLSYMS 选项而编译时可以看到符号的调用堆栈. 否则, 你见到一个裸的, 16 机制列表, 除非你以别的方式对其解码, 它是远远无用的.

4.5.2. 系统挂起

尽管内核代码的大部分 bug 以 oops 消息结束, 有时候它们可能完全挂起系统. 如果系统挂起, 没有消息打印. 例如, 如果代码进入一个无限循环, 内核停止调度,[15] 并且系统不会响应任何动作, 包括魔术 Ctrl-Alt-Del 组合键. 你有 2 个选择来处理系统挂起-- 或者事先阻止它们, 或者能够事后调试它们.

你可阻止无限循环通过插入 schedule 引用在战略点上. schedule 调用(如你可能猜到的)调度器, 因此, 允许别的进程从当前进程偷取 CPU 数据. 如果一个进程由于你的驱动的bug而在内核空间循环, schedule 调用使你能够杀掉进程在跟踪发生了什么之后.

你应当知道, 当然, 如何对 schedule 的调用可能创造一个附加的重入调用源到你的驱动, 因为它允许别的进程运行. 这个重入正常地不应当是问题, 假定你在你的驱动中已经使用了合适的加锁. 然而, 要确认在你的驱动持有一个自旋锁的任何时间不能调用 schedule.

如果你的驱动真正挂起了系统, 并且你不知道在哪里插入 schedule 调用, 最好的方式是加入一些打印消息并且写到控制台(如果需要, 改变 console_loglevel 值).

有时候系统可能看来被挂起, 但是没有. 例如, 这可能发生在键盘以某个奇怪的方式保持锁住的时候. 这些假挂起可通过查看你为此目的运行的程序的输出来检测. 一个你的显示器上的时钟或者系统负载表是一个好的状态监控器; 只要他继续更新, 调度器就在工作.

对许多的上锁一个必不可少的工具是"魔术 sysrq 键", 在大部分体系上都可用. 魔键 sysrq 是 PC 键盘上 alt 和 sysrq 键组合来发出的, 或者在别的平台上使用其他特殊键(详见 documentation/sysrq.txt), 在串口控制台上也可用. 一个第三键, 与这 2 个一起按下, 进行许多有用的动作中的一个:

r 关闭键盘原始模式; 用在一个崩溃的应用程序(例如 X 服务器)可能将你的键盘搞成一个奇怪的状态.

k 调用"安全注意键"(SAK) 功能. SAK 杀掉在当前控制台的所有运行的进程, 给你一个干净的终端.

s 进行一个全部磁盘的紧急同步.

u umount. 试图重新加载所有磁盘在只读模式. 这个操作, 常常在 s 之后马上调用, 可以节省大量的文件系统检查时间, 在系统处于严重麻烦时.

b boot. 立刻重启系统. 确认先同步和重新加载磁盘.

p 打印处理器消息.

t 打印当前任务列表.

m 打印内存信息.

有别的魔术 sysrq 功能存在; 完整内容看内核源码的文档目录中的 sysrq.txt. 注意魔术 sysrq 必须在内核配置中显式使能, 大部分的发布没有使能它, 因为明显的安全理由. 对于用来开发驱动的系统, 然而, 使能魔术 sysrq 值得为它自己建立一个新内核的麻烦. 魔术 sysrq 可能在运行时关闭, 使用如下的一个命令:

```
echo 0 > /proc/sys/kernel/sysrq
```

如果非特权用户能够接触你的系统键盘, 你应当考虑关闭它, 来阻止有意或无意的损坏. 一些以前的内核版本缺省关闭 sysrq, 因此你需要在运行时使能它, 通过向同样的 /proc/sys 文件写入 1.

sysrq 操作是非常有用, 因此它们已经对不能接触到控制台的系统管理员可用. 文件 /proc/sysrq-trigger 是一个只写的入口点, 这里你可以触发一个特殊的 sysrq 动作, 通过写入关联的命令字符; 接着你可收集内核日志的任何输出数据. 这个 sysrq 的入口点是一直工作的, 即便 sysrq 在控制台上被关闭.

如果你经历一个"活挂", 就是你的驱动粘在一个循环中, 但是系统作为一个整体功能正常, 有几个技术值得了解. 经常地,

sysrq p 功能直接指向出错的函数. 如果这个不行, 你还可以使用内核剖析功能. 建立一个打开剖析的内核, 并且用命令行中 profile=2 来启动它. 使用 readprofile 工具复位剖析计数器, 接着使你的驱动进入它的循环. 一会儿后, 使用 readprofile 来看内核在哪里消耗它的时间. 另一个更高级的选择是 oprofile, 你可以也考虑下. 文件 documentation/basic_profiling.txt 告诉你启动剖析器所有需要知道的东西.

在追逐系统挂起时一个值得使用的防范措施是以只读方式加载你的磁盘(或者卸载它们). 如果磁盘是只读或者卸载的, 就没有风险损坏文件系统或者使它处于不一致的状态. 另外的可能性是使用一个通过 NFS, 网络文件系统, 来加载它的全部文件系统的计算机, 内核的"NFS-Root"功能必须打开, 在启动时必须传递特殊的参数. 在这个情况下, 即便不依靠 sysrq 你也会避免文件系统破坏, 因为文件系统的一致有 NFS 服务器来管理, 你的设备驱动不会关闭它.

---

[15] 实际上, 多处理器系统仍然在其他处理器上调度, 甚至一个单处理器的机器可能重新调度, 如果内核抢占被使能. 然而, 对于大部分的通常的情况(单处理器不使能抢占), 系统一起停止调度.

4.6. 调试器和相关工具

调试模块的最后手段是使用调试器来单步调试代码, 查看变量值和机器寄存器. 这个方法费时, 应当尽量避免. 但是, 通过调试器获得的代码的细粒度视角有时是很有价值的.

在内核上使用一个交互式调试器是一个挑战. 内核代表系统中的所有进程运行在自己的地址空间. 结果, 用户空间调试器所提供的一些普通功能, 例如断点和单步, 在内核中更难得到. 本节中, 我们看一下几个调试内核的方法; 每个都有缺点和优点.

4.6.1. 使用 gdb

gdb 对于看系统内部是非常有用. 在这个级别精通调试器的使用要求对 gdb 命令有信心, 需要理解目标平台的汇编代码, 以及对应源码和优化的汇编码的能力.

调试器必须把内核作为一个应用程序来调用. 除了指定内核映象的文件名之外, 你需要在命令行提供一个核心文件的名子. 对于一个运行的内核, 核心文件是内核核心映象, /proc/kcore. 一个典型的 gdb 调用看来如下:

gdb /usr/src/linux/vmlinux /proc/kcore

第一个参数是非压缩的 ELF 内核可执行文件的名子, 不是 zImage 或者 bzImage 或者给启动环境特别编译的任何东东.

gdb 命令行的第二个参数是核心文件的名子. 如同任何 /proc 中的文件, /proc/kcore 是在被读的时候产生的. 当 read 系统调用在 /proc 文件系统中执行时, 它映射到一个数据产生函数,而不是一个数据获取函数; 我们已经在本章"使用 /proc 文件系统"一节中利用了这个特点. kcore 用来代表内核"可执行文件", 以一个核心文件的形式; 它是一个巨大的文件, 因为他代表整个的内核地址空间, 对应于所有的物理内存. 从 gdb 中, 你可查看内核变量,通过发出标准 gdb 命令. 例如, p jiffies 打印时钟的从启动到当前时间的嘀哒数.

当你从gdb打印数据, 内核仍然在运行, 各种数据项在不同时间有不同的值; 然而, gdb 通过缓存已经读取的数据来优化对核心文件的存取. 如果你试图再次查看 jiffies 变量, 你会得到和以前相同的答案. 缓存值来避免额外的磁盘存取对传统核心文件是正确的做法, 但是在使用一个"动态"核心映象时就不方便. 解决方法是任何时候你需要刷新 gdb 缓存时发出命令 core-file /proc/kcore; 调试器准备好使用新的核心文件并且丢弃任何旧信息. 然而, 你不会一直需要发出 core-file 在读取一个新数据时; gdb 读取核心以多个几KB的块的方式, 并且只缓存它已经引用的块.

gdb 通常提供的不少功能在你使用内核时不可用. 例如, gdb 不能修改内核数据; 它希望在操作内存前在它自己的控制下运行一个被调试的程序. 也不可能设置断点或观察点, 或者单步过内核函数.

注意, 为了给 gdb 符号信息, 你必须设置 CONFIG_DEBUG_INFO 来编译你的内核. 结果是一个很大的内核映象在磁盘上, 但是, 没有这个信息, 深入内核变量几乎不可能.

有了调试信息, 你可以知道很多内核内部的事情. gdb 愉快地打印出结构, 跟随指针, 等等. 而有一个事情比较难, 然而, 是检查 modules. 因为模块不是传递给gdb 的 vmlinux 映象, 调试器对它们一无所知. 幸运的是, 作为 2.6.7 内核, 有可能教给 gdb 需要如何检查可加载模块.

Linux 可加载模块是 ELF 格式的可执行映象; 这样, 它们被分成几个节. 一个典型的模块可能包含一打或更多节, 但是有 3 个典型的与一次调试会话相关:

.text

这个节包含有模块的可执行代码. 调试器必须知道在哪里以便能够给出回溯或者设置断点. (这些操作都不相关, 当运行一个调试器在 /proc/kcore 上, 但是它们在使用 kgdb 时可能有用, 下面描述).

.bss
.data

这 2 个节持有模块的变量. 在编译时不初始化的任何变量在 .bss 中, 而那些要初始化的在 .data 里.

使 gdb 能够处理可加载模块需要通知调试器一个给定模块的节加载在哪里. 这个信息在 sysfs 中, 在 /sys/module 下. 例如, 在加载 scull 模块后, 目录 /sys/module/scull/sections 包含名子为 .text 的文件; 每个文件的内容是那个节的基地址.

我们现在该发出一个 gdb 命令来告诉它关于我们的模块. 我们需要的命令是 add-symble-flile; 这个命令使用模块目标文件名, .text 基地址作为参数, 以及一系列描述任何其他感兴趣的节安放在哪里的参数. 在深入位于 sysfs 的模块节数据后, 我们可以构建这样一个命令:

```
(gdb) add-symbol-file .../scull.ko 0xd0832000 \
-s .bss 0xd0837100 \
 -s .data 0xd0836be0
```

我们已经包含了一个小脚本在例子代码里(gdbline), 它为给定的模块可以创建这个命令.

我们现在使用 gdb 检查我们的可加载模块中的变量. 这是一个取自 scull 调试会话的快速例子:

```
(gdb) add-symbol-file scull.ko 0xd0832000 \
```

```
-s .bss 0xd0837100 \
 -s .data 0xd0836be0
add symbol table from file "scull.ko" at
 .text_addr = 0xd0832000
 .bss_addr = 0xd0837100
 .data_addr = 0xd0836be0
(y or n) y
Reading symbols from scull.ko...done.
(gdb) p scull_devices[0]
$1 = {data = 0xcfd66c50,
 quantum = 4000,
 qset = 1000,
 size = 20881,
 access_key = 0,
 ...}
```

这里我们看到第一个 scull 设备当前持有 20881 字节. 如果我们想, 我们可以跟随数据链, 或者查看其他任何感兴趣的模块中的东东.

这是另一个值得知道的有用技巧:

```
(gdb) print *(address)
```

这里, 填充 address 指向的一个 16 进制地址; 输出是对应那个地址的代码的文件和行号. 这个技术可能有用, 例如, 来找出一个函数指针真正指向哪里.

我们仍然不能进行典型的调试任务, 如设置断点或者修改数据; 为进行这些操作, 我们需要使用象 kdb(下面描述) 或者 kgdb (我们马上就到)这样的工具.

4.6.2. kdb 内核调试器

许多读者可能奇怪为什么内核没有建立更多高级的调试特性在里面.答案, 非常简单, 是 Linus 不相信交互式的调试器. 他担心它们会导致不好的修改, 这些修改给问题打了补丁而不是找到问题的真正原因. 因此, 没有内嵌的调试器.

其他内核开发者, 但是, 见到了交互式调试工具的一个临时使用. 一个这样的工具是 kdb 内嵌式内核调试器, 作为来自 oss.sgi.com 的一个非官方补丁. 要使用 kdb, 你必须获得这个补丁(确认获得一个匹配你的内核版本的版本), 应用它, 重建并重安装内核. 注意, 直到本书编写时, kdb 只在IA-32(x86)系统中运行(尽管一个给 IA-64 的版本在主线内核版本存在了一阵子, 在被去除之前.)

一旦你运行一个使能了kdb的内核, 有几个方法进入调试器. 在控制台上按下 Pause(或者 Break) 键启动调试器. kdb 在一个内核 oops 发生时或者命中一个断点时也启动, 在任何一种情况下, 你看到象

这样的一个消息:

```
Entering kdb (0xc0347b80) on processor 0 due to Keyboard Entry
[0]kdb>
```

注意, 在kdb运行时内核停止任何东西. 在你调用 kdb 的系统中不应当运行其他东西; 特别, 你不应当打开网络 -- 除非, 当然, 你在调试一个网络驱动. 一般地以单用户模式启动系统是一个好主意, 如果你将使用 kdb.

作为一个例子, 考虑一个快速 scull 调试会话. 假设驱动已经加载, 我们可以这样告诉 kdb 在 sucll_read 中设置一个断点:

```
[0]kdb> bp scull_read
Instruction(i) BP #0 at 0xcd087c5dc (scull_read)
 is enabled globally adjust 1
[0]kdb> go
```

bp 命令告诉 kdb 在下一次内核进入 scull_read 时停止. 你接着键入 go 来继续执行. 在将一些东西放入一个 scull 设备后, 我们可以试着通过在另一个终端的外壳下运行 cat 命令来读取它, 产生下面:

```
Instruction(i) breakpoint #0 at 0xd087c5dc (adjusted)
0xd087c5dc scull_read: int3

Entering kdb (current=0xcf09f890, pid 1575) on processor 0 due to
Breakpoint @ 0xd087c5dc
[0]kdb>
```

我们现在位于 scull_read 的开始. 为看到我们任何到那里的, 我们可以获得一个堆栈回溯:

```
[0]kdb> bt
 ESP EIP Function (args)
0xcdbddf74 0xd087c5dc [scull]scull_read
0xcdbddf78 0xc0150718 vfs_read+0xb8
0xcdbddfa4 0xc01509c2 sys_read+0x42
0xcdbddfc4 0xc0103fcf syscall_call+0x7
[0]kdb>
```

kdb 试图打印出调用回溯中每个函数的参数. 然而, 它被编译器的优化技巧搞糊涂了. 因此, 它无法打印 scull_read 的参数.

到时候查看一些数据了. mds 命令操作数据; 我们可以查询 schull_devices 指针的值, 使用这样一个命令:

```
[0]kdb> mds scull_devices 1
0xd0880de8 cf36ac00 ....
```

这里我们要求一个(4字节)字, 起始于 scull_devices 的位置; 答案告诉我们的设备数组在地址 0xd0880de8; 第一个设备结构自己在 0xcf36ac00. 为查看那个设备结构, 我们需要使用这个地址:

```
[0]kdb> mds cf36ac00
0xcf36ac00 ce137dbc ....
0xcf36ac04 00000fa0 ....
0xcf36ac08 000003e8 ....
0xcf36ac0c 0000009b ....
0xcf36ac10 00000000 ....
0xcf36ac14 00000001 ....
0xcf36ac18 00000000 ....
0xcf36ac1c 00000001 ....
```

这里的 8 行对应于 scull_dev 结构的开始部分. 因此, 我们看到第一个设备的内存位于 0xce137dbc, quantum 是 4000 (16进制 fa0), 量子集大小是 1000 (16进制 3e8), 当前有 155(16进制 9b) 字节存于设备中.

kdb 也可以改变数据. 假想我们要截短一些数据从设备中:

```
[0]kdb> mm cf26ac0c 0x50
0xcf26ac0c = 0x50
```

在设备上一个后续的 cat 会返回比之前少的数据.

kdb 有不少其他功能, 包括单步(指令, 不是 C 源码的一行), 在数据存取上设置断点, 反汇编代码, 步入链表, 存取寄存器数据, 还有更多. 在你应用了 kdb 补丁后, 一个完整的手册页集能够在你的源码树的 documentation/kdb 下发现.

4.6.3. kgdb 补丁

目前为止我们看到的 2 个交互式调试方法(使用 gdb 于 /proc/kcore 和 kdb) 都缺乏应用程序开发者已经熟悉的那种环境. 如果有一个真正的内核调试器支持改变变量, 断点等特色, 不是很好?

确实, 有这样一个解决方案. 在本书编写时, 2 个分开的补丁在流通中, 它允许 gdb, 具备完全功能, 针对内核运行. 这 2 个补丁都称为 kgdb. 它们通过分开运行测试内核的系统和运行调试器的系统来工作; 这 2 个系统典型地是通过一个串口线连接起来. 因此, 开发者可以在稳定地桌面系统上运行 gdb, 而操作一个运行在专门测试的盒子中的内核. 这种方式建立 gdb 开始需要一些时间, 但是很快会得到回报,当一个难问题出现时.

这些补丁目前处于健壮的状态, 在某些点上可能被合并, 因此我们避免说太多, 除了它们在哪里以及它们的基本特色. 鼓励感兴趣的读者去看这些的当前状态.

第一个 kgdb 补丁当前在 -mm 内核树里 -- 补丁进入 2.6 主线的集结场. 补丁的这个版本支持 x86, SuperH, ia64, x86_64, 和 32位 PPC 体系. 除了通过串口操作的常用模式, 这个版本的 kgdb 可以通过一个局域网通讯. 使能以太网模式并且使用 kgdboe参数指定发出调试命令的 IP 地址来启动内核. 在 Documentation/i386/kgdb 下的文档描述了如何建立.[16]

作为一个选择, 你可使用位于 http://kgdb.sf.net 的kgdb补丁. 这个调试器的版本不支持网络通讯模式(尽管据说在开发中), 但是它确实有内嵌的使用可加载模块的支持. 它支持 x86, x86_64, PowerPC, 和 S/390 体系.

4.6.4. 用户模式 Linux 移植

用户模式 Linux (UML) 是一个有趣的概念. 它被构建为一个分开的 Linux 内核移植, 有它自己的 arch/um 子目录. 它不在一个新的硬件类型上运行, 但是; 相反, 它运行在一个由 Linux 系统调用接口实现的虚拟机上. 如此, UML 使用 Linux 内核来运行, 作为一个Linux 系统上的独立的用户模式进程.

有一个作为用户进程运行的内核拷贝有几个优点. 因为它们运行在一个受限的虚拟的处理器上, 一个错误的内核不能破坏"真实的"系统. 可以在同一台盒子轻易的尝试不同的硬件和软件配置. 并且, 也许对内核开发者而言, 用户模式内核可容易地使用 gdb 和 其他调试器操作.

毕竟, 它只是一个进程. UML 显然有加快内核开发的潜力.

然而, UML 有个大的缺点,从驱动编写者的角度看: 用户模式内核无法存取主机系统的硬件. 因此, 虽然它对于调试大部分本书的例子驱动是有用的, UML 对于不得不处理真实硬件的驱动的调试还是没有用处.

看 http://user-mode-linux.sf.net/ 关于 UML 的更多信息.

4.6.5. Linux 追踪工具

Linux Trace Toolkit (LTT) 是一个内核补丁以及一套相关工具, 允许追踪内核中的事件. 这个追踪包括时间信息, 可以创建一个给定时间段内发生事情的合理的完整图像. 因此, 它不仅用来调试也可以追踪性能问题.

LTT, 同广泛的文档一起, 可以在 http://www.opersys.com/LTT 找到.

4.6.6. 动态探针

Dynamic Probes (DProbes) 是由 IBM 发行的(在 GPL 之下)为 IA-32 体系的 Linux 的调试工具. 它允许安放一个"探针"在几乎系统中任何地方, 用户空间和内核空间都可以. 探针由一些代码组成(有一个特殊的,面向堆栈的语言写成), 当控制命中给定的点时执行. 这个代码可以报告信息给用户空间, 改变寄存器, 或者做其他很多事情. DProbes 的有用特性是, 一旦这个能力建立到内核中, 探针可以在任何地方插入在一个运行中的系统中, 不用内核建立或者重启. DProbes 可以和 LTT 一起来插入一个新的跟踪事件在任意位置.

DProbes 工具可以从 IBM 的开放源码网站:http://oss.sof-ware.ibm.com 下载.

---

[16] 确实是忽略了指出, 你应当使你的网络适配卡建立在内核中, 然而, 否则调试器在启动时找不到它会关掉它自己.

第 5 章 并发和竞争情况

上一页 下一页

第 5 章 并发和竞争情况

目录

迄今, 我们未曾关心并发的问题 -- 就是说, 当系统试图一次做多件事时发生的情况. 然而, 并发的管理是操作系统编程的核心问题之一. 并发相关的错误是一些最易出现又最难发现的问题. 即便是专家级 Linux 内核程序员偶尔也会出现并发相关的错误.

早期的 Linux 内核, 较少有并发的源头. 内核不支持对称多处理器(SMP)系统, 并发执行的唯一原因是硬件中断服务. 那个方法提供了简单性, 但是在有越来越多处理器的系统上注重性能并且坚持系统要快速响应事件的世界中它不再可行了. 为响应现代硬件和应用程序的要求, Linux 内核已经发

展为很多事情在同时进行. 这个进步已经产生了很大的性能和可扩展性. 然而, 它也很大地使内核编程任务复杂化. 设备启动程序员现在必须从一开始就将并发作为他们设计的要素, 并且他们必须对内核提供的并发管理设施有很强的理解.

本章的目的是开始建立那种理解的过程. 为此目的, 我们介绍一些设施来立刻应用到第 3 章的 scull 驱动. 展示的其他设施暂时还不使用. 但是首先, 我们看一下我们的简单 scull 驱动可能哪里出问题并且如何避免这些潜在的问题.

5.1. scull 中的缺陷

让我们快速看一段 scull 内存管理代码. 在写逻辑的深处, scull 必须决定它请求的内存是否已经分配. 处理这个任务的代码是:

```
if (!dptr->data[s_pos]) {
   dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
   if (!dptr->data[s_pos])
      goto out;
}
```

假设有 2 个进程(我们会称它们为"A"和"B") 独立地试图写入同一个 schull 设备的相同偏移. 每个进程同时到达上面片段的第一行的 if 测试. 如果被测试的指针是 NULL, 每个进程都会决定分配内存, 并且每个都会复制结果指针给 dptr->datat[s_pos]. 因为 2 个进程都在赋值给同一个位置, 显然只有一个赋值可以成功.

当然, 发生的是第 2 个完成赋值的进程将"胜出". 如果进程 A 先赋值, 它的赋值将被进程 B 覆盖. 在此, scull 将完全忘记 A 分配的内存; 它只有指向 B 的内存的指针. A 所分配的指针, 因此, 将被丢掉并且不再返回给系统.

事情的这个顺序是一个竞争情况的演示. 竞争情况是对共享数据的无控制存取的结果. 当错误的存取模式发生了, 产生了不希望的东西. 对于这里讨论的竞争情况, 结果是内存泄漏. 这已经足够坏了, 但是竞争情况常常导致系统崩溃和数据损坏. 程序员可能被诱惑而忽视竞争情况为相当低可能性的事件. 但是, 在计算机世界, 百万分之一的事件会每隔几秒发生, 并且后果会是严重的.

很快我们将去掉 scull 的竞争情况, 但是首先我们需要对并发做一个更普遍的回顾.

5.2. 并发和它的管理

在现代 Linux 系统, 有非常多的并发源, 并且因此而来的可能竞争情况. 多个用户空间进程在运行, 它们可能以令人惊讶的方式组合存取你的代码. SMP 系统能够同时在不同处理器上执行你的代码. 内核代码是可抢占的; 你的驱动代码可能在任何时间失去处理器, 代替它的进程可能也在你的驱动中运行. 设备中断是能够导致你的代码并发执行的异步事件. 内核也提供各种延迟代码执行的机制, 例如 workqueue, tasklet, 以及定时器, 这些能够使你的代码在任何时间以一种与当前进程在做的事情无关的方式运行. 在现代的, 热插拔的世界中, 你的设备可能在你使用它们的时候轻易地消失.

避免竞争情况可能是一个令人害怕的工作. 在一个任何时候可能发生任何事的世界, 驱动程序员如何避免产生绝对的混乱? 事实证明, 大部分竞争情况可以避免, 通过一些想法, 内核并发控制原语, 以及几个基本原则的应用. 我们会先从原则开始, 接着进入如何使用它们的细节中

竞争情况来自对资源的共享存取的结果. 当 2 个执行的线路[17]有机会操作同一个数据结构(或者硬件资源), 混合的可能性就一直存在. 因此第一个经验法则是在你设计驱动时在任何可能的时候记住避免共享的资源. 如果没有并发存取, 就没有竞争情况. 因此小心编写的内核代码应当有最小的共享. 这个想法的最明显应用是避免使用全局变量. 如果你将一个资源放在多个执行线路能够找到它的地方, 应当有一个很强的理由这样做.

事实是, 然而, 这样的共享常常是需要的. 硬件资源是, 由于它们的特性, 共享的, 软件资源也必须常常共享给多个线程. 也要记住全局变量远远不是共享数据的唯一方式; 任何时候你的代码传递一个指针给内核的其他部分, 潜在地它创造了一个新的共享情形. 共享是生活的事实.

这是资源共享的硬规则: 任何时候一个硬件或软件资源被超出一个单个执行线程共享, 并且可能存在一个线程看到那个资源的不一致时, 你必须明确地管理对那个资源的存取. 在上面的 scull 例子, 这个情况在进程 B 看来是不一致的; 不知道进程 A 已经为(共享的) 设备分配了内存, 它做它自己的分配并且覆盖了 A 的工作. 在这个例子里, 我们必须控制对 scull 数据结构的存取. 我们需要安排, 这样代码或者看到内存已经分配了, 或者知道没有内存已经或者将要被其他人分配. 存取管理的常用技术是加锁或者互斥 -- 确保在任何时间只有一个执行线程可以操作一个共享资源. 本章剩下的大部分将专门介绍加锁.

然而, 首先, 我们必须简短考虑一下另一个重要规则. 当内核代码创建一个会被内核其他部分共享的对象时, 这个对象必须一直存在(并且功能正常)到它知道没有对它的外部引用存在为止. scull 使它的设备可用的瞬间, 它必须准备好处理对那些设备的请求. 并且 scull 必须一直能够处理对它的设备的请求直到它知道没有对这些设备的引用(例如打开的用户空间文件)存在. 2 个要求出自这个规则: 除非它处于可以正确工作的状态, 不能有对象能对内核可用, 对这样的对象的引用必须被跟踪. 在大部分情况下, 你将发现内核为你处理引用计数, 但是常常有例外.

遵照上面的规则需要计划和对细节小心注意. 容易被对资源的并发存取而吃惊, 你事先并没有认识到被共享. 通过一些努力, 然而, 大部分竞争情况能够在它们咬到你或者你的用户前被消灭.

---

[17] 本章的意图, 一个执行"线程"是任何运行代码的上下文. 每个进程显然是一个执行线程, 但是一个中断处理也是, 或者其他响应一个异步内核事件的代码.

上一页 上一级 下一页
第 5 章 并发和竞争情况 起始页 5.3. 旗标和互斥体

5.3. 旗标和互斥体

让我们看看我们如何给 scull 加锁. 我们的目标是使我们对 scull 数据结构的操作原子化, 就是在有其他执行线程的情况下这个操作一次发生. 对于我们的内存泄漏例子, 我们需要保证, 如果一个线程发现必须分配一个特殊的内存块, 它有机会进行这个分配在其他线程可做测试之前. 为此, 我们必须建立临界区: 在任何给定时间只有一个线程可以执行的代码.

不是所有的临界区是同样的, 因此内核提供了不同的原语适用不同的需求. 在这个例子中, 每个对 scull 数据结构的存取都发生在由一个直接用户请求所产生的进程上下文中; 没有从中断处理或者其他异步上下文中的存取. 没有特别的周期(响应时间)要求; 应用程序程序员理解 I/O 请求常常不是马上就满足的. 进一步讲, scull 没有持有任何其他关键系统资源, 在它存取它自己的数据结构时. 所有这些意味着如果 scull 驱动在等待轮到它存取数据结构时进入睡眠, 没人介意.

"去睡眠" 在这个上下文中是一个明确定义的术语. 当一个 Linux 进程到了一个它无法做进一步处理的地方时, 它去睡眠(或者 "阻塞"), 让出处理器给别人直到以后某个时间它能够再做事情. 进程常常在等待 I/O 完成时睡眠. 随着我们深入内核, 我们会遇到很多情况我们不能睡眠. 然而 scull 中的 write 方法不是其中一个情况. 因此我们可使用一个加锁机制使进程在等待存取临界区时睡眠.

正如重要地, 我们将进行一个可能会睡眠的操作(使用 kmalloc 分配内存) -- 因此睡眠是一个在任何情况下的可能性. 如果我们的临界区要正确工作, 我们必须使用一个加锁原语在一个拥有锁的进程睡眠时起作用. 不是所有的加锁机制都能够在可能睡眠的地方使用(我们在本章后面会看到几个不可以的). 然而, 对我们现在的需要, 最适合的机制时一个旗标.

旗标在计算机科学中是一个被很好理解的概念. 在它的核心, 一个旗标是一个单个整型值, 结合有一对函数, 典型地称为 P 和 V. 一个想进入临界区的进程将在相关旗标上调用 P; 如果旗标的值大于零, 这个值递减 1 并且进程继续. 相反, 如果旗标的值是 0 (或更小), 进程必须等待直到别人释放旗标. 解锁一个旗标通过调用 V 完成; 这个函数递增旗标的值, 并且, 如果需要, 唤醒等待的进程.

当旗标用作互斥 -- 阻止多个进程同时在同一个临界区内运行 -- 它们的值将初始化为 1. 这样的旗标在任何给定时间只能由一个单个进程或者线程持有. 以这种模式使用的旗标有时称为一个互斥锁, 就是, 当然, "互斥"的缩写. 几乎所有在 Linux 内核中发现的旗标都是用作互斥.

5.3.1. Linux 旗标实现

Linux 内核提供了一个遵守上面语义的旗标实现, 尽管术语有些不同. 为使用旗标, 内核代码必须包含 <asm/semaphore.h>. 相关的类型是 struct semaphore; 实际旗标可以用几种方法来声明和初始化. 一种是直接创建一个旗标, 接着使用 sema_init 来设定它:

```
void sema_init(struct semaphore *sem, int val);
```

这里 val 是安排给旗标的初始值.

然而, 通常旗标以互斥锁的模式使用. 为使这个通用的例子更容易些, 内核提供了一套帮助函数和宏定义. 因此, 一个互斥锁可以声明和初始化, 使用下面的一种:

```
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```

这里, 结果是一个旗标变量(称为 name), 初始化为 1 (使用 DECLARE_MUTEX) 或者 0 (使用 DECLARE_MUTEX_LOCKED). 在后一种情况, 互斥锁开始于上锁的状态; 在允许任何线程存取之前将不得不显式解锁它.

如果互斥锁必须在运行时间初始化(这是如果动态分配它的情况, 举例来说), 使用下列中的一个:

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

在 Linux 世界中, P 函数称为 down -- 或者这个名子的某个变体. 这里, "down" 指的是这样的事实, 这个函数递减旗标的值, 并且, 也许在使调用者睡眠一会儿来等待旗标变可用之后, 给予对被保护资源的存取. 有 3 个版本的 down:

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
```

down 递减旗标值并且等待需要的时间. down_interruptible 同样, 但是操作是可中断的. 这个可中断的版本几乎一直是你要的那个; 它允许一个在等待一个旗标的用户空间进程被用户中断. 作为一个通用的规则, 你不想使用不可中断的操作, 除非实在是没有选择. 不可中断操作是一个创建不可杀死的进程(在 ps 中见到的可怕的 "D 状态")和惹恼你的用户的好方法, 使用 down_interruptible 需要一些格外的小心, 但是, 如果操作是可中断的, 函数返回一个非零值, 并且调用者不持有旗标. 正确的使用 down_interruptible 需要一直检查返回值并且针对性地响应.

最后的版本 (down_trylock) 从不睡眠; 如果旗标在调用时不可用, down_trylock 立刻返回一个非零值.

一旦一个线程已经成功调用 down 各个版本中的一个, 就说它持有着旗标(或者已经"取得"或者"获得"旗标). 这个线程现在有权力存取这个旗标保护的临界区. 当这个需要互斥的操作完成时, 旗标必须被返回. V 的 Linux 对应物是 up:

```
void up(struct semaphore *sem);
```

一旦 up 被调用, 调用者就不再拥有旗标.

如你所愿, 要求获取一个旗标的任何线程, 使用一个(且只能一个)对 up 的调用释放它. 在错误路径中常常需要特别的小心; 如果在持有一个旗标时遇到一个错误, 旗标必须在返回错误状态给调用者之前释放旗标. 没有释放旗标是容易犯的一个错误; 这个结果(进程挂在看来无关的地方)可能是难于重现和跟踪的.

5.3.2. 在 scull 中使用旗标

旗标机制给予 scull 一个工具, 可以在存取 scull_dev 数据结构时用来避免竞争情况. 但是正确使用这个工具是我们的责任. 正确使用加锁原语的关键是严密地指定要保护哪个资源并且确认每个对这些资源的存取都使用了正确的加锁方法. 在我们的例子驱动中, 感兴趣的所有东西都包含在 scull_dev 结构里面, 因此它是我们的加锁体制的逻辑范围.

让我们在看看这个结构:

```
struct scull_dev {
   struct scull_qset *data; /* Pointer to first quantum set */
   int quantum; /* the current quantum size */
   int qset; /* the current array size */
   unsigned long size; /* amount of data stored here */
   unsigned int access_key; /* used by sculluid and scullpriv */
   struct semaphore sem; /* mutual exclusion semaphore */
   struct cdev cdev; /* Char device structure */
};
```

到结构的底部是一个称为 sem 的成员, 当然, 它是我们的旗标. 我们已经选择为每个虚拟 scull 设备使用单独的旗标. 使用一个单个的全局的旗标也可能会是同样正确. 通常各种 scull 设备不共享资源, 然而, 并且没有理由使一个进程等待, 而另一个进程在使用不同 scull 设备. 不同设备使用单独的旗标允许并行进行对不同设备的操作, 因此, 提高了性能.

旗标在使用前必须初始化. scull 在加载时进行这个初始化, 在这个循环中:

```
for (i = 0; i < scull_nr_devs; i++) {
   scull_devices[i].quantum = scull_quantum;
   scull_devices[i].qset = scull_qset;
   init_MUTEX(&scull_devices[i].sem);
   scull_setup_cdev(&scull_devices[i], i);
}
```

注意, 旗标必须在 scull 设备对系统其他部分可用前初始化. 因此, init_MUTEX 在 scull_setup_cdev 前被调用. 以相反的次序进行这个操作可能产生一个竞争情况, 旗标可能在它准备好之前被存取.

下一步, 我们必须浏览代码, 并且确认在没有持有旗标时没有对 scull_dev 数据结构的存取. 因此, 例如, scull_write 以这个代码开始:

```
if (down_interruptible(&dev->sem))
   return -ERESTARTSYS;
```

注意对 down_interruptible 返回值的检查; 如果它返回非零, 操作被打断了. 在这个情况下通常要做的是返回 -ERESTARTSYS. 看到这个返回值后, 内核的高层要么从头重启这个调用要么返回这个错误给用户. 如果你返回 -ERESTARTSYS, 你必须首先恢复任何用户可见的已经做了的改变, 以保证当重试系统调用时正确的事情发生. 如果你不能以这个方式恢复, 你应当替之返回 -EINTR.

scull_write 必须释放旗标, 不管它是否能够成功进行它的其他任务. 如果事事都顺利, 执行落到这个函数的最后几行:

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

这个代码释放旗标并且返回任何需要的状态. 在 scull_write 中有几个地方可能会出错; 这些地方包括内存分配失败或者在试图从用户空间拷贝数据时出错. 在这些情况中, 代码进行了一个 goto out, 以确保进行正确的清理.

5.3.3. 读者/写者旗标

旗标为所有调用者进行互斥, 不管每个线程可能想做什么. 然而, 很多任务分为 2 种清楚的类型: 只需要读取被保护的数据结构的类型, 和必须做改变的类型. 允许多个并发读者常常是可能的, 只要没有人试图做任何改变. 这样做能够显著提高性能; 只读的任务可以并行进行它们的工作而不必等待其他读者退出临界区.

Linux 内核为这种情况提供一个特殊的旗标类型称为 rwsem (或者" reader/writer semaphore"). rwsem 在驱动中的使用相对较少, 但是有时它们有用.

使用 rwsem 的代码必须包含 <linux/rwsem.h>. 读者写者旗标 的相关数据类型是 struct rw_semaphore; 一个 rwsem 必须在运行时显式初始化:

```
void init_rwsem(struct rw_semaphore *sem);
```

一个新初始化的 rwsem 对出现的下一个任务(读者或者写者)是可用的. 对需要只读存取的代码的

接口是:

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
```

对 down_read 的调用提供了对被保护资源的只读存取, 与其他读者可能地并发地存取. 注意 down_read 可能将调用进程置为不可中断的睡眠. down_read_trylock 如果读存取是不可用时不会等待; 如果被准予存取它返回非零, 否则是 0. 注意 down_read_trylock 的惯例不同于大部分的内核函数, 返回值 0 指示成功. 一个使用 down_read 获取的 rwsem 必须最终使用 up_read 释放.

读者的接口类似:

```
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);
```

down_write, down_write_trylock, 和 up_write 全部就像它们的读者对应部分, 除了, 当然, 它们提供写存取. 如果你处于这样的情况, 需要一个写者锁来做一个快速改变, 接着一个长时间的只读存取, 你可以使用 downgrade_write 在一旦你已完成改变后允许其他读者进入.

一个 rwsem 允许一个读者或者不限数目的读者来持有旗标. 写者有优先权; 当一个写者试图进入临界区, 就不会允许读者进入直到所有的写者完成了它们的工作. 这个实现可能导致读者饥饿 -- 读者被长时间拒绝存取 -- 如果你有大量的写者来竞争旗标. 由于这个原因, rwsem 最好用在很少请求写的时候, 并且写者只占用短时间.

5.4. Completions 机制

内核编程的一个普通模式包括在当前线程之外初始化某个动作, 接着等待这个动作结束. 这个动作可能是创建一个新内核线程或者用户空间进程, 对一个存在着的进程的请求, 或者一些基于硬件的动作. 在这些情况中, 很有诱惑去使用一个旗标来同步 2 个任务, 使用这样的代码:

```
struct semaphore sem;
init_MUTEX_LOCKED(&sem);
start_external_task(&sem);
down(&sem);
```

外部任务可以接着调用 up(??sem), 在它的工作完成时.

事实证明, 这种情况旗标不是最好的工具. 正常使用中, 试图加锁一个旗标的代码发现旗标几乎在所有时间都可用; 如果对旗标有很多竞争, 性能会受损并且加锁方案需要重新审视. 因此旗标已经对"可用"情况做了很多的优化. 当用上面展示的方法来通知任务完成, 然而, 调用 down 的线程将几乎是一直不得不等待; 因此性能将受损. 旗标还可能易于处于一个(困难的) 竞争情况, 如果它们表明为自动变量以这种方式使用时. 在一些情况中, 旗标可能在调用 up 的进程用完它之前消失.

这些问题引起了在 2.4.7 内核中增加了 "completion" 接口. completion 是任务使用的一个轻量级机制: 允许一个线程告诉另一个线程工作已经完成. 为使用 completion, 你的代码必须包含 <linux/completion.h>. 一个 completion 可被创建, 使用:

```
DECLARE_COMPLETION(my_completion);
```

或者, 如果 completion 必须动态创建和初始化:

```
struct completion my_completion;
/* ... */
init_completion(&my_completion);
```

等待 completion 是一个简单事来调用:

```
void wait_for_completion(struct completion *c);
```

注意这个函数进行一个不可打断的等待. 如果你的代码调用 wait_for_completion 并且没有人完成这

个任务, 结果会是一个不可杀死的进程.[18]

另一方面, 真正的 completion 事件可能通过调用下列之一来发出:

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

如果多于一个线程在等待同一个 completion 事件, 这 2 个函数做法不同. complete 只唤醒一个等待的线程, 而 complete_all 允许它们所有都继续. 在大部分情况下, 只有一个等待者, 这 2 个函数将产生一致的结果.

一个 completion 正常地是一个单发设备; 使用一次就放弃. 然而, 如果采取正确的措施重新使用 completion 结构是可能的. 如果没有使用 complete_all, 重新使用一个 completion 结构没有任何问题, 只要对于发出什么事件没有模糊. 如果你使用 complete_all, 然而, 你必须在重新使用前重新初始化 completion 结构. 宏定义:

```
INIT_COMPLETION(struct completion c);
```

可用来快速进行这个初始化.

作为如何使用 completion 的一个例子, 考虑 complete 模块, 它包含在例子源码里. 这个模块使用简单的语义定义一个设备: 任何试图从一个设备读的进程将等待(使用 wait_for_completion)直到其他进程向这个设备写. 实现这个行为的代码是:

```
DECLARE_COMPLETION(comp);
ssize_t complete_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_for_completion(&comp);
   printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
   return 0; /* EOF */
}

ssize_t complete_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);
   complete(&comp);
   return count; /* succeed, to avoid retrial */
}
```

有多个进程同时从这个设备"读"是有可能的. 每个对设备的写将确切地使一个读操作完成, 但是没有办法知道会是哪个.

completion 机制的典型使用是在模块退出时与内核线程的终止一起. 在这个原型例子里, 一些驱动的内部工作是通过一个内核线程在一个 while(1) 循环中进行的. 当模块准备好被清理时, exit 函数告知线程退出并且等待结束. 为此目的, 内核包含一个特殊的函数给线程使用:

void complete_and_exit(struct completion *c, long retval);

---

[18] 在本书编写时, 添加可中断版本的补丁已经流行但是还没有合并到主线中.

上一页 5.3. 旗标和互斥体
上一级 起始页
下一页 5.5. 自旋锁

5.5. 自旋锁

对于互斥, 旗标是一个有用的工具, 但是它们不是内核提供的唯一这样的工具. 相反, 大部分加锁是由一种称为自旋锁的机制来实现. 不象旗标, 自旋锁可用在不能睡眠的代码中, 例如中断处理. 当正确地使用了, 通常自旋锁提供了比旗标更高的性能. 然而, 它们确实带来对它们用法的一套不同的限制.

自旋锁概念上简单. 一个自旋锁是一个互斥设备, 只能有 2 个值:"上锁"和"解锁". 它常常实现为一个整数值中的一个单个位. 想获取一个特殊锁的代码测试相关的位. 如果锁是可用的, 这个"上锁"位被置位并且代码继续进入临界区. 相反, 如果这个锁已经被别人获得, 代码进入一个紧凑的循环中反复检查这个锁, 直到它变为可用. 这个循环就是自旋锁的"自旋"部分.

当然, 一个自旋锁的真实实现比上面描述的复杂一点. 这个"测试并置位"操作必须以原子方式进行, 以便只有一个线程能够获得锁, 就算如果有多个进程在任何给定时间自旋. 必须小心以避免在超线程处理器上死锁 -- 实现多个虚拟 CPU 以共享一个单个处理器核心和缓存的芯片. 因此实际的自旋锁实现在每个 Linux 支持的体系上都不同. 核心的概念在所有系统上相同, 然而, 当有对自旋锁的竞争, 等待的处理器在一个紧凑循环中执行并且不作有用的工作.

它们的特性上, 自旋锁是打算用在多处理器系统上, 尽管一个运行一个抢占式内核的单处理器工作站的行为如同 SMP, 如果只考虑到并发. 如果一个非抢占的单处理器系统进入一个锁上的自旋, 它将永远自旋; 没有其他的线程再能够获得 CPU 来释放这个锁. 因此, 自旋锁在没有打开抢占的单处理器系统上的操作被优化为什么不作, 除了改变 IRQ 屏蔽状态的那些. 由于抢占, 甚至如果你从不希望你的代码在一个 SMP 系统上运行, 你仍然需要实现正确的加锁.

5.5.1. 自旋锁 API 简介

自旋锁原语要求的包含文件是 <linux/spinlock.h>. 一个实际的锁有类型 spinlock_t. 象任何其他数据结构, 一个 自旋锁必须初始化. 这个初始化可以在编译时完成, 如下:

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

或者在运行时使用:

```
void spin_lock_init(spinlock_t *lock);
```

在进入一个临界区前, 你的代码必须获得需要的 lock , 用:

void spin_lock(spinlock_t *lock);

注意所有的自旋锁等待是, 由于它们的特性, 不可中断的. 一旦你调用 spin_lock, 你将自旋直到锁变为可用.

为释放一个你已获得的锁, 传递它给:

void spin_unlock(spinlock_t *lock);

有很多其他的自旋锁函数, 我们将很快都看到. 但是没有一个背离上面列出的函数所展示的核心概念. 除了加锁和释放, 没有什么可对一个锁所作的. 但是, 有几个规则关于你必须如何使用自旋锁. 我们将用一点时间来看这些, 在进入完整的自旋锁接口之前.

5.5.2. 自旋锁和原子上下文

想象一会儿你的驱动请求一个自旋锁并且在它的临界区里做它的事情. 在中间某处, 你的驱动失去了处理器. 或许它已调用了一个函数(copy_from_user, 假设) 使进程进入睡眠. 或者, 也许, 内核抢占发威, 一个更高优先级的进程将你的代码推到一边. 你的代码现在持有一个锁, 在可见的将来的如何时间不会释放这个锁. 如果某个别的线程想获得同一个锁, 它会, 在最好的情况下, 等待(在处理器中自旋)很长时间. 最坏的情况, 系统可能完全死锁.

大部分读者会同意这个场景最好是避免. 因此, 应用到自旋锁的核心规则是任何代码必须, 在持有自旋锁时, 是原子性的. 它不能睡眠; 事实上, 它不能因为任何原因放弃处理器, 除了服务中断(并且有时即便此时也不行)

内核抢占的情况由自旋锁代码自己处理. 内核代码持有一个自旋锁的任何时间, 抢占在相关处理器上被禁止. 即便单处理器系统必须以这种方式禁止抢占以避免竞争情况. 这就是为什么需要正确的加锁, 即便你从不期望你的代码在多处理器机器上运行.

在持有一个锁时避免睡眠是更加困难; 很多内核函数可能睡眠, 并且这个行为不是都被明确记录了. 拷贝数据到或从用户空间是一个明显的例子: 请求的用户空间页可能需要在拷贝进行前从磁盘上换入, 这个操作显然需要一个睡眠. 必须分配内存的任何操作都可能睡眠. kmalloc 能够决定放弃处理器, 并且等待更多内存可用除非它被明确告知不这样做. 睡眠可能发生在令人惊讶的地方; 编写会在自旋锁下执行的代码需要注意你调用的每个函数.

这有另一个场景: 你的驱动在执行并且已经获取了一个锁来控制对它的设备的存取. 当持有这个锁时, 设备发出一个中断, 使得你的中断处理运行. 中断处理, 在存取设备之前, 必须获得锁. 在一个中断处理中获取一个自旋锁是一个要做的合法的事情; 这是自旋锁操作不能睡眠的其中一个理由. 但是如果中断处理和起初获得锁的代码在同一个处理器上会发生什么? 当中断处理在自旋, 非中断代码不能运行来释放锁. 这个处理器将永远自旋.

避免这个陷阱需要在持有自旋锁时禁止中断(只在本地 CPU). 有各种自旋锁函数会为你禁止中断 (我们将在下一节见到它们). 但是, 一个完整的中断讨论必须等到第 10 章了.

关于自旋锁使用的最后一个重要规则是自旋锁必须一直是尽可能短时间的持有. 你持有一个锁越长, 另一个进程可能不得不自旋等待你释放它的时间越长, 它不得不完全自旋的机会越大. 长时间持有锁也阻止了当前处理器调度, 意味着高优先级进程 -- 真正应当能获得 CPU 的 -- 可能不得不等待. 内核开发者尽了很大努力来减少内核反应时间(一个进程可能不得不等待调度的时间)在 2.5 开发系列. 一个写的很差的驱动会摧毁所有的进程, 仅仅通过持有一个锁太长时间. 为避免产生这类问题, 重视使你的锁持有时间短.

5.5.3. 自旋锁函数

我们已经看到 2 个函数, spin_lock 和 spin_unlock, 可以操作自旋锁. 有其他几个函数, 然而, 有类似的名子和用途. 我们现在会展示全套. 这个讨论将带我们到一个我们无法在几章内适当涵盖的地方; 自旋锁 API 的完整理解需要对中断处理和相关概念的理解.

实际上有 4 个函数可以加锁一个自旋锁:

```
void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock)
```

我们已经看到自旋锁如何工作. spin_loc_irqsave 禁止中断(只在本地处理器)在获得自旋锁之前; 之前的中断状态保存在 flags 里. 如果你绝对确定在你的处理器上没有禁止中断的(或者, 换句话说, 你确信你应当在你释放你的自旋锁时打开中断), 你可以使用 spin_lock_irq 代替, 并且不必保持跟踪 flags. 最后, spin_lock_bh 在获取锁之前禁止软件中断, 但是硬件中断留作打开的.

如果你有一个可能被在(硬件或软件)中断上下文运行的代码获得的自旋锁, 你必须使用一种 spin_lock 形式来禁止中断. 其他做法可能死锁系统, 迟早. 如果你不在硬件中断处理里存取你的锁, 但是你通过软件中断(例如, 在一个 tasklet 运行的代码, 在第 7 章涉及的主题), 你可以使用 spin_lock_bh 来安全地避免死锁, 而仍然允许硬件中断被服务.

也有 4 个方法来释放一个自旋锁; 你用的那个必须对应你用来获取锁的函数.

```
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

每个 spin_unlock 变体恢复由对应的 spin_lock 函数锁做的工作. 传递给 spin_unlock_irqrestore 的 flags 参数必须是传递给 spin_lock_irqsave 的同一个变量. 你必须也调用 spin_lock_irqsave 和

spin_unlock_irqrestore 在同一个函数里. 否则, 你的代码可能破坏某些体系.

还有一套非阻塞的自旋锁操作:

```
int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);
```

这些函数成功时返回非零(获得了锁), 否则 0. 没有"try"版本来禁止中断.

5.5.4. 读者/写者自旋锁

内核提供了一个自旋锁的读者/写者形式, 直接模仿我们在本章前面见到的读者/写者旗标. 这些锁允许任何数目的读者同时进入临界区, 但是写者必须是排他的存取. 读者写者锁有一个类型 rwlock_t, 在 <linux/spinlokc.h> 中定义. 它们可以以 2 种方式被声明和被初始化:

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* Static way */
rwlock_t my_rwlock;
rwlock_init(&my_rwlock); /* Dynamic way */
```

可用函数的列表现在应当看来相当类似. 对于读者, 下列函数是可用的:

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);

void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

有趣地, 没有 read_trylock. 对于写存取的函数是类似的:

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);

void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
```

void write_unlock_bh(rwlock_t *lock);

读者/写者锁能够饿坏读者, 就像 rwsem 一样. 这个行为很少是一个问题; 然而, 如果有足够的锁竞争来引起饥饿, 性能无论如何都不行.

上一页 5.4. Completions 机制
上一级 起始页
下一页 5.6. 锁陷阱

5.6. 锁陷阱

多年使用锁的经验 -- 早于 Linux 的经验 -- 已经表明加锁可能是非常难于正确的. 管理并发是一个固有的技巧性的事情, 有很多出错的方式. 在这一节, 我们快速看一下可能出错的东西.

5.6.1. 模糊的规则

如同上面已经说过的, 一个正确的加锁机制需要清晰和明确的规则. 当你创建一个可以被并发存取的资源时, 你应当定义哪个锁将控制存取. 加锁应当真正在开始处进行; 事后更改会是难的事情. 开始时花费的时间常常在调试时获得回报.

当你编写你的代码, 你会毫无疑问遇到几个函数需要存取通过一个特定锁保护的结构. 在此, 你必须小心: 如果一个函数需要一个锁并且接着调用另一个函数也试图请求这个锁, 你的代码死锁. 不论旗标还是自旋锁都不允许一个持锁者第 2 次请求锁; 如果你试图这样做, 事情就简单地完了.

为使的加锁正确工作, 你不得不编写一些函数, 假定它们的调用者已经获取了相关的锁. 常常地, 只有你的内部的, 静态函数能够这样编写; 从外部调用的函数必须明确处理加锁. 当你编写内部函数对加锁做了假设, 方便自己(和其他使用你的代码的人)并且明确记录这些假设. 在几个月后可能很难回来并记起是否你需要持有一个锁来调用一个特殊函数.

在 sucll 的例子里, 采用的设计决定是要求所有的函数直接从系统调用里调用, 来请求应用到被存取的设备结构上的旗标. 所有的内部函数, 那些只是从其他 scull 函数里调用的, 可以因此假设旗标已经正确获得.

5.6.2. 加锁顺序规则

在有大量锁的系统中(并且内核在成为这样一个系统), 一次需要持有多于一个锁, 对代码是不寻常的. 如果某类计算必须使用 2 个不同的资源进行, 每个有它自己的锁, 常常没有选择只能获取 2 个锁.

获得多个锁可能是危险的, 然而. 如果你有 2 个锁, 称为 Lock1 和 Lock2, 代码需要同时都获取, 你有一个潜在的死锁. 仅仅想象一个线程锁住 Lock1 而另一个同时获得 Lock2. 接着每个线程试图得到它没有的那个. 2 个线程都会死锁.

这个问题的解决方法常常是简单的: 当多个锁必须获得时, 它们应当一直以同样顺序获得. 只要遵照这个惯例, 象上面描述的简单死锁能够避免. 然而, 遵照加锁顺序规则是做比说难. 非常少见这样

的规则真正在任何地方被写下. 常常你能做的最好的是看看别的代码如何做的.

一些经验规则能帮上忙. 如果你必须获得一个对你的代码来说的本地锁(假如, 一个设备锁), 以及一个属于内核更中心部分的锁, 先获取你的. 如果你有一个旗标和自旋锁的组合, 你必须, 当然, 先获得旗标; 调用 down (可能睡眠) 在持有一个自旋锁时是一个严重的错误. 但是最重要的, 尽力避免需要多于一个锁的情况.

5.6.3. 细 -粗- 粒度加锁

第一个支持多处理器系统的 Linux 内核是 2.0; 它只含有一个自旋锁. 这个大内核锁将整个内核变为一个大的临界区; 在任何时候只有一个 CPU 能够执行内核代码. 这个锁足够好地解决了并发问题以允许内核开发者从事所有其他的开发 SMP 所包含的问题. 但是它不是扩充地很好. 甚至一个 2 个处理器的系统可能花费可观数量的时间只是等待这个大内核锁. 一个 4 个处理器的系统的性能甚至不接近 4 个独立的机器的性能.

因此, 后续的内核发布已经包含了更细粒度的加锁. 在 2.2 中, 一个自旋锁控制对块 I/O 子系统的存取; 另一个为网络而工作, 等等. 一个现代的内核能包含几千个锁, 每个保护一个小的资源. 这种细粒度的加锁可能对伸缩性是好的; 它允许每个处理器在它自己特定的任务上工作而不必竞争其他处理器使用的锁. 很少人忘记大内核锁.[19]

但是, 细粒度加锁带有开销. 在有几千个锁的内核中, 很难知道你需要那个锁 -- 以及你应当以什么顺序获取它们 -- 来进行一个特定的操作. 记住加锁错误可能非常难发现; 更多的锁提供了更多的机会使真正有害的加锁 bug 钻进内核中. 细粒度加锁能带来一定水平的复杂性, 长期来, 对内核的可维护性有一个大的, 不利的效果.

在一个设备驱动中加锁常常是相对直接的; 你可以用一个锁来涵盖你做的所有东西, 或者你可以给你管理的每个设备创建一个锁. 作为一个通用的规则, 你应当从相对粗的加锁开始, 除非你有确实的理由相信竞争可能是一个问题. 忍住怂恿去过早地优化; 真实地性能约束常常表现在想不到的地方.

如果你确实怀疑锁竞争在损坏性能, 你可能发现 lockmeter 工具有用. 这个补丁(从 http://oss.sgi.com/projects/lockmeter/ 可得到) 装备内核来测量在锁等待花费的时间. 通过看这个报告, 你能够很快知道是否锁竞争真的是问题.

---

[19] 这个锁仍然存在于 2.6, 几个它现在覆盖内核非常小的部分. 如果你偶然发现一个 lock_kernel 调用, 你已找到了这个大内核锁. 但是, 想都不要想在任何新代码中使用它.

上一页 上一级 下一页
5.5. 自旋锁 起始页 5.7. 加锁的各种选择

5.7. 加锁的各种选择

Linux 内核提供了不少有力的加锁原语能够用来使内核避免被自己绊倒. 但是, 如同我们已见到的, 一个加锁机制的设计和实现不是没有缺陷. 常常对于旗标和自旋锁没有选择; 它们可能是唯一的方法来正确地完成工作. 然而, 有些情况, 可以建立原子的存取而不用完整的加锁. 本节看一下做事情的其他方法.

5.7.1. 不加锁算法

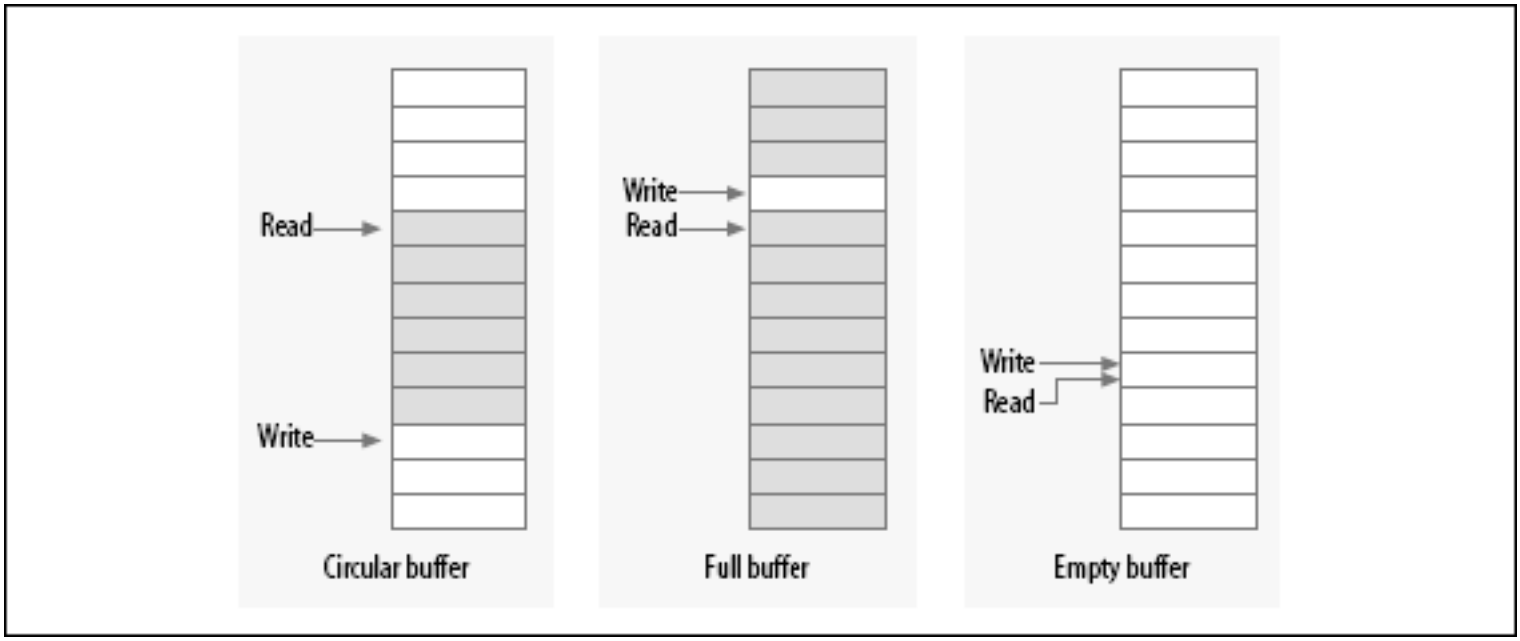
有时, 你可以重新打造你的算法来完全避免加锁的需要. 许多读者/写者情况 -- 如果只有一个写者 -- 常常能够在这个方式下工作. 如果写者小心使数据结构的视图, 由读者所见的, 是一直一致的, 有可能创建一个不加锁的数据结构.

常常可以对无锁的生产者/消费者任务有用的数据结构是环形缓存. 这个算法包含一个生产者安放数据到一个数组的尾端, 而消费者从另一端移走数据. 当到达数组末端, 生产者绕回到开始. 因此一个环形缓存需要一个数组和 2 个索引值来跟踪下一个新值放到哪里, 以及哪个值在下一次应当从缓存中移走.

当小心地实现了, 一个环形缓存在没有多个生产者或消费者时不需要加锁. 生产者是唯一允许修改写索引和它所指向的数组位置的线程. 只要写者在更新写索引之前存储一个新值到缓存中, 读者将一直看到一个一致的视图. 读者, 轮换地, 是唯一存取读索引和它指向的值的线程. 加一点小心到确保 2 个指针不相互覆盖, 生产者和消费者可以并发存取缓存而没有竞争情况.

图环形缓冲展示了在几个填充状态的环形缓存. 这个缓存被定义成一个空情况由读写指针相同来指示, 而满情况发生在写指针紧跟在读指针后面的时候(小心解决绕回!). 当小心地编程, 这个缓存能够不必加锁地使用.

图 5.1. 环形缓冲



在设备驱动中环形缓存出现相当多. 网络适配器, 特别地, 常常使用环形缓存来与处理器交换数据(报文). 注意, 对于 2.6.10, 有一个通用的环形缓存实现在内核中可用; 如何使用它的信息看 <linux/kfifo.h>.

5.7.2. 原子变量

有时, 一个共享资源是一个简单的整数值. 假设你的驱动维护一个共享变量 n_op, 它告知有多少设备操作目前未完成. 正常地, 即便一个简单的操作例如:

```
n_op++;
```

可能需要加锁. 某些处理器可能以原子的方式进行那种递减, 但是你不能依赖它. 但是一个完整的加锁体制对于一个简单的整数值看来过分了. 对于这样的情况, 内核提供了一个原子整数类型称为 atomic_t, 定义在 <asm/atomic.h>.

一个 atomic_t 持有一个 int 值在所有支持的体系上. 但是, 因为这个类型在某些处理器上的工作方式, 整个整数范围可能不是都可用的; 因此, 你不应当指望一个 atomic_t 持有多于 24 位. 下面的操作为这个类型定义并且保证对于一个 SMP 计算机的所有处理器来说是原子的. 操作是非常快的, 因为它们在任何可能时编译成一条单个机器指令.

```
void atomic_set(atomic_t *v, int i);
atomic_t v = ATOMIC_INIT(0);
```

设置原子变量 v 为整数值 i. 你也可在编译时使用宏定义 ATOMIC_INIT 初始化原子值.

```
int atomic_read(atomic_t *v);
```

返回 v 的当前值.

```
void atomic_add(int i, atomic_t *v);
```

由 v 指向的原子变量加 i. 返回值是 void, 因为有一个额外的开销来返回新值, 并且大部分时间不需要知道它.

```
void atomic_sub(int i, atomic_t *v);
```

从 *v 减去 i.

```
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
```

递增或递减一个原子变量.

```
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
```

进行一个特定的操作并且测试结果; 如果, 在操作后, 原子值是 0, 那么返回值是真; 否则, 它是假. 注意没有 atomic_add_and_test.

```
int atomic_add_negative(int i, atomic_t *v);
```

加整数变量 i 到 v. 如果结果是负值返回值是真, 否则为假.

```
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

就像 atomic_add 和其类似函数, 除了它们返回原子变量的新值给调用者.

如同它们说过的, atomic_t 数据项必须通过这些函数存取. 如果你传递一个原子项给一个期望一个整数参数的函数, 你会得到一个编译错误.

你还应当记住, atomic_t 值只在当被置疑的量真正是原子的时候才起作用. 需要多个 atomic_t 变量的操作仍然需要某种其他种类的加锁. 考虑一下下面的代码:

```
atomic_sub(amount, &first_atomic);
atomic_add(amount, &second_atomic);
```

从第一个原子值中减去 amount, 但是还没有加到第二个时, 存在一段时间. 如果事情的这个状态可能产生麻烦给可能在这 2 个操作之间运行的代码, 某种加锁必须采用.

5.7.3. 位操作

atomic_t 类型在进行整数算术时是不错的. 但是, 它无法工作的好, 当你需要以原子方式操作单个位时. 为此, 内核提供了一套函数来原子地修改或测试单个位. 因为整个操作在单步内发生, 没有中断(或者其他处理器)能干扰.

原子位操作非常快, 因为它们使用单个机器指令来进行操作, 而在任何时候低层平台做的时候不用禁止中断. 函数是体系依赖的并且在 <asm/bitops.h> 中声明. 它们保证是原子的, 即便在 SMP 计算机上, 并且对于跨处理器保持一致是有用的.

不幸的是, 键入这些函数中的数据也是体系依赖的. nr 参数(描述要操作哪个位)常常定义为 int, 但是在几个体系中是 unsigned long. 要修改的地址常常是一个 unsigned long 指针, 但是几个体系使用 void * 代替.

各种位操作是:

void set_bit(nr, void *addr);

设置第 nr 位在 addr 指向的数据项中.

void clear_bit(nr, void *addr);

清除指定位在 addr 处的无符号长型数据. 它的语义与 set_bit 的相反.

void change_bit(nr, void *addr);

翻转这个位.

test_bit(nr, void *addr);

这个函数是唯一一个不需要是原子的位操作; 它简单地返回这个位的当前值.

int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);

原子地动作如同前面列出的, 除了它们还返回这个位以前的值.

当这些函数用来存取和修改一个共享的标志, 除了调用它们不用做任何事; 它们以原子发生进行它们的操作. 使用位操作来管理一个控制存取一个共享变量的锁变量, 另一方面, 是有点复杂并且应该有个例子. 大部分现代的代码不以这种方法来使用位操作, 但是象下面的代码仍然在内核中存在.

一段需要存取一个共享数据项的代码试图原子地请求一个锁, 使用 test_and_set_bit 或者 test_and_clear_bit. 通常的实现展示在这里; 它假定锁是在地址 addr 的 nr 位. 它还假定当锁空闲是这个位是 0, 忙为 非零.

```
/* try to set lock */
while (test_and_set_bit(nr, addr) != 0)
```

```
   wait_for_a_while();

/* do your work */

/* release lock, and check... */
if (test_and_clear_bit(nr, addr) == 0)
   something_went_wrong(); /* already released: error */
```

如果你通读内核源码, 你会发现象这个例子的代码. 但是, 最好在新代码中使用自旋锁; 自旋锁很好地调试过, 它们处理问题如同中断和内核抢占, 并且别人读你代码时不必努力理解你在做什么.

5.7.4. seqlock 锁

2.6内核包含了一对新机制打算来提供快速地, 无锁地存取一个共享资源. seqlock 在这种情况下工作, 要保护的资源小, 简单, 并且常常被存取, 并且很少写存取但是必须要快. 基本上, 它们通过允许读者释放对资源的存取, 但是要求这些读者来检查与写者的冲突而工作, 并且当发生这样的冲突时, 重试它们的存取. seqlock 通常不能用在保护包含指针的数据结构, 因为读者可能跟随着一个无效指针而写者在改变数据结构.

seqlock 定义在 <linux/seqlock.h>. 有 2 个通常的方法来初始化一个 seqlock(有 seqlock_t 类型):

```
seqlock_t lock1 = SEQLOCK_UNLOCKED;

seqlock_t lock2;
seqlock_init(&lock2);
```

读存取通过在进入临界区入口获取一个(无符号的)整数序列来工作. 在退出时, 那个序列值与当前值比较; 如果不匹配, 读存取必须重试. 结果是, 读者代码象下面的形式:

```
unsigned int seq;

do {
   seq = read_seqbegin(&the_lock);
   /* Do what you need to do */
} while read_seqretry(&the_lock, seq);
```

这个类型的锁常常用在保护某种简单计算, 需要多个一致的值. 如果这个计算最后的测试表明发生了一个并发的写, 结果被简单地丢弃并且重新计算.

如果你的 seqlock 可能从一个中断处理里存取, 你应当使用 IRQ 安全的版本来代替:

```
unsigned int read_seqbegin_irqsave(seqlock_t *lock, unsigned long flags);
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int seq, unsigned long flags);
```

写者必须获取一个排他锁来进入由一个 seqlock 保护的临界区. 为此, 调用:

```
void write_seqlock(seqlock_t *lock);
```

写锁由一个自旋锁实现, 因此所有的通常的限制都适用. 调用:

```
void write_sequnlock(seqlock_t *lock);
```

来释放锁. 因为自旋锁用来控制写存取, 所有通常的变体都可用:

```
void write_seqlock_irqsave(seqlock_t *lock, unsigned long flags);
void write_seqlock_irq(seqlock_t *lock);
void write_seqlock_bh(seqlock_t *lock);

void write_sequnlock_irqrestore(seqlock_t *lock, unsigned long flags);
void write_sequnlock_irq(seqlock_t *lock);
void write_sequnlock_bh(seqlock_t *lock);
```

还有一个 write_tryseqlock 在它能够获得锁时返回非零.

5.7.5. 读取-拷贝-更新

读取-拷贝-更新(RCU) 是一个高级的互斥方法, 能够有高效率在合适的情况下. 它在驱动中的使用很少但是不是没人知道, 因此这里值得快速浏览下. 那些感兴趣 RCU 算法的完整细节的人可以在由它的创建者出版的白皮书中找到(http://www.rdrop.com/users/paulmck/rclock/intro/rclock_intro.html).

RCU 对它所保护的数据结构设置了不少限制. 它对经常读而极少写的情况做了优化. 被保护的资源应当通过指针来存取, 并且所有对这些资源的引用必须由原子代码持有. 当数据结构需要改变, 写线程做一个拷贝, 改变这个拷贝, 接着使相关的指针对准新的版本 -- 因此, 有了算法的名子. 当内核确认没有留下对旧版本的引用, 它可以被释放.

作为在真实世界中使用 RCU 的例子, 考虑一下网络路由表. 每个外出的报文需要请求检查路由表来决定应当使用哪个接口. 这个检查是快速的, 并且, 一旦内核发现了目标接口, 它不再需要路由表入口项. RCU 允许路由查找在没有锁的情况下进行, 具有相当多的性能好处. 内核中的 Startmode 无线 IP 驱动也使用 RCU 来跟踪它的设备列表.

使用 RCU 的代码应当包含 <linux/rcupdate.h>.

在读这一边, 使用一个 RCU-保护的数据结构的代码应当用 rcu_read_lock 和 rcu_read_unlock 调用将它的引用包含起来. 结果就是, RCU 代码往往是象这样:

```
struct my_stuff *stuff;
rcu_read_lock();
stuff = find_the_stuff(args...);
do_something_with(stuff);
rcu_read_unlock();
```

rcu_read_lock 调用是快的; 它禁止内核抢占但是没有等待任何东西. 在读"锁"被持有时执行的代码必须是原子的. 在对 rcu_read_unlock 调用后, 没有使用对被保护的资源的引用.

需要改变被保护的结构的代码必须进行几个步骤. 第一步是容易的; 它分配一个新结构, 如果需要就从旧的拷贝数据, 接着替换读代码所看到的指针. 在此, 对于读一边的目的, 改变结束了. 任何进入临界区的代码看到数据的新版本.

剩下的是释放旧版本. 当然, 问题是在其他处理器上运行的代码可能仍然有对旧数据的一个引用, 因此它不能立刻释放. 相反, 写代码必须等待直到它知道没有这样的引用存在了. 因为所有持有对这个数据结构引用的代码必须(规则规定)是原子的, 我们知道一旦系统中的每个处理器已经被调度了至少一次, 所有的引用必须消失. 这就是 RCU 所做的; 它留下了一个等待直到所有处理器已经调度的回调; 那个回调接下来被运行来进行清理工作.

改变一个 RCU-保护的数据结构的代码必须通过分配一个 struct rcu_head 来获得它的清理回调, 尽管不需要以任何方式初始化这个结构. 常常, 那个结构被简单地嵌入在 RCU 所保护的大的资源里面. 在改变资源完成后, 应当调用:

```
void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg);
```

给定的 func 在释放资源是安全的时候调用; 传递给 call_rcu的是给同一个 arg. 常常, func 需要的唯一的东西是调用 kfree.

全部 RCU 接口比我们已见的要更加复杂; 它包括, 例如, 辅助函数来使用被保护的链表. 全部内容见相关的头文件.

5.8. 快速参考

本章已介绍了很多符号给并发的管理. 最重要的这些在此总结:

```
#include <asm/semaphore.h>
```

定义旗标和其上操作的包含文件.

```
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```

2 个宏定义, 用来声明和初始化一个在互斥模式下使用的旗标.

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

这 2 函数用来在运行时初始化一个旗标.

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
void up(struct semaphore *sem);
```

加锁和解锁旗标. down 使调用进程进入不可打断睡眠, 如果需要; down_interruptible, 相反, 可以被信号打断. down_trylock 不睡眠; 相反, 它立刻返回如果旗标不可用. 加锁旗标的代码必须最终使用 up 解锁它.

```
struct rw_semaphore;
init_rwsem(struct rw_semaphore *sem);
```

旗标的读者/写者版本和初始化它的函数.

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
```

获得和释放对读者/写者旗标的读存取的函数.

```
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);
```

管理对读者/写者旗标写存取的函数.

```
#include <linux/completion.h>
DECLARE_COMPLETION(name);
init_completion(struct completion *c);
INIT_COMPLETION(struct completion c);
```

描述 Linux completion 机制的包含文件, 已经初始化 completion 的正常方法.
INIT_COMPLETION 应当只用来重新初始化一个之前已经使用过的 completion.

```
void wait_for_completion(struct completion *c);
```

等待一个 completion 事件发出.

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

发出一个 completion 事件. completion 唤醒, 最多, 一个等待着的线程, 而 complete_all 唤醒全部等待者.

```
void complete_and_exit(struct completion *c, long retval);
```

通过调用 complete 来发出一个 completion 事件, 并且为当前线程调用 exit.

```
#include <linux/spinlock.h>
spinlock_t lock = SPIN_LOCK_UNLOCKED;
spin_lock_init(spinlock_t *lock);
```

定义自旋锁接口的包含文件, 以及初始化锁的 2 个方法.

```
void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock);
```

加锁一个自旋锁的各种方法, 并且, 可能地, 禁止中断.

```
int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);
```

上面函数的非自旋版本; 在获取锁失败时返回 0, 否则非零.

```
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

释放一个自旋锁的相应方法.

```
rwlock_t lock = RW_LOCK_UNLOCKED
rwlock_init(rwlock_t *lock);
```

初始化读者/写者锁的 2 个方法.

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);
```

获得一个读者/写者锁的读存取的函数.

```
void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

释放一个读者/写者自旋锁的读存取.

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
```

获得一个读者/写者锁的写存取的函数.

```
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

释放一个读者/写者自旋锁的写存取的函数.

```
#include <asm/atomic.h>
atomic_t v = ATOMIC_INIT(value);
void atomic_set(atomic_t *v, int i);
int atomic_read(atomic_t *v);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
int atomic_add_negative(int i, atomic_t *v);
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

原子地存取整数变量. atomic_t 变量必须只通过这些函数存取.

```
#include <asm/bitops.h>
void set_bit(nr, void *addr);
void clear_bit(nr, void *addr);
void change_bit(nr, void *addr);
test_bit(nr, void *addr);
int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

原子地存取位值; 它们可用做标志或者锁变量. 使用这些函数阻止任何与并发存取这个位相关的竞争情况.

```
#include <linux/seqlock.h>
seqlock_t lock = SEQLOCK_UNLOCKED;
seqlock_init(seqlock_t *lock);
```

定义 seqlock 的包含文件, 已经初始化它们的 2 个方法.

```
unsigned int read_seqbegin(seqlock_t *lock);
unsigned int read_seqbegin_irqsave(seqlock_t *lock, unsigned long flags);
int read_seqretry(seqlock_t *lock, unsigned int seq);
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int seq, unsigned long flags);
```

获得一个 seqlock-保护 的资源的读权限的函数.

```
void write_seqlock(seqlock_t *lock);
void write_seqlock_irqsave(seqlock_t *lock, unsigned long flags);
void write_seqlock_irq(seqlock_t *lock);
void write_seqlock_bh(seqlock_t *lock);
```

获取一个 seqlock-保护的资源的写权限的函数.

```
void write_sequnlock(seqlock_t *lock);
void write_sequnlock_irqrestore(seqlock_t *lock, unsigned long flags);
void write_sequnlock_irq(seqlock_t *lock);
void write_sequnlock_bh(seqlock_t *lock);
```

释放一个 seqlock-保护的资源的写权限的函数.

```
#include <linux/rcupdate.h>
```

需要使用读取-拷贝-更新(RCU)机制的包含文件.

```
void rcu_read_lock;
void rcu_read_unlock;
```

获取对由 RCU 保护的资源的原子读权限的宏定义.

void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg);

安排一个回调在所有处理器已经被调度以及一个 RCU-保护的资源可用被安全的释放之后运行.

上一页 5.7. 加锁的各种选择
上一级 起始页
下一页 第 6 章 高级字符驱动操作

第 6 章 高级字符驱动操作

第 6 章 高级字符驱动操作

目录

在第 3 章, 我们建立了一个完整的设备驱动, 用户可用来写入和读取. 但是一个真正的设备常常提供

比同步读和写更多的功能. 现在我们已装备有调试工具如果发生错误, 并且一个牢固的并发的理解来帮助避免事情进入错误-- 我们可安全地前进并且创建一个更高级的驱动.

本章检查几个你需要理解的概念来编写全特性的字符设备驱动. 我们从实现 ioctl 系统调用开始, 它是用作设备控制的普通接口. 接着我们进入各种和用户空间同步的方法; 在本章结尾, 你有一个充分的认识对于如何使进程睡眠(并且唤醒它们), 实现非阻塞的 I/O, 并且通知用户空间当你的设备可用来读或写. 我们以查看如何在驱动中实现几个不同的设备存取策略来结束.

这里讨论的概念通过 scull 驱动的几个修改版本来演示. 再一次, 所有的都使用内存中的虚拟设备来实现, 因此你可自己试验这些代码而不必使用任何特别的硬件. 到此为止, 你可能在想亲手使用真正的硬件, 但是那将必须等到第 9 章.

6.1. ioctl 接口

大部分驱动需要 -- 除了读写设备的能力 -- 通过设备驱动进行各种硬件控制的能力. 大部分设备可进行超出简单的数据传输之外的操作; 用户空间必须常常能够请求, 例如, 设备锁上它的门, 弹出它的介质, 报告错误信息, 改变波特率, 或者自我销毁. 这些操作常常通过 ioctl 方法来支持, 它通过相同名子的系统调用来实现.

在用户空间, ioctl 系统调用有下面的原型:

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

这个原型由于这些点而凸现于 Unix 系统调用列表, 这些点常常表示函数有数目不定的参数. 在实际系统中, 但是, 一个系统调用不能真正有变数目的参数. 系统调用必须有一个很好定义的原型, 因为用户程序可存取它们只能通过硬件的"门". 因此, 原型中的点不表示一个变数目的参数, 而是一个单个可选的参数, 传统上标识为 char *argp. 这些点在那里只是为了阻止在编译时的类型检查. 第 3 个参数的实际特点依赖所发出的特定的控制命令(第 2 个参数). 一些命令不用参数, 一些用一个整数值, 以及一些使用指向其他数据的指针. 使用一个指针是传递任意数据到 ioctl 调用的方法; 设备接着可与用户空间交换任何数量的数据.

ioctl 调用的非结构化特性使它在内核开发者中失宠. 每个 ioctl 命令, 基本上, 是一个单独的, 常常无文档的系统调用, 并且没有方法以任何类型的全面的方式核查这些调用. 也难于使非结构化的 ioctl 参数在所有系统上一致工作; 例如, 考虑运行在 32-位模式的一个用户进程的 64-位 系统. 结果, 有很大的压力来实现混杂的控制操作, 只通过任何其他的方法. 可能的选择包括嵌入命令到数据流(本章稍后我们将讨论这个方法)或者使用虚拟文件系统, 要么是 sysfs 要么是设备特定的文件系统. (我们将在 14 章看看 sysfs). 但是, 事实是 ioctl 常常是最容易的和最直接的选择,对于真正的设备操作.

ioctl 驱动方法有和用户空间版本不同的原型:

```
int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
```

inode 和 filp 指针是对应应用程序传递的文件描述符 fd 的值, 和传递给 open 方法的相同参数. cmd 参数从用户那里不改变地传下来, 并且可选的参数 arg 参数以一个 unsigned long 的形式传递, 不管它是否由用户给定为一个整数或一个指针. 如果调用程序不传递第 3 个参数, 被驱动操作收到的 arg 值是无定义的. 因为类型检查在这个额外参数上被关闭, 编译器不能警告你如果一个无效的参数被传递给 ioctl, 并且任何关联的错误将难以查找.

如果你可能想到的, 大部分 ioctl 实现包括一个大的 switch 语句来根据 cmd 参数, 选择正确的做法. 不同的命令有不同的数值, 它们常常被给予符号名来简化编码. 符号名通过一个预处理定义来安排. 定制的驱动常常声明这样的符号在它们的头文件中; scull.h 为 scull 声明它们. 用户程序必须, 当然, 包含那个头文件来存取这些符号.

6.1.1. 选择 ioctl 命令

在为 ioctl 编写代码之前, 你需要选择对应命令的数字. 许多程序员的第一个本能的反应是选择一组小数从0或1开始, 并且从此开始向上. 但是, 有充分的理由不这样做. ioctl 命令数字应当在这个系统是唯一的, 为了阻止向错误的设备发出正确的命令而引起的错误. 这样的不匹配不会不可能发生, 并且一个程序可能发现它自己试图改变一个非串口输入系统的波特率, 例如一个 FIFO 或者一个音频设备. 如果这样的 ioctl 号是唯一的, 这个应用程序得到一个 EINVAL 错误而不是继续做不应当做的事情.

为帮助程序员创建唯一的 ioctl 命令代码, 这些编码已被划分为几个位段. Linux 的第一个版本使用 16-位数: 高 8 位是关联这个设备的"魔"数, 低 8 位是一个顺序号, 在设备内唯一. 这样做是因为 Linus 是"无能"的(他自己的话); 一个更好的位段划分仅在后来被设想. 不幸的是, 许多驱动仍然使用老传统. 它们不得不: 改变命令编码会破坏大量的二进制程序,并且这不是内核开发者愿意见到的.

根据 Linux 内核惯例来为你的驱动选择 ioctl 号, 你应当首先检查 include/asm/ioctl.h 和 Documentation/ioctl-number.txt. 这个头文件定义你将使用的位段: type(魔数), 序号, 传输方向, 和参数大小. ioctl-number.txt 文件列举了在内核中使用的魔数,[20] 因此你将可选择你自己的魔数并且避免交叠. 这个文本文件也列举了为什么应当使用惯例的原因.

定义 ioctl 命令号的正确方法使用 4 个位段, 它们有下列的含义. 这个列表中介绍的新符号定义在 <linux/ioctl.h>.

type

魔数. 只是选择一个数(在参考了 ioctl-number.txt之后)并且使用它在整个驱动中. 这个成员是 8 位宽(_IOC_TYPEBITS).

number

序(顺序)号. 它是 8 位(_IOC_NRBITS)宽.

direction

数据传送的方向,如果这个特殊的命令涉及数据传送. 可能的值是 _IOC_NONE(没有数据传输), _IOC_READ, _IOC_WRITE, 和 _IOC_READ|_IOC_WRITE (数据在2个方向被传送). 数据传送是从应用程序的观点来看待的; _IOC_READ 意思是从设备读, 因此设备必须写到用户空间. 注意这个成员是一个位掩码, 因此 _IOC_READ 和 _IOC_WRITE 可使用一个逻辑 AND 操作来抽取.

size

涉及到的用户数据的大小. 这个成员的宽度是依赖体系的, 但是常常是 13 或者 14 位. 你可为你的特定体系在宏 _IOC_SIZEBITS 中找到它的值. 你使用这个 size 成员不是强制的 - 内核不检查它 -- 但是它是一个好主意. 正确使用这个成员可帮助检测用户空间程序的错误并使你实现向后兼容, 如果你曾需要改变相关数据项的大小. 如果你需要更大的数据结构, 但是, 你可忽略这个 size 成员. 我们很快见到如何使用这个成员.

头文件 <asm/ioctl.h>, 它包含在 <linux/ioctl.h> 中, 定义宏来帮助建立命令号, 如下: _IO(type,nr)(给没有参数的命令), _IOR(type, nre, datatype)(给从驱动中读数据的), _IOW(type,nr,datatype)(给写数据), 和 _IOWR(type,nr,datatype)(给双向传送). type 和 number 成员作为参数被传递, 并且 size 成员通过应用 sizeof 到 datatype 参数而得到.

这个头文件还定义宏, 可被用在你的驱动中来解码这个号: _IOC_DIR(nr), _IOC_TYPE(nr), _IOC_NR(nr), 和 _IOC_SIZE(nr). 我们不进入任何这些宏的细节, 因为头文件是清楚的, 并且在本节稍后有例子代码展示.

这里是一些 ioctl 命令如何在 scull 被定义的. 特别地, 这些命令设置和获得驱动的可配置参数.

```
/* 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
```

真正的源文件定义几个额外的这里没有出现的命令.

我们选择实现 2 种方法传递整数参数: 通过指针和通过明确的值(尽管, 由于一个已存在的惯例, ioclt 应当通过指针交换值). 类似地, 2 种方法被用来返回一个整数值:通过指针和通过设置返回值. 这个有效只要返回值是一个正的整数; 如同你现在所知道的, 在从任何系统调用返回时, 一个正值被保留(如同我们在 read 和 write 中见到的), 而一个负值被看作一个错误并且被用来在用户空间设置 errno.[21]

"exchange"和"shift"操作对于 scull 没有特别的用处. 我们实现"exchange"来显示驱动如何结合独立的操作到单个的原子的操作, 并且"shift"来连接"tell"和"query". 有时需要象这样的原子的测试-和-设置操作, 特别地, 当应用程序需要设置和释放锁.

命令的明确的序号没有特别的含义. 它只用来区分命令. 实际上, 你甚至可使用相同的序号给一个读命令和一个写命令, 因为实际的 ioctl 号在"方向"位是不同的, 但是你没有理由这样做. 我们选择在任何地方不使用命令的序号除了声明中, 因此我们不分配一个返回值给它. 这就是为什么明确的号出现在之前给定的定义中. 这个例子展示了一个使用命令号的方法, 但是你有自由不这样做.

除了少数几个预定义的命令(马上就讨论), ioctl 的 cmd 参数的值当前不被内核使用, 并且在将来也很不可能. 因此, 你可以, 如果你觉得懒, 避免前面展示的复杂的声明并明确声明一组调整数字. 另一方面, 如果你做了, 你不会从使用这些位段中获益, 并且你会遇到困难如果你曾提交你的代码来包含在主线内核中. 头文件<linux/kd.h> 是这个老式方法的例子, 使用 16-位的调整值来定义 ioctl 命令. 那个源代码依靠调整数因为使用那个时候遵循的惯例, 不是由于懒惰. 现在改变它可能导致无理由的不兼容.

6.1.2. 返回值

ioctl 的实现常常是一个 switch 语句, 基于命令号. 但是当命令号没有匹配一个有效的操作时缺省的选择应当是什么? 这个问题是有争议的. 几个内核函数返回 -ENIVAL("Invalid argument"), 它有意义是因为命令参数确实不是一个有效的. POSIX 标准, 但是, 说如果一个不合适的 ioctl 命令被发出, 那

么 -ENOTTY 应当被返回. 这个错误码被 C 库解释为"设备的不适当的 ioctl", 这常常正是程序员需要听到的. 然而, 它仍然是相当普遍的来返回 -EINVAL, 对于响应一个无效的 ioctl 命令.

6.1.3. 预定义的命令

尽管 ioctl 系统调用最常用来作用于设备, 内核能识别几个命令. 注意这些命令, 当用到你的设备时, 在你自己的文件操作被调用之前被解码. 因此, 如果你选择相同的号给一个你的 ioctl命令, 你不会看到任何的给那个命令的请求, 并且应用程序获得某些不期望的东西, 因为在 ioctl 号之间的冲突.

预定义命令分为 3 类:

- 可对任何文件发出的(常规, 设备, FIFO, 或者 socket) 的那些.
- 只对常规文件发出的那些.
- 对文件系统类型特殊的那些.

最后一类的命令由宿主文件系统的实现来执行(这是 chattr 命令如何工作的). 设备驱动编写者只对第一类命令感兴趣, 它们的魔数是 "T". 查看其他类的工作留给读者作为练习; ext2_ioctl 是最有趣的函数(并且比预期的要容易理解), 因为它实现 append-only 标志和 immutable 标志.

下列 ioctl 命令是预定义给任何文件, 包括设备特殊的文件:

FIOCLEX

设置 close-on-exec 标志(File IOctl Close on EXec). 设置这个标志使文件描述符被关闭, 当调用进程执行一个新程序时.

FIONCLEX

清除 close-no-exec 标志(File IOctl Not CLose on EXec). 这个命令恢复普通文件行为, 复原上面 FIOCLEX 所做的. FIOASYNC 为这个文件设置或者复位异步通知(如同在本章中"异步通知"一节中讨论的). 注意直到 Linux 2.2.4 版本的内核不正确地使用这个命令来修改 O_SYNC 标志. 因为两个动作都可通过 fcntl 来完成, 没有人真正使用 FIOASYNC 命令, 它在这里出现只是为了完整性.

FIOQSIZE

这个命令返回一个文件或者目录的大小; 当用作一个设备文件, 但是, 它返回一个 ENOTTY 错误.

FIONBIO

"File IOctl Non-Blocking I/O"(在"阻塞和非阻塞操作"一节中描述). 这个调用修改在 filp->f_flags 中的 O_NONBLOCK 标志. 给这个系统调用的第 3 个参数用作指示是否这个标志被置位或者清除. (我们将在本章看到这个标志的角色). 注意常用的改变这个标志的方法是使用 fcntl 系统调用, 使用 F_SETFL 命令.

列表中的最后一项介绍了一个新的系统调用, fcntl, 它看来象 ioctl. 事实上, fcntl 调用非常类似 ioctl, 它也是获得一个命令参数和一个额外的(可选地)参数. 它保持和 ioctl 独立主要是因为历史原因: 当 Unix 开发者面对控制 I/O 操作的问题时, 他们决定文件和设备是不同的. 那时, 有 ioctl 实现的唯一设备是 ttys, 它解释了为什么 -ENOTTY 是标准的对不正确 ioctl 命令的回答. 事情已经改变, 但是 fcntl 保留为一个独立的系统调用.

6.1.4. 使用 ioctl 参数

在看 scull 驱动的 ioctl 代码之前, 我们需要涉及的另一点是如何使用这个额外的参数. 如果它是一个整数, 就容易: 它可以直接使用. 如果它是一个指针, 但是, 必须小心些.

当用一个指针引用用户空间, 我们必须确保用户地址是有效的. 试图存取一个没验证过的用户提供的指针可能导致不正确的行为, 一个内核 oops, 系统崩溃, 或者安全问题. 它是驱动的责任来对每个它使用的用户空间地址进行正确的检查, 并且返回一个错误如果它是无效的.

在第 3 章, 我们看了 copy_from_user 和 copy_to_user 函数, 它们可用来安全地移动数据到和从用户空间. 这些函数也可用在 ioctl 方法中, 但是 ioctl 调用常常包含小数据项, 可通过其他方法更有效地操作. 开始, 地址校验(不传送数据)由函数 access_ok 实现, 它定义在 <asm/uaccess.h>:

```
int access_ok(int type, const void *addr, unsigned long size);
```

第一个参数应当是 VERIFY_READ 或者 VERIFY_WRITE, 依据这个要进行的动作是否是读用户空间内存区或者写它. addr 参数持有一个用户空间地址, size 是一个字节量. 例如, 如果 ioctl 需要从用户空间读一个整数, size 是 sizeof(int). 如果你需要读和写给定地址, 使用 VERIFY_WRITE, 因为它是 VERIRY_READ 的超集.

不象大部分的内核函数, access_ok 返回一个布尔值: 1 是成功(存取没问题)和 0 是失败(存取有问题). 如果它返回假, 驱动应当返回 -EFAULT 给调用者.

关于 access_ok有多个有趣的东西要注意. 首先, 它不做校验内存存取的完整工作; 它只检查看这个内存引用是在这个进程有合理权限的内存范围中. 特别地, access_ok 确保这个地址不指向内核空间内存. 第2, 大部分驱动代码不需要真正调用 access_ok. 后面描述的内存存取函数为你负责这个. 但是, 我们来演示它的使用, 以便你可见到它如何完成.

scull 源码利用了 ioclt 号中的位段来检查参数, 在 switch 之前:

```
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;
```

在调用 access_ok 之后, 驱动可安全地进行真正的传输. 加上 copy_from_user 和 copy_to_user_ 函数, 程序员可利用一组为被最多使用的数据大小(1, 2, 4, 和 8 字节)而优化过的函数. 这些函数在下面列表中描述, 它们定义在 <asm/uaccess.h>:

```
put_user(datum, ptr)
__put_user(datum, ptr)
```

这些宏定义写 datum 到用户空间; 它们相对快, 并且应当被调用来代替 copy_to_user 无论何时要传送单个值时. 这些宏已被编写来允许传递任何类型的指针到 put_user, 只要它是一个用户空间地址. 传送的数据大小依赖 prt 参数的类型, 并且在编译时使用 sizeof 和 typeof 等编译器内建宏确定. 结果是, 如果 prt 是一个 char 指针, 传送一个字节, 以及对于 2, 4, 和 可能的 8 字节.

put_user 检查来确保这个进程能够写入给定的内存地址. 它在成功时返回 0, 并且在错误时返回 -EFAULT. __put_user 进行更少的检查(它不调用 access_ok), 但是仍然能够失败如果被指向的内存对用户是不可写的. 因此, __put_user 应当只用在内存区已经用 access_ok 检查过的时候.

作为一个通用的规则, 当你实现一个 read 方法时, 调用 __put_user 来节省几个周期, 或者当你拷贝几个项时, 因此, 在第一次数据传送之前调用 access_ok 一次, 如同上面 ioctl 所示.

```
get_user(local, ptr)
```

__get_user(local, ptr)

这些宏定义用来从用户空间接收单个数据. 它们象 put_user 和 __put_user, 但是在相反方向传递数据. 获取的值存储于本地变量 local; 返回值指出这个操作是否成功. 再次, __get_user 应当只用在已经使用 access_ok 校验过的地址.

如果做一个尝试来使用一个列出的函数来传送一个不适合特定大小的值, 结果常常是一个来自编译器的奇怪消息, 例如"coversion to non-scalar type requested". 在这些情况中, 必须使用 copy_to_user 或者 copy_from_user.

6.1.5. 兼容性和受限操作

存取一个设备由设备文件上的许可权控制, 并且驱动正常地不涉及到许可权的检查. 但是, 有些情形, 在保证给任何用户对设备的读写许可的地方, 一些控制操作仍然应当被拒绝. 例如, 不是所有的磁带驱动器的用户都应当能够设置它的缺省块大小, 并且一个已经被给予对一个磁盘设备读写权限的用户应当仍然可能被拒绝来格式化它. 在这样的情况下, 驱动必须进行额外的检查来确保用户能够进行被请求的操作.

传统上 unix 系统对超级用户帐户限制了特权操作. 这意味着特权是一个全有-或-全无的东西 -- 超级用户可能任意做任何事情, 但是所有其他的用户被高度限制了. Linux 内核提供了一个更加灵活的系统, 称为能力. 一个基于能力的系统丢弃了全有-或全无模式, 并且打破特权操作为独立的子类. 这种方式, 一个特殊的用户(或者是程序)可被授权来进行一个特定的特权操作而不必泄漏进行其他的, 无关的操作的能力. 内核在许可权管理上排他地使用能力, 并且输出 2 个系统调用 capget 和 capset, 来允许它们被从用户空间管理.

全部能力可在 <linux/capability.h> 中找到. 这些是对系统唯一可用的能力; 对于驱动作者或者系统管理员, 不可能不修改内核源码而来定义新的. 设备驱动编写者可能感兴趣的这些能力的一个子集, 包括下面:

CAP_DAC_OVERRIDE

这个能力来推翻在文件和目录上的存取的限制(数据存取控制, 或者 DAC).

CAP_NET_ADMIN

进行网络管理任务的能力, 包括那些能够影响网络接口的.

CAP_SYS_MODULE

加载或去除内核模块的能力.

CAP_SYS_RAWIO

进行 "raw" I/O 操作的能力. 例子包括存取设备端口或者直接和 USB 设备通讯.

CAP_SYS_ADMIN

一个捕获-全部的能力, 提供对许多系统管理操作的存取.

CAP_SYS_TTY_CONFIG

进行 tty 配置任务的能力.

在进行一个特权操作之前, 一个设备驱动应当检查调用进程有合适的能力; 不这样做可能导致用户进程进行非法的操作, 对系统的稳定和安全有坏的后果. 能力检查是通过 capable 函数来进行的(定义在 <linux/sched.h>):

```
int capable(int capability);
```

在 scull 例子驱动中, 任何用户被许可来查询 quantum 和 quantum 集的大小. 只有特权用户, 但是, 可改变这些值, 因为不适当的值可能很坏地影响系统性能. 当需要时, ioctl 的 scull 实现检查用户的特权级别, 如下:

```
if (! capable (CAP_SYS_ADMIN))
return -EPERM;
```

在这个任务缺乏一个更加特定的能力时, CAP_SYS_ADMIN 被选择来做这个测试.

6.1.6. ioctl 命令的实现

ioctl 的 scull 实现只传递设备的配置参数, 并且象下面这样容易:

```
switch(cmd)
{
case SCULL_IOCRESET:
    scull_quantum = SCULL_QUANTUM;
    scull_qset = SCULL_QSET;
    break;

case SCULL_IOCSQUANTUM: /* Set: arg points to the value */
    if (! capable (CAP_SYS_ADMIN))

        return -EPERM;
    retval = __get_user(scull_quantum, (int __user *)arg);
    break;

case SCULL_IOCTQUANTUM: /* Tell: arg is the value */
    if (! capable (CAP_SYS_ADMIN))
```

```
            return -EPERM;
        scull_quantum = arg;
        break;

case SCULL_IOCGQUANTUM: /* Get: arg is pointer to result */
        retval = __put_user(scull_quantum, (int __user *)arg);
        break;

case SCULL_IOCQQUANTUM: /* Query: return it (it's positive) */
        return scull_quantum;

case SCULL_IOCXQUANTUM: /* eXchange: use arg as pointer */
        if (! capable (CAP_SYS_ADMIN))

            return -EPERM;
        tmp = scull_quantum;
        retval = __get_user(scull_quantum, (int __user *)arg);
        if (retval == 0)

            retval = __put_user(tmp, (int __user *)arg);
        break;

case SCULL_IOCHQUANTUM: /* sHift: like Tell + Query */
        if (! capable (CAP_SYS_ADMIN))
            return -EPERM;
        tmp = scull_quantum;
        scull_quantum = arg;
        return tmp;

default: /* redundant, as cmd was checked against MAXNR */
        return -ENOTTY;
}
return retval;
```

scull 还包含 6 个入口项作用于 scull_qset. 这些入口项和给 scull_quantum 的是一致的, 并且不值得展示出来.

从调用者的观点看(即从用户空间), 这 6 种传递和接收参数的方法看来如下:

```
int quantum;
ioctl(fd,SCULL_IOCSQUANTUM, &quantum);  /* Set by pointer */
ioctl(fd,SCULL_IOCTQUANTUM, quantum);  /* Set by value */
ioctl(fd,SCULL_IOCGQUANTUM, &quantum);  /* Get by pointer */
```

```
quantum = ioctl(fd,SCULL_IOCQQUANTUM);  /* Get by return value */
ioctl(fd,SCULL_IOCXQUANTUM, &quantum);  /* Exchange by pointer */

quantum = ioctl(fd,SCULL_IOCHQUANTUM, quantum); /* Exchange by value */
```

当然, 一个正常的驱动不可能实现这样一个调用模式的混合体. 我们这里这样做只是为了演示做事情的不同方式. 但是, 正常地, 数据交换将一致地进行, 通过指针或者通过值, 并且要避免混合这 2 种技术.

6.1.7. 不用 ioctl 的设备控制

有时控制设备最好是通过写控制序列到设备自身来实现. 例如, 这个技术用在控制台驱动中, 这里所谓的 escape 序列被用来移动光标, 改变缺省的颜色, 或者进行其他的配置任务. 这样实现设备控制的好处是用户可仅仅通过写数据控制设备, 不必使用(或者有时候写)只为配置设备而建立的程序. 当设备可这样来控制, 发出命令的程序甚至常常不需要运行在和它要控制的设备所在的同一个系统上.

例如, setterm 程序作用于控制台(或者其他终端)配置, 通过打印 escape 序列. 控制程序可位于和被控制的设备不同的一台计算机上, 因为一个简单的数据流重定向可完成这个配置工作. 这是每次你运行一个远程 tty 会话时所发生的事情: escape 序列在远端被打印但是影响到本地的 tty; 然而, 这个技术不局限于 ttys.

通过打印来控制的缺点是它给设备增加了策略限制; 例如, 它仅仅当你确信在正常操作时控制序列不会出现在正被写入设备的数据中. 这对于 ttys 只是部分正确的. 尽管一个文本显示意味着只显示 ASCII 字符, 有时控制字符可潜入正被写入的数据中, 并且可能, 因此, 影响控制台的配置. 例如, 这可能发生在你显示一个二进制文件到屏幕时; 产生的乱码可能包含任何东西, 并且最后你常常在你的控制台上出现错误的字体.

通过写来控制是当然的使用方法了, 对于不用传送数据而只是响应命令的设备, 例如遥控设备.

例如, 被你们作者当中的一个编写来好玩的驱动, 移动一个 2 轴上的摄像机. 在这个驱动里, 这个"设备"是一对老式步进电机, 它们不能真正读或写. 给一个步进电机"发送数据流"的概念没有任何意义. 在这个情况下, 驱动解释正被写入的数据作为 ASCII 命令并且转换这个请求为脉冲序列, 来操纵步进电机. 这个概念类似于, 有些, 你发给猫的 AT 命令来建立通讯, 主要的不同是和猫通讯的串口必须也传送真正的数据. 直接设备控制的好处是你可以使用 cat 来移动摄像机, 而不必写和编译特殊的代码来发出 ioctl 调用.

当编写面向命令的驱动, 没有理由实现 ioctl 命令. 一个解释器中的额外命令更容易实现并使用.

有时, 然而, 你可能选择使用其他的方法:不必转变 write 方法为一个解释器和避免 ioctl, 你可能选择完全避免写并且专门使用 ioctl 命令, 而实现驱动为使用一个特殊的命令行工具来发送这些命令到驱动. 这个方法转移复杂性从内核空间到用户空间, 这里可能更易处理, 并且帮助保持驱动小, 而拒

绝使用简单的 cat 或者 echo 命令.

---

[20] 但是, 这个文件的维护在后来有些少见了.

[21] 实际上, 所有的当前使用的 libc 实现(包括 uClibc) 仅将 -4095 到 -1 的值当作错误码. 不幸的是, 能够返回大的负数而不是小的, 没有多大用处.

上一页 下一页
5.8. 快速参考 起始页 6.2. 阻塞 I/O

6.2. 阻塞 I/O

回顾第 3 章, 我们看到如何实现 read 和 write 方法. 在此, 但是, 我们跳过了一个重要的问题:一个驱动当它无法立刻满足请求应当如何响应? 一个对 read 的调用可能当没有数据时到来, 而以后会期待更多的数据. 或者一个进程可能试图写, 但是你的设备没有准备好接受数据, 因为你的输出缓冲满了. 调用进程往往不关心这种问题; 程序员只希望调用 read 或 write 并且使调用返回, 在必要的工作已完成后. 这样, 在这样的情形中, 你的驱动应当(缺省地)阻塞进程, 使它进入睡眠直到请求可继续.

本节展示如何使一个进程睡眠并且之后再次唤醒它. 如常, 但是, 我们必须首先解释几个概念.

6.2.1. 睡眠的介绍

对于一个进程"睡眠"意味着什么? 当一个进程被置为睡眠, 它被标识为处于一个特殊的状态并且从调度器的运行队列中去除. 直到发生某些事情改变了那个状态, 这个进程将不被在任何 CPU 上调度, 并且, 因此, 将不会运行. 一个睡着的进程已被搁置到系统的一边, 等待以后发生事件.

对于一个 Linux 驱动使一个进程睡眠是一个容易做的事情. 但是, 有几个规则必须记住以安全的方式编码睡眠.

这些规则的第一个是: 当你运行在原子上下文时不能睡眠. 我们在第 5 章介绍过原子操作; 一个原子上下文只是一个状态, 这里多个步骤必须在没有任何类型的并发存取的情况下进行. 这意味着, 对于睡眠, 是你的驱动在持有一个自旋锁, seqlock, 或者 RCU 锁时不能睡眠. 如果你已关闭中断你也不能睡眠. 在持有一个旗标时睡眠是合法的, 但是你应当仔细查看这样做的任何代码. 如果代码在持有一个旗标时睡眠, 任何其他的等待这个旗标的线程也睡眠. 因此发生在持有旗标时的任何睡眠应当短暂, 并且你应当说服自己, 由于持有这个旗标, 你不能阻塞这个将最终唤醒你的进程.

另一件要记住的事情是, 当你醒来, 你从不知道你的进程离开 CPU 多长时间或者同时已经发生了什么改变. 你也常常不知道是否另一个进程已经睡眠等待同一个事件; 那个进程可能在你之前醒来并且获取了你在等待的资源. 结果是你不能关于你醒后的系统状态做任何的假设, 并且你必须检查来确保你在等待的条件是, 确实, 真的.

一个另外的相关的点, 当然, 是你的进程不能睡眠除非确信其他人, 在某处的, 将唤醒它. 做唤醒工作的代码必须也能够找到你的进程来做它的工作. 确保一个唤醒发生, 是深入考虑你的代码和对于每次睡眠, 确切知道什么系列的事件将结束那次睡眠. 使你的进程可能被找到, 真正地, 通过一个称为等待队列的数据结构实现的. 一个等待队列就是它听起来的样子:一个进程列表, 都等待一个特定的事件.

在 Linux 中, 一个等待队列由一个"等待队列头"来管理, 一个 wait_queue_head_t 类型的结构, 定义在<linux/wait.h>中. 一个等待队列头可被定义和初始化, 使用:

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

或者动态地, 如下:

```
wait_queue_head_t my_queue;
```

```
init_waitqueue_head(&my_queue);
```

我们将很快返回到等待队列结构, 但是我们知道了足够多的来首先看看睡眠和唤醒.

6.2.2. 简单睡眠

当一个进程睡眠, 它这样做以期望某些条件在以后会成真. 如我们之前注意到的, 任何睡眠的进程必须在它再次醒来时检查来确保它在等待的条件真正为真. Linux 内核中睡眠的最简单方式是一个宏定义, 称为 wait_event(有几个变体); 它结合了处理睡眠的细节和进程在等待的条件的检查. wait_event 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)
```

在所有上面的形式中, queue 是要用的等待队列头. 注意它是"通过值"传递的. 条件是一个被这个宏在睡眠前后所求值的任意的布尔表达式; 直到条件求值为真值, 进程继续睡眠. 注意条件可能被任意次地求值, 因此它不应当有任何边界效应.

如果你使用 wait_event, 你的进程被置为不可中断地睡眠, 如同我们之前已经提到的, 它常常不是你所要的. 首选的选择是 wait_event_interruptible, 它可能被信号中断. 这个版本返回一个你应当检查的整数值; 一个非零值意味着你的睡眠被某些信号打断, 并且你的驱动可能应当返回 -ERESTARTSYS. 最后的版本(wait_event_timeout 和 wait_event_interruptible_timeout)等待一段有限的时间; 在这个时间期间(以嘀哒数表达的, 我们将在第 7 章讨论)超时后, 这个宏返回一个 0 值而不管条件是如何求值的.

图片的另一半, 当然, 是唤醒. 一些其他的执行线程(一个不同的进程, 或者一个中断处理, 也许)必须为你进行唤醒, 因为你的进程, 当然, 是在睡眠. 基本的唤醒睡眠进程的函数称为 wake_up. 它有几个形式(但是我们现在只看其中 2 个):

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

wake_up 唤醒所有的在给定队列上等待的进程(尽管这个情形比那个要复杂一些, 如同我们之后将见到的). 其他的形式(wake_up_interruptible)限制它自己到处理一个可中断的睡眠. 通常, 这 2 个是不用区分的(如果你使用可中断的睡眠); 实际上, 惯例是使用 wake_up 如果你在使用 wait_event , wake_up_interruptible 如果你在使用 wait_event_interruptible.

我们现在知道足够多来看一个简单的睡眠和唤醒的例子. 在这个例子代码中, 你可找到一个称为 sleepy 的模块. 它实现一个有简单行为的设备:任何试图从这个设备读取的进程都被置为睡眠. 无论何时一个进程写这个设备, 所有的睡眠进程被唤醒. 这个行为由下面的 read 和 write 方法实现:

```
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 */

}
```

注意这个例子里 flag 变量的使用. 因为 wait_event_interruptible 检查一个必须变为真的条件, 我们使用 flag 来创建那个条件.

有趣的是考虑当 sleepy_write 被调用时如果有 2 个进程在等待会发生什么. 因为 sleepy_read 重置 flag 为 0 一旦它醒来, 你可能认为醒来的第 2 个进程会立刻回到睡眠. 在一个单处理器系统, 这几乎一直是发生的事情. 但是重要的是要理解为什么你不能依赖这个行为. wake_up_interruptible 调用将使 2 个睡眠进程醒来. 完全可能它们都注意到 flag 是非零, 在另一个有机会重置它之前. 对于这个小模块, 这个竞争条件是不重要的. 在一个真实的驱动中, 这种竞争可能导致少见的难于查找的崩溃. 如果正确的操作要求只能有一个进程看到这个非零值, 它将必须以原子的方式被测试. 我们将见到一个真正的驱动如何处理这样的情况. 但首先我们必须开始另一个主题.

6.2.3. 阻塞和非阻塞操作

在我们看全功能的 read 和 write 方法的实现之前, 我们触及的最后一点是决定何时使进程睡眠. 有时实现正确的 unix 语义要求一个操作不阻塞, 即便它不能完全地进行下去.

有时还有调用进程通知你他不想阻塞, 不管它的 I/O 是否继续. 明确的非阻塞 I/O 由 filp->f_flags 中的 O_NONBLOCK 标志来指示. 这个标志定义于 <linux/fcntl.h>, 被 <linux/fs.h>自动包含. 这个标志得名自"打开-非阻塞", 因为它可在打开时指定(并且起初只能在那里指定). 如果你浏览源码, 你会发现一些对一个 O_NDELAY 标志的引用; 这是一个替代 O_NONBLOCK 的名子, 为兼容 System V 代码而被接受的. 这个标志缺省地被清除, 因为一个等待数据的进程的正常行为仅仅是睡眠. 在一个阻塞操作的情况下, 这是缺省地, 下列的行为应当实现来符合标准语法:

- 如果一个进程调用 read 但是没有数据可用(尚未), 这个进程必须阻塞. 这个进程在有数据达到时被立刻唤醒, 并且那个数据被返回给调用者, 即便小于在给方法的 count 参数中请求的数量.
- 如果一个进程调用 write 并且在缓冲中没有空间, 这个进程必须阻塞, 并且它必须在一个与用作 read 的不同的等待队列中. 当一些数据被写入硬件设备, 并且在输出缓冲中的空间变空闲, 这个进程被唤醒并且写调用成功, 尽管数据可能只被部分写入如果在缓冲只没有空间给被请求的 count 字节.

这 2 句都假定有输入和输出缓冲; 实际上, 几乎每个设备驱动都有. 要求有输入缓冲是为了避免丢失到达的数据, 当无人在读时. 相反, 数据在写时不能丢失, 因为如果系统调用不能接收数据字节, 它们保留在用户空间缓冲. 即便如此, 输出缓冲几乎一直有用, 对于从硬件挤出更多的性能.

在驱动中实现输出缓冲所获得的性能来自减少了上下文切换和用户级/内核级切换的次数. 没有一个输出缓冲(假定一个慢速设备), 每次系统调用接收这样一个或几个字符, 并且当一个进程在 write 中睡眠, 另一个进程运行(那是一次上下文切换). 当第一个进程被唤醒, 它恢复(另一次上下文切换), 写返回(内核/用户转换), 并且这个进程重新发出系统调用来写入更多的数据(用户/内核转换); 这个调用阻塞并且循环继续. 增加一个输出缓冲可允许驱动在每个写调用中接收大的数据块, 性能上有相应的提高. 如果这个缓冲足够大, 写调用在第一次尝试就成功 -- 被缓冲的数据之后将被推到设备 -- 不必控制需要返回用户空间来第二次或者第三次写调用. 选择一个合适的值给输出缓冲显然是设备特定的.

我们不使用一个输入缓冲在 scull中, 因为数据当发出 read 时已经可用. 类似的, 不用输出缓冲, 因为数据被简单地拷贝到和设备关联的内存区. 本质上, 这个设备是一个缓冲, 因此额外缓冲的实现可能是多余的. 我们将在第 10 章见到缓冲的使用.

如果指定 O_NONBLOCK, read 和 write 的行为是不同的. 在这个情况下, 这个调用简单地返回 -EAGAIN(("try it agin")如果一个进程当没有数据可用时调用 read , 或者如果当缓冲中没有空间时它调用 write .

如你可能期望的, 非阻塞操作立刻返回, 允许这个应用程序轮询数据. 应用程序当使用 stdio 函数处理非阻塞文件中, 必须小心, 因为它们容易搞错一个的非阻塞返回为 EOF. 它们始终必须检查 errno.

自然地, O_NONBLOCK 也在 open 方法中有意义. 这个发生在当这个调用真正阻塞长时间时; 例如, 当打开(为读存取)一个 没有写者的(尚无)FIFO, 或者存取一个磁盘文件使用一个悬挂锁. 常常地, 打开一个设备或者成功或者失败, 没有必要等待外部的事件. 有时, 但是, 打开这个设备需要一个长的初始化, 并且你可能选择在你的 open 方法中支持 O_NONBLOCK , 通过立刻返回 -EAGAIN,如果这个标志被设置. 在开始这个设备的初始化进程之后. 这个驱动可能还实现一个阻塞 open 来支持存取策略, 通过类似于文件锁的方式. 我们将见到这样一个实现在"阻塞 open 作为对 EBUSY 的替代"一节, 在本章后面.

一些驱动可能还实现特别的语义给 O_NONBLOCK; 例如, 一个磁带设备的 open 常常阻塞直到插入一个磁带. 如果这个磁带驱动器使用 O_NONBLOCK 打开, 这个 open 立刻成功, 不管是否介质在或不在.

只有 read, write, 和 open 文件操作受到非阻塞标志影响.

6.2.4. 一个阻塞 I/O 的例子

最后, 我们看一个实现了阻塞 I/O 的真实驱动方法的例子. 这个例子来自 scullpipe 驱动; 它是 scull 的一个特殊形式, 实现了一个象管道的设备.

在驱动中, 一个阻塞在读调用上的进程被唤醒, 当数据到达时; 常常地硬件发出一个中断来指示这样一个事件, 并且驱动唤醒等待的进程作为处理这个中断的一部分. scullpipe 驱动不同, 以至于它可运行而不需要任何特殊的硬件或者一个中断处理. 我们选择来使用另一个进程来产生数据并唤醒读进程; 类似地, 读进程被用来唤醒正在等待缓冲空间可用的写者进程.

这个设备驱动使用一个设备结构, 它包含 2 个等待队列和一个缓冲. 缓冲大小是以常用的方法可配置的(在编译时间, 加载时间, 或者运行时间).

```
struct scull_pipe
{
    wait_queue_head_t inq, outq; /* read and write queues */
    char *buffer, *end; /* begin of buf, end of buf */
```

```
    int buffersize; /* used in pointer arithmetic */
    char *rp, *wp; /* where to read, where to write */
    int nreaders, nwriters; /* number of openings for r/w */
    struct fasync_struct *async_queue; /* asynchronous readers */
    struct semaphore sem;  /* mutual exclusion semaphore */
    struct cdev cdev;  /* Char device structure */
};
```

读实现既管理阻塞也管理非阻塞输入, 看来如此:

```
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)
    { /* nothing to read */
        up(&dev->sem); /* release the lock */
        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; /* signal: tell the fs layer to handle it */ /* otherwise loop, but first reacquire the lock */
        if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    }
    /* ok, data is there, return something */

    if (dev->wp > dev->rp)
        count = min(count, (size_t)(dev->wp - dev->rp));
    else /* the write pointer has wrapped, return data up to 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; /* wrapped */
    up (&dev->sem);

    /* finally, awake any writers and return */
    wake_up_interruptible(&dev->outq);
    PDEBUG("\"%s\" did read %li bytes\n",current->comm, (long)count);
```

```
    return count;
}
```

如同你可见的, 我们在代码中留有一些 PDEBUG 语句. 当你编译这个驱动, 你可使能消息机制来易于跟随不同进程间的交互.

让我们仔细看看 scull_p_read 如何处理对数据的等待. 这个 while 循环在持有设备旗标下测试这个缓冲. 如果有数据在那里, 我们知道我们可立刻返回给用户, 不必睡眠, 因此整个循环被跳过. 相反, 如果这个缓冲是空的, 我们必须睡眠. 但是在我们可做这个之前, 我们必须丢掉设备旗标; 如果我们要持有它而睡眠, 就不会有写者有机会唤醒我们. 一旦这个确保被丢掉, 我们做一个快速检查来看是否用户已请求非阻塞 I/O, 并且如果是这样就返回. 否则, 是时间调用 wait_event_interruptible.

一旦我们过了这个调用, 某些东东已经唤醒了我们, 但是我们不知道是什么. 一个可能是进程接收到了一个信号. 包含 wait_event_interruptible 调用的这个 if 语句检查这种情况. 这个语句保证了正确的和被期望的对信号的反应, 它可能负责唤醒这个进程(因为我们处于一个可中断的睡眠). 如果一个信号已经到达并且它没有被这个进程阻塞, 正确的做法是让内核的上层处理这个事件. 到此, 这个驱动返回 -ERESTARTSYS 到调用者; 这个值被虚拟文件系统(VFS)在内部使用, 它或者重启系统调用或者返回 -EINTR 给用户空间. 我们使用相同类型的检查来处理信号, 给每个读和写实现.

但是, 即便没有一个信号, 我们还是不确切知道有数据在那里为获取. 其他人也可能已经在等待数据, 并且它们可能赢得竞争并且首先得到数据. 因此我们必须再次获取设备旗标; 只有这时我们才可以测试读缓冲(在 while 循环中)并且真正知道我们可以返回缓冲中的数据给用户. 全部这个代码的最终结果是, 当我们从 while 循环中退出时, 我们知道旗标被获得并且缓冲中有数据我们可以用.

仅仅为了完整, 我们要注意, scull_p_read 可以在另一个地方睡眠, 在我们获得设备旗标之后: 对 copy_to_user 的调用. 如果 scull 当在内核和用户空间之间拷贝数据时睡眠, 它在持有设备旗标中睡眠. 在这种情况下持有旗标是合理的因为它不能死锁系统(我们知道内核将进行拷贝到用户空间并且在不加锁进程中的同一个旗标下唤醒我们), 并且因为重要的是设备内存数组在驱动睡眠时不改变.

6.2.5. 高级睡眠

许多驱动能够满足它们的睡眠要求, 使用至今我们已涉及到的函数. 但是, 有时需要深入理解 Linux 等待队列机制如何工作. 复杂的加锁或者性能需要可强制一个驱动来使用低层函数来影响一个睡眠. 在本节, 我们在低层看而理解在一个进程睡眠时发生了什么.

6.2.5.1. 一个进程如何睡眠

如果我们深入 <linux/wait.h>, 你见到在 wait_queue_head_t 类型后面的数据结构是非常简单的; 它包含一个自旋锁和一个链表. 这个链表是一个等待队列入口, 它被声明做 wait_queue_t. 这个结构包含关于睡眠进程的信息和它想怎样被唤醒.

使一个进程睡眠的第一步常常是分配和初始化一个 wait_queue_t 结构, 随后将其添加到正确的等待队列. 当所有东西都就位了, 负责唤醒工作的人就可以找到正确的进程.

下一步是设置进程的状态来标志它为睡眠. 在 <linux/sched.h> 中定义有几个任务状态. TASK_RUNNING 意思是进程能够运行, 尽管不必在任何特定的时刻在处理器上运行. 有 2 个状态指示一个进程是在睡眠: TASK_INTERRUPTIBLE 和 TASK_UNTINTERRUPTIBLE; 当然, 它们对应 2 类的睡眠. 其他的状态正常地和驱动

编写者无关.

在 2.6 内核, 对于驱动代码通常不需要直接操作进程状态. 但是, 如果你需要这样做, 使用的代码是:

```
void set_current_state(int new_state);
```

在老的代码中, 你常常见到如此的东西:

```
current->state = TASK_INTERRUPTIBLE;
```

但是象这样直接改变 current 是不鼓励的; 当数据结构改变时这样的代码会轻易地失效. 但是, 上面的代码确实展示了自己改变一个进程的当前状态不能使其睡眠. 通过改变 current 状态, 你已改变了调度器对待进程的方式, 但是你还未让出处理器.

放弃处理器是最后一步, 但是要首先做一件事: 你必须先检查你在睡眠的条件. 做这个检查失败会引入一个竞争条件; 如果在你忙于上面的这个过程并且有其他的线程刚刚试图唤醒你, 如果这个条件变为真会发生什么? 你可能错过唤醒并且睡眠超过你预想的时间. 因此, 在睡眠的代码下面, 典型地你会见到下面的代码:

```
if (!condition)
   schedule();
```

通过在设置了进程状态后检查我们的条件, 我们涵盖了所有的可能的事件进展. 如果我们在等待的条件已经在设置进程状态之前到来, 我们在这个检查中注意到并且不真正地睡眠. 如果之后发生了唤醒, 进程被置为可运行的不管是否我们已真正进入睡眠.

调用 schedule , 当然, 是引用调度器和让出 CPU 的方式. 无论何时你调用这个函数, 你是在告诉内核来考虑应当运行哪个进程并且转换控制到那个进程, 如果必要. 因此你从不知道在 schedule 返回到你的代码会是多长时间.

在 if 测试和可能的调用 schedule (并从其返回)之后, 有些清理工作要做. 因为这个代码不再想睡眠, 它必须保证任务状态被重置为 TASK_RUNNING. 如果代码只是从 schedule 返回, 这一步是不必要的; 那个函数不会返回直到进程处于可运行态. 如果由于不再需要睡眠而对 schedule 的调用被跳过, 进程状态将不正确. 还有必要从等待队列中去除这个进程, 否则它可能被多次唤醒.

6.2.5.2. 手动睡眠

在 Linux 内核的之前的版本, 正式的睡眠要求程序员手动处理所有上面的步骤. 它是一个繁琐的过程, 包含相当多的易出错的样板式的代码. 程序员如果愿意还是可能用那种方式手动睡眠; <linux/sched.h> 包含了所有需要的定义, 以及围绕例子的内核源码. 但是, 有一个更容易的方式.

第一步是创建和初始化一个等待队列. 这常常由这个宏定义完成:

```
DEFINE_WAIT(my_wait);
```

其中, name 是等待队列入口项的名子. 你可用 2 步来做:

```
wait_queue_t my_wait;
init_wait(&my_wait);
```

但是常常更容易的做法是放一个 DEFINE_WAIT 行在循环的顶部, 来实现你的睡眠.

下一步是添加你的等待队列入口到队列, 并且设置进程状态. 2 个任务都由这个函数处理:

void prepare_to_wait(wait_queue_head_t *queue, wait_queue_t *wait, int state);

这里, queue 和 wait 分别地是等待队列头和进程入口. state 是进程的新状态; 它应当或者是 TASK_INTERRUPTIBLE(给可中断的睡眠, 这常常是你所要的)或者 TASK_UNINTERRUPTIBLE(给不可中断睡眠).

在调用 prepare_to_wait 之后, 进程可调用 schedule -- 在它已检查确认它仍然需要等待之后. 一旦 schedule 返回, 就到了清理时间. 这个任务, 也, 被一个特殊的函数处理:

void finish_wait(wait_queue_head_t *queue, wait_queue_t *wait);

之后, 你的代码可测试它的状态并且看是否它需要再次等待.

我们早该需要一个例子了. 之前我们看了 给 scullpipe 的 read 方法, 它使用 wait_event. 同一个驱动中的 write 方法使用 prepare_to_wait 和 finish_wait 来实现它的等待. 正常地, 你不会在一个驱动中象这样混用各种方法, 但是我们这样作是为了能够展示 2 种处理睡眠的方式.

为完整起见, 首先, 我们看 write 方法本身:

```
/* How much space is free? */
static int spacefree(struct scull_pipe *dev)
{
    if (dev->rp == dev->wp)
        return dev->buffersize - 1;
    return ((dev->rp + dev->buffersize - dev->wp) % dev->buffersize) - 1;
}

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;

    /* Make sure there's space to write */
    result = scull_getwritespace(dev, filp);
    if (result)
        return result; /* scull_getwritespace called up(&dev->sem) */
    /* ok, space is there, accept something */
```

```
    count = min(count, (size_t)spacefree(dev));
    if (dev->wp >= dev->rp)
        count = min(count, (size_t)(dev->end - dev->wp)); /* to end-of-buf */
    else /* the write pointer has wrapped, fill up to rp-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; /* wrapped */
    up(&dev->sem);

    /* finally, awake any reader */
    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;
}
```

这个代码看来和 read 方法类似, 除了我们已经将睡眠代码放到了一个单独的函数, 称为 scull_getwritespace. 它的工作是确保在缓冲中有空间给新的数据, 睡眠直到有空间可用. 一旦空间在, scull_p_write 可简单地拷贝用户的数据到那里, 调整指针, 并且唤醒可能已经在等待读取数据的进程.

处理实际的睡眠的代码是:

```
/* Wait for space for writing; caller must hold device semaphore. On
 * error the semaphore will be released before returning. */
static int scull_getwritespace(struct scull_pipe *dev, struct file *filp)
{

    while (spacefree(dev) == 0)
    { /* full */
        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 (spacefree(dev) == 0)
```

```
            schedule();
        finish_wait(&dev->outq, &wait);
        if (signal_pending(current))

            return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
        if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    }
    return 0;

}
```

再次注意 while 循环. 如果有空间可用而不必睡眠, 这个函数简单地返回. 否则, 它必须丢掉设备旗标并且等待. 这个代码使用 DEFINE_WAIT 来设置一个等待队列入口并且 prepare_to_wait 来准备好实际的睡眠. 接着是对缓冲的必要的检查; 我们必须处理的情况是在我们已经进入 while 循环后以及在我们将自己放入等待队列之前 (并且丢弃了旗标), 缓冲中有空间可用了. 没有这个检查, 如果读进程能够在那时完全清空缓冲, 我们可能错过我们能得到的唯一的唤醒并且永远睡眠. 在说服我们自己必须睡眠之后, 我们调用 schedule.

值得再看看这个情况: 当睡眠发生在 if 语句测试和调用 schedule 之间, 会发生什么? 在这个情况里, 都好. 这个唤醒重置了进程状态为 TASK_RUNNING 并且 schedule 返回 -- 尽管不必马上. 只要这个测试发生在进程放置自己到等待队列和改变它的状态之后, 事情都会顺利.

为了结束, 我们调用 finish_wait. 对 signal_pending 的调用告诉我们是否我们被一个信号唤醒; 如果是, 我们需要返回到用户并且使它们稍后再试. 否则, 我们请求旗标, 并且再次照常测试空闲空间.

6.2.5.3. 互斥等待

我们已经见到当一个进程调用 wake_up 在等待队列上, 所有的在这个队列上等待的进程被置为可运行的. 在许多情况下, 这是正确的做法. 但是, 在别的情况下, 可能提前知道只有一个被唤醒的进程将成功获得需要的资源, 并且其余的将简单地再次睡眠. 每个这样的进程, 但是, 必须获得处理器, 竞争资源(和任何的管理用的锁), 并且明确地回到睡眠. 如果在等待队列中的进程数目大, 这个"惊群"行为可能严重降低系统的性能.

为应对实际世界中的惊群问题, 内核开发者增加了一个"互斥等待"选项到内核中. 一个互斥等待的行为非常象一个正常的睡眠, 有 2 个重要的不同:

- 当一个等待队列入口有 WQ_FLAG_EXCLUSEVE 标志置位, 它被添加到等待队列的尾部. 没有这个标志的入口项, 相反, 添加到开始.
- 当 wake_up 被在一个等待队列上调用, 它在唤醒第一个有 WQ_FLAG_EXCLUSIVE 标志的进程后停止.

最后的结果是进行互斥等待的进程被一次唤醒一个, 以顺序的方式, 并且没有引起惊群问题. 但内核仍然每次唤醒所有的非互斥等待者.

在驱动中采用互斥等待是要考虑的, 如果满足 2 个条件: 你希望对资源的有效竞争, 并且唤醒一个进程就足够来完全消耗资源当资源可用时. 互斥等待对 Apacheweb 服务器工作地很好, 例如; 当一个新连接进入, 确实地系统中的一个 Apache 进程应当被唤醒来处理它. 我们在 scullpipe 驱动中不使用互斥等待, 但是; 很少见到竞争数据的读者(或者竞争缓冲空间的写者), 并且我们无法知道一个读者, 一旦被唤醒, 将消耗所有的可用数据.

使一个进程进入可中断的等待, 是调用 prepare_to_wait_exclusive 的简单事情:

void prepare_to_wait_exclusive(wait_queue_head_t *queue, wait_queue_t *wait, int state);

这个调用, 当用来代替 prepare_to_wait, 设置"互斥"标志在等待队列入口项并且添加这个进程到等待队列的尾部. 注意没有办法使用 wait_event 和它的变体来进行互斥等待.

6.2.5.4. 唤醒的细节

我们已展现的唤醒进程的样子比内核中真正发生的要简单. 当进程被唤醒时产生的真正动作是被位于等待队列入口项的一个函数控制的. 缺省的唤醒函数[22]设置进程为可运行的状态, 并且可能地进行一个上下文切换到有更高优先级进程. 设备驱动应当从不需要提供一个不同的唤醒函数; 如果你例外, 关于如何做的信息见 <linux/wait.h>

我们尚未看到所有的 wake_up 变体. 大部分驱动编写者从不需要其他的, 但是, 为完整起见, 这里是整个集合:

wake_up(wait_queue_head_t *queue);
wake_up_interruptible(wait_queue_head_t *queue);

> wake_up 唤醒队列中的每个不是在互斥等待中的进程, 并且就只一个互斥等待者, 如果它存在. wake_up_interruptible 同样, 除了它跳过处于不可中断睡眠的进程. 这些函数, 在返回之前, 使一个或多个进程被唤醒来被调度(尽管如果它们被从一个原子上下文调用, 这就不会发生).

wake_up_nr(wait_queue_head_t *queue, int nr);
wake_up_interruptible_nr(wait_queue_head_t *queue, int nr);

> 这些函数类似 wake_up, 除了它们能够唤醒多达 nr 个互斥等待者, 而不只是一个. 注意传递 0 被解释为请求所有的互斥等待者都被唤醒, 而不是一个没有.

wake_up_all(wait_queue_head_t *queue);
wake_up_interruptible_all(wait_queue_head_t *queue);

> 这种 wake_up 唤醒所有的进程, 不管它们是否进行互斥等待(尽管可中断的类型仍然跳过在做不可中断等待的进程)

wake_up_interruptible_sync(wait_queue_head_t *queue);

> 正常地, 一个被唤醒的进程可能抢占当前进程, 并且在 wake_up 返回之前被调度到处理器. 换句话说, 调用 wake_up 可能不是原子的. 如果调用 wake_up 的进程运行在原子上下文(它可能持有一个自旋锁, 例如, 或者是一个中断处理), 这个重调度不会发生. 正常地, 那个保护是足够的. 但是, 如果你需要明确要求不要被调度出处理器在那时, 你可以使用 wake_up_interruptible 的"同步"变体. 这个函数最常用在当调用者要无论如何重新调度, 并且它会更有效的来首先简单地完成剩下的任何小的工作.

如果上面的全部内容在第一次阅读时没有完全清楚, 不必担心. 很少请求会需要调用 wake_up_interruptible 之外的.

6.2.5.5. 以前的历史: sleep_on

如果你花些时间深入内核源码, 你可能遇到我们到目前忽略讨论的 2 个函数:

```
void sleep_on(wait_queue_head_t *queue);
void interruptible_sleep_on(wait_queue_head_t *queue);
```

如你可能期望的, 这些函数无条件地使当前进程睡眠在给定队列尚. 这些函数强烈不推荐, 但是, 并且你应当从不使用它们. 如果你想想它则问题是明显的: sleep_on 没提供方法来避免竞争条件. 常常有一个窗口在当你的代码决定它必须睡眠时和当 sleep_on 真正影响到睡眠时. 在那个窗口期间到达的唤醒被错过. 因此, 调用 sleep_on 的代码从不是完全安全的.

当前计划对 sleep_on 和 它的变体的调用(有多个我们尚未展示的超时的类型)在不太远的将来从内核中去掉.

6.2.6. 测试 scullpipe 驱动

我们已经见到了 scullpipe 驱动如何实现阻塞 I/O. 如果你想试一试, 这个驱动的源码可在剩下的本书例子中找到. 阻塞 I/O 的动作可通过打开 2 个窗口见到. 第一个可运行一个命令诸如 cat /dev/scullpipe. 如果你接着, 在另一个窗口拷贝文件到 /dev/scullpipe, 你可见到文件的内容出现在第一个窗口.

测试非阻塞的动作是技巧性的, 因为可用于 shell 的传统的程序不做非阻塞操作. misc-progs 源码目录包含下面简单的程序, 称为 nbtest, 来测试非阻塞操作. 所有它做的是拷贝它的输入到它的输出, 使用非阻塞 I/O 和在重试间延时. 延时时间在命令行被传递被缺省是 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);
}
```

如果你在一个进程跟踪工具, 如 strace 下运行这个程序, 你可见到每个操作的成功或者失败, 依赖是否当进行操作时有数据可用.

[22] 它有一个想象的名子 default_wake_function.

6.3. poll 和 select

使用非阻塞 I/O 的应用程序常常使用 poll, select, 和 epoll 系统调用. poll, select 和 epoll 本质上有相同的功能: 每个允许一个进程来决定它是否可读或者写一个或多个文件而不阻塞. 这些调用也可阻塞进程直到任何一个给定集合的文件描述符可用来读或写. 因此, 它们常常用在必须使用多输入输出流的应用程序, 而不必粘连在它们任何一个上. 相同的功能常常由多个函数提供, 因为 2 个是由不同的团队在几乎相同时间完成的: select 在 BSD Unix 中引入, 而 poll 是 System V 的解决方案. epoll 调用[23]添加在 2.5.45, 作为使查询函数扩展到几千个文件描述符的方法.

支持任何一个这些调用都需要来自设备驱动的支持. 这个支持(对所有 3 个调用)由驱动的 poll 方法调用. 这个方法由下列的原型:

unsigned int (*poll) (struct file *filp, poll_table *wait);

这个驱动方法被调用, 无论何时用户空间程序进行一个 poll, select, 或者 epoll 系统调用, 涉及一个和驱动相关的文件描述符. 这个设备方法负责这 2 步:

- 1. 在一个或多个可指示查询状态变化的等待队列上调用 poll_wait. 如果没有文件描述符可用作 I/O, 内核使这个进程在等待队列上等待所有的传递给系统调用的文件描述符.
- 2. 返回一个位掩码, 描述可能不必阻塞就立刻进行的操作.

这 2 个操作常常是直接的, 并且趋向与各个驱动看起来类似. 但是, 它们依赖只能由驱动提供的信息, 因此, 必须由每个驱动单独实现.

poll_table 结构, 给 poll 方法的第 2 个参数, 在内核中用来实现 poll, select, 和 epoll 调用; 它在 <linux/poll.h>中声明, 这个文件必须被驱动源码包含. 驱动编写者不必要知道所有它内容并且必须作为一个不透明的对象使用它; 它被传递给驱动方法以便驱动可用每个能唤醒进程的等待队列来加载它, 并且可改变 poll 操作状态. 驱动增加一个等待队列到 poll_table 结构通过调用函数 poll_wait:

void poll_wait (struct file *, wait_queue_head_t *, poll_table *);

poll 方法的第 2 个任务是返回位掩码, 它描述哪个操作可马上被实现; 这也是直接的. 例如, 如果设备有数据可用, 一个读可能不必睡眠而完成; poll 方法应当指示这个时间状态. 几个标志(通过 <linux/poll.h> 定义)用来指示可能的操作:

POLLIN

如果设备可被不阻塞地读, 这个位必须设置.

POLLRDNORM

这个位必须设置, 如果"正常"数据可用来读. 一个可读的设备返回(POLLIN|POLLRDNORM).

POLLRDBAND

这个位指示带外数据可用来从设备中读取. 当前只用在 Linux 内核的一个地方(DECnet 代码)并且通常对设备驱动不可用.

POLLPRI

高优先级数据(带外)可不阻塞地读取. 这个位使 select 报告在文件上遇到一个异常情况, 因为 selct 报告带外数据作为一个异常情况.

POLLHUP

当读这个设备的进程见到文件尾, 驱动必须设置 POLLUP(hang-up). 一个调用 select 的进程被告知设备是可读的, 如同 selcet 功能所规定的.

POLLERR

一个错误情况已在设备上发生. 当调用 poll, 设备被报告位可读可写, 因为读写都返回一个错误码而不阻塞.

POLLOUT

这个位在返回值中设置, 如果设备可被写入而不阻塞.

POLLWRNORM

这个位和 POLLOUT 有相同的含义, 并且有时它确实是相同的数. 一个可写的设备返回(POLLOUT|POLLWRNORM).

POLLWRBAND

如同 POLLRDBAND , 这个位意思是带有零优先级的数据可写入设备. 只有 poll 的数据报实现使用这个位, 因为一个数据报看传送带外数据.

应当重复一下 POLLRDBAND 和 POLLWRBAND 仅仅对关联到 socket 的文件描述符有意义: 通常设备驱动不使用这些标志.

poll 的描述使用了大量在实际使用中相对简单的东西. 考虑 poll 方法的 scullpipe 实现:

```
static unsigned int scull_p_poll(struct file *filp, poll_table *wait)
{
    struct scull_pipe *dev = filp->private_data;
```

```
    unsigned int mask = 0;

    /*
    * The buffer is circular; it is considered full
    * if "wp" is right behind "rp" and empty if the
    * two are equal.
    */
    down(&dev->sem);
    poll_wait(filp, &dev->inq,  wait);
    poll_wait(filp, &dev->outq, wait);
    if (dev->rp != dev->wp)
        mask |= POLLIN | POLLRDNORM;  /* readable */
    if (spacefree(dev))
        mask |= POLLOUT | POLLWRNORM;  /* writable */
    up(&dev->sem);
    return mask;
}
```

这个代码简单地增加了 2 个 scullpipe 等待队列到 poll_table, 接着设置正确的掩码位, 根据数据是否可以读或写.

所示的 poll 代码缺乏文件尾支持, 因为 scullpipe 不支持文件尾情况. 对大部分真实的设备, poll 方法应当返回 POLLHUP 如果没有更多数据(或者将)可用. 如果调用者使用 select 系统调用, 文件被报告为可读. 不管是使用 poll 还是 select, 应用程序知道它能够调用 read 而不必永远等待, 并且 read 方法返回 0 来指示文件尾.

例如, 对于 真正的FIFO, 读者见到一个文件尾当所有的写者关闭文件, 而在 scullpipe 中读者永远见不到文件尾. 这个做法不同是因为 FIFO 是用作一个 2 个进程的通讯通道, 而 scullpipe 是一个垃圾桶, 人人都可以放数据只要至少有一个读者. 更多地, 重新实现内核中已有的东西是没有意义的, 因此我们选择在我们的例子里实现一个不同的做法.

与 FIFO 相同的方式实现文件尾就意味着检查 dev->nwwriters, 在 read 和 poll 中, 并且报告文件尾(如刚刚描述过的)如果没有进程使设备写打开. 不幸的是, 使用这个实现方法, 如果一个读者打开 scullpipe 设备在写者之前, 它可能见到文件尾而没有机会来等待数据. 解决这个问题的最好的方式是在 open 中实现阻塞, 如同真正的 FIFO 所做的; 这个任务留作读者的一个练习.

6.3.1. 与 read 和 write 的交互

poll 和 select 调用的目的是提前决定是否一个 I/O 操作会阻塞. 在那个方面, 它们补充了 read 和 write. 更重要的是, poll 和 select , 因为它们使应用程序同时等待几个数据流, 尽管我们在 scull 例子里没有采用这个特性.

一个正确的实现对于使应用程序正确工作是必要的: 尽管下列的规则或多或少已经说明过, 我们在此总结它们.

6.3.1.1. 从设备中读数据

- 如果在输入缓冲中有数据, read 调用应当立刻返回, 没有可注意到的延迟, 即便数据少于应用程序要求的, 并且驱动确保其他的数据会很快到达. 你可一直返回小于你被请求的数据, 如果因为任何理由而方便这样(我们在 scull 中这样做), 如果你至少返回一个字节. 在这个情况下, poll 应当返回 POLLIN|POLLRDNORM.
- 如果在输入缓冲中没有数据, 缺省地 read 必须阻塞直到有一个字节. 如果 O_NONBLOCK 被置位, 另一方面, read 立刻返回 -EAGIN (尽管一些老版本 SYSTEM V 返回 0 在这个情况时). 在这些情况中, poll 必须报告这个设备是不可读的直到至少一个字节到达. 一旦在缓冲中有数据, 我们就回到前面的情况.
- 如果我们处于文件尾, read 应当立刻返回一个 0, 不管是否阻塞. 这种情况 poll 应该报告 POLLHUP.

6.3.1.2. 写入设备

- 如果在输出缓冲中有空间, write 应当不延迟返回. 它可接受小于这个调用所请求的数据, 但是它必须至少接受一个字节. 在这个情况下, poll 报告这个设备是可写的, 通过返回 POLLOUT|POLLWRNORM.
- 如果输出缓冲是满的, 缺省地 write 阻塞直到一些空间被释放. 如果 O_NOBLOCK 被设置, write 立刻返回一个 -EAGAIN(老式的 System V Unices 返回 0). 在这些情况下, poll 应当报告文件是不可写的. 另一方面, 如果设备不能接受任何多余数据, write 返回 -ENOSPC("设备上没有空间"), 不管是否设置了 O_NONBLOCK.
- 在返回之前不要调用 wait 来传送数据, 即便当 O_NONBLOCK 被清除. 这是因为许多应用程序选择来找出一个 write 是否会阻塞. 如果设备报告可写, 调用必须不阻塞. 如果使用设备的程序想保证它加入到输出缓冲中的数据被真正传送, 驱动必须提供一个 fsync 方法. 例如, 一个可移除的设备应当有一个 fsnyc 入口.

尽管有一套通用的规则, 还应当认识到每个设备是唯一的并且有时这些规则必须稍微弯曲一下. 例如, 面向记录的设备(例如磁带设备)无法执行部分写.

6.3.1.3. 刷新挂起的输出

我们已经见到 write 方法如何自己不能解决全部的输出需要. fsync 函数, 由同名的系统调用而调用, 填补了这个空缺. 这个方法原型是:

```
int (*fsync) (struct file *file, struct dentry *dentry, int datasync);
```

如果一些应用程序需要被确保数据被发送到设备, fsync 方法必须被实现为不管 O_NONBLOCK 是否被设置. 对 fsync 的调用应当只在设备被完全刷新时返回(即, 输出缓冲为空), 即便这需要一些时

间. datasync 参数用来区分 fsync 和 fdatasync 系统调用; 这样, 它只对文件系统代码有用, 驱动可以忽略它.

fsync 方法没有不寻常的特性. 这个调用不是时间关键的, 因此每个设备驱动可根据作者的口味实现它. 大部分的时间, 字符驱动只有一个 NULL 指针在它们的 fops 中. 阻塞设备, 另一方面, 常常实现这个方法使用通用的 block_fsync, 它接着会刷新设备的所有的块.

6.3.2. 底层的数据结构

poll 和 select 系统调用的真正实现是相当地简单, 对那些感兴趣于它如何工作的人; epoll 更加复杂一点但是建立在同样的机制上. 无论何时用户应用程序调用 poll, select, 或者 epoll_ctl,[24] 内核调用这个系统调用所引用的所有文件的 poll 方法, 传递相同的 poll_table 到每个. poll_table 结构只是对一个函数的封装, 这个函数建立了实际的数据结构. 那个数据结构, 对于 poll和 select, 是一个内存页的链表, 其中包含 poll_table_entry 结构. 每个 poll_table_entry 持有被传递给 poll_wait 的 struct file 和 wait_queue_head_t 指针, 以及一个关联的等待队列入口. 对 poll_wait 的调用有时还添加这个进程到给定的等待队列. 整个的结构必须由内核维护以至于这个进程可被从所有的队列中去除, 在 poll 或者 select 返回之前.

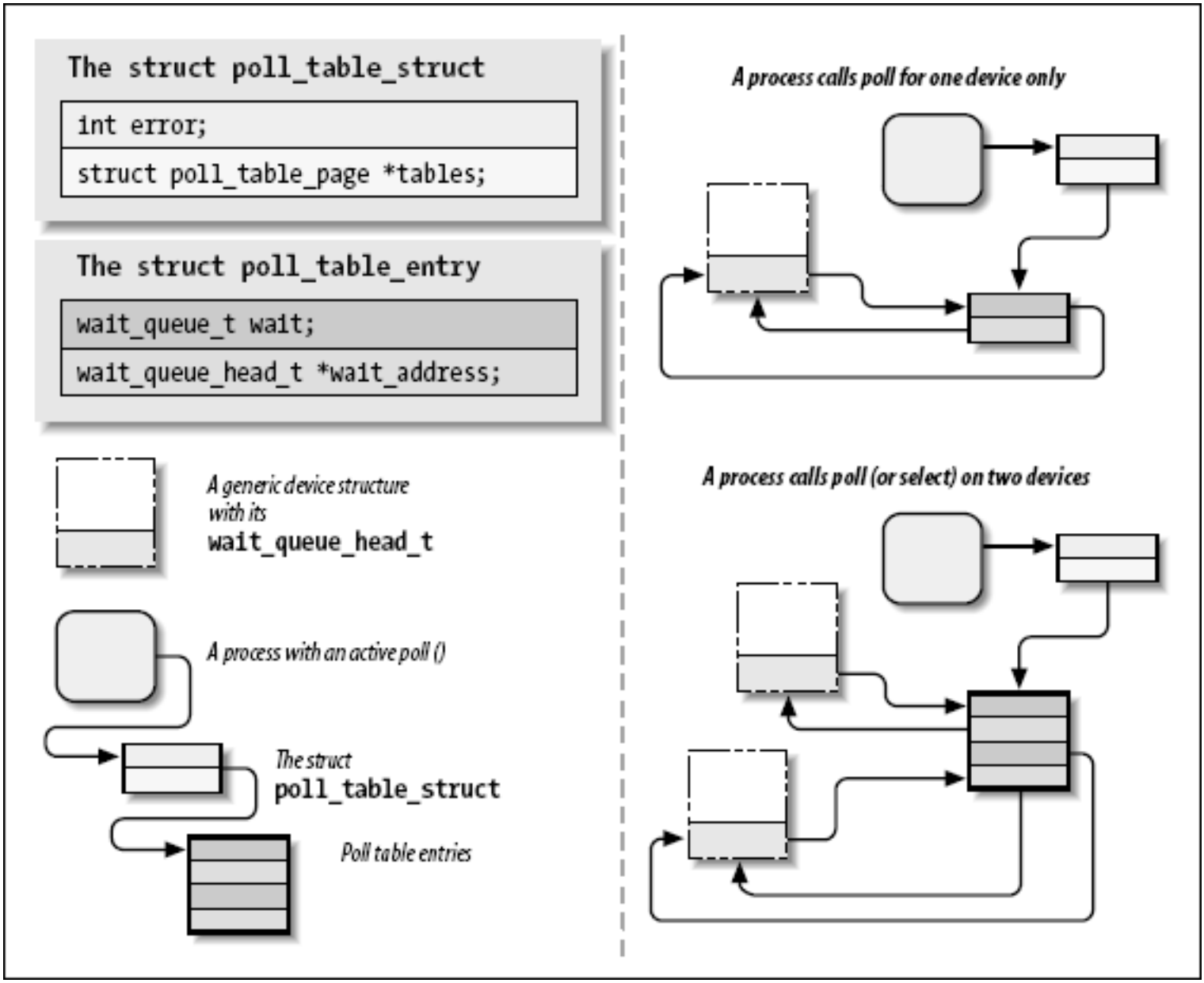
如果被轮询的驱动没有一个指示 I/O 可不阻塞地发生, poll 调用简单地睡眠直到一个它所在的等待队列(可能许多)唤醒它.

在 poll 实现中有趣的是驱动的 poll 方法可能被用一个 NULL 指针作为 poll_table 参数. 这个情况出现由于几个理由. 如果调用 poll 的应用程序已提供了一个 0 的超时值(指示不应当做等待), 没有理由来堆积等待队列, 并且系统简单地不做它. poll_table 指针还被立刻设置为 NULL 在任何被轮询的驱动指示可以 I/O 之后. 因为内核在那一点知道不会发生等待, 它不建立等待队列链表.

当 poll 调用完成, poll_table 结构被去分配, 并且所有的之前加入到 poll 表的等待队列入口被从表和它们的等待队列中移出.

我们试图在图poll 背后的数据结构中展示包含在轮询中的数据结构; 这个图是真实数据结构的简化地表示, 因为它忽略了一个 poll 表地多页性质并且忽略了每个 poll_table_entry 的文件指针.

图 6.1. poll 背后的数据结构



在此, 可能理解在新的系统调用 epoll 后面的动机. 在一个典型的情况中, 一个对 poll 或者 select 的调用只包括一组文件描述符, 所以设置数据结构的开销是小的. 但是, 有应用程序在那里, 它们使用几千个文件描述符. 在这时, 在每次 I/O 操作之间设置和销毁这个数据结构变得非常昂贵. epoll 系统调用家族允许这类应用程序建立内部的内核数据结构只一次, 并且多次使用它们.

---

[23] 实际上, epoll 是一组 3 个调用, 都可用来获得查询功能. 但是, 由于我们的目的, 我们可认为它是一个调用.

[24] 这是建立内部数据结构为将来调用 epoll_wait 的函数.

上一页 上一级 下一页
6.2. 阻塞 I/O 起始页 6.4. 异步通知

6.4. 异步通知

尽管阻塞和非阻塞操作和 select 方法的结合对于查询设备在大部分时间是足够的, 一些情况还不能被我们迄今所见到的技术来有效地解决.

让我们想象一个进程, 在低优先级上执行一个长计算循环, 但是需要尽可能快的处理输入数据. 如果这个进程在响应新的来自某些数据获取外设的报告, 它应当立刻知道当新数据可用时. 这个应用程序可能被编写来调用 poll 有规律地检查数据, 但是, 对许多情况, 有更好的方法. 通过使能异步通知, 这个应用程序可能接受一个信号无论何时数据可用并且不需要让自己去查询.

用户程序必须执行 2 个步骤来使能来自输入文件的异步通知. 首先, 它们指定一个进程作为文件的拥有者. 当一个进程使用 fcntl 系统调用发出 F_SETOWN 命令, 这个拥有者进程的 ID 被保存在 filp->f_owner 给以后使用. 这一步对内核知道通知谁是必要的. 为了真正使能异步通知, 用户程序必须设置 FASYNC 标志在设备中, 通过 F_SETFL fcntl 命令.

在这 2 个调用已被执行后, 输入文件可请求递交一个 SIGIO 信号, 无论何时新数据到达. 信号被发送给存储于 filp->f_owner 中的进程(或者进程组, 如果值为负值).

例如, 下面的用户程序中的代码行使能了异步的通知到当前进程, 给 stdin 输入文件:

```
signal(SIGIO, &input_handler); /* dummy sample; sigaction() is better */
fcntl(STDIN_FILENO, F_SETOWN, getpid());
oflags = fcntl(STDIN_FILENO, F_GETFL);
fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);
```

这个在源码中名为 asynctest 的程序是一个简单的程序, 读取 stdin. 它可用来测试 scullpipe 的异步能力. 这个程序和 cat 类似但是不结束于文件尾; 它只响应输入, 而不是没有输入.

注意, 但是, 不是所有的设备都支持异步通知, 并且你可选择不提供它. 应用程序常常假定异步能力只对 socket 和 tty 可用.

输入通知有一个剩下的问题. 当一个进程收到一个 SIGIO, 它不知道哪个输入文件有新数据提供. 如果多于一个文件被使能异步地通知挂起输入的进程, 应用程序必须仍然靠 poll 或者 select 来找出发生了什么.

6.4.1. 驱动的观点

对我们来说一个更相关的主题是设备驱动如何实现异步信号. 下面列出了详细的操作顺序, 从内核的观点:

- 1. 当发出 F_SETOWN, 什么都没发生, 除了一个值被赋值给 filp->f_owner.
- 2. 当 F_SETFL 被执行来打开 FASYNC, 驱动的 fasync 方法被调用. 这个方法被调用无论何时 FASYNC 的值在 filp->f_flags 中被改变来通知驱动这个变化, 因此它可正确地响应. 这个标志在文件被打开时缺省地被清除. 我们将看这个驱动方法的标准实现, 在本节.
- 3. 当数据到达, 所有的注册异步通知的进程必须被发出一个 SIGIO 信号.

虽然实现第一步是容易的--在驱动部分没有什么要做的--其他的步骤包括维护一个动态数据结构来跟踪不同的异步读者; 可能有几个. 这个动态数据结构, 但是, 不依赖特殊的设备, 并且内核提供了一个合适的通用实现这样你不必重新编写同样的代码给每个驱动.

Linux 提供的通用实现是基于一个数据结构和 2 个函数(它们在前面所说的第 2 步和第 3 步被调用). 声明相关材料的头文件是<linux/fs.h>(这里没新东西), 并且数据结构被称为 struct fasync_struct. 至于等待队列, 我们需要插入一个指针在设备特定的数据结构中.

驱动调用的 2 个函数对应下面的原型:

```
int fasync_helper(int fd, struct file *filp, int mode, struct fasync_struct **fa);
void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

fasync_helper 被调用来从相关的进程列表中添加或去除入口项, 当 FASYNC 标志因一个打开文件而改变. 它的所有参数除了最后一个, 都被提供给 fasync 方法并且被直接传递. 当数据到达时 kill_fasync 被用来通知相关的进程. 它的参数是被传递的信号(常常是 SIGIO)和 band, 这几乎都是 POLL_IN[25](但是这可用来发送"紧急"或者带外数据, 在网络代码里).

这是 scullpipe 如何实现 fasync 方法的:

```
static int scull_p_fasync(int fd, struct file *filp, int mode)
{
 struct scull_pipe *dev = filp->private_data;
 return fasync_helper(fd, filp, mode, &dev->async_queue);
}
```

显然所有的工作都由 fasync_helper 进行. 但是, 不可能实现这个功能在没有一个方法在驱动里的情况下, 因为这个帮忙函数需要存取正确的指向 struct fasync_struct (这里是 与dev->async_queue)的指针, 并且只有驱动可提供这个信息.

当数据到达, 下面的语句必须被执行来通知异步读者. 因为对 sucllpipe 读者的新数据通过一个发出 write 的进程被产生, 这个语句出现在 scullpipe 的 write 方法中.

```
if (dev->async_queue)
 kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
```

注意, 一些设备还实现异步通知来指示当设备可被写入时; 在这个情况, 当然, kill_fasnyc 必须被使用一个 POLL_OUT 模式来调用.

可能会出现我们已经完成但是仍然有一件事遗漏. 我们必须调用我们的 fasync 方法, 当文件被关闭来从激活异步读者列表中去除文件. 尽管这个调用仅当 filp->f_flags 被设置为 FASYNC 时需要, 调用这个函数无论如何不会有问题并且是常见的实现. 下面的代码行, 例如, 是 scullpipe 的 release 方法的一部分:

```
/* remove this filp from the asynchronously notified filp's */
scull_p_fasync(-1, filp, 0);
```

这个在异步通知之下的数据结构一直和结构 struct wait_queue 是一致的, 因为 2 种情况都涉及等待一个事件. 区别是这个 struct file 被用来替代 struct task_struct. 队列中的结构 file 接着用来存取 f_owner, 为了通知进程.

---

[25] POLL_IN 是一个符号, 用在异步通知代码中; 它等同于 POLLIN|POLLRDNORM.

6.5. 移位一个设备

本章最后一个需要我们涉及的东西是 llseek 方法, 它有用(对于某些设备)并且容易实现.

6.5.1. llseek 实现

llseek 方法实现了 lseek 和 llseek 系统调用. 我们已经说了如果 llseek 方法从设备的操作中缺失, 内核中的缺省的实现进行移位通过修改 filp->f_pos, 这是文件中的当前读写位置. 请注意对于 lseek 系统调用要正确工作, 读和写方法必须配合, 通过使用和更新它们收到的作为的参数的 offset 项.

你可能需要提供你自己的方法, 如果移位操作对应一个在设备上的物理操作. 一个简单的例子可在 scull 驱动中找到:

```
loff_t scull_llseek(struct file *filp, loff_t off, int whence)
{
    struct scull_dev *dev = filp->private_data;
    loff_t newpos;

    switch(whence)
    {
    case 0: /* SEEK_SET */
        newpos = off;
        break;

    case 1: /* SEEK_CUR */
        newpos = filp->f_pos + off;
        break;

    case 2: /* SEEK_END */
        newpos = dev->size + off;
        break;

    default: /* can't happen */
        return -EINVAL;
    }
    if (newpos < 0)
        return -EINVAL;
```

```
    filp->f_pos = newpos;
    return newpos;
}
```

唯一设备特定的操作是从设备中获取文件长度. 在 scull 中 read 和 write 方法如需要地一样协作, 如同在第 3 章所示.

尽管刚刚展示的这个实现对 scull 有意义, 它处理一个被很好定义了的数据区, 大部分设备提供了一个数据流而不是一个数据区(想想串口或者键盘), 并且移位这些设备没有意义. 如果这就是你的设备的情况, 你不能只制止声明 llseek 操作, 因为缺省的方法允许移位. 相反, 你应当通知内核你的设备不支持 llseek , 通过调用 nonseekable_open 在你的 open 方法中.

```
int nonseekable_open(struct inode *inode; struct file *filp);
```

这个调用标识了给定的 filp 为不可移位的; 内核从不允许一个 lseek 调用在这样一个文件上成功. 通过用这样的方式标识这个文件, 你可确定不会有通过 pread 和 pwrite 系统调用的方式来试图移位这个文件.

完整起见, 你也应该在你的 file_operations 结构中设置 llseek 方法到一个特殊的帮忙函数 no_llseek, 它定义在 <linux/fs.h>.

上一页 6.4. 异步通知 | 上一级 起始页 | 下一页 6.6. 在一个设备文件上的存取控制

6.6. 在一个设备文件上的存取控制

提供存取控制对于一个设备节点来说有时是至关重要的. 不仅是非授权用户不能使用设备(由文件系统许可位所强加的限制), 而且有时只有授权用户才应当被允许来打开设备一次.

这个问题类似于使用 ttys 的问题. 在那个情况下, login 进程改变设备节点的所有权, 无论何时一个用户登录到系统, 为了阻止其他的用户打扰或者偷听这个 tty 的数据流. 但是, 仅仅为了保证对它的唯一读写而使用一个特权程序在每次打开它时改变一个设备的拥有权是不实际的.

迄今所显示的代码没有实现任何的存取控制, 除了文件系统许可位. 如果系统调用 open 将请求递交给驱动, open 就成功了. 我们现在介绍几个新技术来实现一些额外的检查.

每个在本节中展示的设备有和空的 scull 设备有相同的行为(即, 它实现一个持久的内存区)但是在存取控制方面和 scull 不同, 这个实现在 open 和 release 操作中.

6.6.1. 单 open 设备

提供存取控制的强力方式是只允许一个设备一次被一个进程打开(单次打开). 这个技术最好是避免因为它限制了用户的灵活性. 一个用户可能想运行不同的进程在一个设备上, 一个读状态信息而另一个写数据. 在某些情况下, 用户通过一个外壳脚本运行几个简单的程序可做很多事情, 只要它们可并发存取设备. 换句话说, 实现一个单 open 行为实际是在创建策略, 这样可能会介入你的用户要做的范围.

只允许单个进程打开设备有不期望的特性, 但是它也是一个设备驱动最简单实现的存取控制, 因此它在这里被展示. 这个源码是从一个称为 scullsingle 的设备中提取的.

scullsingle 设备维护一个 atiomic_t 变量, 称为 scull_s_available; 这个变量被初始化为值 1, 表示设备确实可用. open 调用递减并测试 scull_s_available 并拒绝存取如果其他人已经使设备打开.

```
static atomic_t scull_s_available = ATOMIC_INIT(1);
static int scull_s_open(struct inode *inode, struct file *filp)
{

    struct scull_dev *dev = &scull_s_device; /* device information */
    if (! atomic_dec_and_test (&scull_s_available))
    {
        atomic_inc(&scull_s_available);
```

```
        return -EBUSY; /* already open */
    }

    /* then, everything else is copied from the bare scull device */
    if ( (filp->f_flags & O_ACCMODE) == O_WRONLY)

        scull_trim(dev);
    filp->private_data = dev;
    return 0; /* success */
}
```

release 调用, 另一方面, 标识设备为不再忙:

```
static int scull_s_release(struct inode *inode, struct file *filp)
{
    atomic_inc(&scull_s_available); /* release the device */
    return 0;
}
```

正常地, 我们建议你将 open 标志 scul_s_available 放在设备结构中(scull_dev 这里), 因为, 从概念上, 它属于这个设备. scull 驱动, 但是, 使用独立的变量来保持这个标志, 因此它可使用和空 scull 设备同样的设备结构和方法, 并且最少的代码复制.

6.6.2. 一次对一个用户限制存取

单打开设备之外的下一步是使一个用户在多个进程中打开一个设备, 但是一次只允许一个用户打开设备. 这个解决方案使得容易测试设备, 因为用户一次可从几个进程读写, 但是假定这个用户负责维护在多次存取中的数据完整性. 这通过在 open 方法中添加检查来实现; 这样的检查在通常的许可检查后进行, 并且只能使存取更加严格, 比由拥有者和组许可位所指定的限制. 这是和 ttys 所用的存取策略是相同的, 但是它不依赖于外部的特权程序.

这些存取策略实现地有些比单打开策略要奇怪. 在这个情况下, 需要 2 项: 一个打开计数和设备拥有者 uid. 再一次, 给这个项的最好的地方是在设备结构中; 我们的例子使用全局变量代替, 是因为之前为 scullsingle 所解释的的原因. 这个设备的名子是 sculluid.

open 调用在第一次打开时同意了存取但是记住了设备拥有者. 这意味着一个用户可打开设备多次, 因此允许协调多个进程对设备并发操作. 同时, 没有其他用户可打开它, 这样避免了外部干扰. 因为这个函数版本几乎和之前的一致, 这样相关的部分在这里被复制:

```
spin_lock(&scull_u_lock);
if (scull_u_count &&
        (scull_u_owner != current->uid) && /* allow user */
```

```
        (scull_u_owner != current->euid) && /* allow whoever did su */
        !capable(CAP_DAC_OVERRIDE))
{ /* still allow root */
    spin_unlock(&scull_u_lock);
    return -EBUSY; /* -EPERM would confuse the user */
}

if (scull_u_count == 0)
    scull_u_owner = current->uid; /* grab it */

scull_u_count++;
spin_unlock(&scull_u_lock);
```

注意 sculluid 代码有 2 个变量 (scull_u_owner 和 scull_u_count)来控制对设备的存取, 并且这样可被多个进程并发地存取. 为使这些变量安全, 我们使用一个自旋锁控制对它们的存取(scull_u_lock). 没有这个锁, 2 个(或多个)进程可同时测试 scull_u_count , 并且都可能认为它们拥有设备的拥有权. 这里使用一个自旋锁, 是因为这个锁被持有极短的时间, 并且驱动在持有这个锁时不做任何可睡眠的事情.

我们选择返回 -EBUSY 而不是 -EPERM, 即便这个代码在进行许可检测, 为了给一个被拒绝存取的用户指出正确的方向. 对于"许可拒绝"的反应常常是检查 /dev 文件的模式和拥有者, 而"设备忙"正确地建议用户应当寻找一个已经在使用设备的进程.

这个代码也检查来看是否正在试图打开的进程有能力来覆盖文件存取许可; 如果是这样, open 被允许即便打开进程不是设备的拥有者. CAP_DAC_OVERRIDE 能力在这个情况中适合这个任务.

release 方法看来如下:

```
static int scull_u_release(struct inode *inode, struct file *filp)
{
    spin_lock(&scull_u_lock);
    scull_u_count--; /* nothing else */
    spin_unlock(&scull_u_lock);
    return 0;
}
```

再次, 我们在修改计数之前必须获得锁, 来确保我们没有和另一个进程竞争.

6.6.3. 阻塞 open 作为对 EBUSY 的替代

当设备不可存取, 返回一个错误常常是最合理的方法, 但是有些情况用户可能更愿意等待设备.

例如, 如果一个数据通讯通道既用于规律地预期地传送报告(使用 crontab), 也用于根据用户的需要偶尔地使用, 对于被安排的操作最好是稍微延迟, 而不是只是因为通道当前忙而失败.

这是程序员在设计一个设备驱动时必须做的一个选择之一, 并且正确的答案依赖正被解决的实际问题.

对 EBUSY 的替代, 如同你可能已经想到的, 是实现阻塞 open. scullwuid 设备是一个在打开时等待设备而不是返回 -EBUSY 的 sculluid 版本. 它不同于 sculluid 只在下面的打开操作部分:

```
spin_lock(&scull_w_lock);
while (! scull_w_available())
{
      spin_unlock(&scull_w_lock);
      if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
      if (wait_event_interruptible (scull_w_wait, scull_w_available()))
            return -ERESTARTSYS; /* tell the fs layer to handle it */
      spin_lock(&scull_w_lock);
}
if (scull_w_count == 0)
      scull_w_owner = current->uid; /* grab it */
scull_w_count++;
spin_unlock(&scull_w_lock);
```

这个实现再次基于一个等待队列. 如果设备当前不可用, 试图打开它的进程被放置到等待队列直到拥有进程关闭设备.

release 方法, 接着, 负责唤醒任何挂起的进程:

```
static int scull_w_release(struct inode *inode, struct file *filp)
{

 int temp;
 spin_lock(&scull_w_lock);
 scull_w_count--;
 temp = scull_w_count;
 spin_unlock(&scull_w_lock);
   if (temp == 0)
 wake_up_interruptible_sync(&scull_w_wait); /* awake other uid's */
 return 0;
}
```

这是一个例子, 这里调用 wake_up_interruptible_sync 是有意义的. 当我们做这个唤醒, 我们只是要返回到用户空间, 这对于系统是一个自然的调度点. 当我们做这个唤醒时不是潜在地重新调度, 最好只是调用 "sync" 版本并且完成我们的工作.

阻塞式打开实现的问题是对于交互式用户真的不好, 他们不得不猜想哪里出错了. 交互式用户常常调用标准命令, 例如 cp 和 tar, 并且不能增加 O_NONBLOCK 到 open 调用. 有些使用磁带驱动器做备份的人可能喜欢有一个简单的"设备或者资源忙"消息, 来替代被扔在一边猜为什么今天的硬盘驱动器这么安静, 此时 tar 应当在扫描它.

这类的问题(需要一个不同的, 不兼容的策略对于同一个设备)最好通过为每个存取策略实现一个设备节点来实现. 这个做法的一个例子可在 linux 磁带驱动中找到, 它提供了多个设备文件给同一个设备. 例如, 不同的设备文件将使驱动器使用或者不用压缩记录, 或者自动回绕磁带当设备被关闭时.

6.6.4. 在 open 时复制设备

管理存取控制的另一个技术是创建设备的不同的私有拷贝, 根据打开它的进程.

明显地, 这只当设备没有绑定到一个硬件实体时有可能; scull 是一个这样的"软件"设备的例子. /dev/tty 的内部使用类似的技术来给它的进程一个不同的 /dev 入口点呈现的视图. 当设备的拷贝被软件驱动创建, 我们称它们为虚拟设备--就象虚拟控制台使用一个物理 tty 设备.

结构这类的存取控制很少需要, 这个实现可说明内核代码是多么容易改变应用程序的对周围世界的看法(即, 计算机).

/dev/scullpriv 设备节点在 scull 软件包只实现虚拟设备. scullpriv 实现使用了进程的控制 tty 的设备号作为对存取虚拟设备的钥匙. 但是, 你可以轻易地改变代码来使用任何整数值作为钥匙; 每个选择都导致一个不同的策略. 例如, 使用 uid 导致一个不同地虚拟设备给每个用户, 而使用一个 pid 钥匙创建一个新设备为每个存取它的进程.

使用控制终端的决定打算用在易于使用 I/O 重定向测试设备: 设备被所有的在同一个虚拟终端运行的命令所共享, 并且保持独立于在另一个终端上运行的命令所见到的.

open 方法看来象下面的代码. 它必须寻找正确的虚拟设备并且可能创建一个. 这个函数的最后部分没有展示, 因为它拷贝自空的 scull, 我们已经见到过.

```
/* The clone-specific data structure includes a key field */
struct scull_listitem
{
    struct scull_dev device;
    dev_t key;
    struct list_head list;
```

```
};
/* The list of devices, and a lock to protect it */
static LIST_HEAD(scull_c_list);
static spinlock_t scull_c_lock = SPIN_LOCK_UNLOCKED;

/* Look for a device or create one if missing */
static struct scull_dev *scull_c_lookfor_device(dev_t key)
{

    struct scull_listitem *lptr;
    list_for_each_entry(lptr, &scull_c_list, list)
    {
        if (lptr->key == key)
            return &(lptr->device);
    }

    /* not found */
    lptr = kmalloc(sizeof(struct scull_listitem), GFP_KERNEL);
    if (!lptr)
        return NULL;
    /* initialize the device */
    memset(lptr, 0, sizeof(struct scull_listitem));
    lptr->key = key;
    scull_trim(&(lptr->device)); /* initialize it */
    init_MUTEX(&(lptr->device.sem));

    /* place it in the list */
    list_add(&lptr->list, &scull_c_list);

    return &(lptr->device);
}

static int scull_c_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev;

    dev_t key;
    if (!current->signal->tty)
    {
        PDEBUG("Process \"%s\" has no ctl tty\n", current->comm);
        return -EINVAL;

    }
```

```
    key = tty_devnum(current->signal->tty);

    /* look for a scullc device in the list */
    spin_lock(&scull_c_lock);
    dev = scull_c_lookfor_device(key);
    spin_unlock(&scull_c_lock);

    if (!dev)
        return -ENOMEM;

    /* then, everything else is copied from the bare scull device */
```

这个 release 方法没有做特殊的事情. 它将在最后的关闭时正常地释放设备, 但是我们不选择来维护一个 open 计数而来简化对驱动的测试. 如果设备在最后的关闭被释放, 你将不能读相同的数据在写入设备之后, 除非一个后台进程将保持它打开. 例子驱动采用了简单的方法来保持数据, 以便在下一次打开时, 你会发现它在那里. 设备在 scull_cleanup 被调用时释放.

这个代码使用通用的 linux 链表机制, 而不是从头开始实现相同的功能. linux 链表在第 11 章中讨论.

这里是 /dev/scullpriv 的 release 实现, 它结束了对设备方法的讨论.

```
static int scull_c_release(struct inode *inode, struct file *filp)
{
    /*
    *Nothing to do, because the device is persistent.
    *A `real' cloned device should be freed on last close */

    return 0;
}
```

6.7. 快速参考

本章介绍了下面的符号和头文件:

#include <linux/ioctl.h>

声明用来定义 ioctl 命令的宏定义. 当前被 <linux/fs.h> 包含.

_IOC_NRBITS
_IOC_TYPEBITS
_IOC_SIZEBITS
_IOC_DIRBITS

ioctl 命令的不同位段所使用的位数. 还有 4 个宏来指定 MASK 和 4 个指定 SHIFT, 但是它们主要是给内部使用. _IOC_SIZEBIT 是一个要检查的重要的值, 因为它跨体系改变.

_IOC_NONE
_IOC_READ
_IOC_WRITE

"方向"位段可能的值. "read" 和 "write" 是不同的位并且可相或来指定 read/write. 这些值是基于 0 的.

_IOC(dir,type,nr,size)
_IO(type,nr)
_IOR(type,nr,size)
_IOW(type,nr,size)
_IOWR(type,nr,size)

用来创建 ioclt 命令的宏定义.

_IOC_DIR(nr)
_IOC_TYPE(nr)
_IOC_NR(nr)
_IOC_SIZE(nr)

用来解码一个命令的宏定义. 特别地, _IOC_TYPE(nr) 是 _IOC_READ 和 _IOC_WRITE 的 OR 结合.

#include <asm/uaccess.h>
int access_ok(int type, const void *addr, unsigned long size);

检查一个用户空间的指针是可用的. access_ok 返回一个非零值, 如果应当允许存取.
VERIFY_READ
VERIFY_WRITE

access_ok 中 type 参数的可能取值. VERIFY_WRITE 是 VERIFY_READ 的超集.
#include <asm/uaccess.h>
int put_user(datum,ptr);
int get_user(local,ptr);
int __put_user(datum,ptr);
int __get_user(local,ptr);

用来存储或获取一个数据到或从用户空间的宏. 传送的字节数依赖 sizeof(*ptr). 常规的版本
调用 access_ok , 而常规版本(__put_user 和 __get_user) 假定 access_ok 已经被调用了.
#include <linux/capability.h>

定义各种 CAP_ 符号, 描述一个用户空间进程可有的能力.
int capable(int capability);

返回非零值如果进程有给定的能力.
#include <linux/wait.h>
typedef struct { /* ... */ } wait_queue_head_t;
void init_waitqueue_head(wait_queue_head_t *queue);
DECLARE_WAIT_QUEUE_HEAD(queue);

Linux 等待队列的定义类型. 一个 wait_queue_head_t 必须被明确在运行时使用
init_waitqueue_head 或者编译时使用 DEVLARE_WAIT_QUEUE_HEAD 进行初始化.
void wait_event(wait_queue_head_t q, int condition);
int wait_event_interruptible(wait_queue_head_t q, int condition);
int wait_event_timeout(wait_queue_head_t q, int condition, int time);
int wait_event_interruptible_timeout(wait_queue_head_t q, int condition,int time);

使进程在给定队列上睡眠, 直到给定条件值为真值.
void wake_up(struct wait_queue **q);
void wake_up_interruptible(struct wait_queue **q);
void wake_up_nr(struct wait_queue **q, int nr);
void wake_up_interruptible_nr(struct wait_queue **q, int nr);
void wake_up_all(struct wait_queue **q);
void wake_up_interruptible_all(struct wait_queue **q);
void wake_up_interruptible_sync(struct wait_queue **q);

唤醒在队列 q 上睡眠的进程. _interruptible 的形式只唤醒可中断的进程. 正常地, 只有一个互
斥等待者被唤醒, 但是这个行为可被 _nr 或者 _all 形式所改变. _sync 版本在返回之前不重新

调度 CPU.

```
#include <linux/sched.h>
set_current_state(int state);
```

设置当前进程的执行状态. TASK_RUNNING 意味着它已经运行, 而睡眠状态是 TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE.

```
void schedule(void);
```

选择一个可运行的进程从运行队列中. 被选中的进程可是当前进程或者另外一个.

```
typedef struct { /* ... */ } wait_queue_t;
init_waitqueue_entry(wait_queue_t *entry, struct task_struct *task);
```

wait_queue_t 类型用来放置一个进程到一个等待队列.

```
void prepare_to_wait(wait_queue_head_t *queue, wait_queue_t *wait, int state);
void prepare_to_wait_exclusive(wait_queue_head_t *queue, wait_queue_t *wait, int state);
void finish_wait(wait_queue_head_t *queue, wait_queue_t *wait);
```

帮忙函数, 可用来编码一个手工睡眠.

```
void sleep_on(wiat_queue_head_t *queue);
void interruptible_sleep_on(wiat_queue_head_t *queue);
```

老式的不推荐的函数, 它们无条件地使当前进程睡眠.

```
#include <linux/poll.h>
void poll_wait(struct file *filp, wait_queue_head_t *q, poll_table *p);
```

将当前进程放入一个等待队列, 不立刻调度. 它被设计来被设备驱动的 poll 方法使用.

```
int fasync_helper(struct inode *inode, struct file *filp, int mode, struct fasync_struct **fa);
```

一个"帮忙者", 来实现 fasync 设备方法. mode 参数是传递给方法的相同的值, 而 fa 指针指向一个设备特定的 fasync_struct *.

```
void kill_fasync(struct fasync_struct *fa, int sig, int band);
```

如果这个驱动支持异步通知, 这个函数可用来发送一个信号到登记在 fa 中的进程.

```
int nonseekable_open(struct inode *inode, struct file *filp);
loff_t no_llseek(struct file *file, loff_t offset, int whence);
```

nonseekable_open 应当在任何不支持移位的设备的 open 方法中被调用. 这样的设备应当使用 no_llseek 作为它们的 llseek 方法.

6.6. 在一个设备文件上的存取控制 起始页 第 7 章 时间, 延时, 和延后工作

第 7 章 时间, 延时, 和延后工作

上一页 下一页

第 7 章 时间, 延时, 和延后工作

目录

到此, 我们知道了如何编写一个全特性字符模块的基本知识. 真实世界的驱动, 然而, 需要做比实现控制一个设备的操作更多的事情; 它们不得不处理诸如定时, 内存管理, 硬件存取, 等更多. 幸运的是, 内核输出了许多设施来简化驱动编写者的任务. 在下几章中, 我们将描述一些你可使用的内核资源. 这一章引路, 通过描述定时问题是如何阐述. 处理时间问题包括下列任务, 按照复杂度上升的顺序:

- 测量时间流失和比较时间
- 知道当前时间
- 指定时间量的延时操作
- 调度异步函数在之后的时间发生

7.1. 测量时间流失

内核通过定时器中断来跟踪时间的流动. 中断在第 10 章详细描述.

定时器中断由系统定时硬件以规律地间隔产生; 这个间隔在启动时由内核根据 HZ 值来编程, HZ 是一个体系依赖的值, 在 <linux/param.h>中定义或者它所包含的一个子平台文件中. 在发布的内核源码中的缺省值在真实硬件上从 50 到 1200 嘀哒每秒, 在软件模拟器中往下到 24. 大部分平台运行在 100 或者 1000 中断每秒; 流行的 x86 PC 缺省是 1000, 尽管它在以前版本上(向上直到并且包括 2.4)常常是 100. 作为一个通用的规则, 即便如果你知道 HZ 的值, 在编程时你应当从不依赖这个特定值.

可能改变 HZ 的值, 对于那些要系统有一个不同的时钟中断频率的人. 如果你在头文件中改变 HZ 的值, 你需要使用新的值重编译内核和所有的模块. 如果你愿意付出额外的时间中断的代价来获得你的目标, 你可能想提升 HZ 来得到你的异步任务的更细粒度的精度. 实际上, 提升 HZ 到 1000 在使用 2.4 或 2.2 内核版本的 x86 工业系统中是相当普遍的. 但是, 对于当前版本, 最好的方法是保持 HZ 的缺省值, 由于我们完全信任内核开发者, 他们肯定已经选择了最好的值. 另外, 一些内部计算当前实现为只为从 12 到 1535 范围的 HZ (见 <linux/timex.h> 和 RFC-1589).

每次发生一个时钟中断, 一个内核计数器的值递增. 这个计数器在系统启动时初始化为 0, 因此它代表从最后一次启动以来的时钟嘀哒的数目. 这个计数器是一个 64-位 变量(即便在 32-位的体系上) 并且称为 jiffies_64. 但是, 驱动编写者正常地存取 jiffies 变量, 一个 unsigned long, 或者和 jiffies_64 是同一个或者它的低有效位. 使用 jiffies 常常是首选, 因为它更快, 并且再所有的体系上存取 64-位的 jiffies_64 值不必要是原子的.

除了低精度的内核管理的 jiffy 机制, 一些 CPU 平台特有一个高精度的软件可读的计数器. 尽管它的实际使用有些在各个平台不同, 它有时是一个非常有力的工具.

7.1.1. 使用 jiffies 计数器

这个计数器和来读取它的实用函数位于 <linux/jiffies.h>, 尽管你会常常只是包含 <linux/sched.h>, 它会自动地将 jiffies.h 拉进来. 不用说, jiffies 和 jiffies_64 必须当作只读的.

无论何时你的代码需要记住当前的 jiffies 值, 可以简单地存取这个 unsigned long 变量, 它被声明做 volatile 来告知编译器不要优化内存读. 你需要读取当前的计数器, 无论何时你的代码需要计算一个将来的时间戳, 如下面例子所示:

```
#include <linux/jiffies.h>
unsigned long j, stamp_1, stamp_half, stamp_n;

j = jiffies; /* read the current value */
stamp_1 = j + HZ; /* 1 second in the future */
```

```
stamp_half = j + HZ/2; /* half a second */
stamp_n = j + n * HZ / 1000; /* n milliseconds */
```

这个代码对于 jiffies 回绕没有问题, 只要不同的值以正确的方式进行比较. 尽管在 32-位 平台上当 HZ 是 1000 时, 计数器只是每 50 天回绕一次, 你的代码应当准备面对这个事件. 为比较你的被缓存的值(象上面的 stamp_1) 和当前值, 你应当使用下面一个宏定义:

```
#include <linux/jiffies.h>
int time_after(unsigned long a, unsigned long b);
int time_before(unsigned long a, unsigned long b);
int time_after_eq(unsigned long a, unsigned long b);
int time_before_eq(unsigned long a, unsigned long b);
```

第一个当 a, 作为一个 jiffies 的快照, 代表 b 之后的一个时间时, 取值为真, 第二个当 时间 a 在时间 b 之前时取值为真, 以及最后 2 个比较"之后或相同"和"之前或相同". 这个代码工作通过转换这个值为 signed long, 减它们, 并且比较结果. 如果你需要以一种安全的方式知道 2 个 jiffies 实例之间的差, 你可以使用同样的技巧: diff = (long)t2 - (long)t1;.

你可以转换一个 jiffies 差为毫秒, 一般地通过:

```
msec = diff * 1000 / HZ;
```

有时, 但是, 你需要与用户空间程序交换时间表示, 它们打算使用 struct timeval 和 struct timespec 来表示时间. 这 2 个结构代表一个精确的时间量, 使用 2 个成员: seconds 和 microseconds 在旧的流行的 struct timeval 中使用, seconds 和 nanoseconds 在新的 struct timespec 中使用. 内核输出 4 个帮助函数来转换以 jiffies 表达的时间值, 到和从这些结构:

```
#include <linux/time.h>
unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
```

存取这个 64-位 jiffy 计数值不象存取 jiffies 那样直接. 而在 64-位 计算机体系上, 这 2 个变量实际上是一个, 存取这个值对于 32-位 处理器不是原子的. 这意味着你可能读到错误的值如果这个变量的两半在你正在读取它们时被更新. 极不可能你会需要读取这个 64-位 计数器, 但是万一你需要, 你会高兴地得知内核输出了一个特别地帮助函数, 为你完成正确地加锁:

```
#include <linux/jiffies.h>
u64 get_jiffies_64(void);
```

在上面的原型中, 使用了 u64 类型. 这是一个定义在 <linux/types.h> 中的类型, 在 11 章中讨论, 并且

表示一个 unsigned 64-位 类型.

如果你在奇怪 32-位 平台如何同时更新 32-位 和 64-位 计数器, 读你的平台的连接脚本(查找一个文件, 它的名子匹配 valinux*.lds*). 在那里, jiffies 符号被定义来存取这个 64-位 值的低有效字, 根据平台是小端或者大端. 实际上, 同样的技巧也用在 64-位 平台上, 因此这个 unsigned long 和 u64 变量在同一个地址被存取.

最后, 注意实际的时钟频率几乎完全对用户空间隐藏. 宏 HZ 一直扩展为 100 当用户空间程序包含 param.h, 并且每个报告给用户空间的计数器都对应地被转换. 这应用于 clock(3), times(2), 以及任何相关的函数. 对 HZ 值的用户可用的唯一证据是时钟中断多快发生, 如在 /proc/interrupts 所显示的. 例如, 你可以获得 HZ, 通过用在 /proc/uptime 中报告的系统 uptime 除这个计数值.

7.1.2. 处理器特定的寄存器

如果你需要测量非常短时间间隔, 或者你需要非常高精度, 你可以借助平台依赖的资源, 一个要精度不要移植性的选择.

在现代处理器中, 对于经验性能数字的迫切需求被大部分 CPU 设计中内在的指令定时不确定性所阻碍, 这是由于缓存内存, 指令调度, 以及分支预测引起. 作为回应, CPU 制造商引入一个方法来计数时钟周期, 作为一个容易并且可靠的方法来测量时间流失. 因此, 大部分现代处理器包含一个计数器寄存器, 它在每个时钟周期固定地递增一次. 现在, 资格时钟计数器是唯一可靠的方法来进行高精度的时间管理任务.

细节每个平台不同: 这个寄存器可以或者不可以从用户空间可读, 它可以或者不可以写, 并且它可能是 64 或者 32 位宽. 在后一种情况, 你必须准备处理溢出, 就象我们处理 jiffy 计数器一样. 这个寄存器甚至可能对你的平台来说不存在, 或者它可能被硬件设计者在一个外部设备实现, 如果 CPU 缺少这个特性并且你在使用一个特殊用途的计算机.

无论是否寄存器可以被清零, 我们强烈不鼓励复位它, 即便当硬件允许时. 毕竟, 在任何给定时间你可能不是这个计数器的唯一用户; 在一些支持 SMP 的平台上, 例如, 内核依赖这样一个计数器来在处理器之间同步. 因为你可以一直测量各个值的差, 只要差没有超过溢出时间, 你可以通过修改它的当前值来做这个事情不用声明独自拥有这个寄存器.

最有名的计数器寄存器是 TSC (timestamp counter), 在 x86 处理器中随 Pentium 引入的并且在所有从那之后的 CPU 中出现 -- 包括 x86_64 平台. 它是一个 64-位 寄存器计数 CPU 的时钟周期; 它可从内核和用户空间读取.

在包含了 <asm/msr.h> (一个 x86-特定的头文件, 它的名子代表"machine-specific registers"), 你可使用一个这些宏:

```
rdtsc(low32,high32);
rdtscl(low32);
```

```
rdtscll(var64);
```

第一个宏自动读取 64-位 值到 2 个 32-位 变量; 下一个("read low half") 读取寄存器的低半部到一个 32-位 变量, 丢弃高半部; 最后一个读 64-位 值到一个 long long 变量, 由此得名. 所有这些宏存储数值到它们的参数中.

对大部分的 TSC 应用, 读取这个计数器的的低半部足够了. 一个 1-GHz 的 CPU 只在每 4.2 秒溢出一次, 因此你不会需要处理多寄存器变量, 如果你在使用的时间流失确定地使用更少时间. 但是, 随着 CPU 频率不断上升以及定时需求的提高, 将来你会几乎可能需要常常读取 64-位 计数器.

作为一个只使用寄存器低半部的例子, 下面的代码行测量了指令自身的执行:

```
unsigned long ini, end;
rdtscl(ini); rdtscl(end);
printk("time lapse: %li\n", end - ini);
```

一些其他的平台提供相似的功能, 并且内核头文件提供一个体系独立的功能, 你可用来代替 rdtsc. 它称为 get_cycles, 定义在 <asm/timex.h>(由 <linux/timex.h> 包含). 它的原型是:

```
#include <linux/timex.h>
cycles_t get_cycles(void);
```

这个函数为每个平台定义, 并且它一直返回 0 在没有周期-计数器寄存器的平台上. cycles_t 类型是一个合适的 unsigned 类型来持有读到的值.

不论一个体系独立的函数是否可用, 我们最好利用机会来展示一个内联汇编代码的例子. 为此, 我们实现一个 rdtscl 函数给 MIPS 处理器, 它与在 x86 上同样的方式工作.

拖尾的 nop 指令被要求来阻止编译器在 mfc0 之后马上存取指令中的目标寄存器. 这种内部锁在 RISC 处理器中是典型的, 并且编译器仍然可以在延迟时隙中调度有用的指令. 在这个情况中, 我们使用 nop 因为内联汇编对编译器是一个黑盒并且不会进行优化.[26]

```
#define rdtscl(dest) \
__asm__ __volatile__("mfc0 %0,$9; nop" : "=r" (dest))
```

有这个宏在, MIPS 处理器可以执行同样的代码, 如同前面为 x86 展示的一样的代码.

使用 gcc 内联汇编, 通用寄存器的分配留给编译器. 刚刚展示的这个宏使用 %0 作为"参数 0"的一个占位符, 之后它被指定为"任何用作输出(=)的寄存器(r)". 这个宏还声明输出寄存器必须对应 C 表达式 dest. 内联函数的语法是非常强大但是有些复杂, 特别对于那些有限制每个寄存器可以做什么的体系上(就是说, x86 家族). 这个用法在 gcc 文档中描述, 常常在 info 文档目录树中有.

本节已展示的这个简短的 C-代码片段已在一个 K7-级 x86 处理器 和一个 MIPS VR4181 (使用刚刚描述过的宏)上运行. 前者报告了一个 11 个时钟嘀哒的时间流失而后者只是 2 个时钟嘀哒. 小的数字是期望的, 因为 RISC 处理器常常每个时钟周期执行一条指令.

有另一个关于时戳计数器的事情值得知道: 它们在一个 SMP 系统中不必要跨处理器同步. 为保证得到一个一致的值, 你应当为查询这个计数器的代码禁止抢占.

---

[26] 我们在 MIPS 上建立这例子, 因为大部分的 MIPS 处理器特有一个 32-位 计数器作为它们内部"协处理器 0"的寄存器 9. 为存取这个寄存器, 仅仅从内核空间可读, 你可以定义下列的宏来执行一条"从协处理器 0 转移"的汇编指令:

上一页 6.7. 快速参考 起始页 下一页 7.2. 获知当前时间

7.2. 获知当前时间

内核代码能一直获取一个当前时间的表示, 通过查看 jifies 的值. 常常地, 这个值只代表从最后一次启动以来的时间, 这个事实对驱动来说无关, 因为它的生命周期受限于系统的 uptime. 如所示, 驱动可以使用 jiffies 的当前值来计算事件之间的时间间隔(例如, 在输入驱动中从单击中区分双击或者计算超时). 简单地讲, 查看 jiffies 几乎一直是足够的, 当你需要测量时间间隔. 如果你需要对短时间流失的非常精确的测量, 处理器特定的寄存器来帮忙了(尽管它们带来严重的移植性问题).

它是非常不可能一个驱动会需要知道墙上时钟时间, 以月, 天, 和小时来表达的; 这个信息常常只对用户程序需要, 例如 cron 和 syslogd. 处理真实世界的时间常常最好留给用户空间, 那里的 C 库提供了更好的支持; 另外, 这样的代码常常太策略相关以至于不属于内核. 有一个内核函数转变一个墙上时钟时间到一个 jiffies 值, 但是:

```
#include <linux/time.h>
unsigned long mktime (unsigned int year, unsigned int mon,
 unsigned int day, unsigned int hour,
 unsigned int min, unsigned int sec);
```

重复:直接在驱动中处理墙上时钟时间往往是一个在实现策略的信号, 并且应当因此而被置疑.

虽然你不会一定处理人可读的时间表示, 有时你需要甚至在内核空间中处理绝对时间. 为此, <linux/time.h> 输出了 do_gettimeofday 函数. 当被调用时, 它填充一个 struct timeval 指针 -- 和在 gettimeofday 系统调用中使用的相同 -- 使用类似的秒和毫秒值. do_gettimeofday 的原型是:

```
 #include <linux/time.h>
 void do_gettimeofday(struct timeval *tv);
```

这段源代码声明 do_gettimeofday 有" 接近毫秒的精度", 因为它询问时间硬件当前 jiffy 多大比例已经流失. 这个精度每个体系都不同, 但是, 因为它依赖实际使用中的硬件机制. 例如, 一些 m68knommu 处理器, Sun3 系统, 和其他 m68k 系统不能提供大于 jiffy 的精度. Pentium 系统, 另一方面, 提供了非常快速和精确的小于嘀哒的测量, 通过读取本章前面描述的时戳计数器.

当前时间也可用(尽管使用 jiffy 的粒度)来自 xtime 变量, 一个 struct timespec 值. 不鼓励这个变量的直接使用, 因为难以原子地同时存取这 2 个字段. 因此, 内核提供了实用函数 current_kernel_time:

```
#include <linux/time.h>
struct timespec current_kernel_time(void);
```

用来以各种方式获取当前时间的代码, 可以从由 O' Reilly 提供的 FTP 网站上的源码文件的 jit ("just in time") 模块获得. jit 创建了一个文件称为 /proc/currentime, 当读取时, 它以 ASCII 码返回下列项:

- 当前的 jiffies 和 jiffies_64 值, 以 16 进制数的形式.
- 如同 do_gettimeofday 返回的相同的当前时间.
- 由 current_kernel_time 返回的 timespec.

我们选择使用一个动态的 /proc 文件来保持样板代码为最小 -- 它不值得创建一整个设备只是返回一点儿文本信息.

这个文件连续返回文本行只要这个模块加载着; 每次 read 系统调用收集和返回一套数据, 为更好阅读而组织为 2 行. 无论何时你在少于一个时钟嘀哒内读多个数据集, 你将看到 do_gettimeofday 之间的差别, 它询问硬件, 并且其他值仅在时钟嘀哒时被更新.

```
phon% head -8 /proc/currentime
0x00bdbc1f 0x0000000100bdbc1f 1062370899.630126
 1062370899.629161488
0x00bdbc1f 0x0000000100bdbc1f 1062370899.630150
 1062370899.629161488
0x00bdbc20 0x0000000100bdbc20 1062370899.630208
 1062370899.630161336
0x00bdbc20 0x0000000100bdbc20 1062370899.630233
 1062370899.630161336
```

在上面的屏幕快照中, 由 2 件有趣的事情要注意. 首先, 这个 current_kernel_time 值, 尽管以纳秒来表示, 只有时钟嘀哒的粒度; do_gettimeofday 持续报告一个稍晚的时间但是不晚于下一个时钟嘀哒. 第二, 这个 64-位的 jiffies 计数器有 高 32-位字集合的最低有效位. 这是由于 INITIAL_JIFFIES 的缺省值, 在启动时间用来初始化计数器, 在启动时间后几分钟内强加一个低字溢出来帮助探测与这个刚好溢出相关的问题. 这个在计数器中的初始化偏好没有效果, 因为 jiffies 与墙上时钟时间无关. 在 /proc/uptime 中, 这里内核从计数器中抽取 uptime, 初始化偏好在转换前被去除.

7.3. 延后执行

设备驱动常常需要延后一段时间执行一个特定片段的代码, 常常允许硬件完成某个任务. 在这一节我们涉及许多不同的技术来获得延后. 每种情况的环境决定了使用哪种技术最好; 我们全都仔细检查它们, 并且指出每一个的长处和缺点.

一件要考虑的重要的事情是你需要的延时如何与时钟嘀哒比较, 考虑到 HZ 的跨各种平台的范围. 那种可靠地比时钟嘀哒长并且不会受损于它的粗粒度的延时, 可以利用系统时钟. 每个短延时典型地必须使用软件循环来实现. 在这 2 种情况中存在一个灰色地带. 在本章, 我们使用短语" long " 延时来指一个多 jiffy 延时, 在一些平台上它可以如同几个毫秒一样少, 但是在 CPU 和内核看来仍然是长的.

下面的几节讨论不同的延时, 通过采用一些长路径, 从各种直觉上不适合的方法到正确的方法. 我们选择这个途径因为它允许对内核相关定时方面的更深入的讨论. 如果你急于找出正确的代码, 只要快速浏览本节.

7.3.1. 长延时

偶尔地, 一个驱动需要延后执行相对长时间 -- 多于一个时钟嘀哒. 有几个方法实现这类延时; 我们从最简单的技术开始, 接着进入到高级些的技术.

7.3.1.1. 忙等待

如果你想延时执行多个时钟嘀哒, 允许在值中某些疏忽, 最容易的(尽管不推荐) 的实现是一个监视 jiffy 计数器的循环. 这种忙等待实现常常看来象下面的代码, 这里 j1 是 jiffies 的在延时超时的值:

```
while (time_before(jiffies, j1))
   cpu_relax();
```

对 cpu_relex 的调用使用了一个特定于体系的方式来说, 你此时没有在用处理器做事情. 在许多系统中它根本不做任何事; 在对称多线程(" 超线程") 系统中, 可能让出核心给其他线程. 在如何情况下, 无论何时有可能, 这个方法应当明确地避免. 我们展示它是因为偶尔你可能想运行这个代码来更好理解其他代码的内幕.

我们来看一下这个代码如何工作. 这个循环被保证能工作因为 jiffies 被内核头文件声明做易失性的, 并且因此, 在任何时候 C 代码寻址它时都从内存中获取. 尽管技术上正确(它如同设计的一样工

作), 这种忙等待严重地降低了系统性能. 如果你不配置你的内核为抢占操作, 这个循环在延时期间完全锁住了处理器; 调度器永远不会抢占一个在内核中运行的进程, 并且计算机看起来完全死掉直到时间 j1 到时. 这个问题如果你运行一个可抢占的内核时会改善一点, 因为, 除非这个代码正持有一个锁, 处理器的一些时间可以被其他用途获得. 但是, 忙等待在可抢占系统中仍然是昂贵的.

更坏的是, 当你进入循环时如果中断碰巧被禁止, jiffies 将不会被更新, 并且 while 条件永远保持真. 运行一个抢占的内核也不会有帮助, 并且你将被迫去击打大红按钮.

这个延时代码的实现可拿到, 如同下列的, 在 jit 模块中. 模块创建的这些 /proc/jit* 文件每次你读取一行文本就延时一整秒, 并且这些行保证是每个 20 字节. 如果你想测试忙等待代码, 你可以读取 /proc/jitbusy, 每当它返回一行它忙-循环一秒.

为确保读, 最多, 一行(或者几行) 一次从 /proc/jitbusy. 简化的注册 /proc 文件的内核机制反复调用 read 方法来填充用户请求的数据缓存. 因此, 一个命令, 例如 cat /proc/jitbusy, 如果它一次读取 4KB, 会冻住计算机 205 秒.

推荐的读 /proc/jitbusy 的命令是 dd bs=200 < /proc/jitbusy, 可选地同时指定块数目. 文件返回的每 20-字节 的行表示 jiffy 计数器已有的值, 在延时之前和延时之后. 这是一个例子运行在一个其他方面无负担的计算机上:

```
phon% dd bs=20 count=5 < /proc/jitbusy
 1686518 1687518
 1687519 1688519
 1688520 1689520
 1689520 1690520
 1690521 1691521
```

看来都挺好: 延时精确地是 1 秒 (1000 jiffies), 并且下一个 read 系统调用在上一个结束后立刻开始. 但是让我们看看在一个有大量 CPU-密集型进程在运行(并且是非抢占内核)的系统上会发生什么:

```
phon% dd bs=20 count=5 < /proc/jitbusy
 1911226 1912226
 1913323 1914323
 1919529 1920529
 1925632 1926632
 1931835 1932835
```

这里, 每个 read 系统调用精确地延时 1 秒, 但是内核耗费多过 5 秒在调度 dd 进程以便它可以发出下一个系统调用之前. 在一个多任务系统就期望是这样; CPU 时间在所有运行的进程间共享, 并且一个 CPU-密集型 进程有它的动态减少的优先级. (调度策略的讨论在本书范围之外).

上面所示的在负载下的测试已经在运行 load50 例子程序中进行了. 这个程序派生出许多什么都不

做的进程, 但是以一种 CPU-密集的方式来做. 这个程序是伴随本书的例子文件的一部分, 并且缺省是派生 50 个进程, 尽管这个数字可以在命令行指定. 在本章, 以及在本书其他部分, 使用一个有负载的系统的测试已经用 load50 在一个其他方面空闲的计算机上运行来进行了.

如果你在运行一个可抢占内核时重复这个命令, 你会发现没有显著差别在一个其他方面空闲的 CPU 上以及下面的在负载下的行为:

```
phon% dd bs=20 count=5 < /proc/jitbusy
 14940680 14942777
 14942778 14945430
 14945431 14948491
 14948492 14951960
 14951961 14955840
```

这里, 没有显著的延时在一个系统调用的末尾和下一个的开始之间, 但是单独的延时远远比 1 秒长: 直到 3.8 秒在展示的例子中并且随时间上升. 这些值显示了进程在它的延时当中被中断, 调度其他的进程. 系统调用之间的间隙不是唯一的这个进程的调度选项, 因此没有特别的延时在那里可以看到.

7.3.1.2. 让出处理器

如我们已见到的, 忙等待强加了一个重负载给系统总体; 我们乐意找出一个更好的技术. 想到的第一个改变是明确地释放 CPU 当我们对其不感兴趣时. 这是通过调用调度函数而实现地, 在 <linux/sched.h> 中声明:

```
while (time_before(jiffies, j1)) {
   schedule();
}
```

这个循环可以通过读取 /proc/jitsched 如同我们上面读 /proc/jitbusy 一样来测试. 但是, 还是不够优化. 当前进程除了释放 CPU 不作任何事情, 但是它保留在运行队列中. 如果它是唯一的可运行进程, 实际上它运行(它调用调度器来选择同一个进程, 进程又调用调度器, 这样下去). 换句话说, 机器的负载(在运行的进程的平均数) 最少是 1, 并且空闲任务 (进程号 0, 也称为对换进程, 由于历史原因) 从不运行. 尽管这个问题可能看来无关, 在计算机是空闲时运行空闲任务减轻了处理器工作负载, 降低它的温度以及提高它的生命期, 同时电池的使用时间如果这个计算机是你的膝上机. 更多的, 因为进程实际上在延时中执行, 它所耗费的时间都可以统计.

/proc/jitsched 的行为实际上类似于运行 /proc/jitbusy 在一个抢占的内核下. 这是一个例子运行, 在一个无负载的系统:

```
phon% dd bs=20 count=5 < /proc/jitsched
 1760205 1761207
```

```
1761209 1762211
1762212 1763212
1763213 1764213
1764214 1765217
```

有趣的是要注意每次 read 有时结束于等待比要求的多几个时钟嘀哒. 这个问题随着系统变忙会变得越来越坏, 并且驱动可能结束于等待长于期望的时间. 一旦一个进程使用调度来释放处理器, 无法保证进程将拿回处理器在任何时间之后. 因此, 以这种方式调用调度器对于驱动的需求不是一个安全的解决方法, 另外对计算机系统整体是不好的. 如果你在运行 load50 时测试 jitsched, 你可以见到关联到每一行的延时被扩充了几秒, 因为当定时超时的时候其他进程在使用 CPU .

7.3.1.3. 超时

到目前为止所展示的次优化的延时循环通过查看 jiffy 计数器而不告诉任何人来工作. 但是最好的实现一个延时的方法, 如你可能猜想的, 常常是请求内核为你做. 有 2 种方法来建立一个基于 jiffy 的超时, 依赖于是否你的驱动在等待其他的事件.

如果你的驱动使用一个等待队列来等待某些其他事件, 但是你也想确保它在一个确定时间段内运行, 可以使用 wait_event_timeout 或者 wait_event_interruptible_timeout:

```
#include <linux/wait.h>
long wait_event_timeout(wait_queue_head_t q, condition, long timeout);
long wait_event_interruptible_timeout(wait_queue_head_t q, condition, long timeout);
```

这些函数在给定队列上睡眠, 但是它们在超时(以 jiffies 表示)到后返回. 因此, 它们实现一个限定的睡眠不会一直睡下去. 注意超时值表示要等待的 jiffies 数, 不是一个绝对时间值. 这个值由一个有符号的数表示, 因为它有时是一个相减运算的结果, 尽管这些函数如果提供的超时值是负值通过一个 printk 语句抱怨. 如果超时到, 这些函数返回 0; 如果这个进程被其他事件唤醒, 它返回以 jiffies 表示的剩余超时值. 返回值从不会是负值, 甚至如果延时由于系统负载而比期望的值大.

/proc/jitqueue 文件展示了一个基于 wait_event_interruptible_timeout 的延时, 结果这个模块没有事件来等待, 并且使用 0 作为一个条件:

```
wait_queue_head_t wait;
init_waitqueue_head (&wait);
wait_event_interruptible_timeout(wait, 0, delay);
```

当读取 /proc/jitqueue 时, 观察到的行为近乎优化的, 即便在负载下:

```
phon% dd bs=20 count=5 < /proc/jitqueue
 2027024  2028024
 2028025  2029025
```

```
2029026  2030026
2030027  2031027
2031028  2032028
```

因为读进程当等待超时(上面是 dd)不在运行队列中, 你看不到表现方面的差别, 无论代码是否运行在一个抢占内核中.

wait_event_timeout 和 wait_event_interruptible_timeout 被设计为有硬件驱动存在, 这里可以用任何一种方法来恢复执行: 或者有人调用 wake_up 在等待队列上, 或者超时到. 这不适用于 jitqueue, 因为没人在等待队列上调用 wake_up (毕竟, 没有其他代码知道它), 因此这个进程当超时到时一直唤醒. 为适应这个特别的情况, 这里你想延后执行不等待特定事件, 内核提供了 schedule_timeout 函数, 因此你可以避免声明和使用一个多余的等待队列头:

```
#include <linux/sched.h>
signed long schedule_timeout(signed long timeout);
```

这里, timeout 是要延时的 jiffies 数. 返回值是 0 除非这个函数在给定的 timeout 流失前返回(响应一个信号). schedule_timeout 请求调用者首先设置当前的进程状态, 因此一个典型调用看来如此:

```
set_current_state(TASK_INTERRUPTIBLE);
schedule_timeout (delay);
```

前面的行(来自 /proc/jitschedto) 导致进程睡眠直到经过给定的时间. 因为 wait_event_interruptible_timeout 在内部依赖 schedule_timeout, 我们不会费劲显示 jitschedto 返回的数, 因为它们和 jitqueue 的相同. 再一次, 不值得有一个额外的时间间隔在超时到和你的进程实际被调度来执行之间.

在刚刚展示的例子中, 第一行调用 set_current_state 来设定一些东西以便调度器不会再次运行当前进程, 直到超时将它置回 TASK_RUNNING 状态. 为获得一个不可中断的延时, 使用 TASK_UNINTERRUPTIBLE 代替. 如果你忘记改变当前进程的状态, 调用 schedule_time 如同调用 shcedule(即, jitsched 的行为), 建立一个不用的定时器.

如果你想使用这 4 个 jit 文件在不同的系统情况下或者不同的内核, 或者尝试其他的方式来延后执行, 你可能想配置延时量当加载模块时通过设定延时模块参数.

7.3.2. 短延时

当一个设备驱动需要处理它的硬件的反应时间, 涉及到的延时常常是最多几个毫秒. 在这个情况下, 依靠时钟嘀哒显然不对路.

The kernel functions ndelay, udelay, and mdelay serve well for short delays, delaying execution for the specified number of nanoseconds, microseconds, or milliseconds respectively.* Their prototypes are: * The u

in udelay represents the Greek letter mu and stands for micro.

内核函数 ndelay, udelay, 以及 mdelay 对于短延时好用, 分别延后执行指定的纳秒数, 微秒数或者毫秒数. [27]它们的原型是:

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

这些函数的实际实现在 <asm/delay.h>, 是体系特定的, 并且有时建立在一个外部函数上. 每个体系都实现 udelay, 但是其他的函数可能或者不可能定义; 如果它们没有定义, <linux/delay.h> 提供一个缺省的基于 udelay 的版本. 在所有的情况中, 获得的延时至少是要求的值, 但可能更多; 实际上, 当前没有平台获得了纳秒的精度, 尽管有几个提供了次微秒的精度. 延时多于要求的值通常不是问题, 因为驱动中的短延时常常需要等待硬件, 并且这个要求是等待至少一个给定的时间流失.

udelay 的实现(可能 ndelay 也是) 使用一个软件循环基于在启动时计算的处理器速度, 使用整数变量 loos_per_jiffy. 如果你想看看实际的代码, 但是, 小心 x86 实现是相当复杂的一个因为它使用的不同的时间源, 基于什么 CPU 类型在运行代码.

为避免在循环计算中整数溢出, udelay 和 ndelay 强加一个上限给传递给它们的值. 如果你的模块无法加载和显示一个未解决的符号, __bad_udelay, 这意味着你使用太大的参数调用 udleay. 注意, 但是, 编译时检查只对常量进行并且不是所有的平台实现它. 作为一个通用的规则, 如果你试图延时几千纳秒, 你应当使用 udelay 而不是 ndelay; 类似地, 毫秒规模的延时应当使用 mdelay 完成而不是一个更细粒度的函数.

重要的是记住这 3 个延时函数是忙等待; 其他任务在时间流失时不能运行. 因此, 它们重复, 尽管在一个不同的规模上, jitbusy 的做法. 因此, 这些函数应当只用在没有实用的替代时.

有另一个方法获得毫秒(和更长)延时而不用涉及到忙等待. 文件 <linux/delay.h> 声明这些函数:

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds)
```

前 2 个函数使调用进程进入睡眠给定的毫秒数. 一个对 msleep 的调用是不可中断的; 你能确保进程睡眠至少给定的毫秒数. 如果你的驱动位于一个等待队列并且你想唤醒来打断睡眠, 使用 msleep_interruptible. 从 msleep_interruptible 的返回值正常地是 0; 如果, 但是, 这个进程被提早唤醒, 返回值是在初始请求睡眠周期中剩余的毫秒数. 对 ssleep 的调用使进程进入一个不可中断的睡眠给定的秒数.

通常, 如果你能够容忍比请求的更长的延时, 你应当使用 schedule_timeout, msleep, 或者 ssleep.

---

[27] udelay 中的 u 表示希腊字母 mu 并且代表 micro.

上一页 7.2. 获知当前时间 | 上一级 起始页 | 下一页 7.4. 内核定时器

7.4. 内核定时器

无论何时你需要调度一个动作以后发生, 而不阻塞当前进程直到到时, 内核定时器是给你的工具. 这些定时器用来调度一个函数在将来一个特定的时间执行, 基于时钟嘀哒, 并且可用作各类任务; 例如, 当硬件无法发出中断时, 查询一个设备通过在定期的间隔内检查它的状态. 其他的内核定时器的典型应用是关闭软驱马达或者结束另一个长期终止的操作. 在这种情况下, 延后来自 close 的返回将强加一个不必要(并且吓人的)开销在应用程序上. 最后, 内核自身使用定时器在几个情况下, 包括实现 schedule_timeout.

一个内核定时器是一个数据结构, 它指导内核执行一个用户定义的函数使用一个用户定义的参数在一个用户定义的时间. 这个实现位于 <linux/timer.h> 和 kernel/timer.c 并且在"内核定时器"一节中详细介绍.

被调度运行的函数几乎确定不会在注册它们的进程在运行时运行. 它们是, 相反, 异步运行. 直到现在, 我们在我们的例子驱动中已经做的任何事情已经在执行系统调用的进程上下文中运行. 当一个定时器运行时, 但是, 这个调度进程可能睡眠, 可能在不同的一个处理器上运行, 或者很可能已经一起退出.

这个异步执行类似当发生一个硬件中断时所发生的(这在第 10 章详细讨论). 实际上, 内核定时器被作为一个"软件中断"的结果而实现. 当在这种原子上下文运行时, 你的代码易受到多个限制. 定时器函数必须是原子的以所有的我们在第 1 章"自旋锁和原子上下文"一节中曾讨论过的方式, 但是有几个附加的问题由于缺少一个进程上下文而引起的. 我们将介绍这些限制; 在后续章节的几个地方将再次看到它们. 循环被调用因为原子上下文的规则必须认真遵守, 否则系统会发现自己陷入大麻烦中.

为能够被执行, 多个动作需要进程上下文. 当你在进程上下文之外(即, 在中断上下文), 你必须遵守下列规则:

- 没有允许存取用户空间. 因为没有进程上下文, 没有和任何特定进程相关联的到用户空间的途径.
- 这个 current 指针在原子态没有意义, 并且不能使用因为相关的代码没有和已被中断的进程的联系.
- 不能进行睡眠或者调度. 原子代码不能调用 schedule 或者某种 wait_event, 也不能调用任何其他可能睡眠的函数. 例如, 调用 kmalloc(..., GFP_KERNEL) 是违犯规则的. 旗标也必须不能使用因为它们可能睡眠.

内核代码能够告知是否它在中断上下文中运行, 通过调用函数 in_interrupt(), 它不要参数并且如果

处理器当前在中断上下文运行就返回非零, 要么硬件中断要么软件中断.

一个和 in_interrupt() 相关的函数是 in_atomic(). 它的返回值是非零无论何时调度被禁止; 这包含硬件和软件中断上下文以及任何持有自旋锁的时候. 在后一种情况, current 可能是有效的, 但是存取用户空间被禁止, 因为它能导致调度发生. 无论何时你使用 in_interrupt(), 你应当真正考虑是否 in_atomic 是你实际想要的. 2 个函数都在 <asm/hardirq.h> 中声明.

内核定时器的另一个重要特性是一个任务可以注册它本身在后面时间重新运行. 这是可能的, 因为每个 timer_list 结构在运行前从激活的定时器链表中去连接, 并且因此能够马上在其他地方被重新连接. 尽管反复重新调度相同的任务可能表现为一个无意义的操作, 有时它是有用的. 例如, 它可用作实现对设备的查询.

也值得了解在一个 SMP 系统, 定时器函数被注册时相同的 CPU 来执行, 为在任何可能的时候获得更好的缓存局部特性. 因此, 一个重新注册它自己的定时器一直运行在同一个 CPU.

不应当被忘记的定时器的一个重要特性是, 它们是一个潜在的竞争条件的源, 即便在一个单处理器系统. 这是它们与其他代码异步运行的一个直接结果. 因此, 任何被定时器函数存取的数据结构应当保护避免并发存取, 要么通过原子类型(在第 1 章的"原子变量"一节) 要么使用自旋锁(在第 9 章讨论).

7.4.1. 定时器 API

内核提供给驱动许多函数来声明, 注册, 以及去除内核定时器. 下列的引用展示了基本的代码块:

```
#include <linux/timer.h>
struct timer_list
{
      /* ... */
      unsigned long expires;
      void (*function)(unsigned long);
      unsigned long data;
};
void init_timer(struct timer_list *timer);
struct timer_list TIMER_INITIALIZER(_function, _expires, _data);

void add_timer(struct timer_list * timer);
int del_timer(struct timer_list * timer);
```

这个数据结构包含比曾展示过的更多的字段, 但是这 3 个是打算从定时器代码自身以外被存取的. 这个 expires 字段表示定时器期望运行的 jiffies 值; 在那个时间, 这个 function 函数被调用使用 data 作为一个参数. 如果你需要在参数中传递多项, 你可以捆绑它们作为一个单个数据结构并且传递一个转换为 unsiged long 的指针, 在所有支持的体系上的一个安全做法并且在内存管理中相当普遍(如

同 15 章中讨论的). expires 值不是一个 jiffies_64 项因为定时器不被期望在将来很久到时, 并且 64-位操作在 32-位平台上慢.

这个结构必须在使用前初始化. 这个步骤保证所有的成员被正确建立, 包括那些对调用者不透明的. 初始化可以通过调用 init_timer 或者 安排 TIMER_INITIALIZER 给一个静态结构, 根据你的需要. 在初始化后, 你可以改变 3 个公共成员在调用 add_timer 前. 为在到时前禁止一个已注册的定时器, 调用 del_timer.

jit 模块包括一个例子文件, /proc/jitimer (为 "just in timer"), 它返回一个头文件行以及 6 个数据行. 这些数据行表示当前代码运行的环境; 第一个由读文件操作产生并且其他的由定时器. 下列的输出在编译内核时被记录:

```
phon% cat /proc/jitimer
 time delta  inirq  pid  cpu command
 33565837  0  0  1269  0  cat
 33565847  10  1  1271  0  sh
 33565857  10  1  1273  0  cpp0
 33565867  10  1  1273  0  cpp0
 33565877  10  1  1274  0  cc1
 33565887  10  1  1274  0  cc1
```

在这个输出, time 字段是代码运行时的 jiffies 值, delta 是自前一行的 jiffies 改变值, inirq 是由 in_interrupt 返回的布尔值, pid 和 command 指的是当前进程, 以及 cpu 是在使用的 CPU 的数目(在单处理器系统上一直为 0).

如果你读 /proc/jitimer 当系统无负载时, 你会发现定时器的上下文是进程 0, 空闲任务, 它被称为"对换进程"只要由于历史原因.

用来产生 /proc/jitimer 数据的定时器是缺省每 10 jiffies 运行一次, 但是你可以在加载模块时改变这个值通过设置 tdelay (timer delay) 参数.

下面的代码引用展示了 jit 关于 jitimer 定时器的部分. 当一个进程试图读取我们的文件, 我们建立这个定时器如下:

```
unsigned long j = jiffies;
/* fill the data for our timer function */
data->prevjiffies = j;

data->buf = buf2;
data->loops = JIT_ASYNC_LOOPS;

/* register the timer */
```

```
data->timer.data = (unsigned long)data;
data->timer.function = jit_timer_fn;
data->timer.expires = j + tdelay; /* parameter */
add_timer(&data->timer);

/* wait for the buffer to fill */
wait_event_interruptible(data->wait, !data->loops);
```

The actual timer function looks like this:

```
void jit_timer_fn(unsigned long arg)
{
      struct jit_data *data = (struct jit_data *)arg;
      unsigned long j = jiffies;
      data->buf += sprintf(data->buf, "%9li %3li %i %6i %i %s\n",
                  j, j - data->prevjiffies, in_interrupt() ? 1 : 0,
                  current->pid, smp_processor_id(), current->comm);
      if (--data->loops) {
            data->timer.expires += tdelay;
            data->prevjiffies = j;
            add_timer(&data->timer);

      } else {
            wake_up_interruptible(&data->wait);
      }
}
```

定时器 API 包括几个比上面介绍的那些更多的功能. 下面的集合是完整的核提供的函数列表:

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

更新一个定时器的超时时间, 使用一个超时定时器的一个普通的任务(再一次, 关马达软驱定时器是一个典型例子). mod_timer 也可被调用于非激活定时器, 那里你正常地使用 add_timer.

```
int del_timer_sync(struct timer_list *timer);
```

如同 del_timer 一样工作, 但是还保证当它返回时, 定时器函数不在任何 CPU 上运行. del_timer_sync 用来避免竞争情况在 SMP 系统上, 并且在 UP 内核中和 del_timer 相同. 这个函数应当在大部分情况下比 del_timer 更首先使用. 这个函数可能睡眠如果它被从非原子上下文调用, 但是在其他情况下会忙等待. 要十分小心调用 del_timer_sync 当持有锁时; 如果这个定时器函数试图获得同一个锁, 系统会死锁. 如果定时器函数重新注册自己, 调用者必须首先确保这个重新注册不会发生; 这常常同设置一个" 关闭 "标志来实现, 这个标志被定时器函数检查.

```
int timer_pending(const struct timer_list * timer);
```

返回真或假来指示是否定时器当前被调度来运行, 通过调用结构的其中一个不透明的成员.

7.4.2. 内核定时器的实现

为了使用它们, 尽管你不会需要知道内核定时器如何实现, 这个实现是有趣的, 并且值得看一下它们的内部.

定时器的实现被设计来符合下列要求和假设:

- 定时器管理必须尽可能简化.
- 设计应当随着激活的定时器数目上升而很好地适应.
- 大部分定时器在几秒或最多几分钟内到时, 而带有长延时的定时器是相当少见.
- 一个定时器应当在注册它的同一个 CPU 上运行.

由内核开发者想出的解决方法是基于一个每-CPU 数据结构. 这个 timer_list 结构包括一个指针指向这个的数据结构在它的 base 成员. 如果 base 是 NULL, 这个定时器没有被调用运行; 否则, 这个指针告知哪个数据结构(并且, 因此, 哪个 CPU)运行它. 每-CPU 数据项在第 8 章的"每-CPU变量"一节中描述.

无论何时内核代码注册一个定时器(通过 add_timer 或者 mod_timer), 操作最终由 internal_add_timer 进行(在kernel/timer.c), 它依次添加新定时器到一个双向定时器链表在一个关联到当前 CPU 的"层叠表" 中.

这个层叠表象这样工作: 如果定时器在下一个 0 到 255 jiffies 内到时, 它被添加到专供短时定时器 256 列表中的一个上, 使用 expires 成员的最低有效位. 如果它在将来更久时间到时(但是在 16,384 jiffies 之前), 它被添加到基于 expires 成员的 9 - 14 位的 64 个列表中一个. 对于更长的定时器, 同样的技巧用在 15 - 20 位, 21 - 26 位, 和 27 - 31 位. 带有一个指向将来还长时间的 expires 成员的定时器(一些只可能发生在 64-位 平台上的事情) 被使用一个延时值 0xffffffff 进行哈希处理, 并且带有在过去到时的定时器被调度来在下一个时钟嘀哒运行. (一个已经到时的定时器模拟有时在高负载情况下被注册, 特别的是如果你运行一个可抢占内核).

当触发 __run_timers, 它为当前定时器嘀哒执行所有挂起的定时器. 如果 jiffies 当前是 256 的倍数, 这个函数还重新哈希处理一个下一级别的定时器列表到 256 短期列表, 可能地层叠一个或多个别的级别, 根据jiffies 的位表示.

这个方法, 虽然第一眼看去相当复杂, 在几个和大量定时器的时候都工作得很好. 用来管理每个激活定时器的时间独立于已经注册的定时器数目并且限制在几个对于它的 expires 成员的二进制表示的逻辑操作上. 关联到这个实现的唯一的开销是给 512 链表头的内存(256 短期链表和 4 组 64 更长时间的列表) -- 即 4 KB 的容量.

函数 __run_timers, 如同 /proc/jitimer 所示, 在原子上下文运行. 除了我们已经描述过的限制, 这个带

来一个有趣的特性: 定时器刚好在合适的时间到时, 甚至你没有运行一个可抢占内核, 并且 CPU 在内核空间忙. 你可以见到发生了什么当你在后台读 /proc/jitbusy 时以及在前台 /proc/jitimer. 尽管系统看来牢固地被锁住被这个忙等待系统调用, 内核定时器照样工作地不错.

但是, 记住, 一个内核定时器还远未完善, 因为它受累于 jitter 和 其他由硬件中断引起怪物, 还有其他定时器和其他异步任务. 虽然一个关联到简单数字 I/O 的定时器对于一个如同运行一个步进马达或者其他业余电子设备等简单任务是足够的, 它常常是不合适在工业环境中的生产系统. 对于这样的任务, 你将最可能需要依赖一个实时内核扩展.

上一页 7.3. 延后执行 | 上一级 起始页 | 下一页 7.5. Tasklets 机制

7.5. Tasklets 机制

另一个有关于定时问题的内核设施是 tasklet 机制. 它大部分用在中断管理(我们将在第 10 章再次见到).

tasklet 类似内核定时器在某些方面. 它们一直在中断时间运行, 它们一直运行在调度它们的同一个 CPU 上, 并且它们接收一个 unsigned long 参数. 不象内核定时器, 但是, 你无法请求在一个指定的时间执行函数. 通过调度一个 tasklet, 你简单地请求它在以后的一个由内核选择的时间执行. 这个行为对于中断处理特别有用, 那里硬件中断必须被尽快处理, 但是大部分的时间管理可以安全地延后到以后的时间. 实际上, 一个 tasket, 就象一个内核定时器, 在一个"软中断"的上下文中执行(以原子模式), 在使能硬件中断时执行异步任务的一个内核机制.

一个 tasklet 存在为一个时间结构, 它必须在使用前被初始化. 初始化能够通过调用一个特定函数或者通过使用某些宏定义声明结构:

```
#include <linux/interrupt.h>
struct tasklet_struct {
 /* ... */

void (*func)(unsigned long);
 unsigned long data;
};

void tasklet_init(struct tasklet_struct *t,
 void (*func)(unsigned long), unsigned long data);
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
```

tasklet 提供了许多有趣的特色:

- 一个 tasklet 能够被禁止并且之后被重新使能; 它不会执行直到它被使能与被禁止相同的的次数.
- 如同定时器, 一个 tasklet 可以注册它自己.
- 一个 tasklet 能被调度来执行以正常的优先级或者高优先级. 后一组一直是首先执行.
- taslet 可能立刻运行, 如果系统不在重载下, 但是从不会晚于下一个时钟嘀哒.
- 一个 tasklet 可能和其他 tasklet 并发, 但是对它自己是严格地串行的 -- 同样的 tasklet 从不同时运行在超过一个处理器上. 同样, 如已经提到的, 一个 tasklet 常常在调度它的同一个 CPU 上

运行.

jit 模块包括 2 个文件, /proc/jitasklet 和 /proc/jitasklethi, 它返回和在"内核定时器"一节中介绍过的 /proc/jitimer 同样的数据. 当你读其中一个文件时, 你取回一个 header 和 sixdata 行. 第一个数据行描述了调用进程的上下文, 并且其他的行描述了一个 tasklet 过程连续运行的上下文. 这是一个在编译一个内核时的运行例子:

```
phon% cat /proc/jitasklet
 time delta inirq pid cpu command
 6076139 0 0 4370 0 cat
 6076140 1 1 4368 0 cc1
 6076141 1 1 4368 0 cc1
 6076141 0 1 2 0 ksoftirqd/0
 6076141 0 1 2 0 ksoftirqd/0
 6076141 0 1 2 0 ksoftirqd/0
```

如同由上面数据所确定的, tasklet 在下一个时间嘀哒内运行只要 CPU 在忙于运行一个进程, 但是它立刻被运行当 CPU 处于空闲. 内核提供了一套 ksoftirqd 内核线程, 每个 CPU 一个, 只是来运行 "软件中断" 处理, 就像 tasklet_action 函数. 因此, tasklet 的最后 3 个运行在关联到 CPU 0 的 ksoftirqd 内核线程的上下文中发生. jitasklethi 的实现使用了一个高优先级 tasklet, 在马上要来的函数列表中解释.

jit 中实现 /proc/jitasklet 和 /proc/jittasklethi 的实际代码与 /proc/jitimer 的实现代码几乎是一致的, 但是它使用 tasklet 调用代替那些定时器. 下面的列表详细展开了 tasklet 结构已被初始化后的内核对 tasklet 的接口:

```
void tasklet_disable(struct tasklet_struct *t);
```

这个函数禁止给定的 tasklet. tasklet 可能仍然被 tasklet_schedule 调度, 但是它的执行被延后直到这个 tasklet 被再次使能. 如果这个 tasklet 当前在运行, 这个函数忙等待直到这个tasklet退出; 因此, 在调用 tasklet_disable 后, 你可以确保这个 tasklet 在系统任何地方都不在运行.

```
void tasklet_disable_nosync(struct tasklet_struct *t);
```

禁止这个 tasklet, 但是没有等待任何当前运行的函数退出. 当它返回, 这个 tasklt 被禁止并且不会在以后被调度直到重新使能, 但是它可能仍然运行在另一个 CPU 当这个函数返回时.

```
void tasklet_enable(struct tasklet_struct *t);
```

使能一个之前被禁止的 tasklet. 如果这个 tasklet 已经被调度, 它会很快运行. 一个对 tasklet_enable 的调用必须匹配每个对 tasklet_disable 的调用, 因为内核跟踪每个 tasklet 的"禁止次数".

```
void tasklet_schedule(struct tasklet_struct *t);
```

调度 tasklet 执行. 如果一个 tasklet 被再次调度在它有机会运行前, 它只运行一次. 但是, 如果

他在运行中被调度, 它在完成后再次运行; 这保证了在其他事件被处理当中发生的事件收到应有的注意. 这个做法也允许一个 tasklet 重新调度它自己.

void tasklet_hi_schedule(struct tasklet_struct *t);

调度 tasklet 在更高优先级执行. 当软中断处理运行时, 它处理高优先级 tasklet 在其他软中断之前, 包括"正常的" tasklet. 理想地, 只有具有低响应周期要求(例如填充音频缓冲)应当使用这个函数, 为避免其他软件中断处理引入的附加周期. 实际上, /proc/jitasklethi 没有显示可见的与 /proc/jitasklet 的区别.

void tasklet_kill(struct tasklet_struct *t);

这个函数确保了这个 tasklet 没被再次调度来运行; 它常常被调用当一个设备正被关闭或者模块卸载时. 如果这个 tasklet 被调度来运行, 这个函数等待直到它已执行. 如果这个 tasklet 重新调度它自己, 你必须阻止在调用 tasklet_kill 前它重新调度它自己, 如同使用 del_timer_sync.

tasklet 在 kernel/softirq.c 中实现. 2 个 tasklet 链表(正常和高优先级)被声明为每-CPU数据结构, 使用和内核定时器相同的 CPU-亲和 机制. 在 tasklet 管理中的数据结构是简单的链表, 因为 tasklet 没有内核定时器的分类请求.

上一页 | 上一级 | 下一页
7.4. 内核定时器 | 起始页 | 7.6. 工作队列

7.6. 工作队列

工作队列是, 表面上看, 类似于 taskets; 它们允许内核代码来请求在将来某个时间调用一个函数. 但是, 有几个显著的不同在这 2 个之间, 包括:

- tasklet 在软件中断上下文中运行的结果是所有的 tasklet 代码必须是原子的. 相反, 工作队列函数在一个特殊内核进程上下文运行; 结果, 它们有更多的灵活性. 特别地, 工作队列函数能够睡眠.
- tasklet 常常在它们最初被提交的处理器上运行. 工作队列以相同地方式工作, 缺省地.
- 内核代码可以请求工作队列函数被延后一个明确的时间间隔.

两者之间关键的不同是 tasklet 执行的很快, 短时期, 并且在原子态, 而工作队列函数可能有高周期但是不需要是原子的. 每个机制有它适合的情形.

工作队列有一个 struct workqueue_struct 类型, 在 <linux/workqueue.h> 中定义. 一个工作队列必须明确的在使用前创建, 使用一个下列的 2 个函数:

```
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
```

每个工作队列有一个或多个专用的进程("内核线程"), 它运行提交给这个队列的函数. 如果你使用 create_workqueue, 你得到一个工作队列它有一个专用的线程在系统的每个处理器上. 在很多情况下, 所有这些线程是简单的过度行为; 如果一个单个工作者线程就足够, 使用 create_singlethread_workqueue 来代替创建工作队列

提交一个任务给一个工作队列, 你需要填充一个 work_struct 结构. 这可以在编译时完成, 如下:

```
DECLARE_WORK(name, void (*function)(void *), void *data);
```

这里 name 是声明的结构名称, function 是从工作队列被调用的函数, 以及 data 是一个传递给这个函数的值. 如果你需要建立 work_struct 结构在运行时, 使用下面 2 个宏定义:

```
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

INIT_WORK 做更加全面的初始化结构的工作; 你应当在第一次建立结构时使用它.

PREPARE_WORK 做几乎同样的工作, 但是它不初始化用来连接 work_struct 结构到工作队列的指针. 如果有任何的可能性这个结构当前被提交给一个工作队列, 并且你需要改变这个队列, 使用 PREPARE_WORK 而不是 INIT_WORK.

有 2 个函数来提交工作给一个工作队列:

int queue_work(struct workqueue_struct *queue, struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue, struct work_struct *work, unsigned long delay);

每个都添加工作到给定的队列. 如果使用 queue_delay_work, 但是, 实际的工作没有进行直到至少 delay jiffies 已过去. 从这些函数的返回值是 0 如果工作被成功加入到队列; 一个非零结果意味着这个 work_struct 结构已经在队列中等待, 并且第 2 次没有加入.

在将来的某个时间, 这个工作函数将被使用给定的 data 值来调用. 这个函数将在工作者线程的上下文运行, 因此它可以睡眠如果需要 -- 尽管你应当知道这个睡眠可能怎样影响提交给同一个工作队列的其他任务. 这个函数不能做的是, 但是, 是存取用户空间. 因为它在一个内核线程中运行, 完全没有用户空间来存取.

如果你需要取消一个挂起的工作队列入口, 你可以调用:

int cancel_delayed_work(struct work_struct *work);

返回值是非零如果这个入口在它开始执行前被取消. 内核保证给定入口的执行不会在调用 cancel_delay_work 后被初始化. 如果 cancel_delay_work 返回 0, 但是, 这个入口可能已经运行在一个不同的处理器, 并且可能仍然在调用 cancel_delayed_work 后在运行. 要绝对确保工作函数没有在 cancel_delayed_work 返回 0 后在任何地方运行, 你必须跟随这个调用来调用:

void flush_workqueue(struct workqueue_struct *queue);

在 flush_workqueue 返回后, 没有在这个调用前提交的函数在系统中任何地方运行.

当你用完一个工作队列, 你可以去掉它, 使用:

void destroy_workqueue(struct workqueue_struct *queue);

7.6.1. 共享队列

一个设备驱动, 在许多情况下, 不需要它自己的工作队列. 如果你只偶尔提交任务给队列, 简单地使用内核提供的共享的, 缺省的队列可能更有效. 如果你使用这个队列, 但是, 你必须明白你将和别的在共享它. 从另一个方面说, 这意味着你不应当长时间独占队列(无长睡眠), 并且可能要更长时间它们轮到处理器.

jiq ("just in queue") 模块输出 2 个文件来演示共享队列的使用. 它们使用一个单个 work_struct structure, 这个结构这样建立:

```
static struct work_struct jiq_work;
   /* this line is in jiq_init() */
 INIT_WORK(&jiq_work, jiq_print_wq, &jiq_data);
```

当一个进程读 /proc/jiqwq, 这个模块不带延迟地初始化一系列通过共享的工作队列的路线.

```
int schedule_work(struct work_struct *work);
```

注意, 当使用共享队列时使用了一个不同的函数; 它只要求 work_struct 结构作为一个参数. 在 jiq 中的实际代码看来如此:

```
prepare_to_wait(&jiq_wait, &wait, TASK_INTERRUPTIBLE);
schedule_work(&jiq_work);
schedule();
finish_wait(&jiq_wait, &wait);
```

这个实际的工作函数打印出一行就象 jit 模块所作的, 接着, 如果需要, 重新提交这个 work_structcture 到工作队列中. 在这是 jiq_print_wq 全部:

```
static void jiq_print_wq(void *ptr)
{
      struct clientdata *data = (struct clientdata *) ptr;

      if (! jiq_print (ptr))
            return;

      if (data->delay)
            schedule_delayed_work(&jiq_work, data->delay);
      else
            schedule_work(&jiq_work);
}
```

如果用户在读被延后的设备 (/proc/jiqwqdelay), 这个工作函数重新提交它自己在延后的模式, 使用 schedule_delayed_work:

```
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
```

如果你看从这 2 个设备的输出, 它看来如:

```
% cat /proc/jiqwq
 time  delta preempt  pid cpu command
 1113043  0  0  7  1 events/1
 1113043  0  0  7  1 events/1
 1113043  0  0  7  1 events/1
 1113043  0  0  7  1 events/1
 1113043  0  0  7  1 events/1
% cat /proc/jiqwqdelay
 time  delta preempt  pid cpu command
 1122066  1  0  6  0 events/0

1122067  1  0  6  0 events/0
 1122068  1  0  6  0 events/0
 1122069  1  0  6  0 events/0
 1122070  1  0  6  0 events/0
```

当 /proc/jiqwq 被读, 在每行的打印之间没有明显的延迟. 相反, 当 /proc/jiqwqdealy 被读时, 在每行之间有恰好一个 jiffy 的延时. 在每一种情况, 我们看到同样的进程名子被打印; 它是实现共享队列的内核线程的名子. CPU 号被打印在斜线后面; 我们从不知道当读 /proc 文件时哪个 CPU 会在运行, 但是这个工作函数之后将一直运行在同一个处理器.

如果你需要取消一个已提交给工作队列的工作入口, 你可以使用 cancel_delayed_work, 如上面所述. 刷新共享队列需要一个不同的函数, 但是:

```
void flush_scheduled_work(void);
```

因为你不知道别人谁可能使用这个队列, 你从不真正知道 flush_schduled_work 返回可能需要多长时间.

7.7. 快速参考

本章介绍了下面的符号.

7.7.1. 时间管理

```
#include <linux/param.h>
HZ
```

HZ 符号指定了每秒产生的时钟嘀哒的数目.

```
#include <linux/jiffies.h>
volatile unsigned long jiffies;
u64 jiffies_64;
```

jiffies_64 变量每个时钟嘀哒时被递增; 因此, 它是每秒递增 HZ 次. 内核代码几乎常常引用 jiffies, 它在 64-位平台和 jiffies_64 相同并且在 32-位平台是它低有效的一半.

```
int time_after(unsigned long a, unsigned long b);
int time_before(unsigned long a, unsigned long b);
int time_after_eq(unsigned long a, unsigned long b);
int time_before_eq(unsigned long a, unsigned long b);
```

这些布尔表达式以一种安全的方式比较 jiffies, 没有万一计数器溢出的问题和不需要存取 jiffies_64.

```
u64 get_jiffies_64(void);
```

获取 jiffies_64 而没有竞争条件.

```
#include <linux/time.h>
unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
```

在 jiffies 和其他表示之间转换时间表示.

```
#include <asm/msr.h>
rdtsc(low32,high32);
rdtscl(low32);
rdtscll(var32);
```

x86-特定的宏定义来读取时戳计数器. 它们作为 2 半 32-位来读取, 只读低一半, 或者全部读到一个 long long 变量.

```
#include <linux/timex.h>
cycles_t get_cycles(void);
```

以平台独立的方式返回时戳计数器. 如果 CPU 没提供时戳特性, 返回 0.

```
#include <linux/time.h>
unsigned long mktime(year, mon, day, h, m, s);
```

返回自 Epoch 以来的秒数, 基于 6 个 unsigned int 参数.

```
void do_gettimeofday(struct timeval *tv);
```

返回当前时间, 作为自 Epoch 以来的秒数和微秒数, 用硬件能提供的最好的精度. 在大部分的平台这个解决方法是一个微秒或者更好, 尽管一些平台只提供 jiffies 精度.

```
struct timespec current_kernel_time(void);
```

返回当前时间, 以一个 jiffy 的精度.

7.7.2. 延迟

```
#include <linux/wait.h>
long wait_event_interruptible_timeout(wait_queue_head_t *q, condition, signed long timeout);
```

使当前进程在等待队列进入睡眠, 安装一个以 jiffies 表达的超时值. 使用 schedule_timeout(下面) 给不可中断睡眠.

```
#include <linux/sched.h>
signed long schedule_timeout(signed long timeout);
```

调用调度器, 在确保当前进程在超时到的时候被唤醒后. 调用者首先必须调用 set_curret_state 来使自己进入一个可中断的或者不可中断的睡眠状态.

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

引入一个整数纳秒, 微秒和毫秒的延迟. 获得的延迟至少是请求的值, 但是可能更多. 每个函数的参数必须不超过一个平台特定的限制(常常是几千).

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
```

使进程进入睡眠给定的毫秒数(或者秒, 如果使 ssleep).

7.7.3. 内核定时器

```
#include <asm/hardirq.h>
int in_interrupt(void);
int in_atomic(void);
```

返回一个布尔值告知是否调用代码在中断上下文或者原子上下文执行. 中断上下文是在一个进程上下文之外, 或者在硬件或者软件中断处理中. 原子上下文是当你不能调度一个中断上下文或者一个持有一个自旋锁的进程的上下文.

```
#include <linux/timer.h>
void init_timer(struct timer_list * timer);
struct timer_list TIMER_INITIALIZER(_function, _expires, _data);
```

这个函数和静态的定时器结构的声明是初始化一个 timer_list 数据结构的 2 个方法.

```
void add_timer(struct timer_list * timer);
```

注册定时器结构来在当前 CPU 上运行.

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

改变一个已经被调度的定时器结构的超时时间. 它也能作为一个 add_timer 的替代.

```
int timer_pending(struct timer_list * timer);
```

宏定义, 返回一个布尔值说明是否这个定时器结构已经被注册运行.

```
void del_timer(struct timer_list * timer);
void del_timer_sync(struct timer_list * timer);
```

从激活的定时器链表中去除一个定时器. 后者保证这定时器当前没有在另一个 CPU 上运行.

7.7.4. Tasklets 机制

```
#include <linux/interrupt.h>
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data);
```

前 2 个宏定义声明一个 tasklet 结构, 而 tasklet_init 函数初始化一个已经通过分配或其他方式获得的 tasklet 结构. 第 2 个 DECLARE 宏标识这个 tasklet 为禁止的.

```
void tasklet_disable(struct tasklet_struct *t);
void tasklet_disable_nosync(struct tasklet_struct *t);
```

```
void tasklet_enable(struct tasklet_struct *t);
```

禁止和使能一个 tasklet. 每个禁止必须配对一个使能(你可以禁止这个 tasklet 即便它已经被禁止). 函数 tasklet_disable 等待 tasklet 终止如果它在另一个 CPU 上运行. 这个非同步版本不采用这个额外的步骤.

```
void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);
```

调度一个 tasklet 运行, 或者作为一个"正常" tasklet 或者一个高优先级的. 当软中断被执行, 高优先级 tasklets 被首先处理, 而正常 tasklet 最后执行.

```
void tasklet_kill(struct tasklet_struct *t);
```

从激活的链表中去掉 tasklet, 如果它被调度执行. 如同 tasklet_disable, 这个函数可能在 SMP 系统中阻塞等待 tasklet 终止, 如果它当前在另一个 CPU 上运行.

7.7.5. 工作队列

```
#include <linux/workqueue.h>
struct workqueue_struct;
struct work_struct;
```

这些结构分别表示一个工作队列和一个工作入口.

```
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
void destroy_workqueue(struct workqueue_struct *queue);
```

创建和销毁工作队列的函数. 一个对 create_workqueue 的调用创建一个有一个工作者线程在系统中每个处理器上的队列; 相反, create_singlethread_workqueue 创建一个有一个单个工作者进程的工作队列.

```
DECLARE_WORK(name, void (*function)(void *), void *data);
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

声明和初始化工作队列入口的宏.

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue, struct work_struct *work, unsigned long delay);
```

从一个工作队列对工作进行排队执行的函数.

```
int cancel_delayed_work(struct work_struct *work);
void flush_workqueue(struct workqueue_struct *queue);
```

使用 cancel_delayed_work 来从一个工作队列中去除入口; flush_workqueue 确保没有工作队列

入口在系统中任何地方运行.

```
int schedule_work(struct work_struct *work);
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
void flush_scheduled_work(void);
```

使用共享队列的函数.

上一页 | 上一级 | 下一页
7.6. 工作队列 | 起始页 | 第 8 章 分配内存

第 8 章 分配内存

上一页 下一页

第 8 章 分配内存

目录

至此, 我们已经使用 kmalloc 和 kfree 来分配和释放内存. Linux 内核提供了更丰富的一套内存分配原语, 但是. 在本章, 我们查看在设备驱动中使用内存的其他方法和如何优化你的系统的内存资源. 我们不涉及不同的体系实际上如何管理内存. 模块不牵扯在分段, 分页等问题中, 因为内核提供一个统一的内存管理驱动接口. 另外, 我们不会在本章描述内存管理的内部细节, 但是推迟在 15 章.

8.1. kmalloc 的真实故事

kmalloc 分配引擎是一个有力的工具并且容易学习因为它对 malloc 的相似性. 这个函数快(除非它阻塞)并且不清零它获得的内存; 分配的区仍然持有它原来的内容.[28] 分配的区也是在物理内存中连续. 在下面几节, 我们详细讨论 kmalloc, 因此你能比较它和我们后来要讨论的内存分配技术.

8.1.1. flags 参数

记住 kmalloc 原型是:

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
```

给 kmalloc 的第一个参数是要分配的块的大小. 第 2 个参数, 分配标志, 非常有趣, 因为它以几个方式控制 kmalloc 的行为.

最一般使用的标志, GFP_KERNEL, 意思是这个分配((内部最终通过调用 __get_free_pages 来进行, 它是 GFP_ 前缀的来源) 代表运行在内核空间的进程而进行的. 换句话说, 这意味着调用函数是代表一个进程在执行一个系统调用. 使用 GFP_KENRL 意味着 kmalloc 能够使当前进程在少内存的情况下睡眠来等待一页. 一个使用 GFP_KERNEL 来分配内存的函数必须, 因此, 是可重入的并且不能在原子上下文中运行. 当当前进程睡眠, 内核采取正确的动作来定位一些空闲内存, 或者通过刷新缓存到磁盘或者交换出去一个用户进程的内存.

GFP_KERNEL 不一直是使用的正确分配标志; 有时 kmalloc 从一个进程的上下文的外部调用. 例如, 这类的调用可能发生在中断处理, tasklet, 和内核定时器中. 在这个情况下, 当前进程不应当被置为睡眠, 并且驱动应当使用一个 GFP_ATOMIC 标志来代替. 内核正常地试图保持一些空闲页以便来满足原子的分配. 当使用 GFP_ATOMIC 时, kmalloc 能够使用甚至最后一个空闲页. 如果这最后一个空闲页不存在, 但是, 分配失败.

其他用来代替或者增添 GFP_KERNEL 和 GFP_ATOMIC 的标志, 尽管它们 2 个涵盖大部分设备驱动的需要. 所有的标志定义在 <linux/gfp.h>, 并且每个标志用一个双下划线做前缀, 例如 __GFP_DMA. 另外, 有符号代表常常使用的标志组合; 这些缺乏前缀并且有时被称为分配优先级. 后者包括:

GFP_ATOMIC

用来从中断处理和进程上下文之外的其他代码中分配内存. 从不睡眠.

GFP_KERNEL

内核内存的正常分配. 可能睡眠.

GFP_USER

用来为用户空间页来分配内存; 它可能睡眠.

GFP_HIGHUSER

如同 GFP_USER, 但是从高端内存分配, 如果有. 高端内存在下一个子节描述.

GFP_NOIO
GFP_NOFS

这个标志功能如同 GFP_KERNEL, 但是它们增加限制到内核能做的来满足请求. 一个

GFP_NOFS 分配不允许进行任何文件系统调用, 而 GFP_NOIO 根本不允许任何 I/O 初始化. 它们主要地用在文件系统和虚拟内存代码, 那里允许一个分配睡眠, 但是递归的文件系统调用会是一个坏注意.

上面列出的这些分配标志可以是下列标志的相或来作为参数, 这些标志改变这些分配如何进行:

__GFP_DMA

这个标志要求分配在能够 DMA 的内存区. 确切的含义是平台依赖的并且在下面章节来解释.

__GFP_HIGHMEM

这个标志指示分配的内存可以位于高端内存.

__GFP_COLD

正常地, 内存分配器尽力返回"缓冲热"的页 -- 可能在处理器缓冲中找到的页. 相反, 这个标志请求一个"冷"页, 它在一段时间没被使用. 它对分配页作 DMA 读是有用的, 此时在处理器缓冲中出现是无用的. 一个完整的对如何分配 DMA 缓存的讨论看"直接内存存取"一节在第 1 章.

__GFP_NOWARN

这个很少用到的标志阻止内核来发出警告(使用 printk), 当一个分配无法满足.

__GFP_HIGH

这个标志标识了一个高优先级请求, 它被允许来消耗甚至被内核保留给紧急状况的最后的内存页.

__GFP_REPEAT
__GFP_NOFAIL
__GFP_NORETRY

这些标志修改分配器如何动作, 当它有困难满足一个分配. __GFP_REPEAT 意思是" 更尽力些尝试" 通过重复尝试 -- 但是分配可能仍然失败. __GFP_NOFAIL 标志告诉分配器不要失败; 它尽最大努力来满足要求. 使用 __GFP_NOFAIL 是强烈不推荐的; 可能从不会有有效的理由在一个设备驱动中使用它. 最后, __GFP_NORETRY 告知分配器立即放弃如果得不到请求的内存.

8.1.1.1. 内存区

__GFP_DMA 和 __GFP_HIGHMEM 都有一个平台相关的角色, 尽管对所有平台它们的使用都有效.

Linux 内核知道最少 3 个内存区: DMA-能够 内存, 普通内存, 和高端内存. 尽管通常地分配都发生于普通区, 设置这些刚刚提及的位的任一个请求从不同的区来分配内存. 这个想法是, 每个必须知道特殊内存范围(不是认为所有的 RAM 等同)的计算机平台将落入这个抽象中.

DMA-能够 的内存是位于一个优先的地址范围, 外设可以在这里进行 DMA 存取. 在大部分的健全的平台, 所有的内存都在这个区. 在 x86, DMA 区用在 RAM 的前 16 MB, 这里传统的 ISA 设备可以进行 DMA; PCI 设备没有这个限制.

高端内存是一个机制用来允许在 32-位 平台存取(相对地)大量内存. 如果没有首先设置一个特殊的映射这个内存无法直接从内核存取并且通常更难使用. 如果你的驱动使用大量内存, 但是, 如果它能够使用高端内存它将在大系统中工作的更好. 高端内存如何工作以及如何使用它的详情见第 1 章的"高端和低端内存"一节.

无论何时分配一个新页来满足一个内存分配请求, 内核都建立一个能够在搜索中使用的内存区的列表. 如果 __GFP_DMA 指定了, 只有 DMA 区被搜索: 如果在低端没有内存可用, 分配失败. 如果没有特别的标志存取, 普通和 DMA 内存都被搜索; 如果 __GFP_HIGHMEM 设置了, 所有的 3 个区都用来搜索一个空闲的页. (注意, 但是, kmalloc 不能分配高端内存.)

情况在非统一内存存取(NUMA)系统上更加复杂. 作为一个通用的规则, 分配器试图定位进行分配的处理器的本地的内存, 尽管有几个方法来改变这个行为.

内存区后面的机制在 mm/page_alloc.c 中实现, 而内存区的初始化在平台特定的文件中, 常常在 arch 目录树的 mm/init.c. 我们将在第 15 章再次讨论这些主题.

8.1.2. size 参数

内核管理系统的物理内存, 这些物理内存只是以页大小的块来使用. 结果是, kmalloc 看来非常不同于一个典型的用户空间 malloc 实现. 一个简单的, 面向堆的分配技术可能很快有麻烦; 它可能在解决页边界时有困难. 因而, 内核使用一个特殊的面向页的分配技术来最好地利用系统 RAM.

Linux 处理内存分配通过创建一套固定大小的内存对象池. 分配请求被这样来处理, 进入一个持有足够大的对象的池子并且将整个内存块递交给请求者. 内存管理方案是非常复杂, 并且细节通常不是全部设备驱动编写者都感兴趣的.

然而, 驱动开发者应当记住的一件事情是, 内核只能分配某些预定义的, 固定大小的字节数组. 如果你请求一个任意数量内存, 你可能得到稍微多于你请求的, 至多是 2 倍数量. 同样, 程序员应当记住 kmalloc 能够处理的最小分配是 32 或者 64 字节, 依赖系统的体系所使用的页大小.

kmalloc 能够分配的内存块的大小有一个上限. 这个限制随着体系和内核配置选项而变化. 如果你的代码是要完全可移植, 它不能指望可以分配任何大于 128 KB. 如果你需要多于几个 KB, 但是, 有个比 kmalloc 更好的方法来获得内存, 我们在本章后面描述.

[28] 在其他的之中, 这暗含着你应当明确地清零可能暴露给用户空间或者写入设备的内存; 否则, 你可能冒险将应当保密的信息透露出去.

上一页 下一页
7.7. 快速参考 起始页 8.2. 后备缓存

8.2. 后备缓存

一个设备驱动常常以反复分配许多相同大小的对象而结束. 如果内核已经维护了一套相同大小对象的内存池, 为什么不增加一些特殊的内存池给这些高容量的对象? 实际上, 内核确实实现了一个设施来创建这类内存池, 它常常被称为一个后备缓存. 设备驱动常常不展示这类的内存行为, 它们证明使用一个后备缓存是对的, 但是, 有例外; 在 Linux 2.6 中 USB 和 SCSI 驱动使用缓存.

Linux 内核的缓存管理者有时称为" slab 分配器". 因此, 它的功能和类型在 <linux/slab.h> 中声明. slab 分配器实现有一个 kmem_cache_t 类型的缓存; 使用一个对 kmem_cache_create 的调用来创建它们:

```
kmem_cache_t *kmem_cache_create(const char *name, size_t size,
 size_t offset,
 unsigned long flags,
 void (*constructor)(void *, kmem_cache_t *,
 unsigned long flags), void (*destructor)(void *, kmem_cache_t *, unsigned long flags));
```

这个函数创建一个新的可以驻留任意数目全部同样大小的内存区的缓存对象, 大小由 size 参数指定. name 参数和这个缓存关联并且作为一个在追踪问题时有用的管理信息; 通常, 它被设置为被缓存的结构类型的名子. 这个缓存保留一个指向 name 的指针, 而不是拷贝它, 因此驱动应当传递一个指向在静态存储中的名子的指针(常常这个名子只是一个文字字串). 这个名子不能包含空格.

offset 是页内的第一个对象的偏移; 它可被用来确保一个对被分配的对象的特殊对齐, 但是你最可能会使用 0 来请求缺省值. flags 控制如何进行分配并且是下列标志的一个位掩码:

SLAB_NO_REAP

设置这个标志保护缓存在系统查找内存时被削减. 设置这个标志通常是个坏主意; 重要的是避免不必要地限制内存分配器的行动自由.

SLAB_HWCACHE_ALIGN

这个标志需要每个数据对象被对齐到一个缓存行; 实际对齐依赖主机平台的缓存分布. 这个选项可以是一个好的选择, 如果在 SMP 机器上你的缓存包含频繁存取的项. 但是, 用来获得缓存行对齐的填充可以浪费可观的内存量.

SLAB_CACHE_DMA

这个标志要求每个数据对象在 DMA 内存区分配.

还有一套标志用来调试缓存分配; 详情见 mm/slab.c. 但是, 常常地, 在用来开发的系统中, 这些标志通过一个内核配置选项被全局性地设置

函数的 constructor 和 destructor 参数是可选函数(但是可能没有 destructor, 如果没有 constructor); 前者可以用来初始化新分配的对象, 后者可以用来"清理"对象在它们的内存被作为一个整体释放回给系统之前.

构造函数和析构函数会有用, 但是有几个限制你必须记住. 一个构造函数在分配一系列对象的内存时被调用; 因为内存可能持有几个对象, 构造函数可能被多次调用. 你不能假设构造函数作为分配一个对象的一个立即的结果而被调用. 同样地, 析构函数可能在以后某个未知的时间中调用, 不是立刻在一个对象被释放后. 析构函数和构造函数可能或不可能被允许睡眠, 根据它们是否被传递 SLAB_CTOR_ATOMIC 标志(这里 CTOR 是 constructor 的缩写).

为方便, 一个程序员可以使用相同的函数给析构函数和构造函数; slab 分配器常常传递 SLAB_CTOR_CONSTRUCTOR 标志当被调用者是一个构造函数.

一旦一个对象的缓存被创建, 你可以通过调用 kmem_cache_alloc 从它分配对象.

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
```

这里, cache 参数是你之前已经创建的缓存; flags 是你会传递给 kmalloc 的相同, 并且被参考如果 kmem_cache_alloc 需要出去并分配更多内存.

为释放一个对象, 使用 kmem_cache_free:

```
 void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

当驱动代码用完这个缓存, 典型地当模块被卸载, 它应当如下释放它的缓存:

```
 int kmem_cache_destroy(kmem_cache_t *cache);
```

这个销毁操作只在从这个缓存中分配的所有的对象都已返回给它时才成功. 因此, 一个模块应当检查从 kmem_cache_destroy 的返回值; 一个失败指示某类在模块中的内存泄漏(因为某些对象已被丢失.)

使用后备缓存的一方面益处是内核维护缓冲使用的统计. 这些统计可从 /proc/slabinfo 获得.

8.2.1. 一个基于 Slab 缓存的 scull: scullc

是时候给个例子了. scullc 是一个简化的 scull 模块的版本, 它只实现空设备 -- 永久的内存区. 不象 scull, 它使用 kmalloc, scullc 使用内存缓存. 量子的大小可在编译时和加载时修改, 但是不是在运行

时 -- 这可能需要创建一个新内存区, 并且我们不想处理这些不必要的细节.

scullc 使用一个完整的例子, 可用来试验 slab 分配器. 它区别于 scull 只在几行代码. 首先, 我们必须声明我们自己的 slab 缓存:

```
/* declare one cache pointer: use it for all devices */
kmem_cache_t *scullc_cache;
```

slab 缓存的创建以这样的方式处理(在模块加载时):

```
/* scullc_init: create a cache for our quanta */
scullc_cache = kmem_cache_create("scullc", scullc_quantum,
                        0, SLAB_HWCACHE_ALIGN, NULL, NULL); /* no ctor/dtor */

if (!scullc_cache)
{
      scullc_cleanup();
      return -ENOMEM;

}
```

这是它如何分配内存量子:

```
/* Allocate a quantum using the memory cache */
if (!dptr->data[s_pos])
{
      dptr->data[s_pos] = kmem_cache_alloc(scullc_cache, GFP_KERNEL);
      if (!dptr->data[s_pos])

            goto nomem;
      memset(dptr->data[s_pos], 0, scullc_quantum);
}
```

还有这些代码行释放内存:

```
for (i = 0; i < qset; i++)
      if (dptr->data[i])
            kmem_cache_free(scullc_cache, dptr->data[i]);
```

最后, 在模块卸载时, 我们不得不返回缓存给系统:

```
/* scullc_cleanup: release the cache of our quanta */
```

```
if (scullc_cache)
    kmem_cache_destroy(scullc_cache);
```

从 scull 到 scullc 的主要不同是稍稍的速度提升以及更好的内存使用. 因为量子从一个恰好是合适大小的内存片的池中分配, 它们在内存中的排列是尽可能的密集, 与 scull 量子的相反, 它带来一个不可预测的内存碎片.

8.2.2. 内存池

在内核中有不少地方内存分配不允许失败. 作为一个在这些情况下确保分配的方式, 内核开发者创建了一个已知为内存池(或者是 "mempool")的抽象. 一个内存池真实地只是一类后备缓存, 它尽力一直保持一个空闲内存列表给紧急时使用.

一个内存池有一个类型 mempool_t (在 <linux/mempool.h> 中定义); 你可以使用 mempool_create 创建一个:

```
mempool_t *mempool_create(int min_nr,
 mempool_alloc_t *alloc_fn,
 mempool_free_t *free_fn,
 void *pool_data);
```

min_nr 参数是内存池应当一直保留的最小数量的分配的对象. 实际的分配和释放对象由 alloc_fn 和 free_fn 处理, 它们有这些原型:

```
typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);
typedef void (mempool_free_t)(void *element, void *pool_data);
```

给 mempool_create 最后的参数 (pool_data) 被传递给 alloc_fn 和 free_fn.

如果需要, 你可编写特殊用途的函数来处理 mempool 的内存分配. 常常, 但是, 你只需要使内核 slab 分配器为你处理这个任务. 有 2 个函数 (mempool_alloc_slab 和 mempool_free_slab) 来进行在内存池分配原型和 kmem_cache_alloc 和 kmem_cache_free 之间的感应淬火. 因此, 设置内存池的代码常常看来如此:

```
cache = kmem_cache_create(. . .);
pool = mempool_create(MY_POOL_MINIMUM,mempool_alloc_slab, mempool_free_slab, cache);
```

一旦已创建了内存池, 可以分配和释放对象,使用:

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

当内存池创建了, 分配函数将被调用足够的次数来创建一个预先分配的对象池. 因此, 对 mempool_alloc 的调用试图从分配函数请求额外的对象; 如果那个分配失败, 一个预先分配的对象(如果有剩下的)被返回. 当一个对象被用 mempool_free 释放, 它保留在池中, 如果对齐预分配的对象数目小于最小量; 否则, 它将被返回给系统.

一个 mempool 可被重新定大小, 使用:

int mempool_resize(mempool_t *pool, int new_min_nr, int gfp_mask);

这个调用, 如果成功, 调整内存池的大小至少有 new_min_nr 个对象. 如果你不再需要一个内存池, 返回给系统使用:

void mempool_destroy(mempool_t *pool);

你编写返回所有的分配的对象, 在销毁 mempool 之前, 否则会产生一个内核 oops.

如果你考虑在你的驱动中使用一个 mempool, 请记住一件事: mempools 分配一块内存在一个链表中, 对任何真实的使用是空闲和无用的. 容易使用 mempools 消耗大量的内存. 在几乎每个情况下, 首选的可选项是不使用 mempool 并且代替以简单处理分配失败的可能性. 如果你的驱动有任何方法以不危害到系统完整性的方式来响应一个分配失败, 就这样做. 驱动代码中的 mempools 的使用应当少.

8.3. get_free_page 和其友

如果一个模块需要分配大块的内存, 它常常最好是使用一个面向页的技术. 请求整个页也有其他的优点, 这个在 15 章介绍.

为分配页, 下列函数可用:

get_zeroed_page(unsigned int flags);

返回一个指向新页的指针并且用零填充了该页.

__get_free_page(unsigned int flags);

类似于 get_zeroed_page, 但是没有清零该页.

__get_free_pages(unsigned int flags, unsigned int order);

分配并返回一个指向一个内存区第一个字节的指针, 内存区可能是几个(物理上连续)页长但是没有清零.

flags 参数同 kmalloc 的用法相同; 常常使用 GFP_KERNEL 或者 GFP_ATOMIC, 可能带有 __GFP_DMA 标志(给可能用在 ISA DMA 操作的内存) 或者 __GFP_HIGHMEM 当可能使用高端内存时. [29]order 是你在请求的或释放的页数的以 2 为底的对数(即, log2N). 例如, 如果你要一个页 order 为 0, 如果你请求 8 页就是 3. 如果 order 太大(没有那个大小的连续区可用), 页分配失败. get_order 函数, 它使用一个整数参数, 可以用来从一个 size 中提取 order(它必须是 2 的幂)给主机平台. order 允许的最大值是 10 或者 11 (对应于 1024 或者 2048 页), 依赖于体系. 但是, 一个 order-10 的分配在除了一个刚刚启动的有很多内存的系统中成功的机会是小的.

如果你好奇, /proc/buddyinfo 告诉你系统中每个内存区中的每个 order 有多少块可用.

当一个程序用完这些页, 它可以使用下列函数之一来释放它们. 第一个函数是一个落回第二个函数的宏:

void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);

如果你试图释放和你分配的页数不同的页数, 内存图变乱, 系统在后面时间中有麻烦.

值得强调一下, __get_free_pages 和其他的函数可以在任何时候调用, 遵循我们看到的 kmalloc 的相同规则. 这些函数不能在某些情况下分配内存, 特别当使用 GFP_ATOMIC 时. 因此, 调用这些分配函数的程序必须准备处理分配失败.

尽管 kmalloc(GFP_KERNEL)有时失败当没有可用内存时, 内核尽力满足分配请求. 因此, 容易通过分配太多的内存降低系统的响应. 例如, 你可以通过塞入一个 scull 设备大量数据使计算机关机; 系统开始爬行当它试图换出尽可能多的内存来满足 kmalloc 的请求. 因为每个资源在被增长的设备所吞食, 计算机很快就被说无法用; 在这点上, 你甚至不能启动一个新进程来试图处理这个问题. 我们在 scull 不解释这个问题, 因为它只是一个例子模块并且不是一个真正的放入多用户系统的工具. 作为一个程序员, 你必须小心, 因为一个模块是特权代码并且可能在系统中开启新的安全漏洞(最可能是一个服务拒绝漏洞好像刚刚描述过的.)

8.3.1. 一个使用整页的 scull: scullp

为了真实地测试页分配, 我们已随其他例子代码发布了 scullp 模块. 它是一个简化的 scull, 就像前面介绍过的 scullc.

scullp 分配的内存量子是整页或者页集合: scullp_order 变量缺省是 0, 但是可以在编译或加载时改变.

下列代码行显示了它如何分配内存:

```
/* Here's the allocation of a single quantum */
if (!dptr->data[s_pos])
{
    dptr->data[s_pos] =
        (void *)__get_free_pages(GFP_KERNEL, dptr->order);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);
}
```

scullp 中释放内存的代码看来如此:

```
/* This code frees a whole quantum-set */
for (i = 0; i < qset; i++)
    if (dptr->data[i])
        free_pages((unsigned long)(dptr->data[i]), dptr->order);
```

在用户级别, 被感觉到的区别主要是一个速度提高和更好的内存使用, 因为没有内部的内存碎片. 我们运行一些测试从 scull0 拷贝 4 MB 到 scull1, 并且接着从 scullp0 到 scullp1; 结果显示了在内核空间处理器使用率有轻微上升.

性能的提高不是激动人心的, 因为 kmalloc 被设计为快的. 页级别分配的主要优势实际上不是速度, 而是更有效的内存使用. 按页分配不浪费内存, 而使用 kmalloc 由于分配的粒度会浪费无法预测数量的内存.

但是 __get_free_page 函数的最大优势是获得的页完全是你的, 并且你可以, 理论上, 可以通过适当的设置页表来组合这些页为一个线性的区域. 例如, 你可以允许一个用户进程 mmap 作为单个不联系的页而获得的内存区. 我们在 15 章讨论这种操作, 那里我们展示 scullp 如何提供内存映射, 一些 scull 无法提供的东西.

8.3.2. alloc_pages 接口

为完整起见, 我们介绍另一个内存分配的接口, 尽管我们不会准备使用它直到 15 章. 现在, 能够说 struct page 是一个描述一个内存页的内部内核结构. 如同我们将见到的, 在内核中有许多地方有必要使用页结构; 它们是特别有用的, 在任何你可能处理高端内存的情况下, 高端内存在内核空间中没有一个常量地址.

Linux 页分配器的真正核心是一个称为 alloc_pages_node 的函数:

```
struct page *alloc_pages_node(int nid, unsigned int flags,
 unsigned int order);
```

这个函数页有 2 个变体(是简单的宏); 它们是你最可能用到的版本:

```
struct page *alloc_pages(unsigned int flags, unsigned int order);
struct page *alloc_page(unsigned int flags);
```

核心函数, alloc_pages_node, 使用 3 个参数, nid 是要分配内存的 NUMA 节点 ID[30], flags 是通常的 GFP_ 分配标志, 以及 order 是分配的大小. 返回值是一个指向描述分配的内存的第一个(可能许多)页结构的指针, 或者, 如常, NULL 在失败时.

alloc_pages 简化了情况, 通过在当前 NUMA 节点分配内存(它使用 numa_node_id 的返回值作为 nid 参数调用 alloc_pages_node). 并且, 当然, alloc_pages 省略了 order 参数并且分配一个单个页.

为释放这种方式分配的页, 你应当使用下列一个:

```
void __free_page(struct page *page);
void __free_pages(struct page *page, unsigned int order);
void free_hot_page(struct page *page);
void free_cold_page(struct page *page);
```

如果你对一个单个页的内容是否可能驻留在处理器缓存中有特殊的认识, 你应当使用 free_hot_page (对于缓存驻留的页) 或者 free_cold_page 通知内核. 这个信息帮助内存分配器在系统中优化它的内存使用.

8.3.3. vmalloc 和 其友

我们展示给你的下一个内存分配函数是 vmlloc, 它在虚拟内存空间分配一块连续的内存区. 尽管这些页在物理内存中不连续 (使用一个单独的对 alloc_page 的调用来获得每个页), 内核看它们作为一个一个连续的地址范围. vmalloc 返回 0 (NULL 地址) 如果发生一个错误, 否则, 它返回一个指向一个大小至少为 size 的连续内存区.

我们这里描述 vmalloc 因为它是一个基本的 Linux 内存分配机制. 我们应当注意, 但是, vmalloc 的使用在大部分情况下不鼓励. 从 vmalloc 获得的内存用起来稍微低效些, 并且, 在某些体系上, 留给 vmalloc 的地址空间的数量相对小. 使用 vmalloc 的代码如果被提交来包含到内核中可能会受到冷遇. 如果可能, 你应当直接使用单个页而不是试图使用 vmalloc 来掩饰事情.

让我们看看 vmalloc 如何工作的. 这个函数的原型和它相关的东西(ioremap, 严格地不是一个分配函数, 在本节后面讨论)是下列:

```
#include <linux/vmalloc.h>
void *vmalloc(unsigned long size);
void vfree(void * addr);
void *ioremap(unsigned long offset, unsigned long size);
void iounmap(void * addr);
```

值得强调的是 kmalloc 和 _get_free_pages 返回的内存地址也是虚拟地址. 它们的实际值在它用在寻址物理地址前仍然由 MMU (内存管理单元, 常常是 CPU 的一部分)管理.[31] vmalloc 在它如何使用硬件上没有不同, 不同是在内核如何进行分配任务上.

kmalloc 和 __get_free_pages 使用的(虚拟)地址范围特有一个一对一映射到物理内存, 可能移位一个常量 PAGE_OFFSET 值; 这些函数不需要给这个地址范围修改页表. vmalloc 和 ioremap 使用的地址范围, 另一方面, 是完全地合成的, 并且每个分配建立(虚拟)内存区域, 通过适当地设置页表.

这个区别可以通过比较分配函数返回的指针来获知. 在一些平台(例如, x86), vmalloc 返回的地址只是远离 kmalloc 使用的地址. 在其他平台上(例如, MIPS, IA-64, 以及 x86_64), 它们属于一个完全不同的地址范围. 对 vmalloc 可用的地址在从 VMALLOC_START 到 VAMLLOC_END 的范围中. 2 个符号都定义在 <asm/patable.h> 中.

vmalloc 分配的地址不能用于微处理器之外, 因为它们只在处理器的 MMU 之上才有意义. 当一个驱动需要一个真正的物理地址(例如一个 DMA 地址, 被外设硬件用来驱动系统的总线), 你无法轻易使用 vmalloc. 调用 vmalloc 的正确时机是当你在为一个大的只存在于软件中的顺序缓冲分配内存

时. 重要的是注意 vamlloc 比 __get_free_pages 有更多开销, 因为它必须获取内存并且建立页表. 因此, 调用 vmalloc 来分配仅仅一页是无意义的.

在内核中使用 vmalloc 的一个例子函数是 create_module 系统调用, 它使用 vmalloc 为在创建的模块获得空间. 模块的代码和数据之后被拷贝到分配的空间中, 使用 copy_from_user. 在这个方式中, 模块看来是加载到连续的内存. 你可以验证, 同过看 /proc/kallsyms, 模块输出的内核符号位于一个不同于内核自身输出的符号的内存范围.

使用 vmalloc 分配的内存由 vfree 释放, 采用和 kfree 释放由 kmalloc 分配的内存的相同方式.

如同 vmalloc, ioremap 建立新页表; 不同于 vmalloc, 但是, 它实际上不分配任何内存. ioremap 的返回值是一个特殊的虚拟地址可用来存取特定的物理地址范围; 获得的虚拟地址应当最终通过调用 iounmap 来释放.

ioremap 对于映射一个 PCI 缓冲的(物理)地址到(虚拟)内核空间是非常有用的. 例如, 它可用来存取一个 PCI 视频设备的帧缓冲; 这样的缓冲常常被映射在高端物理地址, 在内核启动时建立的页表的地址范围之外. PCI 问题在 12 章有详细解释.

由于可移植性, 值得注意的是你不应当直接存取由 ioremap 返回的地址好像是内存指针.你应当一直使用 readb 和 其他的在第 9 章介绍的 I/O 函数. 这个要求适用因为一些平台, 例如 Alpha, 无法直接映射 PCI 内存区到处理器地址空间, 由于在 PCI 规格和 Alpha 处理器之间的在数据如何传送方面的不同.

ioremap 和 vmalloc 是面向页的(它通过修改页表来工作); 结果, 重分配的或者分配的大小被调整到最近的页边界. ioremap 模拟一个非对齐的映射通过"向下调整"被重映射的地址以及通过返回第一个被重映射页内的偏移.

vmalloc 的一个小的缺点在于它无法在原子上下文中使用, 因为, 内部地, 它使用 kmalloc (GFP_KERNEL) 来获取页表的存储, 并且因此可能睡眠. 这不应当是一个问题 -- 如果 __get_free_page 的使用对于一个中断处理不足够好, 软件设计需要一些清理.

8.3.4. 一个使用虚拟地址的 scull : scullv

使用 vmalloc 的例子代码在 scullv 模块中提供. 如同 scullp, 这个模块是一个 scull 的简化版本, 它使用一个不同的分配函数来为设备存储数据获得空间.

这个模块分配内存一次 16 页. 分配以大块方式进行来获得比 scullp 更好的性能, 并且来展示一些使用其他分配技术要花很长时间的东西是可行的. 使用 __get_free_pages 来分配多于一页是易于失败的, 并且就算它成功了, 它可能是慢的. 如同我们前面见到的, vmalloc 在分配几个页时比其他函数更快, 但是当获取单个页时有些慢, 因为页表建立的开销. scullv 被设计象 scullp 一样. order 指定每个分配的"级数"并且缺省为 4. scullv 和 scullp 之间的位于不同是在分配管理上. 这些代码行使用 vmalloc 来获得新内存:

```
/* Allocate a quantum using virtual addresses */
if (!dptr->data[s_pos])
{
      dptr->data[s_pos] =
            (void *)vmalloc(PAGE_SIZE << dptr->order);
      if (!dptr->data[s_pos])
            goto nomem;
      memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);
}
```

以及这些代码行释放内存:

```
/* Release the quantum-set */
for (i = 0; i < qset; i++)
      if (dptr->data[i])
            vfree(dptr->data[i]);
```

如果你在使能调试的情况下编译 2 个模块, 你能看到它们的数据分配通过读取它们在 /proc 创建的文件. 这个快照从一套 x86_64 系统上获得:

```
salma% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scullpmem
Device 0: qset 500, order 0, sz 1535135

item at 000001001847da58, qset at 000001001db4c000
0:1001db56000
1:1003d1c7000
salma% cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem
Device 0: qset 500, order 4, sz 1535135
item at 000001001847da58, qset at 0000010013dea000
0:ffffff0001177000
1:ffffff0001188000
```

下面的输出, 相反, 来自 x86 系统:

```
rudo% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scullpmem
Device 0: qset 500, order 0, sz 1535135
item at ccf80e00, qset at cf7b9800
0:ccc58000
1:cccdd000
rudo% cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem
Device 0: qset 500, order 4, sz 1535135
```

item at cfab4800, qset at cf8e4000
0:d087a000
1:d08d2000

这些值显示了 2 个不同的行为. 在 x86_64, 物理地址和虚拟地址是完全映射到不同的地址范围 (0x100 和 0xffffff00), 而在 x86 计算机上, vmalloc ；虚拟地址只在物理地址使用的映射之上.

---

[29] 尽管 alloc_pages (稍后描述)应当真正地用作分配高端内存页, 由于某些理由我们直到 15 章才真正涉及.

[30] NUMA (非统一内存存取) 计算机是多处理器系统, 这里内存对于特定的处理器组("节点")是"局部的". 对局部内存的存取比存取非局部内存更快. 在这样的系统, 在当前节点分配内存是重要的. 驱动作者通常不必担心 NUMA 问题, 但是.

[31] 实际上, 一些体系结构定义"虚拟"地址为保留给寻址物理内存. 当这个发生了, Linux 内核利用这个特性, 并且 kernel 和 __get_free_pages 地址都位于这些地址范围中的一个. 这个区别对设备驱动和其他的不直接包含到内存管理内核子系统中的代码是透明的

8.4. 每-CPU 的变量

每-CPU 变量是一个有趣的 2.6 内核的特性. 当你创建一个每-CPU变量, 系统中每个处理器获得它自己的这个变量拷贝. 这个可能象一个想做的奇怪的事情, 但是它有自己的优点. 存取每-CPU变量不需要(几乎)加锁, 因为每个处理器使用它自己的拷贝. 每-CPU 变量也可存在于它们各自的处理器缓存中, 这样对于频繁更新的量子带来了显著的更好性能.

一个每-CPU变量的好的使用例子可在网络子系统中找到. 内核维护无结尾的计数器来跟踪有每种报文类型有多少被接收; 这些计数器可能每秒几千次地被更新. 不去处理缓存和加锁问题, 网络开发者将统计计数器放进每-CPU变量. 现在更新是无锁并且快的. 在很少的机会用户空间请求看到计数器的值, 相加每个处理器的版本并且返回总数是一个简单的事情.

每-CPU变量的声明可在 <linux/percpu.h> 中找到. 为在编译时间创建一个每-CPU变量, 使用这个宏定义:

DEFINE_PER_CPU(type, name);

如果这个变量(称为 name 的)是一个数组, 包含这个类型的维数信息. 因此, 一个有 3 个整数的每-CPU 数组应当被创建使用:

DEFINE_PER_CPU(int[3], my_percpu_array);

每-CPU变量几乎不必使用明确的加锁来操作. 记住 2.6 内核是可抢占的; 对于一个处理器, 在修改一个每-CPU变量的临界区中不应当被抢占. 并且如果你的进程在对一个每-CPU变量存取时将, 要被移动到另一个处理器上, 也不好. 因为这个原因, 你必须显式使用 get_cpu_var 宏来存取当前处理器的给定变量拷贝, 并且当你完成时调用 put_cpu_var. 对 get_cpu_var 的调用返回一个 lvalue 给当前处理器的变量版本并且禁止抢占. 因为一个 lvalue 被返回, 它可被赋值给或者直接操作. 例如, 一个网络代码中的计数器时使用这 2 个语句来递增的:

```
get_cpu_var(sockets_in_use)++;
put_cpu_var(sockets_in_use);
```

你可以存取另一个处理器的变量拷贝, 使用:

per_cpu(variable, int cpu_id);

如果你编写使处理器涉及到对方的每-CPU变量的代码, 你, 当然, 一定要实现一个加锁机制来使存取安全.

动态分配每-CPU变量也是可能的. 这些变量可被分配, 使用:

```
void *alloc_percpu(type);
void *__alloc_percpu(size_t size, size_t align);
```

在大部分情况, alloc_percpu 做的不错; 你可以调用 __alloc_percpu 在需要一个特别的对齐的情况下. 在任一情况下, 一个 每-CPU 变量可以使用 free_percpu 被返回给系统. 存取一个动态分配的每-CPU 变量通过 per_cpu_ptr 来完成:

```
per_cpu_ptr(void *per_cpu_var, int cpu_id);
```

这个宏返回一个指针指向 per_cpu_var 对应于给定 cpu_id 的版本. 如果你在简单地读另一个 CPU 的这个变量的版本, 你可以解引用这个指针并且用它来完成. 如果, 但是, 你在操作当前处理器的版本, 你可能需要首先保证你不能被移出那个处理器. 如果你存取这个每-CPU变量的全部都持有一个自旋锁, 万事大吉. 常常, 但是, 你需要使用 get_cpu 来阻止在使用变量时的抢占. 因此, 使用动态每-CPU变量的代码会看来如此:

```
int cpu;
cpu = get_cpu()
ptr = per_cpu_ptr(per_cpu_var, cpu);
/* work with ptr */
put_cpu();
```

当使用编译时每-CPU 变量时, get_cpu_var 和 put_cpu_var 宏来照看这些细节. 动态每-CPU变量需要更多的显式的保护.

每-CPU变量能够输出给每个模块, 但是你必须使用一个特殊的宏版本:

```
EXPORT_PER_CPU_SYMBOL(per_cpu_var);
EXPORT_PER_CPU_SYMBOL_GPL(per_cpu_var);
```

为在一个模块内存取这样一个变量, 声明它, 使用:

```
DECLARE_PER_CPU(type, name);
```

DECLARE_PER_CPU 的使用(不是 DEFINE_PER_CPU)告知编译器进行一个外部引用.

如果你想使用每-CPU变量来创建一个简单的整数计数器, 看一下在 <linux/percpu_counter.h> 中的

现成的实现. 最后, 注意一些体系有有限数量的地址空间变量给每-CPU变量. 如果你创建每-CPU变量在你自己的代码, 你应当尽量使它们小.

上一页 8.3. get_free_page 和其友 | 上一级 起始页 | 下一页 8.5. 获得大量缓冲

8.5. 获得大量缓冲

我们我们已经在前面章节中注意到的, 大量连续内存缓冲的分配是容易失败的. 系统内存长时间会碎片化, 并且常常出现一个真正的大内存区会完全不可得. 因为常常有办法不使用大缓冲来完成工作, 内核开发者没有优先考虑使大分配能工作. 在你试图获得一个大内存区之前, 你应当真正考虑一下其他的选择. 到目前止最好的进行大 I/O 操作的方法是通过发散/汇聚操作, 我们在第 1 章的"发散-汇聚 映射"一节中讨论了.

8.5.1. 在启动时获得专用的缓冲

如果你真的需要一个大的物理上连续的缓冲, 最好的方法是在启动时请求内存来分配它. 在启动时分配是获得连续内存页而避开 __get_free_pages 施加的对缓冲大小限制的唯一方法, 不但最大允许大小还有限制的大小选择. 在启动时分配内存是一个"脏"技术, 因为它绕开了所有的内存管理策略通过保留一个私有的内存池. 这个技术是不优雅和不灵活的, 但是它也是最不易失败的. 不必说, 一个模块无法在启动时分配内存; 只有直接连接到内核的驱动可以这样做.

启动时分配的一个明显问题是对通常的用户它不是一个灵活的选择, 因为这个机制只对连接到内核映象中的代码可用. 一个设备驱动使用这种分配方法可以被安装或者替换只能通过重新建立内核并且重启计算机.

当内核被启动, 它赢得对系统种所有可用物理内存的存取. 它接着初始化每个子系统通过调用子系统的初始化函数, 允许初始化代码通过减少留给正常系统操作使用的 RAM 数量, 来分配一个内存缓冲给自己用.

启动时内存分配通过调用下面一个函数进行:

```
#include <linux/bootmem.h>
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size);
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
```

这些函数分配或者整个页(如果它们以 _pages 结尾)或者非页对齐的内存区. 分配的内存可能是高端内存除非使用一个 _low 版本. 如果你在为一个设备驱动分配这个缓冲, 你可能想用它做 DMA 操作, 并且这对于高端内存不是一直可能的; 因此, 你可能想使用一个 _low 变体.

很少在启动时释放分配的内存; 你会几乎肯定不能之后取回它, 如果你需要它. 但是, 有一个接口释

放这个内存:

void free_bootmem(unsigned long addr, unsigned long size);

注意以这个方式释放的部分页不返回给系统 -- 但是, 如果你在使用这个技术, 你已可能分配了不少数量的整页来用.

如果你必须使用启动时分配, 你需要直接连接你的驱动到内核. 应当如何完成的更多信息看在内核源码中 Documentation/kbuild 下的文件.

上一页 8.4. 每-CPU 的变量 | 上一级 起始页 | 下一页 8.6. 快速参考

8.6. 快速参考

相关于内存分配的函数和符号是:

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
void kfree(void *obj);
```

内存分配的最常用接口.

```
#include <linux/mm.h>
GFP_USER
GFP_KERNEL
GFP_NOFS
GFP_NOIO
GFP_ATOMIC
```

控制内存分配如何进行的标志, 从最少限制的到最多的. GFP_USER 和 GFP_KERNEL 优先级允许当前进程被置为睡眠来满足请求. GFP_NOFS 和 GFP_NOIO 禁止文件系统操作和所有的 I/O 操作, 分别地, 而 GFP_ATOMIC 分配根本不能睡眠.

```
__GFP_DMA
__GFP_HIGHMEM
__GFP_COLD
__GFP_NOWARN
__GFP_HIGH
__GFP_REPEAT
__GFP_NOFAIL
__GFP_NORETRY
```

这些标志修改内核的行为, 当分配内存时.

```
#include <linux/malloc.h>
kmem_cache_t *kmem_cache_create(char *name, size_t size, size_t offset, unsigned long flags, constructor
(), destructor( ));
int kmem_cache_destroy(kmem_cache_t *cache);
```

创建和销毁一个 slab 缓存. 这个缓存可被用来分配几个相同大小的对象.

```
SLAB_NO_REAP
SLAB_HWCACHE_ALIGN
```

SLAB_CACHE_DMA

在创建一个缓存时可指定的标志.

SLAB_CTOR_ATOMIC
SLAB_CTOR_CONSTRUCTOR

分配器可用传递给构造函数和析构函数的标志.

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

从缓存中分配和释放一个单个对象. /proc/slabinfo 一个包含对 slab 缓存使用情况统计的虚拟文件.

```
#include <linux/mempool.h>
mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn, mempool_free_t *free_fn, void
*data);
void mempool_destroy(mempool_t *pool);
```

创建内存池的函数, 它试图避免内存分配设备, 通过保持一个已分配项的"紧急列表".

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

从(并且返回它们给)内存池分配项的函数.

```
unsigned long get_zeroed_page(int flags);
unsigned long __get_free_page(int flags);
unsigned long __get_free_pages(int flags, unsigned long order);
```

面向页的分配函数. get_zeroed_page 返回一个单个的, 零填充的页. 这个调用的所有的其他版本不初始化返回页的内容.

```
int get_order(unsigned long size);
```

返回关联在当前平台的大小的分配级别, 根据 PAGE_SIZE. 这个参数必须是 2 的幂, 并且返回值至少是 0.

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

释放面向页分配的函数.

```
struct page *alloc_pages_node(int nid, unsigned int flags, unsigned int order);
struct page *alloc_pages(unsigned int flags, unsigned int order);
struct page *alloc_page(unsigned int flags);
```

Linux 内核中最底层页分配器的所有变体.

```
void __free_page(struct page *page);
```

```
void __free_pages(struct page *page, unsigned int order);
void free_hot_page(struct page *page);
```

使用一个 alloc_page 形式分配的页的各种释放方法.

```
#include <linux/vmalloc.h>
void * vmalloc(unsigned long size);
void vfree(void * addr);
#include <asm/io.h>
void * ioremap(unsigned long offset, unsigned long size);
void iounmap(void *addr);
```

分配或释放一个连续虚拟地址空间的函数. iormap 存取物理内存通过虚拟地址, 而 vmalloc 分配空闲页. 使用 ioreamp 映射的区是 iounmap 释放, 而从 vmalloc 获得的页使用 vfree 来释放.

```
#include <linux/percpu.h>
DEFINE_PER_CPU(type, name);
DECLARE_PER_CPU(type, name);
```

定义和声明每-CPU变量的宏.

```
per_cpu(variable, int cpu_id)
get_cpu_var(variable)
put_cpu_var(variable)
```

提供对静态声明的每-CPU变量存取的宏.

```
void *alloc_percpu(type);
void *__alloc_percpu(size_t size, size_t align);
void free_percpu(void *variable);
```

进行运行时分配和释放每-CPU变量的函数.

```
int get_cpu( );
void put_cpu( );
per_cpu_ptr(void *variable, int cpu_id)
```

get_cpu 获得对当前处理器的引用(因此, 阻止抢占和移动到另一个处理器)并且返回处理器的ID; put_cpu 返回这个引用. 为存取一个动态分配的每-CPU变量, 用应当被存取版本所在的 CPU 的 ID 来使用 per_cpu_ptr. 对一个动态的每-CPU 变量当前 CPU 版本的操作, 应当用对 get_cpu 和 put_cpu 的调用来包围.

```
#include <linux/bootmem.h>
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size);
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
void free_bootmem(unsigned long addr, unsigned long size);
```

在系统启动时进行分配和释放内存的函数(只能被直接连接到内核中去的驱动使用)

上一页 8.5. 获得大量缓冲
上一级 起始页
下一页 第 9 章 与硬件通讯

第 9 章 与硬件通讯

上一页 下一页

第 9 章 与硬件通讯

目录

尽管摆弄 scull 和类似的玩具是对于 Linux 设备驱动的软件接口一个很好的入门, 但是实现一个真正的设备需要硬件. 驱动是软件概念和硬件电路之间的抽象层; 如同这样, 需要与两者沟通. 直到现在, 我们已经检查了软件概念的内部; 本章完成这个图像通过向你展示一个驱动如何存取 I/O 端口和 I/O 内存, 同时在各种 Linux 平台是可移植的.

本章继续尽可能保持独立于特殊的硬件的传统. 但是, 在需要一个特殊例子的地方, 我们使用简单的数字 I/O 端口(例如标准的 PC 并口)来展示 I/O 指令如何工作, 以及正常的帧缓存视频内存来展示内存映射的I/O.

我们选择简单的数字 I/O, 因为它是一个输入/输出打开的最简单形式. 同样, 并口实现原始 I/O 并且

在大部分计算机都有: 写到设备的数据位出现在输出管脚上, 并且处理器可直接存取到输入管脚上的电平. 实际上, 你不得不连接 LED 或者一个打印机到端口上来真正地看到一个数组 I/O 操作的结果, 但是底层硬件非常易于使用.

9.1. I/O 端口和 I/O 内存

每个外设都是通过读写它的寄存器来控制. 大部分时间一个设备有几个寄存器, 并且在连续地址存取它们, 或者在内存地址空间或者在 I/O 地址空间.

在硬件级别上, 内存区和 I/O 区域没有概念上的区别: 它们都是通过在地址总线和控制总线上发出电信号来存取(即, 读写信号)[32]并且读自或者写到数据总线.

但是一些 CPU 制造商在他们的芯片上实现了一个单个地址空间, 有人认为外设不同于内存, 因此, 应该有一个分开的地址空间. 一些处理器(最有名的是 x86 家族)有分开的读和写电线给 I/O 端口和特殊的 CPU 指令来存取端口.

因为外设被建立来适合一个外设总线, 并且大部分流行的 I/O 总线成型在个人计算机上, 即便那些没有单独地址空间给 I/O 端口的处理器, 也必须在存取一些特殊设备时伪装读写端口, 常常利用外部的芯片组或者 CPU 核的额外电路. 后一个方法在用在嵌入式应用的小处理器中常见.

由于同样的理由, Linux 在所有它运行的计算机平台上实现了 I/O 端口的概念, 甚至在那些 CPU 实现一个单个地址空间的平台上. 端口存取的实现有时依赖特殊的主机制造和型号(因为不同的型号使用不同的芯片组来映射总线传送到内存地址空间).

即便外设总线有一个单独的地址空间给 I/O 端口, 不是所有的设备映射它们的寄存器到 I/O 端口. 虽然对于 ISA 外设板使用 I/O 端口是普遍的, 大部分 PCI 设备映射寄存器到一个内存地址区. 这种 I/O 内存方法通常是首选的, 因为它不需要使用特殊目的处理器指令; CPU 核存取内存更加有效, 并且编译器当存取内存时有更多自由在寄存器分配和寻址模式的选择上.

9.1.1. I/O 寄存器和常规内存

不管硬件寄存器和内存之间的强相似性, 存取 I/O 寄存器的程序员必须小心避免被 CPU(或者编译器)优化所戏弄, 它可能修改希望的 I/O 行为.

I/O 寄存器和 RAM 的主要不同是 I/O 操作有边际效果, 而内存操作没有: 一个内存写的唯一效果是存储一个值到一个位置, 并且一个内存读返回最近写到那里的值. 因为内存存取速度对 CPU 性能是至关重要的, 这种无边际效果的情况已被多种方式优化: 值被缓存, 并且 读/写指令被重编排.

编译器能够缓存数据值到 CPU 寄存器而不写到内存, 并且即便它存储它们, 读和写操作都能够在缓冲内存中进行而不接触物理 RAM. 重编排也可能在编译器级别和在硬件级别都发生: 常常一个

指令序列能够执行得更快, 如果它以不同于在程序文本中出现的顺序来执行, 例如, 为避免在 RISC 流水线中的互锁. 在CISC 处理器, 要花费相当数量时间的操作能够和其他的并发执行, 更快的.

当应用于传统内存时(至少在单处理器系统)这些优化是透明和有益的, 但是它们可能对正确的 I/O 操作是致命的, 因为它们干扰了那些"边际效果", 这是主要的原因为什么一个驱动存取 I/O 寄存器. 处理器无法预见这种情形, 一些其他的操作(在一个独立处理器上运行, 或者发生在一个 I/O 控制器的事情)依赖内存存取的顺序. 编译器或者 CPU 可能只尽力胜过你并且重编排你请求的操作; 结果可能是奇怪的错误而非常难于调试. 因此, 一个驱动必须确保没有进行缓冲并且在存取寄存器时没有发生读或写的重编排.

硬件缓冲的问题是最易面对的:底层的硬件已经配置(或者自动地或者通过 Linux 初始化代码)成禁止任何硬件缓冲, 当存取 I/O 区时(不管它们是内存还是端口区域).

对编译器优化和硬件重编排的解决方法是安放一个内存屏障在必须以一个特殊顺序对硬件(或者另一个处理器)可见的操作之间. Linux 提供 4 个宏来应对可能的排序需要:

```
#include <linux/kernel.h>
void barrier(void)
```

这个函数告知编译器插入一个内存屏障但是对硬件没有影响. 编译的代码将所有的当前改变的并且驻留在 CPU 寄存器的值存储到内存, 并且后来重新读取它们当需要时. 对屏障的调用阻止编译器跨越屏障的优化, 而留给硬件自由做它的重编排.

```
#include <asm/system.h>
void rmb(void);
void read_barrier_depends(void);
void wmb(void);
void mb(void);
```

这些函数插入硬件内存屏障在编译的指令流中; 它们的实际实例是平台相关的. 一个 rmb (read memory barrier) 保证任何出现于屏障前的读在执行任何后续读之前完成. wmb 保证写操作中的顺序, 并且 mb 指令都保证. 每个这些指令是一个屏障的超集.

read_barrier_depends 是读屏障的一个特殊的, 弱些的形式. 而 rmb 阻止所有跨越屏障的读的重编排, read_barrier_depends 只阻止依赖来自其他读的数据的读的重编排. 区别是微小的, 并且它不在所有体系中存在. 除非你确切地理解做什么, 并且你有理由相信, 一个完整的读屏障确实是一个过度地性能开销, 你可能应当坚持使用 rmb.

```
void smp_rmb(void);
void smp_read_barrier_depends(void);
void smp_wmb(void);
void smp_mb(void);
```

屏障的这些版本仅当内核为 SMP 系统编译时插入硬件屏障; 否则, 它们都扩展为一个简单的屏障调用.

在一个设备驱动中一个典型的内存屏障的用法可能有这样的形式:

```
writel(dev->registers.addr, io_destination_address);
writel(dev->registers.size, io_size);
writel(dev->registers.operation, DEV_READ);
wmb();
writel(dev->registers.control, DEV_GO);
```

在这种情况, 是重要的, 确保所有的控制一个特殊操作的设备寄存器在告诉它开始前已被正确设置. 内存屏障强制写以需要的顺序完成.

因为内存屏障影响性能, 它们应当只用在确实需要它们的地方. 屏障的不同类型也有不同的性能特性, 因此值得使用最特定的可能类型. 例如, 在 x86 体系上, wmb() 目前什么都不做, 因为写到处理器外不被重编排. 但是, 读被重编排, 因此 mb() 被 wmb() 慢.

值得注意大部分的其他的处理同步的内核原语, 例如自旋锁和原子的 _t 操作, 如同内存屏障一样是函数. 还值得注意的是一些外设总线(例如 PCI 总线)有它们自己的缓冲问题; 我们在以后章节遇到时讨论它们.

一些体系允许一个赋值和一个内存屏障的有效组合. 内核提供了几个宏来完成这个组合; 在缺省情况下, 它们如下定义:

```
#define set_mb(var, value) do {var = value; mb();}  while 0
#define set_wmb(var, value) do {var = value; wmb();} while 0
#define set_rmb(var, value) do {var = value; rmb();} while 0
```

在合适的地方, <asm/system.h> 定义这些宏来使用体系特定的指令来很快完成任务. 注意 set_rmb 只在少量体系上定义. (一个 do...while 结构的使用是一个标准 C 用语, 来使被扩展的宏作为一个正常的 C 语句可在所有上下文中工作).

---

[32] 不是所有的计算机平台使用一个读和一个写信号; 有些有不同的方法来寻址外部电路. 这个不同在软件层次是无关的, 但是, 我们将假设全部有读和写来简化讨论.

上一页 下一页

8.6. 快速参考 起始页 9.2. 使用 I/O 端口

9.2. 使用 I/O 端口

I/O 端口是驱动用来和很多设备通讯的方法, 至少部分时间. 这节涉及可用的各种函数来使用 I/O 端口; 我们也触及一些可移植性问题.

9.2.1. I/O 端口分配

如同你可能希望的, 你不应当离开并开始抨击 I/O 端口而没有首先确认你对这些端口有唯一的权限. 内核提供了一个注册接口以允许你的驱动来声明它需要的端口. 这个接口中的核心的函数是 request_region:

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long first, unsigned long n, const char *name);
```

这个函数告诉内核, 你要使用 n 个端口, 从 first 开始. name 参数应当是你的设备的名子. 如果分配成功返回值是非 NULL. 如果你从 request_region 得到 NULL, 你将无法使用需要的端口.

所有的的端口分配显示在 /proc/ioports 中. 如果你不能分配一个需要的端口组, 这是地方来看看谁先到那里了.

当你用完一组 I/O 端口(在模块卸载时, 也许), 应当返回它们给系统, 使用:

```
void release_region(unsigned long start, unsigned long n);
```

还有一个函数以允许你的驱动来检查是否一个给定的 I/O 端口组可用:

```
int check_region(unsigned long first, unsigned long n);
```

这里, 如果给定的端口不可用, 返回值是一个负错误码. 这个函数是不推荐的, 因为它的返回值不保证是否一个分配会成功; 检查和后来的分配不是一个原子的操作. 我们列在这里因为几个驱动仍然在使用它, 但是你调用一直使用 request_region, 它进行要求的加锁来保证分配以一个安全的原子的方式完成.

9.2.2. 操作 I/O 端口

在驱动硬件请求了在它的活动中需要使用的 I/O 端口范围之后, 它必须读且/或写到这些端口. 为

此, 大部分硬件区别8-位, 16-位, 和 32-位端口. 常常你无法混合它们, 象你正常使用系统内存存取一样.[33]

一个 C 程序, 因此, 必须调用不同的函数来存取不同大小的端口. 如果在前一节中建议的, 只支持唯一内存映射 I/O 寄存器的计算机体系伪装端口 I/O , 通过重新映射端口地址到内存地址, 并且内核向驱动隐藏了细节以便易于移植. Linux 内核头文件(特别地, 体系依赖的头文件 <asm/io.h>) 定义了下列内联函数来存取 I/O 端口:

unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);

读或写字节端口(8 位宽). port 参数定义为 unsigned long 在某些平台以及 unsigned short 在其他的上. inb 的返回类型也是跨体系而不同的.

unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);

这些函数存取 16-位 端口(一个字宽); 在为 S390 平台编译时它们不可用, 它只支持字节 I/O.

unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);

这些函数存取 32-位 端口. longword 声明为或者 unsigned long 或者 unsigned int, 根据平台. 如同字 I/O, "Long" I/O 在 S390 上不可用.

从现在开始, 当我们使用 unsigned 没有进一步类型规定时, 我们指的是一个体系相关的定义, 它的确切特性是不相关的. 函数几乎一直是可移植的, 因为编译器自动转换值在赋值时 -- 它们是 unsigned 有助于阻止编译时的警告. 这样的转换不丢失信息, 只要程序员安排明智的值来避免溢出. 我们坚持这个"未完成的类型"传统贯串本章.

注意, 没有定义 64-位 端口 I/O 操作. 甚至在 64-位 体系中, 端口地址空间使用一个32-位(最大)的数据通路.

9.2.3. 从用户空间的 I/O 存取

刚刚描述的这些函数主要打算被设备驱动使用, 但它们也可从用户空间使用, 至少在 PC-类 的计算机. GNU C 库在 <sys/io.h> 中定义它们. 下列条件应当应用来对于 inb 及其友在用户空间代码中使用:

- 程序必须使用 -O 选项编译来强制扩展内联函数.
- ioperm 和 iopl 系统调用必须用来获得权限来进行对端口的 I/O 操作. ioperm 为单独端口获取许可, 而 iopl 为整个 I/O 空间获取许可. 这 2 个函数都是 x86 特有的.
- 程序必须作为 root 来调用 ioperm 或者 iopl.[34] 可选地, 一个它的祖先必须已赢得作为 root 运

行的端口权限.

如果主机平台没有 ioperm 和 iopl 系统调用, 用户空间仍然可以存取 I/O 端口, 通过使用 /dev/prot 设备文件. 注意, 但是, 这个文件的含义是非常平台特定的, 并且对任何东西除了 PC 不可能有用.

例子源码 misc-progs/inp.c 和 misc-progs/outp.c 是一个从命令行读写端口的小工具, 在用户空间. 它们希望被安装在多个名子下(例如, inb, inw, 和 inl 并且操作字节, 字, 或者长端口依赖于用户调用哪个名子). 它们使用 ioperm 或者 iopl 在 x86下, 在其他平台是 /dev/port.

程序可以做成 setuid root, 如果你想过危险生活并且在不要求明确的权限的情况下使用你的硬件. 但是, 请不要在产品系统上以 set-uid 安装它们; 它们是设计上的安全漏洞.

9.2.4. 字串操作

除了单发地输入和输出操作, 一些处理器实现了特殊的指令来传送一系列字节, 字, 或者 长字 到和自一个单个 I/O 端口或者同样大小. 这是所谓的字串指令, 并且它们完成任务比一个 C 语言循环能做的更快. 下列宏定义实现字串处理的概念或者通过使用一个单个机器指令或者通过执行一个紧凑的循环, 如果目标处理器没有进行字串 I/O 的指令. 当编译为 S390 平台时这些宏定义根本不定义. 这应当不是个移植性问题, 因为这个平台通常不与其他平台共享设备驱动, 因为它的外设总线是不同的.

字串函数的原型是:

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
```

读或写从内存地址 addr 开始的 count 字节. 数据读自或者写入单个 port 端口.

```
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
```

读或写 16-位 值到一个单个 16-位 端口.

```
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

读或写 32-位 值到一个单个 32-位 端口.

有件事要记住, 当使用字串函数时: 它们移动一个整齐的字节流到或自端口. 当端口和主系统有不同的字节对齐规则, 结果可能是令人惊讶的. 使用 inw 读取一个端口交换这些字节, 如果需要, 来使读取的值匹配主机字节序. 字串函数, 相反, 不进行这个交换.

9.2.5. 暂停 I/O

一些平台 - 最有名的 i386 - 可能有问题当处理器试图太快传送数据到或自总线. 当处理器对于外设总线被过度锁定时可能引起问题(想一下 ISA)并且可能当设备单板太慢时表现出来. 解决方法是插入一个小的延时在每个 I/O 指令后面, 如果跟随着另一个指令. 在 x86 上, 这个暂停是通过进行一个 outb 指令到端口 0x80 (正常地不是常常用到)实现的, 或者通过忙等待. 细节见你的平台的 asm 子目录的 io.h 文件.

如果你的设备丢失一些数据, 或者如果你担心它可能丢失一些, 你可以使用暂停函数代替正常的那些. 暂停函数正如前面列出的, 但是它们的名子以 _p 结尾; 它们称为 inb_p, outb_p, 等等. 这些函数定义给大部分被支持的体系, 尽管它们常常扩展为与非暂停 I/O 同样的代码, 因为没有必要额外暂停, 如果体系使用一个合理的现代外设总线.

9.2.6. 平台依赖性

I/O 指令, 由于它们的特性, 是高度处理器依赖的. 因为它们使用处理器如何处理移进移出的细节, 是非常难以隐藏系统间的不同. 作为一个结果, 大部分的关于端口 I/O 的源码是平台依赖的.

你可以看到一个不兼容, 数据类型, 通过回看函数的列表, 这里参数是不同的类型, 基于平台间的体系不同点. 例如, 一个端口是 unsigned int 在 x86 (这里处理器支持一个 64-KB I/O 空间), 但是在别的平台是 unsiged long, 这里的端口只是同内存一样的同一个地址空间中的特殊位置.

其他的平台依赖性来自处理器中的基本的结构性不同, 并且, 因此, 无可避免地. 我们不会进入这个依赖性的细节, 因为我们假定你不会给一个特殊的系统编写设备驱动而没有理解底层的硬件. 相反, 这是一个内核支持的体系的能力的概括:

IA-32 (x86)
x86_64

这个体系支持所有的本章描述的函数. 端口号是 unsigned short 类型.

IA-64 (Itanium)

支持所有函数; 端口是 unsigned long(以及内存映射的)). 字串函数用 C 实现.

Alpha

支持所有函数, 并且端口是内存映射的. 端口 I/O 的实现在不同 Alpha 平台上是不同的, 根据它们使用的芯片组. 字串函数用 C 实现并且定义在 arch/alpha/lib/io.c 中定义. 端口是 unsigned long.

ARM

端口是内存映射的, 并且支持所有函数; 字串函数用 C 实现. 端口是 unsigned int 类型.

Cris

这个体系不支持 I/O 端口抽象, 甚至在一个模拟模式; 各种端口操作定义成什么不做.

M68k
M68k

端口是内存映射的. 支持字串函数, 并且端口类型是 unsigned char.

MIPS
MIPS64

MIPS 端口支持所有的函数. 字串操作使用紧凑汇编循环来实现, 因为处理器缺乏机器级别的字串 I/O. 端口是内存映射的; 它们是 unsigned long.

PA

支持所有函数; 端口是 int 在基于 PCI 的系统上以及 unsigned short 在 EISA 系统, 除了字串操作, 它们使用 unsigned long 端口号.

PowerPC
PowerPC64

支持所有函数; 端口有 unsigned char * 类型在 32-位 系统上并且 unsigned long 在 64-位 系统上.

S390

类似于 M68k, 这个平台的头文件只支持字节宽的端口 I/O, 而没有字串操作. 端口是 char 指针并且是内存映射的.

Super

端口是 unsigned int (内存映射的), 并且支持所有函数.

SPARC SPARC64

再一次, I/O 空间是内存映射的. 端口函数的版本定义来使用 unsigned long 端口.

好奇的读者能够从 io.h 文件中获得更多信息, 这个文件有时定义几个结构特定的函数, 加上我们在本章中描述的那些. 但是, 警告有些这些文件是相当难读的.

有趣的是注意没有 x86 家族之外的处理器具备一个不同的地址空间给端口, 尽管几个被支持的家族配备有 ISA 和/或 PCI 插槽 (并且 2 种总线实现分开的 I/O 和地址空间).

更多地, 有些处理器(最有名的是早期的 Alphas)缺乏一次移动一个或 2 个字节的指令.[35] 因此, 它们的外设芯片组模拟 8-位 和 16-位 I/O 存取, 通过映射它们到内存地址空间的特殊的地址范围. 因此, 操作同一个端口的一个 inb 和 一个 inw 指令, 通过 2 个操作不同地址的 32-位内存读来实现. 幸运的是, 所有这些都对设备驱动编写者隐藏了, 通过本节中描述的宏的内部, 但是我们觉得它是一个要注意的有趣的特性. 如果你想深入探究, 查找在 include/asm-alpha/core_lca.h 中的例子.

在每个平台的程序员手册中充分描述了I/O 操作如何在每个平台上进行; 这些手册常常在 WEB 上作为 PDF 下载.

---

[33] 有时 I/O 端口象内存一样安排, 并且你可(例如)绑定 2 个 8-位 写为一个单个 16-位 操作. 例如, 这应用于 PC 视频板. 但是通常, 你不能指望这个特色.

[34] 技术上, 它必须有 CAP_SYS_RAWIO 能力, 但是在大部分当前系统中这是与作为 root 运行是同样的.

[35] 单字节 I/O 不是一个人可能想象的那么重要, 因为它是一个稀少的操作. 为读/写一个单字节到任何地址空间, 你需要实现一个数据通道, 连接寄存器组的数据总线的低位到外部数据总线的任意字节位置. 这些数据通道需要额外的逻辑门在每个数据传输的通道上. 丢掉字节宽的载入和存储能够使整个系统性能受益.

9.3. 一个 I/O 端口例子

我们用来展示一个设备驱动内的端口 I/O 的例子代码, 操作通用的数字 I/O 端口; 这样的端口在大部分计算机系统中找到.

一个数字 I/O 端口, 在它的大部分的普通的化身中, 是一个字节宽的 I/O 位置, 或者内存映射的或者端口映射的. 当你写一个值到一个输出位置, 在输出管脚上见到的电信号根据写入的单个位而改变. 当你从一个输入位置读取一个值, 输入管脚上所见的当前逻辑电平作为单个位的值被返回.

这样的 I/O 端口的实际实现和软件接口各个系统不同. 大部分时间, I/O 管脚由 2 个 I/O 位置控制: 一个允许选择使用那些位作为输入, 哪些位作为输出, 以及一个可以实际读或写逻辑电平的. 有时, 但是, 事情可能更简单, 并且这些位是硬连线为输入或输出(但是, 在这个情况下, 它们不再是所谓的"通用 I/O"); 在所有个人计算机上出现的并口是这样一个非通用 I/O 端口. 任一方式, I/O 管脚对我们马上介绍的例子代码是可用的.

9.3.1. 并口纵览

因为我们期望大部分读者以所谓的"个人计算机"的形式使用一个 x86 平台, 我们觉得值得解释一下 PC 并口如何设计的. 并口是在个人计算机上运行数字 I/O 例子代码的外设接口选择. 尽管大部分读者可能有并口规范用, 为你的方便, 我们在这里总结一下它们.

并口, 在它的最小配置中 (我们浏览一下 ECP 和 EPP 模式) 由 3 个 8-位端口组成. PC 标准在 0x378 开始第一个并口的 I/O 端口并且第 2 个在 0x278. 第一个端口是一个双向数据寄存器; 它直接连接到物理连接器的管脚 2 - 9. 第 2 个端口是一个只读状态寄存器; 当并口为打印机使用, 这个寄存器报告打印机状态的几个方面, 例如正在线, 缺纸, 或者忙. 第 3 个端口是一个只出控制寄存器, 它, 在其他东西中, 控制是否中断使能.

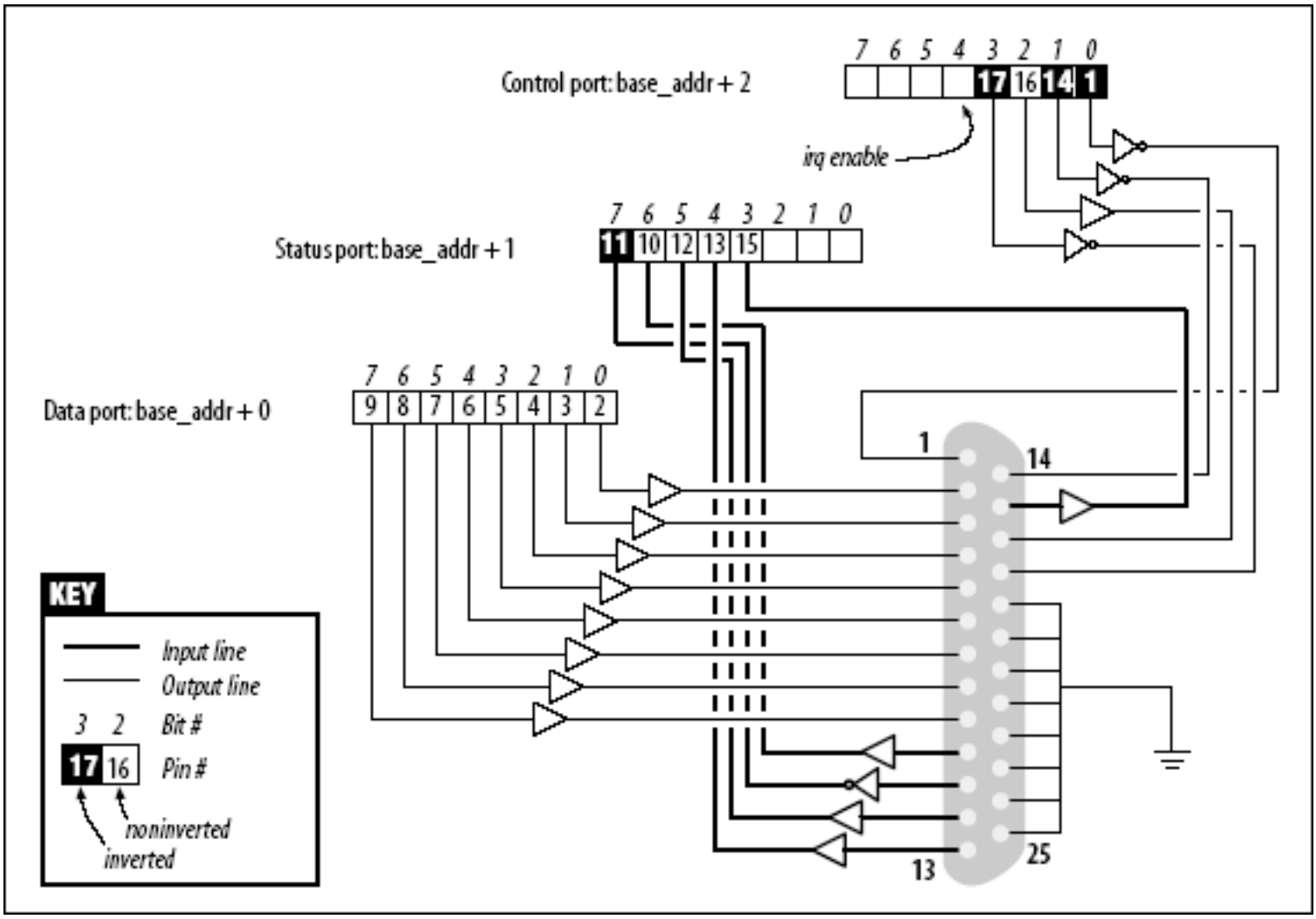
并口通讯中使用的信号电平是标准的 TTL 电平: 0 和 5 伏特, 逻辑门限在大概 1.2 伏特. 你可依靠端口至少符合标准 TTL LS 电流规格, 尽管大部分现代并口在电流和电压额定值都工作的好.

并口连接器和计算机内部电路不隔离, 当你想直接连接逻辑门到这个端口是有用的. 但是你不得不小心地正确连接线; 并口电路当你使用你自己的定制电路时容易损坏, 除非你给你的电路增加绝缘. 你可以选择使用插座并口如果你害怕会损坏你的主板.

位的规范在图 并口的管脚 中概述. 你可以存取 12 个输出位和 5 个输入位, 有些是在它们地信号路径上逻辑地翻转了. 唯一的没有关联信号管脚的位是端口 2 的位 4 (0x10), 它使能来自并口的中断.

我们使用这个位作为我们的在第 10 章中的中断处理的实现的一部分.

图 9.1. 并口的管脚



9.3.2. 一个例子驱动

我们介绍的驱动称为 short (Simple Hardware Operations and Raw Tests). 所有它做的是读和写几个 8-位 端口, 从你在加载时选择的开始. 缺省地, 它使用分配给 PC 并口的端口范围. 每个设备节点(有一个独特的次编号)存取一个不同的端口. short 驱动不做任何有用的事情; 它只是隔离来作为操作端口的单个指令给外部使用. 如果你习惯端口 I/O, 你可以使用 short 来熟悉它; 你能够测量它花费来通过端口传送数据的时间或者其他游戏的时间.

为 short 在你的系统上运行, 必须有存取底层硬件设备的自由(缺省地, 并口); 因此, 不能有其他驱动已经分配了它. 大部分现代发布设置并口驱动作为只在需要时加载的模块, 因此对 I/O 地址的竞争常常不是个问题. 如果, 但是, 你从 short 得到一个"无法获得 I/O 地址" 错误(在控制台上或者在系统 log 文件), 一些其他的驱动可能已经获得这个端口. 一个快速浏览 /proc/ioports 常常告诉你哪个驱动在捣乱. 同样的告诫应用于另外 I/O 设备如果你没有在使用并口.

从现在开始, 我们只是用"并口"来简化讨论. 但是, 你能够设置基本的模块参数在加载时来重定向 short 到其他 I/O 设备. 这个特性允许例子代码在任何 Linux 平台上运行, 这里你对一个数字 I/O 接口有权限通过 outb 和 inb 存取(尽管实际的硬件是内存映射的, 除 x86 外的所有平台). 后面, 在"使用 I/O 内存"的一节, 我们展示 short 如何用来使用通用的内存映射数字 I/O.

为观察在并口上发生了什么以及如果你有使用硬件的爱好, 你可以焊接尽管 LED 到输出管脚. 每个 LED 应当串连一个 1-K 电阻导向一个地引脚(除非, 当然, 你的 LED 有内嵌的电阻). 如果你连接一个输出引脚到一个输入管脚, 你会产生你自己的输入能够从输入端口读到.

注意, 你无法只连接一个打印机到并口并且看到数据发向 short. 这个驱动实现简单的对 I/O 端口的存取, 并且没有进行与打印机需要的来操作数据的握手; 在下一章, 我们展示了一个例子驱动(称为 shortprint), 它能够驱动并口打印机; 这个驱动使用中断, 但是, 因此我们还是不能到这一点.

如果你要查看并口数据通过焊接 LED 到一个 D-型 连接器, 我们建议你不要使用管脚 9 和管脚 10, 因为我们之后连接它们在一起来运行第 10 章展示的例子代码.

只考虑到 short, /dev/short0 写到和读自位于 I/O 基地址的 8-bit 端口(0x378, 除非在加载时间改变). /dev/short1 写到位于基址 + 1 的 8-位, 等等直到基址 + 7.

/dev/short0 进行的实际输出操作是基于使用 outb 的一个紧凑循环. 一个内存屏障指令用来保证输出操作实际发生并且不被优化掉:

```
while (count--) {
 outb(*(ptr++), port);
   wmb();
}
```

你可以运行下列命令来点亮你的 LED:

```
echo -n "any string" > /dev/short0
```

每个 LED 监视一个单个的输出端口位. 记住只有最后写入的字符, 保持稳定在输出管脚上足够长时间你的眼睛能感觉到. 因此, 我们建议你阻止自动插入一个结尾新行, 通过传递一个 -n 选项给 echo.

读是通过一个类似的函数, 围绕 inb 而不是 outb 建立的. 为了从并口读"有意义的"值, 你需要某个硬件连接到连接器的输入管脚来产生信号. 如果没有信号, 你会读到一个相同字节的无结尾的流. 如果你选择从一个输出端口读取, 你极可能得到写到端口的最后的值(这适用于并口和普通使用的其他数字 I/O 电路). 因此, 那些不喜欢拿出他们的烙铁的人可以读取当前的输出值在端口 0x378, 通过运行这样一个命令:

```
dd if=/dev/short0 bs=1 count=1 | od -t x1
```

为演示所有 I/O 指令的使用, 每个 short 设备有 3 个变形: /dev/short0 进行刚刚展示的循环, /dev/short0p 使用 outb_p 和 inb_p 代替"快速"函数, 并且 /dev/short0s 使用字串指令. 有 8 个这样的设备, 从 short0 到 short7. 尽管 PC 并口只有 3 个端口, 你可能需要它们更多如果使用不同的 I/O 设备来运行你的测试.

short 驱动进行一个非常少的硬件控制, 但是足够来展示如何使用 I/O 端口指令. 感兴趣的读者可能想看看 parpor 和 parport_pc 模块的源码, 来知道这个设备在真实生活中能有多复杂来支持一系列并口上的设备(打印机, 磁带备份, 网络接口)

上一页 9.2. 使用 I/O 端口
上一级 起始页
下一页 9.4. 使用 I/O 内存

9.4. 使用 I/O 内存

尽管 I/O 端口在 x86 世界中流行, 用来和设备通讯的主要机制是通过内存映射的寄存器和设备内存. 2 者都称为 I/O 内存, 因为寄存器和内存之间的区别对软件是透明的.

I/O 内存是简单的一个象 RAM 的区域, 它被处理器用来跨过总线存取设备. 这个内存可用作几个目的, 例如持有视频数据或者以太网报文, 同时实现设备寄存器就象 I/O 端口一样的行为(即, 它们有读和写它们相关联的边际效果).

存取 I/O 内存的方式依赖计算机体系, 总线, 和使用的设备, 尽管外设到处都一样. 本章的讨论主要触及 ISA 和 PCI 内存, 而也试图传递通用的信息. 尽管存取 PCI 内存在这里介绍, 一个 PCI 的通透介绍安排在第 12 章.

依赖计算机平台和使用的总线, I/O 内存可以或者不可以通过页表来存取. 当通过页表存取, 内核必须首先安排从你的驱动可见的物理地址, 并且这常常意味着你必须调用 ioremap 在做任何 I/O 之前. 如果不需要页表, I/O 内存位置看来很象 I/O 端口, 并且你只可以使用正确的包装函数读和写它们.

不管是否需要 ioremap 来存取 I/O 内存, 不鼓励直接使用 I/O 内存的指针. 尽管(如同在 "I/O 端口和 I/O 内存" 一节中介绍的)I/O 内存如同在硬件级别的正常 RAM 一样寻址, 在"I/O 寄存器和传统内存"一节中概述的额外的小心建议避免正常的指针. 用来存取 I/O 内存的包装函数在所有平台上是安全的并且在任何时候直接的指针解引用能够进行操作时, 会被优化掉.

因此, 尽管在 x86 上解引用一个指针能工作(在现在), 不使用正确的宏定义阻碍了驱动的移植性和可读性.

9.4.1. I/O 内存分配和映射

I/O 内存区必须在使用前分配. 分配内存区的接口是(在 <linux/ioport.h> 定义):

struct resource *request_mem_region(unsigned long start, unsigned long len, char *name);

这个函数分配一个 len 字节的内存区, 从 start 开始. 如果一切顺利, 一个 非NULL 指针返回; 否则返回值是 NULL. 所有的 I/O 内存分配来 /proc/iomem 中列出.

内存区在不再需要时应当释放:

void release_mem_region(unsigned long start, unsigned long len);

还有一个旧的检查 I/O 内存区可用性的函数:

int check_mem_region(unsigned long start, unsigned long len);

但是, 对于 check_region, 这个函数是不安全和应当避免的.

在存取内存之前, 分配 I/O 内嵌不是唯一的要求的步骤. 你必须也保证这个 I/O 内存已经对内核是可存取的. 使用 I/O 内存不只是解引用一个指针的事情; 在许多系统, I/O 内存根本不是可以这种方式直接存取的. 因此必须首先设置一个映射. 这是 ioremap 函数的功能, 在第 1 章的 "vmalloc 及其友"一节中介绍的. 这个函数设计来特别的安排虚拟地址给 I/O 内存区.

一旦装备了 ioremap (和iounmap), 一个设备驱动可以存取任何 I/O 内存地址, 不管是否它是直接映射到虚拟地址空间. 记住, 但是, 从 ioremap 返回的地址不应当直接解引用; 相反, 应当使用内核提供的存取函数. 在我们进入这些函数之前, 我们最好回顾一下 ioremap 原型和介绍几个我们在前一章略过的细节.

这些函数根据下列定义调用:

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void * addr);
```

首先, 你注意新函数 ioremap_nacache. 我们在第 8 章没有涉及它, 因为它的意思是明确地硬件相关的. 引用自一个内核头文件:"It ’ s useful if some control registers are in such an area, and write combining or read caching is not desirable.". 实际上, 函数实现在大部分计算机平台上与 ioremap 一致: 在所有 I/O 内存已经通过非缓冲地址可见的地方, 没有理由使用一个分开的, 非缓冲 ioremap 版本.

9.4.2. 存取 I/O 内存

在一些平台上, 你可能逃过作为一个指针使用 ioremap 的返回值的惩罚. 这样的使用不是可移植的, 并且, 更加地, 内核开发者已经努力来消除任何这样的使用. 使用 I/O 内存的正确方式是通过一系列为此而提供的函数(通过 <asm/io.h> 定义的).

从 I/O 内存读, 使用下列之一:

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
```

这里, addr 应当是从 ioremap 获得的地址(也许与一个整型偏移); 返回值是从给定 I/O 内存读取的.

有类似的一系列函数来写 I/O 内存:

```
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

如果你必须读和写一系列值到一个给定的 I/O 内存地址, 你可以使用这些函数的重复版本:

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);

void ioread32_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);
```

这些函数读或写 count 值从给定的 buf 到 给定的 addr. 注意 count 表达为在被写入的数据大小; ioread32_rep 读取 count 32-位值从 buf 开始.

上面描述的函数进行所有的 I/O 到给定的 addr. 如果, 相反, 你需要操作一块 I/O 地址, 你可使用下列之一:

```
void memset_io(void *addr, u8 value, unsigned int count);
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);
```

这些函数行为如同它们的 C 库类似物.

如果你通览内核源码, 你可看到许多调用旧的一套函数, 当使用 I/O 内存时. 这些函数仍然可以工作, 但是它们在新代码中的使用不鼓励. 除了别的外, 它们较少安全因为它们不进行同样的类型检查. 但是, 我们在这里描述它们:

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
```

这些宏定义用来从 I/O 内存获取 8-位, 16-位, 和 32-位 数据值.

```
void writeb(unsigned value, address);
```

```
void writew(unsigned value, address);
void writel(unsigned value, address);
```

如同前面的函数, 这些函数(宏)用来写 8-位, 16-位, 和 32-位数据项.

一些 64-位平台也提供 readq 和 writeq, 为 PCI 总线上的 4-字(8-字节)内存操作. 这个 4-字 的命名是一个从所有的真实处理器有 16-位 字的时候的历史遗留. 实际上, 用作 32-位 值的 L 命名也已变得不正确, 但是命名任何东西可能使事情更混淆.

9.4.3. 作为 I/O 内存的端口

一些硬件有一个有趣的特性: 一些版本使用 I/O 端口, 而其他的使用 I/O 内存. 输出给处理器的寄存器在任一种情况中相同, 但是存取方法是不同的. 作为一个使驱动处理这类硬件的生活容易些的方式, 并且作为一个使 I/O 端口和内存存取的区别最小化的方法, 2.6 内核提供了一个函数, 称为 ioport_map:

```
void *ioport_map(unsigned long port, unsigned int count);
```

这个函数重映射 count I/O 端口和使它们出现为 I/O 内存. 从这点以后, 驱动可以在返回的地址上使用 ioread8 和其友并且根本忘记它在使用 I/O 端口.

这个映射应当在它不再被使用时恢复:

```
void ioport_unmap(void *addr);
```

这些函数使 I/O 端口看来象内存. 但是, 注意 I/O 端口必须仍然使用 request_region 在它们以这种方式被重映射前分配.

9.4.4. 重用 short 为 I/O 内存

short 例子模块, 在存取 I/O 端口前介绍的, 也能用来存取 I/O 内存. 为此, 你必须告诉它使用 I/O 内存在加载时; 还有, 你需要改变基地址来使它指向你的 I/O 区.

例如, 这是我们如何使用 short 来点亮调试 LED, 在一个 MIPS 开发板上:

```
mips.root# ./short_load use_mem=1 base=0xb7ffffc0
mips.root# echo -n 7 > /dev/short0
```

使用 short 给 I/O 内存是与它用在 I/O 端口上同样的.

下列片段显示了 short 在写入一个内存位置时用的循环:

```
while (count--) {
 iowrite8(*ptr++, address);
   wmb();
}
```

注意, 这里使用一个写内存屏障. 因为在很多体系上 iowrites8 可能转变为一个直接赋值, 需要内存屏障来保证以希望的顺序来发生.

short 使用 inb 和 outb 来显示它如何完成. 对于读者它可能是一个直接的练习, 但是, 改变 short 来使用 ioport_map 重映射 I/O 端口, 并且相当地简化剩下的代码.

9.4.5. 在 1 MB 之下的 ISA 内存

一个最著名的 I/O 内存区是在个人计算机上的 ISA 范围. 这是在 640 KB(0xA0000)和 1 MB(0x100000)之间的内存范围. 因此, 它正好出现于常规内存 RAM 中间. 这个位置可能看起来有点奇怪; 它是一个在 1980 年代早期所作的决定的产物, 当时 640 KB 内存看来多于任何人可能用到的大小.

这个内存方法属于非直接映射的内存类别. [36]你可以读/写几个字节在这个内存范围, 如同前面解释的使用 short 模块, 就是, 通过在加载时设置 use_mem.

尽管 ISA I/O 内存只在 x86-类 计算机中存在, 我们认为值得用几句话和一个例子驱动.

我们不会谈论 PCI 在本章, 因为它是最干净的一类 I/O 内存: 一旦你知道内存地址, 你可简单地重映射和存取它. PCI I/O 内存的"问题"是它不能为本章提供一个能工作的例子, 因为我们不能事先知道你的 PCI 内存映射到的物理地址, 或者是否它是安全的来存取任一这些范围. 我们选择来描述 ISA 内存范围, 因为它不但少干净并且更适合运行例子代码.

为演示存取 ISA 内存, 我们还使用另一个 silly 小模块(例子源码的一部分). 实际上, 这个称为 silly, 作为 Simple Tool for Unloading and Printing ISA Data 的缩写, 或者如此的东东.

模块补充了 short 的功能, 通过存取整个 384-KB 内存空间和通过显示所有的不同 I/O 功能. 它特有 4 个设备节点来进行同样的任务, 使用不同的数据传输函数. silly 设备作为一个 I/O 内存上的窗口, 以类似 /dev/mem 的方式. 你可以读和写数据, 并且lseek 到一个任意 I/O 内存地址.

因为 silly 提供了对 ISA 内存的存取, 它必须开始于从映射物理 ISA 地址到内核虚拟地址. 在 Linux 内核的早期, 一个人可以简单地安排一个指针给一个感兴趣的 ISA 地址, 接着直接对它解引用. 在现代世界, 但是, 我们必须首先使用虚拟内存系统和重映射内存范围. 这个映射使用 ioremap 完成, 如同前面为 short 解释的:

```
#define ISA_BASE 0xA0000
```

```
#define ISA_MAX 0x100000 /* for general memory access */

/* this line appears in silly_init */
io_base = ioremap(ISA_BASE, ISA_MAX - ISA_BASE);
```

ioremap 返回一个指针值, 它能被用来使用 ioread8 和其他函数, 在"存取 I/O 内存"一节中解释.

让我们回顾我们的例子模块来看看这些函数如何被使用. /dev/sillyb, 特有次编号 0, 存取 I/O 内存使用 ioread8 和 iowrite8. 下列代码显示了读的实现, 它使地址范围 0xA0000-0xFFFF 作为一个虚拟文件在范围 0-0x5FFF. 读函数构造为一个 switch 语句在不同存取模式上; 这是 sillyb 例子:

```
case M_8:
 while (count) {
 *ptr = ioread8(add);
 add++;
 count--;
 ptr++;

 }
 break;
```

实际上, 这不是完全正确. 内存范围是很小和很频繁的使用, 以至于内核在启动时建立页表来存取这些地址. 但是, 这个用来存取它们的虚拟地址不是同一个物理地址, 并且因此无论如何需要 ioremap.

下 2 个设备是 /dev/sillyw (次编号 1) 和 /dev/silly1 (次编号 2). 它们表现象 /dev/sillyb, 除了它们使用 16-位 和 32-位 函数. 这是 sillyl 的写实现, 又一次部分 switch:

```
case M_32:
 while (count >= 4) {
 iowrite8(*(u32 *)ptr, add);
 add += 4;
 count -= 4;
 ptr += 4;

 }
 break;
```

最后的设备是 /dev/sillycp (次编号 3), 它使用 memcpy_*io 函数来进行同样的任务. 这是它的读实现的核心:

```
case M_memcpy:
```

```
memcpy_fromio(ptr, add, count);
break;
```

因为 ioremap 用来提供对 ISA 内存区的存取, silly 必须调用 iounmap 当模块卸载时:

```
iounmap(io_base);
```

9.4.6. isa_readb 和其友

看一下内核源码会展现另一套函数, 有如 isa_readb 的名子. 实际上, 每个刚才描述的函数都有一个 isa_ 对等体. 这些函数提供对 ISA 内存的存取不需要一个单独的 ioremap 步骤. 但是, 来自内核开发者的话, 是这些函数打算用来作为暂时的驱动移植辅助, 并且它可能将来消失. 因此, 你应当避免使用它们.

9.5. 快速参考

本章介绍下列与硬件管理相关的符号:

```
#include <linux/kernel.h>
void barrier(void)
```

这个"软件"内存屏蔽要求编译器对待所有内存是跨这个指令而非易失的.

```
#include <asm/system.h>
void rmb(void);
void read_barrier_depends(void);
void wmb(void);
void mb(void);
```

硬件内存屏障. 它们请求 CPU(和编译器)来检查所有的跨这个指令的内存读, 写, 或都有.

```
#include <asm/io.h>
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
unsigned inl(unsigned port);
void outl(unsigned doubleword, unsigned port);
```

用来读和写 I/O 端口的函数. 它们还可以被用户空间程序调用, 如果它们有正当的权限来存取端口.

```
unsigned inb_p(unsigned port);
```

如果在一次 I/O 操作后需要一个小延时, 你可以使用在前一项中介绍的这些函数的 6 个暂停对应部分; 这些暂停函数有以 _p 结尾的名子.

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

这些"字串函数"被优化为传送数据从一个输入端口到一个内存区, 或者其他的方式. 这些传

送通过读或写到同一端口 count 次来完成.

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long start, unsigned long len, char *name);
void release_region(unsigned long start, unsigned long len);
int check_region(unsigned long start, unsigned long len);
```

I/O 端口的资源分配器. 这个检查函数成功返回 0 并且在错误时小于 0.

```
struct resource *request_mem_region(unsigned long start, unsigned long len, char *name);
void release_mem_region(unsigned long start, unsigned long len);
int check_mem_region(unsigned long start, unsigned long len);
```

为内存区处理资源分配的函数

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void *virt_addr);
```

ioremap 重映射一个物理地址范围到处理器的虚拟地址空间, 使它对内核可用. iounmap 释放映射当不再需要它时.

```
#include <asm/io.h>
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

用来使用 I/O 内存的存取者函数.

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);
```

I/O 内存原语的"重复"版本.

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

```
memset_io(address, value, count);
memcpy_fromio(dest, source, nbytes);
memcpy_toio(dest, source, nbytes);
```

旧的, 类型不安全的存取 I/O 内存的函数.

```
void *ioport_map(unsigned long port, unsigned int count);
void ioport_unmap(void *addr);
```

一个想对待 I/O 端口如同它们是 I/O 内存的驱动作者, 可以传递它们的端口给 ioport_map. 这个映射应当在不需要的时候恢复 (使用 ioport_unmap)

上一页 9.4. 使用 I/O 内存 | 上一级 起始页 | 下一页 第 10 章 中断处理

第 10 章 中断处理

第 10 章 中断处理

目录

尽管一些设备可只使用它们的 I/O 区来控制, 大部分真实的设备比那个要复杂点. 设备不得不和外部世界打交道, 常常包括诸如旋转的磁盘, 移动的磁带, 连到远处的线缆, 等等. 很多必须在一个时间片中完成, 不同于, 并且远慢于处理器. 因为几乎一直是不希望使处理器等待外部事件, 对于设备必须有一种方法使处理器知道有事情发生了.

当然, 那种方法是中断. 一个中断不过是一个硬件在它需要处理器的注意时能够发出的信号. Linux 处理中断非常类似它处理用户空间信号的方式. 对大部分来说, 一个驱动只需要为它的设备中断注册一个处理函数, 并且当它们到来时正确处理它们. 当然, 在这个简单图像之下有一些复杂; 特别地, 中断处理有些受限于它们能够进行的动作, 这是它们如何运行而导致的结果.

没有一个真实的硬件设备来产生中断, 就难演示中断的使用. 因此, 本章使用的例子代码使用并口工作. 这些端口在现代硬件上开始变得稀少, 但是, 运气地, 大部分人仍然能够有一个有可用的端口

的系统. 我们将使用来自上一章的简短模块; 添加一小部分它能够产生并处理来自并口的中断. 模块的名子, short, 实际上意味着 short int (它是 C, 对不?), 来提醒我们它处理中断.

但是, 在我们进入主题之前, 是时候提出一个注意事项. 中断处理, 由于它们的特性, 与其他的代码并行地运行. 因此, 它们不可避免地引起并发问题和对数据结构和硬件的竞争. 如果你屈服于诱惑以越过第 5 章的讨论, 我们理解. 但是我们也建议你转回去并且现在看一下. 一个坚实的并发控制技术的理解是重要的, 在使用中断时.

10.1. 准备并口

尽管并口简单, 它能够触发中断. 这个能力被打印机用来通知 lp 驱动它准备好接收缓存中的下一个字符.

如同大部分设备, 并口实际上不产生中断, 在它被指示这样作之前; 并口标准规定设置 port 2 (0x37a, 0x27a, 或者任何)的 bit 4 就使能中断报告. short 在模块初始化时进行一个简单的 outb 调用来设置这个位.

一旦中断使能, 任何时候在管脚 10 (所谓的 ACK 位)上的电信号从低变到高, 并口产生一个中断. 最简单的方法来强制接口产生中断(没有挂一个打印机到端口)是连接并口连接器的管脚 9 和 管脚 10. 一根短线, 插到你的系统后面的并口连接器的合适的孔中, 就建立这个连接. 并口外面的管脚图示于图并口的管脚

管脚 9 是并口数据字节的最高位. 如果你写二进制数据到 /dev/short0, 你产生几个中断. 然而, 写 ASCII 文本到这个端口不会产生任何中断, 因为 ASCII 字符集没有最高位置位的项.

如果你宁愿避免连接管脚到一起, 而你手上确实有一台打印机, 你可用使用一个真正的打印机来运行例子中断处理, 如同下面展示的. 但是, 注意我们介绍的探测函数依赖管脚 9 和管脚 10 之间的跳线在位置上, 并且你需要它使用你的代码来试验探测.

上一页 9.5. 快速参考 | 起始页 | 下一页 10.2. 安装一个中断处理

10.2. 安装一个中断处理

如果你想实际地"看到"产生的中断, 向硬件设备写不足够; 一个软件处理必须在系统中配置. 如果 Linux 内核还没有被告知来期待你的中断, 它简单地确认并忽略它.

中断线是一个宝贵且常常有限的资源, 特别当它们只有 15 或者 16 个时. 内核保持了中断线的一个注册, 类似于 I/O 端口的注册. 一个模块被希望来请求一个中断通道(或者 IRQ, 对于中断请求), 在使用它之前, 并且当结束时释放它. 在很多情况下, 也希望模块能够与其他驱动共享中断线, 如同我们将看到的. 下面的函数, 声明在 <linux/interrupt.h>, 实现中断注册接口:

```
int request_irq(unsigned int irq,
        irqreturn_t (*handler)(int, void *, struct pt_regs *),
        unsigned long flags,

        const char *dev_name,
        void *dev_id);

void free_irq(unsigned int irq, void *dev_id);
```

从 request_irq 返回给请求函数的返回值或者是 0 指示成功, 或者是一个负的错误码, 如同平常. 函数返回 -EBUSY 来指示另一个驱动已经使用请求的中断线是不寻常的. 函数的参数如下:

unsigned int irq

请求的中断号

irqreturn_t (*handler)

安装的处理函数指针. 我们在本章后面讨论给这个函数的参数以及它的返回值.

unsigned long flags

如你会希望的, 一个与中断管理相关的选项的位掩码(后面描述).

const char *dev_name

这个传递给 request_irq 的字串用在 /proc/interrupts 来显示中断的拥有者(下一节看到)

void *dev_id

用作共享中断线的指针. 它是一个独特的标识, 用在当释放中断线时以及可能还被驱动用来

指向它自己的私有数据区(来标识哪个设备在中断). 如果中断没有被共享, dev_id 可以设置为 NULL, 但是使用这个项指向设备结构不管如何是个好主意. 我们将在"实现一个处理"一节中看到 dev_id 的一个实际应用.

flags 中可以设置的位如下:

SA_INTERRUPT

当置位了, 这表示一个"快速"中断处理. 快速处理在当前处理器上禁止中断来执行(这个主题在"快速和慢速处理"一节涉及).

SA_SHIRQ

这个位表示中断可以在设备间共享. 共享的概念在"中断共享"一节中略述.

SA_SAMPLE_RANDOM

这个位表示产生的中断能够有贡献给 /dev/random 和 /dev/urandom 使用的加密池. 这些设备在读取时返回真正的随机数并且设计来帮助应用程序软件为加密选择安全钥. 这样的随机数从一个由各种随机事件贡献的加密池中提取的. 如果你的设备以真正随机的时间产生中断, 你应当设置这个标志. 如果, 另一方面, 你的中断是可预测的(例如, 一个帧抓取器的场消隐), 这个标志不值得设置 -- 它无论如何不会对系统加密有贡献. 可能被攻击者影响的设备不应当设置这个标志; 例如, 网络驱动易遭受从外部计时的可预测报文并且不应当对加密池有贡献. 更多信息看 drivers/char/random.c 的注释.

中断处理可以在驱动初始化时安装或者在设备第一次打开时. 尽管从模块的初始化函数中安装中断处理可能听来是个好主意, 它常常不是, 特别当你的设备不共享中断. 因为中断线数目是有限的, 你不想浪费它们. 你可以轻易使你的系统中设备数多于中断数.如果一个模块在初始化时请求一个 IRQ, 它阻止了任何其他的驱动使用这个中断, 甚至这个持有它的设备从不被使用. 在设备打开时请求中断, 另一方面, 允许某些共享资源.

例如, 可能与一个 modem 在同一个中断上运行一个帧抓取器, 只要你不同时使用这 2 个设备. 对用户来说是很普通的在系统启动时为一个特殊设备加载模块, 甚至这个设备很少用到. 一个数据获取技巧可能使用同一个中断作为第 2 个串口. 虽然不是太难避免在数据获取时联入你的互联网服务提供商(ISP), 被迫卸载一个模块为了使用 modem 确实令人不快.

调用 request_irq 的正确位置是当设备第一次打开时, 在硬件被指示来产生中断前. 调用 free_irq 的位置是设备最后一次被关闭时, 在硬件被告知不要再中断处理器之后. 这个技术的缺点是你需要保持一个每设备的打开计数, 以便于你知道什么时候中断可以被禁止.

尽管这个讨论, short 还在加载时请求它的中断线. 这样做是为了你可以运行测试程序而不必运行一个额外的进程来保持设备打开. short, 因此, 从它的初始化函数(short_init)请求中断, 不是在 short_open 中做, 象一个真实设备驱动.

下面代码请求的中断是 short_irq. 变量的真正赋值(即, 决定使用哪个 IRQ)在后面显示, 因为它和现在的讨论无关. short_base 是使用的并口 I/O 基地址; 接口的寄存器 2 被写入来使能中断报告.

```
if (short_irq >= 0)
{
      result = request_irq(short_irq, short_interrupt,
                  SA_INTERRUPT, "short", NULL);
      if (result) {
            printk(KERN_INFO "short: can't get assigned irq %i\n",
                short_irq);

            short_irq = -1;
      } else { /* actually enable it -- assume this *is* a parallel port */
            outb(0x10,short_base+2);
      }
}
```

代码显示, 安装的处理是一个快速处理(SA_INTERRUPT), 不支持中断共享(SA_SHIRQ 没有), 并且不对系统加密有贡献(SA_SAMPLE_RANDOM 也没有). outb 调用接着为并口使能中断报告.

由于某些合理原因, i386 和 x86_64 体系定义了一个函数来询问一个中断线的能力:

```
int can_request_irq(unsigned int irq, unsigned long flags);
```

这个函数当试图分配一个给定中断成功时返回一个非零值. 但是, 注意, 在 can_request_irq 和 request_irq 的调用之间事情可能一直改变.

10.2.1. /proc 接口

无论何时一个硬件中断到达处理器, 一个内部的计数器递增, 提供了一个方法来检查设备是否如希望地工作. 报告的中断显示在 /proc/interrupts. 下面的快照取自一个双处理器 Pentium 系统:

```
root@montalcino:/bike/corbet/write/ldd3/src/short# m /proc/interrupts
       CPU0    CPU1
 0: 4848108      34  IO-APIC-edge  timer
 2:       0       0       XT-PIC  cascade
 8:       3       1  IO-APIC-edge  rtc
 10:   4335       1  IO-APIC-level  aic7xxx
 11:   8903       0  IO-APIC-level  uhci_hcd
 12:     49       1  IO-APIC-edge  i8042
NMI:      0       0
LOC: 4848187  4848186
```

```
ERR:      0
MIS:      0
```

第一列是 IRQ 号. 你能够从没有的 IRQ 中看到这个文件只显示对应已安装处理的中断. 例如, 第一个串口(使用中断号 4)没有显示, 指示 modem 没在使用. 事实上, 即便如果 modem 已更早使用了, 但是在这个快照时间没有使用, 它不会显示在这个文件中; 串口表现很好并且在设备关闭时释放它们的中断处理.

/proc/interrupts 的显示展示了有多少中断硬件递交给系统中的每个 CPU. 如同你可从输出看到的, Linux 内核常常在第一个 CPU 上处理中断, 作为一个使 cache 局部性最大化的方法.[37] 最后 2 列给出关于处理中断的可编程中断控制器的信息(驱动编写者不必关心), 以及已注册的中断处理的设备的名子(如同在给 request_irq 的参数 dev_name 中指定的).

/proc 树包含另一个中断有关的文件, /proc/stat; 有时你会发现一个文件更加有用并且有时你会喜欢另一个. /proc/stat 记录了几个关于系统活动的低级统计量, 包括(但是不限于)自系统启动以来收到的中断数. stat 的每一行以一个文本字串开始, 是该行的关键词; intr 标志是我们在找的. 下列(截短了)快照是在前一个后马上取得的:

```
intr 5167833 5154006 2 0 2 4907 0 2 68 4 0 4406 9291 50 0 0
```

第一个数是所有中断的总数, 而其他每一个代表一个单个 IRQ 线, 从中断 0 开始. 所有的计数跨系统中所有处理器而汇总的. 这个快照显示, 中断号 4 已使用 4907 次, 尽管当前没有安装处理. 如果你在测试的驱动请求并释放中断在每个打开和关闭循环, 你可能发现 /proc/stat 比 /proc/interrupts 更加有用.

2 个文件的另一个不同是, 中断不是体系依赖的(也许, 除了末尾几行), 而 stat 是; 字段数依赖内核之下的硬件. 可用的中断数目少到在 SPARC 上的 15 个, 多到 IA-64 上的 256个, 并且其他几个系统都不同. 有趣的是要注意, 定义在 x86 中的中断数当前是 224, 不是你可能期望的 16; 如同在 include/asm-i386/irq.h 中解释的, 这依赖 Linux 使用体系的限制, 而不是一个特定实现的限制(例如老式 PC 中断控制器的 16 个中断源).

下面是一个 /proc/interrupts 的快照, 取自一台 IA-64 系统. 如你所见, 除了不同硬件的通用中断源的路由, 输出非常类似于前面展示的 32-位 系统的输出.

```
       CPU0       CPU1
 27:       1705      34141  IO-SAPIC-level  qla1280
 40:          0          0  SAPIC                       perfmon
 43:        913       6960  IO-SAPIC-level  eth0
 47:      26722        146  IO-SAPIC-level  usb-uhci
 64:          3          6  IO-SAPIC-edge   ide0
 80:          4          2  IO-SAPIC-edge   keyboard
 89:          0          0  IO-SAPIC-edge   PS/2 Mouse
```

```
239:  5606341  5606052        SAPIC   timer

254:  67575  52815  SAPIC  IPI
NMI:  0  0
ERR:  0
```

10.2.2. 自动检测 IRQ 号

驱动在初始化时最有挑战性的问题中的一个是如何决定设备要使用哪个 IRQ 线. 驱动需要信息来正确安装处理. 尽管程序员可用请求用户在加载时指定中断号, 这是个坏做法, 因为大部分时间用户不知道这个号, 要么因为他不配置跳线要么因为设备是无跳线的. 大部分用户希望他们的硬件"仅仅工作"并且不感兴趣如中断号的问题. 因此自动检测中断号是一个驱动可用性的基本需求.

有时自动探测依赖知道一些设备有很少改变的缺省动作的特性. 在这个情况下, 驱动可能假设缺省值适用. 这确切地就是 short 如何缺省对并口动作的. 实现是直接的, 如 short 自身显示的:

```
if (short_irq < 0) /* not yet specified: force the default on */
 switch(short_base) {
 case 0x378: short_irq = 7; break;
 case 0x278: short_irq = 2; break;
 case 0x3bc: short_irq = 5; break;
 }
```

代码根据选择的 I/O 基地址赋值中断号, 而允许用户在加载时覆盖缺省值, 使用如:

```
insmod ./short.ko irq=x
short_base defaults to 0x378, so short_irq defaults to 7.
```

有些设备设计得更高级并且简单地"宣布"它们要使用的中断. 在这个情况下, 驱动获取中断号通过从设备的一个 I/O 端口或者 PCI 配置空间读一个状态字节. 当目标设备是一个有能力告知驱动它要使用哪个中断的设备时, 自动探测中断号只是意味着探测设备, 探测中断没有其他工作要做. 幸运的是大部分现代硬件这样工作; 例如, PCI 标准解决了这个问题通过要求外设来声明它们要使用哪个中断线. PCI 标准在 12 章讨论.

不幸的是, 不是每个设备是对程序员友好的, 并且自动探测可能需要一些探测. 这个技术非常简单: 驱动告知设备产生中断并且观察发生了什么. 如果所有事情进展地好, 只有一个中断线被激活.

尽管探测在理论上简单的, 实际的实现可能不清晰. 我们看 2 种方法来进行这个任务: 调用内核定义的帮助函数和实现我们自己的版本.

10.2.2.1. 内核协助的探测

Linux 内核提供了一个低级设施来探测中断号. 它只为非共享中断, 但是大部分能够在共享中断状态工作的硬件提供了更好的方法来尽量发现配置的中断号.这个设施包括 2 个函数, 在<linux/interrupt.h> 中声明(也描述了探测机制).

unsigned long probe_irq_on(void);

这个函数返回一个未安排的中断的位掩码. 驱动必须保留返回的位掩码, 并且在后面传递给 probe_irq_off. 在这个调用之后, 驱动应当安排它的设备产生至少一次中断.

int probe_irq_off(unsigned long);

在设备已请求一个中断后, 驱动调用这个函数, 作为参数传递之前由 probe_irq_on 返回的位掩码. probe_irq_off 返回在"probe_on"之后发出的中断号. 如果没有中断发生, 返回 0 (因此, IRQ 0 不能探测, 但是没有用户设备能够在任何支持的体系上使用它). 如果多于一个中断发生(模糊的探测), probe_irq_off 返回一个负值.

程序员应当小心使能设备上的中断, 在调用 probe_irq_on 之后以及在调用 probe_irq_off 后禁止它们. 另外, 你必须记住服务你的设备中挂起的中断, 在 probe_irq_off 之后.

short 模块演示了如何使用这样的探测. 如果你加载模块使用 probe=1, 下列代码被执行来探测你的中断线, 如果并口连接器的管脚 9 和 10 连接在一起:

```
int count = 0;
do
{
    unsigned long mask;
    mask = probe_irq_on();
    outb_p(0x10,short_base+2); /* enable reporting */
    outb_p(0x00,short_base); /* clear the bit */
    outb_p(0xFF,short_base); /* set the bit: interrupt! */
    outb_p(0x00,short_base+2); /* disable reporting */
    udelay(5); /* give it some time */
    short_irq = probe_irq_off(mask);

    if (short_irq == 0) { /* none of them? */
        printk(KERN_INFO "short: no irq reported by probe\n");
        short_irq = -1;
    }

    /*
     * if more than one line has been activated, the result is
     * negative. We should service the interrupt (no need for lpt port)
     * and loop over again. Loop at most five times, then give up
```

```
        */
} while (short_irq < 0 && count++ < 5);
if (short_irq < 0)
        printk("short: probe failed %i times, giving up\n", count);
```

注意 udelay 的使用, 在调用 probe_irq_off 之前. 依赖你的处理器的速度, 你可能不得不等待一小段时间来给中断时间来真正被递交.

探测可能是一个长时间的任务. 虽然对于 short 这不是真的, 例如, 探测一个帧抓取器, 需要一个至少 20 ms 的延时(对处理器是一个时代), 并且其他的设备可能要更长. 因此, 最好只探测中断线一次, 在模块初始化时, 独立于你是否在设备打开时安装处理(如同你应当做的), 或者在初始化函数当中(这个不推荐).

有趣的是注意在一些平台上(PoweerPC, M68K, 大部分 MIPS 实现, 以及 2 个 SPARC 版本)探测是不必要的, 并且, 因此, 之前的函数只是空的占位者, 有时称为"无用的 ISA 废话". 在其他平台上, 探测只为 ISA 设备实现. 无论如何, 大部分体系定义了函数(即便它们是空的)来简化移植现存的设备驱动.

10.2.2.2. Do-it-yourself 探测

探测也可以在驱动自身实现没有太大麻烦. 它是一个少有的驱动必须实现它自己的探测, 但是看它是如何工作的能够给出对这个过程的内部认识. 为此目的, short 模块进行 do-it-yourself 的 IRQ 线探测, 如果它使用 probe=2 加载.

这个机制与前面描述的相同: 使能所有未使用的中断, 接着等待并观察发生什么. 我们能够, 然而, 利用我们对设备的知识. 常常地一个设备能够配置为使用一个 IRQ 号从 3 个或者 4 个一套; 只探测这些 IRQ 使我们能够探测正确的一个, 不必测试所有的可能中断.

short 实现假定 3, 5, 7, 和 9 是唯一可能的 IRQ 值. 这些数实际上是一些并口设备允许你选择的数.

下面的代码通过测试所有"可能的"中断并且查看发生的事情来探测中断. trials 数组列出要尝试的中断, 以 0 作为结尾标志; tried 数组用来跟踪哪个处理实际上被这个驱动注册.

```
int trials[] =
        {
                3, 5, 7, 9, 0
        };
int tried[]  = {0, 0, 0, 0, 0};
int i, count = 0;

/*
 * install the probing handler for all possible lines. Remember
 * the result (0 for success, or -EBUSY) in order to only free
```

```
 * what has been acquired */
for (i = 0; trials[i]; i++)
        tried[i] = request_irq(trials[i], short_probing,
                               SA_INTERRUPT, "short probe", NULL);

do
{
        short_irq = 0; /* none got, yet */
        outb_p(0x10,short_base+2); /* enable */
        outb_p(0x00,short_base);
        outb_p(0xFF,short_base); /* toggle the bit */
        outb_p(0x00,short_base+2); /* disable */
        udelay(5); /* give it some time */

        /* the value has been set by the handler */
        if (short_irq == 0) { /* none of them? */

                printk(KERN_INFO "short: no irq reported by probe\n");
        }
        /*
        * If more than one line has been activated, the result is
        * negative. We should service the interrupt (but the lpt port
        * doesn't need it) and loop over again. Do it at most 5 times
        */
} while (short_irq <=0 && count++ < 5);

/* end of loop, uninstall the handler */
for (i = 0; trials[i]; i++)
        if (tried[i] == 0)
                free_irq(trials[i], NULL);

if (short_irq < 0)
        printk("short: probe failed %i times, giving up\n", count);
```

你可能事先不知道"可能的" IRQ 值是什么. 在这个情况, 你需要探测所有空闲的中断, 不是限制你自己在几个 trials[]. 为探测所有的中断, 你不得不从 IRQ 0 到 IRQ NR_IRQS-1 探测, 这里 NR_IRQS 在 <asm/irq.h> 中定义并且是独立于平台的.

现在我们只缺少探测处理自己了. 处理者的角色是更新 short_irq, 根据实际收到哪个中断. short_irq 中的 0 值意味着"什么没有", 而一个负值意味着"模糊的". 这些值选择来和 probe_irq_off 相一致并且允许同样的代码来调用任一种 short.c 中的探测.

```
irqreturn_t short_probing(int irq, void *dev_id, struct pt_regs *regs)
```

```
{

   if (short_irq == 0) short_irq = irq;  /* found */
 if (short_irq != irq) short_irq = -irq; /* ambiguous */
 return IRQ_HANDLED;
}
```

处理的参数在后面描述. 知道 irq 是在处理的中断应当是足够的来理解刚刚展示的函数.

10.2.3. 快速和慢速处理

老版本的 Linux 内核尽了很大努力来区分"快速"和"慢速"中断. 快速中断是那些能够很快处理的, 而处理慢速中断要特别地长一些. 慢速中断可能十分苛求处理器, 并且它值得在处理的时候重新使能中断. 否则, 需要快速注意的任务可能被延时太长.

在现代内核中, 快速和慢速中断的大部分不同已经消失. 剩下的仅仅是一个: 快速中断(那些使用 SA_INTERRUPT 被请求的)执行时禁止所有在当前处理器上的其他中断. 注意其他的处理器仍然能够处理中断, 尽管你从不会看到 2 个处理器同时处理同一个 IRQ.

这样, 你的驱动应当使用哪个类型的中断? 在现代系统上, SA_INTERRUPT 只是打算用在几个, 特殊的情况例如时钟中断. 除非你有一个充足的理由来运行你的中断处理在禁止其他中断情况下, 你不应当使用 SA_INTERRUPT.

这个描述应当满足大部分读者, 尽管有人喜好硬件并且对她的计算机有经验可能有兴趣深入一些. 如果你不关心内部的细节, 你可跳到下一节.

10.2.3.1. x86上中断处理的内幕

这个描述是从 arch/i386/kernel/irq.c, arch/i386/kernel/ apic.c, arch/i386/kernel/entry.S, arch/i386/kernel/ i8259.c, 和 include/asm-i386/hw_irq.h 它们出现于 2.6 内核而推知的; 尽管一般的概念保持一致, 硬件细节在其他平台上不同.

中断处理的最低级是在 entry.S, 一个汇编语言文件处理很多机器级别的工作. 通过一点汇编器的技巧和一些宏定义, 一点代码被安排到每个可能的中断. 在每个情况下, 这个代码将中断号压栈并且跳转到一个通用段, 称为 do_IRQ, 在 irq.c 中定义.

do_IRQ 做的第一件事是确认中断以便中断控制器能够继续其他事情. 它接着获取给定 IRQ 号的一个自旋锁, 因此阻止任何其他 CPU 处理这个 IRQ. 它清除几个状态位(包括称为 IRQ_WAITING 的一个, 我们很快会看到它)并且接着查看这个特殊 IRQ 的处理者. 如果没有处理者, 什么不作; 自旋锁释放, 任何挂起的软件中断被处理, 最后 do_IRQ 返回.

常常, 但是, 如果一个设备在中断, 至少也有一个处理者注册给它的 IRQ. 函数 handle_IRQ_event 被

调用来实际调用处理者. 如果处理者是慢速的(SA_INTERRUPT 没有设置), 中断在硬件中被重新使能, 并且调用处理者. 接着仅仅是清理, 运行软件中断, 以及回到正常的工作. "常规工作"很可能已经由于中断而改变了(处理者可能唤醒一个进程, 例如), 因此从中断中返回的最后的事情是一个处理器的可能的重新调度.

探测 IRQ 通过设置 IRQ_WAITING 状态位给每个当前缺乏处理者的 IRQ 来完成. 当中断发生, do_IRQ 清除这个位并且接着返回, 因为没有注册处理者. probe_irq_off, 当被一个函数调用, 需要只搜索不再有 IRQ_WAITING 设置的 IRQ.

10.2.4. 实现一个处理

至今, 我们已学习了注册一个中断处理, 但是没有编写一个. 实际上, 对于一个处理者, 没什么不寻常的 -- 它是普通的 C 代码.

唯一的特别之处是一个处理者在中断时运行, 因此, 它能做的事情遭受一些限制. 这些限制与我们在内核定时器上看到的相同. 一个处理者不能传递数据到或者从用户空间, 因为它不在进程上下文执行. 处理者也不能做任何可能睡眠的事情, 例如调用 wait_event, 使用除 GFP_ATOMIC 之外任何东西来分配内存, 或者加锁一个旗标. 最后, 处理者不能调用调度.

一个中断处理的角色是给它的设备关于中断接收的回应并且读或写数据, 根据被服务的中断的含义. 第一步常常包括清除接口板上的一位; 大部分硬件设备不产生别的中断直到它们的"中断挂起"位被清除. 根据你的硬件如何工作的, 这一步可能需要在最后做而不是开始; 这里没有通吃的规则. 一些设备不需要这步, 因为它们没有一个"中断挂起"位; 这样的设备是一少数, 尽管并口是其中之一. 由于这个理由, short 不必清除这样一个位.

一个中断处理的典型任务是唤醒睡眠在设备上的进程, 如果中断指示它们在等待的事件, 例如新数据的到达.

为坚持帧抓取者的例子, 一个进程可能请求一个图像序列通过连续读设备; 读调用阻塞在读取每个帧之前, 而中断处理唤醒进程一旦每个新帧到达. 这个假定抓取器中断处理器来指示每个新帧的成功到达.

程序员应当小心编写一个函数在最小量的时间内执行, 不管是一个快速或慢速处理者. 如果需要进行长时间计算, 最好的方法是使用一个 tasklet 或者 workqueue 来调度计算在一个更安全的时间(我们将在"上和下半部"一节中见到工作如何被延迟.).

我们在 short 中的例子代码响应中断通过调用 do_gettimeofday 和 打印当前时间到一个页大小的环形缓存. 它接着唤醒任何读进程, 因为现在有数据可用来读取.

```
irqreturn_t short_interrupt(int irq, void *dev_id, struct pt_regs *regs)
    {
        struct timeval tv;
```

```
        int written;
        do_gettimeofday(&tv);
        /* Write a 16 byte record. Assume PAGE_SIZE is a multiple of 16 */
        written = sprintf((char *)short_head,"%08u.%06u\n",
                          (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
        BUG_ON(written != 16);
        short_incr_bp(&short_head, written);
        wake_up_interruptible(&short_queue); /* awake any reading process */
        return IRQ_HANDLED;
    }
```

这个代码, 尽管简单, 代表了一个中断处理的典型工作. 依次地, 它称为 short_incr_bp, 定义如下:

```
static inline void short_incr_bp(volatile unsigned long *index, int delta)
{
    unsigned long new = *index + delta;
    barrier();  /* Don't optimize these two together */
    *index = (new >= (short_buffer + PAGE_SIZE)) ? short_buffer : new;
}
```

这个函数已经仔细编写来回卷指向环形缓存的指针, 没有暴露一个不正确的值. 这里的 barrier 调用来阻止编译器在这个函数的其他 2 行之间优化. 如果没有 barrier, 编译器可能决定优化掉 new 变量并且直接赋值给 *index. 这个优化可能暴露一个 index 的不正确值一段时间, 在它回卷的地方. 通过小心阻止对其他线程可见的不一致的值, 我们能够安全操作环形缓存指针而不用锁.

用来读取中断时填充的缓存的设备文件是 /dev/shortint. 这个设备特殊文件, 同 /dev/shortprint 一起, 不在第 9 章介绍, 因为它的使用对中断处理是特殊的. /dev/shortint 内部特别地为中断产生和报告剪裁过. 写到设备会每隔一个字节产生一个中断; 读取设备给出了每个中断被报告的时间.

如果你连接并口连接器的管脚 9 和 10, 你可产生中断通过拉高并口数据字节的高位. 这可通过写二进制数据到 /dev/short0 或者通过写任何东西到 /dev/shortint 来完成.

[38]下列代码为 /dev/shortint 实现读和写:

```
ssize_t short_i_read (struct file *filp, char __user *buf, size_t count,
                      loff_t *f_pos)
{
    int count0;
    DEFINE_WAIT(wait);

    while (short_head == short_tail)
    {
```

```
        prepare_to_wait(&short_queue, &wait, TASK_INTERRUPTIBLE);
        if (short_head == short_tail)

            schedule();
        finish_wait(&short_queue, &wait);
        if (signal_pending (current)) /* a signal arrived */
            return -ERESTARTSYS; /* tell the fs layer to handle it */
    } /* count0 is the number of readable data bytes */ count0 = short_head - short_tail;
    if (count0 < 0) /* wrapped */
        count0 = short_buffer + PAGE_SIZE - short_tail;
    if (count0 < count)
        count = count0;

    if (copy_to_user(buf, (char *)short_tail, count))
        return -EFAULT;
    short_incr_bp (&short_tail, count);
    return count;

}
ssize_t short_i_write (struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
    int written = 0, odd = *f_pos & 1;
    unsigned long port = short_base; /* output to the parallel data latch */
    void *address = (void *) short_base;

    if (use_mem)
    {
        while (written < count)
            iowrite8(0xff * ((++written + odd) & 1), address);
    } else
    {

        while (written < count)
            outb(0xff * ((++written + odd) & 1), port);
    }

    *f_pos += count;
    return written;
}
```

其他设备特殊文件, /dev/shortprint, 使用并口来驱动一个打印机; 你可用使用它, 如果你想避免连接一个 D-25 连接器管脚 9 和 10. shortprint 的写实现使用一个环形缓存来存储要打印的数据, 而写实现是刚刚展示的那个(因此你能够读取你的打印机吃进每个字符用的时间).

为了支持打印机操作, 中断处理从刚刚展示的那个已经稍微修改, 增加了发送下一个数据字节到打印机的能力, 如果没有更多数据传送.

10.2.5. 处理者的参数和返回值

尽管 short 忽略了它们, 一个传递给一个中断处理的参数: irq, dev_id, 和 regs. 我们看一下每个的角色.

中断号(int irq)作为你可能在你的 log 消息中打印的信息是有用的, 如果有. 第二个参数, void *dev_id, 是一类客户数据; 一个 void* 参数传递给 request_irq, 并且同样的指针接着作为一个参数传回给处理者, 当中断发生时. 你常常传递一个指向你的在 dev_id 中的设备数据结构的指针, 因此一个管理相同设备的几个实例的驱动不需要任何额外的代码, 在中断处理中找出哪个设备要负责当前的中断事件.

这个参数在中断处理中的典型使用如下:

```
static irqreturn_t sample_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct sample_dev *dev = dev_id;

    /* now `dev' points to the right hardware item */
    /* .... */
}
```

和这个处理者关联的典型的打开代码看来如此:

```
static void sample_open(struct inode *inode, struct file *filp)
{
    struct sample_dev *dev = hwinfo + MINOR(inode->i_rdev);
    request_irq(dev->irq, sample_interrupt,

            0 /* flags */, "sample", dev /* dev_id */);
    /*....*/
    return 0;

}
```

最后一个参数, struct pt_regs *regs, 很少用到. 它持有一个处理器的上下文在进入中断状态前的快照. 寄存器可用来监视和调试; 对于常规地设备驱动任务, 正常地不需要它们.

中断处理应当返回一个值指示是否真正有一个中断要处理. 如果处理者发现它的设备确实需要注

意, 它应当返回 IRQ_HANDLED; 否则返回值应当是 IRQ_NONE. 你也可产生返回值, 使用这个宏:

IRQ_RETVAL(handled)

这里, handled 是非零, 如果你能够处理中断. 内核用返回值来检测和抑制假中断. 如果你的设备没有给你方法来告知是否它确实中断, 你应当返回 IRQ_HANDLED.

10.2.6. 使能和禁止中断

有时设备驱动必须阻塞中断的递交一段时间(希望地短)(我们在第 5 章的 "自旋锁"一节看到过这样的一个情况). 常常, 中断必须被阻塞当持有一个自旋锁来避免死锁系统时. 有几个方法来禁止不涉及自旋锁的中断. 但是在我们讨论它们之前, 注意禁止中断应当是一个相对少见的行为, 即便在设备驱动中, 并且这个技术应当从不在驱动中用做互斥机制.

10.2.6.1. 禁止单个中断

有时(但是很少!)一个驱动需要禁止一个特定中断线的中断递交. 内核提供了 3 个函数为此目的, 所有都声明在 <asm/irq.h>. 这些函数是内核 API 的一部分, 因此我们描述它们, 但是它们的使用在大部分驱动中不鼓励. 在其他的中, 你不能禁止共享的中断线, 并且, 在现代的系统中, 共享的中断是规范. 已说过的, 它们在这里:

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

调用任一函数可能更新在可编程控制器(PIC)中的特定 irq 的掩码, 因此禁止或使能跨所有处理器的特定 IRQ. 对这些函数的调用能够嵌套 -- 如果 disable_irq 被连续调用 2 次, 需要 2 个 enable_irq 调用在 IRQ 被真正重新使能前. 可能调用这些函数从一个中断处理中, 但是在处理它时使能你自己的 IRQ 常常不是一个好做法.

disable_irq 不仅禁止给定的中断, 还等待一个当前执行的中断处理结束, 如果有. 要知道如果调用 disable_irq 的线程持有中断处理需要的任何资源(例如自旋锁), 系统可能死锁. disable_irq_nosync 与 disable_irq 不同, 它立刻返回. 因此, 使用disable_irq_nosync 快一点, 但是可能使你的设备有竞争情况.

但是为什么禁止中断? 坚持说并口, 我们看一下 plip 网络接口. 一个 plip 设备使用裸并口来传送数据. 因为只有 5 位可以从并口连接器读出, 它们被解释为 4 个数据位和一个时钟/握手信号. 当一个报文的第一个 4 位被 initiator (发送报文的接口) 传送, 时钟线被拉高, 使接收接口来中断处理器. plip 处理者接着被调用来处理新到达的数据.

在设备已经被提醒了后, 数据传送继续, 使用握手线来传送数据到接收接口(这可能不是最好的实现, 但是有必要与使用并口的其他报文驱动兼容). 如果接收接口不得不为每个接收的字节处理 2 次中断, 性能可能不可忍受. 因此, 驱动在接收报文的时候禁止中断; 相反, 一个查询并延时的循环用来

引入数据.

类似地, 因为从接收器到发送器的握手线用来确认数据接收, 发送接口禁止它的 IRQ 线在报文发送时.

10.2.6.2. 禁止所有中断

如果你需要禁止所有中断如何? 在 2.6 内核, 可能关闭在当前处理器上所有中断处理, 使用任一个下面 2 个函数(定义在 <asm/system.h>):

```
void local_irq_save(unsigned long flags);
void local_irq_disable(void);
```

一个对 local_irq_save 的调用在当前处理器上禁止中断递交, 在保存当前中断状态到 flags 之后. 注意, flags 是直接传递, 不是通过指针. local_irq_disable 关闭本地中断递交而不保存状态; 你应当使用这个版本只在你知道中断没有在别处被禁止.

完成打开中断, 使用:

```
void local_irq_restore(unsigned long flags);
void local_irq_enable(void);
```

第一个版本恢复由 local_irq_save 存储于 flags 的状态, 而 local_irq_enable 无条件打开中断. 不象 disable_irq, local_irq_disable 不跟踪多次调用. 如果调用链中有多于一个函数可能需要禁止中断, 应该使用 local_irq_save.

在 2.6 内核, 没有方法全局性地跨整个系统禁止所有的中断. 内核开发者决定, 关闭所有中断的开销太高, 并且在任何情况下没有必要有这个能力. 如果你在使用一个旧版本驱动, 它调用诸如 cli 和 sti, 你需要在它在 2.6 下工作前更新它为使用正确的加锁

---

[37] 尽管, 一些大系统明确使用中断平衡机制来在系统间分散中断负载.

[38] 这个 shortint 设备完成它的任务, 通过交替地写入 0x00 和 0xff 到并口.

10.3. 前和后半部

中断处理的一个主要问题是如何在处理中进行长时间的任务. 常常大量的工作必须响应一个设备中断来完成, 但是中断处理需要很快完成并且不使中断阻塞太长. 这 2 个需要(工作和速度)彼此冲突, 留给驱动编写者一点困扰.

Linux (许多其他系统一起)解决这个问题通过将中断处理分为 2 半. 所谓的前半部是实际响应中断的函数 -- 你使用 request_irq 注册的那个. 后半部是由前半部调度来延后执行的函数, 在一个更安全的时间. 最大的不同在前半部处理和后半部之间是所有的中断在后半部执行时都使能 -- 这就是为什么它在一个更安全时间运行. 在典型的场景中, 前半部保存设备数据到一个设备特定的缓存, 调度它的后半部, 并且退出: 这个操作非常快. 后半部接着进行任何其他需要的工作, 例如唤醒进程, 启动另一个 I/O 操作, 等等. 这种设置允许前半部来服务一个新中断而同时后半部仍然在工作.

几乎每个认真的中断处理都这样划分. 例如, 当一个网络接口报告有新报文到达, 处理者只是获取数据并且上推给协议层; 报文的实际处理在后半部进行.

Linux 内核有 2 个不同的机制可用来实现后半部处理, 我们都在第 7 章介绍. tasklet 常常是后半部处理的首选机制; 它们非常快, 但是所有的 tasklet 代码必须是原子的. tasklet 的可选项是工作队列, 它可能有一个更高的运行周期但是允许睡眠.

下面的讨论再次使用 short 驱动. 当使用一个模块选项加载时, short 能够被告知在前/后半部模式使用一个 tasklet 或者工作队列处理者来进行中断处理. 在这个情况下, 前半部快速地执行; 它简单地记住当前时间并且调度后半部处理. 后半部接着负责将时间编码并且唤醒任何可能在等待数据的用户进程.

10.3.1. Tasklet 实现

记住 tasklet 是一个特殊的函数, 可能被调度来运行, 在软中断上下文, 在一个系统决定的安全时间中. 它们可能被调度运行多次, 但是 tasklet 调度不累积; ; tasklet 只运行一次, 即便它在被投放前被重复请求. 没有 tasklet 会和它自己并行运行, 因为它只运行一次, 但是 tasklet 可以与 SMP 系统上的其他 tasklet 并行运行. 因此, 如果你的驱动有多个 tasklet, 它们必须采取某类加锁来避免彼此冲突.

tasklet 也保证作为函数运行在第一个调度它们的同一个 CPU 上. 因此, 一个中断处理可以确保一个 tasklet 在处理者结束前不会开始执行. 但是, 另一个中断当然可能在 tasklet 在运行时被递交, 因此, tasklet 和中断处理之间加锁可能仍然需要.

tasklet 必须使用 DECLARE_TASKLET 宏来声明:

DECLARE_TASKLET(name, function, data);

name 是给 tasklet 的名子, function 是调用来执行 tasklet (它带一个 unsigned long 参数并且返回 void) 的函数, 以及 data 是一个 unsigned long 值来传递给 tasklet 函数.

short 驱动声明它的 tasklet 如下:

```
void short_do_tasklet(unsigned long);
DECLARE_TASKLET(short_tasklet, short_do_tasklet, 0);
```

函数 tasklet_schedule 用来调度一个 tasklet 运行. 如果 short 使用 tasklet=1 来加载, 它安装一个不同的中断处理来保存数据并且调度 tasklet 如下:

```
irqreturn_t short_tl_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    do_gettimeofday((struct timeval *) tv_head); /* cast to stop 'volatile' warning
                */
    short_incr_tv(&tv_head);
    tasklet_schedule(&short_tasklet);
    short_wq_count++; /* record that an interrupt arrived */
    return IRQ_HANDLED;
}
```

实际的 tasklet 函数, short_do_tasklet, 将在系统方便时很快执行. 如同前面提过, 这个函数进行处理中断的大量工作; 它看来如此:

```
void short_do_tasklet (unsigned long unused)
{
    int savecount = short_wq_count, written;
    short_wq_count = 0; /* we have already been removed from the queue */
    /*
    * The bottom half reads the tv array, filled by the top half,
    * and prints it to the circular text buffer, which is then consumed
    * by reading processes */
    /* First write the number of interrupts that occurred before this bh */
    written = sprintf((char *)short_head,"bh after %6i\n",savecount);
    short_incr_bp(&short_head, written);
    /*
    * Then, write the time values. Write exactly 16 bytes at a time,
    * so it aligns with PAGE_SIZE */
```

```
    do {
        written = sprintf((char *)short_head,"%08u.%06u\n",
                (int)(tv_tail->tv_sec % 100000000),
                (int)(tv_tail->tv_usec));
        short_incr_bp(&short_head, written);
        short_incr_tv(&tv_tail);
    } while (tv_tail != tv_head);

    wake_up_interruptible(&short_queue); /* awake any reading process */
}
```

在别的东西中, 这个 tasklet 记录了从它上次被调用以来有多少中断到达. 一个如 short 一样的设备能够在短时间内产生大量中断, 因此在后半部执行前有几个中断到达就不是不寻常的. 驱动必须一直准备这种可能性并且必须能够从前半部留下的信息中决定有多少工作要做.

10.3.2. 工作队列

回想, 工作队列在将来某个时候调用一个函数, 在一个特殊工作者进程的上下文中. 因为这个工作队列函数在进程上下文运行, 它在需要时能够睡眠. 但是, 你不能从一个工作队列拷贝数据到用户空间, 除非你使用我们在 15 章演示的高级技术; 工作者进程不存取任何其他进程的地址空间.

short 驱动, 如果设置 wq 选项为一个非零值来加载, 为它的后半部处理使用一个工作队列. 它使用系统缺省的工作队列, 因此不要求特殊的设置代码; 如果你的驱动有特别的运行周期要求(或者可能在工作队列函数长时间睡眠), 你可能需要创建你自己的, 专用的工作队列. 我们确实需要一个 work_struct 结构, 它声明和初始化使用下列:

```
static struct work_struct short_wq;
/* this line is in short_init() */
INIT_WORK(&short_wq, (void (*)(void *)) short_do_tasklet, NULL);
```

我们的工作者函数是 short_do_tasklet, 我们已经在前面一节看到.

当使用一个工作队列, short 还建立另一个中断处理, 看来如此:

```
irqreturn_t short_wq_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    /* Grab the current time information. */
    do_gettimeofday((struct timeval *) tv_head);
    short_incr_tv(&tv_head);
    /* Queue the bh. Don't worry about multiple enqueueing */
    schedule_work(&short_wq);
    short_wq_count++; /* record that an interrupt arrived */
```

```
    return IRQ_HANDLED;
}
```

如你所见, 中断处理看来非常象这个 tasklet 版本, 除了它调用 schedule_work 来安排后半部处理.

上一页 | 上一级 | 下一页
10.2. 安装一个中断处理 | 起始页 | 10.4. 中断共享

10.4. 中断共享

中断冲突的概念几乎是 PC 体系的同义词. 过去, 在 PC 上的 IRQ 线不能服务多于一个设备, 并且它们从不足够. 结果, 失望的用户花费大量时间开着它们的计算机, 尽力找到一个方法来使它们所有的外设一起工作.

现代的硬件, 当然, 已经设计来允许中断共享; PCI 总线要求它. 因此, Linux 内核支持在所有总线上中断共享, 甚至是那些(例如 ISA 总线)传统上不被支持的. 2.6 内核的设备驱动应当编写来使用共享中断, 如果目标硬件能够支持这个操作模式. 幸运的是, 使用共享中断在大部分时间是容易的.

10.4.1. 安装一个共享的处理者

共享中断通过 request_irq 来安装就像不共享的一样, 但是有 2 个不同:

- SA_SHIRQ 位必须在 flags 参数中指定, 当请求中断时.
- dev_id 参数必须是独特的. 任何模块地址空间的指针都行, 但是 dev_id 明确地不能设置为 NULL.

内核保持着一个与中断相关联的共享处理者列表, 并且 dev_id 可认为是区别它们的签名. 如果 2 个驱动要在同一个中断上注册 NULL 作为它们的签名, 在卸载时事情可能就乱了, 在中断到的时候引发内核 oops. 由于这个理由, 如果在注册共享中断时传给了一个 NULL dev_id , 现代内核会大声抱怨. 当请求一个共享的中断, request_irq 成功, 如果下列之一是真:

- 中断线空闲.
- 所有这条线的已经注册的处理者也指定共享这个 IRQ.

无论何时 2 个或多个驱动在共享中断线, 并且硬件中断在这条线上中断处理器, 内核为这个中断调用每个注册的处理者, 传递它的 dev_id 给每个. 因此, 一个共享的处理者必须能够识别它自己的中断并且应当快速退出当它自己的设备没有被中断时. 确认返回 IRQ_NONE 无论何时你的处理者被调用并且发现设备没被中断.

如果你需要探测你的设备, 在请求 IRQ 线之前, 内核无法帮你. 没有探测函数可给共享处理者使用. 标准的探测机制有效如果使用的线是空闲的, 但是如果这条线已经被另一个有共享能力的驱动持有, 探测失败, 即便你的驱动已正常工作. 幸运的是, 大部分设计为中断共享的硬件能够告知处理器它在使用哪个中断, 因此减少明显的探测的需要.

释放处理者以正常方式进行, 使用 free_irq. 这里 dev_id 参数用来从这个中断的共享处理者列表中选择正确的处理者来释放. 这就是为什么 dev_id 指针必须是独特的.

一个使用共享处理者的驱动需要小心多一件事: 它不能使用 enable_irq 或者 disable_irq. 如果它用了, 对其他共享这条线的设备就乱了; 禁止另一个设备的中断即便短时间也可能产生延时, 这对这个设备和它的用户是有问题的. 通常, 程序员必须记住, 他的驱动不拥有这个 IRQ, 并且它的行为应当比它拥有这个中断线更加"社会性".

10.4.2. 运行处理者

如同前面建议的, 当内核收到一个中断, 所有的注册的处理者被调用. 一个共享的处理者必须能够在它需要的处理的中断和其他设备产生的中断之间区分.

使用 shared=1 选项来加载 short 安装了下列处理者来代替缺省的:

```
irqreturn_t short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    int value, written;
    struct timeval tv;
    /* If it wasn't short, return immediately */
    value = inb(short_base);
    if (!(value & 0x80))
        return IRQ_NONE;
    /* clear the interrupting bit */
    outb(value & 0x7F, short_base);
    /* the rest is unchanged */
    do_gettimeofday(&tv);
    written = sprintf((char *)short_head,"%08u.%06u\n",
                (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
    short_incr_bp(&short_head, written);
    wake_up_interruptible(&short_queue); /* awake any reading process */
    return IRQ_HANDLED;
}
```

这里应该有个解释. 因为并口没有"中断挂起"位来检查, 处理者使用 ACK 位作此目的. 如果这个位是高, 正报告的中断是给 short, 并且这个处理者清除这个位.

处理者通过并口数据端口的清零来复位这个位 -- short 假设管脚 9 和 10 连在一起. 如果其他一个和 short 共享这个 IRQ 的设备产生一个中断, short 看到它的线仍然非激活并且什么不作.

当然, 一个功能齐全的驱动可能将工作划分位前和后半部, 但是容易添加并且不会有任何影响实现共享的代码. 一个真实驱动还可能使用 dev_id 参数来决定, 在很多可能的中, 哪个设备在中断.

注意, 如果你使用打印机(代替跳线)来测试使用 short 的中断管理, 这个共享的处理者不象所说的一样工作,因为打印机协议不允许共享, 并且驱动不知道是否这个中断是来自打印机.

10.4.3. /proc 接口和共享中断

在系统中安装共享处理者不影响 /proc/stat, 它甚至不知道处理者. 但是, /proc/interrupts 稍稍变化.

所有同一个中断号的安装的处理者出现在 /proc/interrupts 的同一行. 下列输出(从一个 x86_64 系统)显示了共享中断处理是如何显示的:

```
           CPU0
  0: 892335412 XT-PIC timer
  1: 453971 XT-PIC i8042
  2: 0 XT-PIC cascade
  5: 0 XT-PIC libata, ehci_hcd
  8: 0 XT-PIC rtc
  9: 0 XT-PIC acpi
 10: 11365067 XT-PIC ide2, uhci_hcd, uhci_hcd, SysKonnect SK-98xx, EMU10K1
 11: 4391962 XT-PIC uhci_hcd, uhci_hcd
 12: 224 XT-PIC i8042
 14: 2787721 XT-PIC ide0
 15: 203048 XT-PIC ide1
NMI: 41234
LOC: 892193503
ERR: 102
MIS: 0
```

这个系统有几个共享中断线. IRQ 5 用来给串行 ATA 和 IEEE 1394 控制器; IRQ 10 有几个设备, 包括一个 IDE 控制器, 2 个 USB 控制器, 一个以太网接口, 以及一个声卡; 并且 IRQ 11 也被 2 个 USB 控制器使用.

10.5. 中断驱动 I/O

无论何时一个数据传送到或自被管理的硬件可能因为任何原因而延迟, 驱动编写者应当实现缓存. 数据缓存帮助来分离数据传送和接收从写和读系统调用, 并且整个系统性能受益.

一个好的缓存机制产生了中断驱动的 I/O, 一个输入缓存在中断时填充并且被读取设备的进程清空; 一个输出缓存由写设备的进程填充并且在中断时清空. 一个中断驱动的输出的例子是 /dev/shortprint 的实现.

为使中断驱动的数据传送成功发生, 硬件应当能够产生中断, 使用下列语义:

- 对于输入, 设备中断处理器, 当新数据到达时, 并且准备好被系统处理器获取. 进行的实际动作依赖是否设备使用 I/O 端口, 内存映射, 或者 DMA.
- 对于输出, 设备递交一个中断, 或者当它准备好接受新数据, 或者确认一个成功的数据传送. 内存映射的和能DMA的设备常常产生中断来告诉系统它们完成了这个缓存.

在一个读或写与实际数据到达之间的时间关系在第 6 章的"阻塞和非阻塞操作"一节中介绍.

10.5.1. 一个写缓存例子

我们已经几次提及 shortprint 驱动; 现在是时候真正看看. 这个模块为并口实现一个非常简单, 面向输出的驱动; 它是足够的, 但是, 来使能文件打印. 如果你选择来测试这个驱动, 但是, 记住你必须传递给打印机一个文件以它理解的格式; 不是所有的打印机在给一个任意数据的流时很好响应.

shortprint 驱动维护一个一页的环形输出缓存. 当一个用户空间进程写数据到这个设备, 数据被填入缓存, 但是写方法实际没有进行任何 I/O. 相反, shortp_write 的核心看来如此:

```
while (written < count)
{
    /* Hang out until some buffer space is available. */
    space = shortp_out_space();
    if (space <= 0) {
        if (wait_event_interruptible(shortp_out_queue,
                                     (space = shortp_out_space()) > 0))
            goto out;
    }
```

```
    /* Move data into the buffer. */
    if ((space + written) > count)
        space = count - written;

    if (copy_from_user((char *) shortp_out_head, buf, space)) {
        up(&shortp_out_sem);
        return -EFAULT;
    }
    shortp_incr_out_bp(&shortp_out_head, space);
    buf += space;
    written += space;

    /* If no output is active, make it active. */
    spin_lock_irqsave(&shortp_out_lock, flags);
    if (! shortp_output_active)
        shortp_start_output();
    spin_unlock_irqrestore(&shortp_out_lock, flags);
}

out:
*f_pos += written;
```

一个旗标 (shortp_out_sem) 控制对这个环形缓存的存取; shortp_write 就在上面的代码片段之前获得这个旗标. 当持有这个旗标, 它试图输入数据到这个环形缓存. 函数 shortp_out_space 返回可用的连续空间的数量(因此, 没有必要担心缓存回绕); 如果这个量是 0, 驱动等到释放一些空间. 它接着拷贝它能够的数量的数据到缓存中.

一旦有数据输出, shortp_write 必须确保数据被写到设备. 数据的写是通过一个工作队列函数完成的; shortp_write 必须启动这个函数如果它还未在运行. 在获取了一个单独的, 控制存取输出缓存的消费者一侧(包括 shortp_output_active)的数据的自旋锁后, 它调用 shortp_start_output 如果需要. 接着只是注意多少数据被写到缓存并且返回.

启动输出进程的函数看来如下:

```
static void shortp_start_output(void)
{
    if (shortp_output_active) /* Should never happen */
        return;

    /* Set up our 'missed interrupt' timer */
    shortp_output_active = 1;
    shortp_timer.expires = jiffies + TIMEOUT;
    add_timer(&shortp_timer);
```

```
        /* And get the process going. */
        queue_work(shortp_workqueue, &shortp_work);
}
```

处理硬件的事实是, 你可以, 偶尔, 丢失来自设备的中断. 当发生这个, 你确实不想你的驱动一直停止直到系统重启; 这不是一个用户友好的做事方式. 最好是认识到一个中断已经丢失, 收拾残局, 继续. 为此, shortprint 甚至一个内核定时器无论何时它输出数据给设备. 如果时钟超时, 我们可能丢失一个中断. 我们很快会看到定时器函数, 但是, 暂时, 让我们坚持在主输出功能上. 那是在我们的工作队列函数里实现的, 它, 如同你上面看到的, 在这里被调度. 那个函数的核心看来如下:

```
spin_lock_irqsave(&shortp_out_lock, flags);
/* Have we written everything? */
if (shortp_out_head == shortp_out_tail)
{ /* empty */
        shortp_output_active = 0;
        wake_up_interruptible(&shortp_empty_queue);
        del_timer(&shortp_timer);

}
/* Nope, write another byte */
else
        shortp_do_write();
/* If somebody's waiting, maybe wake them up. */
if (((PAGE_SIZE + shortp_out_tail -shortp_out_head) % PAGE_SIZE) > SP_MIN_SPACE)
{
        wake_up_interruptible(&shortp_out_queue);
}
spin_unlock_irqrestore(&shortp_out_lock, flags);
```

因为我们在使用共享变量的输出一侧, 我们必须获得自旋锁. 接着我们看是否有更多的数据要发送; 如果无, 我们注意输出不再激活, 删除定时器, 并且唤醒任何在等待队列全空的进程(这种等待当设备被关闭时结束). 如果, 相反, 有数据要写, 我们调用 shortp_do_write 来实际发送一个字节到硬件.

接着, 因为我们可能在输出缓存中有空闲空间, 我们考虑唤醒任何等待增加更多数据给那个缓存的进程. 但是我们不是无条件进行唤醒; 相反, 我们等到有一个最低数量的空间. 每次我们从缓存拿出一个字节就唤醒一个写者是无意义的; 唤醒进程的代价, 调度它运行, 并且使它重回睡眠, 太高了. 相反, 我们应当等到进程能够立刻移动相当数量的数据到缓存. 这个技术在缓存的, 中断驱动的驱动中是普通的.

为完整起见, 这是实际写数据到端口的代码:

```
static void shortp_do_write(void)
{
    unsigned char cr = inb(shortp_base + SP_CONTROL);
    /* Something happened; reset the timer */
    mod_timer(&shortp_timer, jiffies + TIMEOUT);
    /* Strobe a byte out to the device */
    outb_p(*shortp_out_tail, shortp_base+SP_DATA);
    shortp_incr_out_bp(&shortp_out_tail, 1);
    if (shortp_delay)
        udelay(shortp_delay);
    outb_p(cr | SP_CR_STROBE, shortp_base+SP_CONTROL);
    if (shortp_delay)
        udelay(shortp_delay);
    outb_p(cr & ~SP_CR_STROBE, shortp_base+SP_CONTROL);
}
```

这里, 我们复位定时器来反映一个事实, 我们已经作了一些处理, 输送字节到设备, 并且更新了环形缓存指针.

工作队列函数没有直接重新提交它自己, 因此只有一个单个字节会被写入设备. 在某一处, 打印机将, 以它的缓慢方式, 消耗这个字节并且准备好下一个; 它将接着中断处理器. shortprint 中使用的中断处理是简短的:

```
static irqreturn_t shortp_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    if (! shortp_output_active)
        return IRQ_NONE;
    /* Remember the time, and farm off the rest to the workqueue function */
    do_gettimeofday(&shortp_tv);
    queue_work(shortp_workqueue, &shortp_work);
    return IRQ_HANDLED;
}
```

因为并口不要求一个明显的中断确认, 中断处理所有真正需要做的是告知内核来再次运行工作队列函数.

如果中断永远不来如何? 至此我们已见到的驱动代码将简单地停止. 为避免发生这个, 我们设置了一个定时器在几页前. 当定时器超时运行的函数是:

```
static void shortp_timeout(unsigned long unused)
{
    unsigned long flags;
```

```
    unsigned char status;
    if (! shortp_output_active)
        return;
    spin_lock_irqsave(&shortp_out_lock, flags);
    status = inb(shortp_base + SP_STATUS);

    /* If the printer is still busy we just reset the timer */
    if ((status & SP_SR_BUSY) == 0 || (status & SP_SR_ACK)) {

        shortp_timer.expires = jiffies + TIMEOUT;
        add_timer(&shortp_timer);
        spin_unlock_irqrestore(&shortp_out_lock, flags);
        return;
    }
    /* Otherwise we must have dropped an interrupt. */
    spin_unlock_irqrestore(&shortp_out_lock, flags);
    shortp_interrupt(shortp_irq, NULL, NULL);
}
```

如果没有输出要被激活, 定时器函数简单地返回. 这避免了定时器重新提交自己, 当事情在被关闭时. 接着, 在获得了锁之后, 我们查询端口的状态; 如果它声称忙, 它完全还没有时间来中断我们, 因此我们复位定时器并且返回. 打印机能够, 有时, 花很长时间来使自己准备; 考虑一下缺纸的打印机, 而每个人在一个长周末都不在. 在这种情况下, 只有耐心等待直到事情改变.

但是, 如果打印机声称准备好了, 我们一定丢失了它的中断. 这个情况下, 我们简单地手动调用我们的中断处理来使输出处理再动起来.

shortpirnt 驱动不支持从端口读数据; 相反, 它象 shortint 并且返回中断时间信息. 但是一个中断驱动的读方法的实现可能非常类似我们已经见到的. 从设备来的数据可能被读入驱动缓存; 它可能被拷贝到用户空间只在缓存中已经累积了相当数量的数据, 完整的读请求已被满足, 或者某种超时发生.

10.6. 快速参考

本章中介绍了这些关于中断管理的符号:

#include <linux/interrupt.h>
int request_irq(unsigned int irq, irqreturn_t (*handler)( ), unsigned long flags, const char *dev_name, void *dev_id);
void free_irq(unsigned int irq, void *dev_id);

调用这个注册和注销一个中断处理.
#include <linux/irq.h.h>
int can_request_irq(unsigned int irq, unsigned long flags);

这个函数, 在 i386 和 x86_64 体系上有, 返回一个非零值如果一个分配给定中断线的企图成功.
#include <asm/signal.h>
SA_INTERRUPT
SA_SHIRQ
SA_SAMPLE_RANDOM

给 request_irq 的标志. SA_INTERRUPT 请求安装一个快速处理者(相反是一个慢速的).
SA_SHIRQ 安装一个共享的处理者, 并且第 3 个 flag 声称中断时戳可用来产生系统熵.
/proc/interrupts
/proc/stat

报告硬件中断和安装的处理者的文件系统节点.
unsigned long probe_irq_on(void);
int probe_irq_off(unsigned long);

驱动使用的函数, 当它不得不探测来决定哪个中断线被设备在使用. probe_irq_on 的结果必须传回给 probe_irq_off 在中断产生之后. probe_irq_off 的返回值是被探测的中断号.
IRQ_NONE
IRQ_HANDLED
IRQ_RETVAL(int x)

从一个中断处理返回的可能值, 指示是否一个来自设备的真正的中断出现了.
void disable_irq(int irq);
void disable_irq_nosync(int irq);

```
void enable_irq(int irq);
```

驱动可以使能和禁止中断报告. 如果硬件试图在中断禁止时产生一个中断, 这个中断永远丢失了. 一个使用一个共享处理者的驱动必须不使用这个函数.

```
void local_irq_save(unsigned long flags);
void local_irq_restore(unsigned long flags);
```

使用 local_irq_save 来禁止本地处理器的中断并且记住它们之前的状态. flags 可以被传递给 local_irq_restore 来恢复之前的中断状态.

```
void local_irq_disable(void);
void local_irq_enable(void);
```

在当前处理器熵无条件禁止和使能中断的函数.

上一页 上一级 下一页
10.5. 中断驱动 I/O 起始页 第 11 章 内核中的数据类型

第 11 章 内核中的数据类型

上一页 下一页

第 11 章 内核中的数据类型

目录

在我们进入更高级主题之前, 我们需要停下来快速关注一下可移植性问题. 现代版本的 Linux 内核是高度可移植的, 它正运行在很多不同体系上. 由于 Linux 内核的多平台特性, 打算做认真使用的驱动应当也是可移植的.

但是内核代码的一个核心问题是不但能够存取已知长度的数据项(例如, 文件系统数据结构或者设备单板上的寄存器), 而且可以使用不同处理器的能力(32-位 和 64-位 体系, 并且也可能是 16 位).

内核开发者在移植 x86 代码到新体系时遇到的几个问题与不正确的数据类型相关. 坚持严格的数据类型和使用 -Wall -Wstrict-prototypes 进行编译可能避免大部分的 bug.

内核数据使用的数据类型分为 3 个主要类型: 标准 C 类型例如 int, 明确大小的类型例如 u32, 以及用作特定内核对象的类型, 例如 pid_t. 我们将看到这 3 个类型种类应当什么时候以及应当如何使用. 本章的最后的节谈论一些其他的典型问题, 你在移植 x86 的驱动到其他平台时可能遇到的问题, 并且介绍近期内核头文件输出的链表的常用支持.

如果你遵照我们提供的指引, 你的驱动应当编译和运行在你无法测试的平台上.

11.1. 标准 C 类型的使用

尽管大部分程序员习惯自由使用标准类型, 如 int 和 long, 编写设备驱动需要一些小心来避免类型冲突和模糊的 bug.

这个问题是你不能使用标准类型, 当你需要"一个 2-字节 填充者"或者"一个东西来代表一个4-字节字串", 因为正常的 C 数据类型在所有体系上不是相同大小. 为展示各种 C 类型的数据大小, datasize 程序已包含在例子文件 misc-progs 目录中, 由 O' Reilly's FTP 站点提供. 这是一个程序的样例运行, 在一个 i386 系统上(显示的最后 4 个类型在下一章介绍):

```
morgana% misc-progs/datasize
arch Size: char short int long ptr long-long u8 u16 u32 u64
i686     1   2    4  4   4  8       1  2   4   8
```

这个程序可以用来显示长整型和指针在 64-位 平台上的不同大小, 如同在不同 Linux 计算机上运行程序所演示的:

```
arch  Size:  char  short  int  long  ptr long-long  u8 u16 u32 u64
i386       1    2     4   4    4  8       1  2   4   8
alpha      1    2     4   8    8  8       1  2   4   8
armv4l     1    2     4   4    4  8       1  2   4   8
ia64       1    2     4   8    8  8       1  2   4   8
m68k       1    2     4   4    4  8       1  2   4   8
mips       1    2     4   4    4  8       1  2   4   8
ppc        1    2     4   4    4  8       1  2   4   8
sparc      1    2     4   4    4  8       1  2   4   8
sparc64    1    2     4   4    4  8       1  2   4   8
x86_64     1    2     4   8    8  8       1  2   4   8
```

注意有趣的是 SPARC 64 体系在一个 32-位 用户空间运行, 因此那里指针是 32 位宽, 尽管它们在内核空间是 64 位宽. 这可用加载 kdatasize 模块(在例子文件的 misc-modules 目录里)来验证. 这个模块在加载时使用 printk 来报告大小信息, 并且返回一个错误(因此没有必要卸载它):

```
kernel: arch Size: char short int long ptr long-long u8 u16 u32 u64
kernel: sparc64   1   2    4  8   8  8       1  2   4   8
```

尽管在混合不同数据类型时你必须小心, 有时有很好的理由这样做. 一种情况是因为内存存取, 与内核相关时是特殊的. 概念上, 尽管地址是指针, 内存管理常常使用一个无符号的整数类型更好地完成; 内核对待物理内存如同一个大数组, 并且内存地址只是一个数组索引. 进一步地, 一个指针容易解引用; 当直接处理内存存取时, 你几乎从不想以这种方式解引用. 使用一个整数类型避免了这种解引用, 因此避免了 bug. 因此, 内核中通常的内存地址常常是 unsigned long, 利用了指针和长整型一直是相同大小的这个事实, 至少在 Linux 目前支持的所有平台上.

因为其所值的原因, C99 标准定义了 intptr_t 和 uintptr_t 类型给一个可以持有一个指针值的整型变量. 但是, 这些类型几乎没在 2.6 内核中使用.

上一页 下一页
10.6. 快速参考 起始页 11.2. 安排一个明确大小给数据项

11.2. 安排一个明确大小给数据项

有时内核代码需要一个特定大小的数据项, 也许要匹配预定义的二进制结构,[39] 来和用户空间通讯, 或者来用插入"填充"字段来对齐结构中的数据(但是关于对齐问题的信息参考 "数据对齐" 一节).

内核提供了下列数据类型来使用, 无论你什么时候需要知道你的数据的大小. 所有的数据声明在 <asm/types.h>, 它又被 <linux/types.h> 包含.

```
u8; /* unsigned byte (8 bits) */
u16; /* unsigned word (16 bits) */
u32; /* unsigned 32-bit value */
u64; /* unsigned 64-bit value */
```

存在对应的有符号类型, 但是很少需要; 如果你需要它们, 只要在名子里用 s 代替 u.

如果一个用户空间程序需要使用这些类型, 可用使用一个双下划线前缀在名子上: __u8 和其它独立于 __KERNEL__ 定义的类型. 例如, 如果, 一个驱动需要与用户空间中运行的程序交换二进制结构, 通过 ioctl, 头文件应当在结构中声明 32-位 成员为 __u32.

重要的是记住这些类型是 Linux 特定的, 并且使用它们妨碍了移植软件到其他的 Unix 口味上. 使用近期编译器的系统支持 C99-标准 类型, 例如 uint8_t 和 uint32_t; 如果考虑到移植性, 使用这些类型比 Linux-特定的变体要好.

你可能也注意到有时内核使用传统的类型, 例如 unsigned int, 给那些维数与体系无关的项. 这是为后向兼容而做的. 当 u32 和它的类似物在版本 1.1.67 引入时, 开发者不能改变存在的数据结构为新的类型, 因为编译器发出一个警告当在结构成员和安排给它的值之间有一个类型不匹配时.. Linus 不希望他写给自己使用的操作系统称为多平台的; 结果是, 老的结构有时被松散的键入.

事实上, 编译器指示类型不一致, 甚至在 2 个类型只是同一个对象的不同名子, 例如在 PC 上 unsigned long 和 u32.

---

[39] 这发生在当读取分区表时, 当执行一个二进制文件时, 或者当解码一个网络报文时.

上一页 上一级 下一页
第 11 章 内核中的数据类型 起始页 11.3. 接口特定的类型

11.3. 接口特定的类型

内核中一些通常使用的数据类型有它们自己的 typedef 语句, 因此阻止了任何移植性问题. 例如, 一个进程标识符 (pid) 常常是 pid_t 而不是 int. 使用 pid_t 屏蔽了任何在实际数据类型上的不同. 我们使用接口特定的表达式来指一个类型, 由一个库定义的, 以便于提供一个接口给一个特定的数据结构.

注意, 在近期, 已经相对少定义新的接口特定类型. 使用 typedef 语句已经有许多内核开发者不喜欢, 它们宁愿看到代码中直接使用的真实类型信息, 不是藏在一个用户定义的类型后面. 很多老的接口特定的类型在内核中保留, 但是, 并且它们不会很快消失.

甚至当没有定义接口特定的类型, 以和内核其他部分保持一致的方式使用正确的数据类型是一直重要的. 一个嘀哒计数, 例如, 一直是 unsigned long, 独立于它实际的大小, 因此 unsigned long 类型应当在使用 jiffy 时一直使用. 本节我们集中于 _t 类型的使用.

很多 _t 类型在 <linux/types.h> 中定义, 但是列出来是很少有用. 当你需要一个特定类型, 你会在你需要调用的函数的原型中发现它, 或者在你使用的数据结构中.

无论何时你的驱动使用需要这样"定制"类型的函数并且你不遵照惯例, 编译器发出一个警告; 如果你使用 -Wall 编译器标志并且小心去除所有的警告, 你能有信心你的代码是可移植的.

_t 数据项的主要问题是当你需要打印它们时, 常常不容易选择正确的 printk 或 printf 格式, 你在一个体系上出现的警告会在另一个上重新出现. 例如, 你如何打印一个 size_t, 它在一些平台上是 unsigned long 而在其他某个上面是 unsigned int?

无论何时你需要打印某个接口特定的数据, 最好的方法是转换它的值为最大的可能类型(常常是 long 或者 unsigned long) 并且接着打印它通过对应的格式. 这种调整不会产生错误或者警告, 因为格式匹配类型, 并且你不会丢失数据位, 因为这个转换或者是一个空操作或者是数据项向更大数据类型的扩展.

实际上, 我们在谈论的数据项不会常常要打印的, 因此这个问题只适用于调试信息. 常常, 代码只需要存储和比较接口特定的类型, 加上传递它们作为给库或者内核函数的参数.

尽管 _t 类型是大部分情况的正确解决方法, 有时正确的类型不存取. 这发生在某些还未被清理的老接口.

我们在内核头文件中发现的一个模糊之处是用在 I/O 函数的数据类型, 它松散地定义(看第 9 章"平

台相关性" 一节). 松散的类型在那里主要是因为历史原因, 但是在写代码时它可能产生问题. 例如, 交换给函数如 outb 的参数可能会有麻烦; 如果有一个 port_t 类型, 编译器会发现这个类型.

上一页 | 上一级 | 下一页
11.2. 安排一个明确大小给数据项 | 起始页 | 11.4. 其他移植性问题

11.4. 其他移植性问题

除了数据类型, 当编写一个驱动时有几个其他的软件问题要记住, 如果你想在 Linux 平台间可移植.

一个通常的规则是怀疑显式的常量值. 常常通过使用预处理宏, 代码已被参数化. 这一节列出了最重要的可移植性问题. 无论何时你遇到已被参数化的值, 你可以在头文件中以及在随官方内核发布的设备驱动中找到提示.

11.4.1. 时间间隔

当涉及时间间隔, 不要假定每秒有 1000 个嘀哒. 尽管当前对 i386 体系是真实的, 不是每个 Linux 平台都以这个速度运行. 对于 x86 如果你使用 HZ 值(如同某些人做的那样), 这个假设可能是错的, 并且没人知道将来内核会发生什么. 无论何时你使用嘀哒来计算时间间隔, 使用 HZ (每秒的定时器中断数) 来标定你的时间. 例如, 检查一个半秒的超时, 用 HZ/2 和逝去时间比较. 更普遍地, msec 毫秒对应地嘀哒数一直是 msec*HZ/1000.

11.4.2. 页大小

当使用内存时, 记住一个内存页是 PAGE_SIZE 字节, 不是 4KB. 假定页大小是 4KB 并且硬编码这个值是一个 PC 程序员常见的错误, 相反, 被支持的平台显示页大小从 4 KB 到 64 KB, 并且有时它们在相同平台上的不同的实现上不同. 相关的宏定义是 PAGE_SIZE 和 PAGE_SHIT. 后者包含将一个地址移位来获得它的页号的位数. 对于 4KB 或者更大的页这个数当前是 12 或者更大. 宏在 <asm/page.h> 中定义; 用户空间程序可以使用 getpagesize 库函数, 如果它们需要这个信息.

让我们看一下非一般的情况. 如果一个驱动需要 16 KB 来暂存数据, 它不应当指定一个 2 的指数 给 get_free_pages. 你需要一个可移植解决方法. 这样的解决方法, 幸运的是, 已经由内核开发者写好并且称为 get_order:

```
#include <asm/page.h>
int order = get_order(16*1024);
buf = get_free_pages(GFP_KERNEL, order);
```

记住, get_order 的参数必须是 2 的幂.

11.4.3. 字节序

小心不要假设字节序. PC 存储多字节值是低字节为先(小端为先, 因此是小端), 一些高级的平台以另一种方式(大端)工作. 任何可能的时候, 你的代码应当这样来编写, 它不在乎它操作的数据的字节序. 但是, 有时候一个驱动需要使用单个字节建立一个整型数或者相反, 或者它必须与一个要求一个特定顺序的设备通讯.

包含文件 <asm/byteorder.h> 定义了或者 __BIG_ENDIAN 或者 __LITTLE_ENDIAN, 依赖处理器的字节序. 当处理字节序问题时, 你可能编码一堆 #ifdef __LITTTLE_ENDIAN 条件语句, 但是有一个更好的方法. Linux 内核定义了一套宏定义来处理之间的转换, 在处理器字节序和你需要以特定字节序存储和加载的数据之间. 例如:

```
u32 cpu_to_le32 (u32);
u32 le32_to_cpu (u32);
```

这 2 个宏定义转换一个值, 从无论 CPU 使用的什么到一个无符号的, 小端, 32 位数, 并且转换回. 它们不管你的 CPU 是小端还是大端, 不管它是不是 32-位 处理器. 在没有事情要做的情况下它们原样返回它们的参数. 使用这些宏定义易于编写可移植的代码, 而不必使用大量的条件编译建造.

有很多类似的函数; 你可以在 <linux/byteorder/big_endian.h> 和 <linux/byteorder/little_endian.h> 中见到完整列表. 一会儿之后, 这个模式不难遵循. be64_to_cpu 转换一个无符号的, 大端, 64-位 值到一个内部 CPU 表示. le16_to_cpus, 相反, 处理有符号的, 小端, 16 位数. 当处理指针时, 你也会使用如 cpu_to_le32p, 它使用指向一个值的指针来转换, 而不是这个值自身. 剩下的看包含文件.

11.4.4. 数据对齐

编写可移植代码而值得考虑的最后一个问题是如何存取不对齐的数据 -- 例如, 如何读取一个存储于一个不是 4 字节倍数的地址的4字节值. i386 用户常常存取不对齐数据项, 但是不是所有的体系允许这个. 很多现代的体系产生一个异常, 每次程序试图不对齐数据传送时; 数据传输由异常处理来处理, 带来很大的性能牺牲. 如果你需要存取不对齐的数据, 你应当使用下列宏:

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

这些宏是无类型的, 并且用在每个数据项, 不管它是 1 个, 2 个, 4 个, 或者 8 个字节长. 它们在任何内核版本中定义.

关于对齐的另一个问题是跨平台的数据结构移植性. 同样的数据结构(在 C-语言 源文件中定义)可能在不同的平台上不同地编译. 编译器根据各个平台不同的惯例来安排结构成员对齐.

为了编写可以跨体系移动的数据使用的数据结构, 你应当一直强制自然的数据项对齐, 加上对一个特定对齐方式的标准化. 自然对齐意味着存储数据项在是它的大小的整数倍的地址上(例如, 8-byte 项在 8 的整数倍的地址上). 为强制自然对齐在阻止编译器以不希望的方式安排成员量的时候, 你应

当使用填充者成员来避免在数据结构中留下空洞.

为展示编译器如何强制对齐, dataalign 程序在源码的 misc-progs 目录中发布, 并且一个对等的 kdataalign 模块是 misc-modules 的一部分. 这是程序在几个平台上的输出以及模块在 SPARC64 的输出:

```
arch Align: char short int long ptr long-long u8 u16 u32 u64
i386      1   2    4   4   4   4       1  2   4   4
i686      1   2    4   4   4   4       1  2   4   4
alpha     1   2    4   8   8   8       1  2   4   8
armv4l    1   2    4   4   4   4       1  2   4   4
ia64      1   2    4   8   8   8       1  2   4   8
mips      1   2    4   4   4   8       1  2   4   8
ppc       1   2    4   4   4   8       1  2   4   8
sparc     1   2    4   4   4   8       1  2   4   8
sparc64   1   2    4   4   4   8       1  2   4   8
x86_64    1   2    4   8   8   8       1  2   4   8

kernel: arch Align: char short int long ptr long-long u8 u16 u32 u64
kernel: sparc64   1   2    4   8   8   8       1  2   4   8
```

有趣的是注意不是所有的平台对齐 64-位值在 64-位边界上, 因此你需要填充者成员来强制对齐和保证可移植性.

最后, 要知道编译器可能自己悄悄地插入填充到结构中来保证每个成员是对齐的, 为了目标处理器的良好性能. 如果你定义一个结构打算来匹配一个设备期望的结构, 这个自动的填充可能妨碍你的企图. 解决这个问题的方法是告诉编译器这个结构必须是"紧凑的", 不能增加填充者. 例如, 内核头文件 <linux/edd.h> 定义几个与 x86 BIOS 接口的数据结构, 并且它包含下列的定义:

```
struct
{
      u16 id;
      u64 lun;
      u16 reserved1;
      u32 reserved2;
}
__attribute__ ((packed)) scsi;
```

如果没有 __attribute__ ((packed)), lun 成员可能被在前面添加 2 个填充者字节或者 6 个, 如果我们在 64-位平台上编译这个结构.

11.4.5. 指针和错误值

很多内部内核函数返回一个指针值给调用者. 许多这些函数也可能失败. 大部分情况, 失败由返回一个 NULL 指针值来指示. 这个技术是能用的, 但是它不能通知问题的确切特性. 一些接口确实需要返回一个实际的错误码以便于调用者能够基于实际上什么出错来作出正确的判断.

许多内核接口通过在指针值中对错误值编码来返回这个信息. 这样的信息必须小心使用, 因为它们的返回值不能简单地与 NULL 比较. 为帮助创建和使用这类接口, 一小部分函数已可用(在 <linux/err.h>).

一个返回指针类型的函数可以返回一个错误值, 使用:

```
void *ERR_PTR(long error);
```

这里, error 是常见的负值错误码. 调用者可用使用 IS_ERR 来测试是否一个返回的指针是不是一个错误码:

```
long IS_ERR(const void *ptr);
```

如果你需要实际的错误码, 它可能被抽取到, 使用:

```
long PTR_ERR(const void *ptr);
```

你应当只对 IS_ERR 返回一个真值的值使用 PTR_ERR; 任何其他的值是一个有效的指针.

上一页 11.3. 接口特定的类型 | 上一级 起始页 | 下一页 11.5. 链表

11.5. 链表

操作系统内核, 如同其他程序, 常常需要维护数据结构的列表. 有时, Linux 内核已经同时有几个列表实现. 为减少复制代码的数量, 内核开发者已经创建了一个标准环形的, 双链表; 鼓励需要操作列表的人使用这个设施.

当使用链表接口时, 你应当一直记住列表函数不做加锁. 如果你的驱动可能试图对同一个列表并发操作, 你有责任实现一个加锁方案. 可选项(破坏的列表结构, 数据丢失, 内核崩溃) 肯定是难以诊断的.

为使用列表机制, 你的驱动必须包含文件 <linux/list.h>. 这个文件定义了一个简单的类型 list_head 结构:

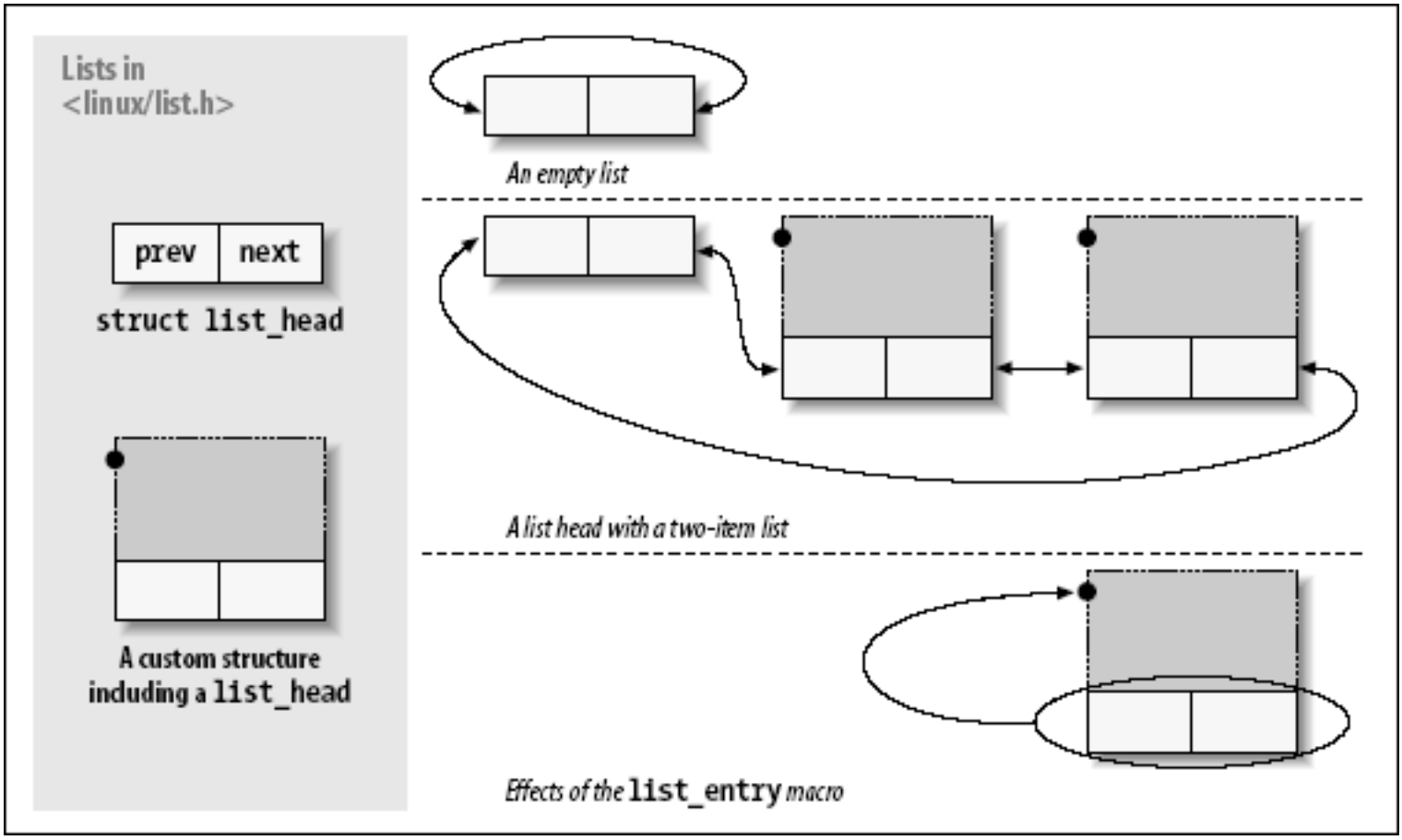
```
struct list_head { struct list_head *next, *prev; };
```

真实代码中使用的链表几乎是不变地由几个结构类型组成, 每一个描述一个链表中的入口项. 为在你的代码中使用 Linux 列表, 你只需要嵌入一个 list_head 在构成这个链表的结构里面. 假设, 如果你的驱动维护一个列表, 它的声明可能看起来象这样:

```
struct todo_struct
{
    struct list_head list;
    int priority; /* driver specific */
    /* ... add other driver-specific fields */
};
```

列表的头常常是一个独立的 list_head 结构. 图链表头数据结构显示了这个简单的 struct list_head 是如何用来维护一个数据结构的列表的.

图 11.1. 链表头数据结构



链表头必须在使用前用 INIT_LIST_HEAD 宏来初始化. 一个"要做的事情"的链表头可能声明并且初始化用:

```
struct list_head todo_list;
INIT_LIST_HEAD(&todo_list);
<para>可选地, 链表可在编译时初始化:</para>
LIST_HEAD(todo_list);
```

几个使用链表的函数定义在 <linux/list.h>:

```
list_add(struct list_head *new, struct list_head *head);
```

在紧接着链表 head 后面增加新入口项 -- 正常地在链表的开头. 因此, 它可用来构建堆栈. 但是, 注意, head 不需要是链表名义上的头; 如果你传递一个 list_head 结构, 它在链表某处的中间, 新的项紧靠在它后面. 因为 Linux 链表是环形的, 链表的头通常和任何其他的项没有区别.

```
list_add_tail(struct list_head *new, struct list_head *head);
```

刚好在给定链表头前面增加一个新入口项 -- 在链表的尾部, 换句话说. list_add_tail 能够, 因此, 用来构建先入先出队列.

```
list_del(struct list_head *entry);
list_del_init(struct list_head *entry);
```

给定的项从队列中去除. 如果入口项可能注册在另外的链表中, 你应当使用 list_del_init, 它重新初始化这个链表指针.

```
list_move(struct list_head *entry, struct list_head *head);
list_move_tail(struct list_head *entry, struct list_head *head);
```

给定的入口项从它当前的链表里去除并且增加到 head 的开始. 为安放入口项在新链表的末尾, 使用 list_move_tail 代替.

```
list_empty(struct list_head *head);
```

如果给定链表是空, 返回一个非零值.

```
list_splice(struct list_head *list, struct list_head *head);
```

将 list 紧接在 head 之后来连接 2 个链表.

list_head 结构对于实现一个相似结构的链表是好的, 但是调用程序常常感兴趣更大的结构, 它组成链表作为一个整体. 一个宏定义, list_entry, 映射一个 list_head 结构指针到一个指向包含它的结构的指针. 它如下被调用:

```
list_entry(struct list_head *ptr, type_of_struct, field_name);
```

这里 ptr 是一个指向使用的 struct list_head 的指针, type_of_struct 是包含 ptr 的结构的类型, field_name 是结构中列表成员的名子. 在我们之前的 todo_struct 结构中, 链表成员称为简单列表. 因此, 我们应当转变一个列表入口项为它的包含结构, 使用这样一行:

```
struct todo_struct *todo_ptr = list_entry(listptr, struct todo_struct, list);
```

list_entry 宏定义使用了一些习惯的东西但是不难用.

链表的遍历是容易的: 只要跟随 prev 和 next 指针. 作为一个例子, 假设我们想保持 todo_struct 项的列表已降序的优先级顺序排列. 一个函数来添加新项应当看来如此:

```
void todo_add_entry(struct todo_struct *new)
{
    struct list_head *ptr;
    struct todo_struct *entry;

    for (ptr = todo_list.next; ptr != &todo_list; ptr = ptr->next)
    {
        entry = list_entry(ptr, struct todo_struct, list);
        if (entry->priority < new->priority) {
```

```
                list_add_tail(&new->list, ptr);
                return;
            }
        }
        list_add_tail(&new->list, &todo_struct)

}
```

但是, 作为一个通用的规则, 最好使用一个预定义的宏来创建循环, 它遍历链表. 前一个循环, 例如, 可这样编码:

```
void todo_add_entry(struct todo_struct *new)
{
        struct list_head *ptr;
        struct todo_struct *entry;

        list_for_each(ptr, &todo_list)
        {
            entry = list_entry(ptr, struct todo_struct, list);
            if (entry->priority < new->priority) {

                list_add_tail(&new->list, ptr);
                return;
            }
        }
        list_add_tail(&new->list, &todo_struct)

}
```

使用提供的宏帮助避免简单的编程错误; 宏的开发者也已做了些努力来保证它们进行正常. 存在几个变体:

list_for_each(struct list_head *cursor, struct list_head *list)

这个宏创建一个 for 循环, 执行一次, cursor 指向链表中的每个连续的入口项. 小心改变列表在遍历它时.

list_for_each_prev(struct list_head *cursor, struct list_head *list)

这个版本后向遍历链表.

list_for_each_safe(struct list_head *cursor, struct list_head *next, struct list_head *list)

如果你的循环可能删除列表中的项, 使用这个版本. 它简单的存储列表 next 中下一个项, 在

循环的开始, 因此如果 cursor 指向的入口项被删除, 它不会被搞乱.
list_for_each_entry(type *cursor, struct list_head *list, member)
list_for_each_entry_safe(type *cursor, type *next, struct list_head *list, member)

这些宏定义减轻了对一个包含给定结构类型的列表的处理. 这里, cursor 是一个指向包含数据类型的指针, member 是包含结构中 list_head 结构的名子. 有了这些宏, 没有必要安放 list_entry 调用在循环里了.

如果你查看 <linux/list.h> 里面, 你看到有一些附加的声明. hlist 类型是一个有一个单独的, 单指针列表头类型的双向链表; 它常用作创建哈希表和类型结构. 还有宏用来遍历 2 种列表类型, 打算作使用 读取-拷贝-更新 机制(在第 5 章的"读取-拷贝-更新"一节中描述). 这些原语在设备驱动中不可能有用; 看头文件如果你愿意知道更多信息关于它们是如何工作的.

11.6. 快速参考

下列符号在本章中介绍了:

```
#include <linux/types.h>
typedef u8;
typedef u16;
typedef u32;
typedef u64;
```

保证是 8-位, 16-位, 32-位 和64-位 无符号整型值的类型. 对等的有符号类型也存在. 在用户空间, 你可用 __u8, __u16, 等等来引用这些类型.

```
#include <asm/page.h>
PAGE_SIZE
PAGE_SHIFT
```

给当前体系定义每页的字节数, 以及页偏移的位数(对于 4 KB 页是 12, 8 KB 是 13)的符号.

```
#include <asm/byteorder.h>
__LITTLE_ENDIAN
__BIG_ENDIAN
```

这 2 个符号只有一个定义, 依赖体系.

```
#include <asm/byteorder.h>
u32 __cpu_to_le32 (u32);
u32 __le32_to_cpu (u32);
```

在已知字节序和处理器字节序之间转换的函数. 有超过 60 个这样的函数: 在 include/linux/byteorder/ 中的各种文件有完整的列表和它们以何种方式定义.

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

一些体系需要使用这些宏保护不对齐的数据存取. 这些宏定义扩展成通常的指针解引用, 为那些允许你存取不对齐数据的体系.

```
#include <linux/err.h>
void *ERR_PTR(long error);
long PTR_ERR(const void *ptr);
```

long IS_ERR(const void *ptr);

允许错误码由返回指针值的函数返回.

```
#include <linux/list.h>
list_add(struct list_head *new, struct list_head *head);
list_add_tail(struct list_head *new, struct list_head *head);
list_del(struct list_head *entry);
list_del_init(struct list_head *entry);
list_empty(struct list_head *head);
list_entry(entry, type, member);
list_move(struct list_head *entry, struct list_head *head);
list_move_tail(struct list_head *entry, struct list_head *head);
list_splice(struct list_head *list, struct list_head *head);
```

操作环形, 双向链表的函数.

```
list_for_each(struct list_head *cursor, struct list_head *list)
list_for_each_prev(struct list_head *cursor, struct list_head *list)
list_for_each_safe(struct list_head *cursor, struct list_head *next, struct list_head *list)
list_for_each_entry(type *cursor, struct list_head *list, member)
list_for_each_entry_safe(type *cursor, type *next struct list_head *list, member)
```

方便的宏定义, 用在遍历链表上.

上一页 11.5. 链表

上一级 起始页