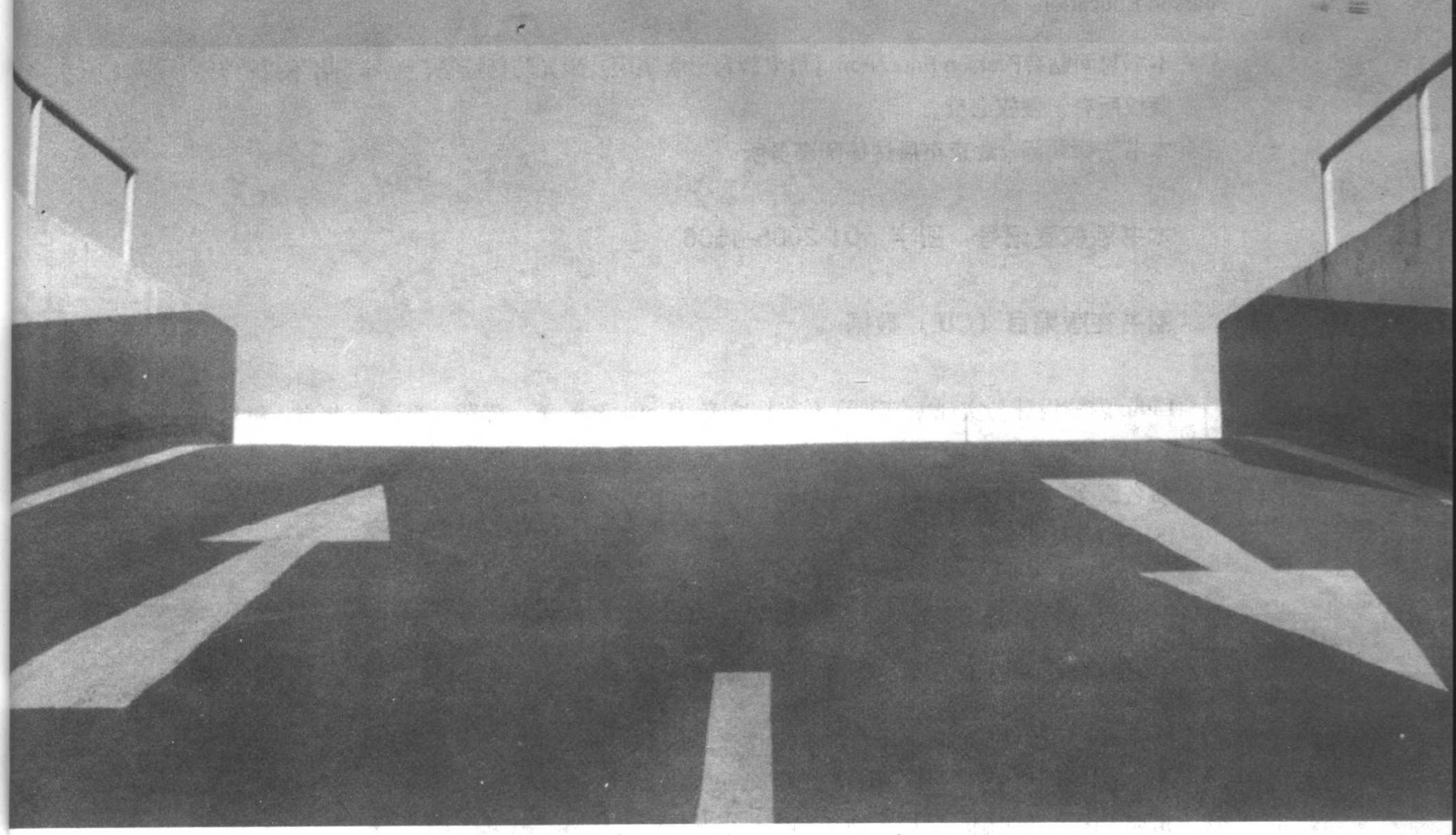
第2版

Linux内核设计与实现

Linux Kernel Development
Second Edition



（美） Robert Love 著

陈莉君 康 华 张 波 译

机械工业出版社
China Machine Press

本书基于Linux 2.6内核系列详细介绍Linux内核系统，覆盖了从核心内核系统的应用到内核设计与实现等各方面内容。主要内容包括：进程管理、系统调用、中断和中断处理程序、内核同步、时间管理、内存管理、地址空间、调试技术等。本书理论联系实践，既介绍理论也讨论具体应用，能够带领读者快速走进Linux内核世界，真正开发内核代码。

本书适合作为高等院校操作系统课程的教材或参考书，也可供相关技术人员参考。

本书版权登记号：图字：01-2005-4506

图书在版编目（CIP）数据

Linux内核设计与实现（第2版）/（美）拉芙（Love, R.）著；陈莉君等译. -北京：机械工业出版社，2006.1

书名原文：Linux Kernel Development, Second Edition

ISBN 7-111-17865-3

Ⅰ. L…　Ⅱ. ①拉…　②陈…　Ⅲ. Linux操作系统-程序设计　Ⅳ. TP316.89

中国版本图书馆CIP数据核字（2005）第133821号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：隋　曦　　吴　怡

北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行

2006年1月第2版第1次印刷

787mm×1020mm 1/16 · 19印张

印数：0 001-4 000 册

定价：38.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

译 者 序

不知不觉涉足Linux内核已经几个年头了，与其他有志（兴趣）于此的朋友一样，我们也经历了学习——实用——追踪——再学习的过程。也就是说，我们也是从漫无边际到茫然无措，再到初窥门径，转而觉得心有戚戚焉这一路走下来的。其中甘苦，犹然在心。

Linux最为人称道的莫过于它的自由精神，所有源代码唾手可得。侯捷先生云："源码在前，了无秘密"。是的，但是我们在面对它的时候，为什么却总是因为这种规模和层面所造就的陡峭学习曲线陷入困顿呢？很多朋友就此倒下，纵然Linux世界繁花似锦，纵然内核天空无边广阔。但是，眼前的迷雾重重，心中的阴霾又怎能被阳光驱散呢？纵有雄心壮志，拔剑四顾心茫然，脚下路在何方？

Linux内核入门是不容易，它之所以难学，在于庞大的规模和复杂的层面。规模一大，就不易现出本来面目，浑然一体，自然不容易找到着手之处；层面一多，就会让人眼花缭乱，盘根错节，怎能让人提纲挈领？

"如果有这样一本书，既能提纲挈领，为我理顺思绪，指引方向，同时又能照顾小节，阐述细微，帮助我们更好更快地理解STL源码，那该有多好。"孟岩先生如此说，虽然针对的是C++，但道出的也是研习源码的人们共同的心声。然而，Linux源码研究的方法却不大相同。这还是由于规模和层面决定的。比如说，在语言学习中，我们可以采取小步快跑的方法，通过一个个小程序和小尝试，就可以取得渐进的成果，就能从新技术中有所收获；而掌握Linux呢？如果没有对整体的把握，即使你对某个局部的算法、技术或是代码再熟悉，也无法将其融入实用。其实，像内核这样的大规模的软件，正是编程技术施展身手的舞台（当然，目前的内核虽然包含了一些面向对象思想，但还不能让C++一展身手）。

那么，我们能不能做出点什么，让Linux的内核学习过程更符合程序员的习惯呢？

Robert Love回答了这个问题。Robert Love是一个狂热的内核爱好者，所以他的想法自然贴近程序员。是的，我们注定要在对所有核心的子系统有了全面认识之后，才能开始自己的实践，但却完全可以舍弃细枝末节，将行李压到最小，自然可以轻装快走，迅速进入动手阶段。

因此，本书相对于Daniel P. Bovet和Marco Cesati的内核巨著《Understanding the Linux Kernel》，少了五分细节；相对于实践经典《Linux Device Drivers》，又多了五分说理。可以说，本书填补了Linux内核理论和实践之间的鸿沟，真可谓"一桥飞架南北，天堑变通途"。

就我们的经验，内核初学者（不是编程初学者）可以从这本书着手，对内核各个核心子系统有个整体把握，包括它们提供什么样的服务，为什么要提供这样的服务，又是怎样实现的。而且，本书还包含了Linux内核开发者在开发时需要用到的很多信息，包括调试技术、编程风格、注意事项等等。在消化这本书的基础上，如果你侧重于了解内核，可以进一步研究《Understanding the Linux Kernel》和源代码本身；如果你侧重于实际编程，可以研读《Linux Device Drivers》，直接开始动手工作；如果你想有一个轻松的内核学习和实践环节，请访问我们的网站www.kerneltravel.net。

依然记得译第1版时的喜悦，第2版的到来自然就爱不释手了。同事贺炎为两版之间差异所费的心思全部体现在了字里行间，请读者欣赏第2版丰富的内容吧。

序　言

随着Linux内核和Linux应用程序越来越成熟，越来越多的系统软件工程师涉足Linux开发和维护领域。他们中有些人纯粹是出于个人爱好，有些人是为Linux公司工作，有些是为硬件厂商做开发，还有一些是为内部项目工作的。

但是所有人都必须直面一个问题：内核的学习曲线变得越来越长，也越来越陡峭。系统规模不断扩大，复杂程度不断提高。长此以往，虽然现在的内核开发者对内核的掌握越发炉火纯青，但却会造成新手无法跟上内核发展步伐，出现青黄不接的断层。

我认为这种新老鸿沟已经成为内核质量的一个隐患，而且问题将继续恶化。所以那些真正关心内核的人已经开始致力于扩大内核开发群体。

解决上述问题的一个方法是尽量保证代码简洁：接口定义合理，代码风格一致，“一次做一件事，做到完美”等等。这也就是Linus Torvalds倡导的解决办法。

我提倡的解决办法是对代码慷慨地加上注释：能够让读者立刻了解代码开发者意图的文字（识别意图和实现之间差异的工作称为调试。如果意图不明确显然调试就难以进行）。

可是，即使有注解，也没办法清楚地展现内核的各个主要子系统的全景，说明它们到底要做什么。那么，这些开发者又该从何下手呢？

由文字材料来说明这些在起步阶段就该理解的材料，其实是最合适的。

Robert Love的贡献就在于此，有经验的开发者可以通过本书全面了解内核子系统提供的服务，同时还可以了解这些服务是怎么实现的。对不少人来说，这些知识就已经足够了：那些好奇的人，那些应用程序开发者，那些想对内核的设计品头论足一番的人，都有足够的谈资了。

但是本书同样可以为那些有抱负的内核开发者更上一层楼提供契机，可以帮他们更改内核代码以达到预定的目标。我建议有抱负的开发者能够亲身实践：理解内核某部分的捷径就是对它做些修改，这样能为开发者揭示仅仅通过看内核代码无法看到的深层机理。

严谨认真的内核开发者都应该加入开发邮件列表，不断和其他开发者交流。这是内核开发者相互切磋和并肩前进的最好方法。Robert在本书中对内核生活中至关重要的文化和技巧都做了精彩介绍。

请学习和欣赏Robert的书吧。想必你也希望能精益求精，继续探索，成为内核开发社区中的一员，那么首先你要清楚的是：社区欢迎你。我们评价和衡量一个人是根据他所做的贡献，当你投身于Linux时，你要明白：虽然你仅仅贡献了一小份力，但马上就会有数千万或上亿人受益。这是我们的欢乐之源，也是我们的责任之本。

Andrew Morton
Open Source Development Labs

前　言

在我刚开始有把自己的内核开发经验集结成册，撰写一本书的念头时，我其实也觉得有点头绪繁多，不知道该从何下手。我实在不想落入传统内核书籍的窠臼，照猫画虎地再写这么一本。不错，前人著述备矣，但我终归是要写出点儿与众不同的东西来，我的书该如何定位，说实话，这确实让人颇费思量。

后来，灵感终于浮现出来，我意识到自己可以从一个全新的视角看待这个主题。开发内核是我的工作，开发内核也是我的嗜好，内核就是我的挚爱。这些年来，我不断搜集与内核有关的奇闻轶事，不断积攒关键的开发诀窍，依靠这些日积月累的材料，我可以写一本关于开发内核该做什么——更重要的是——不该做什么的书籍。本质上，这本书仍旧是描述Linux内核是如何设计和实现的，但是写法却另辟蹊径，所提供的信息更倾向于实用。通过本书，你就可以做一些内核开发的工作了——并且是使用正确的方法去做。我是一个注重实效的人，因此，这是一本实践的书，它应当有趣、易读且有用。

我希望读者可以从这本书中领略到更多Linux内核的精妙之处（写出来的和没写出来的），也希望读者敢于从阅读本书和读内核代码开始跨越到开始尝试开发可用、可靠且清晰的内核代码。当然如果你仅仅是兴致所至，读书自娱，那也希望你能从中找到乐趣。

从第1版到现在，又过了一段时间，我们再次回到本书，修补遗憾。本版比第1版内容更丰富：修订、补充并增加新的内容和章节，使其更加完善。第1版之后内核的变化已基本稳定。更值得一提的是，Linux内核联盟做出决定[⊖]，近期内不进行2.7版内核的开发。于是，内核开发者打算继续开发并稳定2.6版。这很有意义，而本书从中的最大受益就是在2.6版内核上可以稳定相当长的时间。内核的相对稳定为捕获其本质留下余地。一本书能最终获得认可，并成为内核的规范文档，是我的希望。

不管你学习Linux的目的是什么，我都衷心地希望你能喜欢我的书。

作者的体会

开发Linux内核不需要天赋，不需要有什么魔法，连Unix开发者普遍长着的络腮胡子都不一定要有。内核虽然有一些有趣并且独特的规则和要求，但是它和其他大型软件项目相比，并没有太大差别。像所有的大型软件开发一样，要学的东西确实不少，但是内核开发并不神秘，也不深奥，Linux内核开发者其实并不需要付出比其他开发者更多的努力。

认真阅读源码非常必要，Linux系统代码的开放性其实是弥足珍贵的，不要无动于衷地将它搁置一边，从而浪费了大好资源。实际上就是读了代码还远远不够呢，你应该钻研并尝试着动手改动一些代码。寻找一个bug然后去修改它，改进你的硬件设备的驱动程序，总之要有的放矢地做一

⊖ 这一决定是在加拿大渥太华2004年夏季举办的Linux内核年度开发者大会上做出的。

些实际工作！只有动手写代码才能真正融会贯通。

内核版本

本书基于Linux 2.6内核系列，具体地说，是基于最新版的2.6.10。内核总在不断更新，一本书很难捕获其动态变化。不过，观其变而抓其质，才是我努力的方向。本书力图呈现着眼于未来的资料，尽可能提供其广泛应用的素材。

读者范围

本书是写给那些有志于理解Linux内核的软件开发者的。本书并不逐行逐字地注解内核源代码，也不是指导开发驱动程序或是内核API的参考手册（如果存在标准的内核API的话）。本书的初衷是提供足够多的关于Linux内核设计和实现的信息，希望读过本书的程序员能够拥有较为完备的知识，可以真正开始开发内核代码。无论开发内核是为了兴趣还是为了赚钱，我都希望能够带领读者快速走进Linux内核世界。本书不但介绍了理论而且也讨论了具体应用，可以满足不同读者的不同需要。全书无处不理论联系实践，也并非一味强调理论或是实践。无论你研究Linux内核的动机是什么，我都希望这本书能将内核的设计和实现分析清楚，起到抛砖引玉的作用。

因此，本书覆盖了从核心内核系统的应用到内核设计与实现等各方面内容。我认为这点很重要，值得花功夫讨论。例如，第7章讨论的是下半部机制。其中分别讨论了内核下半部机制的设计和实现（核心内核开发者会感兴趣），随即介绍了如何使用内核提供的接口实现你自己的下半部（这对设备驱动开发者很有用处）。其实，我认为上述两部分内容是相得益彰的，虽然核心内核开发者主要关注的问题是内核内部如何工作，但是也应该清楚如何使用接口；同样，如果设备驱动开发者了解了接口背后的实现机制，自然也会受益匪浅。

这好比学习某些库的API函数与研究该库的具体实现。初看，好像应用程序开发者仅仅需要理解API——我们被灌输的思想是，应该像看待黑盒子一样看待接口。而另一方面，库的开发者也只关心库的设计与实现。但是我认为双方都应该花时间相互学习。能深刻了解操作系统本质的应用程序开发者无疑可以更好地利用它。同样，库开发者也决不应该脱离基于此库的应用程序，埋头开发。因此，我既讨论了内核子系统的设计，也讨论了它的用法，希望本书能对核心开发者和应用开发者都有用。

我假设读者已经掌握了C语言，而且对Linux比较熟悉。如果读者还具有与操作系统设计相关的经验和其他计算机科学的概念就更好不过了。当然，我也会尽可能多地解释这些概念，但如果你仍然不能理解这些知识的话，请看本书最后参考资料中给出的一些关于操作系统设计方面的经典书籍。

本书很适合在大学中作为介绍操作系统的辅助教材，与介绍操作系统理论的书相搭配。对于大学高年级课程或者研究生课程来说，可直接使用本书作为教材。

本书的相关网站

我维护了一个包含本书相关信息的网站：http://tech9.net/rml/kernel_book/。其中包括本书的勘误表、内容扩展和修改，同时也提供了未来重印和再版的信息。希望读者多到这个站点看看。我也对曾在句尾所加的介词深表歉意，那有点画蛇添足，使句子晦涩难懂，扰乱了读者的视线。

再版感谢

与其他作者一样，我决非是一个人躲在山洞里孤苦地写出这本书来的（那也是一件美差，因为有熊相伴）。我能最终完成本书原稿是与无数建议和关怀分不开的。一页纸无法容纳我的感激，但我还是要衷心地感谢所有给予我鼓励、给予我知识和给予我灵感的朋友和同事。

首先我要对为此书付出辛勤劳作的所有编辑表示感谢，尤其要感谢我的责任编辑Scott Meyers，他为这版的出版自始至终倾其心血。有幸与制作编辑George Nedeff再次愉快合作，他的有条不紊使一切变得顺利。还要特别感谢我的文字编辑Margo Catts，衷心希望我们能像她驾驭文字那样驾驭内核。

本版的技术编辑Adam Belay、Martin Pool和 Chris Rivera也是我需要倍加感谢的人，他们独到的洞察力和准确地校对使书稿质量大大提高。他们的工作称得上完美，如果书中仍留有错误，那是我的责任。忘不了杰出的技术编辑Zack Brown，他对第1版所做的一切依然记忆犹新。

许多内核开发者为我提供了大力支持，回答了许多问题，还有那些撰写代码的人。本书正是由于有了这些代码，才有了存在的意义。他们是：Andrea Arcangeli、Alan Cox、Greg Kroah-Hartman、Daniel Phillips、Dave Miller、Patrick Mochel、Andrew Morton、Zwane Mwaikambo、Nick Piggin和Linus Torvalds。还要特别感谢内核的策划者。

还有许多人在我写作期间不断鼓励我，在方方面面都给予我关怀。这本书也凝聚着他们的爱心。他们是：Paul Amici、Scott Anderson、Mike Babbitt、Keith Barbag、Dave Camp、Dave Eggers、Richard Erickson、Nat Friedman、Dustin Hall、Joyce Hawkins、Miguel de Icaza、Jimmy Krehl、Patrick LeClair、Doris Love、Jonathan Love、Linda Love、Randy O'Dowd、Sal Ribaudo和他了不起的妈妈、Chris Rivera、Joey Shaw、Jon Stewart、Jeremy VanDoren及其家人、Luis Villa、Steve Weisberg 和他的全家、Helen Whisnant等。

最后，我要感谢我的父母，感谢他们的爱。

内核黑客万岁！

Robert Love

剑桥，

马萨诸塞州

◇ ◇ ◇ ◇ ◇

关于作者

Robert Love很早就开始使用Linux，而且一直活跃于开源社区。Robert还是Linux内核和GNOME社团的积极分子。最近，他作为高级内核工程师，受聘于Novell公司参与Ximian桌面组的开发。在此之前，他作为内核工程师受聘于MontaVista软件公司。

他的内核项目包括抢占式内核、进程调度程序、内核事件层，还有VM增强和多任务处理性能

优化。他创建和维护的另外两个开源项目是schedutils和GNOME卷管理器。

除此之外，他对内核有不少精彩评论，而且他还是*Linux Journal*杂志的特邀编辑。

Robert拥有佛罗里达大学数学专业和计算机科学专业学士学位。他出生于佛罗里达南部，目前居住在马萨诸塞州东部的剑桥市。他爱好足球、摄影及烹饪。

◇ ◇ ◇ ◇ ◇

读者反馈方式

本书的所有读者都是我们尊敬的批评者和评论者，我们渴望获得你的意见，我们希望了解我们所做的哪些工作是正确的，哪些工作可以做得更好，你们希望我们出版什么内容的书，以及任何对我们有帮助的建议。

你可以直接发Email或写信给我，告诉我这本书哪里你喜欢，哪里你不喜欢——我们如何提高书的质量。

请注意我无法回答你们的关于书中主题的相关技术问题，我接收的邮件太多，我难以一一回复。请在写信时注明书的作者和书中的主题，同时也请写清楚你的姓名、电话和Email地址。我一定会详细地阅读你的评论，并且与书的作者和编辑交流。

Email：feedback@novellpress.com

通信地址：Mark Taber

Associate Publisher

Novell Press/Pearson Education

800 East 96th Street

Indianapolis, IN 46240 USA

如果希望了解该书更多信息和Novell出版社的其他图书，请访问我们的网站www.novellpress.com。可以在搜索框中输入ISBN（除去数字间的连字符）或书的标题来寻找你要找的书。

目　录

第①章

Linux内核简介

Unix虽然已经使用了30年，但仍然是现存操作系统中最强大和优秀的系统之一。从1969年诞生以来，由Dennis Ritchie和Ken Thompson灵感火花点亮的这个Unix产物已经成为了一种传奇，它历经了时间的考验依然声名始终不坠。

Unix是从贝尔试验室的一个失败的多用户操作系统Multics中涅磐而生的。Multics项目被终止后，贝尔实验室计算科学研究中心的人们处于一个没有交互式操作系统可用的境地。在这种情况下，1969年的夏天，贝尔实验室的程序员们设计了一个文件系统原型，而这个原型最终发展演化成了Unix。Thompson首先在一台无人问津的PDP-7型机上实现了这个全新的操作系统。1971年，Unix被移植到了PDP-11型机中。1973年，整个Unix操作系统被用C语言重写，正是当时这个并不太引人注目的举动，给后来Unix系统的广泛移植铺平了道路。第一个在贝尔实验室以外被广泛使用的Unix版本是第六版，被称为V6。

许多其他的公司也把Unix移植到新的机型上去。伴随着这些移植，开发者们按照自己的方式不断地增强系统的功能，并由此产生了若干变体。1977年，贝尔实验室综合各种变体推出了Unix System Ⅲ。在1983年AT&T推出了 System V[⊖]。

由于Unix系统设计简洁并且在发布时提供源代码，所以许多其他组织和团体都对它进行了进一步的开发。加州大学伯克利分校是其中影响最大的一个。他们推出的变体叫Berkeley Software Distributions (BSD)。伯克利最早的Unix是1979年的3BSD。在3BSD以后，又陆续推出了一系列4BSD系统，包括4.0BSD、4.1BSD、4.2BSD和4.3BSD。这些变体中加入了虚拟内存、换页机制和TCP/IP网络协议栈。1993年，伯克利推出了最后一个官方Unix版本：4.4BSD。他们在这个版本中重写了虚拟内存（VM）。今天，BSD的开发者们仍然在Darwin、Dragonfly BSD、FreeBSD、NetBSD 和OpenBSD中继续他们的梦想。

上世纪80和90年代，许多工作站和服务器厂商推出了他们自己的Unix，这些Unix大部分是在AT&T或伯克利发行版的基础上加上一些满足他们特定体系结构需要的特性。这其中就包括Digital的Tru64、HP的HP-UX、IBM的AIX、Sequent的DYNIX/ptx、SGI的IRIX和Sun的Solaris。

由于最初一流的设计和以后多年的创新和逐步提高，Unix系统成为了一个强大、健壮和稳定的操作系统。下面的几个特点是使Unix具有如此良好弹性的（如此成功的）原因。首先，Unix很简洁。不像其他动辄提供数千个系统调用并且设计目的不明确的系统，Unix仅仅提供几百个系统调用并且有一个非常明确的设计目的。第二，在Unix中，所有的东西都被当作文件对待[⊜]。这种

⊖ 那么System IV呢？据说是个内部开发版本。

⊜ 嗯，确实不能说得太绝对——可是绝大部分是被当作文件的。更新一点的操作系统，比如Unix的下一代Plan9，基本上已经把所有的东西用文件实现了。

抽象使对数据和对设备的操作是通过一套相同的系统调用界面进行：open()、read()、write()、ioctl()和close()。第三，Unix的内核和相关的系统工具软件是用C语言编写的。正是这个特点赋予了Unix令人惊异的移植能力，并且使广大的开发人员很容易就能接受它。此外，Unix的进程创建非常迅速，并且有一个非常独特的fork()系统调用。最后，Unix提供了一套非常简单但又很稳定的进程间通信元语。快速简洁的进程创建过程使Unix的程序把目标放在一次执行集中完成一个任务上，而简单稳定的进程间通信机制又可以保证这些独立的简单程序可以方便地组合在一起，从而完成复杂的多重任务。

今天，Unix已经发展成为一个支持多任务、多线程、虚拟内存、换页、动态链接和TCP/IP网络的现代操作系统。Unix的不同变体应用在大到数百个CPU的集群，小到嵌入式设备的各种系统上。尽管Unix已经不再被认为是一个实验室项目了，但它仍然伴随着操作系统设计技术的进步而继续成长，人们仍然可以把它作为一个通用的操作系统来使用。

Unix的成功归功于其简洁和一流的设计。它能拥有今天的能力和成就应该归功于Dennis Ritchie、Ken Thompson和其他早期设计人员的决策，同时也要归功于那些永不妥协于成见，从而赋予Unix无穷活力的设计抉择。

1.1 追寻Linus的足迹：Linux简介

1991年，Linus Torvalds 为当时新推出的、使用Intel 80386微处理器的计算机开发了一款全新的操作系统，Linux由此诞生。那时，作为芬兰赫尔辛基大学一名学生的Linus，为不能随心所欲使用强大而自由的Unix系统而苦恼。对Torvalds 而言，使用Microsoft的DOS产品不亚于玩波斯王子游戏。Linus热衷使用Minix，一种教学用的廉价Unix，但是，他不能轻易修改和发布该系统的源代码（由于Minix的授权），也不能对Minix开发者所作的设计轻举妄动，这让他耿耿于怀。

Linus像任何一名生机勃勃的大学生一样决心走出这种困境：开发自己的操作系统。他开始写了一个简单的终端仿真程序，用于把自己的终端连接到本校的大型Unix系统上。他不断改进和完善这个终端仿真程序，不久，Linus手上就有了虽不成熟但五脏俱全的Unix。1991年年底，他在Internet上发布了早期版本。

由于Linux得以广泛使用，因此很快赢得众多用户。而实际上，它成功的重要因素是，Linux很快吸引了很多开发者对其代码进行修改和完善。由于其许可证条款的约定，Linux迅速成为多人的合作开发项目。

到现在，Linux早已羽翼丰满了，它被广泛移植到AMD x86-64、ARM、Compaq Alpha、CRIS、DEC VAX、H8/300、Hitachi SuperH、HP PA-RISC、IBM S/390、Intel IA-64、MIPS、Motorola 68000、PowerPC、SPARC、UltraSPARC和v850等各种体系结构上。它覆盖的领域小到手表，大到超级计算机集群。今天，Linux的商业前景也越来越被看好，不管是新成立的Linux专业公司MontaVista 和Red Hat还是闻名遐迩的计算巨头IBM和 Novell，都提供林林总总的解决方案，从嵌入式系统、桌面环境一直到服务器。

Linux克隆了Unix，但Linux不是Unix。需要说明的是，尽管Linux借鉴了Unix的许多设计并且实现了Unix的API（由Posix标准和其他Single Unix Specification定义的），但Linux没有像其他Unix变种那样直接使用Unix的源代码。必要的时候，它的实现可能和其他各种Unix的实现大相径庭，但它完整地达成了Unix的设计目标并且保证了应用程序编程界面的一致。

Linux是一个非商业化的产品，这是它最让人感兴趣的特征。实际上Linux是一个因特网上的协作开发项目。尽管Linus被认为是Linux之父，并且现在依然是一个内核维护者，但开发工作其实是由一个结构松散的工作组协力完成的。事实上，任何人都可以开发内核。和Linux的大部分应用软件一样，Linux内核也是自由（公开）软件[⊖]。当然，也不是无限自由的。它使用GNU的General Public License(GPL)第二版作为限制条款。这样做的结果是，你可以自由地获取内核代码并随意修改它，但如果你希望发布你修改过的内核，你也得保证让得到你的内核的人同时享有你曾经享受过的所有权利，当然，包括全部的源代码[⊜]。

Linux用途广泛，包含的东西也名目繁多。Linux系统的基础是内核、C库、编译器、工具集和系统的基本工具，如登录程序和shell。Linux系统也支持现代的X Windows系统，这样就可以使用完整的图形用户桌面环境，如GNOME。可以在Linux上使用的商业和自由软件数以千计。在这本书以后的部分，当我使用Linux这个词时，我其实说的是Linux内核。在容易引起混淆的地方，我会具体说明到底我想说的是整个系统还是内核。一般情况下，Linux这个词汇主要还是指内核。

1.2 操作系统和内核简介

由于现行一些商业操作系统日趋庞杂及设计上的缺陷，操作系统这个概念被弄得含混不清。许多用户把他们在显示器屏幕上看到的东西理所当然的认为就是操作系统。通常，当然在本书中也这么认为，操作系统是指在整个系统中负责完成最基本功能和系统管理的那些部分。这些部分应该包括内核、设备驱动程序、启动引导程序、命令行shell或者其他种类的用户界面、基本的文件管理工具和系统工具。这些都是必不可少的东西——别以为只要有浏览器和播放器就行了。系统这个词其实包含了操作系统和所有运行在它之上的应用程序。

当然，本书的“操作系统”是关于内核的。用户界面是操作系统的外在表象，内核才是操作系统的内在核心。系统其他部分必须依靠内核这部分软件提供的服务，像管理硬件设备，分配系统资源等等。内核有时候被称作是超级管理者或者是操作系统核心。通常一个内核由负责响应中断的中断服务程序，负责管理多个进程从而分享处理器时间的调度程序，负责管理进程地址空间的内存管理程序和网络、进程间通信等系统服务程序共同组成。对于提供保护机制的现代系统来说，内核独立于普通应用程序，它一般处于系统态，拥有受保护的内存空间和访问硬件设备的所有权限。这种系统态和被保护起来的内存空间，统称为内核空间。相对的，应用程序在用户空间执行。它们只能看到允许它们使用的部分系统资源，并且不能使用某些特定的系统功能，不能直接访问硬件，还有其他一些使用限制（其实，没有固定的结论，例如，它们不再被使用）。当内核运行的时候，系统以内核态进入内核空间，相反，普通用户程序以用户态进入用户空间。应用程序通过系统调用和内核通信来运行（参见图1-1）。应用程序通常调用库函数——比如说C库函数——再由库函数通过系统调用界面让内核代其完成各种不同任务。许多库函数提供的功能并没有单独的系统调用可以替代，在那些较为复杂的函数中，调用内核的操作通常只是整个工作的一个步骤而已。举个例子，拿printf()函数来说，它提供了数据的缓存和格式化等操作，也只是在执行的末期

⊖ 谁有兴趣了解free VS open的论战，请参见http://www.fsf.org and http://www.opensource.org。

⊜ 如果你没读过GNU GPL，你最好还是先看看它吧。内核代码树中COPYING文件就是它的一份拷贝。你也可以在http://www.fsf.org中找到它。

通过write()系统调用把处理后的最终数据写在终端上而已。不过，也有一些库函数和系统调用就是一一对应的关系，比如open()库函数除了调用open()系统调用，确实其他什么也不做。还有一些C库函数，像strcpy()，根本就不需要调用系统级的操作。当一个应用程序请求执行一条系统调用，我们说内核正在代其执行。如果进一步解释，在这种情况下，应用程序被称为通过系统调用在内核空间运行，而内核被称为运行于进程上下文中。这种交互关系——应用程序通过系统调用陷入内核——是应用程序完成其工作的基本行为方式。

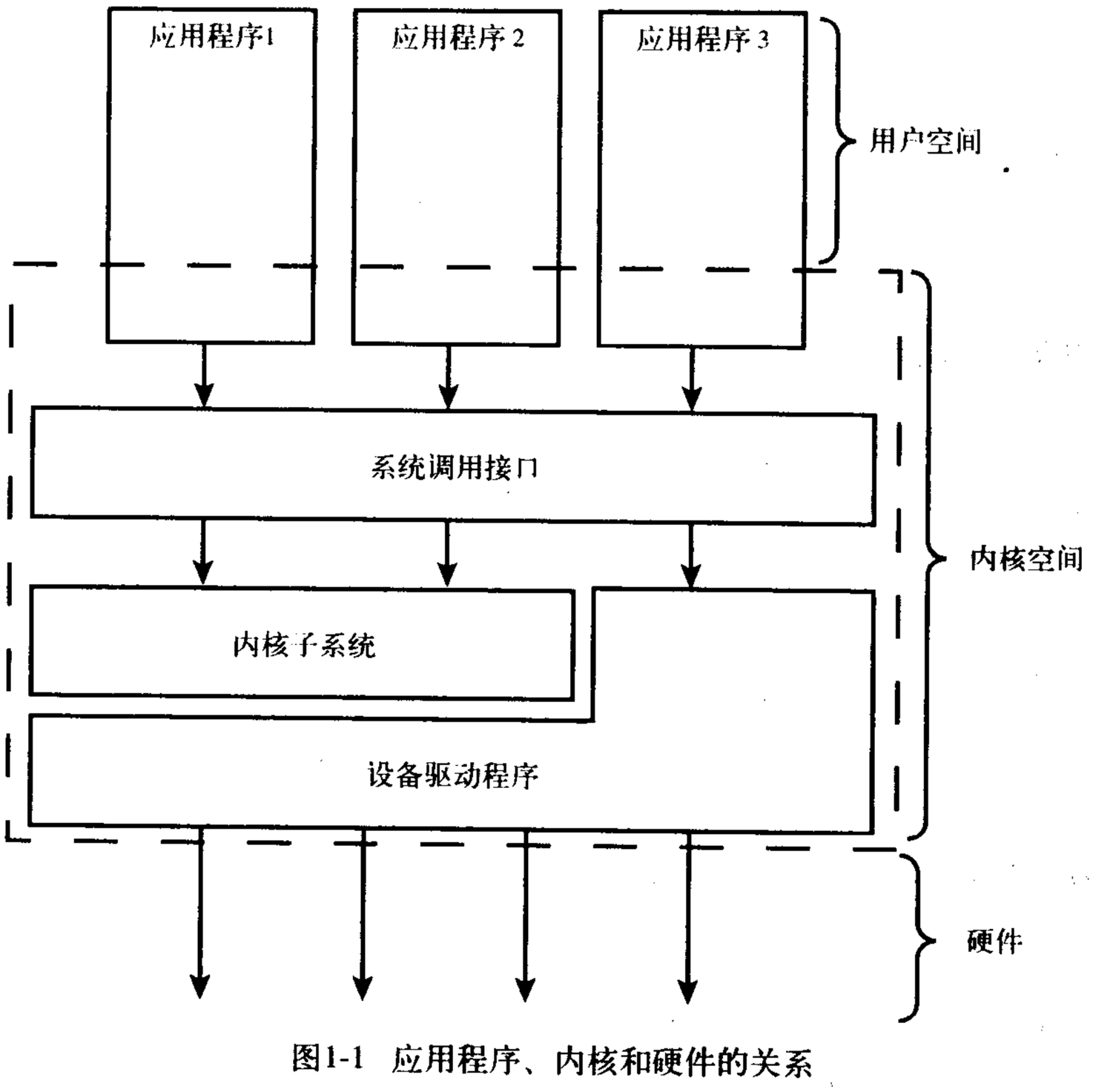


图1-1 应用程序、内核和硬件的关系

内核还要负责管理系统的硬件设备。现有的几乎所有的体系结构，包括全部Linux支持的体系结构，都提供了中断机制。当硬件设备想和系统通信的时候，它首先要发出一个异步的中断信号去打断内核正在执行的工作。中断通常对应着一个中断号，内核通过这个中断号查找相应的中断服务程序，并调用这个程序响应和处理中断。举个例子，当你敲击键盘的时候，键盘控制器发送一个中断信号，告知系统键盘缓冲区有数据到来。内核注意到这个中断对应的中断号，调用相应的中断服务程序。该服务程序处理键盘数据然后通知键盘控制器可以继续输入数据了。为了保证同步，内核可以停用中止——既可以停止所有的中断也可以有选择的停止某个中断号对应的中断。许多操作系统的中断服务程序都不在进程上下文中执行。它们在一个与所有进程都无关的、专门的中断上下文中运行。之所以存在这样一个专门的执行环境，就是为了保证中断服务程序能够在第一时间响应和处理中断请求，然后快速的退出。

这些上下文代表着内核活动的范围。实际上我们可以将处理器在任何指定时间点上的活动范

围概括为下列三者之一：

- 运行于内核空间，处于进程上下文，代表某个特定的进程执行。
- 运行于内核空间，处于中断上下文，与任何进程无关，处理某个特定的中断。
- 运行于用户空间，执行用户进程。

以上所列几乎包括所有情况。即使边边角角的情况也不例外，例如，当CPU空闲时，内核就运行一个空进程，处于进程上下文，但运行于内核空间。

1.3 Linux内核和传统Unix内核的比较

由于所有的Unix内核都同宗同源，并且提供相同的API，现代的Unix内核存在许多设计上的相似之处。Unix内核几乎毫无例外的都是一个不可分割的静态可执行块（文件）。 也就是说，它们必须以完整、单独的可执行块的形式在一个单独的地址空间中运行。Unix内核几乎都需要硬件系统提供页机制以管理内存。这种页机制可以加强内存空间的保护，并保证每个进程都可以运行于不同的虚地址空间上。关于这方面，请参阅我提供的参考书籍中有关Unix内核设计那部分里包含的书籍。

单内核与微内核设计之比较

操作系统内核可以分为两大设计阵营：单内核和微内核（第三阵营外内核，主要用在科研系统中，但也逐渐在现实世界中壮大起来）。

单内核是两大阵营中一种较为简单的设计，在1980年之前，所有的内核都设计成单内核。所谓单内核就是把它从整体上作为一个单独的大过程来实现，并同时运行在一个单独的地址空间。因此，这样的内核通常以单个静态二进制文件的形式存放于磁盘。所有内核服务都在这样的一个大内核空间中运行。内核之间的通信是微不足道的，因为大家都运行在内核态，并身处同一地址空间：内核可以直接调用函数，这与用户空间没有什么区别。这种模式的支持者认为单模块具有简单和高性能的特点。大多数Unix系统都设计为单模块。

另一方面，微内核并不作为一个单独的大过程来实现。相反，微内核的功能被划分为独立的过程，每个过程叫做一个服务器。理想情况下，只有强烈请求特权服务的服务器才运行在特权模式下，其他服务器都运行在用户空间。不过，所有的服务器都保持独立并运行在各自的地址空间。因此，就不可能像单模块内核那样直接调用函数，而是通过消息传递处理微内核通信：系统采用了进程间通信(IPC)机制，因此，各种服务器之间通过IPC机制互通消息，互换“服务”。服务器的各自独立有效地避免了一个服务器的失效祸及另一个。

同样，模块化的系统允许一个服务器为了另一个服务器而换出。因为IPC机制的开销比函数调用多，又因为会涉及内核空间到用户空间的上下文切换，因此，消息传递需要一定的周期，而单内核中简单的函数调用没有这些开销。基于此，付之于实际的微内核系统让大部分或全部服务器位于内核，这样，就可以直接调用函数，消除频繁的上下文切换。Windows NT内核和Mach（Mac OS X的组成部分）是微内核的典型实例。不管是Windows NT还是Mac OS X，都在其新近版本中不让任何微内核服务器运行在用户空间，这违背了微内核设计的初衷。

Linux是一个单内核，也就是说，Linux内核运行在单独的内核地址空间。不过，Linux汲取了微内核的精华：其引以为豪的是模块化设计、抢占式内核、支持内核线程以及动态装载内核

模块的能力。不仅如此，Linux还避其微内核设计上性能损失的缺陷，让所有事情都运行在内核态，直接调用函数，无需消息传递。至今，Linux是模块化的、多线程的以及内核本身可调度的操作系统。实用主义再次占了上风。

当Linus和其他内核开发者设计Linux内核时，他们并没有完全彻底地与Unix诀别。他们充分地认识到，不能忽视Unix的底蕴（特别是Unix的API）。而由于Linux并没有基于某种特定的Unix，Linus和他的伙伴们对每个特定的问题都可以选择已知最理想的解决方案——在有些时候，当然也可以创造一些新的方案。以下是对Linux 内核与Unix各种变体的内核特点所作的分析比较：

- Linux支持动态加载内核模块。尽管Linux内核也是单内核，可是允许在需要的时候动态地卸除和加载部分内核代码。
- Linux支持对称多处理（SMP）机制，尽管许多Unix的变体也支持SMP，但传统的Unix并不支持这种机制。
- Linux内核可以抢占（preemptive）。与传统的Unix不同，Linux内核具有允许在内核运行的任务优先执行的能力。在其他各种Unix产品中，只有Solaris和IRIX支持抢占，但是大多数传统的Unix内核不支持抢占。
- Linux对线程支持的实现比较有意思：内核并不区分线程和其他的一般进程。对于内核来说，所有的进程都一样——只不过其中的一些共享资源而已。
- Linux提供具有设备类的面向对象的设备模型、热插拔事件，以及用户空间的设备文件系统（sysfs）。
- Linux忽略了一些被认为是设计得很拙劣的Unix特性，像STREAMS，它还忽略了那些实际上已经根本不会使用的过时标准。
- Linux体现了自由这个词的精髓。现有的Linux特性集就是Linux公开开发模型自由发展的结果。如果一个特性没有任何价值或者创意很差，没有任何人会被迫去实现它。相反的，在Linux的发展过程中已经形成了一种值得称赞的务实态度：任何改变都要针对现实中确实存在的问题，经过完善的设计并有正确简洁的实现。于是，许多其他现代Unix系统包含的特性，如内核换页机制，都被毫不迟疑的引入进来。

不管Linux和Unix有多大的不同，它身上都深深地打上了Unix烙印。

1.4 Linux内核版本

Linux内核有两种：稳定的和处于开发中的。稳定的内核具有工业级的强度，可以广泛的应用和部署。新推出的稳定内核大部分都只是修正了一些Bug或是加入了一些新的设备驱动程序。相反地，处于开发中的内核中许多东西变化得都很快。而且由于开发者不断试验新的解决方案，内核常常发生剧烈的变化。

Linux通过一个简单的命名机制来区分稳定的和处于开发中的内核（参考图1-2）。这种机制使用三个用“.”分隔的数字来代表不同内核。第一个数字是主版本号，第二个数字是从版本号，第三个数字是修订版本号。从版本号可以反映出该内核是一个稳定版本还是一个处于开发中的版本：该数字如果是偶数，那么此内核就是稳定版，如果是奇数，那么它就是开发版。举例来说，版本号为2.6.0的内核，它就是一个稳定版。这个内核的主板本号是2，从版本号是6，修订版本号是0。

头两个数字在一起描述了“内核系列”——在这个例子中，就是2.6版内核系列。

处于开发中的内核一般要经历几个阶段。最开始，内核开发者们开始试验新的特性，这时候出现错误和混乱是在所难免的。经过一段时间，系统渐渐成熟，直到一个新的特性被宣布通过审定。这时候，就不再允许加入新的特性了。而对已有特性所进行的后续工作会继续进行，当这个新内核确实被认为是稳定下来以后，就开始审定代码。这以后，就只允许再向其中加入修改bug的代码了。在经过一个短暂（希望如此）的准备期，这个内核会作为一个新的稳定版被推出。例如，1.3系列的开发版稳定在2.0，而2.5稳定在2.6。

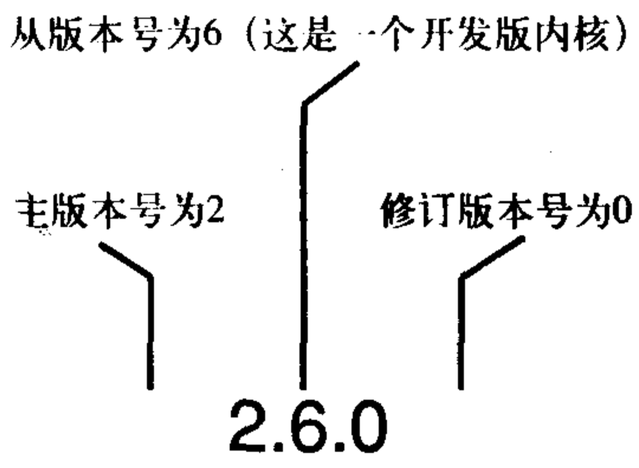


图1-2 Kernel版本命名规则

事实并非如此

从技术上说，如前所述的内核开发过程的确如此，Linux发展史也没有例外。但是，2004年的夏季，在Linux内核开发者联盟的年度例会上做出了一个决策，暂缓2.7开发版系列，稳定2.6内核。这一决策源于2.6内核深受欢迎、稳定可靠并具有高安全性。除此之外，也许更重要的是，当前2.6内核的维护者Linus Torvalds 和Andrew Morton一直不遗余力地工作着。内核开发者坚信这一过程会持续下去，2.6内核系列既保持稳定，又不断吸收新鲜血液。随着时间的推移，我们已经看到，也即将看到，一切像预期的那样向前推进。

这本书是基于2.6这个稳定的Linux版本的。

1.5 Linux内核开发者社区

当你开始开发内核代码时，你就成为全球内核开发社区的一分子了。这个社区最重要的论坛是*Linux kernel mailing list*。你可以在http://vger.kernel.org上订阅邮件。要注意的是这个邮件列表流量很大，每天有超过300条的消息，所以其他的订阅者——包括所有的核心开发人员，甚至包括Linus本人——可没有心思听人说废话。这个邮件列表可以给从事内核开发的人提供价值无穷的帮助，在这里，你可以寻找测试人员，接受评论（peer review），向人求助。

后续章节列出了内核开发过程的全景，并详尽地描述了如何成功地加入到内核开发社区中去。

1.6 小结

这是一本关于Linux内核的书：它怎么工作、为什么这样工作以及为什么你要留意这些等等。本书主要涵盖了内核各个核心子系统的设计和实现，并介绍了它们的接口和程序设计时所遵循的语义。本书侧重实用，但并不走极端，希望把上述方面剖析清楚，力求以一种有趣的方式——伴随着我个人在开发内核过程中收集的种种奇闻逸事和方法技巧——引导你走过跌跌撞撞的起步阶段。

我希望你有一台装有Linux的机器，我希望你能够看到内核代码。其实，这很理想了，因为这意味着你是一位Linux的使用者，并且早已经开始拿起手术刀对着源代码开始刺探了，只不过需要一份结构图以便对整个经脉有个总体把握罢了。相反，你可能没有使用过Linux，只是在好奇心的驱使下希望了解一些内核设计的秘密而已。但是，如果你的目的只是撰写自己的代码，那么，源代码的作用无可替代。而且，你不需要付出任何代价；尽管用吧。

好了，最重要的是，在其中寻找快乐吧。

第②章

从内核出发

在这一章，我们介绍Linux内核一些基本常识：从何处获取源码，如何编译它，又如何安装新内核。那么，让我们考察一下内核的一些状态、内核程序与用户空间程序的差异，以及内核所用一般函数的特点。

内核像性格怪异的猛兽，但并非不可驯服。让我们来驾驭它。

2.1 获取内核源码

在Linux内核官方网站http://www.kernel.org，可以随时获取当前版本的Linux源代码，可以是完整的压缩形式，也可以是增量补丁形式。

除特殊情况下需要Linux源码的旧版本，一般都希望拥有最新的代码。kernel.org是源码的库存之处，那些领导潮流的内核开发者所发布的增量补丁也放在这里。

2.1.1 安装内核源代码

内核压缩以GNU zip(gzip)和bzip2两种形式发布。bzip2是默认和首选形式，因为它在压缩上比gzip有相当的优势。以bzip2形式发布的Linux内核叫做linux-x.y.z.tar.bz2，这里x.y.z是内核源码的具体版本。下载了源代码之后，就可以轻而易举地对其解压。如果压缩形式是bzip2，则运行：

```
$ tar xvjf linux-x.y.z.tar.bz2
```

如果压缩形式是GNU的zip，则运行

```
$ tar xvzf linux-x.y.z.tar.gz
```

解压后的源代码位于linux-x.y.z.目录下。

何处安装源码

内核源码一般安装在/usr/src/linux目录下。但请注意，不要把这个源码树用于开发。相反，编译你的C库所用的内核版本就链接到这棵树。此外，不要以root身份对内核进行修改，而应当是，建立自己的主目录，仅以root身份安装新内核。即使在安装新内核时，/usr/src/linux目录都应当原封不动。

2.1.2 使用补丁

在Linux内核社区中，补丁是通用语。你可以以补丁的形式发布对代码的修改，也可以以补丁的形式接收其他人所做的修改。时间流逝，内核版本在不断更新，增量补丁可以作为版本转移的桥梁。你不再需要下载内核源码的全部压缩，而只需给旧版本打上一个增量补丁，让其旧貌换新

颜。这不仅节约了带宽，还省了时间。要应用增量补丁，从你的内部源码树开始，只需运行:

```
$ patch-p1 < ../patch-x.y.z
```

一般说来，一个给定版本的内核补丁总是打在前一个版本上。

对产生和应用补丁更深入的讨论会在后续章节进行。

2.2 内核源码树

内核源码树由很多目录组成，而大多数目录又包含更多的子目录。源码树的根目录及其子目录如表2-1所示。

表2-1 内核源码树的根目录描述

目 录	描 述
arch	特定体系结构的源码
crypto	Crypto API
Documentation	内核源码文档
drivers	设备驱动程序
fs	VFS和各种文件系统
include	内核头文件
init	内核引导和初始化
ipc	进程间通信代码
kernel	像调度程序这样的核心子系统
lib	通用内核函数
mm	内存管理子系统和VM
net	网络子系统
scripts	编译内核所用的脚本
security	Linux安全模块
sound	语音子系统
usr	早期用户空间代码 (所谓的 initramfs)

在源码树根目录中的很多文件值得提及。COPYING文件是内核许可证（GNU GPL v2）。CREDITS是开发者列表，其中还有不少内核代码的细节。MAINTAINERS是维护者列表，维护内核子系统和驱动程序。最后， Makefile是Makefile内核的基础。

2.3 编译内核

编译内核易如反掌。让人叹为观止的是，这实际上比一些像glibc这样的用户级软件包还要简单。2.6内核提供了一套新工具，使编译内核更加容易，比2.4内核有了长足的进步。

因为Linux源码随手可得，那就意味着在编译它之前可以配置和定制。的确，你可以把自己需要的功能和驱动程序编译进内核。在编译内核之前，首先你必须配置它。由于内核提供了数不胜数的功能，支持了难以计数的硬件，因而有许多东西需要配置。可以配置的各种选项，以CONFIG_FEATURE形式表示，其前缀为CONFIG。例如，对称多处理器（SMP）的配置选项为CONFIG_SMP。如果设置了该选项，则SMP启用，否则，SMP不起作用。配置选项既可以用来决定哪些文件编译进内核，也可以通过预处理命令处理代码。

这些配置项要么是二选一，要么是三选一。二选一就是yes或no。比如说CONFIG_PREEMPT

就是二选一，表示内核抢占功能是否开启。三选一可以是yes、no或module。Module意味着该配置项被选定了，但编译的时候这部分功能的实现代码是以模块（一种可以动态安装的独立代码段）的形式生成。在三选一的情况下，显然yes选项表示把代码编译进主内核映像中，而不是作为一个模块。驱动程序一般都用三选一的配置项。

配置选项也可以是字符串或整数。这些选项并不控制编译过程，而只是指定内核源码可以访问的值，一般以预处理宏的形式表示。比如说，配置选项可以指定静态分配数组的大小。

像Novell和Red Hat这样的商家，他们的发布版中包含了预编译的内核，这样的内核使得所需的功能得以充分的启用，并几乎把所有的驱动程序都编译成模块。这就为大多数硬件作为独立的模块提供了坚实的内核支持。但是，话又说回来，如果你是一个内核黑客，你应当编译自己的内核，并按自己的意愿决定包括或不包含哪一模块。

内核提供了各种不同的工具来简化内核配置。最简单的一种是一个基于文本的命令行工具：

```
$ make config
```

该工具会挨个遍历所有配置项，要求用户选择yes、no或是module（如果是三选一的话）。由于这个过程往往要耗费掉很长时间，所以，除非你的工作是按小时计费的，否则你应该多利用基于ncurse库编制的图形界面工具：

```
$ make menuconfig
```

或者，用基于X11的图形工具:

```
$ make xconfig
```

或者更甚，用基于gtk+图形工具：

```
$ make gconfig
```

这三种工具将所有配置项分门别类放置，比如按“处理器类型和特点”。你可以按类移动、浏览内核选项，当然也可以修改其值。

命令

```
$ make defconfig
```

通过这条命令为你的体系结构创建一个默认的配置。尽管这些默认值有点随意性（在i386上，据说那就是Linus的配置！），但是，如果你以前从未配置过内核，那就会提供一个良好的开端。赶快行动吧，运行这条命令，然后回头看看，确保为你的硬件所配置的选项是启用的。

这些配置项会存放在内核代码树根目录下的config文件中。很容易就能找到它——别的内核开发者差不多都能找到——并且可以直接修改它。在这里面查找和修改内核选项也很容易。在修改过配置文件之后，或者在用旧的配置文件配置新的代码树的时候，应该验证和更新配置：

```
$ make oldconfig
```

事实上，在编译内核之前都应该这么做。一旦内核配置好了，就可以编译它了:

```
$ make
```

这跟2.6以前的版本不同，不用再运行make dep了——代码之间的依赖关系会自动维护。也无需再指定像老版本中bzImage这样的编译方式或独立地编译模块。默认的Makefile规则会打点这一切。

2.3.1 减少编译的垃圾信息

我们希望在编译时看到错误和警告信息，但对匆匆掠过屏幕的垃圾信息不感兴趣，可以用下

面的技巧来实现愿望：

```
$ make >../some_other_file
```

一旦需要查看编译的输出信息，可以查看这个文件。不过，因为错误和警告都会在屏幕上显示，所以需要看这个文件的可能性不大。事实上，只用敲入如下命令：

```
$ make > /dev/null
```

就可把无用的输出信息重定向到永无返回值的黑洞/dev/null。

2.3.2 衍生多个编译作业

make(1)程序能把编译过程拆分成多个作业。其中的每个作业独立并发地运行，这能极大地提高多处理器系统上的编译过程，并有利于改善处理器的利用率，因为编译大型源代码树还包括I/O等待所花费的时间（也就是处理器空下来等待I/O请求完成所花费的时间）。

默认情况下，make(1)只衍生一个作业。Makefiles时常把文件之间的依赖弄乱。对于不正确的依赖，多个作业可能会互相踩踏，导致编译过程出错。当然，内核的Makefiles没有这样的编码错误。为了以多个作业编译内核，使用以下命令：

```
$ make -jn
```

这里，n是要衍生的作业数，在实际中，每个处理器上一般衍生一个或者两个作业。例如，在一个双处理器上，可以输入如下命令：

```
$ make -j4
```

利用出色的distcc(1) 或者 ccache(1)工具，也可以动态地改善内核的编译时间。

2.3.3 安装内核

在内核编译好之后，还需要安装它。怎么安装就和体系结构以及启动引导工具（boot loader）息息相关了——查阅启动引导工具的说明，按照它的指导将内核映像拷贝到合适的位置，并且按照启动要求安装它。一定要保证随时有一个或两个可以启动的内核，以防新编译的内核出现问题。

举个例子，在使用grub的x86系统上，可能需要把arch/i386/boot/bzImage拷贝到/boot目录下，依照vmlinuz-*version*来命名它，并且编辑/boot/grub/grub.conf文件，为新内核建立一个新的启动项。使用LILO启动的系统应当编辑/etc/lilo.conf，然后运行lilo(8)。

所幸，模块的安装是自动的，也是独立于体系结构的。以root身份，只要运行：

```
% make modules_install
```

就可以把所有已编译的模块安装到正确的主目录/lib下。

编译时还会在内核代码树的根目录下创建一个System.map文件。这是一份符号对照表，用以将内核符号和它们的起始地址对应起来。调试的时候，如果把内存地址翻译成容易理解的函数名以及变量名会很有用。

2.4 内核开发的特点

相对于用户空间内应用程序的开发，内核开发有很大的不同。这种差异给开发内核带来了特别的挑战，但这并不意味着开发内核一定比开发应用程序难很多。

这种差异使内核成了一只性格迥异的猛兽。一些常用的准则被颠覆了，而又必须建立许多全新的准则。尽管有许多差异一目了然（人人都知道内核可以做它想做的任何事），但还是有一些差异晦暗不明。最重要的差异包括以下几种：

- 内核编程时不能访问C库。
- 内核编程时必须使用GNU C。
- 内核编程时缺乏像用户空间那样的内存保护机制。
- 内核编程时浮点数很难使用。
- 内核只有一个很小的定长堆栈。
- 由于内核支持异步中断、抢占和SMP，因此必须时刻注意同步和并发。
- 要考虑可移植性的重要性。

让我们仔细考察一下这些要点，所有这些东西在内核开发中必须时刻牢记。

2.4.1 没有libc库

与用户空间的应用程序不同，内核不能链接使用标准C函数库（其他的那些库也不行）。造成这种情况的原因有许多，其中就包括先有鸡还是先有蛋这个悖论。不过最主要的原因还是在于速度和大小。对内核来说，完整的C库太大了——即便是从中抽取一个合适的子集——大小和效率都不能被接受。

别着急，大部分常用的C库函数在内核中都已经得到了实现。比如说操作字符串的函数组就位于lib/string.c文件中。只要包含<linux/string.h>头文件，就可以使用它们。

头文件

当我在本书的任何地方谈及头文件时，都指的是组成内核源代码树的内核头文件。内核源代码文件不能包含外部头文件，就像它们不能用外部库一样。

在所有没有实现的函数中，最著名的就数printf()函数了。内核代码虽然无法调用printf()，但它可以调用printk()函数。printk()函数负责把格式化好的字符串拷贝到内核日志缓冲区上，这样，syslog程序就可以通过读取该缓冲区来获取内核信息。printk()的用法很像printf()：

```
printk("Hello world! A string: %s and an integer: %d\n", a_string, an_integer);
```

printk()和printf()之间的一个显著区别在于printk()允许通过指定一个标志来设置优先级。Syslog会根据这个优先级标志来决定在什么地方显示这条系统消息。下面是一个使用这种优先级标志的例子：

```
printk(KERN_ERR "this is an error!\n");
```

我们在这本书里很多地方都要用到printk()。在后续章节中有关于printk()函数的更详细的信息。

2.4.2 GNU C

像所有自视清高的Unix内核一样，Linux内核是用C语言编写的。让人略感惊讶的是，内核并不完全符合ANSI C标准。实际上，只要有可能，内核开发者总是要用到gcc提供的许多语言扩展部分。（gcc是多种GNU编译器的集合，它包含的C编译器既可以编译内核，也可以编译Linux系统上

用C写的其他代码。）

内核开发者使用的C语言涵盖了ISO C99[⊖]标准和GNU C扩展特性。这其中的种种变化把Linux内核推向了gcc的怀抱。尽管目前出现了一些新的编译器如Intel C，已经支持了足够多的gcc扩展特性，完全可以用来编译Linux内核了。Linux内核用到的ISO C99标准的扩展没有什么特别之处，而且C99作为C语言官方标准的修订本，不可能有大的或是激进的变化。让人感兴趣的，与标准C有区别，而通常也是人们不熟悉的那些变化，多数集中在GNU C上。就让我们研究一下内核代码中所使用到的C语言扩展中让人感兴趣的那部分吧。

1. 内联（inline）函数

GNU的C编译器支持内联函数。Inline这个名称（译者注：inline翻译成内联似乎并不贴切，直译应该是“在字里行间展开”的意思，不过约定俗成，把它翻译成“内联”）就可以反映出它的工作方式，函数会在它所调用的位置上展开。这么做可以消除函数调用和返回所带来的开销（寄存器存储和恢复），而且，由于编译器会把调用函数的代码和函数本身放在一起进行优化，所以也有进一步优化代码的可能。不过，这么做是有代价的（天下没有免费的午餐），代码会变长，这就意味着占用更多的内存空间或者占用更多的指令缓存。内核开发者通常把那些对时间要求比较高，而本身长度又比较短的函数定义成内联函数。如果你把一个大块头的程序做成了内联函数，却并不需要争分夺秒，反而反复调用它，又怎能不让人叹气呢？

定义一个内联函数的时候，需要使用static作为关键字，并且用inline限定它。比如：

```
static inline void dog(unsigned long tail_size)
```

内联函数必须在使用之前就定义好，否则编译器就没法把这个函数展开。实践中一般在头文件中定义内联函数。由于使用了static作为关键字进行限制，所以编译时不会为内联函数单独建立一个函数体。如果一个内联函数仅仅在某个源文件中使用，那么也可以把它定义在该文件开始的地方。

在内核中，为了类型安全的原因，优先使用内联函数而不是复杂的宏。

2. 内联汇编

gcc编译器支持在C函数中嵌入汇编指令。当然，在内核编程的时候，只有知道对应的体系结构，才能使用这个功能。

Linux的内核混合使用了C和汇编语言。在偏近体系结构的底层或对执行时间要求严格的地方，一般使用的是汇编语言。而内核其他部分的大部分代码是C语言写的。

3. 分支声明

对于条件选择语句，gcc内建了一条指令用于优化，在一个条件经常出现，或者该条件很少出现的时候，编译器可以根据这条指令对条件分支选择进行优化。内核把这条指令封装成了宏，比如likely()和unlikely()，这样使用起来比较方便。

例如，下面是一个条件选择语句：

```
if (foo) {
        /* ... */
}
```

如果想要把这个选择标记成绝少发生的分支：

⊖ ISO C99是ISO C的最新修订版。C99相对于前一个修订版做了许多加强，ISO C90引入了命名结构体初始化和complex数据类型。后者不能在内核内安全地使用。

```
/* 我们认为foo绝大多数时间都会为0... */
if (unlikely(foo)) {
        /* ... */
}
```

相反，如果我们想把一个分支标记为通常为真的选择：

```
/* 我们认为foo通常都不会为0 */
if (likely(foo)) {
        /* ... */
}
```

在想要对某个条件选择语句进行优化之前，一定要搞清楚其中是不是存在这么一个条件，在绝大多数情况下都会成立。这点十分重要：如果你的判断正确，确实是这个条件占压倒性的地位，那么性能会得到提升，如果你搞错了，性能反而会下降。在对一些错误条件进行判断的时候，常常用到unlikely()和likely()。可以猜到，unlikely()在内核中得到广泛使用，因为if语句往往判断一种特殊情况。

2.4.3 没有内存保护机制

如果一个用户程序试图进行一次非法的内存访问，内核会发现这个错误，发送 SIGSEGV，并结束整个进程。然而，如果是内核自己非法访问了内存，那后果就很难控制了。（毕竟，有谁能照顾内核呢？）内核中发生的内存错误会导致oops，这是内核中出现的最常见的一类错误。在内核中，不应该去做访问非法的内存地址，引用空指针之类的事情，否则它可能会死掉，却根本不知会你一声——在内核里，风险常常会比外面大一些。

此外，内核中的内存都不分页。也就是说，每用掉一个字节，物理内存就减少一个字节。所以，在你想往内核里加入什么新功能的时候，要记住这一点。

2.4.4 不要轻易在内核中使用浮点数

在用户空间的进程内进行浮点操作的时候，内核会完成从整数操作到浮点数操作的模式转换。在执行浮点指令时到底会做些什么，因体系结构不同，内核的选择也不同，但是，内核通常捕获陷阱并做相应处理。

和用户空间进程不同，内核并不能完美地支持浮点操作，因为它本身不能陷入。 在内核中使用浮点数时，除了要人工保存和恢复浮点寄存器，还有其他一些琐碎的事情要做。如果要直截了当的回答，那就是：别这么做了，不要在内核中使用浮点数。

2.4.5 容积小而固定的栈

用户空间的程序可以从栈上分配大量的空间来存放变量，甚至巨大的结构体或者是包含许多数据项的数组都没有问题。之所以可以这么做，是因为用户空间的栈本身比较大，而且还能动态的增长(年长的开发者回想一下DOS那个年代，这种缺少智能的操作系统即使在用户空间也只有固定大小的栈)。

内核栈的准确大小随体系结构而变。在x86上，栈的大小在编译时配置，可以是4KB也可以是8KB。从历史上说，内核栈的大小是两页，这就意味着，32位机的内核栈是8KB，而64位机是

16KB，这是固定不变的。每个处理器都有自己的栈。

关于内核栈的更多内容，会在后面的章节中讨论。

2.4.6 同步和并发

内核很容易产生竞争条件。和单线程的用户空间程序不同，内核的许多特性都要求能够并发的访问共享数据，这就要求有同步机制保证不出现竞争条件，尤其是：

- Linux是抢占多任务操作系统。内核的进程调度程序即兴对进程进行调度和重新调度。内核必须对这些任务同步。
- Linux内核支持多处理器系统。所以，如果没有适当的保护，在两个或两个以上的处理器上运行的代码很可能会同时访问共享的同一个资源。
- 中断是异步到来的，完全不顾及当前正在执行的代码。也就是说，如果不加以适当的保护，中断完全有可能在代码访问共享资源的当中到来，这样，中段处理程序就有可能访问同一资源。
- Linux内核可以抢占。所以，如果不加以适当的保护，内核中一段正在执行的代码可能会被另外一段代码抢占，从而有可能导致几段代码同时访问相同的资源。

常用的解决竞争的办法是自旋锁和信号量。

我们将在后面的章节中详细讨论同步和并发执行。

2.4.7 可移植性的重要性

尽管用户空间的应用程序不太注意移植问题，然而Linux却是一个可移植的操作系统，并且一直保持这种特点。也就是说，大部分C代码应该与体系结构无关，在许多不同体系结构的计算机上都能够编译和执行，因此，必须把体系结构相关的代码从内核代码树的特定目录中适当地分离出来。

诸如保持字节序、64位对齐、不假定字长和页面长度等一系列准则都有助于移植性。对移植性的深度讨论将在后面的章节进行。

2.5 小结

内核的确是一头独一无二的猛兽：没有内存保护，没有靠得住的libc，小的堆栈，庞大的源码树。Linux内核遵循它自己的游戏规则，以大人物的架势运行，运行足够长的时间后才停止，打破了我们惯以为常的习俗。尽管如此，内核不外乎就是一个程序，它与我们司空见惯的程序没有多大区别。不必望而生畏：直面它、呼唤它、摆布它。

意识到内核并不像乍看起来那样使人畏惧，这就是良好的开端。不过，要梦想成真，必须全身心地投入，阅读源码、剖析源码，并毫不气馁。

希望第1章的介绍和本章的基础知识为贯穿本书其余部分的学习打下良好的基础。在后续的章节中，我们将着眼于介绍内核各个子系统详细而具体的概念。

第③章

进程管理

进程是Unix操作系统最基本的抽象之一[⊖]。一个进程就是处于执行期的程序（目标码存放在某种存储介质上）。但进程并不仅仅局限于一段可执行程序代码（Unix称其为代码段（text section））。通常进程还要包含其他资源，像打开的文件、挂起的信号、内核内部数据、处理器状态、地址空间及一个或多个执行线程（thread of execution）、当然还包括用来存放全局变量的数据段等。 实际上，进程就是正在执行的程序代码的活标本。

执行线程，简称线程（thread），是在进程中活动的对象。每个线程都拥有一个独立的程序计数器、进程栈和一组进程寄存器。内核调度的对象是线程，而不是进程。在传统的Unix系统中，一个进程只包含一个线程，但现在的系统中，包含多个线程的多线程程序司空见惯。稍后你会看到，Linux系统的线程实现非常特别——它对线程和进程并不特别区分。对Linux而言，线程只不过是一种特殊的进程罢了。

在现代操作系统中，进程提供两种虚拟机制：虚拟处理器和虚拟内存。虽然实际上可能是许多进程正在分享一个处理器，但虚拟处理器给进程一种假象，让这些进程觉得自己在独享处理器。第4章将详细描述这种虚拟机制。而虚拟内存让进程在获取和使用内存时觉得自己拥有整个系统的所有内存资源。第11章将描述虚拟内存机制。有趣的是，注意在线程之间（译者注：这里是指包含在同一个进程中的线程）可以共享虚拟内存，但拥有各自的虚拟处理器。

程序本身并不是进程；进程是处于执行期的程序以及它所包含的资源的总称。实际上完全可能存在两个或多个不同的进程执行的是同一个程序。并且两个或两个以上并存的进程还可以共享许多诸如打开的文件、地址空间之类的资源。

无疑，进程在它被创建的时刻开始存活。在Linux系统中，这通常是调用fork()系统调用的结果，该系统调用通过复制一个现有进程来创建一个全新的进程。调用fork()的进程被称为父进程，新产生的进程被称为子进程。在该调用结束时，在返回点这个相同位置上，父进程恢复执行，子进程开始执行。fork() 系统调用从内核返回两次：一次回到父进程，另一此回到新诞生的子进程。

通常，创建新的进程都是为了立即执行新的、不同的程序，而接着调用exec*()这族函数就可以创建新的地址空间，并把新的程序载入。在现代Linux内核中，fork()实际上是由clone()系统调用实现的，后者将在后面讨论。

最终，程序通过exit()系统调用退出执行。这个函数会终结进程并将其占用的资源释放掉。父进程可以通过wait4()[⊜]系统调用查询子进程是否终结，这其实使得进程拥有了等待特定进程执行完

⊖ 另外一个基本抽象是文件。

⊜ 由内核负责实现wait4()系统调用。Linux系统通过C库通常要提供wait()、waitpid()、wait3()和wait4()函数。虽然有些细微的语意差别，但所有函数都返回关于终止进程的状态。

毕的能力。进程退出执行后被设置为僵死状态，直到它的父进程调用wait()或waitpid()为止。

进程的另一个名字是任务（task）。Linux内核通常把进程也叫做任务。在本书中，我会交替使用这两个术语，不过我所说的任务通常指的是从内核观点所看到的进程。

3.1 进程描述符及任务结构

内核把进程存放在叫做任务队列（task list[⊖]）的双向循环链表中。链表中的每一项都是类型为task_struct、称为进程描述符（process descriptor）的结构，该结构定义在<linux/sched.h>文件中。进程描述符中包含一个具体进程的所有信息。

task_struct相对较大，在32位机器上，它大约有1.7K字节。但如果考虑到该结构内包含了内核管理一个进程所需的所有信息，那么它的大小也算相当小了。进程描述符中包含的数据能完整的描述一个正在执行的程序：它打开的文件，进程的地址空间，挂起的信号，进程的状态，还有其他更多信息（参见图3-1）。

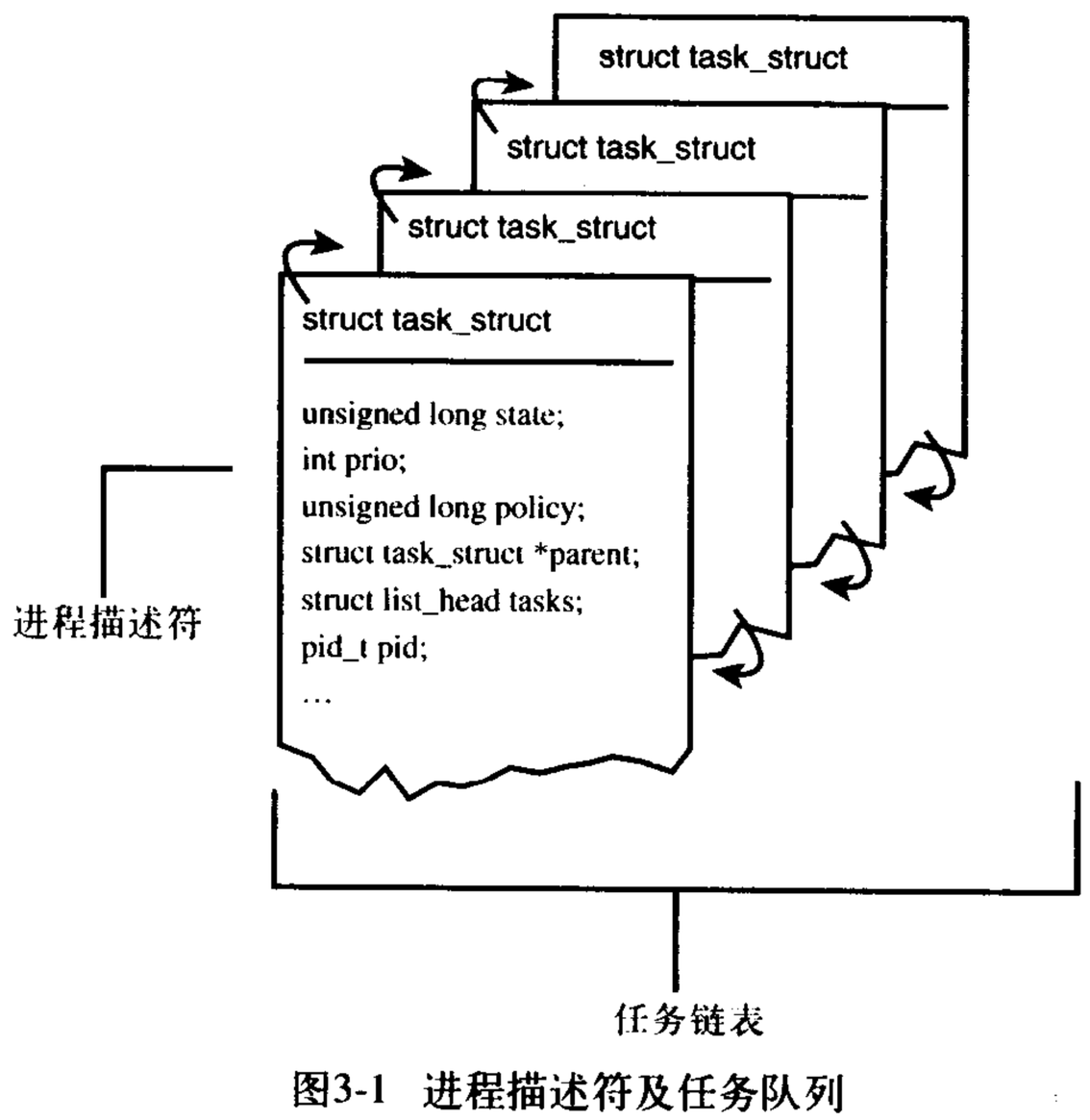


图3-1 进程描述符及任务队列

3.1.1 分配进程描述符

Linux通过slab分配器分配task_struct结构，这样能达到对象复用和缓存着色（cache coloring）（参见第11章）的目的（译者注：通过预先分配和重复使用task_struct，可以避免动态分配和释放所带来的资源消耗，还记得吗，第1章说过，Unix的一个特点就是进程创建迅速）。在2.6以前的内核中，各个进程的task_struct存放在它们内核栈的尾端。这样做是为了让那些像x86这样寄存器较

⊖ 有些操作系统教材称这为任务数组（task array）。由于Linux实现时使用的是链表而不是静态数组，所以就称作任务队列。

少的硬件体系结构只要通过栈指针就能计算出它的位置，从而避免使用额外的寄存器专门记录。由于现在用slab分配器动态生成task_struct，所以只需在栈底（对于向下增长的栈来说）或栈顶（对于向上增长的栈来说）[⊖]创建一个新的结构struct thread_info（参见图3-2）。这个新的结构能使在汇编代码中计算其偏移变得相当容易。

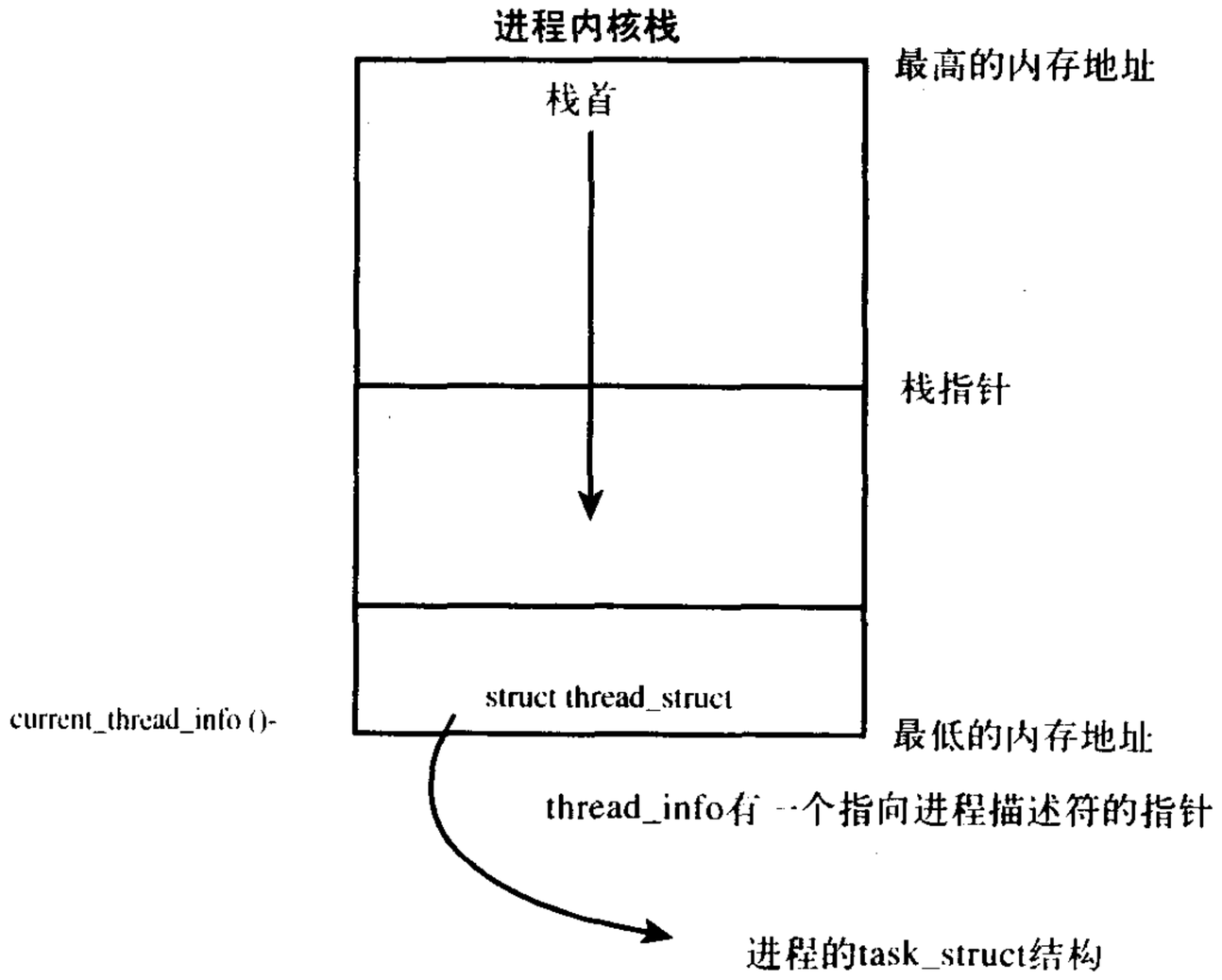


图3-2 进程描述符及内核栈

在x86上，thread_info结构在文件<asm/thread_info.h>中定义如下：

```
struct thread_info {
        struct task_struct      *任务;
        struct exec_domain      *exec_domain;
        unsigned long           flags;
        unsigned long           status;
        __u32                   cpu;
        __s32                   preempt_count;
        mm_segment_t            addr_limit;
        struct restart_block    restart_block;
        unsigned long           previous_esp;
        __u8                    supervisor_stack[0];
};
```

每个任务的thread_info结构在它的内核栈的尾端分配。结构中task域中存放的是指向该任务实际task_struct的指针。

3.1.2 进程描述符的存放

内核通过一个惟一的进程标识值（process identification value）或PID来标识每个进程。PID是

⊖ 寄存器较弱的体系结构不是引入thread_info结构的惟一原因。这个新建的结构能使在汇编代码中计算其偏移变得非常容易。

一个数，表示为pid_t隐含类型[⊖]，实际上就是一个int类型。为了与老版本的Unix和Linux兼容，PID的最大值默认设置为32768（short int短整型的最大值），尽管这个值也可以增加到类型所允许的范围。内核把每个进程的PID存放在它们各自的进程描述符中。

这个最大值很重要，因为它实际上就是系统中允许同时存在的进程的最大数目。尽管32768对于一般的桌面系统足够用了，但是大型服务器可能需要更多进程。这个值越小，转一圈就越快，本来数值大的进程比数值小的进程迟运行，但这样一来就破坏了这一原则。如果确实需要的话，可以不考虑与老式系统的兼容，由系统管理员通过修改/proc/sys/kernel/pid_max来提高上限。

在内核中，访问任务通常需要获得指向其task_struct指针。实际上，内核中大部分处理进程的代码都是直接通过task_struct进行的。因此，通过current宏查找到当前正在运行进程的进程描述符的速度就显得尤为重要。硬件体系结构不同，该宏的实现也不同，它必须针对专门的硬件体系结构做处理。有的硬件体系结构可以拿出一个专门寄存器来存放指向当前进程task_struct的指针，用于加快访问速度。而有些像x86这样的体系结构（其寄存器并不富余），就只能在内核栈的尾端创建thread_info结构，通过计算偏移间接地查找task_struct结构。

在x86系统上，current把栈指针的后13个有效位屏蔽掉，用来计算出thread_info的偏移。该操作通过current_thread_info()函数完成的。汇编代码如下：

```
movl $-8192, %eax
andl %esp, %eax
```

这里假定栈的大小为8KB。当4KB的栈启用时，就要用4096，而不是8192。

最后，current再从thread_info的task域中提取并返回task_struct的地址：

```
current_thread_info()->task;
```

对比一下这部分在PowerPC上的实现（IBM基于RISC的现代微处理器），我们可以发现当前task_struct的地址是保存在一个寄存器中的。也就是说，在PPC上，current宏只需把r2寄存器中的值返回就行了。与x86不一样，PPC有足够多的寄存器，所以它的实现有这样选择的余地。而访问进程描述符是一个重要的频繁操作，所以PPC的内核开发者觉得完全有必要为此使用一个专门的寄存器。

3.1.3 进程状态

进程描述符中的state域描述了进程的当前状态（参见图3-3）。系统中的每个进程都必然处于五种进程状态中的一种。该域的值也必为下列五种状态标志之一：

- TASK_RUNNING（运行）——进程是可执行的；它或者正在执行，或者在运行队列中等待执行（运行队列将会在第4章中讨论）。这是进程在用户空间中执行惟一可能的状态；也可以应用到内核空间中正在执行的进程。
- TASK_INTERRUPTIBLE（可中断）——进程正在睡眠（也就是说它被阻塞），等待某些条件的达成。一旦这些条件达成，内核就会把进程状态设置为运行。处于此状态的进程也会因为接收到信号而提前被唤醒并投入运行。
- TASK_UNINTERRUPTIBLE（不可中断）——除了不会因为接收到信号而被唤醒从而投入运行外，这个状态与可打断状态相同。这个状态通常在进程必须在等待时不受干扰或等待事件

⊖ 隐含类型指数据类型的物理表示是未知的或不相关的。

很快就会发生时出现。由于处于此状态的任务对信号不作响应，所以较之可中断状态[⊖]，使用得较少。

- TASK_ZOMBIE（僵死）——该进程已经结束了，但是其父进程还没有调用wait4()系统调用。为了父进程能够获知它的消息，子进程的进程描述符仍然被保留着。一旦父进程调用了wait4()，进程描述符就会被释放。
- TASK_STOPPED（停止）——进程停止执行；进程没有投入运行也不能投入运行。通常这种状态发生在接收到SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU等信号的时候。此外，在调试期间接收到任何信号，都会使进程进入这种状态。

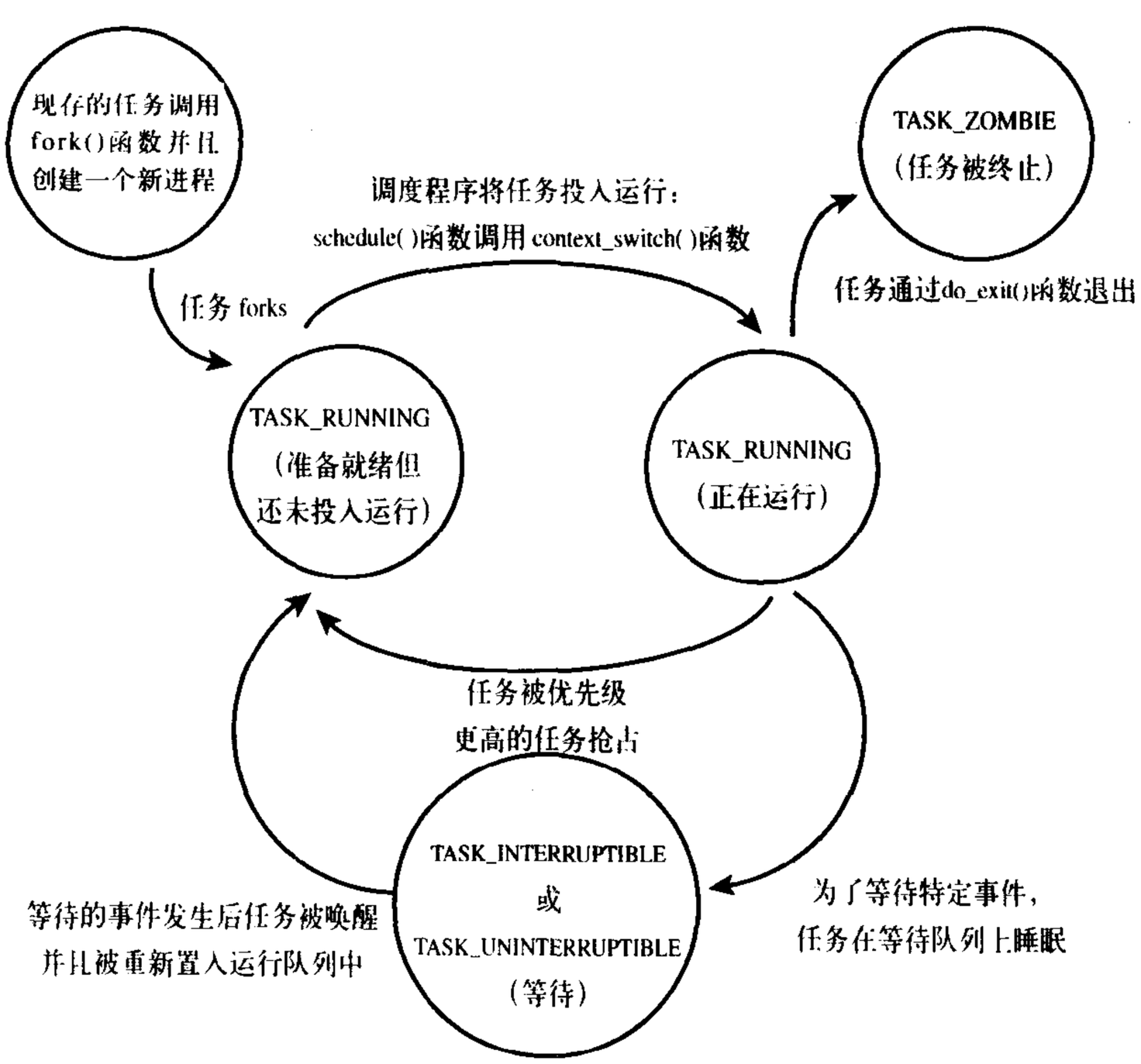


图3-3 进程状态转化

3.1.4 设置当前进程状态

内核经常需要调整某个进程的状态。这时最好使用set_task_state(task，state)函数：

```
set_task_state(task, state);/* 将任务 'task' 的状态设置为 'state' */
```

该函数将指定的进程设置为指定的状态。必要的时候，它会设置内存屏障来强制其他处理器

⊖ 这就是你在执行ps(1)命令时看到那些被标为D状态而又不能被杀死的进程的原因。由于任务对信号不响应，因此没法发送SIGKILL信号给它。进一步说，即使有办法，终结这样一个任务也不是明智的选择，该任务很可能正处于一个重要操作的执行期甚至可能还持有一个信号量。

作重新排序（一般只有在SMP系统中有此必要），否则，它等价于：

```
task->state = state;
```

方法set_current_state(state)和set_task_state(current，state)含义是等同的。

3.1.5 进程上下文

可执行程序代码是进程的重要组成部分。这些代码从可执行文件载入到进程的地址空间执行。一般程序在用户空间执行。当一个程序调执行了系统调用（参见第5章）或者触发了某个异常，它就陷入了内核空间。此时，我们称内核“代表进程执行”并处于进程上下文中。在此上下文中current宏是有效的[⊖]。除非在此间隙有更高优先级的进程需要执行并由调度器做出了相应调整，否则在内核退出的时候，程序恢复在用户空间继续执行。

系统调用和异常处理程序是对内核明确定义的接口。进程只有通过这些接口才能陷入内核执行——对内核的所有访问都必须通过这些接口。

3.1.6 进程家族树

Unix系统的进程之间存在一个明显的继承关系，在Linux系统中也是如此。所有的进程都是PID为1的init进程的后代。内核在系统启动的最后阶段启动init进程。该进程读取系统的初始化脚本（initscript）并执行其他的相关程序，最终完成系统启动的整个过程。

系统中的每个进程必有一个父进程。相应的，每个进程也可以拥有零个或多个子进程。拥有同一个父进程的所有进程被称为*兄弟*。进程间的关系存放在进程描述符中。每个task_struct都包含一个指向其父进程tast_struct、叫做parent的指针，还包含一个称为children的子进程链表。所以，对于当前进程，可以通过下面的代码获得其父进程的进程描述符：

```
struct task_struct *my_parent = current->parent;
```

同样，也可以按以下方式依次访问子进程：

```
struct task_struct *task;
struct list_head *list;

list_for_each(list, &current->children) {
        task = list_entry(list, struct task_struct, sibling);
        /* task 现在指向当前的某个子进程 */
}
```

init进程的进程描述符是作为init_task静态分配的。下面的代码可以很好地演示所有进程之间的关系：

```
struct task_struct *task;

for (task = current; task != &init_task; task = task->parent)
        ;
/* task 现在指向init  */
```

⊖ 除了进程上下文，还有将在第6章讨论的中断上下文（interrupt context）。在中断上下文中，系统不代表进程执行，而是执行一个中断处理程序。不会有进程去干扰这些中断处理程序，所以此时不会存在进程上下文。

实际上，可以通过这种继承体系从系统的任何一个进程出发查找到任意指定的其他进程。但大多数时候，只需要通过简单的重复方式就可以遍历系统中的所有进程。这非常容易做到，因为任务队列本来就是一个双向的循环链表。对于给定的进程，获取链表中的下一个进程：

```
list_entry(task->tasks.next, struct task_struct, tasks)
```

获取前一个进程的方法相同：

```
list_entry(task->tasks.prev, struct task_struct, tasks)
```

这两个例程分别通过next_task(task)宏和prev_task(task)宏实现。而实际上，for_each_process(task)宏提供了依次访问整个任务队列的能力。每次访问，任务指针都指向链表中的下一个元素：

```
struct task_struct *task;

for_each_process(task) {
        /* 它打印出每一个任务的名称和PID*/
        printk("%s[%d]\n", task->comm, task->pid);
}
```

需要注意的是，在一个拥有大量进程的系统中通过重复来遍历所有的进程是非常耗费时的。因此，如果没有充足的理由（或者别无他法）别这样做。

3.2 进程创建

Unix的进程创建很特别。许多其他的操作系统都提供了产生（spawn）进程的机制，首先在新的地址空间里创建进程，读入可执行文件，最后开始执行。Unix采用了与众不同的实现方式，它把上述步骤分解到两个单独的函数中去执行：fork()和exec()[⊖]。首先，fork()通过拷贝当前进程创建一个子进程。子进程与父进程的区别仅仅在于PID（每个进程惟一）、PPID（父进程的进程号，子进程将其设置为被拷贝进程的PID）和某些资源和统计量（例如挂起的信号，它没有必要被继承）。exec()函数负责读取可执行文件并将其载入地址空间开始运行。把这个两个函数组合起来使用的效果跟其他系统使用的单一函数的效果相似。

3.2.1 写时拷贝

传统的fork()系统调用直接把所有的资源复制给新创建的进程。这种实现过于简单并且效率低下，因为它拷贝的数据也许并不共享，更糟的情况是，如果新进程打算立即执行一个新的映像，那么所有的拷贝都将前功尽弃。Linux的fork()使用写时拷贝（copy-on-write）页实现。写时拷贝是一种可以推迟甚至免除拷贝数据的技术。内核此时并不复制整个进程地址空间，而是让父进程和子进程共享同一个拷贝。只有在需要写入的时候，数据才会被复制，从而使各个进程拥有各自的拷贝。也就是说，资源的复制只有在需要写入的时候才进行，在此之前，只是以只读方式共享。这种技术使地址空间上的页的拷贝被推迟到实际发生写入的时候。在页根本不会被写入的情况下——举例来说，fork()后立即调用exec()——它们就无需复制了。fork()的实际开销就是复制父进

⊖ Exec在这里指所有exec()一族的函数。内核系统实现了execve()、execlp()、execle()、execv()、execvp()等各种此类系统调用。

程的页表以及给子进程创建惟一的进程描述符。在一般情况下，进程创建后都会马上运行一个可执行的文件，这种优化可以避免拷贝大量根本就不会被使用的数据（地址空间里常常包含数十兆的数据）。由于Unix强调进程快速执行的能力，所以这个优化是很重要的。

3.2.2 fork()

Linux通过clone()系统调用实现fork()。这个调用通过一系列的参数标志来指明父、子进程需要共享的资源（关于这些标志更多的信息请参考本章后面3.3节）。fork()、vfork()和__clone ()库函数都根据各自需要的参数标志去调用clone()。然后由clone()去调用do_fork()。

do_fork完成了创建中的大部分工作，它的定义在kernel/fork.c文件中。该函数调用copy_process()函数，然后让进程开始运行。copy_process()函数完成的工作很有意思：

- 调用dup_task_struct()为新进程创建一个内核栈、thread_info结构和task_struct，这些值与当前进程的值相同。此时，子进程和父进程的描述符是完全相同的。
- 检查新创建的这个子进程后，当前用户所拥有的进程数目没有超出给他分配的资源的限制。
- 现在，子进程着手使自己与父进程区别开来。进程描述符内的许多成员都要被清0或设为初始值。进程描述符的成员值并不是继承而来的，而主要是统计信息。进程描述符中的大多数数据都是共享的。
- 接下来，子进程的状态被设置为TASK_UNINTERRUPTIBLE以保证它不会投入运行。
- copy_process()调用copy_flags()以更新task_struct的flags成员。表明进程是否拥有超级用户权限的PF_SUPERPRIV标志被清0。表明进程还没有调用exec()函数的PF_FORKNOEXEC标志被设置。
- 调用get_pid()为新进程获取一个有效的PID。
- 根据传递给clone()的参数标志，copy_process()拷贝或共享打开的文件、文件系统信息、信号处理函数、进程地址空间和命名空间等。在一般情况下，这些资源会被给定进程的所有线程共享；否则，这些资源对每个进程是不同的，因此被拷贝到这里。
- 让父进程和子进程平分剩余的时间片（第4章将作具体讨论）。
- 最后，copy_process()作扫尾工作并返回一个指向子进程的指针。

再回到do_fork()函数，如果copy_process()函数成功返回，新创建的子进程被唤醒并让其投入运行。内核有意选择子进程首先执行[⊖]。因为一般子进程都会马上调用exec()函数，这样可以避免写时拷贝的额外开销，如果父进程首先执行的话，有可能会开始向地址空间写入。

3.2.3 vfork()

vfork()系统调用和fork()的功能相同，除了不拷贝父进程的页表项。子进程作为父进程的一个单独的线程在它的地址空间里运行，父进程被阻塞，直到子进程退出或执行exec()。子进程不能向地址空间写入。在过去3BSD时期，这个优化是很有意义的，那时并未使用写时拷贝页来实现fork()。现在由于在执行fork()时引入了写时拷贝页并且明确了子进程先执行，vfork的好处就仅限于不拷贝父进程的页表项了。如果Linux将来fork()有了写时拷贝页表项，那么vfork()就彻底没用

⊖ 有趣的是，虽然想让子进程先运行，但是并非总能如此。

了[⊖]。另外由于vfork语意非常微妙（试想，如果exec()调用失败会发生什么？），所以最好让它逐渐淡出。完全可以把vfork()实现成一个普普通通的fork()——实际上，Linux2.2以前都是这么做的。

vfork()系统调用的实现是通过向clone()系统调用传递一个特殊标志来进行的。

- 在调用copy_process()时，task_struct的vfork_done成员被设置为NULL。
- 在执行do_fork()时，如果给定特别标志，则vfork_done会指向一个特殊地址。
- 子进程开始执行后，父进程不是马上恢复执行，而是一直等待，直到子进程通过vfork_done指针向它发送信号。
- 在调用mm_release()时，该函数用于进程退出内存地址空间，并且检查vfork_done是否为空，如果不为空，则会向父进程发送信号。
- 回到do_fork()，父进程醒来并返回。

如果一切执行顺利，子进程在新的地址空间里运行而父进程也恢复了在原地址空间的运行。这样的实现，开销确实降低了，不过它的设计并不是优良的。

3.3 线程在Linux中的实现

线程机制是现代编程技术中常用的一种抽象。该机制提供了在同一程序内共享内存地址空间运行的一组线程。这些线程还可以共享打开的文件和其他资源。线程机制支持*并发程序设计技术*（concurrent programming），在多处理器系统上，它也能保证真正的*并行处理*（parallelism）。

Linux实现线程的机制非常独特。从内核的角度来说，它并没有线程这个概念。Linux把所有的线程都当作进程来实现。内核并没有准备特别的调度算法或是定义特别的数据结构来表征线程。相反，线程仅仅被视为一个与其他进程共享某些资源的进程。每个线程都拥有惟一隶属于自己的task_struct，所以在内核中，它看起来就像是一个普通的进程（只是该进程和其他一些进程共享某些资源，如地址空间）。

上述线程机制的实现与Microsoft Windows或是Sun Solaris等操作系统的实现差异非常大。这些系统都在内核中提供了专门支持线程的机制（这些系统常常把线程称作*轻量级进程*，lightweight process）。“轻量级进程”这种叫法本身就概括了Linux在此处与其他系统的差异。在其他的系统中，相较于重量级的进程，线程被抽象成一种耗费较少资源，运行迅速的执行单元。而对于Linux来说，它只是一种进程间共享资源的手段（Linux的进程本身就够轻了[⊜]）。举个例子来说，假如我们有一个包含四个线程的进程，在提供专门线程支持的系统中，通常会有一个包含指向四个不同线程的指针的进程描述符。该描述符负责描述像地址空间、打开的文件这样的共享资源。线程本身再去描述它独占的资源。相反，Linux仅仅创建四个进程并分配四个普通的task_sturct结构。建立这四个进程时指定它们共享某些资源就行了。

线程的创建和普通进程的创建类似，只不过在调用clone()的时候需要传递一些参数标志来指明需要共享的资源：

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

上面的代码产生的结果和调用fork()差不多，只是父子俩共享地址空间、文件系统资源、文件

⊖ 其实目前已有补丁可以帮助Linux完成该功能了。

⊜ 作为一个例子，创建Linux进程所花时间和创建其他操作系统进程（尤其是线程）所花时间的比较测评结果非常好。

描述符和信号处理程序。换个说法就是，新建的进程和它的父进程就是流行的所谓线程。

对比一下，一个普通的fork()的实现是：

```
clone(SIGCHLD, 0);
```

而vfork()的实现是：

```
clone(CLONE_VFORK | CLONE_VM | SIGCHLD, 0);
```

传递给clone()的参数标志决定了新创建进程的行为方式和父子进程之间共享的资源种类。表3-1列举了这些clone()用到的参数标志以及它们的作用，这些是在<linux/sched.h>中定义的。

表3-1 clone()参数标志

参数标志	含义
CLONE_FILES	父子进程共享打开的文件
CLONE_FS	父子进程共享文件系统信息
CLONE_IDLETASK	将PID设置为0（只供idle进程使用）
CLONE_NEWNS	为子进程创建新的命名空间
CLONE_PARENT	指定子进程与父进程拥有同一个父进程
CLONE_PTRACE	继续调试子进程
CLONE_SETTID	将TID回写至用户空间
CLONE_SETTLS	为子进程创建新的TLS
CLONE_SIGHAND	父子进程共享信号处理函数
CLONE_SYSVSEM	父子进程共享System V SEM_UNDO语义
CLONE_THREAD	父子进程放入相同的线程组
CLONE_VFORK	调用vfork()，所以父进程准备睡眠等待子进程将其唤醒
CLONE_UNTRACED	防止跟踪进程在子进程上强制执行CLONE_PTRACE
CLONE_STOP	以TASK_STOPPED状态开始进程
CLONE_SETTLS	为子进程创建新的TLS(thread-local storage)
CLONE_CHILD_CLEARTID	清除子进程的TID
CLONE_CHILD_SETTID	设置子进程的TID
CLONE_PARENT_SETTID	设置父进程的TID
CLONE_VM	父子进程共享地址空间

内核线程

内核经常需要在后台执行一些操作。这种任务可以通过内核线程（kernel thread）完成——独立运行在内核空间的标准进程。内核线程和普通的进程间的区别在于内核线程没有独立的地址空间（实际上它的mm指针被设置为NULL）。它们只在内核空间运行，从来不切换到用户空间去。内核进程和普通进程一样，可以被调度，也可以被抢占。

Linux确实会把一些任务交给内核线程去做，像pdflush和ksoftirqd这些任务就是明显的例子。这些线程在系统启动时由另外一些内核线程启动。实际上，内核线程也只能由其他内核线程创建。在现有内核线程中创建一个新的内核线程的方法如下：

```
int kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
```

新的任务也是通过向普通的clone()系统调用传递特定的flags参数而创建的。在上面的函数返回时，父线程退出，并返回一个指向子线程task_struct的指针。子线程开始运行fn指向的函数，arg

是运行时需要用到的参数。一个特殊的clone标志CLONE_KERNEL定义了内核线程常用到的参数标志：CLONE_FS、CLONE_FILES、CLONE_SIGHAND。大部分的内核线程把这个标志传递给它们的flags参数。

一般情况下，内核线程会将它在创建时得到的函数永远执行下去（除非系统重启，但要知道你用的是Linux，谁知道你有没有机会等到那么一天）。该函数通常由一个循环构成，在需要的时候，这个内核线程就会被唤醒和执行，完成了当前任务，它会自行休眠。

我们将在以后的章节中详细讨论具体的内核进程。

3.4 进程终结

虽然让人伤感，但进程终归是要终结的。当一个进程终结时，内核必须释放它所占有的资源并把这一不幸告知其父进程。

一般说来，进程的析构发生在它调用exit()之后，既可能显式地调用这个系统调用，也可能隐式地从某个程序的主函数返回（其实C语言编译器会在main()函数的返回点后面放置调用exit()的代码）。当进程接受到它既不能处理也不能忽略的信号或异常时，它还可能被动地终结。不管进程是怎么终结的，该任务大部分都要靠do_exit()来完成，它要做下面这些繁琐的工作：

- 首先，将task_struct中的标志成员设置为PF_EXITING。
- 其次，调用del_timer_sync()删除任一内核定时器。根据返回的结果，它确保没有定时器在排队，也没有定时器处理程序在运行。
- 如果BSD的进程计账功能是开启的，do_exit()调用acct_process()来输出计账信息。
- 然后调用_exit_mm()函数放弃进程占用的mm_struct，如果没有别的进程使用它们（也就是说，它们没被共享），就彻底释放它们。
- 接下来调用exit__sem()函数。如果进程排队等候IPC信号，它则离开队列。
- 调用_exit_files()、_exit_fs()、exit_namespace()和exit_sighand()，以分别递减文件描述符、文件系统数据，进程名字空间和信号处理函数的引用计数。如果其中某些引用计数的数值降为零，那么就代表没有进程在使用相应的资源，此时可以释放。
- 接着把存放在task_struct的exit_code成员中的任务退出代码置为exit()提供的代码中，或者去完成任何其他由内核机制规定的退出动作。退出代码存放在这里供父进程随时检索。
- 调用exit_notify()向父进程发送信号，将子进程的父进程重新设置为线程组中的其他线程或init进程，并把进程状态设成TASK_ZOMBIE。
- 最后，do_exit()调用schedule()切换到其他进程（参看第4章）。因为处于TASK_ZOMBIE状态的进程不会再被调度，所以这是进程所执行的最后一段代码。

do_exit()的实现在kernel/exit.c文件中可以找到。

至此，与进程相关联的所有资源都被释放掉了（假设进程是这些资源的惟一使用者）。进程不可运行（实际上也没有地址空间让它运行）并处于TASK_ZOMBIE状态。它占用的所有资源就是内核栈、thread_info结构和tast_struct结构。此时进程存在的惟一目的就是向它的父进程提供信息。父进程检索到信息后，或者通知内核那是无关的信息后，由进程所持有的剩余内存被释放，归还给系统使用。

3.4.1 删除进程描述符

在调用了do_exit()之后，尽管线程已经僵死不能再运行了，但是系统还保留了它的进程描述符。前面说过，这样做可以让系统有办法在子进程终结后仍能获得它的信息。因此，进程终结时所需的清理工作和进程描述符的删除被分开执行。在父进程获得已终结的子进程的信息后，或者通知内核它并不关注那些信息后，子进程的task_struct结构才被释放。

wait()这一族函数都是通过惟一（但是很复杂）的一个系统调用wait4()实现的。它的标准动作是挂起调用它的进程，直到其中的一个子进程退出，此时函数会返回该子进程的PID。此外，调用该函数时提供的指针会包含子函数退出时的退出代码。

当最终需要释放进程描述符时，release_task()会被调用，用以完成以下工作：

- 首先，它调用free_uid()来减少该进程拥有者的进程使用计数。Linux用一个单用户高速缓存统计和记录每个用户占用的进程数目、文件数目。如果这些数目都将为0，表明这个用户没有使用任何进程和文件，那么这块缓存就可以销毁了。
- 然后，release_task()调用unhash_process()从pidhash上删除该进程，同时也要从task_list中删除该进程。
- 接下来，如果这个进程正在被ptrace跟踪，release_task()将跟踪进程的父进程重设为其最初的父进程并将它从ptract list上删除。
- 最后，release_task()调用put_task_struct()释放进程内核栈和thread_info结构所占的页，并释放task_struct所占的slab高速缓存。

至此，进程描述符和所有进程独享的资源就全部释放掉了。

3.4.2 孤儿进程造成的进退维谷

如果父进程在子进程之前退出，必须有机制来保证子进程能找到一个新的父亲，否则的话这些成为孤儿的进程就会在退出时永远处于僵死状态，白白的耗费内存。前面的部分已经有所暗示，对于这个问题，解决方法是给子进程在当前线程组内找一个线程作为父亲，如果不行，就让init做它们的父进程。在do_exit()中会调用notify_parent()，该函数会通过forget_original_parent()来执行寻父过程：

```
struct task_struct *p, *reaper = father;
struct list_head *list;

if (father->exit_signal != -1)
        reaper = prev_thread(reaper);
else
        reaper = child_reaper;

if (reaper == father)
        reaper = child_reaper;
```

这段代码将reaper设置为该进程所在的线程组内的其他进程。如果线程组内没有其他的进程，它就将reaper设置为child_reaper，也就是init进程。现在，合适的父进程也经找到了，只需要遍历所有子进程并为它们设置新的父进程：

```
list_for_each(list, &father->children) {
        p = list_entry(list, struct task_struct, sibling);
        reparent_thread(p, reaper, child_reaper);
}
list_for_each(list, &father->ptrace_children) {
        p = list_entry(list, struct task_struct, ptrace_list);
        reparent_thread(p, reaper, child_reaper);
}
```

这段代码遍历了两个链表：子进程链表和ptrace子进程链表，给每个子进程设置新的父进程。这两个链表同时存在的原因很有意思，它也是2.6内核的一个新特性。当一个进程被跟踪时，它被暂时设定为调试进程的子进程。此时如果它的父进程退出了，系统会为它和它的所有兄弟重新找一个父进程。在以前的内核中，这就需要遍历系统所有的进程来找这些子进程。现在的解决办法是在一个单独的被ptrace跟踪的子进程链表中搜索相关的兄弟进程——用两个相关链表减轻了遍历带来的消耗。

一旦系统给进程成功的找到和设置了新的父进程，就不会再有出现驻留僵死进程的危险了。init进程会例行调用wait()来等待其子进程，清除所有与其相关的僵死进程。

3.5 进程小结

在本章中，我们考察了操作系统中闻名遐尔的概念——进程。我们还讨论了进程的一般特性，它为何如此重要，以及进程与线程之间的关系。然后，讨论了Linux如何存放和表示进程（用task_struct 和 thread_info），如何创建进程（通过clone() 和 fork()），如何把新的执行映像装入到地址空间（通过exec()系统调用族），如何表示进程的层次关系，父进程又是如何收集其后代的信息（通过wait()系统调用族），以及进程最终如何死亡（强制或自愿地调用exit()）。

进程是最基本、最重要的一种抽象，位于每个现代操作系统的核心位置，也是最终导致我们拥有操作系统的根源（通过执行程序）。

下一章讨论进程调度，那是微妙而有趣的方式，内核以这种方式决定哪个进程运行，何时运行，以何种顺序运行。

第④章
进程调度

前一章讨论了进程，那是运行程序的操作系统抽象。本章讨论进程调度，这使进程有效工作的一组代码。

调度程序是内核的组成部分，它负责选择下一个要运行的进程。进程调度程序（有时也简称调度程序）可看作在可运行态进程之间分配有限的处理器时间资源的内核子系统。调度程序是诸如Linux这样的多任务操作系统的基础。只有通过调度程序的合理调度，系统资源才能最大限度地发挥作用，多进程才会有并发执行的效果。

调度程序没有太复杂的原理。最大限度地利用处理器时间的原则是，只要有可以执行的进程，那么就总会有进程正在执行。但是只要系统中进程的数目比处理器的个数多，就注定某一给定时刻会有一些进程不能执行。这些进程在等待运行。在一组处于可运行状态的进程中选择一个来执行，是调度程序所需完成的基本工作。

多任务操作系统就是能同时并发地交互执行多个进程的操作系统。在单处理机上，这会产生多个进程在同时运行的幻觉。在多处理机上，这会使多个进程在不同的处理机上真正同时、并行地运行。在任一台机子上，这也能使多个进程在后台运行，也就是说，除非需要，否则并不实际执行。这些任务尽管位于内存，但并不处于可运行状态。相反，这些进程利用内核阻塞自己，直到某一事件（键盘输入、网络数据、过一段时间等等）发生。因此，现代Linux系统也许有100个进程在内存，但是只有一个处于可运行状态。

多任务系统可以划分为两类：非抢占式多任务（cooperative multitasking）和抢占式多任务（preemptive multitasking）。像所有Unix的变体和许多其他现代操作系统一样，Linux提供了抢占式的多任务模式。在此模式下，由调度程序来决定什么时候停止一个进程的运行以便其他进程能够得到执行机会。这个强制的挂起动作就叫做抢占（preemption）。进程在被抢占之前能够运行的时间是预先设置好的，而且有一个专门的名字，叫进程的时间片（timeslice）。时间片实际上就是分配给每个可运行进程的处理器时间段。有效管理时间片能使调度程序从系统全局的角度做出调度决定，这样做还可以避免个别进程独占系统资源。我们将会看到，Linux进程调度程序采用动态方法计算时间片，这样带来了许多好处。

相反，在非抢占式多任务模式下，除非进程自己主动停止运行，否则它会一直执行。进程主动挂起自己的操作称为让步（yielding）。这种机制有很多缺点：调度程序无法对每个进程该执行多长时间做出统一规定，所以进程独占的处理器时间可能超出用户的预料，更糟的是，一个决不做出让步的悬挂进程就能使系统崩溃。幸运的是，10年以来，绝大部分的操作系统的设计都采用了抢占式多任务——除了Mac OS 9以及它的前辈们这些出名且麻烦的异端以外。毫无疑问，Unix从一开始就采用的是抢先式的多任务。

在Linux 2.5开发系列的内核中，调度程序做了大手术。开始采用了一种叫做O(1)[⊖]调度程序的新调度程序——它是因为其算法的行为而得名的。它解决了先前版本Linux调度程序的许多不足，引入了许多强大的新特性和性能特征。本章将介绍调度程序的设计原理以及它们具体是怎么通过O(1)调度程序体现的，内容包括调度程序的目标、设计、实现、算法以及相关的系统调用。

4.1 策略

策略决定调度程序在何时让什么进程运行。调度器的策略往往决定系统的整体印象，并且，还要负责优化使用处理器时间。无论从哪个方面来看，它都是至关重要的。

4.1.1 I/O消耗型和处理器消耗型的进程

进程可以被分为I/O消耗型和处理器消耗型。前者指进程的大部分时间用来提交I/O请求或是等待I/O请求。因此，这样的进程经常处于可运行状态，但通常都是运行短短的一会儿，因为它在等待更多的I/O请求时最后总会阻塞（这里指的是所有的I/O操作，像键盘活动等都包括，并不仅仅局限于磁盘I/O）。

相反，处理器耗费型进程把时间大多用在执行代码上。除非被抢占，否则它们通常都一直不停地运行，因为它们没有太多的I/O需求。但是，因为它们不属于I/O驱动类型，所以从系统响应速度考虑，调度器不应该经常让它们运行。对于这类处理器消耗型的进程，调度策略是尽量降低它们的运行频率，对它们而言，延长其运行时间会更合适些。处理器消耗型进程的极端例子就是无限循环地执行。

当然，这种划分方法并非是绝对的。进程可以同时展示这两种行为：比如，X Windows服务器既是I/O消耗型，也是处理器消耗型。还有些进程可能是I/O消耗型，但属于处理器消耗型活动的范围。其典型的例子就是字处理器，通常坐以等待键盘输入，但不知何时又可能会粘住处理器疯狂地进行拼写检查。

调度策略通常要在两个矛盾的目标中间寻找平衡：进程响应迅速（响应时间短）和最大系统利用率（高吞吐量）。为了满足上述需求，调度程序通常采用一套非常复杂的算法来决定最值得运行的进程投入运行，但是它往往并不保证低优先级进程会被公平对待。因为交互式程序都是I/O消耗型的，所以调度程序向这种类型的进程倾斜会缩短系统响应时间。Linux为了保证交互式应用，所以对进程的响应作了优化（缩短响应时间），更倾向于优先调度I/O消耗型进程。虽然如此，但在下面你会看到，调度程序也并未忽略处理器消耗型的进程。

4.1.2 进程优先级

调度算法中最基本的一类就是基于优先级的调度。这是一种根据进程的价值和其对处理器时间的需求来对进程分级的想法。优先级高的进程先运行，低的后运行，相同优先级的进程按轮转方式进行调度（一个接一个，重复进行）。在包括Linux在内的某些系统中，优先级高的进程使用的时间片也较长。调度程序总是选择时间片未用尽而且优先级最高的进程运行。用户和系统都可

⊖ O(1)用的是大O表示法。简而言之，它是指不管输入有多大，调度程序都可以在恒定时间内完成工作。附录C，“算法复杂度”，是为那些好奇的人们准备的一份完整的大O表示法说明。

以通过设置进程的优先级来影响系统的调度。

Linux根据以上思想实现了一种基于动态优先级的调度方法。一开始，该方法先设置基本的优先级，然而它允许调度程序根据需要来加、减优先级。举个例子，如果一个进程在I/O等待上耗费的时间多于其运行时间，那么该进程明显属于I/O消耗型进程。它的优先级会被动态提高。作为一个反例，如果一个进程的全部时间片一下就被耗尽，那么该进程属于处理器消耗型进程——它的优先级会被动态地降低。

Linux内核提供了两组独立的优先级范围。第一种是nice值，范围从−20到+19，默认值是0。nice的值越大优先级越低——你正在为系统中其他进程做好事（being nice）。nice值小的进程（优先级高）在nice值大的进程（优先级低）之前执行。另外nice值也用来决定分配给进程的时间片的长短。nice值为−20的进程可能获得的时间片最长，nice值为19的进程获得的时间片可能最短。nice是所有Unix系统都用到的标准优先级范围。

第二个范围是实时优先级，其值是可配置的，默认情况下它的变化范围是从0到99。任何实时进程的优先级都高于普通的进程。Linux提供对POSIX实时优先级的支持。大部分现代的Unix操作系统也都提供类似的机制。

4.1.3 时间片

时间片[⊖]是一个数值，它表明进程在被抢占前所能持续运行的时间。调度策略必须规定一个默认的时间片，但这并不是件简单的事。时间片过长会导致系统对交互的响应表现欠佳；让人觉得系统无法并发执行应用程序。时间片太短会明显增大进程切换带来的处理器耗时，因为肯定会有相当一部分系统时间用在进程切换上，而这些进程能够用来运行的时间片却很短。此外，I/O消耗型和处理器消耗型的进程之间的矛盾在这里也再次显露出来：I/O消耗型不需要长的时间片，而处理器消耗型的进程则希望越长越好（比如说这样可以让它们的高速缓存命中率更高）。

从上面的争论中可以看出，任何长时间片都将导致系统交互表现欠佳。很多操作系统中都特别重视这一点，所以默认的时间片很短——如20毫秒。不过Linux利用了优先级最高的进程总是在运行的这一原则。

Linux调度程序提高交互式程序的优先级，让它们运行得更频繁。于是，调度程序提供较长的默认时间片（参看图4-1）给交互式程序。此外，Linux调度程序还能根据进程的优先级动态调整分配给它的时间片。从而保证了优先级高的进程，假定也是重要性高的进程，执行的频率高，执行时间长。通过实现这样一种动态调整优先级和时间片长度的机制，Linux调度性能不但非常稳定而且也很强健。

注意，进程并不是一定非要一次就用完它所有的时间片。举例来说，一个拥有100毫秒时间片的进程并不一定在一次运行中就要用完所有这些时间。相反，进程可以通过重复调度，分五次每次20毫秒用完这些时间片。这样，即使是交互式程序也能从中获益——当它们没必要一次用这么多时间的时候，它们可以分几次使用，这样能保证它们尽可能长时间的处于可运行状态。

⊖ 在其他系统中，时间片有时也称为量子（quantum）或处理器片（processor slice）。但Linux把它叫做时间片，因此你最好也这样叫。

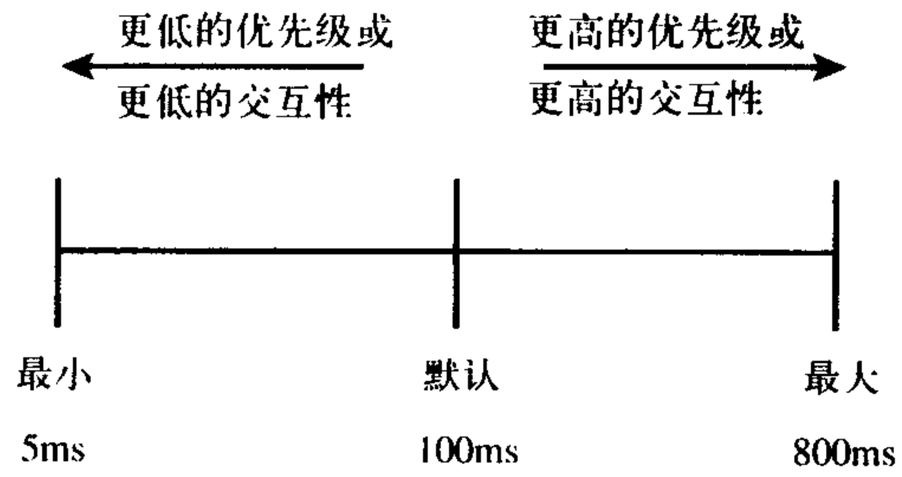


图4-1 进程时间片的计算

当一个进程的时间片耗尽时，就认为进程到期了。没有时间片的进程不会再投入运行，除非等到其他所有的进程都耗尽了它们的时间片（也就是说它们的剩余时间片为0）。在那个时候，所有进程的时间片会被重新计算。本章中，我们会讨论Linux进程调度程序处理时间片耗尽的一些有趣的算法。

4.1.4 进程抢占

像前面所说的，Linux系统是抢占式的。当一个进程进入TASK_RUNNING状态，内核会检查它的优先级是否高于当前正在执行的进程。如果是这样，调度程序会被唤醒，抢占当前正在运行的进程并运行新的可运行进程。此外，当一个进程的时间片变为0时，它会被抢占，调度程序被唤醒以选择一个新的进程。

4.1.5 调度策略的活动

想像下面这样一个系统，它拥有两个运行中的进程：一个文字编辑程序和一个视频编码程序。文字编辑程序显然是I/O消耗型的，因为它在大部分时间都在等待用户的键盘输入（无论用户的输入速度有多快，都不可能赶上处理的速度）。用户总是希望一按下键系统就能马上响应。相反，视频编码程序是处理器消耗型的。除了最开始从磁盘上读出原始数据流和最后把处理好的视频输出，程序所有的时间都用来对原始数据进行视频编码，处理器很轻易地100%被使用。它对什么时间开始运行没有太严格的要求——用户几乎分辨不出也并不关心它到底是立刻就运行还是半秒钟以后才开始的。当然，它完成得越早越好，至于所花时间并不是我们关注的主要问题。

在这样的例子中，理论上认为调度程序给文字编辑程序更高的优先级和更长的时间片，因为它是交互式的。这将保证文字编辑程序有充足的时间片可用。此外，由于拥有较高的优先级，所以文字编辑程序还能在需要的时候抢占视频编码程序——例如，用户敲键盘的瞬间。这样才能保证文字编辑程序对用户键盘输入的即时响应。这样做当然会影响视频编码程序，但由于文字编辑程序仅仅只在当用户按键时间断地运行，所以视频编码程序可以在所有剩余时间中独享处理器。这样的优化同时提高了这两种应用的性能。

4.2 Linux调度算法

在前一节，我们抽象地讨论了进程调度原理，只是偶尔提及Linux如何把给定的理论应用到实际。在已有的调度原理基础上，我们进一步探讨具有Linux特色的进程调度程序。

Linux的调度程序定义于kernel/sched.c中。调度程序的算法和相关支持代码大部分都在2.5版内核的开发版中被重写了。因此，现在的程序与以前版本内核中的调度程序区别很大，几乎是全新的。设计新的调度程序是为了实现下列目标：

- 充分实现O(1)调度。不管有多少进程，新调度程序采用的每个算法都能在恒定时间内完成。
- 全面实现SMP的可扩展性。每个处理器拥有自己的锁和自己的可执行队列。
- 强化SMP的亲和力。尽量将相关一组任务分配给一个CPU进行连续的执行。只有在需要平衡任务队列的大小时才在CPU之间移动进程。
- 加强交互性能。即使在系统处于相当负载的情况下，也能保证系统的响应，并立即调度交互式进程。
- 保证公平。在合理设定的时间范围内，没有进程会处于饥饿状态。同样的，也没有进程能够显失公平地得到大量时间片。
- 虽然最常见的优化情况是系统中只有1～2个可运行进程，但是优化也完全有能力扩展到具有多处理器且每个处理器上运行多个进程的系统中。

新的调度程序实现了上述目标。

4.2.1 可执行队列

调度程序中最基本的数据结构是运行队列（runqueue）。可执行队列定义于kernel/sched.c[⊖]中，由结构runqueue表示。可执行队列是给定处理器上的可执行进程的链表，每个处理器一个。每个可投入运行的进程都惟一的归属于一个可执行队列。此外，可执行队列中还包含每个处理器的调度信息。所以，可执行队列也是每个处理器最重要的数据结构。

让我们观察一下这个结构体，其中的注释对每项进行了描述：

```
struct runqueue {
spinlock_t              lock;                   /* 保护运行队列的自旋锁*/
unsigned long           nr_running;             /* 可运行任务数目 */
unsigned long           nr_switches;            /* 上下文切换数目*/
unsigned long           expired_timestamp;      /* 队列最后被换出时间 */
unsigned long           nr_uninterruptible;     /* 处于不可中断睡眠状态的任务数目*/
unsigned long long      timestamp_last_tick;    /* 最后一个调度程序的节拍 */
struct task_struct      *curr;                  /* 当前运行任务 */
struct task_struct      *idle;                  /* 该处理器的空任务*/
struct mm_struct        *prev_mm;               /* 最后运行任务的mm_struct结构体*/
struct prio_array       *active;                /* 活动优先级队列 */
struct prio_array       *expired;               /* 超时优先级队列 */
struct prio_array       arrays[2];              /* 实际优先级数组*/
struct task_struct      *migration_thread;      /* 移出线程 */
struct list_head        *migration_queue;       /* 移出队列*/
atomic_t                nr_iowait;              /* 等待I/O操作的任务数目 */
};
```

⊖ 为什么是 kernel/sched.c 而不是 <linux/sched.h>? 因为把调度程序的代码抽象出来而只给内核的其余部分提供某些接口是一种理想的做法。把运行队列的代码放在头文件中会使调度程序以外的代码也访问运行队列的代码，这不是一种理想的做法。

由于可执行队列是调度程序的核心数据结构体，所以有一组宏定义用于获取与给定处理器或进程相关的可执行队列。cpu_rq(processor)宏用于返回给定处理器可执行队列的指针。类似地，this_rq()宏用来返回当前处理器的可执行队列。最后，宏task_rq(task)返回给定任务所在的队列指针。

在对可执行队列进行操作以前，应该先锁住它（锁机制在第8章中讨论）。因为每个可执行队列惟一地对应一个处理器，所以很少出现一个处理器需要锁其他处理器的可执行队列的情况（我们将会看到，这种情况还确实可能会出现）。在其拥有者读取或改写队列成员的时候，可执行队列包含的锁用来防止队列被其他代码改动。锁住运行队列的最常见情况发生在你想锁住的运行队列上恰巧有一个特定的任务在运行。此时需要用到task_rq_lock()和task_rq_unlock()函数：

```
struct runqueue *rq;
unsigned long flags;

rq = task_rq_lock(task, &flags);
/* 对任务的队列rq进行操作 */
task_rq_unlock(rq, &flags);
```

或者可以用this_rq_lock()来锁住当前的可执行队列，用rq_unlock(struct runqueue*rq)释放给定队列上的锁。

```
struct runqueue *rq;

rq = this_rq_lock();
/* 操作这个进程的当前运行队列rq */
rq_unlock(rq);
```

为了避免死锁，要锁住多个运行队列的代码必须总是按照同样的顺序获取这些锁：按照可执行队列地址从低向高的顺序（同样的，也是在第8章给出详细解释）。如：

```
/* 锁定 ... */
if (rq1 == rq2)
        spinlock(&rq1->lock);
else{
      if (rq1 < rq2){
            spin_lock(&rq1->lock);
            spin_lock(&rq2->lock);
      } else {
            spin_lock(&rq2->lock);
            spin_lock(&rq1->lock);
      }
}
/* 操作两个运行队列 ... */

/* 释放锁... */
spin_unlock(&rq1->lock);
if (rq1 != rq2)
        spin_unlock(&rq2->lock);
```

这些步骤能通过double_rq_lock()和double_rq_unlock()自动完成。上面的步骤会被简化为：

```
double_rq_lock(rq1, rq2);

/*操作两个运行队列 ... */

double_rq_unlock(rq1, rq2);
```

让我们通过一个简单例子来看为什么按照这种顺序对锁进行操作是重要的。关于死锁的话题在第8章和第9章都会涉及——它不仅仅出现在可执行队列上；嵌套的锁必须以相同的顺序操作。自旋锁用于防止多个任务同时对可执行队列进行操作。它们起的作用就像门钥匙一样。最开始，有一个任务到了大门前，它拿起钥匙开了门，走进去以后回身把大门锁上。如果这时第二个任务到了大门前，它发现门锁着（已经有一个进程在里面了），就坐在门口等，直到第一个任务走出来交出钥匙。这个等待过程叫自旋，因为实际上任务是在不停地执行一个循环操作来查询钥匙是否被交出来了。现在想像一下，一号任务希望先锁住可执行队列甲，然后再去锁队列乙，而二号任务想先锁可执行队列乙，然后再去锁甲。如果在一号任务锁住甲的同时二号任务锁住了乙。现在，一号任务再去尝试着锁乙而二号任务去锁甲。两个都变成了不可完成的任务，因为它俩无论谁需要的那把锁，都掌握在另一个的手中。两个任务就这样一起永远的等待下去。就像相向而行的汽车僵持会造成交通堵塞一样，这种对锁的无序使用会导致任务之间互相等待，不死不休，也就是死锁。如果两个任务用相同的顺序操作锁，这种局面就不会产生了。关于锁的详细内幕，可以参看第8章和第9章。

4.2.2 优先级数组

每个运行队列都有两个优先级数组，一个活跃的和一个过期的。优先级数组在kernel/ sched.c中被定义，它是prio_array类型的结构体。优先级数组是一种能够提供O(1)级算法复杂度的数据结构。优先级数组使可运行处理器的每一种优先级都包含一个相应的队列，而这些队列包含对应优先级上的可执行进程链表。优先级数组还拥有一个优先级位图，当需要查找当前系统内拥有最高优先级的可执行进程时，它可以帮助提高效率。

```
struct prio_array {
        int                 nr_active;              /* 任务数目*/
        unsigned long       bitmap[BITMAP_SIZE];    /* 优先级位图*/
        struct list_head    queue[MAX_PRIO];        /* 优先级队列*/
};
```

MAX_PRIO定义了系统拥有的优先级个数。默认值是140。这样，每个优先级都有一个struct list_head结构体。BITMAP_SIZE是优先级位图数组的大小， 类型为unsigned long长整型，长32位，如果每一位代表一个优先级的话，140个优先级需要5个长整型数才能表示。所以，bitmap就正好含有5个数组项，总共160位。

每个优先级数组都要包含一个这样的位图成员，至少为每个优先级准备一位。一开始，所有的位都被置为0。当某个拥有一定优先级的进程开始准备执行时（也就是状态变为TASK_RUNNING），位图中相应的位就会被置为1。比如，如果一个优先级为7的进程变为可执行状态，第7位就被置为1。这样，查找系统中最高的优先级就变成了查找位图中被设置的第一个位。因为优先级个数是个定值，所以查找时间恒定，并不受系统到底有多少可执行进程的影响。此外，Linux对它支持的每一种体

系结构都提供了对应的快速查找算法，以保证对位图的快速查找。提供该功能的函数叫sched_find_first_bit()。很多体系结构提供了find-first-set指令，这条指令对指定的字操作[⊖]。在这些系统上，找到第一个要设置的位所花的时间至多是执行这条指令的两倍，可以说微不足道。

每个优先级数组还包含一个叫做struct list_head的队列，其中每个元素都是一个struct list_head类型的队列。每个链表与一个给定的优先级相对应，事实上，每个链表都包含该处理器队列上相应优先级的全部可运行进程。所以要找到下一个要运行的任务非常简单，就像在链表中选择下一个元素一样。对于给定的优先级，按轮转方式调度任务。

优先级数组还包含一个计数器nr_active。它保存了该优先级数组内可执行进程的数目。

4.2.3 重新计算时间片

许多操作系统（包括老版本的Linux）在所有进程的时间片都用完时，都采用一种显式的方法来重新计算每个进程的时间片。典型的实现是循环访问每个进程，像这样：

```
for (系统中的每个任务){
        重新计算优先级
        重新计算时间片
}
```

在决定新的时间片长短时会用到进程的优先级和其他一些属性。但这种实现存在一些弊端：

- 可能会耗费相当长的时间。最坏情况下，有n个进程的系统复杂度可能达到O(n)。
- 重算时必须靠锁的形式来保护任务队列和每个进程描述符。这样做会加剧对锁的争用。
- 重新计算时间片的时机是不确定的，这会给时间确定性要求很高的实时程序带来麻烦。
- 它实现的很粗糙（这也是为什么要改善Linux某些部分的一个很重要的原因）。

新的Linux调度程序减少了对循环的依赖。取而代之的是它为每个处理器维护两个优先级数组：既有活动数组也有过期数组。活动数组内的可执行队列上的进程都还有时间片剩余；而过期数组内的可执行队列上的进程都耗尽了时间片。当一个进程的时间片耗尽时，它会被移至过期数组，但在此之前，时间片已经给它重新计算好了。重新计算时间片现在变得非常简单，只要在活动和过期数组之间来回切换就行了。因为数组是通过指针进行访问的，所以交换它们用的时间就是交换指针需要的时间。这个动作由schedule()完成：

```
struct prio_array * array = rq->active;
if (!array->nr_active) {
        rq->active = rq->expired;
        rq->expired = array;
}
```

这种交换是O(1)级调度程序的核心。O(1)级的调度程序根本不需要从头到尾都忙着重新计算时间片，它只要完成一个两个步骤就能实现数组的切换。这种实现完美地解决了前面列举的所有弊端。

4.2.4 schedule()

选定下一个进程并切换到它去执行是通过schedule()函数实现的。当内核代码想要休眠时，会

⊖ 在x86体系结构上，这条指令叫做bsfl。在PPC，cntlzw用于此目的。

直接调用该函数，另外，如果有哪个进程将被抢占，那么该函数也会被唤起执行。schedule()函数独立于每个处理器运行。因此，每个CPU都要对下一次该运行哪个进程做出自己的判断。

考虑到它要完成如此繁多的任务，schedule()函数的实现实在算是简洁了。下面的代码用来判断谁是优先级最高的进程：

```
struct task_struct *prev, *next;
struct list_head *queue;
struct prio_array *array;
int idx;

prev = current;
array = rq->active;
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, struct task_struct, run_list);
```

首先，要在活动优先级数组中找到第一个被设置的位。该位对应着优先级最高的可执行进程。然后，调度程序选择这个级别链表里的头一个进程。这就是系统中优先级最高的可执行程序，也是马上会被调度执行的进程，参见图4-2。

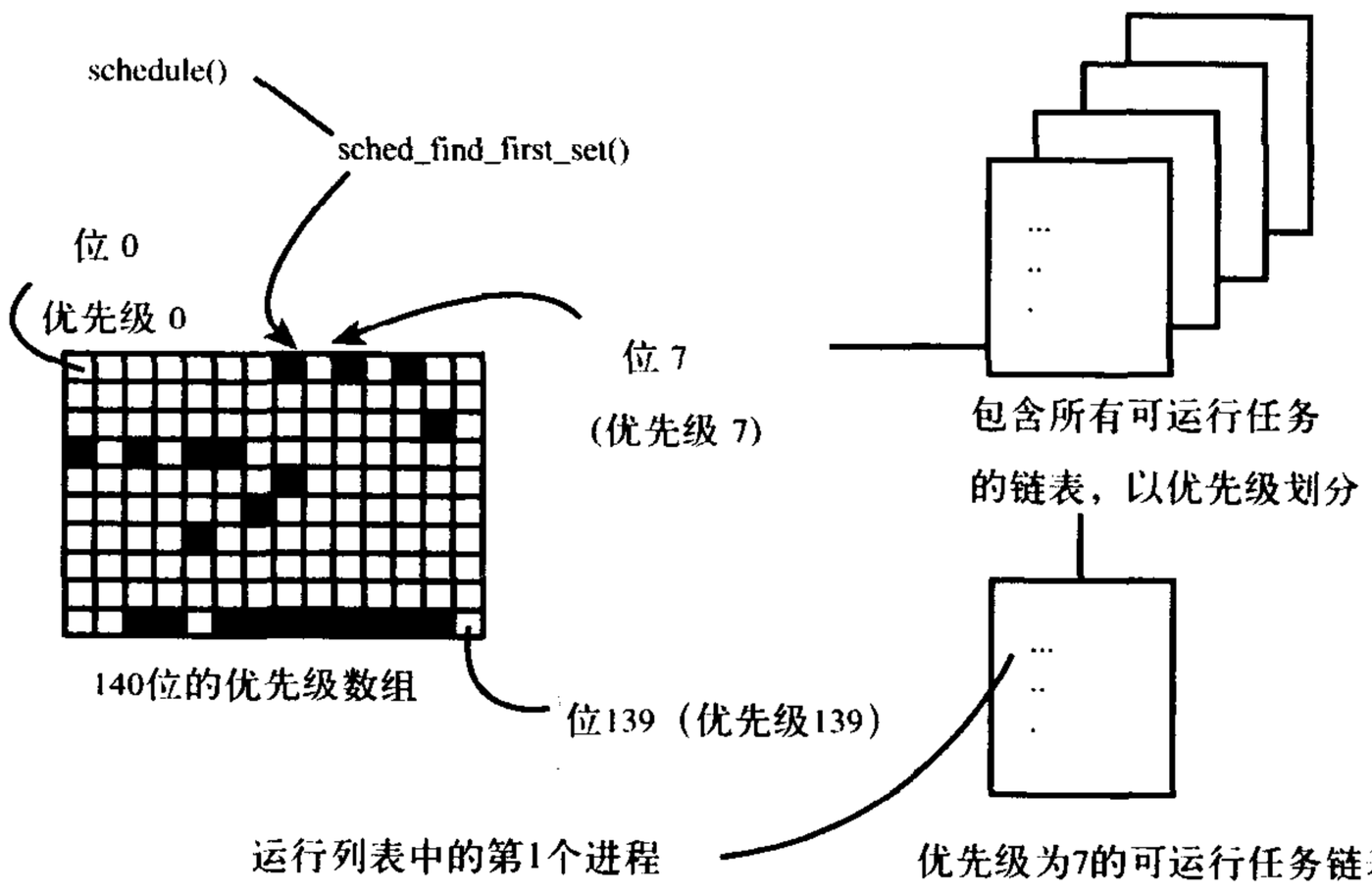


图4-2 Linux O（1）级调度程序的算法

如果prev和next不等，说明被选中的进程不是当前进程。此时函数context_switch()被调用，负责从prev切换到next。我们将在后续章节中讨论上下文切换。

上面的代码中有两点特别值得注意：首先，代码很简单，因此执行起来也很迅速。其次，系统中进程的数目对这段代码要用多长时间没有任何影响——因为根本没有用到循环来搜寻链表里最适宜的进程。实际上，不存在任何影响schedule()执行时间长短的因素。它所用的时间是恒定的。

4.2.5 计算优先级和时间片

在本章的开头，我们看到了如何利用优先级和时间片来影响调度程序做出决定。另外，我们

还知道了I/O消耗型和处理器消耗型进程以及为什么提高I/O消耗型进程的优先级会有好处。现在，我们看实际的代码是如何实现这些设计的。

进程拥有一个初始的优先级，叫做nice值。该数值变化范围为-20到+19，默认值为0。19优先级最低，-20最高。进程task_struct的static_prio域就存放着这个值。之所以起名为静态优先级(static priority)，是因为它从一开始由用户指定后，就不能改变。而调度程序要用到的动态优先级存放在prio域里。动态优先级通过一个关于静态优先级和进程交互性的函数关系计算而来。

effective_prio()函数可以返回一个进程的动态优先级。这个函数以nice值为基数，再加上-5到+5之间的进程交互性的奖励或罚分。举例来说，一个交互性很强的进程，即使它的nice值为10，它的动态优先级最终也有可能达到5。相反，一个温和的处理器吞噬者，虽然本来nice值一样是10，它最后的动态优先级却可能是12。交互性不强不弱的进程——位于I/O消耗型进程与处理器消耗型进程之间——不会得到优先级的奖励，同样也不会被罚分，所以它的动态优先级和它的nice值相等。

当然，调度程序不可能通过魔法来了解一个进程的交互性到底强不强。它必需通过一些推断来获取准确反映进程到底是I/O消耗型的还是处理器消耗型的。最明显的标准莫过于进程休眠的时间长短了。如果一个进程的大部分时间都在休眠，那么它就是I/O消耗型的。如果一个进程执行的时间比休眠的时间长，那它就是处理器消耗型的。这个标准可以向极端延伸；一个进程如果几乎所有的时间都用在休眠上，那么它就是一个纯粹的I/O消耗型进程，相反，一个进程如果几乎所有的时间都在执行，那么它就是纯粹的处理器消耗型进程。

为了支持这种推断机制，Linux记录了一个进程用于休眠和用于执行的时间。该值存放在task_struct的sleep_avg域中。它的范围从0到MAX_SLEEP_AVG。它的默认值为10毫秒。当一个进程从休眠状态恢复到执行状态时，sleep_avg会根据它休眠时间的长短而增长，直到达到MAX_SLEEP_AVG为止。相反，进程每运行一个时钟节拍，sleep_avg就做相应的递减，到0为止。

这种推断准确得让人吃惊。它的计算不仅仅基于休眠时间有多长，而且运行时间的长短也要被计算进去。所以，尽管一个进程休眠了不少时间，但它如果总是把自己的时间片用得一干二净，那么它就不会得到大额的优先级奖励——这种推断机制不仅会奖励交互性强的进程，它还会惩罚处理器耗费量大的进程，并且不会滥用这些奖惩手段。如果一个进程发生了变化，开始大量占用处理器时间，那么，它很快就会失去曾经得到的优先级提升。最后要说的是，这种衡量标准还有很快的反应速度。一个新创建的交互性进程的sleep_avg很快就会涨得很高。即便如此，由于奖励和罚分都加在作为基数的nice值上，所以用户还是可以通过改变进程的nice值来对调度程序施加影响。

另一方面，重新计算时间片相对简单了。它只要以静态优先级为基础就可以了。在一个进程创建的时候，新建的子进程和父进程均分父进程剩余的进程时间片。这样的分配很公平并且防止用户通过不断创建新进程来不停地攫取时间片。然而，当一个任务的时间片用完之后，就要根据任务的静态优先级重新计算时间片。task_timeslice()函数为给定任务返回一个新的时间片。时间片的计算只需要把优先级按比例缩放，使其符合时间片的数值范围要求就可以了。进程的优先级越高，它每次执行得到的时间片就越长。优先级最高的进程（其nice值是-20）能获得的最大时间片长度（MAX_TIMESLICE）是800毫秒。而优先级最低的进程（其nice值是+19）获得的最短时间片长度（MIN_TIMESLICE）为5毫秒或一个时钟滴答（参见第10章）。默认优先级（nice值为0）的进程得到的时间片长度为100毫秒，参见表4-1。

表4-1 调度程序时间片

进程类型	nice值	时间片长度
初始创建的进程	父进程的值	父进程的一半
优先级最低的进程	+19	5毫秒（MIN_TIMESLICE）
默认优先级的进程	0	100毫秒（DEF_TIMESLICE）
优先级最高的进程	−20	800毫秒（MAX_TIMESLICE）

调度程序还提供了另外一种机制以支持交互进程：如果一个进程的交互性非常强，那么当它时间片用完后，它会被再放置到活动数组而不是过期数组中。回忆一下，重新计算时间片是通过活动数组与过期数组之间的切换来进行的。一般进程在用尽它们的时间片后，都会被移至过期数组，当活动数组中没有剩余进程的时候，这两个数组就会被交换；活动数组变成过期数组，过期数组替代活动数组。这种操作提供了时间复杂度为O(1)的时间片重新计算。但在这种交换发生之前，交互性很强的一个进程有可能已经处于过期数组中了，当它需要交互的时候，它却无法执行，因为必须等到数组交换发生为止才可执行。将交互式的进程重新插入到活动数组可以避免这种问题。但该进程不会被立即执行，它会和优先级相同的进程轮流着被调度和执行。该逻辑在scheduler_tick()中实现，该函数会被定时器中断调用（在第10章中讨论）：

```
struct task_struct *task;
struct runqueue *rq;

task = current;
rq = this_rq();

if (!--task->time_slice) {
        if (!TASK_INTERACTIVE(task) || EXPIRED_STARVING(rq))
                enqueue_task(task, rq->expired);
        else
                enqueue_task(task, rq->active);
}
```

首先，这段代码减小进程时间片的值，再看它是否为0。如果是的话就说明进程的时间片已经耗尽，需要把它插入到一个数组中，所以该代码先通过TASK_INTERACTIVE()宏来查看这个进程是不是交互型的进程。该宏主要基于进程的nice值来判定它是不是一个“交互性十足”的进程。nice值越小（优先级越高），越能说明该进程交互性越高。一个nice值为19的进程交互性表现得再强，它也不可能被重新加入到活动数组中去。而一个nice值为−20的进程要想不被重新加入，只有拼命的占用处理器才行。一个拥有默认优先级的进程，需要表现出一定的交互性才能被重新加入，但这通常很容易就能满足。接着，EXPIRED_STARVING()宏负责检查过期数组内的进程是否处于饥饿状态——是否已经有相对较长的时间没有发生数组切换了。如果最近一直没有发生切换，那么再把当前的进程放置到活动数组会进一步拖延切换—过期数组内的进程会越来越饿。只要不发生这种情况，进程就会被重新放置在活动数组里。否则，进程会被放入过期数组里，这也是最普通的一种操作。

4.2.6 睡眠和唤醒

休眠（被阻塞）的进程处于一个特殊的不可执行状态。这点非常重要，否则，没有这种特殊

状态的话，调度程序就可能选出一个本不愿意被执行的进程，更糟糕的是，休眠就必须以轮询的方式实现了。进程休眠有各种原因，但肯定都是为了等待一些事件。事件可能是一段时间、从文件I/O读更多数据，或者是某个硬件事件。一个进程还有可能在尝试获取一个已被占用的内核信号量时被迫进入休眠（这部分在第9章中讨论）。休眠的一个常见原因就是文件I/O——如进程对一个文件执行了read()操作，而这需要从磁盘里读取。还有，进程在获取键盘输入的时候也需要等待。无论哪种情况，内核的操作都相同：进程把它自己标记成休眠状态，把自己从可执行队列移出，放入等待队列，然后调用schedule()选择和执行一个其他进程。唤醒的过程刚好相反：进程被设置为可执行状态，然后再从等待队列中移到可执行队列。

在前一章里曾经讨论过，休眠有两种相关的进程状态：TASK_INTERRUPTIBLE和TASK_UNINTERRUPTIBLE。它们的惟一区别是处于TASK_UNINTERRUPTIBLE的进程会忽略信号，而处于TASK_INTERRUPTIBLE状态的进程如果接收到一个信号会被提前唤醒并响应该信号。两种状态的进程位于同一个等待队列上，等待某些事件，不能够运行。

休眠通过等待队列进行处理。等待队列是由等待某些事件发生的进程组成的简单链表。内核用wake_queue_head_t来代表等待队列。等待队列可以通过DECLARE_WAITQUEUE()静态创建，也可以由init_waitqueue_head()动态创建。进程把自己放入等待队列中并设置成不可执行状态。当与等待队列相关的事件发生的时候，队列上的进程会被唤醒。为了避免产生竞争条件，休眠和唤醒的实现不能有纰漏。

针对休眠，以前曾经使用过一些简单的接口。但那些接口会带来竞争条件；有可能导致在判定条件变为真后进程却开始了休眠，那样就会使进程无限期地休眠下去。所以，在内核中进行休眠的推荐操作就相对复杂了一些：

```
/* 'q' 是我们希望睡眠的等待队列 */
DECLARE_WAITQUEUE(wait, current);

add_wait_queue(q, &wait);
while (!condition) {  /* 'condition' 是我们在等待的事件 */
        set_current_state(TASK_INTERRUPTIBLE); /*or TASK_UNINTERRUPTIBLE*/
        if(signal_pending(current))
                /* 处理信号 */
        schedule();
}
set_current_state(TASK_RUNNING);
remove_wait_queue(q, &wait);
```

进程通过执行下面几个步骤将自己加入到一个等待队列中：

1) 调用DECLARE_WAITQUEUE()创建一个等待队列的项。

2) 调用add_wait_queue()把自己加入到队列中。该队列会在进程等待的条件满足时唤醒它。当然我们必须在其他地方撰写相关代码，在事件发生时，对等待队列执行wake_up()操作。

3) 将进程的状态变更为 TASK_INTERRUPTIBLE或TASK_UNINTERRUPTIBLE。

4) 如果状态被置为TASK_INTERRUPTIBLE,则信号唤醒进程。这就是所谓的伪唤醒（唤醒不是因为事件的发生），因此检查并处理信号。

5) 检查条件是否为真；如果是的话，就没必要休眠了。如果条件不为真，调用schedule()。

6) 当进程被唤醒的时候，它会再次检查条件是否为真。如果是，它就退出循环，如果不是，它再次调用schedule()并一直重复这步操作。

7) 当条件满足后，进程将自己设置为TASK_RUNNING并调用remove_wait_queue()把自己移出等待队列。

如果在进程开始休眠之前条件就已经达成了，那么循环会退出，进程不会存在错误的进入休眠的倾向。需要注意的是，内核代码在循环体内常常需要完成一些其他的任务，比如，它可能在调用schedule()之前需要释放掉锁，而在这以后再重新获取它们，或者响应其他什么事件。

唤醒操作通过函数wake_up()进行，它会唤醒指定的等待队列上的所有进程。它调用函数try_to_wake_up()，该函数负责将进程设置为TASK_RUNNING状态，调用activate_task()将此进程放入可执行队列，如果被唤醒的进程优先级比当前正在执行的进程的优先级高，还要设置need_resched标志。通常哪段代码促使等待条件达成，它就要负责随后调用wake_up()函数。举例来说，当磁盘数据到来以后，VFS就要负责对等待队列调用wake_up()，以便唤醒队列中等待这些数据的进程。

关于休眠有一点需要注意，存在虚假的唤醒。有时候进程被唤醒并不是因为它所等待的条件达成了，所以才需要用一个循环处理来保证它等待的条件真正达成。图4-3描述了每个调度程序状态之间的关系。

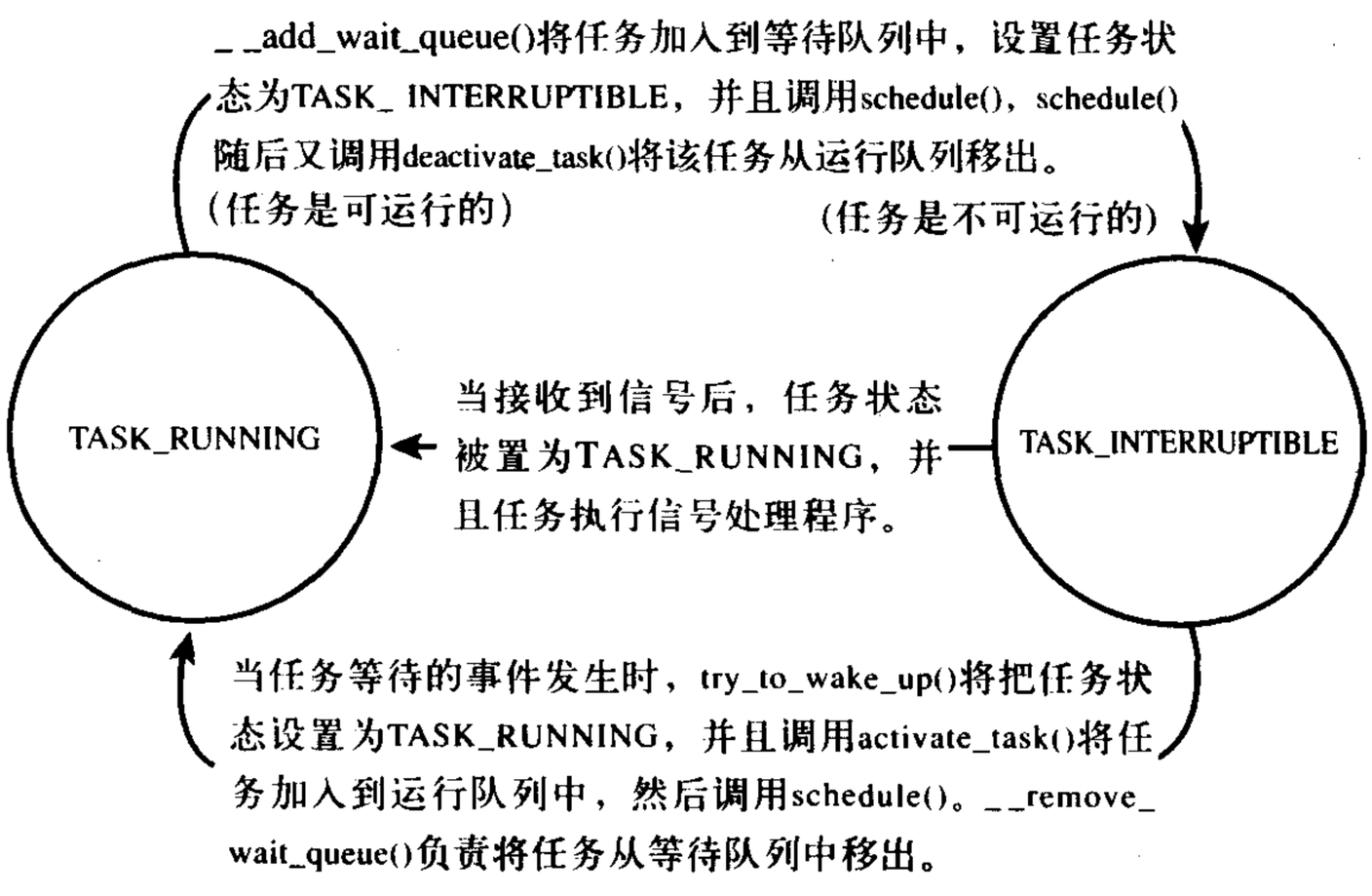


图4-3 休眠和唤醒

4.2.7 负载平衡程序

前面提到过，Linux的调度程序为对称多处理系统的每个处理器准备了单独的可执行队列和锁。也就是说，每个处理器拥有一个自己的进程链表，而它只对属于自己的这些进程进行调度操作。出于效率的考虑，整个调度系统从每个处理器来看都是独立的。那么，针对对称多处理系统，调度程序有没有提供什么加强型的策略来提升整体的调度呢？如果可执行队列间出现负载不均衡的情况时，比如一个处理器的队列上有五个进程，而另外一个处理器的队列上只有一个进程，该怎

么办呢？这些问题由负载平衡程序解决，它负责保证可执行队列之间的负载处于均衡状态。负载平衡程序会拿当前处理器的可执行队列和系统中的其他可执行队列做比较。如果它发行了不均衡，就会把相对繁忙的队列中的进程抽到当前的可执行队列中来。理想情况下，每个队列上的进程数目应该相等。这是个难以企及的目标，但负载平衡程序做得已经非常接近了。

负载平衡程序由kernel/sched.c中的函数load_balance()来实现。它有两种调用方法。在schedule()执行的时候，只要当前的可执行队列为空，它就会被调用。此外，它还会被定时器调用：系统空闲时每隔1毫秒调用一次或者在其他情况下每隔200毫秒调用一次。在单处理器系统中，load_balance()不会被调用，它甚至不会被编译进内核。因为那里面只有一个可执行队列，因此根本没有平衡负载的必要。

负载平衡程序调用时需要锁住当前处理器的可执行队列并且屏蔽中断，以避免可执行队列被并发地访问。在执行schedule()的时候调用load_balance()，这种情况下要做的工作相当明显，因为当前的可执行队列为空，所以要找到一些进程并且插入到这个队列里，好处显而易见。然而在定时器中调用load_balance()的时候，要做的工作就不是那么明显；它需要解决所有运行队列间所有的失衡，使它们大致持平，参见图4-4。

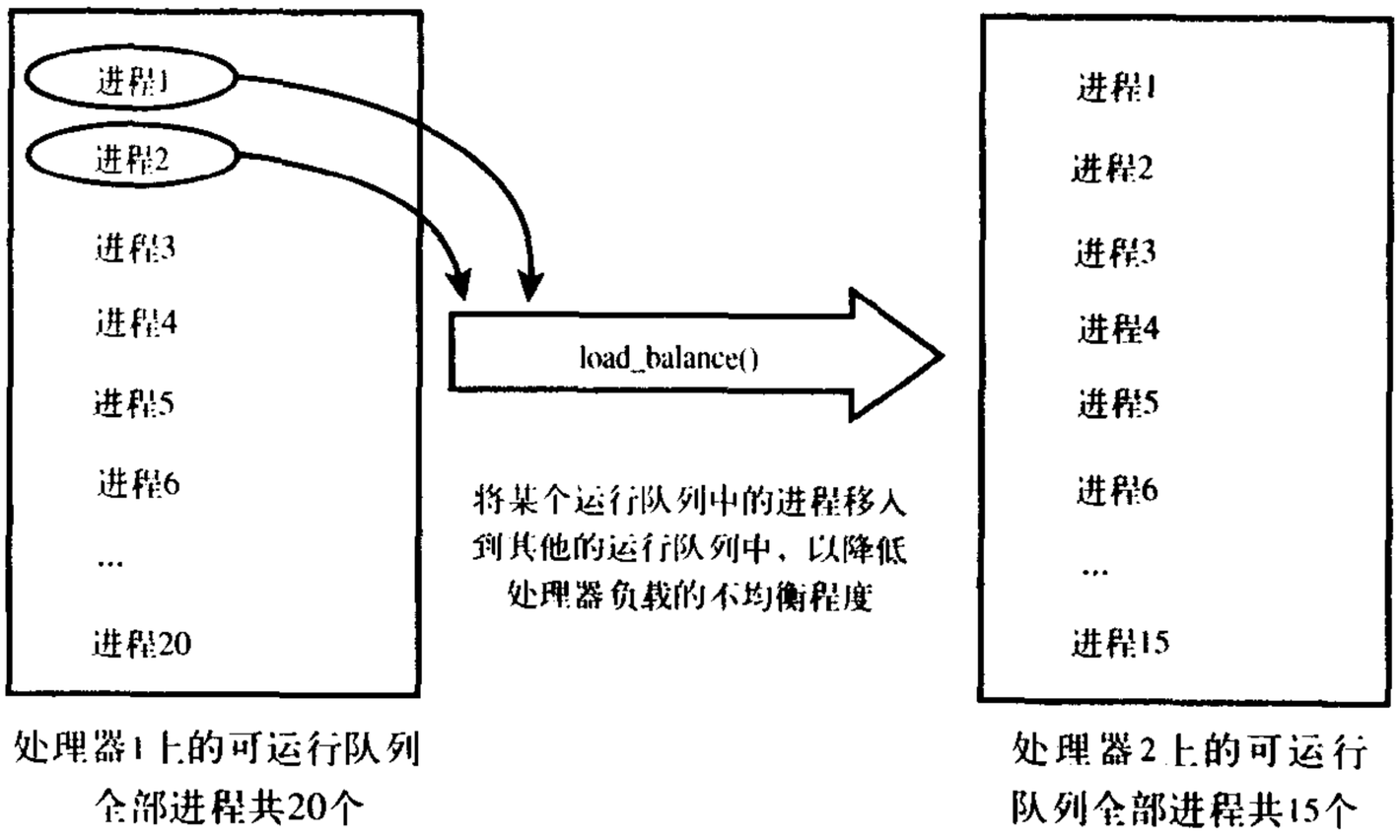


图4-4 负载平衡程序

尽管load_balance()和其他相关函数算得上是又大又复杂，但它们所完成的操作步骤理解起来却并不很困难：

1) 首先，load_balance()调用find_busiest_queue()，找到最繁忙的可执行队列。也就是说，该队列中的进程数目最多。如果没有哪个可执行队列中进程的数目比当前队列中的数目多25%或25%以上，find_busiest_queue()返回NULL，并且load_balance()也返回。否则，最繁忙的可执行队列就被返回。

2) 其次，load_balance()从最繁忙的运行队列中选择一个优先级数组以便抽取进程。最好是过期数组，因为那里面的进程已经有相对较长的一段时间没有运行了，很可能不在处理器的高速缓存中（换句话说，它们不是高速缓存命中（cache hot））。如果过期数组为空，那就只能选活动数组。

3) 接着，load_balance()寻找到含有进程并且优先级最高（值最小）的链表，因为把优先级高

的进程平均分散开来才是最重要的。

4) 分析找到的所有这些优先级相同的进程，选择一个不是正在执行，也不会因为处理器相关性而不可移动，并且不在高速缓存中的进程。如果有进程满足这些条件，调用pull_task()将其从最繁忙的队列中抽取到当前队列。

5) 只要可执行队列之间仍然不均衡，就重复上面两个步骤，继续从繁忙的队列中抽取进程到当前队列。这最终会消除不平衡，此时，解除对当前运行队列进行的锁定，从load_balance()返回。

下面是load_balance()，稍稍简化了一下但总的来说比较完整：

```
static int load_balance(int this_cpu, runqueue_t *this_rq,
                        struct sched_domain *sd, enum idle_type idle)
{
        struct sched_group *group;
        runqueue_t *busiest;
        unsigned long imbalance;
        int nr_moved;

        spin_lock(&this_rq->lock);

        group = find_busiest_group(sd, this_cpu, &imbalance, idle);
        if (!group)
                goto out_balanced;

        busiest = find_busiest_queue(group);
        if (!busiest)
                goto out_balanced;

        nr_moved = 0;
        if (busiest->nr_running > 1) {
                double_lock_balance(this_rq, busiest);
                nr_moved = move_tasks(this_rq, this_cpu, busiest,
                                      imbalance, sd, idle);
                spin_unlock(&busiest->lock);
        }
        spin_unlock(&this_rq->lock);

        if (!nr_moved) {
                sd->nr_balance_failed++;

                if (unlikely(sd->nr_balance_failed > sd->cache_nice_tries+2)) {
                        int wake = 0;

                        spin_lock(&busiest->lock);
                        if (!busiest->active_balance) {
                                busiest->active_balance = 1;
                                busiest->push_cpu = this_cpu;
                                wake = 1;
                        }
                        spin_unlock(&busiest->lock);
```

```
                        if (wake)
                                wake_up_process(busiest->migration_thread);
                        sd->nr_balance_failed = sd->cache_nice_tries;
                }
        } else
                sd->nr_balance_failed = 0;

        sd->balance_interval = sd->min_interval;

        return nr_moved;
out_balanced:

        spin_unlock(&this_rq->lock);

        if (sd->balance_interval < sd->max_interval)
                sd->balance_interval *= 2;
        return 0;
}
```

4.3 抢占和上下文切换

上下文切换，也就是从一个可执行进程切换到另一个可执行进程，由定义在kernel/ sched.c中的context_switch()函数负责处理。每当一个新的进程被选出来准备投入运行的时候，schedule()就会调用该函数。它完成了两项基本的工作：

- 调用定义在<asm/mmu_context.h>中的switch_mm()，该函数负责把虚拟内存从上一个进程映射切换到新进程中。
- 调用定义在<asm/system.h>中的switch_to()，该函数负责从上一个进程的处理器状态切换到新进程的处理器状态。这包括保存、恢复栈信息和寄存器信息。

内核必须知道在什么时候调用schedule()。如果仅靠用户程序代码显式地调用schedule()，它们可能就会永远地执行下去。相反，内核提供了一个need_resched标志来表明是否需要重新执行一次调度（见表4-2）。当某个进程耗尽它的时间片时，scheduler_tick()就会设置这个标志；当一个优先级高的进程进入可执行状态的时候，try_to_wake_up()也会设置这个标志。

表4-2 用于访问和操作need_resched的函数

函 数	目 的
set_tsk_need_resched()	设置指定进程中的need_resched标志
clear_tsk_need_resched()	清除指定进程中的need_resched标志
need_resched()	检查need_resched标志的值；如果被设置就返回真，否则返回假

再返回用户空间以及从中断返回的时候，内核也会检查need_resched标志。如果已被设置，内核会在继续执行之前调用调度程序。

每个进程都包含一个need_resched标志，这是因为访问进程描述符内的数值要比访问一个全局变量快（因为current宏速度很快并且描述符通常都在高速缓存中）。在2.2以前的内核版本中，该标志曾经是一个全局变量。2.2到2.4版内核中它在task_struct中。而在2.6版中，它被移到thread_info结构体里，用一个特别的标志变量中的一位来表示。可见，内核开发者总是在不断改进。

4.3.1 用户抢占

内核即将返回用户空间的时候，如果need_resched标志被设置，会导致schedule()被调用，此时就会发生用户抢占。在内核返回用户空间的时候，它知道自己是安全的，因为既然它可以继续去执行当前进程，那么它当然可以再去选择一个新的进程去执行。所以，内核无论是在从中断处理程序还是在系统调用后返回，都会检查need_resched标志。如果它被设置了，那么，内核会选择一个其他（更合适的）进程投入运行。从中断处理程序或系统调用返回的代码都是跟体系结构相关的，在entry.S（此文件不仅包含内核入口部分的程序，内核退出部分的相关代码也在其中）文件中通过汇编语言来实现。

简而言之，用户抢占在以下情况时产生：

- 从系统调返回用户空间。
- 从中断处理程序返回用户空间。

4.3.2 内核抢占

与其他大部分的Unix变体和其他大部分的操作系统不同，Linux完整地支持内核抢占。在不支持内核抢占的内核中，内核代码可以一直执行，到它完成为止。也就是说，调度程序没有办法在一个内核级的任务正在执行的时候重新调度——内核中的各任务是协作方式调度的，不具备抢占性。内核代码一直要执行到完成（返回用户空间）或明显的阻塞为止。在2.6版的内核中，内核引入了抢占能力；现在，只要重新调度是安全的，那么内核就可以在任何时间抢占正在执行的任务。

那么，什么时候重新调度才是安全的呢？只要没有持有锁，内核就可以进行抢占。锁是非抢占区域的标志。由于内核是支持SMP的，所以，如果没有持有锁，那么正在执行的代码就是可重新导入的，也就是可以抢占的。

为了支持内核抢占所作的第一处变动就是为每个进程的thread_info引入了preempt_count计数器。该计数器初始值为0，每当使用锁的时候数值加1，释放锁的时候数值减1。当数值为0的时候，内核就可执行抢占。从中断返回内核空间的时候，内核会检查need_resched和preempt_count的值。如果need_resched被设置，并且preempt_count为0的话，这说明有一个更为重要的任务需要执行并且可以安全地抢占，此时，调度程序就会被调用。如果preempt_ count不为0，说明有当前任务持有锁，所以抢占是不安全的。这时，就会像通常那样直接从中断返回当前执行进程。如果当前进程持有的所有的锁都被释放了，那么preempt_count就会重新为0。此时，释放锁的代码会检查need_resched是否被设置。如果是的话，就会调用调度程序。有些内核代码需要允许或禁止内核抢占，相关内容会在第9章讨论。

如果内核中的进程被阻塞了，或它显式地调用了schedule()，内核抢占也会显式地发生。这种形式的内核抢占从来都是受支持的，因为根本无需额外的逻辑来保证内核可以安全地被抢占。如果代码显式的调用了schedule()，那么它应该清楚自己是可以安全地被抢占的。

内核抢占会发生在：

- 当从中断处理程序正在执行，且返回内核空间之前。
- 当内核代码再一次具有可抢占性的时候。
- 如果内核中的任务显式的调用schedule()。
- 如果内核中的任务阻塞（这同样也会导致调用schedule()）。

4.4 实时

Linux提供了两种实时调度策略：SCHED_FIFO和SCHED_RR。而普通的、非实时的调度策略是SCHED_NORMAL。SCHED_FIFO实现了一种简单的、先入先出的调度算法，它不使用时间片。SCHED_FIFO级的进程会比任何SCHED_NORMAL级的进程都先得到调度。一旦一个SCHED_FIFO级进程处于可执行状态，就会一直执行，直到它自己受阻塞或显式地释放处理器为止；它不基于时间片，可以一直执行下去。只有较高优先级的SCHED_FIFO或者SCHED_RR任务才能抢占SCHED_FIFO任务。如果有两个或者更多的SCHED_FIFO级进程，它们会轮流执行，但是在它们愿意让出处理机时会再次让出。只要有SCHED_FIFO级进程在执行，其他级别较低的进程就只能等待它结束后才有机会执行。

SCHED_RR与SCHED_FIFO大体相同，只是SCHED_RR级的进程在耗尽事先分配给它的时间后就不能再接着执行了。也就是说，SCHED_RR是带有时间片的SCHED_FIFO——这是一种实时轮流调度算法。当SCHED_RR任务耗尽它的时间片，在同一优先级的其他实时进程被轮流调度。时间片只用来重新调度同一优先级的进程。对于SCHED_FIFO进程，高优先级总是立即抢占低优先级，但低优先级进程决不能抢占SCHED_RR任务，即使它的时间片耗尽。

这两种实时算法实现的都是静态优先级。内核不为实时进程计算动态优先级。这能保证给定优先级别的实时进程总能抢占优先级比它低的进程。

Linux的实时调度算法提供了一种软实时工作方式。软实时的含义是，内核调度进程，尽力使进程在它的限定时间到来前运行，但内核不保证总能满足这些进程的要求。相反，硬实时系统保证在一定条件下，可以满足任何调度的要求。Linux对于实时任务的调度不做任何保证。虽然不能保证硬实时工作方式，但Linux的实时调度算法的性能还是很不错的。2.6版的内核可以满足非常严格的时间要求。

实时优先级范围从0到MAX_RT_PRIO减1。默认情况下，MAX_RT_PRIO为100——所以默认的实时优先级范围是从0到99。SCHED_NORMAL级进程的nice值共享了这个取值空间；它的取值范围是从MAX_RT_PRIO到(MAX_RT_PRIO + 40)。也就是说，在默认情况下，nice值从−20到+19直接对应的是从100到139的实时优先级范围。

4.5 与调度相关的系统调用

Linux提供了一族系统调用，用于管理与调度程序相关的参数。这些系统调用可以用来操作和处理进程优先级、调度策略及处理器，同时还提供了显式的将处理器交给其他进程的机制。

许多书籍——还有友善的man帮助文件——都提供了这些系统调用（它们都包含在C库中，没用什么太多的封装，基本上只调用了系统调用而已）的说明。表4-3列举了这些系统调用并给出了简短的说明。第4章会讨论它们是如何实现的。

表4-3 与调度相关的系统调用

系统调用	描 述
nice()	设置进程的nice值
sched_setscheduler()	设置进程的调度策略

（续）

系统调用	描　述
sched_getscheduler()	获取进程的调度策略
sched_setparam()	设置进程的实时优先级
sched_getparam ()	获取进程的实时优先级
sched_get_priority_max ()	获取实时优先级的最大值
sched_get_priority_min ()	获取实时优先级的最小值
sched_rr_get_interval()	获取进程的时间片值
sched_setaffinity()	设置进程的处理器的亲和力
sched_getaffinity ()	获取进程的处理器的亲和力
sched_yield()	暂时让出处理器

4.5.1 与调度策略和优先级相关的系统调用

sched_setscheduler()和sched_getscheduler()分别用于设置和获取进程的调度策略和实时优先级。与其他的系统调用相似，它们的实现也是由许多参数检查、初始化和清理构成的。其实最重要的工作在于读取或改写进程task_struct的policy和rt_priority的值。

sched_setparam()和sched_getparam()分别用于设置和获取进程的实时优先级。这两个系统调用获取封装在sched_param特殊结构体中的rt_priority。sched_get_priority_max ()和sched_get_priority_min ()分别用于返回给定调度策略的最大和最小优先级。实时调度策略的最大优先级是MAX_USER_RT_PRIO减1，最小优先级等于1。

对于一个普通的进程，nice()函数可以将给定进程的静态优先级增加一个给定的量。只有超级用户才能在调用它时使用负值，从而提高进程的优先级。nice()函数会调用内核的set_ user_nice()函数，这个函数会设置进程的task_struct的static_prio和prio值。

4.5.2 与处理器绑定有关的系统调用

Linux调度程序提供强制的处理器绑定（processor affinity）机制。也就是说，虽然它尽力通过一种软的或者说自然的亲和性试图使进程尽量在同一个处理器上运行，但它也允许用户强制指定“这个进程无论如何都必须在这些处理器上运行。”这种强制的亲合性保存在进程task_struct的cpus_allowed这个位掩码标志中。该掩码标志的每一位对应一个系统可用的处理器。默认情况下，所有的位都被设置，进程可以在系统中所有可用的处理器上执行。用户可以通过sched_ setaffinity()设置一个不同的一个或几个位组合的位掩码。而调用sched_ getaffinity()则返回当前的cpus_allowed位掩码。

内核提供的强制处理器绑定的方法很简单。首先，当处理进行第一次创建，它继承了其父进程的相关掩码。由于父进程运行在指定处理器上，子进程也运行在相应处理器上。其次，当处理器绑定关系改变时，内核会采用“移植线程”把任务推到合法的处理器上。最后，加载平衡器只把任务拉到允许的处理上。因此，进程只运行在指定处理器上，对处理器的指定是由该进程描述符的cpus_allowed域设置的。

4.5.3 放弃处理器时间

Linux通过sched_yield()系统调用，提供了一种让进程显式地将处理器时间让给其他等待执行

进程的机制。它是通过将进程从活动队列中（因为进程正在执行，所以它肯定位于此队列当中）移到过期队列中实现的。由此产生的效果不仅抢占了该进程并将其放入优先级队列的最后面，还将其放入过期队列中——这样能确保在一段时间内它都不会再被执行了。由于实时进程不会过期，所以属于例外。它们只被移动到其优先级队列的最后面（不会放到过期队列中）。在Linux的早期版本中，sched_yield()的语义有所不同，进程只会被放置到优先级队列的末尾。放弃的时间往往不会太长。现在，应用程序甚至内核代码在调用sched_yield()前，应该仔细考虑是否真的希望放弃处理器时间。

内核代码为了方便，可以直接调用yield()，它先要确定给定进程确实处于可执行状态，然后再调用sched_yield()。用户空间的应用程序直接使用sched_yield()系统调用就可以了。

4.6 调度程序小结

进程调度程序是内核重要的组成部分，因为运行着的进程首先在使用计算机（至少在我们大多数人看来）。然而，满足进程调度的各种需要决不是轻而易举的：很难找到“一刀切”的算法，既适合众多的可运行进程，又具有可伸缩性，还能在调度周期和吞吐量之间求得平衡，同时还满足各种负载的需求。不过，Linux内核的新调度程序尽量满足各个方面，并以较完善的可伸缩性和外在的魅力为所有的情况提供了最佳的解决方案。

遗留问题包括微调（或完全替代）交互性评价程序，当这种程序给出正确的预测时，那是幸运的，但当它猜测失误时，就是全方位的疼痛。二中择一的选择还会继续；总有一天我们会看到在主流内核中有一种新的实现。

改进NUMA（非一致存储结构）机器的功能变得越来越重要，因为NUMA逐渐流行起来。为了支持调度程序的范围，一种描述进程布局的调度程序抽象结构溶入到2.6系列早期的主流内核中。

本章考察了进程调度所遵循的基本原理、具体实现、调度算法以及目前Linux内核所使用的接口。下一章将涵盖内核提供给运行进程的主要接口——系统调用。

第5章

系统调用

为了和用户空间上运行的进程进行交互，内核提供了一组接口。透过该接口，应用程序可以访问硬件设备和其他操作系统资源。这组接口在应用程序和内核之间扮演了使者的角色，应用程序发送各种请求，而内核负责满足这些请求（或者让应用程序暂时搁置）。实际上提供这组接口主要是为了保证系统稳定可靠，避免应用程序恣意妄行，惹出大麻烦。

系统调用在用户空间进程和硬件设备之间添加了一个中间层。该层主要作用有三个。第一，它为用户空间提供了一种硬件的抽象接口。举例来说，当需要读些文件的时候，应用程序就可以不去管磁盘类型和介质，甚至不用去管文件所在的文件系统到底是哪种类型。第二，系统调用保证了系统的稳定和安全。作为硬件设备和应用程序之间的中间人，内核可以基于权限和其他一些规则对需要进行的访问进行裁决。举例来说，这样可以避免应用程序不正确地使用硬件设备，窃取其他进程的资源，或做出其他什么危害系统的事情。第三，在第3章中曾经提到过，每个进程都运行在虚拟系统中，而在用户空间和系统的其余部分提供这样一层公共接口，也是出于这种考虑。如果应用程序可以随意访问硬件而内核又对此一无所知的话，几乎就没法实现多任务和虚拟内存，当然也不可能实现良好的稳定性和安全性。在Linux中，系统调用是用户空间访问内核的惟一手段；除异常和陷入外，它们是内核惟一的合法入口。实际上，其他的像设备文件和/proc之类的方式，最终也还是要通过系统调用进行访问的。而有趣的是，Linux提供的系统调用却比大部分操作系统都少得多[⊖]。

本章重点强调Linux系统调用的规则和实现方法。

5.1 API、POSIX和C库

一般情况下，应用程序通过应用编程接口(API)而不是直接通过系统调用来编程。这点很重要，因为应用程序使用的这种编程接口实际上并不需要和内核提供的系统调用对应。一个API定义了一组应用程序使用的编程接口。它们可以实现成一个系统调用，也可以通过调用多个系统调用来实现，而完全不使用任何系统调用也不存在问题。实际上，API可以在各种不同的操作系统上实现，给应用程序提供完全相同的接口，而它们本身在这些系统上的实现却可能迥异。

在Unix世界中，最流行的应用编程接口是基于POSIX标准的。从纯技术的角度看，POSIX是由IEEE[⊜]的一组标准组成，其目标是提供一套大体上基于Unix的可移植操作系统标准。Linux是与POSIX兼容的。

⊖ x86系统上大概有250个系统调用（每种体系结构都会定义一些独特的系统调用）。尽管有些系统还没有完全公布所有的系统调用，但据估计某些操作系统的系统调用数有上千个。

⊜ IEEE(eye-triple-E)是电气和电子工程师协会。这是一个非盈利组织，它涉及的技术领域非常广泛，并且对许多重要标准负责。欲了解更多信息，请访问http://www.ieee.org。

POSIX是说明API和系统调用之间关系的一个极好例子。在大多数Unix系统上，根据POSIX而定义的API函数和系统调用之间有着直接关系。实际上，POSIX标准就是仿照早期Unix系统的界面建立的。另一方面，许多操作系统，像Windows NT，尽管和Unix没有什么关系，也提供了与POSIX兼容的库。

Linux的系统调用像大多数Unix系统一样，作为C库的一部分提供如图5-1所示。如图5-1所示C库实现了Unix系统的主要API，包括标准C库函数和系统调用。所有的C程序都可以使用C库，而由于C语言本身的特点，其他语言也可以很方便地把它们封装起来使用。此外，C库提供了POSIX的绝大部分API。

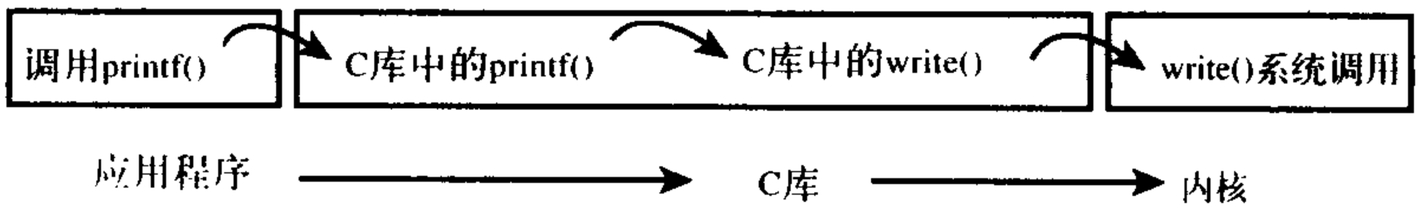


图5-1 调用printf()函数时，应用程序、C库和内核之间的关系

从程序员的角度看，系统调用无关紧要；他们只需要跟API打交道就可以了。相反，内核只跟系统调用打交道；库函数及应用程序是怎么使用系统调用不是内核所关心的。但是，内核必须时刻牢记系统调用所有潜在的用途并保证它们有良好的通用性和灵活性。

关于Unix的界面设计有一句通用的格言“提供机制而不是策略”。换句话说，Unix的系统调用抽象出了用于完成某种确定目的的函数。至于这些函数怎么用完全不需要内核去关心。（区别对待机制（mechanism）和策略（policy）是Unix设计中的一大亮点。大部分的编程问题都可以被切割成两个部分：“需要提供什么功能”（机制）和“怎样实现这些功能”（策略）。如果由程序中的独立部分分别负责机制和策略的实现，那么开发软件就更容易，也更容易适应不同的需求。——译者注）

5.2 系统调用

系统调用（在Linux中常称作syscalls）通常通过函数进行调用。它们通常都需要定义一个或几个参数（输入）而且可能产生一些副作用[⊖]，例如写某个文件或向给定的指针拷贝数据等等。系统调用还会通过一个long类型[⊜]的返回值来表示成功或者错误。通常，但也不绝对，用一个负的返回值来表明错误。返回一个0值通常（当然仍不是绝对的）表明成功。Unix系统调用在出现错误的时候会把错误码写入errno全局变量。通过调用perror()库函数，可以把该变量翻译成用户可以理解的错误字符串。

当然，系统调用最终具有一种明确的操作。举例来说，如getpid()系统调用，根据定义它会返回当前进程的PID。内核中它的实现非常简单：

```
asmlinkage long sys_getpid(void)
{
        return current->tgid;
}
```

⊖ 注意这里用的是可能。尽管绝大部分系统调用都会产生某种副作用（就是说，它们会使系统的状态发生某种变化），但还是有一些系统调用，如getpid()，仅仅返回一些内核数据。

⊜ 使用long类型是为了与64位的硬件体系结构保持兼容。

注意，定义中并没有规定它要如何实现。内核必须提供系统调用所希望完成的功能，但它完全可以按照自己预期的方式去实现，只要最后的结果正确就行了。当然，上面的系统调用太简单，也没有什么更多的实现手段（当然没有更简单的方法）[⊖]。

上述的系统调用尽管非常简单，但我们还是可以从中发现两个特别之处。首先，注意函数声明中的asmlinkage限定词，这是一个小戏法，用于通知编译器仅从栈中提取该函数的参数。所有的系统调用都需要这个限定词。其次，注意系统调用get_pid()在内核中被定义成sys_ getpid()。这是Linux中所有系统调用都应该遵守的命名规则：系统调用bar()在内核中也实现为sys_bar()函数。

5.2.1 系统调用号

在Linux中，每个系统调用被赋予一个系统调用号。这样，通过这个独一无二的号就可以关联系统调用。当用户空间的进程执行一个系统调用的时候，这个系统调用号就被用来指明到底是要执行哪个系统调用；进程不会提及系统调用的名称。

系统调用号相当关键；一旦分配就不能再有任何变更，否则编译好的应用程序就会崩溃。此外，如果一个系统调用被删除，它所占用的系统调用号也不允许被回收利用，否则，以前编译过的代码会调用这个系统调用，但事实上缺调用的是另一个系统调用。Linux有一个“未实现”系统调用sys_ni_syscall()，它除了返回-ENOSYS外不做任何其他工作，这个错误号就是专门针对无效的系统调用而设的。虽然很罕见，但如果一个系统调用被删除，或者变得不可用，这个函数就要负责“填补空位”。

内核记录了系统调用表中的所有已注册过的系统调用的列表，存储在sys_call_table中。它与体系结构有关，一般在entry.s中定义。这个表中为每一个有效的系统调用指定了惟一的系统调用号。

5.2.2 系统调用的性能

Linux系统调用比其他许多操作系统执行得要快。Linux令人难以置信的上下文切换时间是一个重要原因；进出内核都被优化得简洁高效。另外一个原因是系统调用处理程序和每个系统调用本身也都非常简洁。

5.3 系统调用处理程序

用户空间的程序无法直接执行内核代码。它们不能直接调用内核空间中的函数，因为内核驻留在受保护的地址空间上。如果进程可以直接在内核的地址空间上读写的话，系统安全就会失去控制。

所以，应用程序应该以某种方式通知系统，告诉内核自己需要执行一个系统调用，希望系统切换到内核态，这样内核就可以代表应用程序来执行该系统调用了。

通知内核的机制是靠软中断实现的：通过引发一个异常来促使系统切换到内核态去执行异常处理程序。此时的异常处理程序实际上就是系统调用处理程序。x86系统上的软中断由int $0x80指令产生。这条指令会触发一个异常导致系统切换到内核态并执行第128号异常处理程序，而该程序

⊖ 你可能会想为什么getpid()返回的是tgid（即线程组ID）呢？原因在于，对于普通进程来说，TGID和PID相等。对于线程来说，同一线程组内的所有线程其TGID都相等。这使得这些线程能够调用getpid()并得到相同的PID。

正是系统调用处理程序。这个处理程序名字起得很贴切，叫system_call()。它与硬件体系结构紧密相关[⊖]，通常在entry.S文件中用汇编语言编写。最近，x86处理器增加了一条叫做sysenter的指令。与int中断指令相比，这条指令提供了更快、更专业的陷入内核执行系统调用的方式。对这条指令的支持很快被加入内核。且不管系统调用处理程序被如何调用，单是用户空间引起异常或陷入内核就是一种重要的概念。

5.3.1 指定恰当的系统调用

因为所有的系统调用陷入内核的方式都一样，所以仅仅是陷入内核空间是不够的。因此必须把系统调用号一并传给内核。在x86上，系统调用号是通过eax寄存器传递给内核的。在陷入内核之前，用户空间就把相应系统调用所对应的号放入eax中了。这样系统调用处理程序一旦运行，就可以从eax中得到数据。其他体系结构上的实现也都类似。

system_call()函数通过将给定的系统调用号与NR_syscalls做比较来检查其有效性。如果它大于或者等于NR_syscalls，该函数就返回-ENOSYS。否则，就执行相应的系统调用。

```
call *sys_call_table(, %eax, 4)
```

由于系统调用表中的表项是以32位（4字节）类型存放的，所以内核需要将给定的系统调用号乘以4，然后用所得的结果在该表中查询其位置，参见图5-2。

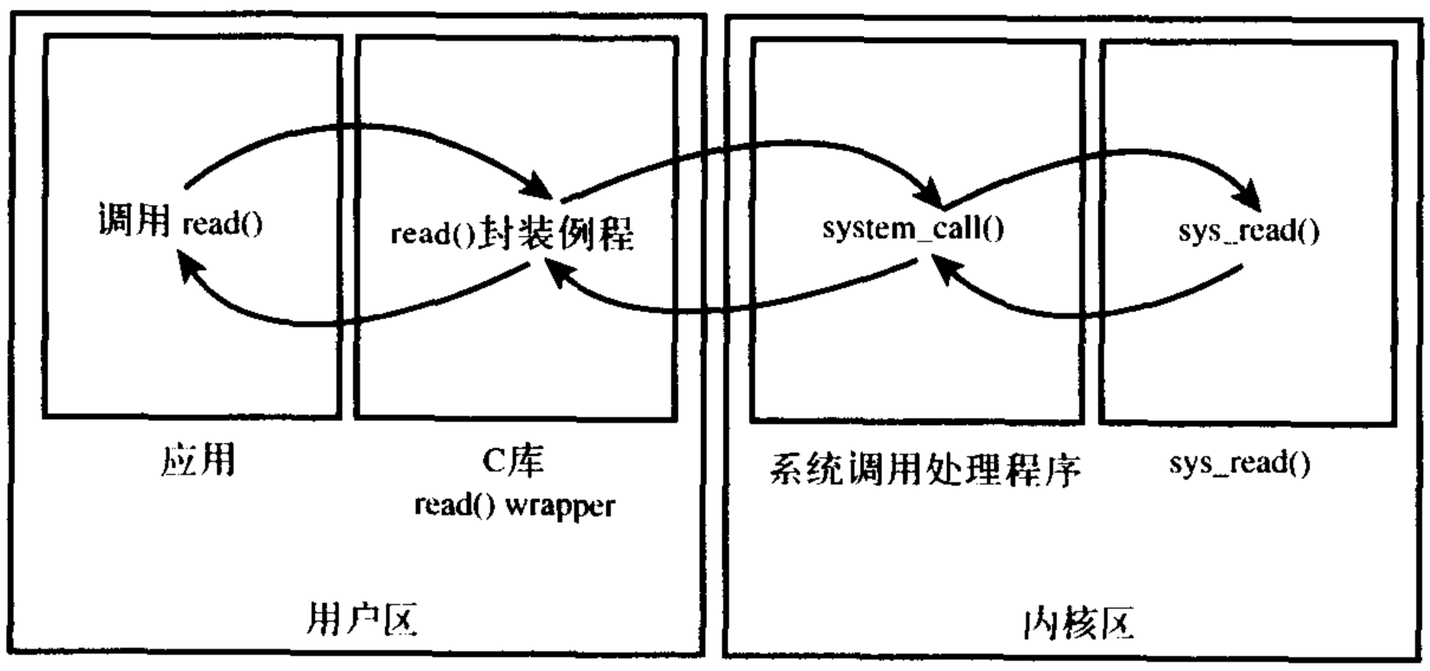


图5-2 调用系统调用处理程序以执行一个系统调用

5.3.2 参数传递

除了系统调用号以外，大部分系统调用都还需要一些外部的参数输入。所以，在发生异常的时候，应该把这些参数从用户空间传给内核。最简单的办法就是像传递系统调用号一样：把这些参数也存放在寄存器里。在x86系统上，ebx、ecx、edx、esi和edi按照顺序存放前五个参数。需要六个或六个以上参数的情况不多见，此时，应该用一个单独的寄存器存放指向所有这些参数在用户空间地址的指针。

给用户空间的返回值也通过寄存器传递。在x86系统上，它存放在eax寄存器中。

⊖ 接下来许多关于系统调用处理程序的描述都是针对x86版本的。但不用担心，所有体系结构上的实现都很类似。

5.4 系统调用的实现

实际上，一个Linux的系统调用在实现时并不需要太关心它和系统调用处理程序之间的关系。给Linux添加一个新的系统调用是件相对容易的工作。怎样设计和实现一个系统调用是难题所在，而把它加到内核里却无须太多周折。让我们关注一下实现一个新的Linux系统调用所需的步骤。

实现一个新的系统调用的第一步是决定它的用途。它要做些什么？每个系统调用都应该有一个明确的用途。在Linux中不提倡采用多用途的系统调用（一个系统调用通过传递不同的参数值来选择完成不同的工作）。ioctl()就应该被视为一个反例。

新系统调用的参数、返回值和错误码又该是什么呢？系统调用的接口应该力求简洁，参数尽可能少。系统调用的语义和行为非常关键；因为应用程序依赖于它们，所以它们应力求稳定，不做改动。

设计接口的时候要尽量为将来多做考虑。你是不是对函数做了不必要的限制？系统调用设计得越通用越好。不要假设这个系统调用现在怎么用将来也一定就是这么用。系统调用的目的可能不变，但它的用法却可能改变。这个系统调用可移植吗？别对机器的字节长度和字节序做假设。第19章将讨论这个话题。要确保不对系统调用做错误的假设，否则将来这个调用就可能会崩溃。记住Unix的格言："提供机制而不是策略"。

当你写一个系统调用的时候，要时刻注意可移植性和健壮性，不但要考虑当前，还要为将来做打算。基本的Unix系统调用经受住了时间的考验；它们中的很大一部分到现在都还和30年前一样适用和有效。

参数验证

系统调用必须仔细检查它们所有的参数是否合法有效。系统调用在内核空间执行，如果任由用户将不合法的输入传递给内核，那么系统的安全和稳定将面临极大的考验。

举例来说，与文件I/O相关的系统调用必须检查文件描述符是否有效。与进程相关的函数必须检查提供的PID是否有效。必须检查每个参数，保证它们不但合法有效，而且正确。

最重要的一种检查就是检查用户提供的指针是否有效。试想，如果一个进程可以给内核传递指针而又无须被检查，那么它就可以给出一个它根本就没有访问权限的指针，哄骗内核去为它拷贝本不允许它访问的数据，如原本属于其他进程的数据。在接收一个用户空间的指针之前，内核必须保证：

- 指针指向的内存区域属于用户空间。进程决不能哄骗内核去读内核空间的数据。
- 指针指向的内存区域在进程的地址空间里。进程决不能哄骗内核去读其他进程的数据。
- 如果是读，该内存应被标记为可读。如果是写，该内存应被标记为可写。进程决不能绕过内存访问限制。

内核提供了两个方法来完成必须的检查和内核空间与用户空间之间数据的来回拷贝。注意，内核无论何时都不能轻率地接受来自用户空间的指针！这两个方法中必须有一个被调用。为了向用户空间写入数据，内核提供了copy_to_user()，它需要三个参数。第一个参数是进程空间中的目的内存地址。第二个是内核空间内的源地址。最后一个参数是需要拷贝的数据长度（字节数）。

为了从用户空间读取数据，内核提供了copy_from_user()，它和copy_to_user()相似。该函数把第二个参数指定的位置上的数据拷贝到第一个参数指定的位置上，拷贝的数据长度由第三个参数决定。

如果执行失败，这两个函数返回的都是没能完成拷贝的数据的字节数。如果成功，返回0。当出现上述错误时，系统调用返回标准-EFAULT。

让我们以一个既用了copy_from_user()又用了copy_to_user()的系统调用作例子进行考察。这个系统调用silly_copy()毫无实际用处，它从第一个参数里拷贝数据到第二个参数。这种用途让人无法理解，它毫无必要地让内核空间作为中转站，把用户空间的数据从一个位置复制到另外一个位置。但它却能演示出上述函数的用法。

```
/*
 * silly_copy - 没有实际价值的系统调用，它把len 字节的数据从*'src' 拷贝到'dst'，毫无理由地让内
 * 核空间作为中转站。但这的确是个好例子。
 */
asmlinkage long sys_silly_copy(unsigned long*src, unsigned long *dst, unsigned long len)
{
        unsigned long buf;
        /* 如果内核字长与用户字长不匹配，则失败 */
        if(len!=sizeof(buf))
              return -EINVAL;

        /* 将用户地址空间中的src拷贝进buf*/
        if (copy_from_user(&buf, src, len) )
              return  -EFAULT;

        /*将buf拷贝进用户地址空间中的dst*/
        if (copy_to_user(dst, &buf, len) )
              return -EFAULT;
        /*返回拷贝的数据量*/
        return len;
}
```

注意，copy_to_user()和copy_from_user()都有可能引起阻塞。当包含用户数据的页被换出到硬盘上而不是在物理内存上的时候，这种情况就会发生。此时，进程就会休眠，直到缺页处理程序将该页从硬盘重新换回物理内存。

最后一项检查针对是否有合法权限。在老版本的Linux内核中，需要超级用户权限的系统调用可以通过调用suser()函数这个标准动作来完成检查。这个函数只能检查用户是否是超级用户；现在它已经被一个更细粒度的“权能”机制代替。新的系统允许检查针对特定资源的特殊权限。调用者可以使用capable()函数来检查是否有权能对指定的资源进行操作，如果它返回非零值，调用者就有权进行操作，返回零则无权操作。举个例子，capable(CAP_ SYS_NICE)可以检查调用者是否有权改变其他进程的nice值。默认情况下，属于超级用户的进程拥有所有权利而非超级用户没有任何权利。下面是另一个系统调用，展示了权能的使用：

```
asmlinkage long sys_am_i_popular (void)
{
```

```
        /* 检查用户进程是否具有CAP_SYS_NICE 权能 */
        if (!capable(CAP_SYS_NICE))
                return-EPERM;

        /*成功返回0 */
        return 0;
}
```

参见<linux/capability.h>，其中包含一份所有这些权能和其对应的权限的列表。

5.5 系统调用上下文

在第3章中曾经讨论过，内核在执行系统调用的时候处于进程上下文。current指针指向当前任务，即引发系统调用的那个进程。

在进程上下文中，内核可以休眠（比如在系统调用阻塞或显式调用schedule()的时候）并且可以被抢占。这两点都很重要。首先，能够休眠说明系统调用可以使用内核提供的绝大部分功能。在第6章中我们会看到，休眠的能力会给内核编程带来极大便利[⊖]。在进程上下文中能够被抢占，其实表明，像用户空间内的进程一样，当前的进程同样可以被其他进程抢占。因为新的进程可以使用相同的系统调用，所以必须小心，保证该系统调用是可重入的。当然，这也是在对称多处理中必须同样关心的问题。关于可重入的保护涵盖在第8章和第9章中。

当系统调用返回的时候，控制权仍然在system_call()中，它最终会负责切换到用户空间并让用户进程继续执行下去。

5.5.1 绑定一个系统调用的最后步骤

当编写完一个系统调用后，把它注册成一个正式的系统调用是件琐碎的工作：

- 首先，在系统调用表的最后加入一个表项。每种支持该系统调用的硬件体系都必须做这样的工作（大部分的系统调用都针对所有的体系结构）。从0开始算起，系统调用在该表中的位置就是它的系统调用号。如第10个系统调用分配到的系统调用号为9。
- 对于所支持的各种体系结构，系统调用号都必须定义于<asm/unistd.h>中。
- 系统调用必须被编译进内核映象（不能被编译成模块）。这只要把它放进kernel/下的一个相关文件中就可以。

让我们通过一个虚构的系统调用foo()来仔细观察一下这些步骤。首先，我们要把sys_foo加入到系统调用表中去。对于大多数体系结构来说，该表位于entry.s文件中，形式如下：

```
ENTRY(sys_call_table)
        .long sys_restart_syscall                    /* 0 */
        .long sys_exit
        .long sys_fork
        .long sys_read
        .long sys_write
```

⊖ 中断处理程序不能休眠，这使得中断处理程序所能进行的操作较之运行在进程上下文中的系统调用所能进行的操作受到了极大的限制。

```
        .long sys_open                  /* 5 */

...

        .long sys_mq_unlink
        .long sys_mq_timedsend
        .long sys_mq_timedreceive       /* 280 */
        .long sys_mq_notify
        .long sys_mq_getsetattr
```

我们把新的系统调用加到这个表的末尾：

```
        .long sys_foo
```

虽然没有明确地指定编号，但我们加入的这个系统调用被按照次序分配给了283这个系统调用号。对于每种需要支持的体系结构，我们都必须将自己的系统调用加入到其系统调用表中去。每种体系结构不需要对应相同的系统调用号。系统调用号是专属于体系结构ABI（应用程序二进制接口）的部分。通常，你需要让系统调用适应每种体系结构。你可以注意一下每隔5个表项就加入一个调用号注释的习惯，这可以在查找系统调用对应的调用号时提供方便。

接下来，我们把系统调用号加入到<asm/unistd.h>中，它的格式如下：

```
/*
 * 本文件包含系统调用号
 */

#define _NR_restart_syscall      0
#define _NR_exit                 1
#define _NR_fork                 2
#define _NR_read                 3
#define _NR_write                4
#define _NR_open                 5

...
#define _NR_ mq_unlink           278
#define _NR_ mq_timedsend        279
#define _NR_ mq_timedreceive     280
#define _NR_ mq_notify           281
#define _NR_ mq_getsetattr       282
```

然后，我们在该列表中加入下面这行：

```
#define _NR_foo                  283
```

最后，我们来实现foo()系统调用。无论何种配置，该系统调用都必须编译到核心的内核映象中去，所以我们把它放进kernel/sys.c文件中。你也可以将其放到与其功能联系最紧密的代码中去，假如它的功能与调度相关，那么你也可以把它放到kernel/sched.c中去。

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

/*
 * sys_foo - 人人都喜欢的系统调用
```

```
/*
 * 返回每个进程的内核栈大小
 */
asmlinkage long sys_foo(void)
{
        return THREAD_SIZE;
}
```

就是这样！严格说来，现在就可以在用户空间调用foo()系统调用了。

5.5.2 从用户空间访问系统调用

通常，系统调用靠C库支持。用户程序通过包含标准头文件并和C库链接，就可以使用系统调用（或者调用库函数，再由库函数实际调用）。但如果你仅仅写出系统调用，glibc库恐怕并不提供支持。

值得庆幸的是，Linux本身提供了一组宏，用于直接对系统调用进行访问。它会设置好寄存器并调用陷入指令。这些宏是_syscalln()，其中n的范围从0到6。代表需要传递给系统调用的参数个数，这是由于该宏必须了解到底有多少参数按照什么次序压入寄存器。举个例子，open()系统调用的定义是：

```
long open(const char *filename, int flags, int mode)
```

而不靠库支持，直接调用此系统调用的宏的形式为：

```
#define_NR_open    5
_syscall3(long, open, const char*, filename, int, flags, int, mode)
```

这样，应用程序就可以直接使用open()。

对于每个宏来说，都有$2+2\times n$个参数。第一个参数对应着系统调用的返回值类型。第二个参数是系统调用的名称。再以后是按照系统调用参数的顺序排列的每个参数的类型和名称。_NR_open在<asm/unistd.h>中定义，是系统调用号。该宏会被扩展成为内嵌汇编的C函数；由汇编语言执行前一节所讨论的步骤，将系统调用号和参数压入寄存器并触发软中断来陷入内核。调用open()系统调用直接把上面的宏放置在应用程序中就可以了。

让我们写一个宏来使用前面编写的foo()系统调用，然后再写出测试代码炫耀一下我们所做的努力。

```
#define __NR_foo 283
__syscall0(long, foo)

int main ()
{
        long stack_size;

        stack_size = foo ();
        printf ("The kernel stack size is %ld\n", stack_size);

        return 0;
}
```

5.5.3 为什么不通过系统调用的方式实现

值得庆幸的是，前面的章节已经告诉大家，建立一个新的系统调用非常容易，但却绝不提倡

这么做。的确，在我尽力描述系统调用如何工作，如何增加新的系统调用后，我现在反倒建议在增加新的系统调用时千万慎重考虑。通常都会有更好的办法用来代替新建一个系统调用以作实现。让我们看看采用系统调用作为实现方式的利弊和替代的方法。

建立一个新的系统调用的好处：

- 系统调用创建容易且使用方便。
- Linux系统调用的高性能显而易见。

问题是：

- 你需要一个系统调用号，而这需要在一个内核在处于开发版本的时候由官方分配给你。
- 系统调用被加入稳定内核后就被固化了，为了避免应用程序的崩溃，它的界面不允许做改动。
- 需要将系统调用分别注册到每个需要支持的体系结构中去。
- 在脚本中不容易调用系统调用，也不能从文件系统直接访问系统调用。
- 如果仅仅进行简单的信息交换，系统调用就大材小用了。

替代方法：

- 创建一个设备节点，通过read()和write()访问它。用ioctl()进行特别的设置操作和获取特别信息。
- 一些接口如信号量，可以用文件描述符表示以进行操作。像信号量这样的某些接口，可以用文件描述符来表示，因此也就可以按上述方式对其进行操作。
- 把增加的信息作为一个文件放在sysfs的合适位置。

对于许多接口来说，系统调用都被视为正确的解决之道。但Linux系统尽量避免每出现一种新的抽象就简单的加入一个新的系统调用。这使得它的系统调用接口简洁得令人叹为观止，也就避免了许多后悔和反对意见（系统调用再也不被使用或支持）。新系统调用增添频率很低也反映出Linux是一个相对较为稳定并且功能已经较为完善的操作系统。

5.6 系统调用小结

在本章中，我们描述了系统调用到底是什么，它们与库函数和应用程序接口（API）有怎样的关系。然后，我们考察了Linux内核如何实现系统调用，以及执行系统调用的连锁反映：陷入内核，传递系统调用号和参数，执行正确的系统调用函数，并把返回值带回用户空间。

然后，我们讨论了如何增加系统调用，并提供了从用户空间调用系统调用的简单例子。整个过程相当容易！增加一个新的系统调用没有什么难的，这一过程也就是系统调用的实现过程。本书的其余部分讨论了编写循规蹈矩、最佳、安全的系统调用所遵循的概念和内核接口规范。

最后，我们集中精力讨论了实现系统调用的正反两方面，并对增加系统调用的优缺点给予了简单;陈述。

第⑥章

中断和中断处理程序

Linux内核要对连接到计算机上的所有硬件设备进行管理，这是它份内的工作。而想要管理这些设备，首先要能和它们互通音信才行。众所周知，处理器的速度跟外围硬件设备的速度往往不在一个数量级上，因此，如果内核采取让处理器向硬件发出一个请求，然后专门等待回应的办法，显然差强人意。既然硬件的响应这么慢，那么内核就应该在此期间处理其他事务，等到硬件真正完成了请求的操作之后，再回过头来对它进行处理。想要实现这种功能，轮询（polling）可能会是一种解决办法。可以让内核定期对设备的状态进行查询，然后做出相应的处理。不过这种方法很可能会让内核做不少无用功，因为无论硬件设备是正在忙碌着完成任务还是已经大功告成，轮询总会周期性地重复执行。更好的办法是由我们来提供一种机制，让硬件在需要的时候再向内核发出信号（变内核主动为硬件主动——译者注）。这就是中断机制。

6.1 中断

中断使得硬件得以与处理器进行通信。举个例子，在你敲打键盘的时候，键盘控制器（控制键盘的硬件设备）会发送一个中断，通知操作系统有键按下。中断本质上是一种特殊的电信号，由硬件设备发向处理器。处理器接收到中断后，会马上向操作系统反映此信号的到来，然后就由OS负责处理这些新到来的数据。硬件设备生成中断的时候并不考虑与处理器的时钟同步——换句话说就是中断随时可以产生。因此，内核随时可能因为新到来的中断而被打断。

从物理学的角度看，中断是一种电信号，由硬件设备生成，并直接送入中断控制器的输入引脚上。然后再由中断控制器向处理器发送相应的信号。处理器一经检测到此信号，便中断自己的当前工作转而处理中断。此后，处理器会通知操作系统已经产生中断，这样，操作系统就可以对这个中断进行适当的处理了。

不同的设备对应的中断不同，而每个中断都通过一个惟一的数字标识。因此，来自键盘的中断就有别于来自硬盘的中断，从而使得操作系统能够对中断进行区分，并知道哪个硬件设备产生了哪个中断。这样，操作系统才能给不同的中断提供不同的中断处理程序。

这些中断值通常被为中断请求（IRQ）线。通常IRQ都是一些数值量——例如，在PC上，IRQ0是时钟中断，而IRQ1是键盘中断。但并非所有的中断号都是这样严格定义的。例如，对于连接在PCI总线上的设备而言，中断是动态分配的。而且其他非PC的体系结构也具有动态分配可用中断的特性。重点在于特定的中断总是与特定的设备相关联，并且内核要知道这些信息。实际上，硬件发出中断是为了引起内核的关注：嗨，我有新的按键等待处理呢；读取并处理这些调皮鬼吧！

异常

在操作系统中，讨论中断就不能不提及异常。异常与中断不同，它在产生时必须考虑与处理器时钟同步。实际上，异常也常常称为同步中断。在处理器执行到由于编程失误而导致的错误指令（例如被0除）的时候，或者是在执行期间出现特殊情况（例如缺页），必须靠内核来处理的时候，处理器就会产生一个异常。因为许多处理器体系结构处理异常与处理中断的方式类似，因此，内核对它们的处理也很类似。本章对中断（由硬件产生的异步中断）的讨论，大部分也适合于异常（由处理器本身产生的同步中断）。

我们已经熟悉一种异常：从前一章中看到，在x86体系结构上如何通过软中断实现系统调用，那就是陷入内核，然后引起一种特殊的异常——系统调用处理程序异常。你将会看到，中断的工作方式与之类似，其差异只在于中断是由硬件而不是软件引起的。

6.2 中断处理程序

在响应一个特定中断的时候，内核会执行一个函数，该函数叫做*中断处理程序*（interrupt handler）或*中断服务例程*（interrupt service routine，ISR）。产生中断的每个设备都有一个（中断处理程序通常不是和特定设备关联，而是和特定中断关联的，也就是说，如果一个设备可以产生多种不同的中断，那么该设备就可以对应多个中断处理程序，相应的，该设备的驱动程序也就需要准备多个这样的函数。——译者注）相应的中断处理程序。例如，由一个函数专门处理来自系统时钟的中断，而另外一个函数专门处理由键盘产生的中断。一个设备的中断处理程序是它设备驱动程序（driver）的一部分——设备驱动程序是用于对设备进行管理的内核代码。

在Linux中，中断处理程序看起来就是普普通通的C函数。只不过这些函数必须按照特定的类型声明，以便内核能够以标准的方式传递处理程序的信息，在其他方面，它们与一般的函数看起来别无二致。中断处理程序与其他内核函数的真正区别在于：中断处理程序是被内核调用来响应中断的，而它们运行于我们称之为中断上下文的特殊上下文中。关于中断上下文，我们将在后面讨论。

中断可能随时发生，因此中断处理程序也就随时可能执行。所以必须保证中断处理程序能够快速执行，这样才能保证尽可能快地恢复中断代码的执行。因此，尽管对硬件而言，迅速对其中断进行服务非常重要，但对系统的其他部分而言，让中断处理程序在尽可能短的时间内完成运行也同样重要。

即使是最精简版的中断服务程序，它也要与硬件进行交互，告诉该设备中断已被接收。嗨，硬件，我听到了呀，现在，你回去工作吧！但通常我们不能像这样给中断服务程序随意减负，相反，我们要靠它完成大量的其他工作。举一个例子，我们可以考虑一下网络设备的中断处理程序面临的挑战。该处理程序除了要对硬件应答，还要把来自硬件的网络数据包拷贝到内存，对其进行处理后再交给合适的协议栈或应用程序。显而易见，这种工作量不会太小，尤其对于如今的千兆比特和万兆比特以太网卡而言。

上半部与下半部的对比

又想程序运行得快，又想程序完成的工作量多，这两个目的显然有所抵触。鉴于两个目的之间存在不可调和的矛盾，所以我们一般把中断处理切为两个部分或两半。中断处理程序是上半部

(top half)——接收到一个中断，它就立即开始执行，但只做有严格时限的工作，例如对接收的中断进行应答或复位硬件，这些工作都是在所有中断被禁止的情况下完成的。能够被允许稍后完成的工作会推迟到下半部（bottom half）去。此后，在合适的时机，下半部会被开中断执行。Linux提供了实现下半部的各种机制，下一章会讨论这些机制。

让我们考察一下上半部和下半部分割的例子，还是以我们的老朋友——网卡作为实例。当网卡接收流入网络的数据包时，需要通知内核数据包到了。网卡需要立即完成这件事，从而优化网络的吞吐量和传输周期，以避免超时。因此，网卡立即发出中断：嗨，内核，我这里有最新数据包了。内核通过执行网卡已注册的中断处理程序来做出应答。

中断开始执行，应答硬件，拷贝最新的网络数据包到内存，然后读取网卡更多的数据包。这些都是重要、紧迫而又与硬件相关的工作。处理和操作数据包的其他工作在随后的下半部中进行。本章考察上半部，下一章关注下半部。

6.3 注册中断处理程序

中断处理程序是管理硬件的驱动程序的组成部分。每一设备都有相关的驱动程序，如果设备使用中断（大部分设备如此），那么相应的驱动程序就注册一个中断处理程序。

驱动程序可以通过下面的函数注册并激活一个中断处理程序，以便处理中断：

```
/* request_irq: 分配一条给定的中断线 */
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long irqflags,
                const char * devname,
                void *dev_id)
```

第一个参数irq表示要分配的中断号。对某些设备，如传统PC设备上的系统时钟或键盘，这个值通常是预先定死的。而对于大多数其他设备来说，这个值要么是可以通过探测获取，要么可以通过编程动态确定。

第二个参数handler是一个指针，指向处理这个中断的实际中断处理程序。只要操作系统一接收到中断，该函数就被调用。要注意，handler函数的原型是特定的——它接受三个参数，并有一个类型为irqreturn_t的返回值。我们将在本章随后的部分讨论这个函数。

第三个参数irqflags可以为0，也可能是下列一个或多个标志的位掩码：

- SA_INTERRUPT：此标志表明给定的中断处理程序是一个*快速中断处理程序*（fast interrupt handler）。过去，Linux将中断处理程序分为快速和慢速两种。那些可以迅速执行但调用频率可能会很高的中断服务程序，会被贴上这样的标签。通常这样做需要修改中断处理程序的行为，使它们能够尽可能快地执行。现在，加不加此标志的区别只剩下一条了：在本地处理器上，快速中断处理程序在禁止所有中断的情况下运行。这使得快速中断处理程序能够不受其他中断干扰，得以迅速执行。而默认情况下（没有这个标志），除了正运行的中断处理程序对应的那条中断线被屏蔽外，其他所有中断都是激活的。除了时钟中断外，绝大多数中断都不使用该标志。
- SA_SAMPLE_RANDOM：此标志表明这个设备产生的中断对内核熵池（entropy pool）有贡献。内核熵池负责提供从各种随机事件导出的真正的随机数。如果指定了该标志，那么来自

该设备的中断间隔时间就会作为熵填充到熵池。如果你的设备以预知的速率产生中断（比如系统定时器），或者可能受外部攻击者（例如连网设备）的影响，那么就不要设置这个标志。相反，有其他很多硬件产生中断的速率是不可预知的，所以都能成为一种较好的熵源。关于内核熵池更多的信息，请参考附录B。

- SA_SHIRQ：此标志表明可以在多个中断处理程序之间共享中断线。在同一个给定线上注册的每个处理程序必须指定这个标志；否则，在每条线上只能有一个处理程序。有关共享中断处理程序的更多信息将在下面的小节中提供。

第四个参数devname是与中断相关的设备的ASCII文本表示法。例如， PC机上键盘中断对应的这个值为“keyboard”。这些名字会被/proc/irq和/proc/interrupt文件使用，以便与用户通信，稍后我们将对此进行简短讨论。

第五个参数dev_id主要用于共享中断线。当一个中断处理程序需要释放时（稍后讨论），dev_id将提供惟一的标志信息（cookie），以便从共享中断线的诸多中断处理程序中删除指定的那一个。如果没有这个参数，那么内核不可能知道在给定的中断线上到底要删除哪一个处理程序。如果无需共享中断线，那么将该参数赋为空值（NULL）就可以了，但是，如果中断线是被共享的，那么就必须传递惟一的信息（除非设备又旧又破且位于ISA总线上，那么就必须支持共享中断）。另外，内核每次调用中断处理程序时，都会把这个指针传递给它（中断处理程序都是预先在内核进行注册的回调函数（callback function），而不同的函数位于不同的驱动程序中，所以在这些函数共享同一个中断线时，内核必须准确的为它们创造执行环境，此时就可以通过这个指针将有用的环境信息传递给它们了。——译者注）。实践中往往会通过它传递驱动程序的设备结构：这个指针是惟一的，而且有可能在中断处理程序内及设备模式中被用到。

request_irq()成功执行会返回0。如果返回非0值，就表示有错误发生，在这种情况下，指定的中断处理程序不会被注册。最常见的错误是-EBUSY，它表示给定的中断线已经在使用（或者当前用户或者你没有指定SA_SHIRQ）。

注意，request_irq()函数可能会睡眠，因此，不能在中断上下文或其他不允许阻塞的代码中调用该函数。天真地在睡眠不安全的上下文中调用request_irq()函数，是一种常见错误。造成这种错误的部分原因是为什么request_irq()会引起睡眠——这确实让人费解。在注册的过程中，内核需要在/proc/irq文件创建一个与中断对应的项。函数proc_ mkdir()就是用来创建这个新的procfs项的。proc_makedir()通过调用函数proc_create()对这个新的profs项进行设置，而proc_create()会调用函数kmalloc()来请求分配内存。我们在第11章中会看到，函数kmalloc()是可以睡眠的。看清楚了，你的程序就是跑到那里小憩去了！

无论如何，细节是讲的足够了。在一个驱动程序中请求一个中断线，并在通过request_irq()安装中断处理程序：

```
if (request_irq(irqn,my_interrupt,SA_SHIRQ,"my_device",dev)){
        printk(KERN_ERR "my_device:cannot register IRQ %d\n",irqn);
        return -EIO;
}
```

在这个例子中，irqn是请求的中断线，my_interrupt是中断处理程序，中断线可以共享，设备名为“my_device”，而且我们通过dev_id传递dev结构体。如果请求失败，那么这段代码将打印出

一个错误并返回。如果调用返回0，则说明处理程序已经成功安装。此后，处理程序就会在响应该中断时被调用。有一点很重要，初始化硬件和注册中断处理程序的顺序必须正确，以防止中断处理程序在设备初始化完成之前就开始执行。

释放中断处理程序

卸载驱动程序时，需要注销相应的中断处理程序，并释放中断线。可以调用`void free_irq(unsigned int irq,void *dev_id)`来释放中断线。

如果指定的中断线不是共享的，那么，该函数删除处理程序的同时将禁用这条中断线。如果中断线是共享的，则仅删除dev_id所对应的处理程序，而这条中断线本身只有在删除了最后一个处理程序时才会被禁用。由此可以看出为什么惟一的dev_id如此重要。对于共享的中断线，需要一个惟一的信息来区分其上面的多个处理程序，并让free_irq()仅仅删除指定的处理程序。不管在哪种情况下（共享或不共享），如果dev_id非空，它都必须与需要删除的处理程序相匹配。

必须从进程上下文中调用free_irq()。

表6-1 中断注册方法表

函 数	描 述
request_irq()	在给定的中断线上注册一给定的中断处理程序
free_irq()	如果在中断线上没有中断处理程序，则注销给定的处理程序，并禁用其中断线

6.4 编写中断处理程序

以下是一个典型的中断处理程序声明：

```
static irqreturn_t intr_handler(int irq, void *dev_id,struct pt_regs *regs)
```

注意，它的类型与request_irq()参数中handler所要求的参数类型相匹配。第一个参数irq就是这个处理程序要响应的中断的中断线号。如今，这个参数已经没有太大用处了，可能只是在打印日志信息时会用到。而在2.0以前的内核中，由于没有dev_id这个参数，必须通过irq才能区分使用相同驱动程序因而也使用相同的中断处理程序的多个设备。例如，考虑具有多个相同类型硬盘驱动控制器的计算机。

第二个参数dev_id是一个通用指针，它与在中断处理程序注册时传递给request_irq()的参数dev_id必须一致。如果该值有惟一确定性（建议采用这样的值，以便支持共享），那么它就相当于一个cookie，可以用来区分共享同一中断处理程序的多个设备。另外dev_id也可能指向中断处理程序使用的一个数据结构。因为对每个设备而言，设备结构都是惟一的，而且可能在中断处理程序中也用得到，因此，也通常会把设备结构传递给dev_id。

最后一个参数regs是一个指向结构的指针，该结构包含处理中断之前处理器的寄存器和状态。除了调试的时候，它们很少使用到。事实上，目前开发者的兴趣显示这个参数可能不再使用。考虑到现有的中断处理程序中，reg的使用越来越少，因此应该忽略它。

中断处理程序的返回值是一个特殊类型：irqreturn_t。中断处理程序可能返回两个特殊的值：IRQ_NONE和IRQ_HANDLED。当中断处理程序检测到一个中断，但该中断对应的设备并不是在注册处理函数期间指定的产生源时，返回IRQ_NONE；当中断处理程序被正确调用，且确实是它

所对应的设备产生了中断时，返回IRQ_HANDLED。另外，也可以使用宏IRQ_RETVAL(x)。如果x为非0值，那么该宏返回IRQ_HANDLED；否则，返回IRQ_NONE。利用这些特殊的值，内核可以知道设备发出的是否是一种虚假的（未请求）中断。如果给定中断线上所有中断处理程序返回的都是IRQ_NONE，那么，内核就可以检测到出了问题。注意，irqreturn_t这个返回类型实际上就是一个int型。之所以使用这些特殊值是为了与早期的内核保持兼容——2.6版之前的内核并不支持这种特性，中断处理程序只需返回void就行了。如果要在2.4或更早的内核上使用这样的驱动程序，只需简单地将typedef irqreturn_t 改为void，屏蔽掉此特性，并给noops定义不同的返回值，其他用不着做什么大的修改。

中断处理程序通常会标记为static，因为它从来不会被别的文件中的代码直接调用。

中断处理程序扮演什么样的角色要取决于产生中断的设备和该设备为什么要发送中断。即使其他什么工作也不做，绝大部分的中断处理程序至少需要知道产生中断的设备，告诉它已经收到中断了。对于复杂一些的设备，可能还需要在中断处理程序中发送和接收数据，以及执行一些扩充的工作。如前所述，应尽可能将扩充的工作推给下半部处理程序，这点将在下一章中进行讨论。

重入和中断处理程序

Linux中的中断处理程序是无需重入的。当一个给定的中断处理程序正在执行时，相应的中断线在所有处理器上都会被屏蔽掉，以防止在同一中断线上接收另一个新的中断。通常情况下，所有其他的中断都是打开的，所以这些不同中断线上的其他中断都能被处理，但当前中断线总是被禁止的。由此可以看出，同一个中断处理程序绝对不会被同时调用以处理嵌套的中断。这极大地简化了中断处理程序的编写。

6.4.1 共享的中断处理程序

共享的处理程序与非共享的处理程序在注册和运行方式上比较相似，但差异主要有以下三处：

- request_irq()的参数flags必须设置SA_SHIRQ标志。
- 对每个注册的中断处理程序来说，dev_id参数必须惟一。指向任一设备结构的指针就可以满足这一要求；通常会选择设备结构，因为它是惟一的，而且中断处理程序可能会用到它。不能给共享的处理程序传递NULL值。
- 中断处理程序必须能够区分它的设备是否真的产生了中断。这既需要硬件的支持，也需要处理程序中有相关的处理逻辑。如果硬件不支持这一功能，那中断处理程序肯定会束手无策，它根本没法知道到底是与它对应的设备发出了这个中断，还是共享这条中断线的其他设备发出了这个中断。

所有共享中断线的驱动程序都必须满足以上要求。只要有任何一个设备没有按规则进行共享，那么中断线就无法共享了。指定SA_SHIRQ标志以调用request_irq()时，只有在以下两种情况下才可能成功：中断线当前未被注册，或者在该线上的所有已注册处理程序都指定了SA_SHIRQ。注意，在这一点上2.6与以前的内核是不同的，共享的处理程序可以混用SA_ INTERRUPT。

内核接收一个中断后，它将依次调用在该中断线上注册的每一个处理程序。因此，一个处理程序必须知道它是否应该为这个中断负责。如果与它相关的设备并没有产生中断，那么处理程序应该立即退出。这需要硬件设备提供状态寄存器（或类似机制），以便中断处理程序进行检查。毫

无疑问，大多数硬件都提供这种功能。

6.4.2 中断处理程序实例

让我们考察一个实际的中断处理程序，它来自RTC（real_time clock）驱动程序，可以在drivers/char/rtc.c中找到。很多机器（包括PC）都可以找到RTC。它是一个从系统定时器中独立出来的设备，用于设置系统时钟，提供报警器（alarm）或周期性的定时器。对大多数体系结构而言，系统时钟的设置，通常只需要向某个特定的寄存器或I/O地址上写入想要的时间就可以。然而报警器或周期性定时器通常就得靠中断来实现。这种中断与生活中的闹铃差不多：中断发出时，报警器或定时器就会启动。

RTC驱动程序装载时，rtc_init()函数会被调用，对这个驱动程序进行初始化。它的职责之一就是注册中断处理程序：

```
/*对RTC_IRQ 注册rtc_interrupt */
if (request_irq(RTC_IRQ,rtc_interrupt,SA_INTERRUPT,"rtc",NULL){
        printk(KERN_ERR "rtc:cannot register IRQ %d\n",RTC_IRQ);
        return -EIO;
}
```

从中我们看到，中断线号由RTC_IRQ指定。这是一个预处理定义，为给定体系结构指定RTC中断。例如，在PC上，RTC总是位于IRQ8。第二个参数是我们的中断处理程序rtc_interrupt，在它执行时要禁止所有的中断，这要归功于SA_INTERRUPT标志。从第四个参数我们看到，驱动程序的名称为“rtc”。因为这个设备不允许共享中断线，且处理程序没有用到什么特殊的值，因此给dev_id的值是NULL。

最后要展示的是处理程序本身：

```
/*
 * 很小的中断处理程序，运行时设置了SA_INTERRUPT，但是，
 *可能与set_rtc_mmss() 调用产生冲突（rtc中断和时钟中断可以轻易地
 *在两个不同的CPU上同时运行）。因此，我们需要以rtc_lock 自旋锁串行
 * 访问这一芯片，自旋锁的实现在时钟代码中，与体系结构相关。
 * (关于set_rtc_mmss()函数，参见 ./arch/XXXX/kernel/time.c)
 */
static irqreturn_t rtc_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        /*
         *可以是报警器中断、更新近完成的中断或周期性中断。
         *我们把状态保存在rtc_irq_data的低字节中，
         *而把从最后一次读之后所接收的中断号保存在其余字节中
         */

        spin_lock (&rtc_lock);

        rtc_irq_data += 0x100;
        rtc_irq_data &= ~0xff;
        rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);
```

```
if (rtc_status & RTC_TIMER_ON)
    mod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);

spin_unlock (&rtc_lock);

/*
 * 现在执行其余的操作
 */
spin_lock(&rtc_task_lock);
if (rtc_callback)
        rtc_callback->func(rtc_callback->private_data);
spin_unlock(&rtc_task_lock);
wake_up_interruptible(&rtc_wait);

        kill_fasync (&rtc_async_queue, SIGIO, POLL_IN);

        return IRQ_HANDLED;
}
```

只要计算机一接收到RTC中断，就会调用这个函数。首先要注意的是使用了自旋锁——第一次调用是为了保证rtc_irq_data不被SMP机器上的其他处理器同时访问，第二次调用避免rtc_callback出现相同的情况。锁机制在第9章中进行讨论。

rtc_irq_data变量是无符号长整数，存放有关RTC的信息，每次中断时都会更新以反映中断的状态。

接下来，如果设置了RTC周期性定时器，就要通过函数mod_timer()对其更新。定时器在第10章中进行讨论。

代码的最后一部分要通过设置自旋锁进行保护，它会执行一个可能被预先设置好的回调函数。RTC驱动程序允许注册一个回调函数，并在每个RTC中断到来时执行。

最后，这个函数会返回IRQ_HANDLED，表明已经正确地完成了对此设备的操作。因为这个中断处理程序不支持共享，而且RTC也没有什么用来测试虚假中断的机制，所以该处理程序总是返回IRQ_HANDLED。

6.5 中断上下文

当执行一个中断处理程序或下半部时，内核处于中断上下文（interrput context）中。让我们先回忆一下进程上下文。进程上下文是一种内核所处的操作模式，此时内核代表进程执行——例如，执行系统调用或运行内核线程。在进程上下文中，可以通过current宏关联当前进程。此外，因为进程是以进程上下文的形式连接到内核中的，因此，在进程上下文可以睡眠，也可以调用调度程序。

与之相反，中断上下文和进程并没有什么瓜葛。与current宏也是不相干的（尽管它会指向被中断的进程）。因为没有进程的背景，所以中断上下文不可以睡眠——否则又怎能再对它重新调度呢？因此，不能从中断上下文中调用某些函数。如果一个函数睡眠，就不能在你的中断处理程序中使用它——这是对什么样的函数可以在中断处理程序中使用的限制。

中断上下文具有较为严格的时间限制，因为它打断了其他代码。中断上下文中的代码应当迅

速简洁，尽量不要使用循环去处理繁重的工作。有一点非常重要，请永远牢记：中断处理程序打断了其他的代码（甚至可能是打断了在其他中断线上的另一中断处理程序）。正是因为这种异步执行的特性，所以所有的中断处理程序必须尽可能的迅速、简洁。尽量把工作从中断处理程序中分离出来，放在下半部来执行，因为下半部可以在更合适的时间运行。

中断处理程序栈的设置是一个配置选项。曾经，中断处理程序并不具有自己的栈。相反，它们共享所中断进程的内核栈[⊖]。内核栈的大小是两页，具体地说，在32位体系结构上是8KB，在64位体系结构上是16KB。因为在这种设置中，中断处理程序共享别人的堆栈，所以它们在栈中获取空间时必须非常节省。当然，内核栈本来就很有限，因此，所有的内核代码都应该谨慎地利用它。

在2.6早期的内核中，增加了一个选项，把栈的大小从两页减到一页，也就是在32位的系统上只提供4KB的栈。这就减轻了内存的压力，因为系统中每个进程原先都需要两页不可换出的内核内存。为了应对栈大小的减少，中断处理程序拥有了自己的栈，每个处理器一个，大小为一页。这个栈就称为中断栈，尽管中断栈的大小是原先共享栈的一半，但平均可用栈空间大得多，因为中断处理程序把这一整页拥为己有。

你的中断处理程序不必关心栈如何设置，或者内核栈的大小是多少。总而言之，尽量节约内核栈空间。

6.6 中断处理机制的实现

中断处理系统在Linux中的实现是非常依赖于体系结构的，想必你对此不会感到特别惊讶。实现依赖于处理器、所使用的中断控制器的类型、体系结构的设计及机器本身。

图6-1是中断从硬件到内核的路由。设备产生中断，通过总线把电信号发送给中断控制器。如果中断线是激活的（它们是允许被屏蔽的），那么中断控制器就会把中断发往处理器。在大多数体系结构中，这个工作就是通过电信号给处理器的特定管脚发送一个信号。除非在处理器上禁止该中断，否则，处理器会立即停止它正在做的事，关闭中断系统，然后跳到内存中预定义的位置开始执行那里的代码。这个预定义的位置是由内核设置的，是中断处理程序的入口点。

在内核中，中断的旅程开始于预定义入口点，这类似于系统调用通过预定义的异常句柄进入内核。对于每条中断线，处理器都会跳到对应的一个惟一的位置。这样，内核就可知道所接收中断的IRQ号了。初始入口点只是在栈中保存这个号，并存放当前寄存器的值（这些值属于被中断的任务）；然后，内核调用函数do_IRQ()。从这里开始，大多数中断处理代码是用C写的——但它们依然与体系结构相关。

do_IRQ()的声明如下：

```
unsigned int do_IRQ(struct pt_regs regs)
```

因为C的调用惯例是要把函数参数放在栈的顶部，因此pt_regs结构包含原始寄存器的值，这些值是以前在汇编入口例程中保存在栈中的。中断的值也会得以保存，所以，do_IRQ()可以将它提取出来。x86的代码为：

```
int irq=regs.orig_eax & 0xff;
```

⊖ 进程总是在运行。实在没有可调度来运行时，就运行空闲的任务。

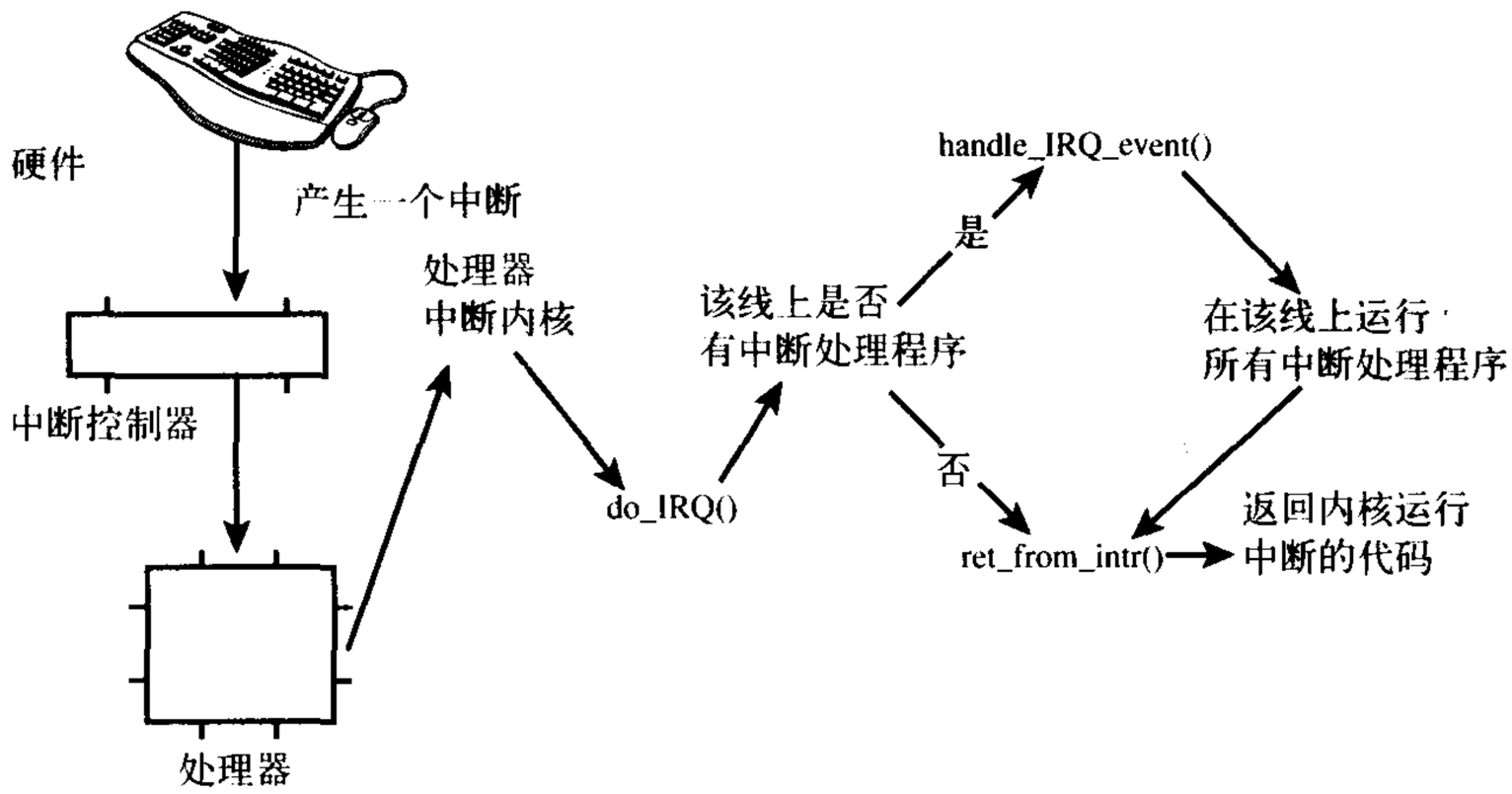


图6-1 中断从硬件到内核的路由

计算出中断号后，do_IRQ()对所接收的中断进行应答，禁止这条线上的中断传递。在普通的PC机器上，这些操作是由mask_and_ack_8259A()来完成的，该函数由do_IRQ()调用。

接下来，do_IRQ()需要确保在这条中断线上有一个有效的处理程序，而且这个程序已经启动但是当前并没有执行。如果这样的话，do_IRQ()就调用handle_IRQ_event()来运行为这条中断线所安装的中断处理程序。在x86上，handle_IRQ_event()为：

```
asmlinkage int handle_IRQ_event(unsigned int irq,struct pt_regs *regs, struct
                                irqaction *action)
{
        int status = 1;
        int retval = 0;

        if (!(action->flags & SA_INTERRUPT))
                local_irq_enable();

        do {
                status |= action->flags;
                retval |= action->handler(irq, action->dev_id, regs);
                action = action->next;
        } while (action);

        if (status & SA_SAMPLE_RANDOM)
                add_interrupt_randomness(irq);

        local_irq_disable();

        return retval;
}
```

首先，因为处理器禁止中断，这里要把它们打开，就必须在处理程序注册期间指定SA_INTERRUPT标志。回想一下，SA_INTERRUPT表示处理程序必须在中断禁止的情况下运行。接下来，每个潜在的处理程序在循环中依次执行。如果这条线不是共享的，第一次执行后就退出循

环。否则，所有的处理程序都要被执行。之后，如果在注册期间指定了SA_SAMPLE_ RANDOM标志，则还要调用函数add_interrupt_randomness()。这个函数使用中断间隔时间为随机数产生器产生熵。在附录B中有更多关于内核随机数产生器的信息。最后，再将中断禁止（do_IRQ()期望中断一直是禁止的），函数返回。回到do_IRQ()，该函数做清理工作并返回到初始入口点，然后再从这个入口点跳到函数ret_from_intr()。

ret_from_intr()例程类似于初始入口代码，以汇编编写。这个例程检查重新调度是否正在挂起（回想一下第4章，这意味着设置了need_resched）。如果重新调度正在挂起，而且内核正在返回用户空间（也就是说，中断了用户进程），那么，schedule()被调用。如果内核正在返回内核空间（也就是说，中断了内核本身），只有在preempt_count为0时，schedule()才会被调用（否则，抢占内核便是不安全的）。在schedule()返回之后，或者如果没有挂起的工作，那么，原来的寄存器被恢复，内核恢复到曾经中断的点。

在x86上，初始的汇编例程位于arch/i386/kernel/entry.S，C方法位于arch／i386／kernel/irq.c。其他所支持的结构与此类似。

/proc/interrupts

procfs是一个虚拟文件系统，它只存在于内核内存，一般安装于/proc目录下。在procfs中读写文件都要调用内核函数，这些函数模拟从真实文件中读或写。与此相关的例子是/proc/interrupts文件，该文件存放的是系统中与中断相关的统计信息。下面是从单处理器PC上输出的信息：

```
           CPU0
  0:    3602371    XT-PIC    timer
  1:       3048    XT-PIC    i8042
  2:          0    XT-PIC    cascade
  4:    2689466    XT-PIC    uhci-hcd,eth0
  5:          0    XT-PIC    EMU10k1
 12:      85077    XT-PIC    uhci-hcd
 15:      24571    XT-PIC    aic7xxx
NMI:          0
LOC:    3602236
ERR:          0
```

第1列是中断线。在这个系统中，现有的中断号为0~2、4、5、12及15。这里没有显示没有安装处理程序的中断线。第2列是一个接收中断数目的计数器。事实上，系统中的每个处理器都存在这样的列，但是，这个机器只有一个处理器。我们看到，时钟中断已接收3,602,371次中断[⊖]，这里，声卡（EMU10K1）没有接收一次中断（这表示机器启动以来还没有使用它）。第3列是处理这个中断的中断控制器。XT-PIC对应于标准的可编程中断控制器。在具有I/O APIC的系统上，大多数中断会列出IO-APIC-level或IO-APIC-edge，作为自己的中断控制器。最后一列是与这个中断相关的设备名字。这个名字是通过参数devname提供给函数request_irq( )的，前面已讨论过了。如果中断是共享的（例子中的4号中断就是这种情况），则这条中断线上注册的所有设备都会列出来。

⊖ 作为一个练习，读过第10章后，你能在知道时钟产生的中断次数的情况下说出系统已经工作了多久了吗（根据HZ值）？

对于想深入探究profs内部的人来说，profs代码位于fs/proc中。不必惊讶，提供/proc/interrupts的函数是与体系结构相关的，叫做show_interrupts()。

6.7 中断控制

Linux内核提供了一组接口用于操作机器上的中断状态。这些接口为我们提供了能够禁止当前处理器的中断系统，或屏蔽掉整个机器的一条中断线的能力，这些例程都是与体系结构相关的，可以在<asm/system.h>和 <asm/irq.h>中找到。本章稍后给出的表6-2是接口的完整列表。

一般来说，控制中断系统的原因归根结底是需要提供同步。通过禁止中断，可以确保某个中断处理程序不会抢占当前的代码。此外，禁止中断还可以禁止内核抢占。然而，不管是禁止中断还是禁止内核抢占，都没有提供任何保护机制来防止来自其他处理器的并发访问。Linux支持多处理器，因此，内核代码一般都需要获取某种锁，防止来自其他处理器对共享数据的并发访问。获取这些锁的同时也伴随着禁止本地中断。锁提供保护机制，防止来自其他处理器的并发访问，而禁止中断提供保护机制，则是防止来自其他中断处理程序的并发访问。第8章和第9章着重讨论同步的各种问题及其对策。

因此，必须理解内核中断的控制接口。

6.7.1 禁止和激活中断

用于禁止当前处理器（仅仅是当前处理器）上的本地中断，随后又激活它们的语句为：

```
local_irq_disable();
/*禁止中断..*/
local_irq_enable();
```

这两个函数通常以单个汇编指令来实现（当然，这取决于体系结构）。实际上，在x86中，local_irq_discable()仅仅是cli指令，而local_irq_enable()只不过是sti指令。对于非x86的黑客而言，cli和sti分别是对clear和set允许中断（allow interrupt）标志的汇编调用。换句话说，在发出中断的处理器上，它们将禁止和激活中断的传递。

如果在调用local_irq_discable()例程之前已经禁止了中断，那么该例程往往会带来潜在的危险；同样相应的local_irq_enable()例程也存在潜在危险，因为它将无条件地激活中断，尽管这些中断可能在开始时就是关闭的。所以我们需要一种机制把中断恢复到以前的状态而不是简单地禁止或激活。内核普遍关心这点，是因为内核中一个给定的代码路径既可以在中断激活的情况下达到，也可以在中断禁止的情况下达到，这取决于具体的调用链。例如，想像一下前面的代码片段是一个大函数的组成部分。这个函数被另外两个函数调用：其中一个函数禁止中断，而另一个函数不禁止中断。因为随着内核的不断增长，要想知道到达这个函数的所有代码路径将变得越来越困难，因此，在禁止中断之前保存中断系统的状态会更加安全一些。相反，在准备激活中断时，只需把中断恢复到它们原来的状态。

```
unsigned long flags;

local_irq_save(flags);              /* 禁止中断..*/
/*    ...          */
local_irq_restore(flags) ;          /*中断被恢复到它们原来的状态..*/
```

这些方法至少部分要以宏的形式实现，因此表面上flags参数(这些参数必须定义为unsigned long类型)是以值传递的。该参数包含具体体系结构的数据，也就是包含中断系统的状态。至少有一种体系结构把栈信息与值相结合（SPARC），因此flags不能传递给另一个函数（特别是它必须驻留在同一栈帧中）。基于这个原因，对local_irq_save()的调用和对local_irq_restore()的调用必须在同一个函数中进行。

前面的所有函数既可以在中断中调用，也可以在进程上下文中调用。

不再使用全局的cli()

以前的内核中提供了一种“能够禁止系统中所有处理器上的中断”方法。而且，如果另一个处理器调用这个方法，那么它就不得不等待，直到中断重新被激活才能继续执行。这个函数就是cli()，相应的激活中断函数为sti()——虽然适用于所有体系结构，但完全以x86为中心。这些接口在2.5版本开发期间被取消了，相应地，所有的中断同步现在必须结合使用本地中断控制和自旋锁，（在第9章中进行讨论）。这就意味着，为了确保对共享数据的互斥访问，以前代码仅仅需要通过全局禁止中断达到互斥，而现在则需要多做些工作了。

以前，驱动程序编写者可能假定在他们的中断处理程序中，任何访问共享数据地方都可以使用cli()提供互斥访问。cli()调用将确保没有其他的中断处理程序（因而只有它们特定的处理程序）会运行。此外，如果另一个处理器进入了cli保护区，那么它不可能继续运行，直到原来的处理器退出它们的cli()保护区，并调用了sti()后才能继续运行。

取消全局cli()有不少优点。首先，强制驱动程序编写者实现真正的加锁。要知道具有特定目的细粒度锁比全局锁要快许多，而且也完全吻合cli()的使用初衷。其次，这也使得很多代码更具流线型，避免了代码的成簇布局。所以由此得到的中断系统更简单也更易于理解。

6.7.2 禁止指定中断线

在前一节中，我们看到了禁止整个处理器上所有中断的函数。在某些情况下，只禁止整个系统中一条特定的中断线就够了。这就是所谓的*屏蔽掉*（masking out）一条中断线。作为例子，你可能想在对中断的状态操作之前禁止设备中断的传递。为此，Linux提供了四个接口：

```
void disable_irq(unsigned int irq);
void disable_irq_nosync(unsigned int irq);
void enable_irq(unsigned int irq);
void synchronize_irq(unsigned int irq);
```

前两个函数禁止中断控制器上指定的中断线，即禁止给定中断向系统中所有处理器的传递。另外，函数只有在当前正在执行的所有处理程序完成后，disable_irq()才能返回。因此，调用者不仅要确保不在指定线上传递新的中断，同时还要确保所有已经开始执行的处理程序已全部退出。函数disable_irq_nosync()不会等待当前中断处理程序执行完毕。

函数synchronize_riq()等待一个特定的中断处理程序的退出。如果该处理程序正在执行，那么该函数必须退出后才能返回。

对这些函数的调用可以嵌套。但要记住在一条指定的中断线上，对disable_irq()或disable_irq_nosync()的每次调用，都需要相应地调用一次enable_irq()。只有在对enable_irq()完成最后一次

调用后，才真正重新激活了中断线。例如，如果disable_irq()被调用了两次，那么直到第二次调用enable_irq()后，才能真正地激活中断线。

其中有三个函数可以从中断或进程上下文中调用，而且不会睡眠。但如果从中断上下文中调用，就要特别小心！例如，当你正在处理一条中断线时，并不想激活它（回想当某个处理程序的中断线正在被处理时，它被屏蔽掉）。

禁止多个中断处理程序共享的中断线是不合适的。禁止中断线也就禁止了这条线上所有设备的中断传递。因此，用于新设备的驱动程序应该倾向于不使用这些接口[⊖]。根据规范，PCI设备必须支持中断线共享，因此，它们根本不应该使用这些接口。所以，disable_irq()及其相关函数在老式传统设备（如PC并口）的驱动程序中更容易被找到。

6.7.3 中断系统的状态

通常有必要了解中断系统的状态（例如中断是禁止的还是激活的），或者你当前是否正处于中断上下文的执行状态中。

宏irqs_disable()定义在<asm/system.h>中。如果本地处理器上的中断系统被禁止，则它返回非0，否则，返回0。

在<asm/hardirq.h>中定义的两个宏提供一个用来检查内核的当前上下文的接口，它们是：

```
in_interrupt()
in_irq()
```

第一个宏最有用：如果内核处于中断上下文中，它返回非0。说明内核此刻正在执行中断处理程序，或者正在执行下半部处理程序。宏in_irq()只有在内核确实正在执行中断处理程序时才返回非0。

通常情况下，你要检查自己是否处于进程上下文中。也就是说，你希望确保自己不在中断上下文中。这种情况很常见，因为代码要做一些像睡眠这样只能从进程上下文中做的事。如果in_interrupt()返回0，则此刻内核处于进程上下文。

是的，名字有点混淆，但可以稍加区别它们的含义。表6-2是中断控制方法和其描述的摘要。

表6-2 中断控制方法的列表

函 数	说 明
local_irq_disable()	禁止本地中断传递
local_irq_enable()	激活本地中断传递
local_irq_save()	保存本地中断传递的当前状态，然后禁止本地中断传递
local_irq_restore()	恢复本地中断传递到给定的状态
disable_irq()	禁止给定中断线，并确保该函数返回之前在该中断线上没有处理程序在运行
disable_irq_nosync()	禁止给定中断线
enable_irq()	激活给定中断线
irqs_disabled()	如果本地中断传递被禁止，则返回非0；否则返回0
in_interrupt()	如果在中断上下文中，则返回非0，如果在进程上下文中，则返回0
in_irq()	如果当前正在执行中断处理程序，则返回非0，否则，返回0

⊖ 很多老式设备，尤其是ISA设备，不提供方法检测它们是否产生了中断。因为这一点，ISA的中断线常常不能共享。因为PCI规范要求中断共享，因此，现代基于PCI的设备支持中断共享。在当代计算机中，几乎所有的中断线都可以共享。

6.8 别打断我，马上结束

本章考察了中断是一种由设备使用的硬件资源异步地向处理器发信号。实际上，中断就是由硬件来打断操作系统。

大多数现代硬件都通过中断与操作系统通信。对给定硬件进行管理的驱动程序注册中断处理程序，这是为了响应并处理来自相关硬件的中断。中断过程所做的工作包括应答并重新设置硬件，从设备拷贝数据到内存以及反之，处理硬件请求，并发送新的硬件请求。

内核提供的接口包括注册和注销中断处理程序，禁止中断，屏蔽中断线，以及检查中断系统的状态。表6-2提供了这些函数的概述。

因为中断打断了其他代码的执行（进程，内核本身，甚至其他中断处理程序），它们必须赶快执行完。但通常是还有很多工作要做。为了在大量的工作与必须快速执行之间求得一种平衡，内核把处理中断的工作分为两半。中断处理程序，也就是上半部在本章讨论。现在，让我们考察下半部。

第⑦章

下半部和推后执行的工作

在上一章中，我们研究了内核为处理中断而提供的中断处理程序机制。中断处理程序是内核中很有用的——实际上也是必不可少的——部分。但是，由于本身存在一些局限，所以它只能完成整个中断处理流程的上半部分。这些局限包括：

- 中断处理程序以异步方式执行并且它有可能会打断其他重要代码(甚至包括其他中断处理程序)的执行。因此，为了避免被打断的代码停止时间过长，中断处理程序应该执行得越快越好。
- 如果当前有一个中断处理程序正在执行，在最好的情况下(如果设置了SA_INTERRUPT)，与该中断同级的其他中断会被屏蔽，在最坏的情况下，当前处理器上所有其他中断都会被屏蔽。因此，仍应该让它们执行得越快越好。
- 由于中断处理程序往往需要对硬件进行操作，所以它们通常有很高的时限要求。
- 中断处理程序不在进程上下文中运行，所以它们不能阻塞。这限制了它们所做的事情。

现在，为什么中断处理程序只能作为整个硬件中断处理流程一部分的原因就很明显了。我们必须有一个快速、异步、简单的处理程序负责对硬件做出迅速响应并完成那些时间要求很严格的操作。中断处理程序很适合于实现这些功能，可是，对于那些其他的、对时间要求相对宽松的任务，就应该推后到中断被激活以后再去运行。

这样，整个中断处理流程就被分为了两个部分，或叫两半。第一个部分是中断处理程序（上半部），就像我们在上一章讨论的那样，内核通过对它的异步执行完成对硬件中断的即时响应。在本章中，我们要研究的是中断处理流程中的另外那一部分，下半部（bottom half）。

7.1 下半部

下半部的任务就是执行与中断处理密切相关但中断处理程序本身不执行的工作。在理想的情况下，最好是中断处理程序将所有工作都交给下半部分执行，因为我们希望在中断处理程序中完成的工作越少越好（也就是越快越好）。我们期望中断处理程序能够尽可能快地返回。

但是，中断处理程序注定要完成一部分工作。例如，中断处理程序几乎都需要通过操作硬件对中断的到达进行确认。有时它还会从硬件拷贝数据。因为这些工作对时间非常敏感，所以只能靠中断处理程序自己去完成。

剩下的几乎所有其他工作都是下半部执行的目标。例如，如果你在上半部中把数据从硬件拷贝到了内存，那么当然应该在下半部中处理它们。遗憾的是，并不存在严格明确的规定来说明到底什么任务应该在哪个部分中完成——如何做决定完全取决于驱动程序开发者自己的判断。尽管在理论上不存在什么错误，但轻率的实现效果往往不很理想。记住，中断处理程序会异步执行，并且在最好的情况下它也会锁定当前的中断线。因此将中断处理程序持续执行的时间缩短到最小

程度显得非常重要。对于在上半部和下半部之间划分工作，尽管不存在某种严格的规则，但还是有一些提示可供借鉴：

- 如果一个任务对时间非常敏感，将其放在中断处理程序中执行。
- 如果一个任务和硬件相关，将其放在中断处理程序中执行。
- 如果一个任务要保证不被其他中断（特别是相同的中断）打断，将其放在中断处理程序中执行。
- 其他所有任务，考虑放置在下半部执行。

当你开始尝试写自己的驱动程序的时候，读一下别人的中断处理程序和相应的下半部可能会另你受益匪浅。在决定怎样把你的中断处理流程中的工作划分到上半部和下半部中去的时候，问问自己什么必须放进上半部而什么可以放进下半部。通常，中断处理程序要执行得越快越好。

7.1.1 为什么要用下半部

理解为什么要让工作推后执行以及在什么时候推后执行非常关键。我们希望尽量减少中断处理程序中需要完成的工作量，因为在它运行的时候当前的中断线在所有处理器上都会被屏蔽。更糟糕的是如果一个处理程序是SA_INTERRUPT类型，它执行的时候会禁止所有本地中断（而且把本地中断线全局地屏蔽掉）。而缩短中断被屏蔽的时间对系统的响应能力和性能都至关重要。再加上中断处理程序要与其他程序——甚至是其他的中断处理程序——异步执行，所以很明显，我们必须尽力缩短中断处理程序的执行。解决的方法就是把一些工作放到以后去做。

但具体放到以后的什么时候去做呢？在这里，以后仅仅用来强调不是马上而已，理解这一点相当重要。下半部并不需要指明一个确切时间，只要把这些任务推迟一点，让它们在系统不太繁忙并且中断恢复后执行就可以了。通常下半部在中断处理程序一返回就会马上运行。下半部执行的关键在于当它们运行的时候，允许响应所有的中断。

不仅仅是Linux，许多操作系统也把处理硬件中断的过程分为两个部分。上半部分简单快速，执行的时候禁止一些或者全部中断。下半部分（无论具体如何实现）稍后执行，而且执行期间可以响应所有的中断。这种设计可使系统处于中断屏蔽状态的时间尽可能的短，以此来提高系统的响应能力。

7.1.2 下半部的环境

和上半部分只能通过中断处理程序实现不同，下半部可以通过多种机制实现。这些用来实现下半部的机制分别由不同的接口和子系统组成。上一章，我们了解到实现中断处理程序的方法只有一种（在Linux中，由于上半部从来都只能通过中断处理程序实现，所以它和中断处理程序可以说是等价的。——译者注）但在本章中你会发现，实现一个下半部会有许多不同的方法。实际上，在Linux发展的过程中曾经出现过多种下半部机制。让人倍受困扰的是，其中不少机制名字起得很相像，甚至还有一些机制名字起得辞不达意。这就需要专门的程序员来给下半部命名。

在本章中，我们将要讨论2.6版本的内核中的下半部机制是如何设计和实现的。同时我们还会讨论怎么在你自己编写的内核代码中使用它们。而那些过去使用的、已经废除了有一段时间的机制，由于曾经闻名遐尔，所以在相关的时候我们还是会有所提及。

最早的Linux只提供“bottom half”这种机制用于实现下半部。这个名字在那个时候毫无异义，因为当时它是将工作推后的惟一方法。这种机制也被称为“BH”，我们现在也这么叫它，以避免

和“下半部”这个通用词汇混淆。像过往的那段美好岁月中的许多东西一样，BH接口也非常简单。它提供了一个静态创建、由32个bottom half组成的链表。上半部通过一个32位整数中的一位来标识出哪个bottom half可以执行。每个BH都在全局范围内进行同步。即使分属于不同的处理器，也不允许任何两个bottom half同时执行。这种机制使用方便却不够灵活，简单却有性能瓶颈。

不久，内核开发者们就引入了任务队列（task queue）机制来实现工作的推后执行，并用它来代替BH机制。内核为此定义了一组队列。其中每个队列都包含一个由等待调用的函数组成链表。根据其所处队列的位置，这些函数会在某个时刻被执行。驱动程序可以把它们自己的下半部注册到合适的队列上去。这种机制表现得还不错，但仍不够灵活，没法代替整个BH接口。对于一些性能要求较高的子系统，像网络部分，它也不能胜任。

在2.3这个开发版本中，内核开发者引入了软中断（softirqs）（这里的软中断与第4章实现系统调用所提到的软中断（准确说该叫它软件中断）指的是不同的概念——译者注）和tasklet。如果无须考虑和过去开发的驱动程序兼容的话，软中断和tasklet可以完全代替BH接口[⊖]。软中断是一组静态定义的下半部接口，有32个，可以在所有处理器上同时执行——即使两个类型相同也可以。tasklet这一名称起得很糟糕，让人费解[⊜]，它们是一种基于软中断实现的灵活性强、动态创建的下半部实现机制。两个不同类型的tasklet可以在不同的处理器上同时执行，但类型相同的tasklet不能同时执行。tasklet其实是一种在性能和易用性之间寻求平衡的产物。对于大部分下半部处理来说，用tasklet就足够了。像网络这样对性能要求非常高的情况才需要使用软中断。可是，使用软中断需要特别小心，因为两个相同的软中断有可能同时被执行。此外，软中断还必须在编译期间就进行静态注册。与此相反，tasklet可以通过代码进行动态注册。

有些人被这些概念彻底搞糊涂了，他们把所有的下半部都当成是软件产生的中断或软中断。换句话说就是他们把软中断机制和下半部统统都叫软中断。别管他们好了。软中断和BH和tasklet并驾齐名。

在开发2.5版本的内核时，BH接口最终被弃置了，所有的BH使用者必须转而使用其他下半部接口。此外，任务队列接口也被工作队列接口取代了。工作队列是一种简单但很有用的方法，它们先对要推后执行的工作排队，稍后在进程上下文中执行它们。稍后的内容中我们再来探究它们。

综上所述，在2.6这个当前版本中，内核提供了三种不同形式的下半部实现机制：软中断、tasklet和工作队列。内核过去曾经用过的BH和任务队列接口，现在已经被湮没在记忆中了。

内核定时器

另外一个可以用于将工作推后执行的机制是内核定时器。不像本章到目前为止介绍到的所有这些机制，内核定时器把操作推迟到某个确定的时间段之后执行。也就是说，尽管本章讨论的其他机制可以把操作推后到除了现在以外的任何时间进行，但是当你必须保证在一个确定的时间段过去以后再运行时，你应该使用内核定时器。

较之本章讨论到的这些机制，定时器还有一些其他功能。有关定时器的详细内容在第10章中讨论。

⊖ 把BH转换为软中断或者tasklet 并不是轻而易举的事，因为BH是全局同步的，因此，在其执行期间假定没有其他BH在执行。但是，这种转换最终还是在内核2.5中实现了。

⊜ 它们和进程没有一点关系。可以把一个tasklet当作一个简单易用的软中断。

混乱的下半部概念

这些东西确实把人搅得很混乱，但它们其实只不过是一些起名的问题，让我们再来梳理一遍。

“下半部（bottom half）”是一个操作系统通用词汇，用于指代中断处理流程中推后执行的那一部分，之所以这样命名是因为它表示中断处理方案一半的第二部分或者下半部。在Linux中，这个词目前确实就是这个含义。所有用于实现将工作推后执行的内核机制都被称为“下半部机制”。一些人错误地把所有的下半部机制都叫做“软中断”，真是在自寻烦恼。

“下半部”这个词也指代Linux最早提供的那种将工作推后执行的实现机制。由于该机制也被叫做“BH”，所以，我们就使用它的这个名称，而让 “下半部”这个词仍然保持它通常的含义。BH机制很早以前就被反对使用了，在2.5版内核中，它被完全去除了。

当前，有三种机制可以用来实现将工作推后执行：软中断、tasklet和工作队列。tasklet通过软中断实现，而工作队列与它们完全不同。表7-1揭示了下半部机制的演化历程。

表7-1 下半部状态

下半部机制	状 态
BH	在2.5中去除
任务队列（task queue）	在2.5中去除
软中断（softirq）	从2.3开始引入
tasklet	从2.3开始引入
工作队列（work queue）	从2.5开始引入

在搞清楚了这些混乱的命名之后，让我们开始具体研究各个机制。

7.2 软中断

我们的讨论从实际的下半部实现——软中断方法开始。软中断使用得比较少；而tasklet是下半部更常用的一种形式。但是，由于tasklet是通过软中断实现的，所以我们先来研究软中断。软中断的代码位于kernel/softirq.c文件中。

7.2.1 软中断的实现

软中断是在编译期间静态分配的。它不像tasklet那样能被动态地注册或去除。软中断由softirq_action结构表示，它定义在<linux/interrupt.h>中：

```
/ *
 *本结构代表一个软中断项
 */
struct softirq_action {
        void (*action) (struct softirq_action *);        /* 待执行的函数 */
        void *data;                                      /* 传给函数的参数 */
};
```

kernel/softirq.c中定义了一个包含有32个该结构体的数组。

```
static struct softirq_action softirq_vec[32];
```

每个被注册的软中断都占据该数组的一项。因此最多可能有32个软中断。注意，这是一个定

值——注册的软中断数目的最大值没法动态改变。在当前版本的内核中，这32个项中只用到6个[⊖]。

1. 软中断处理程序

软中断处理程序action的函数原型如下：

```
void softirq_handler(struct softirq_action *)
```

当内核运行一个软中断处理程序的时候，它就会执行这个action函数，其惟一的参数为指向相应softirq_action结构体的指针。例如，如果my_softirq指向softirq_vec数组的某项，那么内核会用如下的方式调用软中断处理程序中的函数：

```
my_softirq->action(my_softirq)
```

当你看到内核把整个结构体都传递给软中断处理程序而不仅仅是传递数据值的时候，你可能会很吃惊。这个小技巧可以保证将来在结构体中加入新的域时，无须对所有的软中断处理程序都进行变动。如果需要，软中断处理程序可以方便地解析它的参数，从数据成员中提取数值。

一个软中断不会抢占另外一个软中断。实际上，惟一可以抢占软中断的是中断处理程序。不过，其他的软中断——甚至是相同类型的软中断——可以在其他处理器上同时执行。

2. 执行软中断

一个注册的软中断必须在被标记后才会执行。这被称作触发软中断（raising the softirq）。通常，中断处理程序会在返回前标记它的软中断，使其在稍后被执行。于是，在合适的时刻，该软中断就会运行。在下列地方，待处理的软中断会被检查和执行：

- 从一个硬件中断代码处返回时。
- 在ksoftirqd内核线程中。
- 在那些显式检查和执行待处理的软中断的代码中，如网络子系统中。

不管是用什么办法唤起，软中断都要在do_softirq()中执行。该函数很简单。如果有待处理的软中断，do_softirq()会循环遍历每一个，调用它们的处理程序。让我们观察一下do_softirq()经过简化后的核心部分：

```
u32 pending = softirq_pending(cpu);

if (pending) {
    struct softirq_action *h = softirq_vec;

    softirq_pending(cpu) = 0;

    do {
        if (pending &1)
            h->action(h);
        h++;
        pending >> = 1;
    }while(pending);
}
```

以上摘录的是软中断处理的核心部分。它检查并执行所有待处理的软中断，具体要做的包括：

⊖ 大部分驱动程序都使用tasklet来实现它们的下半部。我们将在下一节看到，tasklet是用软中断实现的。

1) 用局部变量pending保存softirq_pending()宏的返回值。它是待处理的软中断的32位位图——如果第n位被设置为1，那么第n位对应类型的软中断等待处理。

2) 现在待处理的软中断位图已经被保存，可以将实际的软中断位图清零了[⊖]。

3) 将指针h指向softirq_vec的第一项。

4) 如果pending的第一位被置为1，h->action(h)被调用。

5) 指针加1，所以现在它指向softirq_vec数组的第二项。

6) 位掩码pending右移一位。这样会丢弃第一位，然后让其他各位依次向右移动一个位置。于是，原来的第二位现在就在第一位的位置上了（依次类推）。

7) 现在指针h指向数组的第二项，pending位掩码的第二位现在也到了第一位上。重复执行上面的步骤。

8) 一直重复下去，直到pending变为0，这表明已经没有待处理的软中断了，我们的任务也就完成了。注意，这种检查足以保证h总指向softirq_vec的有效项，因为pending最多只可能设置32位，循环最多也只能执行32次。

7.2.2 使用软中断

软中断保留给系统中对时间要求最严格以及最重要的下半部使用。目前，只有两个子系统——网络和SCSI——直接使用软中断。此外，内核定时器和tasklet都是建立在软中断上的。如果你想加入一个新的软中断，首先应该问问自己为什么用tasklet实现不了。tasklet可以动态生成，由于它们对加锁的要求不高，所以使用起来也很方便，而且它们的性能也非常不错。当然，对于时间要求严格并能自己高效地完成加锁工作的应用，软中断会是正确的选择。

1. 分配索引

在编译期间，可以通过<linux/interrupt.h>中定义的一个枚举类型来静态地声明软中断。内核用这些从0开始的索引来表示一种相对优先级。索引号小的软中断在索引号大的软中断之前执行。

表7-2 tasklet类型列表

tasklet	优 先 级	软中断描述
HI_SOFTIRQ	0	优先级高的tasklet
TIMER_SOFTIRQ	1	定时器的下半部
NET_TX_SOFTIRQ	2	发送网络数据包
NET_RX_SOFTIRQ	3	接收网络数据包
SCSI_SOFTIRQ	4	SCSI的下半部
TASKLET_SOFTIRQ	5	tasklet

建立一个新的软中断必须在此枚举类型中加入新的项。而加入时，不能像在其他地方一样，简单地把新项加到列表的末尾。相反，必须根据希望赋予它的优先级来决定加入的位置。习惯上，HI_SOFTIRQ通常作为第一项，而TASKLET_SOFTIRQ作为最后一项。新项可能插在网络相关的

⊖ 实际上在执行此步操作时需要禁止本地中断。但在这个简化的例子中被省略了。如果中断不被屏蔽，在保存位图和清除它的间隙，可能会有一个新的软中断被唤醒（它自然也就会等待处理）。这可能会造成对此待处理的位进行不应该的清0。

那些项之后、TASKLET_SOFTIRQ之前。表7-2列举出已有的tasklet类型。

2. 注册你的处理程序

接着，在运行时通过调用open_softirq()注册软中断处理程序，该函数有三个参数：软中断的索引号、处理函数和data域存放的数值。例如网络子系统，通过以下方式注册自己的软中断：

```
open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
```

软中断处理程序执行的时候，允许响应中断，但它自己不能休眠。在一个处理程序运行的时候，当前处理器上的软中断被禁止。但其他的处理器仍可以执行别的软中断。实际上，如果同一个软中断在它被执行的同时再次被触发了，那么另外一个处理器可以同时运行其处理程序。这意味着任何共享数据——甚至是仅在软中断处理程序内部使用的全局变量——都需要严格的锁保护（下面两个章节会讨论）。这点很重要，它也是为什么tasklet更受青睐的原因。单纯地禁止你的软中断处理程序同时执行不是很理想。如果仅仅通过互斥的加锁方式来防止它自身的并发执行，那么使用软中断就没有任何意义。因此，大部分软中断处理程序都通过采取单处理器数据（仅属于某一个处理器的数据，因此根本不需要加锁）或其他一些技巧来避免显式地加锁，从而提供更出色的性能。

引入软中断的主要原因是其可扩展性。如果不需要扩展到多个处理器，那么，就使用tasklet吧。tasklet本质上也是软中断，只不过同一个处理程序的多个实例不能在多个处理器上同时运行。

3. 触发你的软中断

通过在枚举类型的列表中添加新项以及调用open_softirt()进行注册以后，新的软中断处理程序就能够运行。raise_softirq()函数可以将一个软中断设置为挂起状态，让它在下次调用do_softirq()函数时投入运行。举个例子，网络子系统可能会调用。

```
raise_softirq(NET_TX_SOFTIRQ);
```

这会触发NET_TX_SOFTIRQ软中断。它的处理程序net_tx_action()就会在内核下一次执行软中断时投入运行。该函数在触发一个软中断之前先要禁止中断，触发后再恢复回原来的状态。如果中断本来就已经被禁止了，那么可以调用另一函数raise_softirq_irqoff()，这会带来一些优化效果。如：

```
/*
 *中断已经被禁止
 */
raise_softirq_irqoff(NET_TX_SOFTIRQ);
```

在中断处理程序中触发软中断是最常见的形式。在这种情况下，中断处理程序执行硬件设备的相关操作，然后触发相应的软中断，最后退出。内核在执行完中断处理程序以后，马上就会调用do_softirq()函数。于是软中断开始执行中断处理程序留给它去完成的剩余任务。在这个例子中，“上半部”和“下半部”名字中的含义一目了然。

7.3 tasklet

tasklet是利用软中断实现的一种下半部机制。我们之前提到过，它和进程没有任何关系。tasklet和软中断在本质上很相似，行为表现也相近，但是，它的接口更简单，锁保护也要求较低。

选择到底是用软中断还是tasklet其实很简单：通常你应该用tasklet。就像我们在前面看到的，

软中断的使用者屈指可数。它只在那些执行频率很高和连续性要求很高的情况下才需要。而tasklet却有更广泛的用途。大多数情况下用tasklet效果都不错，而且它们还非常容易使用。

7.3.1 tasklet的实现

因为tasklet是通过软中断实现的，所以它们本身也是软中断。前面讨论过了，tasklet由两类软中断代表：HI_SOFTIRQ和TASKLET_SOFTIRQ。这两者之间惟一的实际区别在于HI_SOFTIRQ类型的软中断先于TASKLET_SOFTIRQ类型的软中断执行。

1. tasklet结构体

tasklet由tasklet_struct结构表示。每个结构体单独代表一个tasklet，它在<linux/interrupt.h>中定义：

```
struct tasklet_struct {
        struct tasklet_struct *next;     /* 链表中的下一个tasklet */
        unsigned long state;             /* tasklet的状态 */
        atomic_t count;                  /* 引用计数器 */
        void (*func) (unsigned long);    /* tasklet处理函数 */
        unsigned long data;              /* 给tasklet处理函数的参数 */
};
```

结构体中的func成员是tasklet的处理程序（像软中断中的action一样），data是它惟一的参数。

state成员只能在0、TASKLET_STATE_SCHED和TASKLET_STATE_RUN之间取值。TASKLET_STATE_SCHED表明tasklet已被调度，正准备投入运行，TASKLET_STATE_RUN表明该tasklet正在运行。TASKLET_STATE_RUN只有在多处理器的系统上才会作为一种优化来使用，单处理器系统任何时候都清楚单个tasklet是不是正在运行（它要么就是当前正在执行的代码，要么不是）。

count成员是tasklet的引用计数器。如果它不为0，则tasklet被禁止，不允许执行；只有当它为0时，tasklet才被激活，并且在被设置为挂起状态时，该tasklet才能够执行。

2. 调度tasklet

已调度的tasklet(等同于被触发的软中断)[⊖]存放在两个单处理器数据结构：tasklet_vec（普通tasklet）和tasklet_hi_vec（高优先级的tasklet）中。这两个数据结构都是由tasklet_struct结构体构成的链表。链表中的每个tasklet_struct代表一个不同的tasklet。

tasklet由tasklet_schedule()和tasklet_hi_schedule()函数进行调度，它们接受一个指向tasklet_struct结构的指针作为参数。两个函数非常类似（区别在于一个使用TASKLET_ SOFTIRQ而另一个用HI_SOFTIRQ）。在下一节我们将仔细研究怎么编写和使用tasklet。现在，让我们先考察一下tasklet_schedule()的细节：

1) 检查tasklet的状态是否为TASKLET_STATE_SCHED。如果是，说明tasklet已经被调度过了(有可能是一个tasklet已经被调度过但还没来得及执行，而该tasklet又被唤起了一次。——译者注)，函数立即返回。

2) 保存中断状态，然后禁止本地中断。在我们执行tasklet代码时，这么做能够保证当tasklet_schedule()处理这些tasklet时，处理器上的数据不会弄乱。

⊖ 此处又是一个命名混乱的实例。为什么软中断是唤醒而tasklet是调度呢？谁能说得清？两个词其实都是表示将此下半部设置为待执行状态以便稍后执行。

3) 把需要调度的tasklet加到每个处理器一个的tasklet_vec链表或tasklet_hi_vec链表的表头上去。

4) 唤起TASKLET_SOFTIRQ或HI_SOFTIRQ软中断，这样在下一次调用do_softirq()时就会执行该tasklet。

5) 恢复中断到原状态并返回。

在前面的小节中我们曾经提到过挂起，do_softirq()会尽可能早地在下一个合适的时机执行。由于大部分tasklet和软中断都是在中断处理程序中被设置成待处理状态，所以最近一个中断返回的时候看起来就是执行do_softirq()的最佳时机。因为TASKLET_SOFTIRQ和HI_SOFTIRQ已经被触发了，所以do_softirq()会执行相应的软中断处理程序。而这两个处理程序，tasklet_action()和tasklet_hi_action()，就是tasklet处理的核心。让我们观察它们做了什么：

1) 禁止中断，（没有必要首先保存其状态，因为这里的代码总是作为软中断被调用，而且中断总是被激活的）并为当前处理器检索tasklet_vec或tasklet_hig_vec链表。

2) 将当前处理器上的该链表设置为NULL，达到清空的效果。

3) 允许响应中断。没有必要再恢复它们回原状态，因为这段程序本身就是作为软中断处理程序被调用的，所以中断是应该被允许的。

4) 循环遍历获得链表上的每一个待处理的tasklet。

5) 如果是多处理器系统，通过检查TASKLET_STATE_RUN状态标志来判断这个tasklet是否正在其他处理器上运行。如果它正在运行，那么现在就不要执行，跳到下一个待处理的tasklet去（回忆一下，同一时间里，相同类型的tasklet只能有一个执行）。

6) 如果当前这个tasklet没有执行，将其状态标志设置为TASKLET_STATE_RUN，这样别的处理器就不会再去执行它了。

7) 检查count值是否为0，确保tasklet没有被禁止。如果tasklet被禁止了，则跳到下一个挂起的tasklet去。

8) 我们已经清楚的知道这个tasklet没有在其他地方执行，并且被我们设置成执行状态，这样它在其他部分就不会被执行，并且引用计数为0，现在可以执行tasklet的处理程序了。

9) tasklet运行完毕，清除tasklet的state域的TASKLET_STATE_RUN状态标志。

10) 重复执行下一个tasklet，直至没有剩余的等待处理的tasklet。

tasklet的实现很简单，但非常巧妙。我们可以看到，所有的tasklet都通过重复运用HI_SOFTIRQ和TASKLET_SOFTIRQ这两个软中断来实现。当一个tasklet被调度时，内核就会唤起这两个软中断中的一个。随后，该软中断会被特定的函数处理，执行所有已调度的tasklet。这个函数保证同一时间里只有一个给定类别的tasklet会被执行（但其他不同类型的tasklet可以同时执行）。所有这些复杂性都被一个简洁的接口隐藏起来了。

7.3.2 使用tasklet

大多数情况下，为了控制一个寻常的硬件设备，tasklet机制都是实现你自己的下半部的最佳选择。tasklet可以动态创建，使用方便，执行起来也还算快。此外，尽管它们的名字使人混淆，但加深你的印象：那是逗人喜爱的。

1. 声明你自己的tasklet

你既可以静态地创建tasklet，也可以动态地创建它。选择哪种方式取决于你到底是有（或者是

想要）一个对tasklet的直接引用还是间接引用。如果你准备静态地创建一个tasklet（也就是有一个它的直接引用），使用下面<linux/interrupt.h>中定义的两个宏中的一个：

```
DECLARE_TASKLET(name, func, data)
DECLARE_TASKLET_DISABLED(name, func, data);
```

这两个宏都能根据给定的名称静态地创建一个tasklet_struct结构。当该tasklet被调度以后，给定的函数func会被执行，它的参数由data给出。这两个宏之间的区别在于引用计数器的初始值设置不同。前面一个宏把创建的tasklet的引用计数器设置为0，该tasklet处于激活状态。另一个把引用计数器设置为1，所以该tasklet处于禁止状态。下面是一个例子：

```
DECLARE_TASKLET(my_tasklet, my_tasklet_handler, dev);
```

这行代码其实等价于

```
struct tasklet_struct my_tasklet = { NULL, 0, ATOMIC_INIT(0),
                                     my_tasklet_handler, dev};
```

这样就创建了一个名为my_tasklet，处理程序为tasklet_handler并且已被激活的tasklet。当处理程序被调用的时候，dev就会被传递给它。

还可以通过将一个间接引用（一个指针）赋给一个动态创建的tasklet_struct结构的方式来初始化一个tasklet：

```
tasklet_init(t, tasklet_handler, dev);   /* 动态而不是静态创建 */
```

2. 编写你自己的tasklet处理程序

tasklet处理程序必须符合规定的函数类型：

```
void tasklet_handler(unsigned long data)
```

因为是靠软中断实现，所以tasklet不能睡眠。这意味着你不能在tasklet中使用信号量或者其他什么阻塞式的函数。由于tasklet运行时允许响应中断，所以你必须做好预防工作（比如屏蔽中断然后获取一个锁），如果你的tasklet和中断处理程序之间共享了某些数据的话。两个相同的tasklet决不会同时执行，这点和软中断不同——尽管两个不同的tasklet可以在两个处理器上同时执行。如果你的tasklet和其他的tasklet或者是软中断共享了数据，你必须进行适当地锁保护。（参看第8章和第9章）。

3. 调度你自己的tasklet

通过调用tasklet_schedule()函数并传递给它相应的tasklet_struct的指针，该tasklet就会被调度以便执行：

```
tasklet_schedule(&my_tasklet);   /*把 my_tasklet 标记为挂起 */
```

在tasklet被调度以后，只要有机会它就会尽可能早地运行。在它还没有得到运行机会之前，如果有一个相同的tasklet又被调度了（这里应该是唤起的意思，在前面讲述调度流程的小节里可以看到，调度tasklet的第一个步骤就是检查是否重复，所以这里根本不会完成调度。——译者注）那么它仍然只会运行一次。而如果这时它已经开始运行了，比如说在另外一个处理器上，那么这个新的tasklet会被重新调度并再次运行。作为一种优化措施，一个tasklet总在调度它的处理器上执行——这是希望能更好地利用处理器的高速缓存。

你可以调用tasklet_disable()函数来禁止某个指定的tasklet。如果该tasklet当前正在执行，这个函数会等到它执行完毕再返回。你也可以调用tasklet_disable_nosync()函数，它也可用来禁止指定

的tasklet，不过它无须在返回前等待tasklet执行完毕。这么做往往不太安全，因为你无法估计该tasklet是否仍在执行。调用tasklet_enable()函数可以激活一个tasklet，如果希望激活DECLARE_TASKLET_DISABLED()创建的tasklet，你也得调用这个函数，如：

```
tasklet_disable(&my_tasklet);     /* tasklet现在被禁止 */

/* 我们现在毫无疑问地知道tasklet不能运行 .. */

tasklet_enable(&my_tasklet);      /*  tasklet现在被激活 */
```

你可以通过调用tasklet_kill()函数从挂起的队列中去掉一个tasklet。该函数的参数是一个指向某个tasklet的tasklet_struct的长指针。在处理一个经常重新调度它自身的tasklet的时候，从挂起的队列中移去已调度的tasklet会很有用。这个函数首先等待该tasklet执行完毕，然后再将它移去。当然，没有什么可以阻止其他地方的代码重新调度该tasklet。由于该函数可能会引起休眠，所以禁止在中断上下文中使用它。

7.3.3 ksoftirqd

每个处理器都有一组辅助处理软中断（和tasklet）的内核线程。当内核中出现大量软中断的时候，这些内核进程就会辅助处理它们。

我们前面曾经阐述过，对于软中断，内核会选择在几个特殊时机进行处理。而在中断处理程序返回时处理是最常见的。软中断被触发的频率有时可能很高（像在进行大流量的网络通信期间）。更不利的是，处理函数有时还会自行重复触发。也就是说，当一个软中断执行的时候，它可以重新触发自己以便再次得到执行（事实上，网络子系统就会这么做）。如果软中断本身出现的频率就高，再加上它们又有将自己重新设置为可执行状态的能力，那么就会导致用户空间进程无法获得足够的处理器时间，因而处于饥饿状态。而且，单纯的对重新触发的软中断采取不立即处理的策略，也无法让人接受。当软中断最初提出时，就是一个让人进退维谷的问题，亟待解决，而直观的解决方案都不理想。首先，就让我们看看两种最容易想到的直观的方案。

第一种方案是只要还有被触发并等待处理的软中断，本次执行就要负责处理，重新触发的软中断也在本次执行返回前被处理。这样做可以保证对内核的软中断采取即时处理的方式，关键在于，对重新触发的软中断也会立即处理。当负载很高的时候这样做就会出问题，此时会有大量被触发的软中断，而它们本身又会重复触发。系统可能会一直处理软中断，根本不能完成其他任务。用户空间的任务被忽略了——实际上，只有软中断和中断处理程序轮流执行，而系统的用户只能等待。只有在系统永远处于低负载的情况下，这种方案才会有理想的运行效果；只要系统有哪怕是中等程度的负载量，这种方案就无法让人满意。用户空间根本不能容忍有明显的停顿出现。

第二种方案选择不处理重新触发的软中断。在从中断返回的时候，内核和平常一样，也会检查所有挂起的软中断并处理它们。但是，任何自行重新触发的软中断都不会马上处理，它们被放到下一个软中断执行时机去处理。而这个时机通常也就是下一次中断返回的时候，这等于是说，一定得等一段时间，新的（或者重新触发的）软中断才能被执行。可是，在比较空闲的系统中，立即处理软中断才是比较好的做法。很不幸，这个方案显然又是一个时好时坏的选择。尽管它能保证用户空间不处于饥饿状态，但它却让软中断忍受饥饿的痛苦，而根本没有好好利用闲置的系统资源。

在设计软中断时，开发者要意识到需要一些折中。最终在内核中实现的方案是不会立即处理重新触发的软中断。而作为改进，当大量软中断出现的时候，内核会唤醒一组内核线程来处理这些负载。这些线程在最低的优先级上运行（nice值是19），这能避免它们跟其他重要的任务抢夺资源。但它们最终肯定会被执行，所以，这个折中方案能够保证在软中断负担很重的时候用户程序不会因为得不到处理时间而处于饥饿状态。相应的，也能保证“过量”的软中断终究会得到处理。最后，在空闲系统上，这个方案同样表现良好，软中断处理得非常迅速（因为仅存的内核线程肯定会马上调度）。

每个处理器都有一个这样的线程。所有线程的名字都叫做ksoftirad/n，区别在于n，它对应的是处理器的编号。在一个双CPU的机器上就有两个这样的线程，分别叫ksoftirad/0和ksoftirad/1。为了保证只要有空闲的处理器，它们就会处理软中断，所以给每个处理器都分配一个这样的线程。一旦该线程被初始化，它就会执行类似下面这样的死循环：

```
for (;;) {
        if ( !softirq_pending(cpu))
                schedule();

        set_current_state(TASK_RUNNING);

        while( softirq_pending(cpu)) {
                do_softirq();
                if (need_resched())
                    schedule();
        }

        set_current_state(TASK_INTERRUPTIBLE);
}
```

只要有待处理的软中断（由softirq_pending()函数负责发现），ksoftirq就会调用do_softirq去处理它们。通过重复执行这样的操作，重新触发的软中断也会被执行。如果有必要的话，每次迭代后都会调用schedule()以便让更重要的进程得到处理机会。当所有需要执行的操作都完成以后，该内核线程将自己设置为TASK_INTERRUPTIBLE状态，唤起调度程序选择其他可执行进程投入运行。

只要do_softirq()函数发现已经执行过的内核线程重新触发了它自己，软中断内核线程就会被唤醒。

7.3.4 老的BH机制

尽管BH机制令人欣慰地退出历史舞台，在2.6版内核中已经难觅踪迹了。可是，它毕竟曾经历经了漫长的时光——从最早版本的内核就开始了。由于其余威尚存，所以仅仅不经意地提起它是不够的，尽管2.6版本中已经不再使用它了，但历史就是历史，应该被了解。

BH很古老，但它能揭示一些东西。所有BH都是静态定义的，最多可以有32个。由于处理函数必须在编译时就被定义好，所以实现模块时不能直接使用BH接口。不过业已存在的BH倒是可以利用。随着时间的推移，这种静态要求和最大为32个的数目限制最终妨碍了它们的应用。

每个BH处理程序都严格地按顺序执行——不允许任何两个BH处理程序同时执行，即使它们的

类型不同。这样做倒是使同步变得简单了，可是却不利于多处理器的可扩展性，也不利于大型SMP的性能。使用BH的驱动程序很难从多个处理器上受益，特别是网络层，可以说为此饱受困扰。

除了这些特点，BH机制和tasklet就很像了。实际上，在2.4内核中，BH就是基于tasklet实现的。所有可能的32个BH都通过在<linux/interrupt.h>中定义的常量表示。如果需要将一个BH标志为挂起状态，可以把相应的BH号传给mark_bh()函数。在2.4内核中，这将导致随后调度BH tasklet，具体工作是由函数 bh_action()完成的。而在2.4内核以前， BH机制独立实现，不依赖任何低级BH机制，这和现在的软中断很像。

由于这种形式的下半部机制存在缺点，内核开发者们希望引入任务队列机制来代替它。尽管任务队列得到了不少使用者的认可，但它实际上并没有达成这个目的。在2.3版的内核中，引入新的软中断和tasklet机制也就结束了对BH的使用。BH机制基于tasklet重新实现。不幸的是，因为新接口本身降低了对执行的序列化（serialization）保障，所以从BH接口移植到tasklet或软中断接口上操作起来非常复杂[⊖]。在2.5版中，这种移植最终在定时器和SCSI（最后的BH使用者）转换到软中断机制后完成了。于是内核开发者们立即除去BH接口。终于解脱了，BH!

7.4 工作队列

工作队列（work queue）是另外一种将工作推后执行的形式，它和我们前面讨论的所有其他形式都不相同。工作队列可以把工作推后，交由一个内核线程去执行——这个下半部分总是会在进程上下文执行。这样，通过工作队列执行的代码能占尽进程上下文的所有优势。最重要的就是工作队列允许重新调度甚至是睡眠。

通常，在工作队列和软中断/tasklet中作出选择非常容易。如果推后执行的任务需要睡眠，那么就选择工作队列。如果推后执行的任务不需要睡眠，那么就选择软中断或tasklet。实际上，工作队列通常可以用内核线程替换。但是由于内核开发者们非常反对创建新的内核线程（在有些场合，使用这种冒失的方法可能会吃到苦头），所以我们也推荐使用工作队列。当然，这种接口也的确很容易使用。

如果你需要用一个可以重新调度的实体来执行你的下半部处理，你应该使用工作队列。它是惟一能在进程上下文运行的下半部实现的机制，也只有它才可以睡眠。这意味着在你需要获得大量的内存时、在你需要获取信号量时，在你需要执行阻塞式的I/O操作时，它都会非常有用。如果你不需要用一个内核线程来推后执行工作，那么就考虑使用tasklet吧。

7.4.1 工作队列的实现

工作队列子系统是一个用于创建内核线程的接口，通过它创建的进程负责执行由内核其他部分排到队列里的任务。它创建的这些内核线程被称作*工作者线程*（worker thread）。工作队列可以让你的驱动程序创建一个专门的工作者线程来处理需要推后的工作。不过，工作队列子系统提供了一个默认的工作者线程来处理这些工作。因此，工作队列最基本的表现形式就转变成了一个把需要推后执行的任务交给特定的通用线程这样一种接口。

⊖ 实际上，接口降低对执行的序列化保障能够提高安全性，但却难于编程。移植一个BH到tasklet，需要仔细地斟酌代码与其他tasklet同时执行是否安全。不过，当最终完成这样的移植后，性能上的提高会使这些额外工作物有所值。

默认的工作者线程叫做events/n，这里n是处理器的编号；每个处理器对应一个线程。比如，单处理器的系统只有events/0这样一个线程。而双处理器的系统就会多一个events/1线程。默认的工作者线程会从多个地方得到被推后的工作。许多内核驱动程序都把它们的下半部交给默认的工作者线程去做。除非一个驱动程序或者子系统必须建立一个属于它自己的内核线程，否则最好使用默认线程。

不过并不存在什么东西能够阻止代码创建属于自己的工作者线程。如果你需要在工作者线程中执行大量的处理操作，这样做或许会带来好处。处理器密集型和性能要求严格的任务会因为拥有自己的工作者线程而获得好处。此时这么做也有助于减轻默认线程的负担，避免工作队列中其他需要完成的工作处于饥饿状态。

1. 表示线程的数据结构

工作者线程用workqueue_struct结构表示：

```
/*
 * 外部可见的工作队列抽象是
 * 由每个CPU的工作队列组成的数组
 */
struct workqueue_struct {
        struct cpu_workqueue_struct cpu_wq[NR_CPUS];
        const char *name;
        struct list_head list;
};
```

该结构内是一个由cpu_workqueue_struct结构组成的数组，它定义在kernel/workqueue.c中，数组的每一项对应系统中的一个处理器。由于系统中每个处理器对应一个工作者线程。所以对于给定的某台计算机来说，就是每个处理器，每个工作者线程对应一个这样的cpu_workqueue_struct结构体。cpu_work queue_struct是kernel/workqueue.c中的核心数据结构：

```
struct cpu_workqueue_struct {
        spinlock_t lock;                /* 锁定以保护该结构体 */

        long remove_sequence;           /* 最近一个被加上的(下一个要运行的) */
        long insert_sequence;           /* 下一个要加上的 */
        struct list_head worklist;      /* 工作列表 */
        wait_queue_head_t more_work;
        wait_queue_head_t work_done;

        struct workqueue_struct *wq;    /* 有关联的 workqueue_struct结构 */
        task_t *thread;                 /* 有关联的线程 */

        int run_depth;                  /* run_workqueue() 循环深度 */
};
```

注意，每个工作者线程类型关联一个自己的workqueue_struct。在该结构体里面，给每个线程分配一个cpu_workqueue_struct，因而也就是给每个处理器分配一个，因为每个处理器都有一个该类型的工作者线程。

2. 表示工作的数据结构

所有的工作者线程都是用普通的内核线程实现的，它们都要执行worker_thread()函数。在它初

始化完以后，这个函数执行一个死循环并开始休眠。当有操作被插入到队列里的时候，线程就会被唤醒，以便执行这些操作。当没有剩余的操作时，它又会继续休眠。

工作用<linux/workqueue.h>中定义的work_struct结构体表示：

```
struct work_struct{
        unsigned long pending;          /* 这个工作正在等待处理吗? */
        struct list_head entry;         /* 连接所有工作的链表 */
        void (*func) (void *);          /* 处理函数 */
        void *data;                     /* 传递给处理函数的参数 */
        void *wq_data;                  /* 内部使用 */
        struct timer_list timer;        /* 延迟的工作队列所用到的定时器 */
};
```

这些结构体被连接成链表，在每个处理器上的每种类型的队列都对应这样一个链表。比如，每个处理器上用于执行被推后的工作的那个通用线程就有一个这样链表。当一个工作者线程被唤醒时，它会执行它的链表上的所有工作。工作被执行完毕，它就将相应的work_struct对象从链表上移去。当链表上不再有对象的时候，它就会继续休眠。

我们可以看一下worker_thread()函数的核心流程，简化如下：

```
for (;;) {
        set_task_state(current, TASK_INTERRUPTIBLE);
        add_wait_queue(&cwq->more_work, &wait);

        if (list_empty(&cwq->worklist))
                schedule();
        else
                set_task_state(current, TASK_RUNNING);
        remove_wait_queue(&cwq->more_work, &wait);

        if (!list_empty(&cwq->worklist))
                run_workqueue(cwq);
}
```

该函数在死循环中完成了以下功能：

- 线程将自己设置为休眠状态（state被设成TASK_INTERRUPTIBLE）并把自己加入到等待队列上。
- 如果工作链表是空的，线程调用schedule()函数进入睡眠状态。
- 如果链表中有对象，线程不会睡眠。相反，它将自己设置成TASK_RUNNING，脱离等待队列。
- 如果链表非空，调用run_workqueue()函数执行被推后的工作。

3. run_workqueue()

下一步，由run_workqueue()函数来实际完成推后到此的工作：

```
while (!list_empty(&cwq->worklist)) {
        struct work_struct *work ;
        void (*f) (void *);
        void *data;

        work= list_entry(cwq->worklist.next,struct work_struct, entry);
```

```
        f = work->func;
        data = work->data;

        list_del_init(cwq->worklist.next);

        clear_bit(0, &work->pending);
        f(data);
    }
```

该函数循环遍历链表上每个待处理的工作，执行链表每个节点上的workqueue_struct(应该是work_struct。——译者注)中的func成员函数：

- 当链表不为空时，选取下一个节点对象。
- 获取我们希望执行的函数func及其参数data。
- 把该节点从链表上解下来，将待处理标志位pending清0。
- 调用函数。
- 重复执行。

4. 对不起，你到底都说了些什么？

这些数据结构之间的关系确实让人觉得混乱，难以摸清头绪。图7-1给出了示意图，把所有这些关系放在一起进行解释。

位于最高一层的是工作者线程。系统允许有多种类型的工作者线程存在。对于指定的一个类型，系统的每个CPU上都有一个该类的工作者线程。内核中有些部分可以根据需要来创建工作者线程。而在默认情况下内核只有events这一种类型的工作者线程。每个工作者线程都由一个cpu_workequeue_struct结构体表示。而workqueue_struct结构体则表示给定类型的所有工作者线程。

举个例子，在系统默认的通用events 工作者类型之外，我自己加入了一种falcon工作者类型。并且我使用的是一个拥有四个处理器的计算机。那么，系统中现在有四个events类型的线程（因而也就有四个cpu_workqueue_struct结构体）和四个falcon类型的线程（因而会有另外四个cpu_workqueue_struct结构体）。同时，有一个对应events类型的_workqueue_struct和一个对应falcon类型的workqueue_struct。

工作处于最底一层，让我们从这里开始。你的驱动程序创建这些需要推后执行的工作(这其实可以理解成用“工作”这种接口封装我们实际需要推后的工作。以便后续的工作者线程处理——译者注)。它们用work_struct结构来表示。这个结构体中最重要的部分是一个指针，它指向一个函数，而正是该函数负责处理需要推后执行的具体任务。工作会被提交给某个具体的工作者线程——在这种情况下，就是特殊的falcon 线程。然后这个工作者线程会被唤醒并执行这些排好的工作。

大部分驱动程序都使用的是现存的默认工作者线程。它们使用起来简单、方便。可是，在有些要求更严格的情况下，驱动程序需要自己的工作者线程。比如说XFS文件系统就为自己创建了两种新的工作者线程。

7.4.2 使用工作队列

工作队列的使用非常简单。我们先来看一下默认的events任务队列，然后再看看创建新的工作者线程。

1. 创建推后的工作

首先要做的是实际创建一些需要推后完成的工作。可以通过DECLARE_WORK在编译时静态地创建该结构体：

```
DECLARE_WORK(name, void (*func) (void *), void *data);
```

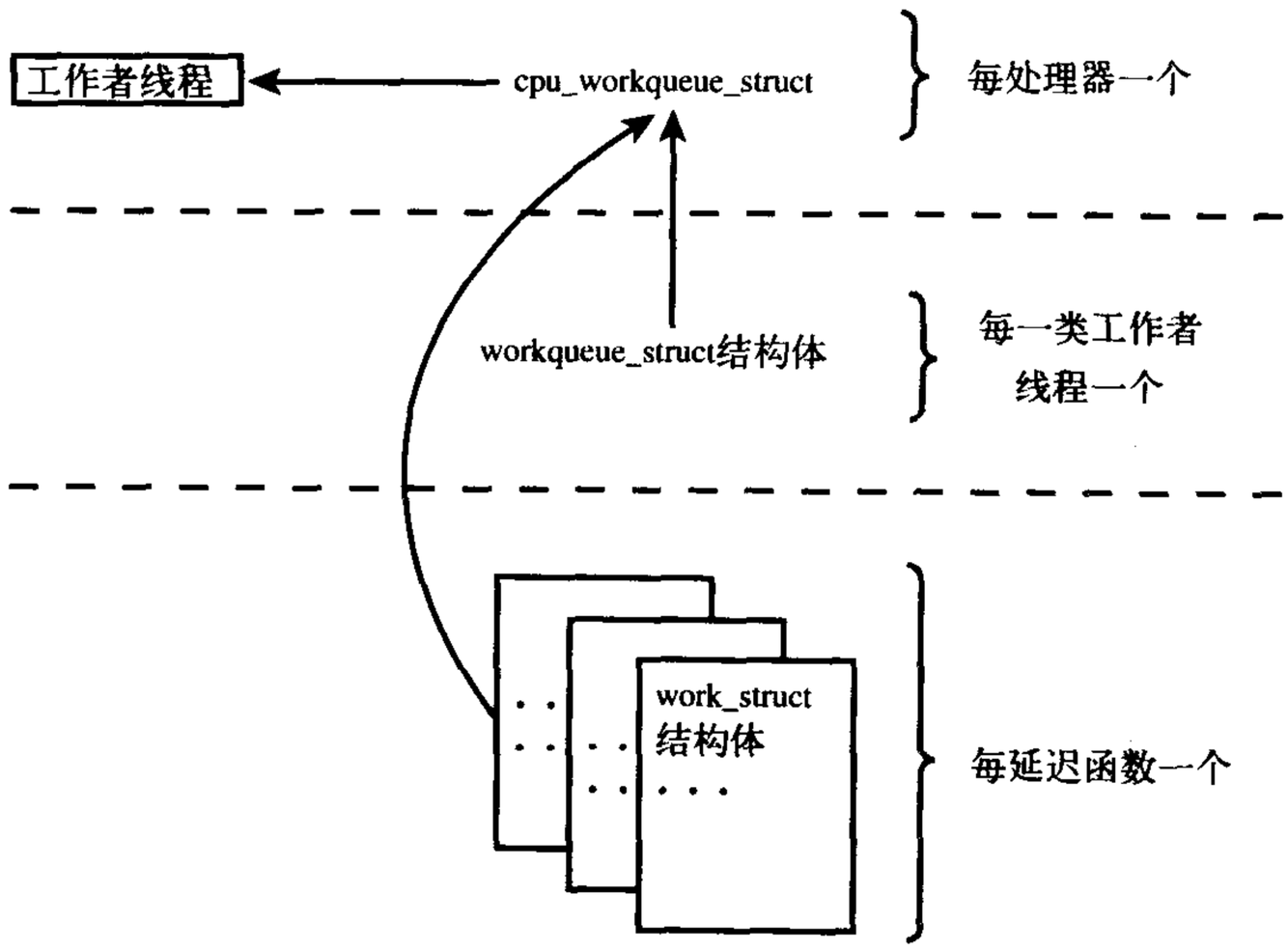


图7-1 工作(work)、工作队列和工作者线程之间的关系

这样就会静态地创建一个名为name，处理函数为func，参数为data的work_struct结构体。同样，也可以在运行时通过指针创建一个工作：

```
INIT_WORK(struct work_struct *work, void(*func) (void *), void *data);
```

这会动态地初始化一个由work指向的工作，处理函数为func，参数为data。

2. 工作队列处理函数

工作队列处理函数的原型是：

```
void work_handler(void *data)
```

这个函数会由一个工作者线程执行，因此，函数会运行在进程上下文中。默认情况下，允许响应中断，并且不持有任何锁。如果需要，函数可以睡眠。需要注意的是，尽管操作处理函数运行在进程上下文中，但它不能访问用户空间，因为内核线程在用户空间没有相关的内存映射。通常在系统调用发生时，内核会代表用户空间的进程运行，此时它才能访问用户空间，也只有在此时它才会映射用户空间的内存。

在工作队列和内核其他部分之间使用锁机制就像在其他的进程上下文中使用锁机制一样方便。这使编写处理函数变得相对容易。接着的两章会讨论到锁机制。

3. 对工作进行调度

现在工作已经被创建，我们可以调度它了。想要把给定工作的处理函数提交给默认的events工作线程，只需调用

```
schedule_work(&work);
```

work马上就会被调度，一旦其所在的处理器上的工作者线程被唤醒，它就会被执行。有时候你并不希望工作马上就被执行，而是希望它经过一段延迟以后再执行。在这种情况下，你可以调度它在指定的时间执行：

```
schedule_delayed_work(&work, delay);
```

这时，&work指向的work_struct直到delay指定的时钟节拍用完以后才会执行。在第10章将介绍这种使用时钟节拍作为时间单位的方法。

4. 刷新操作

排入队列的工作会在工作者线程下一次被唤醒的时候执行。有时，在继续下一步工作之前，你必须保证一些操作已经执行完毕了。这一点对模块来说就很重要，在卸载之前，它就有可能需要调用下面的函数。而在内核的其他部分，为了防止竞争条件的出现，也可能需要确保不再有待处理的工作。

出于以上目的，内核准备了一个用于刷新指定工作队列的函数：

```
void flush_scheduled_work(void);
```

函数会一直等待，直到队列中所有对象都被执行以后才返回。在等待所有待处理的工作执行的时候，该函数会进入休眠状态，所以只能在进程上下文中使用它。

注意，该函数并不取消任何延迟执行的工作。就是说，任何通过schedule_delayed_work()调度的工作，如果其延迟时间未结束，它并不会因为调用flush_scheduled_work()而被刷新掉。取消延迟执行的工作应该调用：

```
int cancel_delayed_work(struct work_struct *work);
```

这个函数可以取消任何与work_struct相关的挂起工作。

5. 创建新的工作队列

如果默认的队列不能满足你的需要，你应该创建一个新的工作队列和与之相应的工作者线程。由于这么做会在每个处理器上都创建一个工作者线程，所以只有在你明确了必须要靠自己的一套线程来提高性能的情况下，再创建自己的工作队列。

创建一个新的任务队列和与之相关的工作者线程，只需调用一个简单的函数：

```
struct workqueue_struct *create_workqueue(const char *name);
```

name参数用于该内核线程的命名。比如，默认的events队列的创建就调用的是

```
struct workqueue_struct *keventd_wq ;
keventd_wq = create_workqueue("events");
```

这个函数会创建所有的工作者线程（系统中的每个处理器都有一个）并且做好所有开始处理工作之前的准备工作。

创建一个工作的时候无须考虑工作队列的类型。在创建之后，可以调用下面列举的函数。这些函数与schedule_work()以及schedule_delayed_work()相近，惟一的区别就在于它们针对给定的工作队列而不是默认的event队列进行操作。

```
int queue_work(struct workqueue_struct *wq, struct work_struct *work);

int queue_delayed_work(struct workqueue_struct *wq,
                       struct work_struct *work,
                       unsigned long delay)
```

最后，你可以调用下面的函数刷新指定的工作队列：

```
flush_workqueue(struct workqueue_struct *wq)
```

该函数和前面讨论过的flush_scheduled_work()作用相同，只是它在返回前等待清空的是给定的队列。

7.4.3 老的任务队列机制

像BH接口被软中断和tasklet代替一样，由于任务队列接口存在的种种缺陷，它也被工作队列接口取代了。像tasklet一样，任务队列接口（内核中常常称作tq）其实也和进程没有什么相关之处[⊖]。任务队列接口的使用者在2.5开发版中分为了两部分。其中一部分转向了使用tasklet。还有另外一部分继续使用任务队列接口。而目前任务队列接口剩余的部分已经演化成了工作队列接口。由于任务队列在内核中曾经使用过一段时间，所以出于了解历史的目的，我们对它进行一个大体回顾。

任务队列机制通过定义一组队列来实现其功能。每个队列都有自己的名字，比如调度程序队列、立即队列和定时器队列。不同的队列在内核中的不同场合使用。keventd内核线程负责执行调度程序队列的相关任务。它是整个工作队列接口的先驱。定时器队列会在系统定时器的每个时间节拍时执行，而立即队列能够得到双倍的运行机会，以保证它能“立即”执行。当然，还有其他一些队列。此外，你还可以动态地创建自己的新队列。

这些听起来都挺有用，但任务队列接口实际上是一团乱麻。这些队列基本上都是些随意创建的抽象概念，散落在内核各处，就像飘散在空气中。惟有调度队列有点意义，它能用来把工作推后到进程上下文完成。

任务队列的另一好处就是接口特别简单。如果不考虑这些队列的数量和执行时随心所欲的规则，它的接口确实够简单。但这也就是全部意义所在了——任务队列剩下的东西乏善可陈。

许多任务队列接口的使用者都已经转向使用其他的下半部实现机制了。大部分选择了tasklet。只有调度程序队列的使用者在苦苦支撑。最终，keventd代码演化成了我们今天使用的工作队列机制，而任务队列最终退出了历史舞台。

7.5 下半部机制的选择

在各种不同的下半部实现机制之间做出选择是很重要的。在当前的2.6版内核中，有三种可能的选择：软中断、tasklet和工作队列。tasklet基于软中断实现，所以两者很相近。工作队列机制与它们完全不同，它靠内核线程实现。

从设计的角度考虑，软中断提供的执行序列化的保障最少。这就要求软中断处理函数必须格外小心地采取一些步骤确保共享数据的安全，两个甚至更多相同类别的软中断有可能在不同的处理器上同时执行。如果被考察的代码本身多线索化的工作就做得非常好，比如像网络子系统，它完全使用单处理器变量，那么软中断就是非常好的选择。对于时间要求严格和执行频率很高的应用来说，它执行得也最快。如果代码多线索化考虑得并不充分，那么选择tasklet意义更大。它的接

⊖ 下半部的各种命名简直可以算得上是迷惑内核开发新手的杀手锏。老实说，这些名字简直就是恶梦！

口非常简单，而且，由于两个同种类型的tasklet不能同时执行，所以实现起来也会简单一些。tasklet是有效的软中断，但不能并发运行。驱动程序开发者应当尽可能选择tasklet而不是软中断，当然，如果准备利用每一处理器上的变量或者类似的情形，以确保软中断能安全地在多个处理器上并发地运行，那么还是选择软中断。

如果你需要把任务推后到进程上下文中完成，那么在这三者中就只能选择工作队列了。如果进程上下文并不是必须的条件——明确点说，就是如果并不需要睡眠——那么软中断和tasklet可能更合适。工作队列造成的开销最大，因为它要牵扯到内核线程甚至是上下文切换。这并不是说工作队列的效率低，如果每秒钟有几千次中断，就像网络子系统时常经历的那样，那么采用其他的机制可能更合适一些。尽管如此，针对大部分情况，工作队列都能提供足够的支持。

如果讲到易于使用，工作队列就当仁不让了。使用默认的events队列简直不费吹灰之力。接下来是tasklet，它的接口也很简单。最后才是软中断，它必须静态创建，并且需要慎重考虑其实现。

表7-3是对三种下半部接口的比较。

表7-3 对下半部的比较

下半部	上下文	顺序执行保障
软中断	中断	没有
tasklet	中断	同类型不能同时执行
工作队列	进程	没有（和进程上下文一样被调度）

简单地说，一般的驱动程序的编写者需要做两个选择。首先，你是不是需要一个可调度的实体来执行需要推后完成的工作——从根本上来说，你有休眠的需要吗？要是有，工作队列就是你的惟一选择。否则最好用tasklet。要是必须专注于性能的提高，那么就考虑软中断吧。

7.6 在下半部之间加锁

到现在为止，我们还没讨论过锁机制，这是一个非常有趣且广泛的话题，我们将在接下来的两章里仔细讨论它。不过，在这里还是应该对它的重要性有所了解，在使用下半部机制时，即使是在一个单处理器的系统上，避免共享数据被同时访问也是至关重要的。记住，一个下半部实际上可能在任何时候执行。如果你对锁机制一无所知的话，你也可以在读完后面两章以后再回过头来看这部分。

使用tasklet的一个好处在于它自己负责执行的序列化保障：两个相同类型的taslet不允许同时执行，即使在不同的处理器上也不行。这意味着你无须为intratasklet[⊖]的同步问题操心了。tasklet之间的同步（就是当两个不同类型的tasklet共享同一数据时）需要正确使用锁机制。

因为软中断根本不保障执行序列化（即使相同类型的软中断也有可能有两个实例在同时执行），所以所有的共享数据都需要合适的锁。

如果进程上下文和一个下半部共享数据，在访问这些数据之前，需要禁止下半部的处理并得到锁的使用权。所做的这些是为了本地和SMP的保护并且防止死锁的出现。

如果中断上下文和一个下半部共享数据，在访问数据之前，需要禁止中断并得到锁的使用权。

⊖ 这个词是我造的。

所做的这些也是为了本地和SMP的保护并且防止死锁的出现。

任何在工作队列中被共享的数据也需要使用锁机制。其中有关锁的要点和在一般内核代码中没什么区别，因为工作队列本来就是在进程上下文中执行的。

在第8章里，我们会揭示锁的奥妙。而在第9章中，我们将讲述内核的加锁原语。这两个章节会描述如何保护下半部使用的数据。

禁止下半部

一般单纯禁止下半部的处理是不够的。为了保证共享数据的安全，更常见的做法是先得到一个锁然后再禁止下半部的处理。驱动程序中通常使用的都是这种方法，在第9章会详细介绍。然而，如果你编写的是内核的核心代码，你也可能仅需要禁止下半部就可以了。

如果需要禁止所有的下半部处理（确切点说，就是所有的软中断和所有的tasklet），可以调用local_bh_diasble()函数。允许下半部进行处理，可以调用local_bh_enable()函数。没错，这些函数的命名也有问题；可是既然BH接口早就让位给软中断了，那么谁又会去改变这些名称呢。表7-4是这些函数的一份摘要。

表7-4 下半部机制控制函数的清单

函 数	描 述
void local_bh_disable()	禁止本地处理器的软中断和tasklet的处理
void local_bh_enable()	激活本地处理器的软中断和tasklet的处理

这些函数有可能被嵌套使用——最后被调用的local_bh_enable()最终激活下半部。比如，第一次调用local_bh_disable()，则本地软中断处理被禁止。如果local_bh_disable()被调用三次，则本地处理仍然被禁止。只有当第四次调用 local_bh_enable()时，软中断处理才被重新激活。

函数通过preempt_count（很有意思，还是这个计数器，内核抢占的时候用的也是它）[⊖]为每个进程维护一个计数器。当计数器变为0时，下半部才能够被处理。因为下半部的处理已经被禁止，所以local_bh_enable()还需要检查所有现存的待处理的下半部并执行它们。

这些函数与硬件体系结构相关，它们位于<asm/softirq.h>中，通常由一些复杂的宏实现。下面是为那些好奇的人准备的C语言的近似描述：

```
/*
 * 通过增加preempt_count禁止本地下半部
 */
void local_bh_disable(void)
{
        struct thread_info *t = current_thread_info();

        t->preempt_count += SOFTIRQ_OFFSET;
}

/*
```

⊖ 实际上，中断和下半部子系统都用到了这个计数器。其实，在Linux中，每个任务都有的单个计数器代表了任务的原子性。在类似调试sleeping-while-atomic之类的错误时，这种做法已被证明是非常有效的。

```
 * 减少preempt_count - 如果该返回值为0，
 * 将导致自动激活下半部
 * 执行挂起的下半部
 */
void local_bh_enable(void)
{
        struct thread_info *t = current_thread_info();

        t->preempt_count -= SOFTIRQ_OFFSET;

        /*
         * preempt_count是否为0，另外是否有挂起的下半部，如果都满足，则执行待执
         * 行的下半部
         */
        if (unlikely(!t->preempt_count && softirq_pending(smp_processor_id())))
                do_softirq();
}
```

这些函数并不能禁止工作队列的执行。因为工作队列是在进程上下文中运行的，不会涉及异步执行的问题，所以也就没有必要禁止它们执行。由于软中断和tasklet是异步发生的（就是说，在中断处理返回的时候），所以，内核代码必须禁止它们。另一方面，对工作队列来说，它保护共享数据所做的工作和其他任何进程上下文中所做的都差不多。第8章和第9章将揭示其中的细节。

7.7 下半部处理小结

在这一章中，我们涵盖了用于延迟Linux内核工作的三种机制：软中断、tasklet和工作队列。我们考察了其设计和实现，讨论了如何把这些机制应用到你的代码中，也调侃了易于混淆的命名。为了完整起见，我们还考察了过去的下半部机制：BH和任务队列，这些用在以前的Linux内核版本中。

因为下半部中相当程度地用到同步和并发，因此本章谈了很多相关的话题。我们甚至围绕本章还讨论了禁止下半部的问题，这是因为并发保护引起的，这一话题到此只是刚刚引入。下一章将从理论上讨论内核同步和并发，为理解这一问题的本质打下基础。随后的一章讨论我们心爱的内核为解决这一问题所提供的具体接口。以这两章为基石，你的梦想就得以实现。

第⑧章

内核同步介绍

共享内存的应用程序必须特别留意保护共享资源，防止共享资源被并发访问。内核也不例外。共享资源之所以要防止并发访问，是因为如果多个执行线程[⊖]同时访问和操作数据，就有可能发生各线程之间相互覆盖共享数据的情况，造成被访问数据处于不一致状态。并发访问共享数据是造成系统不稳定的一类隐患，而且这种错误一般难以跟踪和调试——所以首先应该认识到这个问题的重要性。

要做到对共享资源的恰当保护往往很困难。多年之前，在Linux还未支持对称多处理器的时候，避免并发访问数据的方法相对来说比较简单。在单一处理器的时候，只有在中断发生的时候，或在内核代码显式地请求重新调度，执行另一个任务的时候，数据才可能被并发访问。回想起来，那时的生活何其简单啊！

那只是过去，从2.0开始，内核就开始支持对称多处理器了，而且从那以后对它的支持不断地加强和完善。支持多处理意味着内核代码可以同时运行在两个或更多的处理器上。因此，如果不加保护，运行在两个不同处理器上的内核代码完全可能在同一时刻里并发访问共享数据。随着2.6内核的出现，Linux内核已发展成为了抢占式内核，这意味着（当然，还是指不加保护的情况下）调度程序可以在任何时刻抢占正在运行的内核代码，重新调度其他的进程执行。现在，内核代码中有不少部分都能够同步执行，而它们都必须被妥善地保护起来。

这一章重点讨论操作系统内核中的并发和同步问题，下一章将详细介绍Linux内核为解决同步问题和防止产生竞争条件而提供的机制及接口。

8.1 临界区和竞争条件

所谓临界区（critical region）就是访问和操作共享数据的代码段。多个执行线程并发访问同一个资源通常是不安全的，为了避免在临界区中并发访问，编程者（也就是你）必须保证这些代码原子地执行——也就是说，代码在执行结束前不可被打断，就如同整个临界区是一个不可分割的指令一样。如果两个执行线程有可能处于同一个临界区中，那么这就是程序包含的一个bug。如果这种情况确实发生了，我们就称它是竞争条件（race condition），这样命名是因为这里会存在线程竞争。这种情况出现的机会非常小——就是因为竞争引起的错误非常不易重现，所以调试这种错误才会非常困难。避免并发和防止竞争条件被称为同步（synchronization）。

1. 为什么我们需要保护

为了认清竞争条件的本来面目，我们首先要明白临界区无处不在。作为第一个例子，让我们

⊖ 术语执行线程（thread of execution）指任何正在执行的代码实例，比如，一个在内核执行的进程，一个中断处理程序，或一个内核线程。在这章中将执行线程简称为线程，记住，这个术语指代的是任何正在执行的代码。

考察一个现实世界的情况：ATM（自动柜员机或叫自动提款机）

自动提款机所进行的主要操作就是从个人银行账户取钱。某人走到机器前，插入ATM卡，输入密码作为验证，选择取款，输入金额，敲确认，取出钱，然后发信息通知我。

当用户要求取某一特定的金额后，提款机需要确保在其账户上的确有那么多钱。如果有，取款机就要从现有的总额中扣除取款额。实现这一描述的代码如下：

```
int total = get_total_from_account();        /* 账户上的总额*/
int withdrawal = get_withdrawal_amount();    /* 用户要求的取款额 */

/* 检查用户账户上是否有足够的金额*/
if (total < withdrawal)
        error("You do not have that much money!")

/* 好啦，用户有足够的金额，从总额中扣除取款额*/
total -= withdrawal;
update_total_funds(total);

/* 把钱给用户 */
spit_out_money(withdrawal);
```

现在让我们假定在用户账户上的另一个扣款同时发生。看看扣款是如何同时发生的：假定用户的配偶在另一台ATM上开始另外的取款，或者以电子形式从账户传出资金，或者银行从账户上正在扣除某一费用（因为现在的银行总是这么干），或者其他形式。

正在取款的两个系统都会执行我们刚刚看到的类似的代码：首先检查扣款是否可能，然后计算新的总额，最后进行实际的扣款。现在让我们捏造一些数字。假定第一次从ATM扣款额是100美元，第二次扣除银行申请费10美元，客户走入银行办理，(不允许进来，你必须使用ATM，我们不愿意看到你)。假定客户在这之前在银行总共有105美元。显然，如果账户不出现赤字，这两个操作中有一个就无法完成。

你可能希望发生的顺序是这样的：收费事务先发生。10美元小于105美元，因此，从105中减去10得到新的总额95，这10美元就装在银行的口袋里。之后，ATM取款发生，但未取到，因为$95小于$100。

实际生活可能更有趣，假定两个事务几乎同时开始。两个事务都验证是否有足够的金额存在：$105既大于 $100 ，也大于 $10, 所以，两个条件都满足。于是，取款过程从$105 减去$100，剩余$5。收费事务也如法炮制，从 $105减去$10，剩余$95。此刻，收费事务也更新新的总额，结果得到$95。这可是余额呀!

显而易见，金融机构必须确保类似情况决不能发生。他们必须在某些操作期间对账户加锁，确保每个事务相对其他任何事务的操作是原子的。这样的事务必须完整地发生，要么干脆不发生，否则决不能打断。

2. 单个变量

现在，让我们看一个特殊计算的例子。考虑一个非常简单的共享资源：一个全局整型变量和一个简单的临界区，其中的操作仅仅是将整型变量的值增加1。

```
i++;
```

该操作可以转化成类似于下面动作的机器指令序列：

得到当前变量i的值并且拷贝到一个寄存器中

将寄存器中的值加1

把i的新值写回到内存中

现在假定有两个执行线程同时进入这个临界区，如果i的初始值是7，那么，我们所期望的结果应该像下面这样（每一行代表一个时间单元）：

线程 1	线程 2
获得i(7)	-
增加 i(7->8)	-
写回 i(8)	-
	获得 i(8)
	增加 i(8->9)
	写回 i(9)

正如所期望的，7被两个线程分别加1变为9。但是，实际的执行序列却可能如下：

线程 1	线程 2
获得 i(7)	-
-	获得 i(7)
增加 i(7->8)	-
-	增加 i(7->8)
写回 i(8)	-
-	写回 i(8)

如果两个执行线程都在变量i值增加前读取它的初值，进而又分别增加变量i的值，最后再保存该值，那么变量i的值就变成了8，而变量i的值本该是9的。这是最简单的临界区例子，幸好对这种简单竞争条件的解决方法也同样简单——我们仅仅需要将这些指令作为一个不可分割的整体来执行就万事大吉了。多数处理器都提供了指令来原子地读变量、增加变量然后再写回变量，使用这样的指令就能解决一些问题。使用这个原子指令，惟一可能的结果是:

线程 1	线程 2
增加 i(7->8)	-
-	增加 i(8->9)

或者是:

线程 1	线程 2
-	增加 i(7->8)
增加 i(8->9)	-

两个原子操作交错执行根本就不可能发生，因为处理器会从物理上确保这种不可能。使用这样的指令可缓解这种问题，内核还提供了一组实现这些原子操作的接口，我们将在下一章讨论它们。

8.2 加锁

现在我们来讨论一个更为复杂的竞争条件，相应的解决方法也更为复杂。假设需要处理一个

队列上的所有服务请求，我们可以任意选一种方法实现这个队列，这里我们假定该队列是一个链表，链表中的每个节点就代表一个请求。有两个函数可以用来操作此队列：一个函数将新请求添加到队列尾部，另一个函数从队列头删除请求，然后处理它。内核各个部分都会调用这两个函数，所以内核会频繁地将在队列中加入请求，从队列中删除和处理请求。对请求队列的操作无疑要用到多条指令。如果一个线程试图读取队列，而这时正好另一个线程正在处理该队列，那么读取线程就会发现队列此刻正处于不一致状态。很明显，如果允许并发访问队列，就会产生危害。当共享资源是一个复杂的数据结构时，竞争条件往往会使该数据结构遭到破坏。

表面上看，这种情况好像没有一个好的方法来解决，一个处理器读取队列的时候，我们怎么能禁止另一个处理器更新队列呢？虽然有些体系结构提供简单的原子指令，实现算术运算和比较之类的原子操作，但让体系结构提供专门的指令，对像上例中那样的不定长度的临界区进行保护，就强人所难了。我们需要一种方法确保一次有且只有一个线程对数据结构操作，或者当另一个线程在对临界区标记时，就禁止（或者说锁定）其他访问。

锁提供的就是这种机制：它就如同一把门锁，门后的房间可想像成一个临界区。在一个指定时间内，房间里只能有一个执行线程存在，当一个线程进入房间后，它会锁住身后的房门；当它结束对共享数据的操作后，就会走出房间，打开门锁。如果另一个线程在房门上锁时来了，那么它就必须等待房间内的线程出来并打开门锁后，才能进入房间。

前面例子中讲到的请求队列，可以使用一个单独的锁进行保护。每当有一个新请求要加入队列，线程会首先要占住锁，然后就可以安全地将请求加入到队列中，结束操作后再释放该锁；同样当一个线程想从请求队列中删除一个请求时，也需要先占住锁，然后才能从队列中读取和删除请求，而且在完成操作后也必须释放锁。任何要访问队列的其他线程也类似，必须占住锁后才能进行操作。因为在一个时刻只能有一个线程持有锁，所以在一个时刻只有一个线程可以操作队列。由此可见锁机制可以防止并发执行，并且保护队列不受竞争条件的影响。

任何要访问队列的代码首先都需要占住相应的锁，这样该锁就能阻止来自其他执行线程的并发访问：

线程 1	线程 2
试图锁定队列	试图锁定队列
成功：获得锁	失败：等待…
访问队列…	等待…
为队列解除锁	等待…
…	成功：获得锁
	访问队列…
	为队列解除锁

请注意锁的使用是自愿的、非强制的，它完全属于一种编程者自选的编程手段。没有什么可以强制编程者在操作我们虚构的队列时必须使用锁。当然，如果不这么做，无疑会造成竞争条件而破坏队列。

锁有多种多样的形式，而且加锁的粒度范围也各不相同——Linux自身实现了几种不同的锁机制。各种锁机制之间的区别主要在于当锁被争用时（已经被使用）的行为表现——一些锁被争用

时会简单地执行忙等待[⊖]，而有些锁会使当前任务睡眠直到锁可用为止。下一章我们将讨论Linux中不同锁之间的行为差别及它们的接口。

机灵的读者此时会尖叫起来，锁根本解决不了什么问题，它只不过把临界区缩小到加锁和开锁之间（也许更小）的代码，但是仍然有潜在的竞争！所幸，锁是采用原子操作实现的，而原子操作不存在竞争。其实现是与具体的体系结构密切相关的，但是，几乎所有的处理器都实现了测试和设置指令，这一指令测试整数的值，如果其值为0，就设置一新值。0意味着开锁。

8.2.1 到底是什么造成了并发执行

用户空间之所以需要同步，是因为用户程序会被调度程序抢占和重新调度。由于用户进程可能在任何时刻被抢占，而调度程序完全可能选择另一个高优先级的进程到处理器上执行，所以就有可能在一个程序正处于临界区时，就被非自愿地的抢占了，如果新调度的进程随后也进入同一个临界区（比如说，这两个进程是同一个进程的两个可执行线程，它们两个要访问共享的内存），前后两个进程相互之间就会产生竞争。另外，因为信号处理是异步发生的，所以，即使是单线程的多个进程共享文件，或者在一个程序内部处理信号，也有可能产生竞争条件。这种类型的并发操作——这里其实两者并不真是同时发生的，但它们相互交叉进行，所以也可称作伪并发执行。

如果你有一台支持对称多处理器的机器，那么两个进程就可以真正的在临界区中同时执行了，这种类型被称为*真并发*。虽然真并发和伪并发的原因和含义不同，但它们都同样会造成竞争条件，而且也需要同样的保护。

内核中有类似可能造成并发执行的原因。它们是：

- 中断——中断几乎可以在任何时刻异步发生，也就可能随时打断当前正在执行的代码。
- 软中断和tasklet——内核能在任何时刻唤醒或调度软中断和tasklet，打断当前正在执行的代码。
- 内核抢占——因为内核具有抢占性，所以内核中的任务可能会被另一任务抢占。
- 睡眠及与用户空间的同步——在内核执行的进程可能会睡眠，这就会唤醒调度程序，从而导致调度一个新的用户进程执行。
- 对称多处理——两个或多个处理器可以同时执行代码。

对内核开发者来说，必须理解上述这些并发执行的原因，并且为它们事先做足准备工作。如果在一段内核代码操作某资源的时候系统产生了一个中断，而且该中断的处理程序还要访问这一资源，这就是一个bug；类似地，如果一段内核代码在访问一个共享资源期间可以被抢占，这也是一个bug；还有，如果内核代码在临界区里睡眠，那简直就是鼓掌欢迎竞争条件的到来。最后还要注意，两个处理器绝对不能同时访问同一共享数据。当我们清楚什么样的数据需要保护时，提供锁来保护代码安全也就不难做到了。然而，真正困难的就是发现上述的潜在并发执行的可能，并有意识的采取某些措施来防止并发执行。我们要重申这点，因为它实在是很重要。其实，真正用锁来保护共享资源并不困难，在设计代码的早期就这么做了，事情就更简单了。辨认出真正需要共享的数据和相应的临界区，才是真正有挑战性的地方。要记住，最开始设计代码的时候就要考虑加入锁，而不是事后才想到。如果代码已经写好了，再在其中找到需要上锁的部分并向其中追加锁，是非常困难的，结果也往往不尽人意。所以，这里的基本原则是：在编写代码的开始阶段

⊖ 不断进行循环，等待锁重新可用。

就要设计恰当的锁。

在中断处理程序中能避免并发访问的安全代码称作中断安全代码（interrupt-saft），在对称多处理的机器中能避免并发访问的安全代码称为SMP安全代码（SMP-safe），在内核抢占时能避免并发访问的安全代码称为抢占安全代码（preempt-safe[⊖]）。在下一章会重点讲述为了提供同步和避免所有上述竞争条件，内核所使用的实际方法。

8.2.2 要保护些什么

找出哪些数据需要保护是关键所在。由于任何可能被并发访问的代码都可能需要保护，所以寻找哪些代码不需要保护反而相对更容易些，我们也就从这里入手。执行线程的局部数据仅仅被它本身访问，显然不需要保护，比如，局部自动变量（还有动态分配的数据结构，其地址仅存放在堆栈中）不需要任何形式的锁，因为它们独立存在于执行线程的栈中。类似，如果数据只会被特定的进程访问，那么也不需要加锁（因为进程一次只在一个处理器上执行）。

到底什么数据需要加锁呢？大多数内核数据结构都需要加锁！有一条很好的经验可以帮助我们判断：如果有其他执行线程可以访问这些数据，那么就给这些数据加上某种形式的锁；如果任何其他什么东西能看到它，那么就要锁住它。记住：要给数据而不是代码加锁。

配置选项：SMP与UP

因为Linux内核可在编译时配置，所以你可以针对指定机器进行内核裁剪。更重要的是，CONFIG_SMP配置选项控制内核是否支持SMP。许多加锁问题在单处理器上是不存在的，因而当CONFIG_SMP没被设置时，不必要的代码就不会被编入针对单处理器的内核映像中。这样做可以使单处理器机器避免使用自旋锁带来的开销。同样的技巧也适用于CONFIG_PREEMPT(允许内核抢占的配置选项)。这种设计真的很优越——内核只用维护一些简洁的基础资源，各种各样的锁机制当需要时可随时被编译进内核使用。在不同的体系结构上，CONFIG_SMP和CONFIG_PREEMPT设置不同，实际编译时包含的锁就不同。

在你的代码中，要为大多数糟糕的情况提供适当的保护，例如具有内核抢占的SMP，并且要考虑到所有的情况。

在编写内核代码时，你要问自己下面这些问题：

- 这个数据是不是全局的？除了当前线程外,其他线程能不能访问它？
- 这个数据会不会在进程上下文和中断上下文中共享？它是不是要在两个不同的中断处理程序中共享？
- 进程在访问数据时可不可能被抢占？被调度的新程序会不会访问同一数据？
- 当前进程是不是会睡眠（阻塞）在某些资源上，如果是，它会让共享数据处于何种状态？
- 怎样防止数据失控？
- 如果这个函数又在另一个处理器上被调度将会发生什么呢？
- 你要对这些代码做什么？

简而言之，几乎访问所有的内核全局变量和共享数据都需要某种形式的同步方法，具体加锁

⊖ 我们将看到，除了一些异常情况，SMP安全代码等价于抢占安全代码。

方法将在下章进行讨论。

8.3 死锁

死锁的产生需要一定条件：要有一个或多个执行线程和一个或多个资源，每个线程都在等待其中的一个资源，但所有的资源都已经被占用了。所有线程都在相互等待，但它们永远不会释放已经占有的资源。于是任何线程都无法继续，这便发生了死锁。

一个很好的死锁例子是四路交通堵塞问题。如果每一个停止的车都决心等待其他的车开动后自己再启动，那么就没有任何一辆车能启动，于是发生了交通死锁。

最简单的死锁例子是自死锁[⊖]：如果一个执行线程试图去获得一个自己已经持有的锁，它将不得不等待锁被释放，但因为它正在忙着等待这个锁，所以自己永远也不会有机会释放锁，最终结果就是死锁，如下所示：

```
获得锁
再次试图获得锁
等待锁重新可用
...
```

同样道理，考虑有n个线程和n个锁，如果每个线程都持有一把其他进程需要得到的锁，那么所有的线程都将阻塞地等待它们希望得到的锁重新可用。最常见的例子是有两个线程和两把锁，它们通常称为ABBA死锁，如下所示：

线程1	线程2
获得A锁	获得B锁
试图获得B锁	试图获得A锁
等待B锁	等待A锁

每个线程都在等待其他线程持有的锁，但是绝没有一个线程会释放它们一开始就持有的锁，所以没有任何锁会释放后被其他线程使用。

预防死锁的发生非常重要，虽然很难证明代码不会发生死锁，但是可以写出避免死锁的代码，以下一些简单的规则对避免死锁大有帮助：

1) 加锁的顺序是关键。使用嵌套的锁时必须保证以相同的顺序获取锁，这样可以阻止致命拥抱类型的死锁。最好能记录下锁的顺序，以便其他人也能照此顺序使用。

2) 防止发生饥饿，试问，这个代码的执行是否一定会结束？如果“张”不发生？“王”要一直等待下去吗？

3) 不要重复请求同一个锁。

4) 越复杂的加锁方案越有可能造成死锁——设计应力求简单。

最值得强调的是第一点，它最为重要。如果有两个或多个锁曾在同一时间里被请求，那么以后其他函数请求它们也必须按照前次的加锁顺序进行。假设有cat、dog和fox这几个锁来保护某同名的多个数据结构，同时假设有一个函数对这三个锁保护的数据结构进行操作——可能在它们之间进行拷贝。不管哪种情况，这些数据结构都需要保护才能被安全访问。如果有一个函数以cat、

⊖ 有些内核使用递归锁来防止自死锁现象，递归锁可以被一个执行线程多次请求。幸好Linux没有提供这样的递归锁。不用递归锁通常被认为是一件好事，虽然递归锁缓和了自死锁问题，但它们很容易使加锁逻辑变得杂乱无章。

dog，然后是fox的顺序获得了锁，那么其他任何函数都必须以同样的顺序来获得这些锁（或是它们的子集）。如果其他函数首先获得fox锁，然后获得dog锁（因为dog锁总是应该先于fox锁被获得），就有发生死锁的可能（所以是个bug）。为更直观地说明，下面给出一个造成死锁的例子：

线程1	线程2
获得 cat锁	获得 fox锁
获得 dog锁	试图获得 dog锁
试图获得 fox锁	等待 dog锁
等待 fox锁	…

线程1在等待fox锁，而该锁此刻被线程2持有；同样线程2正在等待dog锁，而该锁此刻又被线程1持有。任何一方都不会放弃自己已持有的锁，于是双方都会永远的等待下去——也就是死锁。但是，只要线程都按照相同的顺序去获取这些锁，就可以避免上述的死锁情况。

只要嵌套地使用多个锁，就必须按照相同的顺序去获取它们。在代码中使用锁的地方，对锁的获取顺序加上注释是个良好的习惯。下面的例子就做得很不错：

```
/*
 * cat_lock——总是要在获得dog锁前先获得
 */
```

注意，尽管释放锁的顺序和死锁是无关的，但最好还是以获得锁的相反顺序来释放锁。

防止死锁非常重要，所以Linux内核提供了一些简单易用的调试工具，可以在运行时检测死锁，我们将在下一章讨论它们。

8.4 争用和扩展性

锁的争用(lock contention)，或简称争用，是指当锁正在被占用时，有其他线程试图获得该锁。说一个锁处于高度争用状态，就是指有多个其他线程在等待获得该锁。由于锁的作用是使程序以串行方式对资源进行访问，所以使用锁无疑会降低系统的性能。被高度争用的锁会成为系统的瓶颈，严重降低系统性能。即使是这样，相对于被几个相互抢夺共享资源的线程撕成碎片，搞得内核崩溃，还是这种同步保护来得好一点。当然，如果有办法能解决高度争用问题，就更好不过了。

扩展性（scalability）是对系统可扩展程度的一个量度。对于操作系统，我们在谈及可扩展性时就会和大量的进程、处理器、或是大量的内存等联系起来。其实任何可以被计量的计算机组件都可以涉及可扩展性。理想情况下，处理器的数量加倍应该会使系统处理性能翻倍。而实际上，这是不可能达到的。

自从2.0内核引入多处理支持后，Linux对集群处理器的可扩展性大大提高了。在Linux刚加入对多处理器支持的时候，一个时刻只能有一个任务在内核中执行；在2.2 版本中，当加锁机制发展到细粒度(fine-grained)加锁后，便取消了这种限制，而在2.4和后续版本中，内核加锁的粒度变得越来越精细。如今，在 Linux 2.6内核中，内核加的锁是非常细的粒度，可扩展性也很好。

加锁粒度用来描述加锁保护的数据规模。一个过粗的锁保护大块数据——比如，一个子系统用到的所有的数据结构；相反，一个过于精细的锁保护很小的一块数据——比如，一个大数据结构中的一个元素。在实际使用中，绝大多数锁的加锁范围都处于上述两种极端之间，保护的既不是一个完整的子系统也不是一个独立元素，而可能是一个单独的数据结构。许多锁的设计在开始

阶段都很粗，但是当锁的争用问题变得严重时，设计就向更加精细的加锁方向进化。

在第4章中讨论过的运行队列，就是一个锁从粗到精细化的实例。在2.4版和更早的内核中，调度程序有一个单独的调度队列（回忆一下，调度队列是一个由可调度进程组成的链表），在2.6版中，O(1)调度程序为每个处理器单独配备一个运行队列，每个队列拥有自己的锁，于是加锁由一个全局锁逐步精细化到了每个处理器拥有各自的锁。这是一种重要的优化，因为运行队列锁在大型机器上被争着用，本质上就是要在调度程序中每次都把整个调度进程下至到单个处理器上执行。

一般来说，提高可扩展性是件好事，因为它可以提高Linux在更大型的，处理能力更强大的系统上的性能。但是一味的“提高”可扩展性，却会导致Linux在小型SMP和UP机器上的性能降低，这是因为小型机器可能用不到特别精细的锁，锁得过细只会增加复杂度，并加大开销。考虑一个链表，最初的加锁方案可能就是用一个锁来保护链表，后来发现，在拥有集群处理器机器上，当各个处理器需要频繁访问该链表的时候，只用单独一个锁却成了扩展性的瓶颈。为解决这个瓶颈，我们将原来加锁的整个链表变成为链表中的每一个节点都加入自己的锁，这样一来，如果要对节点进行读写，必须先得到这个节点对应的锁。将加锁粒度变细后，多处理器访问同一个节点时，只会争用一个锁。可是这时锁的争用仍然没有完全避免，那么，能不能为每个节点中的每个元素都提供一个锁呢？（答案是：不能）。严格地讲，即使这么细的锁可以在大规模SMP机器上执行得很好，但它在双处理器机器上的表现又会怎样呢？如果在双处理器机器锁争用表现得并不明显，那么多余的锁会加大系统开销，造成很大浪费。

不管怎么说，可扩展性都是很重要的，需要慎重考虑。关键在于，在设计锁的开始阶段就应该考虑到要保证良好的扩展性。因为即使在小型机器上，如果对重要资源锁得太粗，也很容易造成系统性能瓶颈。锁加得过粗或过细，差别往往在一线之间。当锁争用严重时，加锁太粗会降低可扩展性；而锁争用不明显时，加锁过细会加大系统开销，带来浪费，这两种情况都会造成系统性能下降。但要记住：设计初期加锁方案应该力求简单，仅当需要时再进一步细化加锁方案。*精髓在于力求简单。*

8.5 小结

要编写SMP安全代码，不能等到编码完成后才考虑如何加锁。恰当的同步（也就是加锁）——既要满足不死锁、可扩展，而且还要清晰、简洁——需要从头到尾，在整个编码过程中不断考虑并完善。无论编写哪种内核代码，是新的系统调用也好，还是重写驱动程序也好，首先应该考虑的就是保护数据不被并发访问，记住，加锁你的代码。

下一章将讨论如何为SMP、内核抢占和其他各种情况提供充分的同步保护，确保数据在任何机器和配置中的安全。

了解了同步、并发和加锁的基本原理之后，让我们现在潜心钻研Linux内核提供的实际工具，以确保你的代码竞争力强而有能力免于死锁。

第⑨章
内核同步方法

上一章讨论了竞争条件为何会产生以及怎么去解决。幸运的是，Linux内核提供了一组相当完备的同步方法。这一章讨论的就是这些方法，包括它们的接口、行为和用途。这些方法能使内核开发者们编写出高效而又自由竞争的代码。

9.1 原子操作

原子操作可以保证指令以原子的方式执行——执行过程不被打断。众所周知，原子原本指的是不可分割的微粒，所以原子操作也就是不能够被分割的指令。例如，前一章曾提到过的原子方式的加操作，它通过把读取和增加变量的行为包含在一个单步中执行，从而防止了竞争的发生，保证了操作结果总是一致的（假定i初值是7）：

线程 1	线程 2
atomic increment i(7->8)	-
-	atomic increment i(8->9)

最后得到的9，毫无疑问是正确结果。两个原子操作绝对不可能并发地访问同一个变量，这样加操作也就绝不可能引起竞争。

内核提供了两组原子操作接口——一组针对整数进行操作，另一组针对单独的位进行操作。在Linux支持的所有体系结构上都实现了这两组接口。大多数体系结构要么本来就支持简单的原子操作，要么就为单步执行提供了锁内存总线的指令（这就确保其他操作不能同时发生）。而有些体系结构本身就不太支持原子操作，比如SPARC，但还是有一些办法可想，而且也确实做到了(在所有的SPARC机器上确保都存在的惟一原子指令为ldstub)。

9.1.1 原子整数操作

针对整数的原子操作只能对atomic_t类型的数据进行处理。在这里之所以引入了一个特殊数据类型，而没有直接使用C语言的int类型，主要是出于两个原因：首先，让原子函数只接受atomic_t类型的操作数，可以确保原子操作只与这种特殊类型数据一起使用。同时，这也保证了该类型的数据不会被传递给其他任何非原子函数。实际上，对一个数据一会儿要采用原子操作，一会儿又不用原子操作了，这又能有什么好处？其次，使用atomic_t类型确保编译器不对（不能说完美地完成了任务但不乏自知之明）相应的值进行访问优化——这点使得原子操作最终接收到正确的内存地址，而不只是一个别名。最后，在不同体系结构上实现原子操作的时候，使用atomic_t可以屏蔽其间的差异。

尽管Linux支持的所有机器上的整型数据都是32位的，但是使用atomic_t的代码只能将该类型

的数据当作24位来用。这个限制完全是因为在SPARC体系结构上，原子操作的实现不同于其他体系结构：32位int类型的低8位中嵌入了一个锁（如图9-1所示），因为SPARC体系结构对原子操作缺乏指令级的支持，所以只能利用该锁来避免对原子类型数据的并发访问。所以在SPARC机器上就只能使用24位了。虽然其他机器上的代码完全可以使用这全部的32位，但在 SPARC机上却可能造成一些奇怪和微妙的错误——这简直太不和谐了。最近，机灵的黑客已经允许SPARC提供全32位的atomic_t，这一限制不存在了。原来24位的实现还在某些SPARC内部代码中使用，但是，位于SPARC的<asm/atomic.h>。

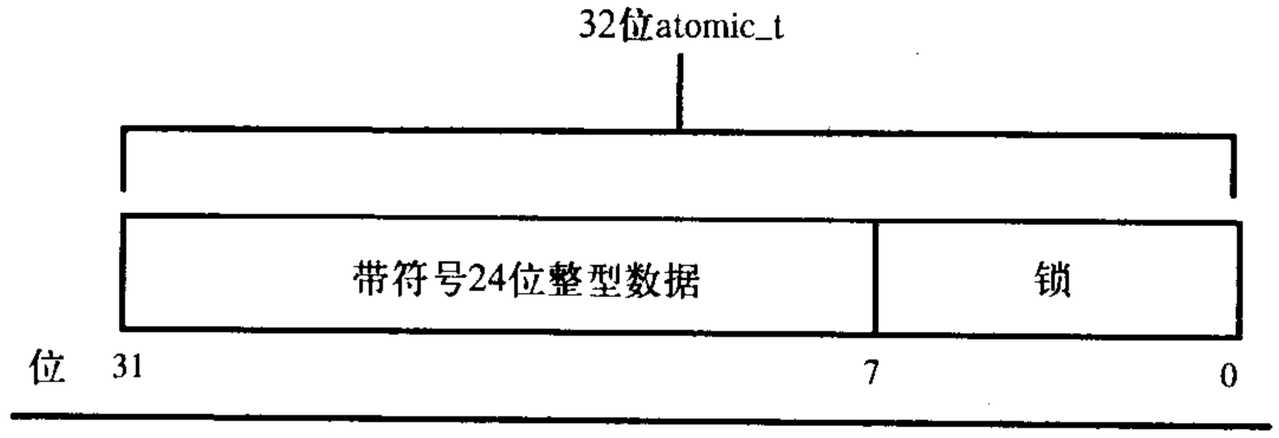


图9-1 SPARC上的32位atomic_t的布局

使用原子整型操作需要的声明都在<asm/atomic.h>文件中。有些体系结构会提供一些只能在该体系结构上使用的额外原子操作方法，但所有的体系结构都能保证内核使用到的所有操作的最小集。在你写内核代码时，可以肯定，这个最小操作集在所有体系结构上都已实现好了。

定义一个atomic_c类型的数据方法很平常，你还可以在定义时给它设定初值：

```
atomic_t v;                          /*定义 v */
atomic_t u = ATOMIC_INIT(0);   /*定义 u 并把它初始化为0*/
```

操作也都非常简单：

```
atomic_set(&v,4);   /* v = 4 (原子地)*/
atomic_add(2,&v);   /* v = v + 2 = 6 (原子地) */
atomic_inc(&v);     /* v = v + 1 =7(原子地)*/
```

如果需要将atomic_t转换成int型，可以使用atomic_read()来完成：

```
printk("%d\n",atomic_read(&v)); /* 将打印"7"*/
```

原子整数操作最常见的用途就是实现计数器。使用复杂的锁机制来保护一个单纯的计数器是很笨拙的，所以，开发者最好使用atomic_inc()和atomic_dec()这两个相对来说轻便一点的操作。还可以用原子整数操作原子地执行一个操作并检查结果。一个常见的例子就是原子的减操作和检查。

```
int atomic_dec_and_test(atomic_t *v)
```

这个函数将给定的原子变量减1，如果结果为0，就返回真；否则返回假。表9-1列出了所有的标准原子整数操作（所有体系结构都包含这些操作）。某种特定的体系结构上实现的所有操作可以在文件<asm/atomic.h>中找到。

原子操作通常是内联函数，往往是通过内嵌汇编指令来实现的（内核开发者喜欢使用内联函数）。如果某个函数本来就是原子的，那么它往往会被定义成一个宏。例如，在大部分健全的体系结构上，读取一个字本身就是一种原子操作，也就是说，在对一个字进行写入操作的期间不可能完成对该字的读取。这样，把atomic_read()定义成一个宏，只须返回atomic_t类型的整数值就可以了。

表9-1 原子整数操作列表

原子整数操作	描 述
ATOMIC_INIT(int i)	在声明一个atomic_t变量时，将它初始化为i
int atomic_read(atomic_t *v)	原子地读取整数变量v
void atomic_set(atomic_t *v, int i)	原子地设置v值为1
void atomic_add(int i,atomic_t *v)	原子地给v加i
void atomic_sub(int i,atomic_t *v)	原子地从v减i
void atomic_inc(atomic_t *v)	原子地给v加1
void atomic_dec(atomic_t *v)	原子地从v减1
int atomic_sub_and_test(int i,atomic_t *v)	原子地从v减i,如果结果等于0返回真；否则返回假
int atomic_add_negative(int i,atomic_t *v)	原子地给v加i,如果结果是负数，返回真；否则返回假
int atomic_dec_and_test(atomic_t *v)	原子地给v减1,如果结果是0，返回真；否则返回假
int atomic_inc_and_test(atomic_t *v)	原子地给v加1,如果结果是0，返回真；否则返回假

原子性与顺序性的比较

关于原子读取的上述讨论引发了原子性与顺序性之间差异的讨论。正如所讨论的，一个字长的读取总是原子地发生，绝不可能对同一个字交错地进行写；读总是返回一个完整的字，这或者发生在写操作之前，或者之后，绝不可能发生在写的过程中。例如，如果一个整数初始化为42，然后又置为365，那么读取这个整数肯定会返回42或者365，而决不会是二者的混合。这就是原子性。

也许你的代码比这还有更多的要求，或许读必须在待定的写之前发生。这就不是原子的了，而是顺序的。原子性确保指令执行期间不被打断，要么全部执行完，要么根本不执行。而顺序性确保即使两条或多条指令出现在独立的执行线程中，甚至独立的处理器上，但它们本该的执行顺序依然要保持。

在本节讨论的原子操作只保证原子性。顺序性通过屏障（barrier）指令来实施，这将在本章的后面讨论。

在你编写代码的时候，能使用原子操作的时候，就尽量不要使用复杂的加锁机制。对多数体系结构来讲，原子操作与更复杂的同步方法相比较，给系统带来的开销小，对高速缓存行（cache-line）的影响也小。但是，对于那些有高性能要求的代码，对多种同步方法进行测试比较，不失为一种明智的作法。

9.1.2 原子位操作

除了原子整数操作外，内核还提供了一组针对位这一级数据进行操作的函数。没什么好奇怪的，它们是与体系结构相关的操作，定义在文件<asm/bitops.h>中。

令人感到奇怪的是位操作函数是对普通的内存地址进行操作的。它的参数是一个指针和一个位号，第0位是给定地址的最低有效位。在32位机上，第31位是给定地址的最高有效位而第32位是下一个字的最低有效位。虽然使用原子位操作在多数情况下是对一个字长的内存进行访问，因而位号应该位于0～31之间（在64位机器中是0～63之间），但是，对位号的范围并没有限制。

由于原子位操作是对普通的指针进行的操作，所以不像原子整型对应atomic_t，这里没有特殊

的数据类型。相反，只要指针指向了任何你希望的数据，你就可以对它进行操作。来看一个例子：

```
unsigned long word = 0;

set_bit(0,&word);          /*第0位被设置（原子地）*/
set_bit(1,&word);          /*第1位被设置（原子地）*/
printk("%ul\n",word);      /*打印3*/
clear_bit(1,&word);        /*清空第1位（原子地）*/
change_bit(0,&word);       /*翻转第0位的值，这里它被清空（原子地）*/

/*原子地设置第0位并且返回设置前的值（0）*/
if(test_and_set_bit(0,&word)){
        /*永远不为真…*/
}

/* 下面的语句是合法的；你可以把原子位指令与普通的C语句混在一起*/
word = 7;
```

在表9-2中给出了标准原子位操作列表。

表9-2 原子位操作的列表

原子位操作	描 述
void set_bit(int nr,void *addr)	原子地设置addr所指对象的第nr位
void clear_bit(int nr,void *addr)	原子地清空addr所指对象的第nr位
void change_bit(int nr,void *addr)	原子地翻转addr所指对象的第nr位
int test_and_set_bit(int nr,void *addr)	原子地设置addr所指对象的第nr位，并返回原先的值
int test_and_clear _bit(int nr,void *addr)	原子地清空addr所指对象的第nr位，并返回原先的值
int test_and_change_bit(int nr,void *addr)	原子地翻转addr所指对象的第nr位，并返回原先的值
int test_bit(int nr,void *addr)	原子地返回addr所指对象的第nr位

为方便起见，内核还提供了一组与上述操作对应的非原子位函数。非原子位函数与原子位函数的操作完全相同，但是，前者不保证原子性，且其名字前缀多两个下划线。例如，与test_bit()对应的非原子形式是__test_bit()。如果你不需要原子性操作（比如说，如果你已经用锁保护了自己的数据），那么这些非原子的位函数相比原子的位函数可能会执行得更快些。

非原子位操作到底是什么？

乍一看，非原子位操作没有任何意义。因为仅仅涉及一个位，所以不存在发生矛盾的可能。只要其中的一个操作成功，还会有什么事？的确，顺序性可能是重要的，但我们在此正谈论原子性。到了最后，如果这一位有了任一条指令所设置的值，我们应当友好地离开，对吗？

让我们跳回到原子性看看到底意味着什么。原子性意味着，或者指令完整地成功执行完，不被打断，或者根本不执行。当然，位需要有一致而正确的值（最后一次成功操作而得到的具体值，就像在前面所提到的）。不过，除此之外，如果其他的操作成功，那么在某些点上，位也需要一些中间值。

例如，假定你给出两个原子位操作：先对某位置位，然后清0。如果没有原子操作，那么，这一位可能的确清0了，但是可能根本没有置位。置位操作可能与清除操作同时发生，但没有

成功。清除操作可能成功了，这一位如愿呈现为清0。但是，有了原子操作，置位会真正发生，可能有那么一刻，读操作显示所置的位，然后清除操作才执行，该位变为0了。

这种行为可能是重要的，尤其当顺序性开始起作用的时候。

内核还提供了两个例程用来从指定的地址开始搜索第一个被设置（或未被设置）的位。

```
int find_first_bit(unsigned long *addr,unsigned int size)
int find_first_zero_bit(unsigned long *addr,unsigned int size)
```

这两个函数中第一个参数是一个指针，第二个参数是要搜索的总位数，返回值分别是第一个被设置的（或没被设置的）位的位号。如果你的搜索范围仅限于一个字，使用__ffs()和__ffz()这两个函数更好，它们只需要给定一个要搜索的地址做参数。

与原子整数操作不同，代码一般无法选择是否使用位操作——它们是惟一的、具有可移植性的设置特定位方法，需要选择的是使用原子位操作还是非原子位操作。如果你的代码本身已经避免了竞争条件，你可以使用非原子位操作，通常这样执行得更快，当然，这还要取决于具体的体系结构。

9.2 自旋锁

如果每个临界区都能像增加变量这样简单就好了，可惜现实总是残酷的。现实世界里，临界区甚至可以跨越多个函数。举个例子，我们经常会碰到这种情况：先得从一个数据结构中移出数据，对其进行格式转换和解析，最后再把它加入到另一个数据结构中。整个执行过程必须是原子的，在数据被更新完毕前，不能有其他代码读取这些数据。显然，简单的原子操作对此无能为力，这就需要使用更为复杂的同步方法——锁来提供保护。

Linux内核中最常见的锁是*自旋锁*（spin lock）。自旋锁最多只能被一个可执行线程持有。如果一个执行线程试图获得一个*被争用*（已经被持有）的自旋锁，那么该线程就会一直进行忙循环—旋转—等待锁重新可用。要是锁未被争用，请求锁的执行线程便能立刻得到它，继续执行。在任意时间，自旋锁都可以防止多于一个的执行线程同时进入临界区。注意同一个锁可以用在多个位置——例如，对于给定数据的所有访问都可以得到保护和同步。

再回到上一章门和锁的例子，自旋锁相当于坐在门外等待同伴从里面出来，并把钥匙交给你。如果你到了门口，发现里面没有人，就可以抓到钥匙进入房间。如果你到了门口发现里面正好有人，就必须在门外等待钥匙，不断地检查房间是否为空。当房间为空时，你就可以抓到钥匙进入。正是因为有了钥匙（相当于自旋锁），才允许一次只有一个人（相当于执行线程）进入房间（相当于临界区）。

一个被争用的自旋锁使得请求它的线程在等待锁重新可用时自旋（特别浪费处理器时间），这种行为是自旋锁的要点。所以自旋锁不应该被长时间持有。事实上，这点正是使用自旋锁的初衷：在短期间内进行轻量级加锁。还可以采取另外的方式来处理对锁的争用：让请求线程睡眠，直到锁重新可用时再唤醒它。这样处理器就不必循环等待，可以去执行其他代码。这也会带来一定的开销——这里有两次明显的上下文切换，被阻塞的线程要换出和换入，与实现自旋锁的少数几行代码相比，上下文切换当然有较多的代码。因此，持有自旋锁的时间最好小于完成两次上下文切

换的耗时。当然我们大多数人都不会无聊到去测量上下文切换的耗时，所以我们让持有自旋锁的时间应尽可能的短就可以了[⊖]。下一节讨论信号量，信号量便提供了上述第二钟锁机制，它使得在发生争用时，等待的线程能投入睡眠，而不是旋转。

自旋锁的实现和体系结构密切相关，代码往往通过汇编实现。这些与体系结构相关的代码定义在文件<asm/spinlock.h>中，实际需要用到的接口定义在文件<linux/spinlock.h>中。自旋锁的基本使用形式如下：

```
spinlock_tmr_lock=SPIN_LOCK_UNLOCKED;

spin_lock(&mr_lock);
/*临界区 ...*/
spin_unlock(&mr_lock);
```

因为自旋锁在同一时刻至多被一个执行线程持有，所以一个时刻只能有一个线程位于临界区内，这就为多处理器机器提供了防止并发访问所需的保护机制。注意在单处理器机器上，编译的时候并不会加入自旋锁。它仅仅被当作一个设置内核抢占机制是否被启用的开关。如果禁止内核抢占，那么在编译时自旋锁会被完全剔除出内核。

警告：自旋锁是不可递归的！

Linux内核实现的自旋锁是不可递归的，这点不同于自旋锁在其他操作系统中的实现。所以如果你试图得到一个你正持有的锁，你必须自旋，等待你自己释放这个锁。但你处于自旋忙等待中，所以永远没有机会释放锁，于是你被自己锁死了。千万要小心自加锁！

自旋锁可以使用在中断处理程序中（此处不能使用信号量，因为它们会导致睡眠）。在中断处理程序中使用自旋锁时，一定要在获取锁之前，首先禁止本地中断（在当前处理器上的中断请求），否则，中断处理程序就会打断正持有锁的内核代码，有可能会试图去争用这个已经被持有的自旋锁。这样一来，中断处理程序就会自旋，等待该锁重新可用，但是锁的持有者在这个中断处理程序执行完毕前不可能运行。这正是我们在前一章节中提到的双重请求死锁。注意，需要关闭的只是当前处理器上的中断。如果中断发生在不同的处理器上，即使中断处理程序在同一锁上自旋，也不会妨碍锁的持有者（在不同处理器上）最终释放锁。

内核提供的禁止中断同时请求锁的接口，使用起来很方便，方法如下。

```
spinlock_t mr_lock=SPIN_LOCK_UNLOCKED;
unsigned long flags;

spin_lock_irqsave(&mr_lock,flags);
/*临界区...*/
spin_unlock_irqrestore(&mr_lock,flags);
```

函数spin_lock_irqsave()保存中断的当前状态，并禁止本地中断，然后再去获取指定的锁。反过来spin_lock_irqsave()对指定的锁解锁，然后让中断恢复到加锁前的状态。所以即使中断最初是

⊖ 在现在的抢占式内核中，这点尤为重要。锁的持有时间等价于系统的调度等待时间。

被禁止的，你的代码也不会错误地激活它们，相反，会继续让它们禁止。注意，flags变量看起来像是由数值传递的，这是因为这些锁函数有些部分是通过宏的方式实现的。

在单处理器系统上，虽然在编译时抛弃掉了锁机制，但在上面例子中仍需要关闭中断，以禁止中断处理程序访问共享数据。加锁和解锁分别可以禁止和允许内核抢占。

锁什么？

使用锁的时候一定要对症下药，要有针对性。要知道需要保护的是数据而不是代码。尽管本章的例子讲的都是保护临界区的重要性，但是真正保护的其实是临界区中的数据，而不是代码。一个大原则是：针对代码加锁会使得程序难以理解，并且容易引发竞争条件，正确的做法应该是对数据而不是代码加锁。

既然不是对代码加锁，那就一定要用特定的锁来保护自己的共享数据。例如，“struct foo由loo_lock加锁”。无论你何时需要访问数据，一定要先保证数据是安全的。而保证数据安全往往就意味着在对数据进行操作前，首先占用恰当的锁，完成操作后再释放它。

如果你能确定中断在加锁前是激活的，那就不需要在解锁后恢复中断以前的状态了。你可以无条件地在解锁时激活中断。这时，使用spin_lock_irq()和spin_unlock_irq()会更好一些。

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;

spin_lock_irq(&mr_lock);
/*临界区 ... */
spin_unlock_irq(&mr_lock);
```

由于内核变得庞大而复杂，因此，在内核的执行路线上，你很难搞清楚中断在当前调用点上到底是不是处于激活状态。也正因为如此，我们并不提倡使用spin_lock_irq()方法。如果你一定要使用它，那你应该确定中断原来就处于激活状态，否则当其他人期望中断处于未激活状态时却发现处于激活状态，可能会非常不开心。

调试自旋锁

配置选项CONFIG_DEBUG_SPINLOCK为使用自旋锁的代码加入了许多调试检测手段。例如，激活了该选项，内核就会检查是否使用了未初始化的锁，是否在还没加锁的时候就要对锁执行开锁操作。在测试代码时，总是应该激活这个选项。

9.2.1 其他针对自旋锁的操作

spin_lock_init()用来初始化动态创建的自旋锁（此时你只有一个指向spinlock_t类型的指针，没有它的实体）。

spin_try_lock()试图获得某个特定的自旋锁，如果该锁已经被争用，那么该方法会立刻返回一个非0值，而不会自旋等待锁被释放；如果成功地获得了这个自旋锁，该函数返回0。同理，spin_is_locked()方法用于检查特定的锁当前是否已被占用，如果已被占用，返回非0值；否则返回0。

该方法只做判断，并不实际占用[⊖]。

表9-3给出了标准的自旋锁操作的完整列表。

表9-3 自旋锁方法列表

方 法	描 述
spin_lock()	获取指定的自旋锁
spin_lock_irq()	禁止本地中断并获取指定的锁
spin_lock_irqsave()	保存本地中断的当前状态，禁止本地中断，并获取指定的锁
spin_unlock()	释放指定的锁
spin_unlock_irq()	释放指定的锁，并激活本地中断
spin_unlock_irqrestore()	释放指定的锁，并让本地中断恢复到以前状态
spin_lock_init()	动态初始化指定的spinlock_t
spin_trylock()	试图获取指定的锁；如果未获取，则返回非0
spin_is_locked()	如果指定的锁当前正在被获取，则返回非0，否则，返回0

9.2.2 自旋锁和下半部

在第7章中曾经提到，在与下半部配合使用时，必须小心地使用锁机制。函数spin_lock_bh()用于获取指定锁，同时它会禁止所有下半部的执行。相应的spin_unlock_bh()函数执行相反的操作。

由于下半部可以抢占进程上下文中的代码，所以当下半部和进程上下文共享数据时，必须对进程上下文中的共享数据进行保护，所以需要加锁的同时还要禁止下半部执行。同样，由于中断处理程序可以抢占下半部，所以如果中断处理程序和下半部共享数据，那么就必须在获取恰当的锁的同时还要禁止中断。

回忆一下，同类的tasklet不可能同时运行，所以对于同类tasklet中的共享数据不需要保护。但是当数据被两个不同种类的tasklet共享时，就需要在访问下半部中的数据前先获得一个普通的自旋锁。这里不需要禁止下半部，因为在同一个处理器上决不会有tasklet相互强占的情况。

对于软中断，无论是否同种类型，如果数据被软中断共享，那么它必须得到锁的保护。这是因为即使是同种类型的两个软中断也可以同时运行在一个系统的多个处理器上。但是，同一处理器上的一个软中断绝不会抢占另一个软中断，因此，根本没必要禁止下半部。

9.3 读-写自旋锁

有时，锁的用途可以明确地分为读取和写入。例如，对一个链表可能既要更新又要检索。当更新（写入）链表时，不能有其他代码并发地写链表或从链表中读取数据，写操作要求完全互斥。另一方面，当对其检索（读取）链表时，只要其他程序不对链表进行写操作就行了。只要没有写操作，多个并发的读操作都是安全的。任务链表的存取模式（在第3章中讨论过）就非常类似于这种情况，它就是通过读——写自旋锁获得保护的。

当对某个数据结构的操作可以像这样被划分为读/写两种类别时，类似读/写锁这样的机制就很有用了。为此，Linux提供了专门的读——写自旋锁。这种自旋锁为读和写分别提供了不同的锁。一个或多个读任务可以并发的持有读者锁；相反，用于写的锁最多只能被一个写任务持有，而且

⊖ 这两个方法让代码变得复杂，本来是用不着经常检查自旋锁的——你的代码本身应该要么直接请求占用锁，要么应该在占用锁之后才能被调用。不过，它们还是有些合理用途，所以Linux内核也就提供了这样的接口。

此时不能有并发的读操作。有时把读/写锁叫做共享/排斥锁，或者并发/排斥锁，因为这种锁以共享（对读者而言）和排斥（对写者而言）的形式获得使用。

读/写自旋锁的使用方法类似于普通自旋锁，它们通过下面的方法初始化：

```
rwlock_t mr_rwlock = RW_LOCK_UNLOCKED;
```

然后，在读者的代码分支中使用如下函数：

```
read_lock(&mr_rwlock);
/*临界区(只读)*/
read_unlock(&mr_rwlock);
```

最后，在写者的代码分支中使用如下函数：

```
write_lock(&mr_rwlock);
/* 临界区(读写)…*/
write_unlock(&mr_lock);
```

通常情况下，读锁和写锁会位于完全分割开的代码分支中，如上例所示。

注意，不能把一个读锁“升级”为写锁。这种代码：

```
read_lock(&mr_rwlock);
write_lock(&mr_rwlock);
```

将会带来死锁，因为写锁会不断自旋，等待所有的读者释放锁，其中也包括它自己。所以当确实需要写操作时，要在一开始就请求写锁。如果写和读不能清晰地分开的话，那么使用一般的自旋锁就行了，不要使用读-写自旋锁。

多个读者可以安全地获得同一个读锁，事实上，即使一个线程递归地获得同一读锁也是安全的。这个特性使得读—写自旋锁真正成为一种有用并且常用的优化手段。如果在中断处理程序中只有读操作而没有写操作，那么，就可以混合使用“中断禁止”锁，使用read_lock()而不是read_lock_irqsave()对读进行保护。不过，你还是需要用write_lock_irqsave()禁止有写操作的中断，否则，中断里的读操作就有可能锁死在写锁上（假如读者正在进行操作，包含写操作的中断发生了，由于读锁还没有全部被释放，所以写操作会自旋，而读操作只能在包含写操作的中断返回后才能继续，释放读锁，此时死锁就发生了——译者注）。表9-4列出了针对读-写自旋锁的所有操作。

表9-4 读者-写者自旋锁方法列表

方 法	描 述
read_lock()	获得指定的读锁
read_lock_irq()	禁止本地中断并获得指定读锁
read_lock_irqsave()	存储本地中断的当前状态，禁止本地中断并获得指定读锁
read_unlock()	释放指定的读锁
read_unlock_irq()	释放指定的读锁并激活本地中断
read_unlock_irqrestore()	释放指定的读锁并将本地中断恢复到指定的前状态
write_lock()	获得指定的写锁
write_lock_irq()	禁止本地中断并获得指定写锁
write_lock_irqsave()	存储本地中断的当前状态，禁止本地中断并获得指定写锁
write_unlock()	释放指定的写锁
write_unlock_irq()	释放指定的写锁并激活本地中断
write_unlock_irqrestore()	释放指定的写锁并将本地中断恢复到指定的前状态
write_trylock()	试图获得指定的写琐；如果写锁不可用，返回非0值
rw_lock_init()	初始化指定的rwlock_t
rw_is_locked()	如果指定的锁当前已被持有，该函数返回非0值，否则返回0

在使用Linux读-写自旋锁时，最后要考虑的一点是这种锁机制照顾读比照顾写要多一点。当读锁被持有时，写操作为了互斥访问只能等待，但是，读者却可以继续成功地作占用锁。而自旋等待的写者在所有读者释放锁之前是无法获得锁的。所以，大量读者必定会使挂起的写者处于饥饿状态，在你自己设计锁时一定要记住这一点。

自旋锁提供了一种快速简单的锁实现方法。如果加锁时间不长并且代码不会睡眠（比如中断处理程序），利用自旋锁是最佳选择。如果加锁时间可能很长或者代码在持有锁时有可能睡眠，那么最好使用信号量来完成加锁功能。

9.4 信号量

Linux中的信号量是一种睡眠锁。如果有一个任务试图获得一个已经被占用的信号量时，信号量会将其推进一个等待队列，然后让其睡眠。这时处理器能重获自由，从而去执行其他代码。当持有信号量的进程[⊖]将信号量释放后，处于等待队列中的那个任务将被唤醒，并获得该信号量。

让我们再一次回到门和钥匙的例子。当某个人到了门前，他抓取钥匙，然后进入房间。最大的差异在于当另一个人到了门前，但无法得到钥匙时会发生什么情况。在这种情况下，这家伙不是在徘徊等待，而是把自己的名字写在一个列表中，然后打盹去了。当里面的人离开房间时，就在门口查看一下列表。如果列表上有名字，他就对第一个名字仔细检查，并在胸部给他一拳，叫醒他，让他进入房间。在这种方式中，钥匙（相当于信号量）继续确保一次只有一个人（相当于执行线程）进入房间（相当于临界区）。如果房间被占用，那么，那个人不是徘徊等待，而是把自己的名字写在列表（相当于等待队列）上，然后打一会儿盹（相当于在等待队列上阻塞，并且入睡），让处理器离开这里到别的地方执行代码。这就比自旋锁提供了更好的处理器利用率，因为没有把时间花费在忙等上，但是，信号量比自旋锁有更大的开销。生活总是一分为二的。

我们可以从信号量的睡眠特性得出一些有意思的结论：

- 由于争用信号量的进程在等待锁重新变为可用时会睡眠，所以信号量适用于锁会被长时间持有的情况。
- 相反，锁被短时间持有时，使用信号量就不太适宜了。因为睡眠、维护等待队列以及唤醒所花费的开销可能比锁被占用的全部时间还要长。
- 由于执行线程在锁被争用时会睡眠，所以只能在进程上下文中才能获取信号量锁，因为在中断上下文中是不能进行调度的。
- 你可以在持有信号量时去睡眠（当然你也可能并不需要睡眠），因为当其他进程试图获得同一信号量时不会因此而死锁（因为该进程也只是去睡眠而已，而你最终会继续执行的）。
- 在你占用信号量的同时不能占用自旋锁。因为在你等待信号量时可能会睡眠，而在持有自旋锁时是不允许睡眠的。

以上这些结论阐明了信号量和自旋锁在使用上的差异。在使用信号量的大多数时候，你的选择余地并不大。往往在需要和用户空间同步时，你的代码会需要睡眠，此时使用信号量是惟一的选择。由于不受睡眠的限制，使用信号量通常来说更加容易一些。如果需要在自旋锁和信号量中作选择，应该根据锁被持有的时间长短做判断。理想情况当然是所有的锁定操作都应该越短越好。但如果你

⊖ 我们将会看到，如果需要，多进程可以同时持有一个信号量。

用的是信号量，那么锁定的时间长一点也能够接受。另外，信号量不同于自旋锁，它不会禁止内核抢占，所以持有信号量的代码可以被抢占。这意味着信号量不会对调度的等待时间带来负面影响。

最后要讨论的是信号量的一个有用特性，它可以同时允许任意数量的锁持有者，而自旋锁在一个时刻最多允许一个任务持有它。信号量同时允许的持有者数量可以在声明信号量时指定。这个值称为使用者数量（usage count）或简单地叫数量(count)。通常情况下，信号量和自旋锁一样，在一个时刻仅允许有一个锁持有者。这时计数等于1，这样的信号量被称为二值信号量（因为它或者由一个任务持有，或者根本没有任务持有它）或者称为互斥信号量（因为它强制进行互斥）。另一方面，初始化时也可以把数量设置为大于1的非0值。这种情况，信号量被称为计数信号量（counting semaphone），它允许在一个时刻至多有count个锁持有者。计数信号量不能用来进行强制互斥，因为它允许多个执行线程同时访问临界区。相反，这种信号量用来对特定代码加以限制，内核中使用它的机会不多。在使用信号量时，基本上你用到的都是互斥信号量（计数等于1的信号量）。

信号量在1968年由Edsger Wybe DijKstra[⊖]提出，此后它逐渐成为一种常用的锁机制。信号量支持两个原子操作P()和V()，这两个名字来自荷兰语 Proberen和Vershogen。前者叫做测试操作（字面意思是探查），后者叫做增加操作。后来的系统把两种操作分别叫做down()和up()，Linux也遵从这种叫法。down()操作通过对信号量计数减1来请求获得一个信号量。如果结果是0或大于0，获得信号量锁，任务就可以进入临界区。如果结果是负数，任务会被放入等待队列，处理器执行其他任务。该函数如同一个动词，降低（down）一个信号量就等于获取该信号量。相反，当临界区中的操作完成后，up()操作用来释放信号量，该操作也被称作是提升（upping）信号量，因为它会增加信号量的计数值。如果在该信号量上的等待队列不为空，那么处于队列中等待的任务在被唤醒的同时会获得该信号量。

9.4.1 创建和初始化信号量

信号量的实现是与体系结构相关的，具体实现定义在文件<asm/semaphore.h>中。struct semaphore类型用来表示信号量。可以通过以下方式静态地声明信号量：

```
static DECLARE_SEMAPHORE_GENERIC (name,count)
```

其中name是信号量变量名，count是信号量的使用者数量。创建更为普通的互斥信号量可以使用以下快捷方式：

```
static DECLARE_MUTEX(name);
```

name同样指信号量变量名。更常见的情况是，信号量作为一个大数据结构的一部分被动态创建。此时，你只有指向该动态创建的信号量的间接指针，你可以使用如下函数来对它进行初始化：

```
sema_init(sem,count);
```

sem是指针，count是信号量的使用者数量。与前面类似，初始化一个动态创建的互斥信号量时使用以下函数：

```
init_MUTEX(sem);
```

我不明白为什么“mutex”在init_MUTEX()中是大写，或者为什么“init”在这个函数名中放

⊖ Dijkstra博士（1930~2002）是计算机科学史上最为成功的科学家之一，他在操作系统设计、算法理论和信号量概念的创建等诸多领域做出了卓越的贡献。他出生于荷兰的鹿特丹，曾在得克萨斯大学任教15年。不过他恐怕对Linux中间夹杂的大量GOTO语句不太满意。

在前面，而在sema_init()中放在后面。不过，我的确知道，这看起来让人无话可说，我为这种不一致而感到抱歉。但是，我希望，在你读过第7章之后，内核符号的命名没有打闷你。

9.4.2 使用信号量

函数down_interruptible()试图获取指定的信号量，如果获取失败，它将以TASK_ INTERRUPTIBLE状态进入睡眠。回忆第3章的内容，这种进程状态意味着任务可以被信号唤醒，一般来说这是件好事。如果进程在等待获取信号量的时候接收到了信号，那么该进程就会被唤醒，而函数down_interruptible()会返回-EINTR。另外一个函数down()会让进程在TASK_UNINTERRUPTIBLE状态下睡眠。你应该不希望这种情况发生，因为这样一来，进程在等待信号量的时候就不再响应信号了。因此，使用down_interruptible()比使用down()更为普遍。也许你会觉得这两个函数名字起得有点不恰当，的确，这些命名并不很理想。

使用down_trylock()函数，你可以尝试以堵塞方式来获取指定的信号量。在信号量已被占用时，它立刻返回非0值；否则，它返回0，而且让你成功持有信号量锁。

要释放指定的信号量，需要调用up()函数。请看例子：

```
/* 定义并声明一个信号量，名字为mr_sem,  用于信号量计数*/
static DECLARE_MUTEX(mr_sem);

/* 试图获取信号量 ... */
if (down_interruptible(&mr_sem)){
        /* 信号被接收 , 信号量还未获取 */
}

/* 临界区... */

/* 释放给定的信号量 */
up(&mr_sem);
```

表9-5 给出了针对信号量的操作的完整列表。

表9-5 信号量方法列表

方 法	描 述
sema_init(struct semaphore *,int)	以指定的计数值初始化动态创建的信号量
init_MUTEX(struct semaphore *)	以计数值1初始化动态创建的信号量
init_MUTEX_LOCKED(struct semaphore *)	以计数值0初始化动态创建的信号量(初始为加锁状态)
down_interruptible(struct semaphore *)	以试图获得指定的信号量，如果信号量已被争用，则进入可中断睡眠状态
down(struct semaphore *)	以试图获得指定的信号量，如果信号量已被争用，则进入不可中断睡眠状态
down_trylock(struct semaphore *)	以试图获得指定的信号量，如果信号量已被争用，则立刻返回非0值
up(struct semaphore *)	以释放指定的信号量，如果睡眠队列不空，则唤醒其中一个任务

9.5 读-写信号量

与自旋锁一样，信号量也有区分读-写访问的可能。与读-写自旋锁和普通自旋锁之间的关系

差不多，读-写信号量也要比普通信号量更具优势。

读-写信号量在内核中是由rw_semaphore结构表示的，定义在文件<linux/rwsem.h>中。通过以下语句可以创建静态声明的读-写信号量：

```
static DECLARE_RWSEM(name);
```

其中name是新信号量名。

动态创建的读-写信号量可以通过以下函数初始化：

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

所有的读-写信号量都是互斥信号量（也就是说，它们的引用计数等于1）。只要没有写者，并发持有读锁的读者数不限。相反，只有惟一的写者（在没有读者时）可以获得写锁。所有读-写锁的睡眠都不会被信号打断，所以它只有一个版本的down()操作。例如：

```
static DECLARE_RWSEM(mr_rwsem);

/* 试图获取信号量用于读 ... */
down_read(&mr_rwsem);

/*临界区 (只读)...*/
/* 释放信号量 */

up_read(&mr_rwsem);

/* ... */

/* 试图获取信号量用于写 ... */
down_write(&mr_rwsem);

/*临界区 (读和写)...*/

/* 释放信号量 */
up_write(&mr_sem);
```

与标准信号量一样，读-写信号量也提供了down_read_trylock()和down_write_trylock()方法。这两个方法都需要一个指向读-写信号量的指针作为参数。如果成功获得了信号量锁，它们返回非0值；如果信号量锁被争用，则返回0。要小心——不知道为什么要这样——这与普通信号量的情形完全相反。

读-写信号量相比读-写自旋锁多一种特有的操作：downgrade_writer()。这个函数可以动态地将获取的写锁转换为读锁。

读-写信号量和读-写自旋锁一样，除非代码中的读和写可以明白无误地分割开来，否则最好不使用它。再强调一次，读者－写者机制使用是有条件的，只有在你的代码可以自然地界定出读／写时才有价值。

9.6 自旋锁与信号量

了解何时使用自旋锁，何时使用信号量对编写优良代码很重要，但是多数情况下，并不需要太多的考虑，因为在中断上下文中只能使用自旋锁，而在任务睡眠时只能使用信号量。表9-6回顾

了一下各种锁的需求情况。

表9-6 使用什么：自旋锁与信号量的比较

需　求	建议的加锁方法
低开销加锁	优先使用自旋锁
短期锁定	优先使用自旋锁
长期加锁	优先使用信号量
中断上下文中加锁	使用自旋锁
持有锁需要睡眠	使用信号量

9.7 完成变量

如果在内核中一个任务需要发出信号通知另一任务发生了某个特定事件，利用完成变量（completion variable）是使两个任务得以同步的简单方法。如果一个任务要执行一些工作时，另一个任务就会在完成变量上等待。当这个任务完成工作后，会使用完成变量去唤醒在等待的任务。这听起来很像一个信号量，的确如此——思想是一样的。事实上，完成变量仅仅提供了代替信号量的一个简单的解决方法。例如，当子进程执行或者退出时，vfork() 系统调用使用完成变量唤醒父进程。

完成变量由结构completion表示，定义在<linux/completion.h>中。通过以下宏静态地创建完成变量并初始化它：

```
DECLARE_COMPLETION(mr_comp);
```

通过 init_completion()动态创建并初始化完成变量。

在一个指定的完成变量上，需要等待的任务调用wait_for_completion()来等待特定事件。当特定事件发生后，产生事件的任务调用complete()来发送信号唤醒正在等待的任务。表9-7列出了完成变量的方法。

表9-7 完成变量方法

方　法	描　述
init_completion(struct completion *)	初始化指定的动态创建的完成变量
wait_for_completion(struct completion *)	等待指定的完成变量接受信号
complete(struct completion *)	发信号唤醒任何等待任务

使用完成变量的例子可以参考kernel/sched.c和kernel/fork.c。完成变量的通常用法是，将完成变量作为数据结构中的一项动态创建，而完成数据结构初始化工作的内核代码将调用wait_for_completion()进行等待。初始化完成后，初始化函数调用completion()唤醒在等待的内核任务。

9.8 BKL

欢迎来到内核的原始混沌时期。BKL（大内核锁）是一个全局自旋锁，使用它主要是为了方便实现从Linux最初的SMP过渡到细粒度加锁机制。我们下面来介绍BKL的一些有趣的特性：

- 持有BKL的任务仍然可以睡眠。因为当任务无法被调度时，所加锁会自动被丢弃；当任务被调度时，锁又会被重新获得。当然，这并不是说，当任务持有BKL时，睡眠是安全的，仅仅是可以这样做，因为睡眠不会造成任务死锁。

- BKL是一种递归锁。一个进程可以多次请求一个锁，并不会像自旋锁那样产生死锁现象。
- BKL可以用在进程上下文中。
- BKL是有害的。

这些特性有助于2.0版本的内核向2.2版本过渡。在SMP支持被引入到2.0版本时，内核中一个时刻上只能有一个任务运行（当然，经过长期发展，现在内核已经被很好的线程化了）。2.2版本的目标是允许多处理器在内核中并发执行程序。引入BKL是为了使到细粒度加锁机制的过渡更容易些，虽然当时BKL对内核过渡很有帮助，但是目前它已成为内核可扩展性的障碍了[⊖]。

在内核中不鼓励使用BKL。事实上，新代码中不再使用 BKL,但是这种锁仍然在部分内核代码中得到沿用。所以我们仍然需要理解BKL以及它的接口。除了前面提到的以外，bkl的使用方式和自旋锁类似。函数lock_kernel()请求锁，unlock_kernel()释放锁。一个执行线程可以递归的请求锁，但是，释放锁时也必须调用同样次数的unlock_kernel()操作，在最后一个解锁操作完成后，锁才会被释放。函数kernel_locked()检测锁当前是否被持有，如果被持有，返回一个非0值，否则返回0。这些接口被声明在文件<linux/smp_lock.h>中，简单的用法如下：

```
lock_kernel();

/* 临界区，对所有其他的BLK用户进行同步
 * 注意，你可以安全地在此睡眠，锁会悄无声息地被释放
 * 当你的任务被重新调度时，锁又会被悄无声息地获取
 * 这意味着你不会处于死锁状态，但是，如果你需要锁保护这里的数据
 * 你还是不需要睡眠
 */

unlock_kernel();
```

BKL在被持有时同样会禁止内核抢占。在单一处理器内核中，BKL并不执行实际的加锁操作。表9-8列出了所有BKL函数。

表9-8 BKL函数列表

函 数	描 述
lock_kernel()	获得BKL
unlock_kernel()	释放BKL
kernel_locked()	如果锁被持有返回非0值，否则返回0（UP总是返回非零）

对于BKL最主要的问题是确定BKL锁保护的到底是什么。多数情况，BKL更像是保护代码（比如“它保护对foo()函数的调用者进行同步”）而不保护数据（比如“保护结构foo”）。这个问题给利用自旋锁取代BKL造成了很大困难，因为难以判断BKL到底锁的是什么，更难的是发现所有使用BKL的用户之间的关系。

Seq 锁

Seq锁是在2.6内核版本中才引入的一种新型锁。这种锁提供了一种很简单的机制，用于读写

⊖ 虽然这并不可怕，但还是有人认为这是内核恶魔的化身。

共享数据。实现这种锁主要依靠一个序列计数器。当有疑义的数据被写入时，会得到一个锁，并且序列值会增加。在读取数据之前和之后，序列号都被读取。如果读取的序列号值相同，说明在读操作进行的过程中没有被写操作打断过。此外，如果读取的值是偶数，那么就表明写操作没有发生（要明白因为锁的初值是0，所以写锁会使值成奇数，释放的时候变成偶数）。定义一个seq锁的形式为：

```
seqlock_t mr_seq_lock = SEQLOCK_UNLOCKED;
```

然后，写锁的方法如下：

```
write_seqlock(&mr_seq_lock);
/*写锁被获取 ...*/
write_sequnlock(&mr_seq_lock);
```

这和普通自旋锁类似。不同的情况发生在读时，与自旋锁有很大不同：

```
unsigned long seq;

do{
        seq = read_seqbegin(&mr_seq_lock);
        /*读这里的数据 ...*/
}while(read_seqretry(&mr_seq_lock,seq));
```

在多个读者和少数写者共享一把锁的时候，seq锁有助于提供一种非常轻量级和具有可扩展性的外观。但是seq 锁对写者更有利。只要没有其他写者，写锁总是能够被成功获得。读者不会影响写锁，这点和读者－写者自旋锁及信号量一样。另外，挂起的写者会不断地使得读操作循环（前一个例子），直到不再有任何写者持有锁为止。

9.9 禁止抢占

由于内核是抢占性的，内核中的进程在任何时刻都可能停下来以便另一个具有更高优先权的进程运行。这意味着一个任务与被抢占的任务可能会在同一个临界区内运行。为了避免这种情况，内核抢占代码使用自旋锁作为非抢占区域的标记。如果一个自旋锁被持有，内核便不能进行抢占。因为内核抢占和SMP面对相同的并发问题，并且内核已经是SMP安全的（SMP_safe），因此，这种简单的变化使得内核也是抢占安全的（preempt_safe）。

或许这就是我们希望的。实际中，某些情况并不需要自旋锁，但是仍然需要关闭内核抢占。出现得最频繁的情况就是每个处理器上的数据。如果数据对每个处理器是惟一的，那么，这样的数据可能就不需要使用锁来保护，因为数据只能被一个处理器访问。如果自旋锁没有被持有，内核又是抢占式的，那么一个新调度的任务就可能访问同一个变量，如下所示：

```
任务A对每个处理器中未被锁保护的变量foo进行操作
任务A被抢占
任务B被调度
任务B操作变量foo
任务B完成
任务A被调度
任务A继续操作变量foo
```

这样，即使这是一个单处理器计算机，变量foo也会被多个进程以伪并发的方式访问。通常，

这个变量会请求得到一个自旋锁（防止多处理器机器上的真并发）。但是如果这是每个处理器上独立的变量，可能就不需要锁。

为了解决这个问题，可以通过preempt_disable()禁止内核抢占。这是一个可以嵌套调用的函数，可以调用任意次。每次调用都必须有一个相应的preempt_enable()调用。当最后一次preempt_enable()被调用后，内核抢占才重新启用。例如：

```
preempt_disable();
/*抢占被禁止...*/
preempt_enable();
```

抢占计数存放着被持有锁的数量和preempt_disable()的调用次数，如果计数是0，那么内核可以进行抢占，如果为1或更大的值，那么，内核就不会进行抢占。这个计数非常有用一它是一种对原子操作和睡眠很有效的调试方法。函数preempt_count()返回这个值。表9-9列出了内核抢占相关的函数。

表9-9 内核抢占的相关函数

函 数	描 述
preempt_disable()	增加抢占计数值，从而禁止内核抢占
preempt_enable()	减少抢占计数，并当该值降为0时检查和执行被挂起的需调度的任务
preempt_enable_no_resched()	激活内核抢占但不再检查任何被挂起的需调度任务
preempt_count()	返回抢占计数

为了用更简洁的方法解决每个处理器上的数据访问问题，可以通过get_cpu()获得处理器编号（假定是用这种编号来对每个处理器的数据进行索引的）。这个函数在返回当前处理器号前首先会关闭内核抢占。

```
int cpu ;

/* 禁止内核抢占，并将cpu设置为当前处理器 */
cpu= get_cpu();
/* 对每个处理器的数据进行操作 ...*/

/* 再给予内核抢占性，"cpu"可改变故它不再效 "cpu" can change and so is no longer valid */
put_cpu();
```

9.10 顺序和屏障

当处理多处理器之间或硬件设备之间的同步问题时，有时需要在你的程序代码中以指定的顺序发出读内存（读入）和写内存（存储）指令。在和硬件交互时，时常需要确保一个给定的读操作发生在其他读或写操作之前。另外，在多处理器上，可能需要按写数据的顺序读数据（通常确保后来以同样的顺序进行读取）。但是编译器和处理器为了提高效率，可能对读和写重新排序[⊖]，这样无疑使问题复杂化了。幸好，所有可能重新排序和写的处理器提供了机器指令来确保顺序要求。同样也可以指示编译器不要对给定点周围的指令序列进行重新排序。这些确保顺序的指令称做*屏障*（barrier）。

⊖ 虽然Intel x86处理器不会对写进行重新排序，也就是说，它不进行打乱顺序的存储，但是其他处理器会这么做。

注意，在某些处理器上，存在以下代码：

```
a = 1;
b = 2;
```

有可能会在a中存放新值之前就在b中存放新值。编译器和处理器都看不出a和b之间的关系。编译器会在编译时按这种顺序编译；这种顺序会是静态的，编译的目标代码就只把a放在b之前。但是，处理器会重新动态排序，因为处理器在执行指令期间，会在取指令和分派时，把表面上看似无关的指令按自认为最好的顺序排列。大多数情况下，这样的排序是最佳的，因为a和b之间没有明显的关系。尽管程序员知道什么是最好的顺序。

尽管前面的例子可能被重新排序,但是处理器和编译器绝不会对下面的代码重新排序：

```
a = 1;
b = a;
```

此处a和b均为全局变量。因为a与b之间有明确的数据依赖关系。但是不管是编译器还是处理器都不知道其他上下文中的相关代码。偶然情况下，有必要让写操作被其他代码识别，也让你所期望的指定顺序之外的代码识别。这种情况常常发生在硬件设备上，但是在多处理器机器上也很常见。

rmb()方法提供了一个“读”内存屏障，它确保跨越rmb()的载入动作不会发生重排序。也就是说，在rmb()之前的载入操作不会被重新排在该调用之后，同理，在rmb()之后的载入操作不会被重新排在该调用之前。

wmb()方法提供了一个“写”内存屏障，这个函数的功能和rmb()类似，区别仅仅是它是针对存储而非载入——它确保跨越屏障的存储不发生重排序。

mb()方法既提供了读屏障也提供了写屏障。载入和存储动作都不会跨越屏障重新排序。这是因为一条单独的指令（通常和rmb()使用同一个指令）既可以提供载入屏障，也可以提供存储屏障。

read_barrier_depends()是rmb()的变种，它提供一个读屏障，但是仅仅是针对后续读操作所依靠的那些载入。因为屏障后的读操作依赖于屏障前的读操作，因此，该屏障确保屏障前的读操作在屏障后的读操作之前完成。明白了吗？基本上说，该函数设置一个读屏障，如rmb()，但是只针对特定的读——也就是那些相互依赖的读操作。在有些体系结构上，read_barrier_depends()比rmb()执行得快，因为它仅仅是个空操作，实际并不需要。

看看使用了mb()和rmb()的一个例子，其中a 的初始值是1，b的初始值是2。

线程1	线程 2
a = 3;	-
mb();	-
b = 4;	c = b;
-	rmb();
-	d = a;

如果不使用内存屏障，在某些处理器上，c可能接受了b的新值，而d接收了a原来的值。比如 c可能等于4（正是我们希望的），然而d可能等于1（不是我们希望的）。使用mb()能确保a和b按照预定的顺序写入，而rmb()确保c和d按照预定的顺序读取。

这种重排序的发生是因为现代处理器为了优化其传送管道（pipeline），打乱了分派和提交指令的顺序。如果上例中读入a、b时的顺序被打乱的话，又会发生什么情况呢？rmb()或wmb()函数

相当于指令，它们告诉处理器在继续执行前提交所有尚未处理的载入或存储指令。

看一个类似的例子，但是其中一个线程用read_barrier_depends代替了rmb()。例子中a 的初始值是1，b是2，p是&b。

线程 1	线程 2
a = 3;	-
mb();	-
p = &a;	pp = p;
-	read_barrier_depends();
-	b = *pp;

再一次声明，如果没有内存屏障，有可能在pp被设置成p前，b就被设置为pp了。由于载入*pp依靠载入p，所以read_barrier_depends()提供了一个有效的屏障。虽然使用rmb()同样有效，但是因为读是数据相关的，所以我们使用read_barrier_depends()可能更快。注意，不管在哪种情况下，左边的线程都需要mb()操作来确保预定的载入或存储顺序。

宏 smp_rmb()、smp_wmb()、smp_mb()和smp_read_barrier_depends()提供了一个有用的优化。在SMP内核中它们被定义成常用的内存屏障，而在单处理机内核中，它们被定义成编译器的屏障。对于SMP系统，在有顺序限定要求时，可以使用SMP的变种。

barrier()方法可以防止编译器跨屏障对载入或存储操作进行优化。编译器不会重新组织存储或载入操作而防止改变C代码的效果和现有数据的依赖关系。但是，它不知道在当前上下文之外会发生什么事。例如，编译器不可能知道有中断发生，这个中断有可能在读取正在被写入的数据。这时就要求存储操作发生在读取操作前。 前面讨论的内存屏障可以完成编译器屏障的功能，但是编译器屏障要比内存屏障轻量（它实际上是轻快的 ）得多。实际上，编译器屏障几乎是空闲的，因为它只是防止编译器可能重排指令。

表9-10给出了内核中所有体系结构提供的完整的内存和编译器屏障方法。

表9-10 内存和编译器屏障方法

屏 障	描 述
rmb()	阻止跨跃屏障的载入动作发生重排序
read_barrier_depends()	阻止跨越屏障的具有数据依赖关系的载入动作重排序
wmb()	阻止跨越屏障的存储动作发生重排序
mb()	阻止跨越屏障的载入和存储动作重排序
smp_rmb()	在SMP上提供rmb()功能，在UP上提供barrier()功能
smp_read_barrier_depends()	在SMP上提供read_barrier_depends()功能，在UP上提供barrier()功能
smp_wmb()	在SMP上提供wmb()功能，在UP上提供barrier()功能
smp_mb()	在SMP上提供mb()功能，在UP上提供barrier()功能
barrier()	阻止编译器跨屏障对载入或存储操作进行优化

注意，对于不同体系结构，屏障的实际效果差别很大。例如，如果一个体系结构不执行打乱存储（比如Intel x86芯片就不会），那么wmb()就什么也不做。但应该为最坏的情况（即排序能力最弱的处理器）使用恰当的内存屏蔽，这样代码才能在编译时执行针对体系结构的优化。

9.11 小结

本章应用了上一章的概念和原理，这使得你能理解Linux内核用于同步和并发的具体方法。我们一开始先讲述了最简单的确保同步的方法——原子操作，然后考察了自旋锁，这是内核中最普通的锁，它提供了轻量级单独持有者的锁，即争用时忙等。我们还讨论了信号量，这是一种睡眠锁，至于专用的加锁原语像完成变量、seq锁，只是稍稍提及。我们取笑BKL，考察了禁止抢占，并理解了屏障，它曾难以驾驭。

以这两章的同步方法为基础，现在你可以编写防止竞争条件的内核代码，并选取最好的工具确保想要的同步。

第⑩章

定时器和时间管理

时间管理在内核中占有非常重要的地位。相对于事件驱动[⊖]而言，内核中有大量的函数都是基于时间驱动的。其中有些函数是周期执行的，像对调度程序中的运行队列进行平衡调整或对屏幕进行刷新这样的函数，都需要定期执行，比如说，每秒执行100次；有些函数，比如需要推后执行的磁盘I/O操作等，则需要等待一个相对时间后才运行——比如说，内核会在500毫秒后再执行某个任务。除了上述两种函数需要内核提供时间外，内核还必须管理系统的运行时间以及当前日期和时间。

请注意相对时间和绝对时间之间的差别。如果某个事件在5秒后被调度执行，那么系统所需要的不是绝对时间——而是相对时间（比如，相对现在起5秒后）；相反，如果要求管理当前日期和当前时间，则内核不但要计算流逝的时间而且还要计算绝对时间。所以这两种时间概念对内核时间管理来说都至关重要。

另外还请注意周期性产生的事件与推迟执行的事件之间的差别。周期性产生的事件——比如每10毫秒一次——都是由系统定时器驱动的。系统定时器是一种可编程硬件芯片，它能以固定频率产生中断。该中断就是所谓的定时器中断，它所对应的中断处理程序负责更新系统时间，还负责执行需要周期性运行的任务。系统定时器和时钟中断处理程序是Linux系统内核管理机制中的中枢，本章将着重讨论它们。

另外一个关注的焦点是动态定时器——一种用来推迟执行程序的工具。比如说，如果软驱马达在一定时间内都未活动，那么软盘驱动程序会使用动态定时器关闭软驱马达。内核可以动态创建或销毁动态定时器。本章将介绍动态定时器在内核中的实现，同时给出在内核代码中可以供使用的定时器接口。

10.1 内核中的时间概念

时间概念对计算机来说有些模糊，事实上内核必须在硬件的帮助下才能计算和管理时间。硬件为内核提供了一个系统定时器用以计算流逝的时间，该时钟在内核中可看成是一个电子时间资源，比如数字时钟或处理器频率等。系统定时器以某种频率自行触发（经常被称为击中（hitting）或射中（popping））时钟中断，该频率可以通过编程预定，称作节拍率（tick rate）。当时钟中断发生时，内核就通过一种特殊的中断处理程序对其进行处理。

因为预编的节拍率对内核来说是可知的,所以内核知道连续两次时钟中断的间隔时间。这个间隔时间就称为节拍（tick），它等于节拍率分之一（one-over-the-tick-rate）秒。正如你所看到的，内核就是靠这种已知的时钟中断间隔来计算墙上时间和系统运行时间。墙上时间——也就是实际

⊖ 更准确地讲，时间驱动事件也属于事件驱动的一种——时间的流逝本身就是一种事件。然而，由于时间驱动事件的频率非常高，且对内核而言至关重要，因此，本章中我们仅仅分析一下时间驱动事件。

时间——对用户空间的应用程序来说是最重要的。内核通过控制时钟中断维护实际时间，另外内核也为用户空间提供了一组系统调用以获取实际日期和实际时间。系统运行时间——自系统启动开始所经过的时间——对用户空间和内核都很有用，因为许多程序都必须清楚流逝过的时间。通过两次（现在和以后）读取运行时间再计算它们的差，就可以得到相对的流逝过的时间了。

时钟中断对于管理操作系统尤为重要，大量内核函数的生命周期都离不开流逝的时间的控制。下面给出一些利用时间中断周期执行的工作：

- 更新系统运行时间。
- 更新实际时间。
- 在smp系统上，均衡调度程序中各处理器上的运行队列。如果运行队列负载不均衡的话，尽量使它们均衡（在第4章中讨论过）。
- 检查当前进程是否用尽了自己的时间片。如果用尽，就重新进行调度（在第4章中讨论过）。
- 运行超时的动态定时器。
- 更新资源消耗和处理器时间的统计值。

这其中有些工作在每次的时钟中断处理程序中都要被处理——也就是说这些工作随时钟的频率反复运行。另一些也是周期性的执行，但只需要每*n*次时钟中断运行一次，也就是说，这些函数在累计了一定数量的时钟节拍数时才被执行。在10.5小节中，我们将详细讨论时钟中断处理程序。

10.2 节拍率：HZ

系统定时器频率（节拍率）是通过静态预处理定义的，也就是HZ（赫兹），在系统启动时按照Hz值对硬件进行设置。体系结构不同，HZ的值也不同，实际上，对于某些体系结构来说，甚至是机器不同，它的值都会不一样。

内核在文件<asm/param.h>中定义了HZ的实际值，节拍率就等于HZ，周期为1/HZ秒。比如，i386体系结构在include/asm-i386/param.h中对HZ值定义如下：

```
#define HZ 1000 /*内核时间频率*/
```

可以看到i386体系结构中系统定时器频率为1000Hz，也就是说每秒钟1000次（每毫秒产生一次）。但其他体系结构的节拍率绝大多数都等于100。表10-1列出了对应各种体系结构的节拍率。

表10-1 时钟中断频率

体系结构	频率（单位：Hz）	体系结构	频率（单位：Hz）
alpha	1024	parisc	100或1000
arm	100	ppc	1000
cris	100	ppc64	1000
h8300	100	s390	100
i386	1000	sh	100或1000
ia64	32或1024①	sparc	100
m68k	100	sparc64	1000
m68knommu	50、100或1000	um	100
mips	100	v850	24、100或122
mips64	100或1000	x86-64	1000

① 模拟IA-64机器的节拍率为32Hz，而真实IA-64机器的节拍率是1024Hz。

编写内核代码时，不要认为HZ值是一个固定不变的值。这不是一个常见的错误，因为大多数体系结构的节拍率都是可调的。但是在过去，只有Alpha一种机型的节拍率不等于100，所以很多本该使用HZ的地方，都错误地在代码中直接硬编码（hard-code）成100这个值。稍后，我们会给出内核代码中使用HZ的例子。

正如我们所看到的，时钟中断能处理许多内核任务,所以它对内核来说极为重要。事实上，内核中的全部时间概念都来源于周期运行的系统时钟。所以选择一个合适的频率，就如同在人际交往中建立和谐关系一样，必须取得各方面的折衷。

理想的HZ值

自Linux问世以来，i386体系结构中时钟中断频率就设定为100Hz，但是在2.5开发版内核中，中断频率被提高到1000Hz。当然，是否应该提高频率（如同其他绝大多数事情一样）是饱受争议的。由于内核中众多子系统都必须依赖时钟中断工作，所以改变中断频率必然会对整个系统造成很大的冲击。但是，任何事情总是有两面性的，我们接下来就来分析系统定时器使用高频率与使用低频率各有哪些优劣。

提高节拍率意味着时钟中断产生得更加频繁，所以中断处理程序也会更频繁地执行。如此一来会给整个系统带来如下好处：

- 更高的时钟中断解析度（resolution）可提高时间驱动事件的解析度。
- 提高了时间驱动事件的准确度（accuracy）。

提高节拍率等同于提高中断解析度。比如HZ=100的时钟的执行粒度为10毫秒，即系统中的周期事件最快为每10毫秒运行一次，而不可能有更高的精度[⊖]，但是当HZ=1000时，解析度就为1毫秒——精细了10倍。虽然内核可以提供频度为1毫秒的时钟，但是并没有证据显示对系统中所有程序而言，频率为1000Hz的时钟率相比频率为100Hz的时钟都更合适。

另外，提高解析度的同时也提高了准确度。假定内核在某个随机时刻触发定时器，而它可能在任何时间超时，但由于只有在时钟中断到来时才可能执行它，所以平均误差大约为半个时钟中断周期。比如说，如果时钟周期为HZ=100，那么事件平均在设定时刻的+/-5毫秒内发生，所以平均误差为5毫秒。如果HZ=1000，那么平均误差可降低到0.5毫秒——准确度提高了10倍。

更高的时钟中断频度和更高的准确度又会带来如下优点：

- 内核定时器能够以更高的频度和更高的准确度（它带来了大量的好处，下一条便是其中之一）运行。
- 依赖定时值执行的系统调用，比如poll()和select()，能够以更高的精度运行。
- 对诸如资源消耗和系统运行时间等的测量会有更精细的解析度。
- 提高进程抢占的准确度。

对poll()和select()超时精度的提高会给系统性能带来极大的好处。提高精度可以大幅度提高系统性能。频繁使用上述两种系统调用的应用程序往往在等待时钟中断上浪费大量的时间，而事实上，定时值可能早就超时了。回忆一下，平均误差（也就是，可能浪费的时间）可是时钟中断周

⊖ 这里所说的是计算机意义上的精度，而不是科学意义上的精度。科学意义上的精度是统计反复性的量度，在计算机领域内，精度是表示一个值的有效位的个数。

期的一半。

更高的准确率也使进程抢占更准确，同时还会加快调度响应时间。第4章中提到过，时钟中断处理程序负责减少当前进程的时间片计数。当时间片计数跌到0时，而又设置了need_resched 标志的话，内核便立刻重新运行调度程序。假定有一个正在运行的进程，它的时间片只剩下2毫秒了，此时如果调度程序要求抢占该进程，然后去运行另一个新进程，然而该抢占行为不会在下一个时钟中断到来前发生，也就是说，在这2毫秒内不可能进行抢占。实际上，对于频率为100Hz的时钟来说，最坏要在10毫秒后，当下一个时钟中断到来时才能进行抢占，所以新进程也就可能要比要求的晚10毫秒才能执行。当然，进程之间也是平等的，因为对所有的进程都是一视同仁的，调度起来都不是很准确——但关键不在于此。问题在于由于耽误了抢占，所以对于类似于填充音频缓冲区这样有严格时间要求的任务来说，结果是无法接受的。如果将节拍率提高到1000Hz，在最坏情况下，也能将调度延误时间降低到1毫秒，平均情况下，能降到0.5毫秒左右。

现在该谈谈另一面了，提高节拍率会产生负作用。事实上，提高节拍率即把节拍率提高到1000Hz(甚至更高)会带来一个大问题：节拍率越高，意味着时钟中断频率越高，也就意味着系统负担越重。因为处理器必须花时间来执行时钟中断处理程序，所以节拍率越高,中断处理程序占用的处理器的时间越多。这样不但减少了处理器处理其他工作的时间，而且还会更频繁地打乱处理器高速缓存。负载造成的影响值得进一步的探讨。将时钟频率从100Hz提高到1000Hz必然会使时钟中断的负载增加10倍。可是增加前的系统负载又是多少呢？最后的结论是：至少在现代计算机系统上，时钟频率为1000Hz不会导致难以接受的负担，并且不会对系统性能造成较大的影响。尽管如此，在2.6内核中还是允许在编译内核时选定不同的Hz值。

无节拍的OS？

也许你疑惑操作系统是否一定要有固定时钟，能不能设计出没有节拍的操作系统呢？回答是肯定的，这是可能的，但不是个好主意！

操作系统并非一定需要固定的时钟中断，实际上，内核可以使用动态编程定时器操作挂起事件。但如果这样，马上会引起巨大的定时器负担，所以更理想的办法是只使用一个定时器，通过编程，让它在前一个事件到期时开始计时，在它被触发执行后，再设置定时器为下一个事件服务，如此反复。利用这种方法就不再需要周期性的时钟，也不需要Hz值了。

但这样做需要解决两个问题，第一个是如何管理和节拍相关的某些概念，内核如何跟踪系统的相对运行时间。这个还好处理一点，但更难处理的是第二个问题——如何克服管理动态定时器所带来的负担。即使使用相对优化的算法——实现起来很复杂，也很难减轻负担。这样做引起的负担和带来的复杂性实在让处理器难以承受，所以Linux内核没有采用上述方法。不过，确实有人曾做过这样的尝试，结果也很有趣——如果有兴趣的话可以在网上搜索相关文档。

10.3 jiffies

全局变量jiffies用来记录自系统启动以来产生的节拍的总数。启动时，内核将该变量初始化为0，此后，每次时钟中断处理程序都会增加该变量的值。因为一秒内时钟中断的次数等于Hz，所以jiffies一秒内增加的值也就为Hz。系统运行时间以秒为单位计算，就等于jiffies/Hz。

Jiffy的语源

术语jiffy起源是未知的。据说这个短语起源于18世纪的英国。最初，jiffy所指含义不明确，但简单地表示时间周期。

在科学应用中，jiffy表示各种时间间隔，通常指10ms。在物理学中，jiffy有时表示光传播某一特定距离（大抵1英尺，或者1厘米，或者跨越1个核子）所花的时间。

在计算机工程中，jiffy常常是指两次连续的时钟周期之间的时间。在电机工程中，jiffy是完成一次AC（交流电）周期的时间。在美国，这是1/60秒。

在操作系统中，尤其是在Unix中，jiffy是两次连续的时钟节拍之间的时间。历史上，这是10毫秒。但是，我们在本章已经看到，jiffy在Linux中已经有所变化。

jiffies定义于文件<linux/jiffies.h>中：

```
extern unsigned long volatile jiffies;
```

下一节，我们会看到它的实际定义，它看起来有点特殊。现在我们先来看一些用到jiffies的内核代码。将以秒为单位的时间转化为jiffies：

```
(seconds * HZ)
```

相反，将jiffies转换为以秒为单位的时间：

```
(jiffies/HZ)
```

比较而言，内核中将秒转换为jiffies用得多一些，比如代码经常需要设置一些将来的时间：

```
unsigned long time_stamp = jiffies;    /*现在*/
unsigned long next_tick = jiffies+_1; /* 从现在开始1个节拍*/
unsigned long later = jiffies+5*HZ;  /*从现在开始5秒*/
```

上面这种操作经常会用在内核和用户空间进行交互的时候，而内核本身很少用到绝对时间。

注意jiffies类型为无符号长整型（unsigned long），用其他任何类型存放它都不正确。

10.3.1 jiffies的内部表示

jiffies变量总是无符号长整数（unsigned long），因此，在32位体系结构上是32位，在64位体系结构上是64位。32位的jiffies变量，在时钟频率为100Hz的情况下，497天后会溢出。如果频率为1000Hz，49.7天后就会溢出。而使用64位的jiffies变量，任何人都别指望会看到它溢出。

由于性能与历史的原因，主要还是考虑到与现有内核代码的兼容性，内核开发者希望jiffies依然为unsigned long。有一些巧妙的思想和少数神奇的链接程序扭转了这一局面。

前面已经看到，jiffies定义为unsigned long：

```
extern unsigned long volatile jiffies;
```

第二个变量也定义在 <linux/jiffies.h>中:

```
extern u64 jiffies_64;
```

ld(1) 脚本用于连接主内核映像 (在x86上位于arch/i386/kernel/vmlinux.lds.S中)，然后用jiffies_64变量的初值覆盖jiffies 变量:

```
jiffies = jiffies_64;
```

因此， jiffies 取整个64位jiffies_64变量的低32位。代码可以完全像以前一样继续访问jiffies。

因为大多数代码只不过使用jiffies存放流失的时间，因此，也就只关心低32位。不过，时间管理代码使用整个64位，以此来避免整个64位的溢出。

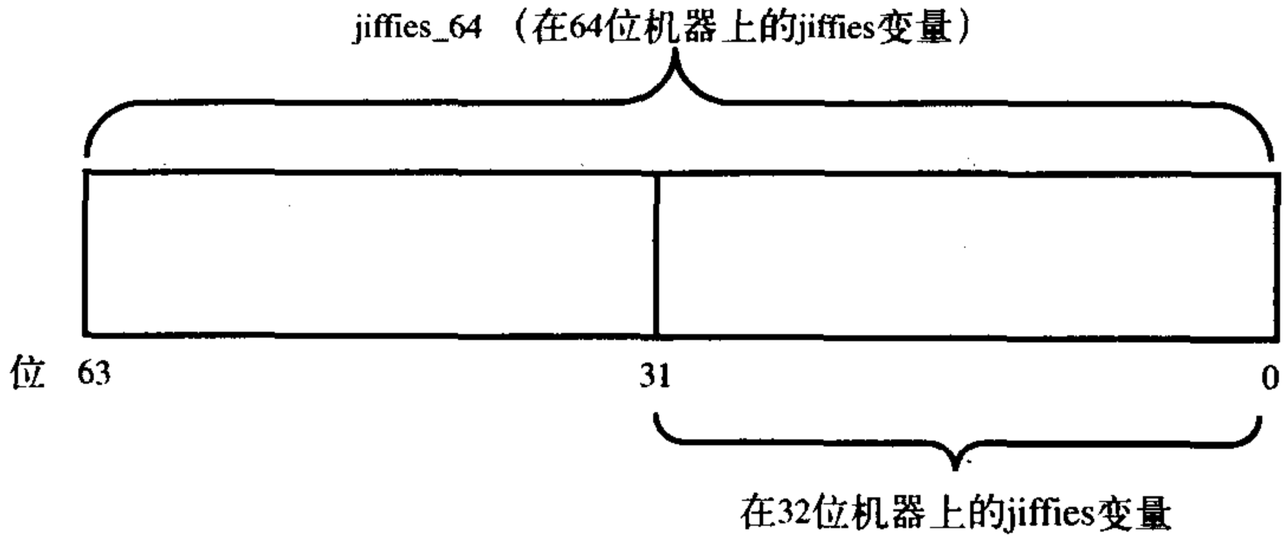


图10-1 jiffies在jiffies_64上的布局

访问jiffies的代码只会读取jiffies_64的低32位。通过get_jiffies_64()函数，就可以读取整个64位数值[⊖]。但是这种需求很少，多数代码仍然只要能通过jiffies变量读取低32位就够了。

在64位体系结构上，jiffies_64和jiffies指的是同一个变量，代码既可以直接读取jiffies也可以调用get_jiffies_64()函数，它们的作用相同。

10.3.2 jiffies 的回绕

和任何C整型一样，当jiffies变量的值超过它的最大存放范围后就会发生溢出。对于32位无符号长整型，最大取值为$2^{32}-1$。所以在溢出前，定时器节拍计数最大为4294967295。如果节拍计数达到了最大值后还要继续增加的话，它的值会回绕（wrap around）到0。

请看下面一个回绕的例子：

```
unsigned long timeout = jiffies + HZ/2;     /* 0.5秒后超时 */

/* 执行一些任务 ... */

/* 然后查看是否花的时间过长 */
if(timeout>jiffies){
        /* 没有超时，很好 ... */
}else {
        /* 超时了，发生错误...*/
}
```

上面这一小片代码是希望设置一个准确的超时时间——本例中从现在开始计时，时间为半秒。然后再去处理一些工作，比如探测硬件然后等待它的响应。如果处理这些工作的时间超过了设定的超时时间，代码就要做相应的出错处理。

这里有很多种发生溢出的可能，我们只分析其中之一：考虑如果在设置完timeout变量后，jiffies重新回绕为0将会发生什么？此时，第一个判断会返回假，因为尽管实际上用去的时间可能

⊖ 因为32位体系结构不能原子地一次访问64位变量中的两个32位数值，还因为 jiffies_64没有 volatile标记，所以需要特殊函数来保证。该函数读取jiffies时会利用xtime_lock锁对jiffies变量进行锁定。

比timeout值要大，但是由于溢出后回绕为0，所以jiffies这时肯定会小于timeout的值。jiffies本该是个非常大的数值——大于timeout，但是因为超过了它的最大值，所以反而变成了一个很小的值——也许仅仅只有几个节拍计数。由于发生了回绕，所以if判断语句的结果刚好相反。

幸好，内核提供了四个宏来帮助比较节拍计数，它们能正确地处理了节拍计数回绕情况。 这些宏定义在文件<linux/jiffies.h>中：

```
#define time_after(unknown,known)    ((long)(known) -  (long)(unknown)<0)
#define time_before(unknown,known)   ((long)(unknown) - (long)(known)<0)
#define time_after_eq(unknown,known) ((long)(unknown) - (long)(known)>=0)
#define time_before_eq (unknown,known) ((long)(known) - (long)(unknown)>=0)
```

其中unkown参数通常是jiffies，known参数是需要对比的值。

宏time_after(unknown,known)，当时间unknown超过指定的known时，返回真，否则返回假；宏time_before(unknown,known)，当时间unknown没超过指定的known时，返回真，否则返回假。后面两个宏作用和前面两个宏一样，只有当两个参数相等时，它们才返回真。

所以前面的例子可以改造成时钟－回绕－安全（timer-wraparound-safe）的版本，形式如下：

```
unsigned long timeout = jiffies + HZ/2 ;       /* 0.5秒后超时*/

/*...*/
if(time_before(jiffies,timeout)){
        /* 没有超时，很好 ...*/
}else {
        /* 超时了，发生错误 ...*/
}
```

如果你对这些宏能避免因为回绕而产生的错误而感到好奇，那么，你可以试一试对这两个参数取不同的值。然后，设定一个参数回绕到0值，看看会发生什么。

10.3.3 用户空间和HZ

在2.6以前的内核中，如果改变内核中HZ的值会给用户空间中某些程序造成异常结果。这是因为内核是以节拍数/秒的形式给用户空间导出这个值的，在这个接口稳定了很长一段时间后，应用程序便逐渐依赖于这个特定的HZ值了。所以如果在内核中更改了HZ的定义值，就打破了用户空间的常量关系——用户空间并不知道新的HZ值。所以用户空间可能认为系统运行时间已经是20个小时了，但实际上系统仅仅启动了两个小时。

要想避免上面的错误，内核必须更改所有导出的jiffies值。因而内核定义了USER_HZ来代表用户空间看到的HZ值。在0x86体系结构上，由于HZ值原来一直是100，所以USER_HZ值就定义为100。内核可以使用宏jiffies_to_clock_t()将一个由HZ表示的节拍计数转换成一个由USER_HZ表示的节拍计数。该宏的用法取决于USER_HZ是否为HZ的整数倍或相反。当是整数倍时，宏的形式相当简单：

```
#define jiffies_to_clock_t(x)    ((x)/ (HZ/USER_HZ))
```

如果不是整数倍关系，那么该宏就得用到更为复杂的算法了。

最后还要说明，内核使用函数jiffies_64_to_clock_t()将64位的jiffies值的单位从HZ转换为USER_HZ。

在需要把以节拍数/秒为单位的值导出到用户空间时，需要使用上面这几个函数。比如：

```
unsigned long start ;
unsigned long total_time;

start = jiffies;
/* 执行一些任务...*/
total_time = jiffies - start;
printk("That took %lu ticks\n",jiffies_to_clock_t(total_time));
```

用户空间期望HZ=USER_HZ，但是如果它们不相等，则由宏完成转换，这样的结果自然是皆大欢喜。说实话，上面的例子看起来是挺简单的，如果以秒为单位而不是以节拍为单位，打印信息会执行得好一些。比如像下面这样：

```
printk("That took %lu seconds \n",total_time/HZ);
```

10.4 硬时钟和定时器

体系结构提供了两种设备进行计时——一种是我们前面讨论过的系统定时器；另一种是实时时钟。虽然在不同机器上这两种时钟的实现并不相同，但是它们有着相同的作用和设计思路。

10.4.1 实时时钟

实时时钟（RTC）是用来持久存放系统时间的设备，即便系统关闭后，它也可以靠主板上的微型电池提供的电力保持系统的计时。在PC体系结构中，RTC和CMOS集成在一起，而且RTC的运行和BIOS的保存设置都是通过同一个电池供电的。

当系统启动时，内核通过读取RTC来初始化墙上时间，该时间存放在xtime变量中。虽然内核通常不会在系统启动后再读取xtime变量，但是有些体系结构，比如x86，会周期性地将当前时间值存回RTC中。尽管如此，实时时钟最主要的作用仍是在启动时初始化xtime变量。

10.4.2 系统定时器

系统定时器是内核定时机制中最为重要的角色。尽管不同体系结构中的定时器实现不尽相同，但是系统定时器的根本思想并没有区别——提供一种周期性触发中断机制。有些体系结构是通过对电子晶振进行分频来实现系统定时器；还有些体系结构则提供了一个衰减测量器（decrementer）——衰减测量器设置一个初始值，该值以固定频率递减，当减到零时，触发一个中断。无论哪种情况，其效果都一样。

在0x86体系结构中，主要采用可编程中断时钟（PIT）。PIT在PC机器中普遍存在，而且从DOS时代，就开始以它作时钟中断源了。内核在启动时对PIT进行编程初始化，使其能够以Hz/秒的频率产生时钟中断。虽然PIT设备很简单，功能也有限，但它却足以满足我们的需要。x86体系结构中的其他的时钟资源还包括本地APIC时钟和时间戳计数（TSC）等。

10.5 时钟中断处理程序

现在我们已经理解了HZ、jiffies等概念以及系统定时器的功能。下面将分析时钟中断处理程序是如何实现的。时钟中断处理程序可以划分为两个部分：体系结构相关部分和体系结构无关部分。

与体系结构相关的例程作为系统定时器的中断处理程序而注册到内核中，以便在产生

时钟中断时，它能够相应地运行。虽然处理程序的具体工作依赖于特定的体系结构，但是绝大多数处理程序最低限度都要执行如下工作：

- 获得xtime_lock锁，以便对访问jiffies_64和墙上时间xtime进行保护。
- 需要时应答或重新设置系统时钟。
- 周期性地使用墙上时间更新实时时钟。
- 调用体系结构无关的时钟例程：do_timer()。

中断服务程序主要通过调用与体系结构无关的例程do_timer()执行下面的工作：

- 给jiffies_64变量增加1（这个操作即使是在32位体系结构上也是安全的，因为前面已经获得了xtime_lock锁）。
- 更新资源消耗的统计值，比如当前进程所消耗的系统时间和用户时间。
- 执行已经到期的动态定时器（下一节将讨论）。
- 执行第4章曾讨论的scheduler_tick()函数。
- 更新墙上时间，该时间存放在xtime变量中。
- 计算平均负载值。

因为上述工作分别都由单独的函数负责完成，所以实际上do_timer()例程的执行代码看起来非常简单。

```
void do_timer(struct pt_regs *regs)
{
        jiffies_64++;

        update_process_times(user_mode(regs));
        update_times();
}
```

user_mode()宏查询处理器寄存器regs的状态。如果时钟中断发生在用户空间，它返回1；如果发生在内核模式，则返回0。update_process_times()函数根据时钟中断产生的位置，对用户或对系统进行相应的时间更新。

```
void update_process_times(int user_tick)
{
        struct task_struct *p = current ;
        int cpu = smp_processor_id();
        int system = user_tick^1;

        update_one_process(p,user_tick,system,cpu);
        run_local_timers();
        scheduler_tick(user_tick,system);
}
```

update_one_process()函数的作用是更新进程时间。它的实现是相当细致的，但是要注意，因为使用了XOR操作，所以user_tick和system两个变量只要其中有一个为1，则另外一个就必为0。updates_one_process()函数可以通过判断分支，将user_tick和system加到进程相应的计数上：

```
/*
 * 更新恰当的时间计数器，给其加一个jiffy
```

```
 */
p->utime += user;
p->stime += system;
```

上述操作将适当的计数值增加1，而另一个值保持不变。也许你已经发现了，这样做意味着内核对进程进行时间计数时，是根据中断发生时处理器所处的模式进行分类统计的，它把上一个tick全部算给进程。但是事实上进程在上一个节拍期间可能多次进入和退出内核模式，而且在上一个节拍期间，该进程也不一定是惟一运行进程。很不幸，这种粒度的进程统计方式是传统的Unix所具有的，现在还没有更加精密的统计算法的支持，内核现在只能做到这个程度。这也是内核应该采用更高频率的另一个原因。

接下来的run_local_timers()函数标记了一个软中断（请参考第7章）去处理所有到期的定时器，在下一节中将具体讨论定时器。

最后 scheduler_tick()函数负责减少当前运行进程的时间片计数值并且在需要时设置need_resched标志。在SMP机器中，该函数还要负责平衡每个处理器上的运行队列，这点在第4章曾讨论过。

当update_process_times()函数返回后，do_timer()函数接着会调用update_times()函数更新墙上时钟。

```
void update_times(void)
{
        unsigned long ticks;

        ticks = jiffies - wall_jiffies;
        if(ticks){
                wall_jiffies + = ticks;
                update_wall_time(ticks);
        }
        last_time_offset = 0;
        calc_load(ticks);
}
```

ticks记录最近一次更新后新产生的节拍数。通常情况下ticks显然应该等于1。但是时钟中断也有可能丢失，因而节拍也会丢失。在中断长时间被禁止的情况下，就会出现这种现象——但这种现象并不正常，往往是个bug。wall_jiffies值随后被加上ticks——所以此刻wall_jiffies值就等于最新的墙上时间的更新值jiffies——接着调用update_wall_time()函数更新存储墙上时间的xtime，最后调用calc_load()更新载入平均值，到此，update_times()执行完毕。

do_timer()函数执行完毕后返回与体系结构相关的中断处理程序，继续执行后面的工作，释放xtime_lock锁，然后退出。

以上全部工作每1/Hz秒都要发生一次，也就是说在你的PC机上时钟中断处理程序每秒执行1000次。

10.6 实际时间

当前实际时间（墙上时间）定义在文件kernel/timer.c中：

```
struct timespec xtime;
```

timespec 数据结构定义在文件<linux/time.h>中，形式如下：

```
struct timespec{
        time_t tv_sec;          /* 秒 */
        long tv_nsec;           /* 纳秒 */
};
```

xtime.tv_sec以秒为单位，存放着自1970年7月1日（UTC）以来经过的时间，1970年1月1日被称为纪元，多数Unix系统的墙上时间都是基于该纪元而言的。xtime.tv_nsec记录自上一秒开始经过的纳秒数。

读写xtime变量需要使用xtime_lock锁，该锁不是普通自旋锁而是一个seqlock锁，在第9章中曾讨论过seqlock锁。

更新xtime首先要申请一个seqlock锁：

```
write_seqlock(&xtime_lock);

/* 更新xtime... */

write_sequnlock(&xtime_lock);
```

读取xtime时也要使用read_seqbegin()和read_seqretry()函数：

```
do{
        unsigned long lost;
        seq = read_seqbegin(&xtime_lock);

        usec = timer->get_offset();
        lost = jiffies- wall_jiffies;
        if(lost)
                usec += lost *(1000000/HZ);
        sec = xtime.tv_sec;
        usec + = (xtime.tv_nsec/1000);
}while(read_seqretry(&xtime_lock,seq));
```

该循环不断重复，直到读者确认读取数据时没有写操作介入。如果发现循环期间有时钟中断处理程序更新xtime，那么read_seqretry()函数就返回无效序列号，继续循环等待。

从用户空间取得墙上时间的主要接口是gettimeofday()，在内核中对应系统调用为sys_gettimeofday()：

```
asmlinkage long sys_gettimeofday(struct timeval *tv, struct timezone *tz)
{
        if (likely(tv)) {
                struct timeval ktv;
                do_gettimeofday(&ktv);
        if (copy_to_user(tv,&ktv,sizeof(ktv)))
                return -EFAULT;
        }
        if (unlikely(tz)){
                if(copy_to_user(tz,&sys_tz,sizeof(sys_tz)))
```

```
                return  -EFAULT;
        }
        return 0;
}
```

如果用户提供的tv参数非空，那么与体系结构相关的do_gettimeofday()函数将被调用。该函数执行的就是上面提到的循环读取xtime的操作。如果tz参数为空，该函数将把系统时区（存放在sys_tz中）返回用户。如果在给用户空间拷贝墙上时间或时区时发生错误，该函数返回-EFAULT；如果成功，则返回0。

虽然内核也实现了time()[⊖]系统调用，但是gettimeofday()几乎完全取代了它。另外C库函数也提供了一些墙上时间相关的库调用，比如ftime()和ctime()。

另外，系统调用settimeofday()来设置当前时间，它需要具有CAP_SYS_TIME权能。

除了更新xtime时间以外，内核不会像用户空间程序那样频繁使用xtime。但也有需要注意的特殊情况，那就是在文件系统的实现代码中存放访问时间戳（创建、存取、修改等等）时需要使用xtime。

10.7 定时器

定时器——有时也称为动态定时器或内核定时器——是管理内核时间的基础。内核经常需要推后执行某些代码，比如以前章节提到的下半部机制就是为了将工作放到以后执行。但不幸的是，“以后”这个概念很含糊，下半部的本意并非是放到以后的某个时间去执行任务，而仅仅是不在当前时间执行就可以了。我们所需要的是一种工具，能够使工作在指定时间点上执行——不长不短，正好在希望的时间点上。内核定时器正是解决这个问题的理想工具。

定时器的使用很简单。你只需要执行一些初始化工作，设置一个超时时间，指定超时发生后执行的函数，然后激活定时器就可以了。指定的函数将在定时器到期时自动执行。注意定时器并不周期运行，它在超时后就自行销毁，这也正是这种定时器被称为动态定时器[⊜]的一个原因；动态定时器不断地创建和销毁，而且它的运行次数也不受限制。定时器在内核中应用得非常普遍。

10.7.1 使用定时器

定时器由结构time_list表示，定义在文件<linux/timer.h>中。

```
struct timer_list {
        struct list_head entry;             /* 定时器链表的入口 */
        unsigned long expires;              /* 以jiffies为单位的定时值 */
        spinlock_t lock;                    /* 保护定时器的锁 */
        void (*function)(unsigned long);/* 定时器处理函数 */
        unsigned long data;                 /* 传给处理函数的长整形参数 */
        struct tvec_t_base_s *base;         /* 定时器内部值，用户不要使用 */
};
```

幸运的是，使用定时器并不需要深入了解该数据结构。事实上，过深的陷入该结构，反而会

⊖ 但在某些体系结构中，并没有实现sys_time()，而是用C库中的gettimeofday()函数模拟它。

⊜ 另一个原因是（2.3版本前）内核也存在静态定时器。这种定时器在编译时创建，而不是实时创建。由于静态定时器存在缺陷，已经被淘汰了。

使你的代码不能保证对可能发生的变化提供支持。内核提供了一组与定时器相关的接口用来简化管理定时器的操作。所有这些接口都声明在文件<linux/timer.h>中，大多数接口在文件kernel/timer.c中获得实现。

创建定时器时需要先定义它：

```
struct timer_list my_timer;
```

接着需要通过一个辅助函数初始化定时器数据结构的内部值，初始化必须在使用其他定时器管理函数对定时器进行操作前完成。

```
init_timer(&my_timer);
```

现在你可以填充结构中需要的值了：

```
my_timer.expires = jiffies + delay;        /* 定时器超时时的节拍数*/
my_timer.data = 0;                         /* 给定时器处理函数传入0值*/
my_timer.function = my_function;           /* 定时器超时时调用的函数*/
```

my_timer.expires表示超时时间，它是以节拍为单位的绝对计数值。如果当前jiffies计数等于或大于my_timer.expires，那么my_timer.function指向的处理函数就会开始执行，另外该函数还要使用长整型参数my_timer.data。所以正如我们从timer_list结构看到的形式，处理函数必须符合下面的函数原形：

```
void my_timer_function(unsigned long data);
```

data参数使你可以利用同一个处理函数注册多个定时器，只需通过该参数就能区别对待它们。如果你不需要这个参数，可以简单地传递0（或任何其他值）给处理函数。

最后，必须激活定时器：

```
add_timer(&my_timer);
```

大功告成，定时器可以工作了！但请注意定时值的重要性。当前节拍计数等于或大于指定的超时时，内核就开始执行定时器处理函数。虽然内核可以保证不会在超时时间到期前运行定时器处理函数，但是有可能延误定时器的执行。一般来说，定时器都在超时后马上就会执行，但是也有可能被推迟到下一次时钟节拍时才能运行，所以不能用定时器来实现任何硬实时任务。

有时可能需要更改已经激活的定时器超时时间，所以内核通过函数mod_timer()来实现该功能，该函数可以改变指定的定时器超时时间：

```
mod_timer(&my_timer,jiffies+new_delay); /*new expiration*/
```

mod_timer()函数也可操作那些已经初始化，但还没有被激活的定时器，如果定时器未被激活，mod_timer()会激活它。如果调用时定时器未被激活，该函数返回 0 ；否则返回 1 。但不论哪种情况，一旦从mod_timer()函数返回，定时器都将被激活而且设置了新的定时值。

如果需要在定时器超时前停止定时器，可以使用del_timer()函数：

```
del_timer(&my_timer);
```

被激活或未被激活的定时器都可以使用该函数，如果定时器还未被激活，该函数返回 0 ；否则返回1。注意，你不需要为已经超时的定时器调用该函数，因为它们会自动被删除。

当删除定时器时，必须小心一个潜在的竞争条件。当del_timer()返回后，可以保证的只是：定时器不会再被激活（也就是，将来不会执行），但是在多处理器机器上定时器中断可能已经在其他处理器上运行了，所以删除定时器时需要等待可能在其他处理器上运行的定时器处理程序都退出，

这时就要使用del_timer_sync()函数执行删除工作：

```
del_timer_sync(&my_timer);
```

和del_timer()函数不同，del_timer_sync()函数不能在中断上下文中使用。

10.7.2 定时器竞争条件

因为定时器与当前执行代码是异步的，因此就有可能存在潜在的竞争条件。所以，首先绝不能用如下所示的代码替代mod_timer()函数，来改变定时器的超时时间。这样的代码在多处理器机器上是不安全的：

```
del_timer(my_timer)
my_timer->expires = jiffies + new_delay;
add_timer(my_timer);
```

其次，一般情况下应该使用del_timer_sync()函数取代del_timer()函数，因为无法确定在删除定时器时，它是否正在其他处理器上运行。为了防止这种情况的发生，应该调用del_timer_sync()函数，而不是del_timer()函数。否则，对定时器执行删除操作后，代码会继续执行，但它有可能会去操作在其他处理器上运行的定时器正在使用的资源，因而造成并发访问。所以请优先使用删除定时器的同步方法。

最后，因为内核异步执行中断处理程序，所以应该重点保护定时器中断处理程序中的共享数据。定时器数据的保护问题曾在第8章和第9章讨论过。

10.7.3 实现定时器

内核在时钟中断发生后执行定时器，定时器作为软中断在下半部上下文中执行。具体来说，时钟中断处理程序会执行update_process_timers()函数，该函数随即调用run_local_timers()函数：

```
void run_local_timers(void)
{
        raise_softirq(TIMER_SOFTIRQ);
}
```

run_timer_softirq()函数处理软中断TIMER_SOFTIRQ，从而在当前处理器上运行所有的（如果有的话）超时定时器。

虽然所有定时器都以链表形式存放在一起，但是让内核经常为了寻找超时定时器而遍历整个链表是不明智的。同样，将链表以超时时间进行排序也是很不明智的做法，因为这样一来在链表中插入和删除定时器都会很费时。为了提高搜索效率，内核将定时器按它们的超时时间划分为五组。当定时器超时时间接近时，定时器将随组一起下移。采用分组定时器的方法可以在执行软中断的多数情况下，确保内核尽可能减少搜索超时定时器所带来的负担。因此定时器管理代码是非常高效的。

10.8 延迟执行

内核代码（尤其是驱动程序）除了使用定时器或下半部机制以外还需要其他方法来推迟执行任务。这种推迟通常发生在等待硬件完成某些工作时，而且等待的时间往往非常短，比如，重新设置网卡的以太模式需要花费2毫秒,所以在设定网卡速度后，驱动程序必须至少等待两毫秒才能

继续运行。

内核提供了许多延迟方法处理各种延迟要求。不同的方法有不同的处理特点，有些是在延迟任务时挂起处理器，防止处理器执行任何实际工作；另一些不会挂起处理器，所以也不能确保被延迟的代码能够在指定的延迟时间[⊖]运行。

10.8.1 忙等待

最简单的延迟方法（虽然通常也是最不理想的办法）是忙等待（或者说忙循环）。但要注意该方法仅仅在想要延迟的时间是节拍的整数倍，或者精确率要求不高时才可以使用。

忙循环实现起来很简单：在循环中不断旋转直到希望的时钟节拍数耗尽，比如：

```
unsigned long delay = jiffies+10;          /* 10个节拍*/

while(time_before(jiffies,delay))
        ;
```

循环不断执行，直到jiffies大于delay为止，总共的循环时间为10个节拍。在HZ值等于1000的x86体系结构上，耗时为10毫秒。同样地：

```
unsigned long delay = jiffies + 2*HZ;     /* 2秒*/

while(time_before(jiffies,delay))
        ;
```

程序要循环等待2×HZ个时钟节拍，也就是说无论时钟节拍率如何，都将等待两秒钟。

对于系统的其他部分，忙循环方法算不上一个好办法。因为当代码等待时，处理器只能在原地旋转等待——它不会去处理其他任何任务！事实上，你几乎不会用到这种低效率的办法，这里介绍它仅仅是因为它是最简单最直接的延迟方法。当然你也可能在那些蹩脚的代码中发现它的身影。

更好的方法应该是在代码等待时，允许内核重新调度执行其他任务：

```
unsigned long delay = jiffies +5*HZ;

while(time_before(jiffies,delay))
        cond_resched();
```

cond_resched()函数将调度一个新程序投入运行，但它只有在设置完need_resched标志后，才能生效。换句话说，该方法有效的条件是系统中存在更重要的任务需要运行。注意因为该方法需要调用调度程序，所以它不能在中断上下文中使用——只能在进程上下文中使用。事实上，所有延迟方法在进程上下文中使用，因为中断处理程序都应该尽可能快地执行（忙循环与这种目标绝对是背道而驰）。另外，延迟执行不管在哪种情况下都不应该在持有锁时或禁止中断时发生。

C语言的推崇者可能会问什么能保证前面的循环已经执行了。C编译器通常只将变量装载一次。一般情况下不能保证循环中的jiffies变量在每次循环中被读取时都重新被载入。但是我们要求jiffies在每次循环时都必须重新装载，因为在后台jiffies值会随时钟中断的发生而不断增加。为了解决这

⊖ 事实上，没有方法能保证实际的延迟刚好等于指定的延迟时间，虽然可以非常接近，但是最精确的情况也只能达到接近，多数情况都要长于指定时间。

个问题，<linux/jiffies.h>中jiffies变量被标记为关键字volatile，关键字volatile指示编译器在每次访问变量时都重新从主内存中获得，而不是通过寄存器中的变量别名来访问，从而确保前面的循环能按预期的方式执行。

10.8.2 短延迟

有时内核代码（通常也是驱动程序）不但需要很短暂的延迟（比时钟节拍还短）而且还要求延迟的时间很精确。这种情况多发生在和硬件同步时，也就是说需要短暂等待某个动作的完成——等待时间往往小于1毫秒，所以不可能使用像前面例子中那种基于jiffies的延迟方法。对于频率为100Hz的时钟中断，它的节拍间隔甚至会超过10毫秒！即使频率为1000Hz的时钟中断，节拍间隔也只能到1毫秒，所以我们必须寻找其他方法满足更短、更精确的延迟要求。

幸运的是，内核提供了两个可以处理微秒和毫秒级别的延迟的函数，它们都定义在文件<linux/delay.h>中，可以看到它们并不使用jiffies：

```
void udelay(unsigned long usecs)
void mdelay(unsigned long msecs)
```

前一个函数利用忙循环将任务延迟到指定的微秒数后运行，后者延迟指定的毫秒数。众所周知，1秒等于1000毫秒，等于1000000微秒。用起来很简单：

```
udelay(150);                    /*延迟150微秒*/
```

udelay()函数依靠执行数次循环达到延迟效果，而mdelay()函数又是通过udelay()函数实现的。因为内核知道处理器在一秒内能执行多少次循环（请看下面框中的BogoMips内容），所以udelay()函数仅仅需要根据指定的延迟时间在1秒中占的比例，就能决定需要进行多少次循环就能达到要求的推迟时间。

undelay()函数仅能在要求的延迟时间很短的情况下执行，而在高速机器中时间较长的延迟会造成溢出。经验证明，不要使用udelay()函数处理超过1毫秒的延迟。延迟超过1毫秒的情况下，使用mdelay()函数更为安全。和其他忙等待延迟方法一样，这些函数（特别是mdelay()函数，因为它延迟时间较长）如果不是非常必要的情况下，尽量少用。千万注意不要在持有锁时或禁止中断时使用忙等待，因为这时忙等待会使系统响应速度和性能大打折扣。但是如果你真的需要精确延迟，那么用这些函数冒一下险，可能是你最好的选择。通常这些忙等待推迟函数都用在短暂延迟的情况下，往往是微秒级的推迟。

我的BogoMIPS比你的大

BogoMIPS值总是让人觉得糊涂，也让人觉得有意思。其实，计算BogoMIPS并不是为了表现你的机器性能，它主要被udelay()函数和mdelay()函数使用。它的名字取自bogus（也就是伪的）和MIPS（每秒处理百万条指令）。大家都熟悉下面这样的系统启动信息（摘自一个装配主频为1GHz的奔腾3处理器的机器启动信息）：

```
Detected 1004.932  MHz processor.
Calibrating delay loop ... 1990.65 BogoMIPS
```

BogoMIPS值记录处理器在给定时间内忙循环执行的次数。其实，BogoMIPS记录处理器在空闲时速度有多快！该值存放在变量loops_per_jiffy中，可以从文件/proc/cpuinfo中读到它。延

迟循环函数使用loops_per_jiffy值来计算(相当准确)为提供精确延迟而需要进行多少次循环。

内核在启动时利用calibrate_delay()计算loops_per_jiffy值，该函数在文件init/main.c中。

udelay()函数应当只在小延迟中调用，因为在快速机器上的大延迟可能会导致溢出。通常，超过1ms的范围不要使用udelay()进行延迟。对于较长的延迟，用mdelay()效果良好。像其他忙等而延迟执行的方案，除非绝对必要，这两个函数（尤其是mdelay()，因为用于长的延迟）都不应当使用。记住，持锁忙等或禁止中断是一种粗鲁的做法，因为系统响应时间和性能都会大受影响。不过，如果你需要精确的延迟，这些调用是最好的办法。这些忙等函数主要用在延迟小的地方，通常在微秒范围内。

10.8.3 schedule_timeout()

更理想的延迟执行方法是使用schedule_timeout()函数，该方法会让需要延迟执行的任务睡眠到指定的延迟时间耗尽后再重新运行。但该方法也不能保证睡眠时间正好等于指定的延迟时间——只能尽量使睡眠时间接近指定的延迟时间。当指定的时间到期后，内核唤醒被延迟的任务并将其重新放回运行队列，用法如下：

```
/* 将任务设置为可中断睡眠状态 */
set_current_state(TASK_INTERRUPTIBLE);

/* 小睡一会儿，"s"秒后唤醒*/
schedule_timeout(s*HZ);
```

惟一的参数是延迟的相对时间，单位为jiffies，上例中将相应的任务推入可中断睡眠队列，睡眠s秒。因为任务处于可中断状态，所以如果任务收到信号将被唤醒。如果睡眠任务不想接收信号，可以将任务状态设置为TASK_UNINTERRUPTIBLE，然后睡眠。注意在调用schedule_timeout()函数前必须首先将任务设置成上面两种状态之一，否则任务不会睡眠。

注意，由于schedule_timeout()函数需要调用调度程序，所以调用它的代码必须保证能够睡眠(请参考第8章和第9章)。简而言之，调用代码必须处于进程上下文中，并且不能持有锁。

schedule_timeout()函数的用法相当简单、直接。其实，它是内核定时器的一个简单应用。请看下面的代码：

```
signed long schedule_timeout(signed long timeout)
{
        timer_t timer;
        unsigned long expire;

        switch (timeout)
        {
        case MAX_SCHEDULE_TIMEOUT:
                schedule();
                goto out;
        default:
                if (timeout < 0)
                {
                        printk(KERN_ERR"schedule_timeout: wrong timeout "
```

```
                    "value %lx from %p\n", timeout,
                    __builtin_return_address(0));
                current->state = TASK_RUNNING;
                goto out;
        }
}

expire = timeout + jiffies;

init_timer(&timer);
timer.expires = expire;
timer.data = (unsigned long) current;
timer.function = process_timeout;

add_timer(&timer);
schedule();
del_timer_sync(&timer);

timeout = expire - jiffies;

out:
        return timeout < 0 ? 0 : timeout;
}
```

该函数用原始的名字timer创建一个定时器timer，然后设置它的超时时间timeout，设置超时执行函数process_timeout()，然后激活定时器而且调用schedule()。因为任务被标识为TASK_INTERRUPTIBLE或TASK_UNINTERRUPTIBLE，所以调度程序不会再选择该任务投入运行，而会选择其他新任务运行。

当定时器超时，process_timeout()函数被调用:

```
void process_timeout(unsigned long data)
{
        wake_up_process((task_t *)data);
}
```

该函数将任务置为TASK_RUNNING状态，然后将其放入运行队列。

当任务重新被调度时，将返回代码进入睡眠前的位置继续执行（正好在调用schedule()后）。如果任务提前被唤醒（比如收到信号），那么定时器被销毁，process_timeout()函数返回剩余的时间。

在switch()括号中的代码是为处理特殊情况而写的，正常情况不会用到它们。MAX_SCHEDULE_TIMEOUT是用来检查任务是否无限期地睡眠，如果那样的话，函数不会为它设置定时器（因为睡眠时间没有期限），这时调度程序会立刻被调用。如果你需要无限期的让任务睡眠，最好使用其他方法唤醒任务。

10.8.4 设置超时时间，在等待队列上睡眠

第4章我们已经看到进程上下文中的代码为了等待特定事件发生，可以将自己放入等待队列，然后调用调度程序去执行新任务。一旦事件发生后，内核调用wake_up()函数唤醒在睡眠队列上的

任务，使其重新投入运行。

有时，等待队列上的某个任务可能既在等待一个特定事件到来，又在等待一个特定时间到期——就看谁来得更快。这种情况下，代码可以简单地使用schedule_timeout()函数代替schedule()函数，这样一来，当希望指定时间到期，任务都会被唤醒。当然，代码需要检查被唤醒的原因——有可能是被事件唤醒，也有可能是因为延迟的时间到期，还可能是因为接收到了信号——然后执行相应的操作。

10.9 小结

在这一章，我们考察了时间的概念，并知道了如何管理墙上时钟与计算机的正常运行时间。我们对比了相对时间和绝对时间，以及绝对事件与周期事件。我们还涵盖了诸如时钟中断、时钟节拍、HZ以及jiffies等概念。

我们考察了定时器的实现，了解了如何把这些用到自己的内核代码中。本章最后，浏览了开发者用于延迟的其他方法。

你写的大多数内核代码都需要对时间及其走过的时间有一些理解。最大的可能是，如果你编写驱动程序，那么就需要处理内核定时器。让时间悄悄溜走，还不如阅读本章。

第⑪章

内存管理

在内核里分配内存可不像在其他地方分配内存那么容易。造成这种局面的因素很多。从根本上讲，是因为内核本身不能像用户空间那样奢侈地使用内存。内核与用户空间不同，它不具备这种能力，它不支持简单便捷的内存分配方式。比如，内核一般不能睡眠。此外，处理内存错误对内核来说也决非易事。正是由于这些限制，再加上内存分配机制不能太复杂，所以在内核中获取内存要比在用户空间复杂得多。不过，我们会看到，也不是说内核的内存分配就困难得不得了。只是和用户空间中的内存分配不太一样而已。

本章讨论的是在内核之中获取内存的方法。在深入研究实际的分配接口之前，我们需要理解内核是如何管理内存的。

11.1 页

内核把物理页作为内存管理的基本单位。尽管处理器的最小可寻址单位通常为字（甚至字节），但是，内存管理单元（MMU，管理内存并把虚地址转换为物理地址的硬件）通常以页为单位进行处理。正因为如此，MMU以页（page）大小为单位来管理系统中的页表（这也是页表名的来由）。从虚拟内存的角度来看，页就是最小单位。

我们在第19章中将会看到，体系结构不同，支持的页大小也不尽相同。还有些体系结构甚至支持几种不同的页大小。大多数32位体系结构支持4KB的页，而64位体系结构一般会支持8KB的页。这就意味着，在支持4KB页大小并有1GB物理内存的机器上，物理内存会被划分为262144个页。

内核用struct page结构表示系统中的每个物理页，该结构位于<linux/mm.h>中：

```
struct page {
        page_flags_t            flags;
        atomic_t                _count;
        atomic_t                _mapcount;
        unsigned long           private;
        struct address_space    *mapping;
        pgoff_t                 index;
        struct list_head        lru;
        void                    *virtual;
};
```

让我们看一下其中比较重要的域。flag域用来存放页的状态。这些状态包括页是不是脏的，是不是被锁定在内存中等等。flag的每一位单独表示一种状态，所以它至少可以同时表示出32种不同的状态。这些标志定义在<linux/page-flags.h>中。

_count域存放页的引用计数——也就是这一页被引用了多少次。当计数值变为0时，就说明当

前内核并没有引用这一页，于是，在新的分配中就可以使用它。内核代码不应当直接检查该域，而是调用page_count()函数进行检查，该函数惟一的参数就是 page结构。当页空闲时，尽管该结构内部的_count值是负的，但是对page_count()函数而言，返回0表示页空闲，返回一个正整数表示页在使用。一个页可以由页缓存使用（这时，mapping域指向和这个页关联的address_space对象），或者作为私有数据(由private指向)，或者作为进程页表中的映射。

virtual域是页的虚拟地址。通常情况下，它就是页在虚拟内存中的地址。有些内存（即所谓的高端内存）并不永久地映射到内核地址空间上。在这种情况下，这个域的值为NULL，需要的时候，必须动态地映射这些页。稍后我们将讨论高端内存。

必须要理解的一点是page结构与物理页相关，而并非与虚拟页相关。因此，该结构对页的描述只是短暂的。即使页中所包含的数据继续存在，但是由于交换等原因，它们可能并不再和同一个page结构相关联。内核仅仅用这个数据结构来描述当前时刻在相关的物理页中存放的东西。这种数据结构的目的在于描述物理内存本身，而不是描述包含在其中的数据。

内核用这一结构来管理系统中所有的页，因为内核需要知道一个页是否空闲（也就是页有没有被分配）。如果页已经被分配，内核还需要知道谁拥有这个页。拥有者可能是用户空间进程、动态分配的内核数据、静态内核代码，或页高速缓存等等。

系统中的每个物理页都要分配一个这样的结构体，开发者常常对此感到惊讶。他们会想“这得消耗多少内存呀！”。让我们来算算对所有这些页都这么做，到底要消耗掉多少内存。就算struct page占40字节的内存吧，假定系统的物理页为4KB大小，系统有128MB物理内存。那么，系统中所有page结构消耗的内存只不过是1MB多一点——其实，要管理系统中这么多物理页面，这个代价并不算太高。

11.2 区

由于硬件的限制，内核并不能对所有的页一视同仁。有些页位于内存中特定的物理地址上，所以不能将其用于一些特定的任务。由于存在这种限制，所以内核把页划分为不同的区（zone）。内核使用区对具有相似特性的页进行分组。Linux必须处理如下两种由于硬件存在缺陷而引起的内存寻址问题：

- 一些硬件只能用某些特定的内存地址来执行DMA（直接内存访问）。
- 一些体系结构其内存的物理寻址范围比虚拟寻址范围大得多。这样，就有一些内存不能永久地映射到内核空间上。

因为存在这些制约条件，Linux使用了三种区：

- ZONE_DMA——这个区包含的页能用来执行DMA操作。
- ZONE_NORMAL——这个区包含的都是能正常映射的页。
- ZONE_HIGHMEM——这个区包含“高端内存”，其中的页并能不永久地映射到内核地址空间。

这些区定义于 <linux/mmzone.h>中。

区的实际使用和分布是与体系结构相关的。例如，某些体系结构在内存的任何地址上执行DMA都没有问题。在这些体系结构中，ZONE_DMA为空，ZONE_NORMAL就可以直接用于分配。与此相反，在x86体系结构上，ISA设备[⊖]就不能在整个32位的地址空间中执行DMA，因为ISA设备

⊖ 某些过时的PCI设备只能在24位的地址空间执行DMA。但是它们已经被淘汰。

只能访问物理内存的前16MB。因此，ZONE_DMA在x86上包含的页都在0~16MB的内存范围里。

ZONE_HIGHMEM的工作方式也差不多。能否直接映射取决于体系结构。在x86上，ZONE_HIGHMEM为高于896MB的所有物理内存。在其他体系结构上，由于所有内存都被直接映射，所以ZONE_HIGHMEM为空。ZONE_HIGHMEM所在的内存就是所谓的高端内存（high memory）[⊖]。系统的其余内存就是所谓的低端内存（low memory）。

前两个区各取所需之后，剩余的就由ZONE_NORMAL区独享了。在x86上，ZONE_NORMAL是从16MB到896MB的所有物理内存。在其他（更幸运）的体系结构上，ZONE_NORMAL是所有的可用物理内存。表11-1是每个区及其在x86上所占页的列表。

表11-1 x86上的区

区	描 述	物理内存
ZONE_DMA	DMA使用的页	<16MB
ZONE_NORMAL	正常可寻址的页	16~896MB
ZONE_HIGHMEM	动态映射的页	>896MB

Linux把系统的页划分为区，形成不同的内存池，这样就可以根据用途进行分配了。例如，ZONE_DMA内存池让内核有能力为DMA分配所需的内存。如果需要这样的内存，那么，内核就可以从ZONE_DMA中按照请求的数目取出页。注意，区的划分没有任何物理意义；这只不过是内核为了管理页而采取的一种逻辑上的分组。

尽管某些分配可能需要从特定的区中获取页，但这并不是说，某种用途的内存一定要从对应的区获取。尽管用于DMA的内存必须从ZONE_DMA中进行分配，但是一般用途的内存却既能从ZONE_DMA分配，也能从ZONE_NORMAL分配。当然，内核更希望一般用途的内存从常规区分配，这样能节省ZONE_DMA中的页，保证满足DMA的使用需求。但是，如果可供分配的资源不够用了（如果内存已经变得很少了），那么，内核就会去占用其他可用区的内存。

每个区都用struct zone表示，定义在<linux/mmzone.h>中：

```
struct zone{
        spinlock_t              lock;
        unsigned long           free_pages;
        unsigned long           pages_min;
        unsigned long           pages_low;
        unsigned long           pages_high;
        unsigned long           protection[MAX_NR_ZONES];
        spinlock_t              lru_lock;
        struct list_head        active_list;
        struct list_head        inactive_list;
        unsigned long           nr_scan_active;
        unsigned long           nr_scan_inactive;
        unsigned long           nr_active;
        unsigned long           nr_inactive;
        int                     all_unreclaimable;
```

⊖ 这与DOS中的高端内存毫无关系。

```
    unsigned long           pages_scanned;
    int                     temp_priority;
    int                     prev_priority;
    struct free_area        free_area[MAX_ORDER];
    wait_queue_head_t       *wait_table;
    unsigned long           wait_table_size;
    unsigned long           wait_table_bits;
    struct per_cpu_pageset  pageset[NR_CPUS];
    struct pglist_data      *zone_pgdat;
    struct page             *zone_mem_map;
    unsigned long           zone_start_pfn;
    char                    *name;
    unsigned long           spanned_pages;
    unsigned long           present_pages;
};
```

这个结构体很大，但是，系统中只有三个区，因此也只有三个这样的结构。让我们看一下其中一些重要的域。

lock域是一个自旋锁，它防止该结构被并发访问。注意，这个域只保护结构，而不保护驻留在这个区中的所有页。没有特定的锁来保护单个页，但是，部分内核可以锁住在页中驻留的数据。

free_pages域是这个区中空闲页的个数。内核尽可能保证（通过交换达到目的）有pages_min个空闲页可用。

name域是一个以NULL结束的字符串，表示这个区的名字。内核启动期间初始化这个值，其代码位于mm/page_alloc.c中。三个区的名字分别为“DMA”、“Normal”和“HighMem”。

11.3 获得页

我们已经对内核如何管理内存（页、区等等）有所了解了，现在让我们看一下内核实现的接口，我们正是通过这些接口在内核内分配和释放内存的。

内核提供了一种请求内存的底层机制，并提供了对它进行访问的几个接口。所有这些接口都以页为单位分配内存，定义于<linux/gfp.h>中。最核心的函数是：

```
struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)
```

该函数分配2^{order}（即1<<order）个连续的物理页，并返回一个指针，该指针指向第一个页的page结构体；如果出错，就返回NULL。在下一节我们再研究gfp_mask参数。你可以用下面这个函数把给定的页转换成它的逻辑地址：

```
void *page_address(struct page *page)
```

该函数返回一个指针，指向给定物理页当前所在的逻辑地址。如果你无须用到struct page，你可以调用：

```
unsigned long__get_free_pages(unsigned int gfp_mask, unsigned int order)
```

这个函数与alloc_pages()作用相同，不过它直接返回所请求的第一个页的逻辑地址。因为页是连续的，因此其他页也会紧随其后。

如果你只需一页，就可以用下面两个封装好的函数，能让你少敲几下键盘：

```
struct page * alloc_page(unsigned int gfp_mask)
unsigned long __get_free_page(unsigned int gfp_mask)
```

这两个函数与其兄弟函数工作方式相同，只不过传递给order的值为0（$2^0=1$页）。

11.3.1 获得填充为0的页

如果你需要让返回的页的内容全为0，请用下面这个函数：

```
unsigned long  get_zeroed_page(unsigned int gfp_mask)
```

这个函数与_get_free_page()工作方式相同，只不过把分配好的页都填充成了0。如果分配的页是给用户空间的，这个函数就非常有用了。虽说分配好的页中应该包含的都是随机产生的垃圾信息，但其实这些信息可能并不是完全随机的——它很可能“随机地”包含某些敏感数据。用户空间的页在返回之前，所有数据必须填充为0，或做其他清理工作，在保障系统安全这一点上，我们绝不妥协。表11-2是所有底层的页分配方法的列表。

表11-2 低级页分配方法

标 志	描 述
alloc_page(gfp_mask)	只分配一页，返回指向页结构的指针
alloc_pages(gfp_mask,order)	分配2^{order}个页，返回指向第一页页结构的指针
_get_free_page(gfp_mask)	只分配一页，返回指向其逻辑地址的指针
__get_free_pages(gfp_mask,order)	分配2^{order}页，返回指向第一页逻辑地址的指针
get_zeroed_page(gfp_mask)	只分配一页，让其内容填充0，返回指向其逻辑地址的指针

11.3.2 释放页

当你不再需要页时可以用下面的一族函数来释放它们：

```
void __free_pages(struct page *page, unsigned int order)
void free_pages(unsigned long addr, unsigned int order)
void free_page(unsigned long addr)
```

释放页时要谨慎，只能释放属于你的页。传递了错误的struct page或地址，用了错误的order值，这些都可能导致系统崩溃。请记住，内核是完全信赖自己的。这点与用户空间不同，如果你有非法操作，内核会开开心心地把自己挂起来，停止运行。

让我们看一个例子。其中，我们想得到8个页：

```
unsigned long page;

page = __get_free_pages(GFP_KERNEL,3);
if (!page){
        /* 没有足够的内存：你必须处理这种错误！ */
        return -ENOMEM;
}

/*'page'现在指向8个连续页中第1个页的地址...*/

free_pages(page,3);
```

```
/*
 * 页现在已经被释放了，我们不应该再访问
 *   存放在'page'中的地址了
 */
```

GFP_KERNEL参数是gfp_mask标志的一个例子。稍后我们将对此进行简要讨论。

调用_get_free_pages()之后要注意进行错误检查。内核分配可能失败，因此你的代码必须进行检查并做相应的处理。这意味在此之前，你所做的所有工作可能前功尽弃，甚至还需要回滚到原来的状态。正因为如此，在程序开始的地方就先进行内存分配是很有意义的，这能让错误处理来得容易一点。如果你不这么做，那么在你想要分配内存的时候如果失败了，局面可能就难以控制了。

当你需要以页为单位的一簇连续物理页时，尤其是在你只需要一、两页时，这些低级页函数很有用。对于常用的以字节为单位的分配来说，内核提供的函数是kmalloc()。

11.4 kmalloc()

kmalloc()函数与用户空间的malloc()一族函数非常类似，只不过它多了一个flags参数。kmalloc()函数是一个简单的接口，用它可以获得以字节为单位的一块内核内存。如果你需要整个页，那么，前面讨论的页分配接口可能是更好的选择。但是，对于大多数内核分配来说，kmalloc()接口用得更多。

kmalloc()在<linux/slab.h>中声明：

```
void *kmalloc(size_t size, int flags)
```

这个函数返回一个指向内存块的指针，其内存块至少要有size大小[⊖]。所分配的内存区在物理上是连续的。在出错时，它返回NULL。除非没有足够的内存可用，否则内核总能分配成功。在对kmalloc()调用之后，你必须检查返回的是不是NULL，如果是，要适当地处理错误。

让我们看一个例子。我们随便假定存在一个dog结构体，现在需要为它动态地分配足够的空间：

```
struct dog  *ptr;

ptr = kmalloc(sizeof(struct dog), GFP_KERNEL);
if (!ptr)
        /* 处理错误... */
```

如果kmalloc()调用成功，那么，ptr现在指向一个内存块，内存块的大小至少为所请求的大小。GFP_KERNEL标志表示在试图获取内存并返回给kmalloc()的调用者的过程中，内存分配器将要采取的行为。

11.4.1 gfp_mask标志

我们已经看过了几个例子，发现不管是在低级页分配函数中，还是在kmalloc()中，都用到了分配器标志。现在，我们就深入讨论一下这些标志。

⊖ 分配的内存可能比你请求的还多，但是你无法知道到底多了多少！因为内核分配器本质上是基于页的，因此在可用内存内，某些分配可能向上取整。内核分配的内存绝不会少于所需要的内存。如果内核不能找到所需要的最少量，那么，分配就会失败，函数返回NULL。

这些标志可分为三类：行为修饰符、区修饰符及类型。行为修饰符表示内核应当如何分配所需的内存。在某些特定情况下，只能使用某些特定的方法分配内存。例如，中断处理程序就要求内核在分配内存的过程中不能睡眠（因为中断处理程序不能被重新调度）。区修饰符表示从哪儿分配内存。前面我们已经看到，内核把物理内存分为多个区，每个区用于不同的目的。区修饰符指明到底从这些区中的哪一区中进行分配。类型标志组合了行为修饰符和区修饰符，将各种可能用到的组合归纳为不同类型，简化了修饰符的使用；这样，你只须指定一个类型标志就可以了。GFP_KERNEL就是一种类型标志，内核中进程上下文相关的代码可以使用它。我们来看一下这些标志。

1. 行为修饰符

所有这些标志，包括行为描述符都是在<linux/gfp.h>中声明的。不过，在<linux/slab.h>中包含有这个头文件，因此，你一般不必直接包含引用它。实际上，一般只使用类型修饰符就够了，我们随后会看到这点。因此，最好对每个标志都有所了解。表11-3是行为修饰符的列表。

表11-3 行为修饰符

标 志	描 述
__GFP_WAIT	分配器可以睡眠
__GFP_HIGH	分配器可以访问紧急事件缓冲池
__GFP_IO	分配器可以启动磁盘I/O
__GFP_FS	分配器可以启动文件系统I/O
__GFP_COLD	分配器应该使用高速缓存中快要淘汰出去的页
__GFP_NOWARN	分配器将不打印失败警告
__GFP_REPEAT	分配器在分配失败时重复进行分配，但是这次分配还存在失败的可能
__GFP_NOFAIL	分配器将无限地重复进行分配，分配不能失败
__GFP_NORETRY	分配器在分配失败时绝不会重新分配
__GFP_NO_GROW	由slab层内部使用
__GFP_COMP	添加混合页元数据，在hugetlb的代码内部使用

可以同时指定这些分配标志。例如：

```
ptr=kmalloc(size, __GFP_WAIT | __GFP_IO | __GFP_FS);
```

说明页分配器（最终调用alloc_pages()）在分配时可以阻塞、执行I/O，在必要时还可以执行文件系统操作。这就让内核有很大的自由度，以便它尽可能找到空闲的内存来满足分配请求。

大多数分配都会指定这些修饰符，但一般不是这样直接指定，而是采用我们随后讨论的类型标志。别担心，你不会在分配内存时为怎样使用这些标志而犯愁的！

2. 区修饰符

区修饰符表示内存区应当从何处分配。通常，分配可以从任何区开始。不过，内核优先从ZONE_NORMAL开始，这样可以确保其他区在需要时有足够的空闲页可供使用。

实际上只有两个区修饰符，因为除了ZONE_NORMAL之外只有两个区（默认都是从ZONE_NORMAL区进行分配）。表11-4是区修饰符的列表。

表11-4 区修饰符

标 志	描 述
__GFP_DMA	从ZONE_DMA分配
__GFP_HIGHMEM	从ZONE_HIGHMEM或ZONE_NORMAL分配

指定以上标志中的一个就可以改变内核试图进行分配的区。__GFP_DMA标志强制内核从ZONE_DMA分配。这个标志在说，有了这种奇怪的标识，我绝对可以拥有进行DMA的内存。相反，如果指定__GFP_HIGHMEM标志，则从ZONE_HIGHMEM（优先）或ZONE_NORMAL分配。这个标志在说，我可以使用高端内存，因此，我可以是一个玩偶，给你退还一些内存，但是，常规内存还照常工作。如果没有指定任何标志，则内核从ZONE_DMA或ZONE_NORMAL进行分配，当然优先从ZONE_NORMAL进行分配。没有区标志说什么了，只要它行为正常，我漠不关心了。

你不能给_get_free_pages()或kmalloc()指定_GFP_HIGHMEM，因为这两个函数返回的都是逻辑地址，而不是page结构，这两个函数分配的内存当前有可能还没有映射到内核的虚拟地址空间，因此，也可能根本就没有逻辑地址。只有alloc_pages()才能分配高端内存。实际上，你的分配在大多数情况下都不必指定修饰符，ZONE_NORMAL就足矣。

3. 类型标志

类型标志指定所需的行为和区描述符以完成特殊类型的处理。正因为这一点，内核代码趋向于使用正确的类型标志，而不是一味地指定它可能需要用到的多个描述符。这么做既简单又不容易出错误。表11-5是类型标志的列表，而表11-6显示了每个类型标志与哪些修饰符相关联。

表11-5 类型标志

标 志	描 述
GFP_ATOMIC	分配是高优先级的，且不会睡眠。这个标志用在中断处理程序、下半部、持有自旋锁，以及其他不能睡眠的地方
GFP_NOIO	这种分配可以阻塞，但不会启动磁盘I/O。这个标志在不能引发更多磁盘I/O时能阻塞I/O代码，这可能导致令人不愉快的递归
GFP_NOFS	这种分配在必要时可能阻塞，也可能启动磁盘I/O，但是不会启动文件系统操作。这个标志在你不能再启动另一个文件系统的操作时用在文件系统部分的代码中
GFP_KERNEL	这是一种常规分配方式，可能会阻塞。这个标志在睡眠安全时用在进程上下文代码中。为了获得调用者所需的内存，内核会尽力而为。这个标志应当是首选标志
GFP_USER	这是一种常规分配方式，可能会阻塞。这个标志用于为用户空间进程分配内存时
GFP_HIGHUSER	这是从ZONE_HIGHMEM进行分配，可能会阻塞。这个标志用于为用户空间进程分配内存
GFP_DMA	这是从ZONE_DAM进行分配。需要获取能供DMA使用的内存的设备驱动程序使用这个标志，通常与以上的某个标志组合在一起使用

表11-6 在每种类型标志后隐含的修饰符列表

标 志	修饰符标志
GFP_ATOMIC	__GFP_HIGH
GFP_NOIO	__GFP_WAIT
GFP_NOFS	(__GFP_WAIT|__GFP_IO)
GFP_KERNEL	(__GFP_WAIT|__GFP_IO|__GFP_FS)
GFP_USER	(__GFP_WAIT|__GFP_IO|__GFP_FS)
GFP_HIGHUSER	(__GFP_WAIT|__GFP_IO|__GFP_FS|__GFP_HIGHMEM)
GFP_DMA	__GFP_DMA

让我们看一下最常用的标志，以及你什么时候、为什么需要使用它们。内核中最常用的标志是GFP_KERNEL。这种分配可能会引起睡眠，它使用的是普通优先级。因为调用可能阻塞，因此

这个标志只用在可以重新安全调度的进程上下文中（也就是没有锁被持有等情况）。因为这个标志对内核如何获取请求的内存没有任何约束，所以内存分配成功的可能性很高。

另一个全然相反的标志是GFP_ATOMIC。因为这个标志表示不能睡眠的内存分配，因此想要满足调用者获取内存的请求将会受到很严格的限制。即使没有足够的连续内存块可供使用，内核也很可能无法释放出可用内存来，因为内核不能让调用者睡眠。相反，GFP_KERNEL分配可以让调用者睡眠，交换、刷新一些页到硬盘等等。因为GFP_ATOMIC不能执行以上任何操作，因此与GFP_KERNEL相比较，它分配成功的机会较小（尤其在内存短缺时）。即便如此，在当前代码（例如中断处理程序、软中断和tasklet）不能睡眠时，也只能选择GFP_ATOMIC。

在以上两种标志中间的是GFP_NOIO和GFP_NOFS。以这两个标志进行的分配可能会引起阻塞，但它们会避免执行某些其他操作。GFP_NOIO分配决不会启动任何磁盘I/O来帮助满足请求。而GFP_NOFS可能会启动磁盘I/O，但是它不会启动文件系统I/O。你为什么需要这些标志？它们分别用在某些低级块I/O或文件系统的代码中。设想，如果文件系统代码中需要分配内存，但没有使用GFP_NOFS。这种分配可能会引起更多的文件系统操作，而这些操作又会导致另外的分配，从而再引起更多的文件系统操作！这样一直持续下去。这样的代码在调用分配器的时候，必须确保分配器不会再执行到代码本身，否则，分配就可能产生死锁。也别紧张，内核使用这两个标志的地方是极少的。

GFP_DMA标志表示分配器必须满足从ZONE_DMA进行分配的请求。这个标志用在需要DMA的内存的设备驱动程序中。一般你会把这个标志与GFP_ATOMIC和GFP_KERNEL结合起来使用。

在你编写的绝大多数代码中，用到的要么是GFP_KERNEL，要么是GFP_ATOMIC。表11-7是通常情形和所用标志的列表。不管使用那种分配类型，你都必须进行检查，并对错误进行处理。

表11-7 什么时候用哪种标志

情 形	相应标志
进程上下文，可以睡眠	使用GFP_KERNEL
进程上下文，不可以睡眠	使用GFP_ATOMIC，在你睡眠之前或之后用GFP_KERNEL执行内存分配
中断处理程序	使用GFP_ATOMIC
软中断	使用 GFP_ATOMIC
tasklet	使用 GFP_ATOMIC
需要用于DMA的内存，可以睡眠	使用（GFP_DMA | GFP_KERNEL）
需要用于DMA的内存，不可以睡眠	使用（GFP_DMA | GFP_ATOMIC），或在你睡眠之前执行内存分配

11.4.2 kfree()

kmalloc()的另一端就是kfree()，kfree()声明于<linux/slab.h>中：

```
void kfree(const void *ptr)
```

kfree()函数释放由kmalloc()分配出来的内存块。如果想要释放的内存不是由kmalloc()分配的，或者想要释放的内存早就被释放了，比如说释放属于内核其他部分的内存，调用这个函数会导致严重的后果。与用户空间类似，分配和回收要注意配对使用，以避免内存泄漏和其他bug。注意，调用kfree(NULL)是安全的。

让我们看一个在中断处理程序中分配内存的例子。在这个例子中，中断处理程序想分配一个缓冲区来保存输入数据。BUF_SIZE预定义为以字节为单位的缓冲区长度，它应该是大于两个字节的。

```
char *buf;

buf = kmalloc(BUF_SIZE, GFP_ATOMIC);
if (!buf)
        /*内存分配出错! */
```

之后，当我们不再需要这个内存时，别忘了释放它：

```
kfree(buf);
```

11.5 vmalloc()

vmalloc()函数的工作方式类似于kmalloc()，只不过前者分配的内存虚拟地址是连续的，而物理地址则无需连续。这也是用户空间分配函数的工作方式：由malloc()返回的页在进程的虚拟地址空间内是连续的，但是，这并不保证它们在物理RAM中也连续。kmalloc()函数确保页在物理地址上是连续的（虚拟地址自然也是连续的）。vmalloc()函数只确保页在虚拟地址空间内是连续的。它通过分配非连续的物理内存块，再“修正”页表，把内存映射到逻辑地址空间的连续区域中，就能做到这点。

大多数情况下，只有硬件设备需要得到物理地址连续的内存。在很多体系结构上，硬件设备存在于内存管理单元以外，它根本不理解什么是虚拟地址。因此，硬件设备用到的任何内存区都必须是物理上连续的块，而不仅仅是虚地址连续的块。而仅供软件使用的内存块（例如与进程相关的缓冲区）就可以使用只有虚拟地址连续的内存块。但在你的编程中，你根本察觉不到这种差异。对内核而言，所有内存看起来都是逻辑上连续的。

尽管仅仅在某些情况下才需要物理上连续的内存块，但是，很多内核代码都用kmalloc()来获得内存，而不是vmalloc()。这主要是出于性能的考虑。vmalloc()函数为了把物理上不连续的页转换为虚拟地址空间上连续的页，必须专门建立页表项。糟糕的是，通过vmalloc()获得的页必须一个一个地进行映射（因为它们物理上是不连续的），这就会导致比直接内存映射大得多的TLB[⊖]抖动。因为这些原因，vmalloc()仅在不得已时才会使用——一般是在为了获得大块内存时，例如，当模块被动态插入到内核中时，就把模块装载到由vmalloc()分配的内存上。

vmalloc()函数在<linux/vmalloc.h>中声明，在<mm/vmalloc.c>中定义。用法与用户空间的malloc()相同：

```
void* vmalloc(unsigned long size)
```

该函数返回一个指针，指向逻辑上连续的一块内存区，其大小至少为size。在发生错误时，函数返回NULL。函数可能睡眠，因此，不能从中断上下文中进行调用，也不能从其他不允许阻塞的情况下进行调用。

要释放通过vmalloc()所获得的内存，使用下面的函数：

```
void vfree (void *addr)
```

这个函数会释放从addr开始的内存块，其中addr是以前由vmalloc()分配的内存块的地址。这个函数也可以睡眠，因此，不能从中断上下文中调用。它没有返回值。

⊖ TLB (translation lookaside buffer)是一种硬件缓冲区，很多体系结构用它来缓存虚拟地址到物理地址的映射关系。它极大地提高了系统的性能，因为大多数内存访问都要进行虚拟寻址。

这个函数用起来比较简单：

```
char *buf;

buf = vmalloc (16*PAGE_SIZE) ;  /*获得16页*/
if  (!buf)
        /* 错误! 不能分配内存*/
/* buf现在指向虚地址连续的一块内存区，其大小至少为16*PAGE_SIZE*/
```

在分配内存之后，一定要释放它：

```
vfree (buf) ;
```

11.6 slab层

分配和释放数据结构是所有内核中最普遍的操作之一。为了便于数据的频繁分配和回收，编程者常常会用到一个空闲链表。该空闲链表包含有可供使用的、已经分配好的数据结构块。当代码需要一个新的数据结构实例时，就可以从空闲链表中抓取一个，而不需要分配内存，再把数据放进去。以后，当不再需要这个数据结构的实例时，就把它放回空闲链表，而不是释放掉它。从这个意义上说，空闲链表相当于对象高速缓存以便快速存储频繁使用的对象类型。

在内核中，空闲链表面临的主要问题之一是不能全局控制。当可用内存变得紧缺时，内核无法通知每个空闲链表，让其收缩缓存的大小以便释放出一些内存来。实际上，内核根本就不知道存在任何空闲链表。为了弥补这一缺陷，也为了使代码更加稳固，Linux内核提供了slab层（也就是所谓的slab分配器）。slab分配器扮演了通用数据结构缓存层的角色。

slab分配器的概念首先在Sun Microsystem 的SunOS 5.4操作系统[⊖]中得以实现。Linux数据结构缓存层分享了它的名字和基本设计思想。

slab分配器试图在几个基本原则之间寻求一种平衡：

- 频繁使用的数据结构也会频繁分配和释放，因此应当缓存它们。
- 频繁分配和回收必然会导致内存碎片（难以找到大块连续的可用内存）。为了避免这种现象，空闲链表的缓存会连续地存放。因为已释放的数据结构又会放回空闲链表，因此不会导致碎片。
- 回收的对象可以立即投入下一次分配，因此，对于频繁的分配和释放，空闲链表能够提高其性能。
- 如果分配器知道对象大小、页大小和总的高速缓存的大小这样的概念，它会做出更明智的决策。
- 如果让部分缓存专属于单个处理器（对系统上的每个处理器独立而惟一），那么，分配和释放就可以在不加SMP锁的情况下进行。
- 如果分配器是与NUMA相关的，它就可以从相同的内存节点为请求者进行分配。
- 对存放的对象进行着色（colored），以防止多个对象映射到相同的高速缓存行（cache line）。

Linux的slab层在设计和实现时充分考虑了上述原则。

slab层的设计

slab层把不同的对象划分为所谓高速缓存（cache）组，其中每个高速缓存都存放不同类型的

⊖ 参看Bonwick, J.所著“The Slab Allocator: An Object-Caching Kernel Memory Allocator” USENIX, 1994。

对象。每种对象类型对应一个高速缓存。例如，一个高速缓存用于存放进程描述符（task_struct结构的一个空闲链表），而另一个高速缓存存放索引节点对象（struct inode）。有趣的是，kmalloc()接口建立在slab层之上，使用了一组通用高速缓存。

然后，这些高速缓存又被划分为slab（这也是这个子系统名字的来由）。slab由一个或多个物理上连续的页组成。一般情况下，slab也就仅仅由一页组成。每个高速缓存可以由多个slab组成。

每个slab都包含一些对象成员，这里的对象指的是被缓存的数据结构。每个slab处于三种状态之一：满、部分满或空。一个满的slab没有空闲的对象（slab中的所有对象都已被分配）。一个空的slab没有分配出任何对象（slab中的所有对象都是空闲的）。一个部分满的slab有一些对象已分配出去，有些对象还空闲着。当内核的某一部分需要一个新的对象时，先从部分满的slab中进行分配。如果没有部分满的slab，就从空的slab中进行分配。如果没有空的slab，就要创建一个slab了。显然，满的slab无法满足请求，因为它根本就没有空闲的对象。这种策略能减少碎片。

作为一个例子，让我们考察一下inode结构，该结构是磁盘索引节点在内存中的体现（参见第12章）。这些数据结构经常会频繁地创建和释放，因此，用slab分配器来管理它们就很有必要。因而struct inode就由inode_cachep高速缓存（这是一种标准的命名规范）进行分配的。这种高速缓存由一个或多个slab组成——由多个slab组成的可能性大一些，因为这样的对象数量很大。每个slab包含尽可能多的struct inode对象。当内核请求分配一个新的inode结构时，内核就从部分满的slab或空slab（如果没有部分满的slab）返回一个指向已分配但未使用的结构的指针。当内核用完inode对象后，slab分配器就把该对象标记为空闲。图11-1显示了高速缓存、slab及对象之间的关系。

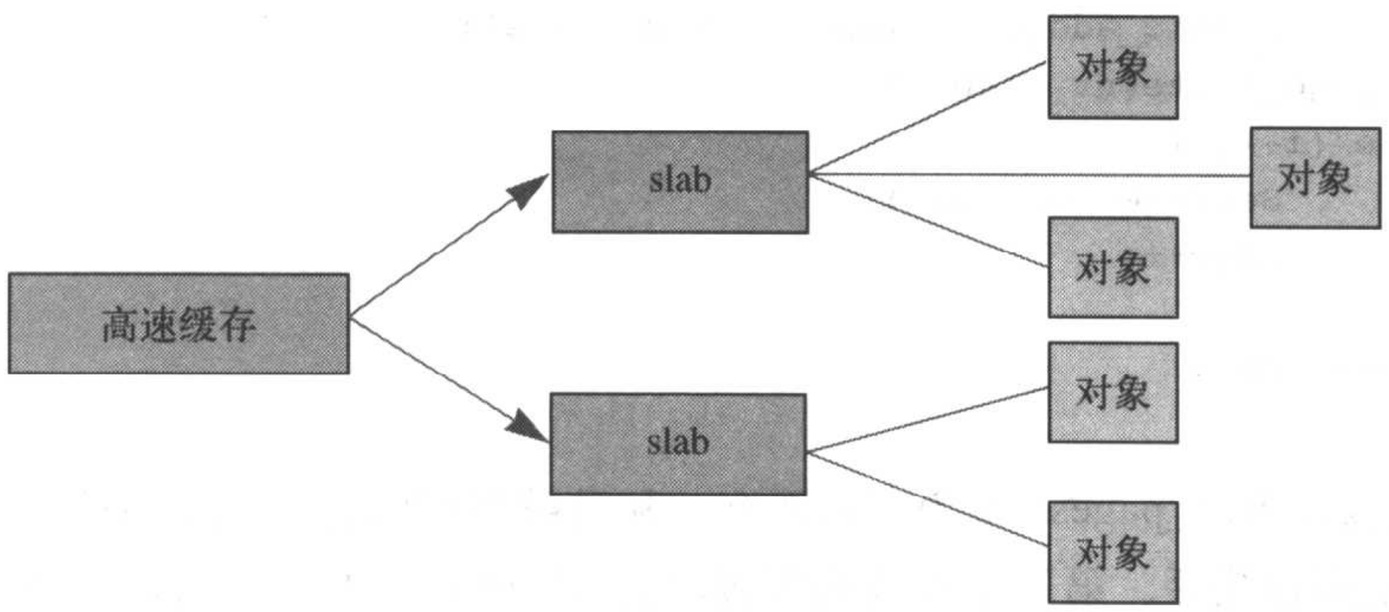


图11-1 高速缓存、slab及对象之间的关系

每个高速缓存都是用kmem_cache_s结构来表示。这个结构包含三个链表slabs_full、slabs_partial 和slabs_empty，均存放在kmem_list3结构内。这些链表包含高速缓存中的所有slab。slab描述符struct slab用来描述每个slab:

```
struct slab{
        struct list_head  list;                 /* 满、部分满或空链表 */
        unsigned long     colouroff;            /* slab着色的偏移量 */
        void              *s_mem;               /* 在slab中的第一个对象 */
        unsigned int      inuse;                /* 已分配的对象数*/
        kmem_bufctl_t     free;                 /* 第一个空闲对象（如果有的话）*/
};
```

slab描述符要么在slab之外另行分配，要么就放在slab自身最开始的地方。如果slab很小，或者

slab内部有足够的空间容纳slab描述符，那么描述符就存放在slab里面。

slab分配器可以创建新的slab，这是通过__get_free_pages()低级内核页分配器进行的：

```
static void *kmem_getpages(kmem_cache_t *cachep, int flags, int nodeid)
{
        struct page *page;
        void *addr;
        int i;

        flags |= cachep->gfpflags;
        if (likely(nodeid == -1)) {
                addr = (void*)__get_free_pages(flags, cachep->gfporder);
                if (!addr)
                        return NULL;
                page = virt_to_page(addr);
        } else {
                page = alloc_pages_node(nodeid, flags, cachep->gfporder);
                if (!page)
                        return NULL;
                addr = page_address(page);
        }

        i = (1 << cachep->gfporder);
        if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
                atomic_add(i, &slab_reclaim_pages);
        add_page_state(nr_slab, i);
        while (i--) {
                SetPageSlab(page);
                page++;
        }
        return addr;
}
```

该函数使用__get_free_pages()来为高速缓存分配足够多的内存。该函数的第一个参数就指向需要很多页的特定高速缓存。第二个参数是要传给__get_free_pages()的标志，注意，这个标志是如何与另一个值进行二进制“或”运算，这相当于把高速缓存需要的默认标志加到flags参数上。分配的页大小为2的幂次方，存放在cachep->gfporder中。前面这个函数比想像的要复杂一些，这是因为代码使得分配器与NUMA相关的。当nodeid是一个非负数时，分配器就试图从相同的内存节点给发出的请求进行分配。这在NUMA系统上提供了较好的性能，但是访问节点之外的内存会导致性能的损失。

为了便于理解，我们可以忽略与NUMA相关的代码，写一个简单的kmem_getpages()函数：

```
static inline void * kmem_getpages(kmem_cache_t *cachep, unsigned long flags)
{
        void *addr;

        flags |= cachep->gfpflags;
        addr = (void*) __get_free_pages(flags, cachep->gfporder);
```

```
        return addr;
}
```

接着，调用kmem_freepages()释放内存，而对给定的高速缓存页，kmem_freepages()最终调用的是free_pages()。当然，slab层的关键就是避免频繁分配和释放页。由此可知，slab层只有当给定的高速缓存中既没有部分满也没有空的slab时才会调用页分配函数。而只有在下列情况下才会调用释放函数：当可用内存变得紧缺时，系统试图释放出更多内存以供使用，或者当高速缓存显式地被销毁时。

slab层的管理是在每个高速缓存的基础上，通过提供给整个内核一个简单的接口来完成的。通过接口就可以创建和销毁新的高速缓存，并在高速缓存内分配和释放对象。高速缓存及其内slab的复杂管理完全通过slab层的内部机制来处理。当你创建了一个高速缓存后，slab层所起的作用就像一个专用的分配器，可以为具体的对象类型进行分配。

11.7 slab分配器的接口

一个新的高速缓存是通过以下函数创建：

```
kmem_cache_t  *kmem_cache_create(const char *name, size_t size,
                    size_t align, unsigned long flags,
                    void (*ctor)(void*, kmem_cache_t *,unsigned long),
                    void (* dtor)(void*, kmem_cache_t *,unsigned long))
```

第一个参数是一个字符串，存放着高速缓存的名字。第二个参数是高速缓存中每个元素的大小。第三个参数是高速缓存内第一个对象的偏移。这用来确保在页内进行特定的对齐。通常情况，0就可以满足要求，也就是标准对齐。flags参数是可选的设置项，用来控制高速缓存的行为。它可以为0，表示没有特殊的行为，或者与以下标志中的一个或多个进行“或”运算：

- SLAB_NO_REAP 这个标志命令slab层不要在内存短缺时自动回收对象（也就是说，不要通过释放不使用的对象来释放内存）。通常情况下，不应该设置这个标志，因为当内存紧缺时，这样设置的高速缓存就会阻止系统回收内存，从而阻止系统继续完成操作。
- SLAB_HWCACHE_ALIGN 这个标志命令slab层把一个slab内的所有对象按高速缓存行对齐。这就防止了“错误的共享”（两个或多个对象尽管位于不同的内存地址，但映射到相同的高速缓存行）。这可以提高性能，但以增加内存踪迹为代价，因为对齐越严格，浪费的内存就越多。到底会耗费掉多少内存取决于对象的大小，以及对象相对于系统高速缓存行对齐的方式。对于会频繁使用的高速缓存，代码本身对性能要求又很严格的话，这一操作是理想的选择；否则，请三思而后行。
- SLAB_MUST_HWCACHE_ALIGN 如果调试被激活，那么，既想进行调试又想缓存对齐对象是不大可能的。这个标志强制slab层缓存对齐对象。通常情况下，这个标志是不需要的，上一个标志就足够了。在slab调试被激活（默认是禁止的）时，指定这个标志可能会极大地增加内存消耗。只有像进程描述符这样、非得在高速缓存中对齐对象这种情况，才应该设置这个标志。
- SLAB_POISON 这个标志使slab层用已知的值（a5a5a5a5）填充slab。这就是所谓的“中毒”，有利于对未初始化内存的访问。
- SLAB_RED_ZONE 这个标志导致slab层在已分配的内存周围插入“红色警界区”以探测缓

冲越界。

- SLAB_PANIC 这个标志当分配失败时提醒slab层。这在要求分配只能成功的时候非常有用。比如，在系统初启时分配一个VMA结构的高速缓存（参见14章，“进程地址空间”）。
- SLAB_CACHE_DMA 这个标志命令slab层使用可以执行DMA的内存给每个slab分配空间。只有在分配的对象用于DMA，而且必须驻留在ZONE_DMA区时才需要这个标志。否则，你既不需要也不应该设置这个标志。

最后两个参数ctor和dtor分别是高速缓存的构造和析构函数。只有在新的页追加到高速缓存时，构造函数才被调用。只有从高速缓存中删去页时，析构函数才被调用。有一个析构函数就要有一个构造函数。实际上，Linux内核的高速缓存不使用析构函数或构造函数。你可以将这两个参数都赋为NULL。

kmem_cache_create()在成功时会返回一个指向所创建高速缓存的指针，否则，返回NULL。这个函数不能在中断上下文中调用，因为它可能会睡眠。

要销毁一个高速缓存，则调用：

```
int kmem_cache_destroy(kmem_cache_t *cachep)
```

顾名思义，这样就可以销毁给定的高速缓存。这个函数通常在模块的注销代码中被调用，当然，这里指创建了自己的高速缓存的模块。同样，也不能从中断上下文中调用这个函数，因为它也可能睡眠。调用该函数之前必须确保存在以下两个条件：

- 高速缓存中的所有slab都必须为空。其实，不管哪个slab中只要还有一个对象被分配出去，并正在使用的话，那怎么可能销毁这个高速缓存呢？
- 在调用kmem_cache_destroy()期间（更不用说在调用之后了）不再访问这个高速缓存。调用者必须确保这种同步。

该函数成功执行，返回0；否则，返回非0值。

创建高速缓存之后，就可以通过下列函数从中获取对象：

```
void *kmem_cache_alloc(kmem_cache_t *cachep, int flags)
```

该函数从给定的高速缓存cachep中返回一个指向对象的指针。如果高速缓存的所有slab中都没有空闲的对象，那么slab层必须通过kmem_getpages()获取新的页，flags的值传递给__get_free_pages()。这与我们前面看到的标志相同，你用到的应该是GFP_KERNEL或GFP_ ATOMIC。

最后释放一个对象，并把它返回给原先的slab，可以使用下面这个函数：

```
void kmem_cache_free(kmem_cache_t *cachep, void *objp)
```

这样就能把cachep中的对象objp标记为空闲了。

slab分配器的使用实例

让我们考察一个鲜活的实例，这个例子用的是task_struct结构（进程描述符）。代码稍微有点复杂，取自kernel/fork.c。

首先，内核用一个全局变量存放指向task_struct高速缓存的指针：

```
kmem_cache_t *task_struct_cachep;
```

在内核初始化期间，在fork_init()中会创建高速缓存：

```
task_struct_cachep = kmem_cache_create("task_struct",
                                       sizeof(struct task_struct),
                                       ARCH_MIN_TASKALIGN,
                                       SLAB_PANIC,
                                       NULL,
                                       NULL);
```

这样就创建了一个名为task_struct_cachep的高速缓存，其中存放的就是类型为struct task_struct的对象。该对象被创建后存放在slab中偏移量为ARCH_MIN_TASKALIGN个字节的地方，ARCH_MIN_TASKALIGN预定义值是与体系结构相关的值。没有构造函数或析构函数。注意不用检查返回值是否为失败标记NULL，因为SLAB_PANIC标志已经被设置了。如果分配失败，slab分配器就调用panic()函数。如果没有提供SLAB_PANIC标志，就必须自己检查返回值。SLAB_PANIC标志用在这儿是因为这是系统操作必不可少的高速缓存（没有进程描述符，机器自然不能正常运行）。

每当进程调用fork()时，一定会创建一个新的进程描述符（回忆一下第3章）。这是在dup_task_struct()中完成的，而该函数会被do_fork()调用：

```
struct task_struct *tsk;

tsk = kmem_cache_alloc(task_struct_cachep, GFP_KERNEL);
if (!tsk)
        return NULL;
```

进程执行完后，如果没有子进程在等待的话，它的进程描述符就会被释放，并返回给task_struct_cachep slab高速缓存。这是在free_task_struct()中执行的（这里，tsk是现有的进程）：

```
kmem_cache_free(task_struct_cachep, tsk);
```

由于进程描述符是内核的核心组成部分，时刻都要用到，因此task_struct_cachep高速缓存绝不会被销毁掉。即使真能销毁，我们也要通过下列函数阻止其被销毁：

```
int err;

err = kmem_cache_destroy(task_struct_cachep);
if (err)
        /*撤销高速缓存出错*/
```

很容易吧？slab层负责其间所有底层的对齐、着色、分配、释放和回收等等。如果你要频繁创建很多相同类型的对象，那么，就应该考虑使用slab高速缓存。也就是说，不要自己去实现空闲链表！

11.8 在栈上的静态分配

在用户空间，我们以前所讨论到的那些分配的例子，有不少都可以在栈上发生。因为我们毕竟可以事先知道所分配空间的大小。用户空间能够奢侈地负担起非常大的栈，而且栈空间还可以动态增长，相反，内核却不能这么奢侈——内核栈大小固定。当给每个进程分配一个固定大小的小栈，不但可以减少内存的消耗，而且内核也无需负担太重的栈管理任务。

每个进程的内核栈大小既取决于体系结构，也与编译时的选项有关。历史上，每个进程都有两页的内核栈。因为32位和64位体系结构的页面大小分别是4KB和8KB，所以通常它们内核栈的大小分别是8KB和16KB。

但是，在2.6系列内核的早期，引入了一个选项可以设置单页内核栈。当这个选项激活时，每个进程的内核栈只有一页那么大，根据体系结构的不同，或为4KB，或为8KB。这么做出于两个原因：首先，可以让每个进程减少内存消耗；另外，更最重要的是，随着机器运行时间的增加，寻找两个未分配的、连续的页变得越来越困难。越来越多的物理内存变为碎片， 因此，给一个新进程分配虚拟内存（VM）的压力也增大。

还有一个更复杂的原因。请继续跟随我探究：我们几乎掌握了关于内核栈的全部知识。现在，每个进程的整个调用链必须放在自己的内核栈中。不过，中断处理程序也曾经使用它们所中断的进程的内核栈，这样，中断处理程序也要放在内核栈中。这当然有效而简单，但是，这同时会把更严格的约束条件加在这可怜的内核栈上。当我们把栈移到只有一个页面的内核栈时，中断处理程序就不放在栈中了。

为了矫正这个问题，实现了一个附加选项：中断栈。中断栈为每个进程提供一个用于中断处理程序的栈。有了这个选项，中断处理程序不用再和被中断进程共享一个内核栈，它们可以使用自己的栈了。对每个进程来说只不过耗费了一页而已。

总的来说，内核栈可以是1页，也可以是2页，这取决于编译时配置选项。栈大小因此在4KB~16KB的范围内。历史上，中断处理程序和被中断进程共享一个栈。当1页栈的选项激活时，中断处理程序获得了自己的栈。在任何情况下，无限制的递归和alloca()显然是不允许的。

好，就讲到这里。大家明白了吗?

在栈上静态分配

在任意一个函数中，你都必须尽量节省栈资源。这并不难，也没有什么固定的办法，你只需要在具体的函数中让所有局部变量（即所谓的自动变量）所占空间之和不要超过几百字节。在栈上进行大量静态分配，比如分配大型数组和大型结构体，是很危险的。要不然，在内核中和在用户空间中进行的栈分配就没有什么差别了。栈溢出发生时悄无声息，但势必会引起严重的问题。因为内核没有在管理内核栈上做足工作，因此，当栈溢出时，多出的数据就会直接溢出来，覆盖掉紧邻堆栈末端的东西。首先面临考验的就是thread_info结构（回想一下第3章，这个结构就贴着每个进程内核堆栈的末端）。在堆栈之外，任何内核数据都可能存在潜在的危险。当栈溢出时，最好的情况是机器宕机，最坏的情况是悄无声息地破坏数据。

因此，进行动态分配是一种明智的选择，本章前面有关大块内存的分配就是采用这种方式。

11.9 高端内存的映射

根据定义，在高端内存中的页不能永久地映射到内核地址空间上。因此，通过alloc_pages()函数以__GFP_HIGHMEM标志获得的页不可能有逻辑地址。

在x86体系结构上，高于896MB的所有物理内存的范围大都是高端内存，它并不会永久地或自动地映射到内核地址空间，尽管x86处理器能够寻址物理RAM的范围达到4GB（启用PAE可以寻址到64GB）。一旦这些页被分配，就必须映射到内核的逻辑地址空间上。在x86上，高端内存中的页被映射到3GB到4GB之间。

11.9.1 永久映射

要映射一个给定的page结构到内核地址空间，可以使用：

```
void *kmap(struct page *page)
```

这个函数在高端内存或低端内存上都能用。如果page结构对应的是低端内存中的一页，函数只会单纯地返回该页的虚拟地址。如果页位于高端内存，则会建立一个永久映射，再返回地址。这个函数可以睡眠，因此kmap()只能用在进程上下文中。

因为允许永久映射的数量是有限的（如果没有这个限制，我们就不必搞得这么复杂，把所有内存通通映射为永久内存就行了），当不再需要高端内存时，应该解除映射，这可以通过下列函数完成：

```
void kunmap(struct page* page)
```

该函数解除对给定页的映射。

11.9.2 临时映射

当必须创建一个映射而当前的上下文又不能睡眠时，内核提供了临时映射（也就是所谓的原子映射）。有一组保留的映射，它们可以存放新创建的临时映射。内核可以原子地把高端内存中的一个页映射到某个保留的映射中。因此，临时映射可以用在不能睡眠的地方，比如中断处理程序中，因为获取映射时绝不会阻塞。

可通过下列函数建立一个临时映射：

```
void *kmap_atomic(struct page *page, enum km_type type)
```

参数type是下列枚举类型之一，这些枚举类型描述了临时映射的目的。它们定义于<asm/kmap_types.h>中：

```
enum km_type{
        KM_BOUNCE_READ,
        KM_SKB_SUNRPC_DATA,
        KM_SKB_DATA_SOFTIRQ,
        KM_USER0,
        KM_USER1,
        KM_BIO_SRC_IRQ,
        KM_BIO_DST_IRQ,
        KM_PTE0,
        KM_PTE1,
        KM_PTE2,
        KM_IRQ0,
        KM_IRQ1,
        KM_SOFTIRQ0,
        KM_SOFTIRQ1,
        KM_TYPE_NR
};
```

这个函数不会阻塞，因此可以用在中断上下文和其他不能重新调度的地方。它还能禁止内核抢占，这是有必要的，因为映射对每个处理器都是惟一的（调度可能对哪个处理器执行哪个进程做变动）。

可通过下列函数取消映射：

```
void knumap_atomic(void *kvaddr, enum km_type type)
```

这个函数也不会阻塞。事实上，在很多体系结构中，除非激活了内核抢占，否则 kmap_atomic() 根本就无事可做，因为只有在下一个临时映射到来前上一个临时映射才有效。因此，内核完全可以

“忘掉”kmap_atomic()映射，kunmap_atomic()也无需做什么实际的事情。下一个原子映射将自动覆盖前一个映射。

11.10 每个CPU的分配

支持SMP的现代操作系统使用每个CPU上的数据，对于给定的处理器其数据是惟一的。一般来说，每个CPU的数据存放在一个数组中。数组中的每一项对应着系统上一个存在的处理器。当前处理器号确定这个数组的当前元素，这就是2.4内核处理每个CPU数据的方式。这种方式还不错，因此，2.6内核的很多代码依然用它。可以像下面这样声明数据：

```
unsigned long my_percpu[NR_CPUS];
```

然后，按如下方式访问它

```
int cpu;
cpu = get_cpu();    /* 获得当前处理器，并禁止内核抢占*/
my_percpu[cpu]++;   /* ... 或者无论什么 */
printk("my_percpu on cpu=%d is %lu\n", cpu, my_percpu[cpu]);
put_cpu();          /* 激活内核抢占*/
```

注意，上面的代码中并没有出现锁，这是因为所操作的数据对当前处理器来说是惟一的。除了当前处理器之外，没有其他处理器可接触到这个数据，不存在并发访问问题，所以当前处理器可以在不用锁的情况下安全地访问它。

现在，内核抢占成为了惟一需要关注的问题了，内核抢占会引起下面提到的两个问题：

- 如果你的代码被其他处理器抢占并重新调度，那么这时cpu变量就会无效，因为它指向的是错误的处理器（通常，代码获得当前处理器后是不可以睡眠的）。
- 如果别的任务抢占了你的代码，那么有可能在同一个处理器上发生并发访问my_percpu的情况，显然这处于一种竞争状况。

虽然如此，但是你大可不必惊慌，因为在获取当前处理器号，即调用get_cpu()时，就已经禁止了内核抢占。相应地，smp_processor_id()在调用put_cpu()时又会重新激活当前处理器号。注意，如果你使用对smp_processor_id()的调用来获得当前处理器号，只要你总使用上述方法来保护数据安全，那么内核抢占并不需要你自己去禁止。

11.11 新的每个CPU接口

2.6内核为了方便创建和操作每个CPU数据，从而引进了新的操作接口，称作percpu。该接口归纳了前面所述的操作行为，并使每个CPU数据的创建和操作得以简化。

但前面我们讨论的创建和访问每个CPU数据的方法依然有效，不过大型对称多处理器计算机要求对每个CPU数据操作更简单，功能更强大，正是在这种背景下，新接口应运而生。

头文件<linux/percpu.h>声明了所有的接口操作例程，可以在文件mm/slab.c和 <asm/percpu.h>中找到它们的定义。

11.11.1 编译时的每个CPU数据

在编译时定义每个cpu变量易如反掌：

```
DEFINE_PER_CPU(type, name);
```

这个语句为系统中的每一个处理器都创建了一个内型为type，名字为name的变量实例，如果你需要在别处声明变量，以防范编译时警告，那么下面的宏将是你的好帮手：

```
DECLARE_PER_CPU(type, name);
```

你可以利用get_cpu_var()和put_cpu_var()例程操作变量。调用get_cpu_var()返回当前处理器上的指定变量，同时它将禁止抢占；另一方面put_cpu_var()将相应地重新激活抢占。

```
get_cpu_var(name)++;      /* 增加该处理器上的name变量的值*/
put_cpu_var(name);        /* 完成；重新激活内核抢占*/
```

你也可以获得别的处理器上的每个CPU数据：

```
per_cpu(name, cpu)++;     /* 增加指定处理器上的name变量的值*/
```

使用此方法时需格外小心，因为per_cpu()函数既不会禁止内核抢占，也不会提供任何形式的锁保护。如果一些处理器可以接触到其他处理器的数据，那么就必须要给数据上锁。注意，第8章和第9章为我们详细讨论了数据上锁问题。

另外还有一个需要提醒的问题：这些编译时每个CPU数据的例子并不能在模块内使用，因为连接程序实际上将它们创建在一个惟一的可执行段中（.data.percpu），如果你需要从模块中访问每个CPU数据，或者如果你需要动态创建这些数据，那还是有希望的。

11.11.2 运行时的每个CPU数据

内核实现每个CPU数据的动态分配方法类似于kmalloc()。该例程为系统上的每个处理器创建所需内存的实例，其原型在文件<linux/percpu.h>中：

```
void *alloc_percpu(type);  /* a macro */
void *__alloc_percpu(size_t size, size_t align);
void free_percpu(const void *);
```

宏alloc_percpu()给系统中的每个处理器分配一个指定类型对象的实例。它其实是宏__alloc_percpu()的一个封装，这个原始宏接收的参数有两个：一个是要分配的实际字节数；一个是分配时要按多少字节对齐。而封装后的_alloc_percpu()按照单字节对齐——按照给定类型的自然边界对齐。这种对齐方式最为常用。比如：

```
struct rabid_cheetah = alloc_percpu(struct rabid_cheetah);
```

它等价于：

```
struct rabid_cheetah = __alloc_percpu(sizeof (struct rabid_cheetah),
                                      __alignof__ (struct rabid_cheetah));
```

__alignof__结构是gcc的一个功能，它会返回指定类型或lvalue所需的（或建议的，要知道有些古怪的体系结构并没有字节对齐的要求）对齐字节数。它语义和sizeof一样，比如：

```
__alignof__ (unsigned long)
```

在x86体系中将返回4。如果指定一个lvalue，那么将返回lvalue的最大对齐字节数。比如一个结构中的lvalue相比结构外的lvalue可能有更大的对齐字节需求，这是结构本身的对齐要求的缘故。有关对齐的进一步讨论我们放在第19章中介绍。

相应调用free_percpu()将释放所有处理器上指定的每个CPU数据。

无论是alloc_percpu() 还是 __alloc_percpu()都会返回一个指针，它用来间接引用动态创建的每个CPU数据，内核提供了两个宏来利用指针获取每个CPU数据：

```
get_cpu_ptr(ptr);     /* 返回一个void类型指针，该指针指向处理器的ptr的拷贝 */
put_cpu_ptr(ptr);     /* 完成：重新激活内核抢占 */
```

get_cpu_ptr()宏返回了一个指向当前处理器数据的特殊实例，它同时会禁止内核抢占；而在put_cpu_ptr() 宏中会重新激活内核抢占。

我们来看一个使用这些函数的完整例子。当然这个例子有点无聊，因为通常你会一次分配够内存（比如，在某些初始化函数中），然后可以在各种地方使用它，然后再一次释放（比如，在一些清理函数中）。不过，这个例子可很清楚地说明如何使用这些函数。

```
void *percpu_ptr;
unsigned long *foo;

percpu_ptr = alloc_percpu(unsigned long);
if (!ptr)
        /* 内存分配错误 */

foo = get_cpu_ptr(percpu_ptr);
/* 操作foo .. */
put_cpu_ptr(percpu_ptr);
```

最后，函数per_cpu_ptr()返回了指定处理器的惟一数据。

```
per_cpu_ptr(ptr, cpu);
```

请注意，上面的函数不会禁止内核抢占。因此如果你需要访问另外处理器的数据，一定要记住你可能需要给数据上锁。

11.12 使用每个CPU数据的原因

使用每个CPU数据具有不少好处。首先是减少了数据锁定。因为按照每个处理器访问每个CPU数据的逻辑，你可以不再需要任何锁。记住“只有这个处理器能访问这个数据”的规则纯粹是一个编程约定。你需要确保本地处理器只会访问它自己的惟一数据。系统本身并不存在任何措施禁止你从事欺骗活动。

第二个好处是使用每个CPU数据可以大大减少缓存失效。失效发生在处理器试图使它们的缓存保持同步时。如果一个处理器操作某个数据，而该数据又存放在其他处理器缓存中，那么存放该数据的那个处理器必须清理或刷新它自己的缓存。持续不断的缓存失效称为*缓存抖动*，这样对系统性能影响颇大。使用每个CPU数据将使得缓存影响降至最低，因为理想情况下只会访问它自己的数据。percpu接口缓存一对齐（cache-aligns）所有数据，以便确保在访问一个处理器的数据时，不会将另一个处理器的数据带入同一个缓存线上。

综上所述，使用每个CPU数据会省去许多（或最小化）数据上锁，它惟一的安全要求就是要禁止内核抢占。而这点代价相比上锁要小得多，而且接口会自动帮你做这个步骤。每个CPU数据在中断上下文或进程上下文中使用都很安全。但要注意，你可不能在访问每个CPU数据过程中睡眠（否则，你就又可能醒来后已经到了其他处理器上了）。

目前并不要求必须使用每个CPU的新接口。只要你禁止了内核抢占，用手动方法（利用我们原来讨论的数组）就很好，但是新接口在将来将更容易使用，而且功能也会得到长足的优化。如果你真决定在你的内核中使用每个CPU数据，请考虑使用新接口。但我要提醒的是——新接口并不向后兼容老内核。

11.13 分配函数的选择

如果你需要连续的物理页，就可以使用某个低级页分配器或kmalloc()。这是内核中内存分配的常用方式，也是大多数情况下你自己应该使用的内存分配方式。回忆一下，传递给这些函数的两个最常用的标志是GFP_ATOMIC和GFP_KERNEL。GFP_ATOMIC表示进行不睡眠的高优先级分配。这是中断处理程序和其他不能睡眠的代码段的需要。对于可以睡眠的代码，比如没有持自旋锁的进程上下文代码，则应该使用GFP_KERNEL获取所需的内存。这个标志表示如果有必要，分配时可以睡眠。

如果你想从高端内存进行分配，就使用alloc_pages()。alloc_pages()函数返回一个指向struct page结构的指针，而不是一个指向某个逻辑地址的指针。因为高端内存很可能并没有被映射，因此，访问它的惟一方式就是通过相应的struct page结构。为了获得真正的指针，应该调用kmap()，把高端内存映射到内核的逻辑地址空间。

如果你不需要物理上连续的页，而仅仅需要虚拟地址上连续的页，那么就使用vmalloc()（不过要记住vmalloc()相对kmalloc()来说，有一定的性能损失）。vmalloc()函数分配的内存虚地址是连续的，但它本身并不保证物理上的连续。这与用户空间的分配非常类似，它也是把物理内存块映射到连续的逻辑地址空间上。

如果你要创建和销毁很多较大的数据结构，那么应考虑建立slab高速缓存。slab层会给每个处理器维持一个对象高速缓存（空闲链表），这种高速缓存会极大地提高对象分配和回收的性能。slab层不是频繁地分配和释放内存，而是为你把事先分配好的对象存放到高速缓存中。当你需要一块新的内存来存放数据结构时，slab层一般无需另外去分配内存，而只需要从高速缓存中得到一个对象就可以了。

第⑫章

虚拟文件系统

虚拟文件系统（有时也称作虚拟文件交换，更常见的是简称VFS）作为内核子系统，为用户空间程序提供了文件系统相关的接口。系统中所有文件系统不但依赖VFS共存，而且也依靠VFS系统协同工作。通过虚拟文件系统，程序可以利用标准的UNIX文件系统调用对不同介质上的不同文件系统进行读写操作，如图12-1所示。

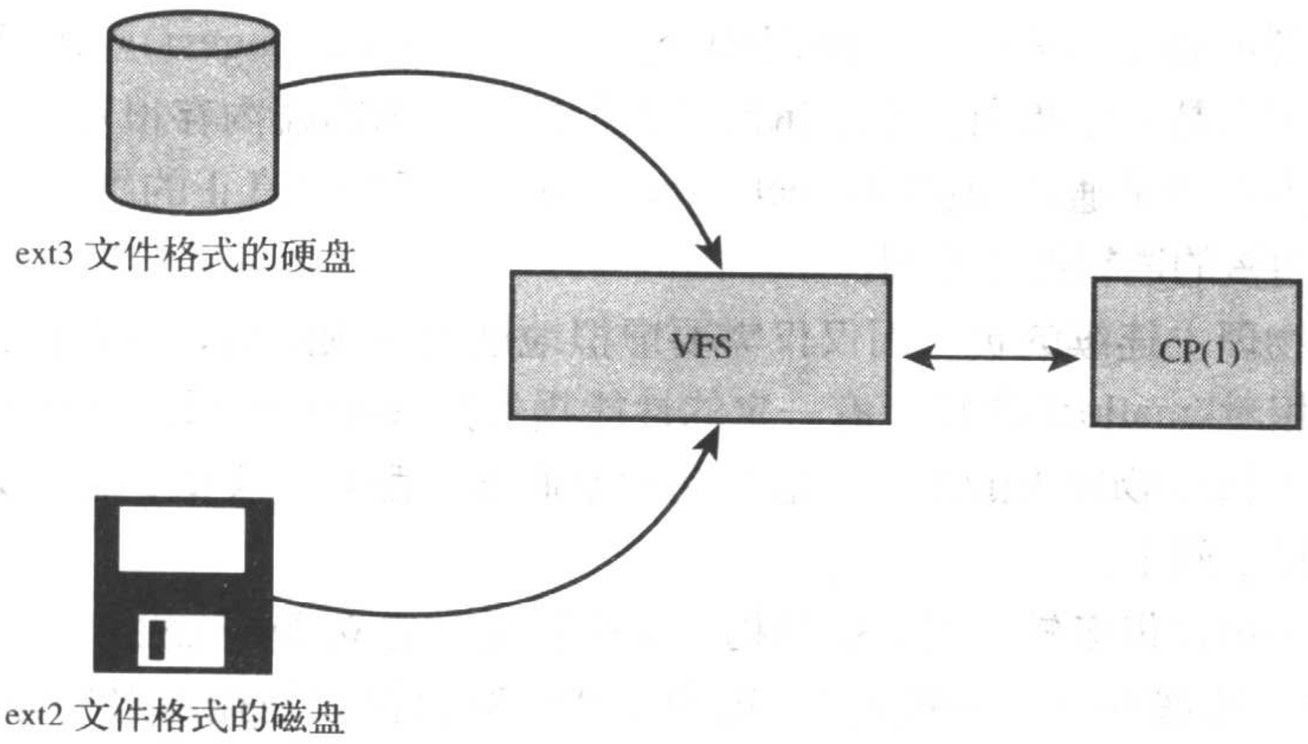


图12-1　VFS执行的动作：使用cp(1)命令从ext3文件系统格式的硬盘拷贝数据到ext2文件系统格式的可移动磁盘上。两种不同的文件系统，两种不同的介质，连接到同一个VFS上

12.1　通用文件系统接口

VFS使得用户可以直接使用open()、read()和write()这样的系统调用而无需考虑具体文件系统和实际物理介质。现在听起来这并没什么新奇的——我们早就认为这是理所当然的——但是，使得这些通用的系统调用可以跨所有介质和文件系统执行，绝非是微不足道的成绩。更了不起的是，系统调用可以在这些不同的文件系统和介质之间执行——我们可以使用标准的系统调用从一个文件系统拷贝或移动数据到另一个文件系统。老式的操作系统（想想DOS）是无力完成上述工作的，任何对非本地文件系统的访问都必须依靠特殊工具才能完成。正是由于包括Linux在内的现代操作系统引入抽象层，通过虚拟接口访问文件系统，才使得这种协作性和通用性成为可能。新的文件系统和新种类的存储介质都能找到进入Linux之路，程序无需重写，甚至无需重新编译。

12.2　文件系统抽象层

之所以可以使用这种通用接口对所有类型的文件系统进行操作，是因为内核在它的底层文件

系统接口上建立了一个抽象层。该抽象层使Linux能够支持各种文件系统，即便是它们在功能和行为上存在很大差别。为了支持多文件系统，VFS提供了一个通用文件系统模型，该模型囊括了我们所能想到的文件系统的常用功能和行为。当然，该模型源于Unix风格的文件系统（我们将在后面的小节看到）。但即使这样，Linux仍然可以支持很多种差异很大的文件系统。

VFS抽象层之所以能衔接各种各样的文件系统，是因为它定义了所有文件系统都支持的基本的、概念上的接口和数据结构。同时实际文件系统也将自身的诸如“如何打开文件”，“目录是什么”等概念在形式上与VFS的定义保持一致。因为实际文件系统的代码在统一的接口和数据结构下隐藏了具体的实现细节，所以在VFS层和内核的其他部分看来，所有文件系统都是相同的，它们都支持像文件和目录这样的概念，同时也支持像创建文件和删除文件这样的操作。

内核通过抽象层能够方便、简单地支持各种类型的文件系统。实际文件系统通过编程提供VFS所期望的抽象接口和数据结构，这样，内核就可以毫不费力地和任何文件系统协同工作。并且这样提供给用户空间的接口，也可以和任何文件系统无缝地连接在一起，完成实际工作。

其实在内核中，除了文件系统本身外，其他部分并不需要了解文件系统的内部细节。比如一个简单的用户空间程序执行如下的操作：

```
write(f, &buf, len);
```

该代码将&buf指针指向的、长度为len字节的数据写入文件描述符f对应的文件的当前位置。这个用户调用首先被一个通用系统调用sys_write()处理，sys_write()函数要找到f所在的文件系统实际给出的是哪个写操作，然后再执行该操作。实际文件系统的写方法是文件系统实现的一部分，数据最终通过该操作写入介质（或执行这个文件系统想要完成的写动作）。图12-2描述了从用户空间的write()调用到数据被写入磁盘介质的整个流程。一方面，系统调用是通用VFS接口，提供给用户空间的前端；另一方面，系统调用是具体文件系统的后端，处理实现细节。接下来的小节中我们会具体看到VFS抽象模型以及它提供的接口。

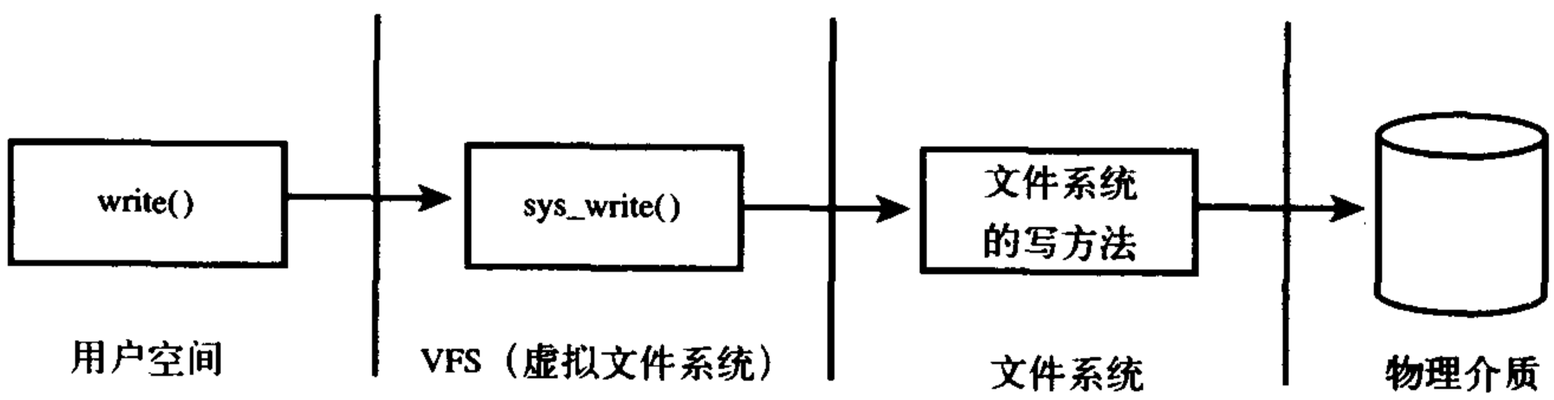


图12-2 write()调用将来自用户空间的数据流，首先通过VFS的通用系统调用，其次通过文件系统的特殊写方法，最后写入物理介质中

12.3 Unix 文件系统

Unix使用了四种和文件系统相关的传统抽象概念：文件、目录项、索引节点和安装点(mount point)。

从本质上讲文件系统是特殊的数据分层存储结构，它包含文件、目录和相关的控制信息。文件系统的通用操作包含创建、删除和安装等等。在Unix中，文件系统被安装在一个特定的安装点

上，该安装点在全局层次结构[⊖]中被称作命名空间，所有的已安装文件系统都作为根文件系统树[⊜]的枝叶出现在系统中。

文件其实可以看作是一个有序字节串，字节串中第一个字节是文件的头，最后一个字节是文件的尾。每一个文件为了便于系统和用户识别，都被分配了一个便于理解的名字。典型的文件操作有读、写、创建和删除等。Unix的文件概念与面向记录的文件系统（如OpenVMS的File-11）形成鲜明的对照。面向记录的文件系统提供更丰富、更结构化的表示，而简单的面向字节流抽象的Unix文件则以简单性和相当的灵活性为代价。

文件通过目录组织起来。文件目录好比一个文件夹，用来容纳相关文件。因为目录也可以包含子目录，所以目录可以层层嵌套，形成文件路径。路径中的每一部分都被称作目录条目。“/home/wolfman/butter”是文件路径的一个例子——根目录是/，目录home,wolfman和文件butter都是目录条目，它们被统称为*目录项*。在Unix中，目录属于普通文件，它列出包含在其中的所有文件。由于VFS把目录当作文件对待，所以可以对目录执行和文件相同的操作。

Unix系统将文件的相关信息和文件本身这两个概念加以区分，例如访问控制权限、大小、拥有者、创建时间等等信息。文件相关信息，有时被称作文件的元数据（也就是说，文件的相关数据），被存储在一个单独的数据结构中，该结构被称为*索引节点*（inode），它其实是index node的缩写，不过近来术语“inode”使用得更为普遍一些。

所有这些信息都和文件系统的控制信息密切交融，文件系统的控制信息存储在超级块中，超级块是一种包含文件系统信息的数据结构。有时，把这些收集起来的信息称为文件系统数据元，它集单独文件信息和文件系统的信息于一身。

一直以来，Unix文件系统在它们物理磁盘布局中也是按照上述概念实现的。比如说在磁盘上，文件(目录也属于文件)信息按照索引节点形式存储在单独的块中，控制信息被集中存储在磁盘的超级块中，等等。Linux 的VFS的设计目标就是要保证能与支持和实现了这些概念的文件系统协同工作。像如FAT或NTFS这样的非Unix风格的文件系统，虽然也可以在Linux上工作，但是它们必须经过封装，提供一个符合这些概念的界面。比如，即使一个文件系统不支持索引节点，它也必须在内存中装配索引节点结构体，就像它本身包含索引节点一样。再比如，如果一个文件系统将目录看作是一种特殊对象，那么要想使用VFS，就必须将目录重新表示为文件形式。通常，这种转换需要在使用现场（on the fly）引入一些特殊处理，使得非Unix文件系统能够兼容Unix文件系统的使用规则并满足VFS的需求。通过这些处理，非Unix文件系统便可以和VFS一同工作了，只是在性能上多少会受一些影响。

12.4 VFS 对象及其数据结构

VFS其实采用的是面向对象[⊜]的设计思路，使用一族数据结构来代表通用文件对象。这些数据

⊖ 近来，Linux已经将这种层次化概念引入了单个进程中，每个进程都指定一个惟一的命名空间。因为每个进程都会继承父进程的命名空间（除非是特别声明的情况），所以所有进程往往都只有一个全局命名空间。

⊖ 这点区别于指定驱动盘符，如“C:”，这打破了分区和设备之间名字空间的界限。因为这呈现给用户的有点随意性，因此一点也不理想。

⊜ 人们时常忽略，甚至会否认，但是在内核中确实存在很多利用面向对象思想编程的例子。虽然内核开发者可能有意避免C++和其他面向对象语言，但是面向对象的思想仍然经常被借鉴——虽然C语言缺乏面向对象的机制。VFS就是一个利用C代码来有效和简洁地实现OOP的例子。

结构表现得就像是对象。因为内核纯粹使用C代码实现，没有直接利用面向对象的语义，所以内核中的数据结构都使用C结构体实现，但这些结构体包含数据的同时也包含操作这些数据的函数指针，其中的操作函数由具体文件系统实现。

VFS中有四个主要的对象类型，它们分别是：

- 超级块对象，它代表一个已安装文件系统。
- 索引节点对象，它代表一个文件。
- 目录项对象，它代表一个目录项，是路径的一个组成部分。
- 文件对象，它代表由进程打开的文件。

注意，因为VFS将目录作为一个文件来处理，所以不存在目录对象。回忆本章前面所提到的目录项代表的是路径中的一个组成部分，它可能包括一个普通文件。换句话说，目录项不同于目录，但目录却和文件相同，明白了吗？

每个主要对象中都包含一个操作对象，这些操作对象描述了内核针对主要对象可以使用的方法。最主要的几种操作对象如下：

- super_operations对象，其中包括内核针对特定文件系统所能调用的方法，比如read_inode()和sync_fs()等方法。
- inode_operations对象，其中包括内核针对特定文件所能调用的方法，比如create()和link()等方法。
- dentry_operations对象，其中包括内核针对特定目录所能调用的方法，比如d_compare()和d_delete()等方法。
- file对象，其中包括进程针对已打开文件所能调用的方法，比如read()和write()等方法。

操作对象作为一个指针结构体被实现，此结构体中包含指向操作其父对象的函数指针。对于其中许多方法来说，可以继承使用VFS提供的通用函数，如果通用函数提供的基本功能无法满足需要，那么就必须使用实际文件系统的独有方法填充这些函数指针，使其指向文件系统实例。

再次提醒，我们这里所说的对象就是指结构体——而不是像C++或Jave那样的真正的对象数据类型。但是这些结构体的确代表的是一个对象，它含有相关的数据和对这些数据的操作，所以可以说它们就是对象。

其他VFS对象

VFS使用了大量结构体对象，它所包括的对象远远多于上面提到的这几种主要对象。比如每个注册的文件系统都由file_system_type结构体来表示，它描述了文件系统及其能力；另外，每一个安装点也都用vfsmount结构体表示，它包含的是安装点的相关信息，如位置和安装标志等。

在本章的最后还要介绍三个与进程相关的结构体，它们描述了文件系统以及和进程相关的文件，这三个结构体分别是file_struct、fs_struct和namespace。

后面的小节将着重讨论这些对象以及它们在VFS层的实现中扮演的角色。

12.5 超级块对象

各种文件系统都必须实现超级块，该对象用于存储特定文件系统的信息，通常对应于存放在磁盘特定扇区中（所以叫超级块对象）的文件系统超级块或文件系统控制块。对于并非基于磁盘的文

件系统（如基于内存的文件系统，比如sysfs），它们会在使用现场创建超级块并将其保存到内存中。

超级块对象由super_block结构体表示，定义在文件<linux/fs.h>中，下面给出它的结构和各个域的描述：

```
struct super_block {
        struct list_head        s_list;         /* 指向超级块链表的指针 */
        dev_t                   s_dev;          /* 设备标识符 */
        unsigned long           s_blocksize;    /* 以字节为单位的块大小 */
        unsigned long           s_old_blocksize; /* 以位为单位的旧的块大小 */
        unsigned char           s_blocksize_bits; /* 以位为单位的块大小 */
        unsigned char           s_dirt;         /* 修改（脏）标志 */
        unsigned long long      s_maxbytes;     /* 文件大小上限 */
        struct file_system_type s_type;         /* 文件系统类型 */
        struct super_operations s_op;           /* 超级块方法 */
        struct dquot_operations *dq_op;         /* 磁盘限额方法 */
        struct quotactl_ops     *s_qcop;        /* 限额控制方法 */
        struct export_operations *s_export_op;  /* 导出方法 */
        unsigned long           s_flags;        /* 登录标志 */
        unsigned long           s_magic;        /* 文件系统的魔数 */
        struct dentry           *s_root;        /* 目录登录点 */
        struct rw_semaphore     s_umount;       /* 卸载信号量 */
        struct semaphore        s_lock;         /* 超级块信号量 */
        int                     s_count;        /* 超级块引用计数 */
        int                     s_syncing;      /* 文件系统同步标志 */
        int                     s_need_sync_fs; /* 尚未同步标志 */
        atomic_t                s_active;       /* 活动引用计数 */
        void                    *s_security;    /* 安全模块 */
        struct list_head        s_dirty;        /* 脏节点链表*/
        struct list_head        s_io;           /* 回写链表 */
        struct hlist_head       s_anon;         /* 匿名目录项 */
        struct list_head        s_files;        /* 被分配文件链表 */
        struct block_device     *s_bdev;        /* 相关的块设备 */
        struct list_head        s_instances;    /* 该类型文件系统 */
        struct quota_info       s_dquot;        /* 限额相关选项 */
        char                    s_id[32];       /* 文本(text)名字*/
        void                    *s_fs_info;     /* 文件系统特殊信息*/
        struct semaphore        s_vfs_rename_sem; /* 重命名信号量 */
};
```

创建、管理和销毁超级块对象的代码位于文件fs/super.c中。超级块对象通过alloc_super()函数创建并初始化。在文件系统安装时，内核会调用该函数以便从磁盘读取文件系统超级块，并且将其信息填充到内存中的超级块对象中。

超级块操作

超级块对象中最重要的一个域是s_op，它指向超级块的操作函数表。超级块操作函数表由super_operations结构体表示，定义在文件<linux/fs.h>中，其形式如下：

```
struct super_operations {
        struct inode *(*alloc_inode)(struct super_block *sb);
        void (*destroy_inode)(struct inode *);
        void (*read_inode) (struct inode *);
        void (*dirty_inode) (struct inode *);
        void (*write_inode) (struct inode *, int);
        void (*put_inode) (struct inode *);
        void (*drop_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        int (*sync_fs)(struct super_block *, int );
        void (*write_super_lockfs) (struct super_block *);
        void (*unlockfs) (struct super_block *);
        int (*statfs) (struct super_block *, struct statfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount_begin) (struct super_block *);
        int (*show_options)(struct seq_file *, struct vfsmount *);
};
```

该结构体中的每一项都是一个指向超级块操作函数的指针，超级块操作函数执行文件系统和索引节点的低层操作。

当文件系统需要对其超级块执行操作时，首先要在超级块对象中寻找需要的操作方法。比如，如果一个文件系统要写自己的超级块，需要调用：

```
sb->s_op->write_super(sb);
```

这里的sb是指向文件系统超级块的指针，沿着该指针进入超级块操作函数表，并从表中取得希望得到的write_super()函数，该函数执行写入超级块的实际操作。注意，尽管write_super()方法来自超级块，但是在调用时，还是要把超级块作为参数传递给它，这是因为C语言中缺少对面向对象的支持，而在C++或C#中，使用如下的调用就足够了：

```
sb.write_super();
```

由于在C中无法直接得到操作函数的父对象，所以必须将父对象以参数形式传给操作函数。

下面给出super_operation中，超级块操作函数的用法说明。

- `struct inode *`
  `alloc_inode(struct super_block *sb)`

该方法在给定的超级块下创建并初始化一个新的索引节点对象。

- `void destroy_inode(struct inode *inode)`

该函数用于释放给定的索引节点。

- `void read_inode(struct inode *inode)`

该函数以inode->i_ino为索引，从磁盘上读取索引节点，并填充内存中对应的索引节点结构的剩余部分。

- `void dirty_inode(struct inode *inode)`

VFS在索引节点脏（被修改）时会调用此函数。日志文件系统（如ext3）执行该函数进行日志更新。

- void write_inode(struct inode *inode,
  int wait)

该函数用于将给定的索引节点写入磁盘。wait参数指明写操作是否需要同步。

- void put_inode(struct inode *inode)

该函数用于释放给定索引节点。

- void drop_inode(struct inode *inode)

在最后一个指向索引节点的引用被释放后，VFS会调用该函数。VFS只需要简单地删除这个索引节点后，普通Unix文件系统就不会定义这个函数了。注意调用者必须持有inode_lock锁。

- void delete_inode(struct inode *inode)

该函数用于从磁盘上删除给定的索引节点。

- void put_super(struct super_block *sb)

该函数在卸载文件系统时由VFS调用，用来释放超级块。

- void write_super(struct super_block *sb)

该函数用给定的超级块更新磁盘上的超级块。VFS通过该函数对内存中的超级块和磁盘中的超级块进行同步。

- int sync_fs(struct super_block *sb, int wait)

该函数使文件系统的数据元与磁盘上的文件系统同步。wait参数指定操作是否同步。

- void
  write_super_lockfs(struct super_block *sb)

该函数首先禁止对文件系统作改变，再使用给定的超级块更新磁盘上的超级块。目前LVM(逻辑卷标管理)会调用该函数。

- void unlockfs(struct super_block *sb)

该函数对文件系统解除锁定，它是write_super_lockfs()的逆操作。

- int statfs(struct super_block *sb,
  struct statfs *statfs)

VFS通过调用该函数获取文件系统状态。指定文件系统相关的统计信息将放置在statfs中。

- int remount_fs(struct super_block *sb,
  int *flags,char *data)

当指定新的安装选项重新安装文件系统时，VFS会调用该函数。

- void clear_inode(struct inode *)

VFS调用该函数释放索引节点，并清空包含相关数据的所有页面。

- void umount_begin(struct super_block *sb)

VFS调用该函数中断安装操作。该函数被网络文件系统使用，如NFS。

所有以上函数都是由VFS在进程上下文中调用。必要时，它们都可以阻塞。这其中的一些函数是可选的：在超级块操作表中，文件系统可以将不需要的函数指针设置成NULL。如果VFS发现操作函数指针是NULL，那它要么就会调用通用函数执行相应操作，要么什么也不做，如何选择取决于具体函数。

12.6 索引节点对象

索引节点对象包含了内核在操作文件或目录时需要的全部信息。对于Unix风格的文件系统来

说，这些信息可以从磁盘索引节点直接读入。如果一个文件系统没有索引节点，那么，不管这些相关信息在磁盘上是怎么存放的⊖，文件系统都必须从中提取这些信息。

索引节点对象由inode结构体表示，定义在文件<linux/fs.h>中，下面给出它的结构体和各项的描述。

```
struct inode {
        struct hlist_node       i_hash;                 /*散列表*/
        struct list_head        i_list;                 /*索引节点链表*/
        struct list_head        i_dentry;               /*目录项链表*/
        unsigned long           i_ino;                  /*节点号*/
        atomic_t                i_count;                /*引用计数*/
        umode_t                 i_mode;                 /*访问权限控制*/
        unsigned int            i_nlink;                /*硬连接数*/
        uid_t                   i_uid;                  /*使用者的id*/
        gid_t                   i_gid;                  /*使用者的组id*/
        kdev_t                  i_rdev;                 /*实设备标识符*/
        loff_t                  i_size;                 /*以字节为单位的文件大小*/
        struct timespec         i_atime;                /*最后访问时间*/
        struct timespec         i_mtime;                /*最后修改时间*/
        struct timespec         i_ctime;                /*最后改变时间*/
        unsigned int            i_blkbits;              /*以位为单位的块大小*/
        unsigned long           i_blksize;              /*以字节为单位的块大小*/
        unsigned long           i_version;              /*版本号*/
        unsigned long           i_blocks;               /*文件的块数*/
        unsigned short          i_bytes;                /*使用的字节数*/
        spinlock_t              i_lock;                 /*自旋锁*/
        struct rw_semaphore     i_alloc_sem;            /*嵌入在i_sem内部 */
        struct semaphore        i_sem;                  /*索引节点信号量*/
        struct inode_operations *i_op;                  /*索引节点操作表*/
        struct file_operations  *i_fop;                 /*默认的索引节点操作*/
        struct super_block      *i_sb;                  /*相关的超级块*/
        struct file_lock        *i_flock;               /*文件锁链表*/
        struct address_space    *i_mapping;             /*相关的地址映射*/
        struct address_space    i_data;                 /*设备地址映射*/
        struct dquot            *i_dquot[MAXQUOTAS];    /*节点的磁盘限额*/
        struct list_head        i_devices;              /*块设备链表*/
        struct pipe_inode_info  *i_pipe;                /*管道信息*/
        struct block_device     *i_bdev;                /*块设备驱动*/
        unsigned long           i_dnotify_mask;         /*目录通知掩码*/
        struct dnotify_struct   *i_dnotify;             /*目录通知*/
        unsigned long           i_state;                /*状态标志*/
        unsigned long           dirtied_when;           /*首次修改时间*/
        unsigned int            i_flags;                /*文件系统标志*/
        unsigned char           i_sock;                 /*是个套接字吗？ */
        atomic_t                i_writecount;           /*写者计数*/
        void                    *i_security;            /*安全模块*/
```

⊖ 没有索引节点的文件系统通常将信息作为文件的一部分存储起来，有些现代文件系统使用数据库来存储文件的数据。不管哪种情况，采用哪种方式，索引节点对象必须在内核中创建，以便文件系统使用。

```
        __u32                   i_generation;       /*索引节点版本号*/
        union {
                void            *generic_ip;        /*文件特殊信息*/
        } u;
};
```

一个索引节点代表文件系统中（虽然索引节点仅当文件被访问时，才在内存中创建）的一个文件，它也可以是设备或管道这样的特殊文件。因此索引节点结构体中有一些和特殊文件相关的项，比如i_pipe项就指向一个代表命名管道的数据结构，如果索引节点并非代表一个命名管道，那么该项就被简单地置为NULL。和特殊文件相关的项还有i_devices、i_bdev和i_cdev等。

有时，某些文件系统可能并不能完整地包含索引节点结构体要求的所有信息。举个例子，有的文件系统可能并不记录文件的访问时间，这时，该文件系统就可以在实现中选择任意合适的办法来解决这个问题。它可以在i_atime中存储0，或者让i_atime等于i_mtime，甚至任意什么其他值。

索引节点操作

和超级块操作一样，索引节点对象中的inode_operations项也非常重要，因为它描述了VFS用以操作索引节点对象的所有方法——这些方法由文件系统实现。与超级块类似，对索引节点的操作调用方式如下：

```
i->i_op->truncate(i)
```

i指向给定的索引节点，truncate()函数是由索引节点i所在的文件系统提供操作。inode_operations结构体定义在文件<linux/fs.h>中。

```
struct inode_operations {
        int (*create) (struct inode *,struct dentry *,int)
        struct dentry * (*lookup) (struct inode *,struct dentry *);
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct inode *,struct dentry *,const char *);
        int (*mkdir) (struct inode *,struct dentry *,int);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,int,dev_t);
        int (*rename) (struct inode *, struct dentry *,
                        struct inode *, struct dentry *);
        int (*readlink) (struct dentry *, char *,int);
        int (*follow_link) (struct dentry *, struct nameidata *);
        int (*put_link) (struct dentry *,struct nameidata *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *, struct dentry *, struct kstat *);
        int (*setxattr) (struct dentry *, const char *,
                         const void *,size_t, int);
        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*removexattr) (struct dentry *, const char *);
};
```

下面这些接口由各种函数组成，在给定的节点上，可能由VFS执行这些函数，也可能由具体的文件系统执行。

```
• int create(struct inode *dir,
             struct dentry *dentry, int mode)
```

VFS通过系统调用create()和open()来调用该函数从而为dentry对象创建一个新的索引节点。在创建时使用mode指定的初始模式。

```
• struct dentry * lookup(struct inode *dir,
                         struct dentry *dentry)
```

该函数在特定目录中寻找索引节点，该索引节点要对应于denrty中给出的文件名。

```
• int link(struct dentry *old_dentry,
           struct inode *dir,
           struct dentry *dentry)
```

该函数被系统调用link()调用，用来创建硬连接。硬连接名称由dentry参数指定，连接对象是dir目录中old_dentry目录项所代表的文件。

```
• int unlink(struct inode *dir,
             struct dentry *dentry)
```

该函数被系统调用unlink()调用，从目录dir中删除由目录项dentry指定的索引节点对象。

```
• int symlink(struct inode *dir,
              struct dentry *dentry,
              const char *symname)
```

该函数被系统调用symlik()调用，创建符号连接。该符号连接名称由symname指定，连接对象是dir目录中的dentry目录项。

```
• int mkdir(struct inode *dir,
            struct dentry *dentry ,int mode)
```

该函数被系统调用mkdir()调用，创建一个新目录。创建时使用mode指定的初始模式。

```
• int rmdir(struct inode *dir,
            struct dentry *dentry)
```

该函数被系统调用rmdir()调用，删除dir目录中的dentry目录项代表的文件。

```
• int mknod(struct inode *dir,
            struct dentry *dentry,
            int mode ,dev_t rdev)
```

该函数被系统调用mknod()调用，创建特殊文件（设备文件、命名管道或套接字）。要创建的文件放在dir目录中，其目录项为dentry，关联的设备为rdev，初始权限由mode指定。

```
• int rename(struct inode *old_dir,
             struct dentry *old_dentry,
             struct inode *new_dir,
             struct dentry *new_dentry)
```

VFS调用该函数来移动文件。文件源路径在old_dir目录中，源文件由old_dentry目录项所指定，目标路径在new_dir目录中，目标文件由new_dentry指定。

```
• int readlink(struct dentry *dentry,
               char *buffer, int buflen)
```

该函数被系统调用readlink()调用，拷贝数据到特定的缓冲buffer中。拷贝的数据来自dentry指

定的符号连接，拷贝大小最大可达buflen字节。

- int follow_link(struct dentry *dentry,
  struct nameidata *nd)

该函数由VFS调用，从一个符号连接查找它指向的索引节点。由dentry指向的连接被解析，其结果存放在由nd指向的nameidata结构中。

- int put_link(struct dentry *dentry,
  struct nameidata *nd)

在follow_link ()调用之后，该函数由VFS调用进行清除工作。

- void truncate(struct inode *inode)

该函数由VFS调用，修改文件的大小。在调用前，索引节点的i_size项必须被设置为预期的大小。

- int permission(struct inode *inode ,int mask)

该函数用来检查给定的inode所代表的文件是否允许特定的访问模式。如果允许特定的访问模式，返回零，否则返回负值的错误码。多数文件系统都将此区域设置为NULL，使用VFS提供的通用方法进行检查。这种检查操作仅仅比较索引节点对象中的访问模式位是否和mask一致。比较复杂的系统，比如支持访问控制链（ACL）的文件系统，需要使用特殊的permission()方法。

- int setattr(struct dentry *dentry,
  struct iattr *attr)

该函数被 notify_change()调用，在修改索引节点后，通知发生了“改变事件”。

- int getattr(struct vfsmount *mnt,
  struct dentry *dentry,
  struct kstat *stat)

在通知索引节点需要从磁盘中更新时，VFS会调用该函数。

- int setxattr(struct dentry *dentry,
  const char *name,
  const void *value, size_t size,
  int flags)

该函数由VFS调用，给dentry指定的文件设置扩展属性[⊖]。属性名为name，值为value。

- ssize_t getxattr(struct dentry *dentry,
  const char *name,
  void *value,size_t size)

该函数被VFS调用，向value中拷贝给定文件的扩展属性name对应的数值。

- ssize_t listxattr(struct dentry *dentry ,
  char *list ,size_t size)

该函数将特定文件的所有属性列表拷贝到一个缓冲列表中。

- int removexattr(struct dentry *dentry ,
  const char *name)

该函数从给定文件中删除指定的属性。

⊖ 扩展属性是2.6内核引入的一个新特色，它将键码/数值对标签传递给文件——形式类似于数据库。它们允许指定用户任意的元数据与文件相关联。

12.7 目录项对象

VFS把目录当作文件对待，所以在路径/bin/vi中，bin和vi都属于文件——bin是特殊的目录文件而vi是一个普通文件，路径中的每个组成部分都由一个索引节点对象表示。虽然它们可以统一由索引节点表示，但是VFS经常需要执行目录相关的操作，比如路径名查找等。路径名查找需要解析路径中的每一个组成部分，不但要确保它有效，而且还需要再进一步寻找路径中的下一个部分。

为了方便查找操作，VFS引入了目录项的概念。每个dentry代表路径中的一个特定部分。对前一个例子来说，/、bin和vi都属于目录项对象。前两个是目录，最后一个是普通文件。必须明确一点：在路径中，包括普通文件在内，每一个部分都是目录项对象。解析一个路径并走查其分量绝非简单的演练，它是耗时的、常规的字符串比较过程。

目录项也可包括安装点。在路径/mnt/cdrom/foo中，/、mnt.cdrom和foo都属于目录项对象。VFS在执行目录操作时——如果需要的话——会现场创建目录项对象。

目录项对象由dentry结构体表示，定义在文件<linux/dcache.h>中。下面给出该结构体和其中各项的描述：

```
struct dentry {
        atomic_t                d_count;        /*使用记数*/
        unsigned long           d_vfs_flags;    /*目录项缓存标志*/
        spinlock_t              d_lock;         /*单目录项锁*/
        struct inode            *d_inode;       /*相关的索引节点*/
        struct list_head        d_lru;          /*未用的链表*/
        struct list_head        d_child;        /*父目录中目录项对象的链表*/
        struct list_head        d_subdirs;      /*子目录*/
        struct list_head        d_alias;        /*索引节点的别名链表*/
        unsigned long           d_time;         /*重新生效时间*/
        struct dentry_operations *d_op;         /*目录项操作表*/
        struct super_block      *d_sb;          /*文件超级块*/
        unsigned int            d_flags;        /*目录项标识*/
        int                     d_mounted;      /*是登录点的目录项吗？ */
        void                    *d_fsdata;      /*文件系统特殊的数据*/
        struct rcu_head         d_rcu;          /*RCU锁*/
        struct dcookie_struct   *d_cookie;      /*cookie*/
        struct dentry           *d_parent;      /*父目录的目录项对象*/
        struct qstr             d_name;         /*目录项的名字*/
        struct hlist_node       d_hash;         /*散列表*/
        struct hlist_head       *d_bucket;      /*散列表头*/
        unsigned char           d_iname
                                [DNAME_INLINE_LEN_MIN]; /*短文件名*/
} ;
```

不同于前面的两个对象，目录项对象没有对应的磁盘数据结构，VFS根据字符串形式的路径名现场创建它。而且由于目录项对象并非真正保存在磁盘上，所以目录项结构体没有是否被修改的标志（也就是，是否为脏，是否需要写回磁盘的标志）。

12.7.1 目录项状态

目录项对象有三种有效状态：被使用、未被使用和负状态。

一个被使用的目录项对应一个有效的索引节点（即，d_inode指向相应的索引节点）并且表明该对象存在一个或多个使用者（即，d_count为正值）。一个目录项处于被使用状态，意味着它正被VFS使用并且指向有效的索引节点，因此不能被丢弃。

一个未被使用的目录项对应一个有效的索引节点（d_inode指向一个索引节点），但是应指明VFS当前并未使用它（d_count为0）。该目录项对象仍然指向一个有效对象，而且被保留在缓存中以便需要时再使用它。由于该目录项不会过早地被销毁，所以在以后再需要用到它时，不必重新创建，从而使路径查找更迅速。但如果要回收内存的话，可以销毁未使用的目录项。

一个负状态的目录项[⊖]没有对应的有效索引节点（d_inode为NULL），因为索引节点已被删除了，或路径不再正确了，但是目录项仍然保留，以便快速解析以后的路径查询。虽然负状态的目录项有些用处，但是如果需要的话，可以销毁它，因为毕竟在实际中很少用到它。目录项对象释放后也可以保存到slab对象缓存中去，这点在前一章讨论过。此时，任何VFS或文件系统代码都没有指向该目录项对象的有效引用。

12.7.2 目录项缓存

如果VFS层遍历路径名中所有的元素并将它们逐个地解析成目录项对象，这将是一件非常费力的工作，会浪费大量的时间。所以内核将目录项对象缓存在目录项缓存（简称dcache）中。

目录项缓存包括三个主要部分：

- “被使用的”目录项链表。该链表通过索引节点对象中的i_dentry项连接相关的索引节点，因为一个给定的索引节点可能有多个链接，所以就可能有多个目录项对象，因此用一个链表来连接它们。
- “最近被使用的”双向链表。该链表含有未被使用的和负状态的目录项对象。由于该链以时间顺序插入，所以链头的节点是最新数据。当内核必须通过删除节点项回收内存时，会从链尾删除节点项，因为尾部的节点最旧，也许它们在近期内再次被使用的可能性最小。
- 散列表和相应的散列函数用来快速地将给定路径解析为相关目录项对象。

散列表由数组dentry_hashtable表示，其中每一个元素都是一个指向具有相同键值的目录项对象链表的指针。数组的大小取决于系统中物理内存的大小。

实际的散列值由d_hash()函数计算，它是内核提供给文件系统的惟一的一个散列函数。

查找散列表要通过d_lookup()函数，如果该函数在dcache中发现了与其相匹配的目录项对象，则匹配的对象被返回；否则，返回NULL指针。

举例说明，假设你需要在自己目录中编译一个源文件，/home/dracula/src/the_sun_sucks.c，每一次对foo.c文件进行访问（比如说首先要打开它，然后要存储它，还要进行编辑等等），VFS都必须沿着嵌套的目录依次解析全部路径:/、home、draclula、src和最终的the_sun_sucks.c。为了避免每次访问该路径名都进行这种耗时的操作，VFS会先在目录项缓存中搜索路径名，如果找到了，就无需花费那么大的气力了。相反，如果该目录项在目录项缓存中并不存在，VFS就必须自己通过走查文件系统为每个路径分量解析路径，解析完毕后，再将目录项对象加入dcache中，以便以后可以快速查找到它。

⊖ 这个名字容易产生误导，其实它并没有和任何负数或负状态有联系。更准确的名称应该是无效目录项。

而dcache在一定意义上也提供对索引节点的缓存。和目录项对象相关的索引节点对象不会被释放，因为目录项会让相关索引节点的使用计数为正，这样就可以确保索引节点留在内存中。只要目录项被缓存，其相应的索引节点也就被缓存了。所以像前面的例子，只要路径名在缓存中找到了，那么相应的索引节点肯定也在内存中缓存着。

12.7.3 目录项操作

dentry_operation结构体指明了VFS操作目录项的所有方法。

该结构定义在文件<linux/dcache.h>中。

```
struct dentry_operations {
        int (*d_revalidate)(struct dentry *, int);
        int (*d_hash) (struct dentry *, struct qstr *);
        int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
        int (*d_delete)(struct dentry *);
        void (*d_release)(struct dentry *);
        void (*d_iput)(struct dentry *, struct inode *);
};
```

下面给出函数的具体用法：

- `int d_revalidate(struct dentry *dentry ,`
  `int flags)`

该函数判断目录对象是否有效。VFS准备从dcache中使用一个目录项时，会调用该函数。大部分文件系统将该方法置NULL，因为它们认为dcache中的目录项对象总是有效的。

- `int d_hash(struct dentry *dentry,`
  `struct qstr *name)`

该函数为目录项生成散列值，当目录项需要加入到散列表中时，VFS调用该函数。

- `int d_compare(struct dentry *dentry,`
  `struct qstr *name1,`
  `struct qstr *name2)`

VFS调用该函数来比较name1 和 name2这两个文件名。多数文件系统使用VFS默认的操作，仅仅作字符串比较。对有些文件系统，比如FAT，简单的字符串比较不能满足其需要。因为FAT文件系统不区分大小写，所以需要实现一种不区分大小写的字符串比较函数。注意使用该函数时需要加dcache_lock锁。

- `int d_delete(struct dentry *dentry)`

当目录项对象的d_count计数值等于零时，VFS调用该函数。注意使用该函数需要加dcache_lock锁。

- `void d_release(struct dentry *dentry)`

当目录项对象将要被释放时，VFS调用该函数，默认情况下，它什么也不做。

- `void d_iput(struct dentry *dentry,`
  `struct inode *inode)`

当一个目录项对象丢失了其相关的索引节点时（也就是说磁盘索引节点被删除了），VFS调用该函数。默认情况下VFS会调用iput()函数释放索引节点。如果文件系统重载了该函数，那么除了

执行此文件系统特殊的工作外，还必须调用iput()函数。

12.8 文件对象

VFS的最后一个主要对象是文件对象。文件对象表示进程已打开的文件。如果我们站在用户空间来看待VFS，文件对象会首先进入我们的视野。进程直接处理的是文件，而不是超级块、索引节点或目录项。所以不必奇怪：文件对象包含我们非常熟悉的信息（如访问模式、当前偏移等），同样道理，文件操作和我们非常熟悉的系统调用read()和write()等也很类似。

文件对象是已打开的文件在内存中的表示。该对象（不是物理文件）由相应的open()系统调用创建，由close()系统调用销毁，所有这些文件相关的调用实际上都是文件操作表中定义的方法。因为多个进程可以同时打开和操作同一个文件，所以同一个文件也可能存在多个对应的文件对象。文件对象仅仅在进程观点上代表已打开文件，它反过来指向目录项对象（反过来指向索引节点），其实只有目录项对象才表示已打开的实际文件。虽然一个文件对应的文件对象不是惟一的，但对应的索引节点和目录项对象无疑是惟一的。

文件对象由file结构体表示，定义在文件<linux/fs.h>中，下面给出该结构体和各项的描述。

```
struct file {
        struct list_head        f_list;         /*文件对象链表*/
        struct dentry           *f_dentry;      /*相关目录项对象*/
        struct vfsmount         *f_vfsmnt;      /*相关的安装文件系统*/
        struct file_operations  *f_op;          /*文件操作表*/
        atomic_t                f_count;        /*文件对象的使用计数*/
        unsigned int            f_flags;        /*当打开文件时所指定的标志*/
        mode_t                  f_mode;         /*文件的访问模式*/
        loff_t                  f_pos;          /*文件当前的位移量（文指针）*/
        struct fown_struct      f_owner;        /*通过信号进行异步I/O数据的传送*/
        unsigned int            f_uid;          /*用户的UID*/
        unsigned int            f_gid;          /*用户的GID*/
        int                     f_error;        /*错误码*/
        struct file_ra_state    f_ra;           /*预读状态*/
        unsigned long           f_version;      /*版本号*/
        void                    *f_security;    /*安全模块*/
        void                    *private_data;  /*tty设备hook*/
        struct list_head        f_ep_links;     /*事件池链表*/
        spinlock_t              f_ep_lock;      /*事件池锁*/
        struct address_space    *f_mapping;     /*页缓存映射 */
};
```

类似于目录项对象，文件对象实际上没有对应的磁盘数据。所以在结构体中没有代表其对象是否为脏，是否需要写回磁盘的标志。文件对象通过f_dentry指针指向相关的目录项对象。目录项会指向相关的索引节点，索引节点会记录文件是否是脏的。

文件操作

和VFS的其他对象一样，文件操作表在文件对象中也非常重要。跟file结构体相关的操作与系统调用很类似，这些操作是标准Unix 系统调用的基础。

文件对象的操作由file_operations结构体表示，定义在文件<linux/fs.h>中。

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, char *, size_t, loff_t);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        ssize_t (*aio_write) (struct kiocb *, const char *, size_t, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int);
        int (*aio_fsync) (struct kiocb *, int);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *,
                          unsigned long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *,
                           unsigned long, loff_t *);
        ssize_t (*sendfile) (struct file *, loff_t *, size_t,
                             read_actor_t, void *);
        ssize_t (*sendpage) (struct file *, struct page *, int,
                             size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                                           unsigned long, unsigned long,
                                           unsigned long);
        int (*check_flags) (int flags);
        int (*dir_notify) (struct file *filp, unsigned long arg);
        int (*flock) (struct file *filp, int cmd, struct file_lock *fl);
};
```

具体的文件系统可以为每一种操作做专门的实现，如果存在通用操作，也可以使用通用操作。一般在基于Unix的文件系统上，这些通用操作效果都不错。并不要求实际文件系统实现文件操作函数表中的所有方法——虽然不实现最基础的那些操作显然是很不明智的——对不感兴趣的操作完全可以简单地将该函数指针置为NULL。

下面给出操作的用法说明。

- `loff_t llseek(struct file *file,`
  `loff_t offset ,int origin)`

该函数用于更新偏移量指针。由系统调用llseek()调用它。

- `ssize_t read(struct file *file,`
  `char *buf,size_t count,`
  `loff_t *offset)`

该函数从给定文件的offset偏移处读取conut字节的数据到buf中，同时更新文件指针。由系统

调用read()调用它。

```
• ssize_t aio_read(struct kiocb *iocb,
                   char *buf, size_t count,
                   loff_t offset)
```

该函数从iocb描述的文件里，以同步方式读取count字节的数据到buf中。由系统调用aio_read()调用它。

```
• ssize_t write(struct file *file,
                const char *buf,size_t count,
                loff_t *offset)
```

该函数从给定的buf中取出count字节的数据，写入给定文件的offset偏移处，同时更新文件指针。由系统调用write()调用它。

```
• ssize_t aio_write(struct kiocb *iocb,
                    const char *buf,
                    size_t count, loff_t offset)
```

该函数以同步方式从给定的buf中取出count字节的数据，写入由iocb描述的文件中。由系统调用aio_write()调用它。

```
• int readdir(struct file *file ,void *dirent,
              filldir_t filldir)
```

该函数返回目录列表中的下一个目录。由系统调用readdir()调用它。

```
• unsigned int poll(struct file *file,
     struct poll_table_struct *poll_table)
```

该函数睡眠等待给定文件的活动。由系统调用poll()调用它。

```
• int ioctl(struct inode *inode,
            struct file *file,
            unsigned int cmd,
            unsigned long arg)
```

该函数用来给设备发送命令参数对。当文件是一个被打开的设备节点时，可以通过它进行设置操作。由系统调用ioctl()调用它。

```
• int mmap(struct file *file,
           struct vm_area_struct *vma)
```

该函数将给定的文件映射到指定的地址空间上。由系统调用mmap()调用它。

```
• int open(struct inode *inode,
           struct file *file)
```

该函数创建一个新的文件对象，并将它和相应的索引节点对象关联起来。由系统调用open()调用它。

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

当已打开文件的引用计数减少时，该函数被VFS调用。它的作用根据具体文件系统而定。

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

当文件的最后一个引用被注销时——比如，当最后一个共享文件描述符的进程调用了close()或

退出时，该函数会被VFS调用。它的作用根据具体文件系统而定。

```
• int fsync(struct file *file,
            struct dentry *dentry,
            int datasync)
```

将给定文件的所有被缓存的数据写回磁盘。该函数由系统调用fsync()调用。

```
• int aio_fsync(struct kiocb *iocb,
                int datasync)
```

将iocb描述的文件的所有被缓存数据写回到磁盘。该函数由系统调用aio_fsync()调用。

```
• int fasync(int fd,struct file *file ,int on)
```

该函数用于打开或关闭异步I/O的通告信号。

```
• int lock (struct file *file,int cmd,
          struct file_lock *lock)
```

该函数用于给指定文件上锁。

```
• ssize_t readv(struct file *file,
                const struct iovec *vector,
                unsigned long count,
                loff_t *offset)
```

该函数从给定文件中读取数据，并将其写入由vector描述的count个缓冲中去，同时增加文件的偏移量。由系统调用readv()调用它。

```
• ssize_t writev(struct file *file,
                 const struct iovec *vector,
                 unsigned long count,
                 loff_t *offset)
```

该函数将由vector描述的count个缓冲中的数据写入到由file指定的文件中去，同时减小文件的偏移量。由系统调用writev()调用它。

```
• ssize_t sendfile(struct file *file,
                   loff_t *offset,
                   size_t size,
                   read_actor_t actor,
                   void *target)
```

该函数用于从一个文件拷贝数据到另一个文件中，它执行的拷贝操作完全在内核中完成，避免了向用户空间进行不必要的拷贝。由系统调用sendfile()调用它。

```
• ssize_t sendpage(struct file *file,
                   struct page *page,
                   int offset,size_t size,
                   loff_t *pos, int more)
```

该函数用来从一个文件向另一个文件发送数据。

```
• unsigned long get_unmapped_area(struct file
              *file,
              unsigned long addr,
              unsigned long len,
```

```
                unsigned long offset,
                unsigned long flags)
```

该函数用于获取未使用的地址空间来映射给定的文件。

- `int check_flags(int flags)`

当给出SETFL命令时，这个函数用来检查传递给fcntl()系统调用的标志的有效性。与大多数VFS操作一样，文件系统不必实现check_flags()，目前，只有在NFS文件系统上实现了。这个函数能使文件系统限制无效的SETFL标志，不进行限制的话，普通的fcntl()函数能使标志生效。在NFS文件系统中，不允许把O_APPEND和O_DIRECT相结合。

```
int flock(struct file *filp,
          int cmd,
          struct file_lock *fl)
```

这个函数用来实现flock()系统调用，该调用提供忠告锁。

12.9 和文件系统相关的数据结构

除了以上几种VFS基础对象外，内核还使用了另外一些标准数据结构来管理文件系统的其他相关数据。第一个结构体是file_system_type，用来描述各种特定文件系统类型，比如ext3或XFS。第二个结构体是vfsmount，用来描述一个安装文件系统的实例。

因为Linux支持众多不同的文件系统，所以内核必须由一个特殊的结构来描述每种文件系统的功能和行为。

```
struct file_system_type {
        const char              *name;          /*文件系统的名字*/
        struct subsystem        subsys;         /*sysfs子系统对象*/
        int                     fs_flags;       /*文件系统类型标志*/

        /*下面的函数用来从磁盘中读取超级块*/
        struct super_block      *(*get_sb) (struct file_system_type *, int,
                                            char *, void *);

        /*下面的函数用来终止访问超级块*/
        void                    (*kill_sb) (struct super_block *);

        struct module           *owner;         /* 文件系统模块*/
        struct file_system_type *next;          /*链表中下一个文件系统类型*/
        struct list_head        fs_supers;      /*超级块对象链表*/
};
```

get_sb()函数从磁盘上读取超级块，并且在文件系统被安装时，在内存中组装超级块对象。剩余的函数描述文件系统的属性。

每种文件系统，不管有多少个实例安装到系统中，还是根本就没有安装到系统中，都只有一个file_system_type结构。

更有趣的事情是，当文件系统被实际安装时，将有一个vfsmount结构体在安装点被创建。该结构体用来代表文件系统的实例——换句话说，代表一个安装点。

vfsmount结构被定义在<linux/mount.h>中，下面是具体结构。

```
struct vfsmount {
        struct list_head        mnt_hash;           /*散列表*/
        struct vfsmount         *mnt_parent;        /*父文件系统*/
        struct dentry           *mnt_mountpoint;    /*安装点的目录项对象*/
        struct dentry           *mnt_root;          /*该文件系统的根目录项对象*/
        struct super_block      *mnt_sb;            /*该文件系统的超级块*/
        struct list_head        mnt_mounts;         /*子文件系统链表*/
        struct list_head        mnt_child;          /*子文件系统链表*/
        atomic_t                mnt_count;          /*使用计数*/
        int                     mnt_flags;          /*安装标志*/
        char                    *mnt_devname;       /*设备文件名*/
        struct list_head        mnt_list;           /*描述符链表*/
        struct list_head        mnt_fslink;         /* 具体文件系统的到期列表 */
        struct namespace        *mnt_namespace      /* 相关的名字空间 */
};
```

理清文件系统和所有其他安装点间的关系，是维护所有安装点链表中最复杂的工作。所以vfsmount结构体中维护的各种链表就是为了能够跟踪这些关系信息。

vfsmount结构还保存了在安装时指定的标志信息，该信息存储在mnt_flags域中。表12-1列出了标准的安装标志。

表12-1 标准安装标志列表

标 志	描 述
MNT_NOSUID	禁止该文件系统的可执行文件设置setuid和setgid标志
MNT_NODEV	禁止访问该文件系统上的设备文件
MNT_NOEXEC	禁止执行该文件系统上的可执行文件

安装那些管理员不充分信任的移动设备时，这些标志很有用处。它们定义在<linux/mount.h>中。

12.10 和进程相关的数据结构

系统中的每一个进程都有自己的一组打开的文件，像根文件系统、当前工作目录、安装点等等。有三个数据结构将VFS层和系统的进程紧密联系在一起，它们分别是：files_struct 、fs_struct和namespace结构体。

files_struct结构体定义在文件<linux/file.h>中。该结构体由进程描述符中的files域指向。所有与每个进程(per-process)相关的信息如打开的文件及文件描述符都包含在其中，其结构和描述如下：

```
struct files_struct {
        atomic_t            count;              /*结构的使用计数*/
        spinlock_t          file_lock;          /*保护该结构体的锁*/
        int                 max_fds;            /*文件对象数的上限*/
        int                 max_fdset;          /*文件描述符的上限*/
        int                 next_fd;            /*下一个文件描述符*/
        struct file         **fd;               /*全部文件对象数组*/
        fd_set              *close_on_exec;     /*exec()关闭的文件描述符*/
        fd_set              *open_fds;          /*指向打开文件的描述符*/
        fd_set              close_on_exec_init; /*exec()关闭的初始文件*/
```

```
        fd_set              open_fds_init;              /*文件描述符的初始集合*/
        struct file         *fd_array[NR_OPEN_DEFAULT]; /*默认的文件对象数组*/
};
```

fd数组指针指向已打开的文件对象链表，默认情况下，指向fd_array 数组。因为NR_OPEN_DEFAULT等于32，所以该数组可以容纳32个文件对象。如果一个进程所打开的文件对象超过32个。内核将分配一个新数组，并且将fd指针指向它。所以对适当数量的文件对象的访问会执行得很快，因为它是对静态数组进行的操作；如果一个进程打开的文件数量过多，那么内核就需要建立新数组。所以如果系统中有大量的进程都要打开超过32个文件，为了优化性能，管理员可以适当增大NR_OPEN_DEFAULT的预定义值。

和进程相关的第二个结构体是fs_struct。该结构由进程描述符的fs域指向。它包含文件系统和进程相关的信息，定义在文件<linux/fs_struct.h>中，下面是它的具体结构体和各项描述。

```
struct fs_struct {
        atomic_t            count;          /*结构的使用计数*/
        rwlock_t            lock;           /*保护该结构体的锁*/
        int                 umask;          /*默认的文件访问权限*/
        struct dentry       *root;          /*根目录的目录项对象*/
        struct dentry       *pwd;           /*当前工作目录的目录项对象*/
        struct dentry       *altroot;       /*可供选择的根目录的目录项对象*/
        struct vfsmount     *rootmnt;       /*根目录的安装点对象*/
        struct vfsmount     *pwdmnt;        /*pwd的安装点对象*/
        struct vfsmount     *altrootmnt;    /*可供选择的根目录的安装点对象*/
};
```

该结构包含了当前进程的当前工作目录（pwd）和根目录。

第三个也是最后一个相关结构体是namespace 。它定义在文件<linux/namespace.h>中，由进程描述符中的namespace域指向。2.4版内核以后，单进程命名空间被加入到内核中，它使得每一个进程在系统中都看到惟一的安装文件系统——不仅是惟一的根目录，而且是惟一的文件系统层次结构。下面是其具体结构和描述：

```
struct namespace {
        atomic_t            count;      /*结构的使用计数*/
        struct vfsmount     *root;      /*根目录的安装点对象*/
        struct list_head    list;       /*安装点链表*/
        struct rw_semaphore sem;        /*保护命名空间的信号量*/
};
```

list域是连接已安装文件系统的双向链表，它包含的元素组成了全体命名空间。

上述这些数据结构都是通过进程描述符连接起来的。对多数进程来说，它们的描述符都指向惟一的files_struct 和fs_struct结构体。但是，对于那些使用克隆标志CLONE_FILES或CLONE_FS创建的进程，会共享[⊖]这两个结构体。所以多个进程描述符可能指向同一个files_struct或fs_struct结构体。每个结构体都维护一个count域作为引用记数，它防止在进程正使用该结构时，该结构被销毁。

namespace结构体的使用方法却和前两种结构体完全不同，默认情况下，所有的进程共享同样

⊖ 线程通常在创建时使用CLONE_FILES和CLONE_FS标志，所以多个线程共享一个file_struct结构体和fs_struct结构体。但另一方面，普通进程没有指定这些标志，所以它们有自己的文件系统信息和打开文件表。

的命名空间（也就是，它们都从相同的挂载表中看到同一个文件系统层次结构）。只有在进行clone()操作时使用CLONE_NEWNS标志，才会给进程一个另外的命名空间结构体的拷贝。因为大多数进程不提供这个标志，所有进程都继承其父进程的名字空间。因此，在大多数系统上只有一个名字空间，不过，CLONE_NEWNS标志可以使这一功能失效。

12.11 Linux中的文件系统

Linux支持了相当多种类的文件系统。从本地文件系统，如ext2和ext3，到网络文件系统，如NFS和Coda,Linux在标准内核中已支持的文件系统超过50种。VFS层提供给这些不同文件系统一个统一的实现框架，而且还提供了能和标准系统调用交互工作的统一接口。由于VFS层的存在，使得在Linux上实现新文件系统的工作变得简单起来，它可以轻松地使这些文件系统通过标准Unix系统调用而协同工作。

本章描述了VFS的目的，讨论了各种数据结构，包括最重要的索引节点、目录项以及超级块对象。下一章将讨论数据如何从物理上存放在文件系统中。

第⑬章

块I/O层

系统中能够随机（不需要按顺序）访问固定大小数据片（chunk）的设备被称作块设备，这些数据片就称作块。最常见的块设备是硬盘，除此以外，还有软盘驱动器、CD-ROM驱动器和闪存等许多其他块设备。注意，它们都是以安装文件系统的方式使用的——这也是块设备通常的访问方式。

另一种基本的设备类型是字符设备。字符设备按照字符流的方式被有序访问，像串口和键盘就都属于字符设备。如果一个硬件设备是以字符流的方式被访问的话，那就应该将它归于字符设备；反过来，如果一个设备是随机（无序的）访问的，那么它就属于块设备。

这两种类型的设备的根本区别在于它们是否可以被随机访问——换句话说，就是能否在访问设备时随意地从一个位置跳转到另一个位置。举个例子，键盘这种设备提供的就是一个数据流，当你敲入“fox”这个字符串时，键盘驱动程序会按照和输入完全相同的顺序返回这个由三个字符组成的数据流。如果让键盘驱动程序打乱顺序来读字符串，或读取其他字符，都是没有意义的。所以键盘就是一种典型的字符设备，它提供的就是用户从键盘输入的字符流。对键盘进行读操作会得到一个字符流，首先是“f”，然后是“o”，最后是“x”，最终是文件的结束(EOF)。当没人敲键盘时，字符流就是空的。硬盘设备的情况就不大一样了。硬盘设备的驱动可能要求读取磁盘上任意块的内容，然后又转去读取别的块的内容，而被读取的块在磁盘上位置不一定要连续，所以说硬盘可以被随机访问，而不是以流的方式被访问，显然它是一个块设备。

内核管理块设备要比管理字符设备细致得多，需要考虑的问题和完成的工作相比字符设备来说要复杂许多。这是因为字符设备仅仅需要控制一个位置——当前位置——而块设备访问的位置必须能够在介质的不同区间前后移动。所以事实上内核不必提供一个专门的子系统来管理字符设备，但是对块设备的管理却必须要有一个专门的提供服务的子系统。不仅仅是因为块设备的复杂性远远高于字符设备，更重要的原因是块设备对执行性能的要求很高；对硬盘每多一分利用都会对整个系统的性能带来提升，其效果要远远比键盘吞吐速度成倍的提高大得多。另外，我们将会看到，块设备的复杂性会为这种优化留下很大的施展空间。这一章的主题就是讨论如何管理块设备和如何管理对块设备的请求。该部分在内核中被称作块I/O层。有趣的是，改写块I/O层正是2.5开发版内核的主要目标。本章涵盖了2.6内核中所有新的块I/O层。

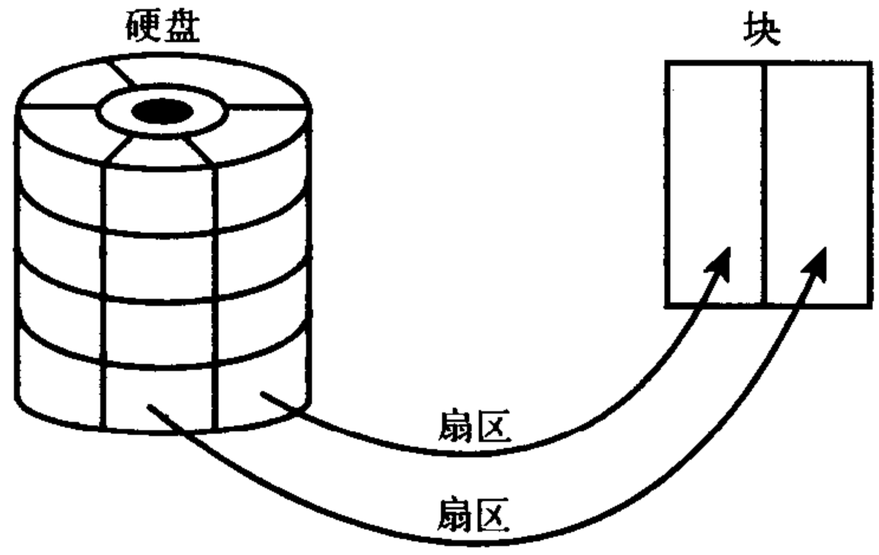
13.1 解剖一个块设备

块设备中最小的可寻址单元是扇区。扇区大小一般是2的整数倍，而最常见的大小是512个字节。扇区的大小是设备的物理属性，扇区是所有块设备的基本单元——块设备无法对比它还小的单元进行寻址和操作，不过许多块设备能够一次就传输多个扇区。虽然大多数块设备的扇区大小

都是512字节，不过其他大小的扇区也很常见（比如，很多CD-ROM盘的扇区都是2K大小）。

虽然各种软件的用途不同，但是它们都会用到自己的最小逻辑可寻址单元——块。块是文件系统的一种抽象——只能基于块来访问文件系统。虽然物理磁盘寻址是按照扇区级进行的，但是内核执行的所有磁盘操作都是按照块进行的。由于扇区是设备的最小可寻址单元，所以块不能比扇区还小，只能数倍于扇区大小。另外内核（对有扇区的硬件设备）还要求块大小是2的整数倍，而且不能超过一个页的长度（请看第11章和第19章）[⊖]。所以，对块大小的最终要求是，必须是扇区大小的2的整数倍，并且要小于页面大小。所以通常块大小是512字节、1K或4K。

扇区和块还有一些不同的叫法，为了不引起混淆，我们在这里简要介绍一下它们的其他名称。扇区——设备的最小寻址单元，有时会被称作“硬扇区”或“设备块”；同样地，块——文件系统的最小寻址单元，有时会被称作“文件块”或“I/O块”。在这一章里，会一直使用“扇区”和“块”这两个术语，但你还是应该记住它们的这些别名。图13-1 是扇区和缓冲区之间的关系图。



从扇区映射到块

图13-1 扇区和缓冲区之间的关系

和硬盘相关的其他常见术语还有——簇、柱面、磁头等。这些术语都和具体的块设备相关，一般情况下，用户空间的软件用不到这些概念。扇区对内核的重要性在于所有设备的I/O操作都必须基于扇区来进行；反过来，块是内核使用的较高层概念，它是比扇区高一层的抽象。

13.2 缓冲区和缓冲区头

当一个块被调入内存时（也就是说，在读入后或等待写出时），它要存储在一个缓冲区中。每个缓冲区与一个块对应，它相当于是磁盘块在内存中的表示。前面提到过，块包含一个或多个扇区，但大小不能超过一个页面，所以一个页可以容纳一个或多个内存中的块。由于内核在处理数据时需要一些相关的控制信息（比如块属于那一个块设备，块对应于哪个缓冲区等），所以每一个缓冲区都有一个对应的描述符。该描述符用buffer_head结构体表示，被称作缓冲区头，在文件<linux/buffer_head.h>中定义，它包含了内核操作缓冲区所需要的全部信息。

下面给出缓冲区头结构体和其中各个域的说明：

```
struct buffer_head {
        unsigned long           b_state;        /*缓冲区状态标志*/
        atomic_t                b_count;        /*缓冲区使用计数*/
        struct buffer_head      *b_this_page;   /*页面中的缓冲区*/
        struct page             *b_page;        /*存储缓冲区的页面*/
        sector_t                 b_blocknr;     /*逻辑块号*/
        u32                      b_size;        /*块大小（以字节为单位）*/
        char                    *b_data;        /*页面中的缓冲区*/
```

⊖ 这个认为的限制可能会遗留到以后，但是强制块的大小等于或小于页大小无疑简化了内核。

```
        struct block_device     *b_bdev;            /*块设备*/
        bh_end_io_t             *b_end_io;          /*I/O完成方法*/
        void                    *b_private;         /*完成方法数据*/
        struct list_head         b_assoc_buffers;   /*相关的映射链表*/
};
```

b_state域表示缓冲区的状态，可以是表13-1中一种标志或多种标志的组合。合法的标志存放在bh_state_bits枚举中，该枚举在<linux/buffer_head.h>中定义。

表13-1 bh_state 标志

状态标志	意 义
BH_Uptodate	该缓冲区包含可用数据
BH_Dirty	该缓冲区是脏的（缓存中的内容比磁盘中的块内容新，所以缓冲区内容必须被写回磁盘）
BH_Lock	该缓冲区正在被I/O操作使用，被锁定以防被并发访问
BH_Req	该缓冲区有I/O请求操作
BH_Mapped	该缓冲区是映射磁盘块的可用缓冲区
BH_New	缓冲区是通过get_block()刚刚映射的，尚且不能访问
BH_Async_Read	该缓冲区正通过end_buffer_async_read()被异步I/O读操作使用
BH_Async_Write	该缓冲区正通过end_buffer_async_write()被异步I/O写操作使用
BH_Delay	该缓冲区尚未和磁盘块关联
BH_Boundary	该缓冲区处于连续块区的边界——下一个块不再连续

bh_state_bits列表还包含了一个特殊标志——BH_PrivateStart，该标志不是可用状态标志，使用它是为了指明可被其他代码使用的起始位。块I/O层不会使用BH_PrivateStart或更高的位。那么某个驱动程序希望通过b_state域存储信息时就可以安全地使用这些位。驱动程序可以在这些位中定义自己的状态标志，只要保证自定义的状态标志不与块I/O层的专用位发生冲突就可以了。

b_count域表示缓冲区的使用记数，可通过两个定义在文件<linux/buffer_head.h>中的内联函数对此域进行增减。

```
static inline void get_bh(struct buffer_head *bh)
{
        atomic_inc(&bh->b_count);
}

static inline void put_bh(struct buffer_head *bh)
{
        atomic_dec(&bh->b_count);
}
```

在操作缓冲区头之前，应该先使用get_bh()函数增加缓冲区头的引用计数，确保该缓冲区头不会再被分配出去；当完成对缓冲区头的操作之后，还必须使用put_bh()函数减少引用计数。

与缓冲区对应的磁盘物理块由b_blocknr域索引，该值是b_bdev域指明的块设备中的逻辑块号。

与缓冲区对应的内存物理页由b_page域表示，另外，b_data域直接指向相应的块（它位于b_page域所指明的页面中的某个位置上），块的大小由b_size域表示，所以块在内存中的起始位置在b_data处，结束位置在(b_data + b_size)处。

缓冲区头的目的在于描述磁盘块和物理内存缓冲区（在特定页面上的字节序列）之间的映射

关系。这个结构体在内核中只扮演一个描述符的角色，说明从缓冲区到块的映射关系。

在2.6内核以前，缓冲区头的作用比现在还要重要。因为缓冲区头作为内核中的I/O操作单元，不仅仅描述了从磁盘块到物理内存的映射，而且还是所有块I/O操作的容器。可是，将缓冲区头作为I/O操作单元带来了两个弊端。首先，缓冲区头是一个很大且不易控制的数据结构体（现在是缩减过的了），而且缓冲区头对数据的操作既不方便也不清晰。对内核来说，它更倾向于操作页面结构，因为页面操作起来更为简便，同时效率也高。使用一个巨大的缓冲区头表示每一个独立的缓冲区（可能比页面小）效率低下，所以在2.6版本中，许多I/O操作都是通过内核直接对页面或地址空间进行操作来完成，不再使用缓冲区头了。这其中所做的一些工作会在第15章“页高速缓存和页回写”中进行讨论，具体情况请参考address_space结构和pdflush等守护进程（daemon）部分。

缓冲区头带来的第二个弊端是：它仅能描述单个缓冲区，当作为所有I/O的容器使用时，缓冲区头会迫使内核打断对大块数据的I/O操作（比如写操作），使其成为对多个buffer_head结构体进行操作。这样做必然会造成不必要的负但和空间浪费。所以2.5开发版内核的主要目标就是为块I/O操作引入一种新型、灵活并且轻量级的容器，也就是下一节要介绍的bio结构体。

13.3 bio结构体

目前内核中块I/O操作的基本容器由bio结构体表示，它定义在文件<linux/bio.h>中。该结构体代表了正在现场的（活动的）以片断（segment）链表形式组织的块I/O操作。一个片段是一小块连续的内存缓冲区。这样的话，就不需要保证单个缓冲区一定要连续。所以通过用片段来描述缓冲区，即使一个缓冲区分散在内存的多个位置上，bio结构体也能对内核保证I/O操作的执行。像这样的向量I/O就是所谓的聚散I/O。

bio结构体定义于<linux/bio.h>中，下面给出bio结构体和各个域的描述。

```
struct bio {
        sector_t                bi_sector;          /*磁盘上相关的扇区*/
        struct bio              *bi_next;           /*请求链表*/
        struct block_device     *bi_bdev;           /*相关的块设备*/
        unsigned long           bi_flags;           /*状态和命令标志*/
        unsigned long           bi_rw;              /*读还是写？ */
        unsigned short          bi_vcnt;            /*bio_vecs偏移的个数 */
        unsigned short          bi_idx;             /*bi_io_vec的当前索引*/
        unsigned short          bi_phys_segments;   /*结合后的片断数目*/
        unsigned short          bi_hw_segments;     /*重映射后的片断数目*/
        unsigned int            bi_size;            /*I/O计数*/
        unsigned int            bi_hw_front_size;   /* 第一个可合并的段大小*/
        unsigned int            bi_hw_back_size;    /* 最后一个可合并的段大小*/
        unsigned int            bi_max_vecs;        /*bio_vecs数目上限*/
        struct bio_vec          *bi_io_vec;         /*bio_vec链表*/
        bio_end_io_t            *bi_end_io;         /*I/O完成方法*/
        atomic_t                bi_cnt;             /*使用计数*/
        void                    *bi_private;        /*拥有者的私有方法*/
        bio_destructor_t        *bi_destructor;     /*销毁方法*/
};
```

使用bio结构体的目的主要是代表正在现场执行的I/O操作，所以该结构体中的主要域都是用来

管理相关信息的，其中最重要的几个域是bi_io_vecs、bi_vcnt和bi_idx。图13-2显示了bio结构及其朋友之间的关系。

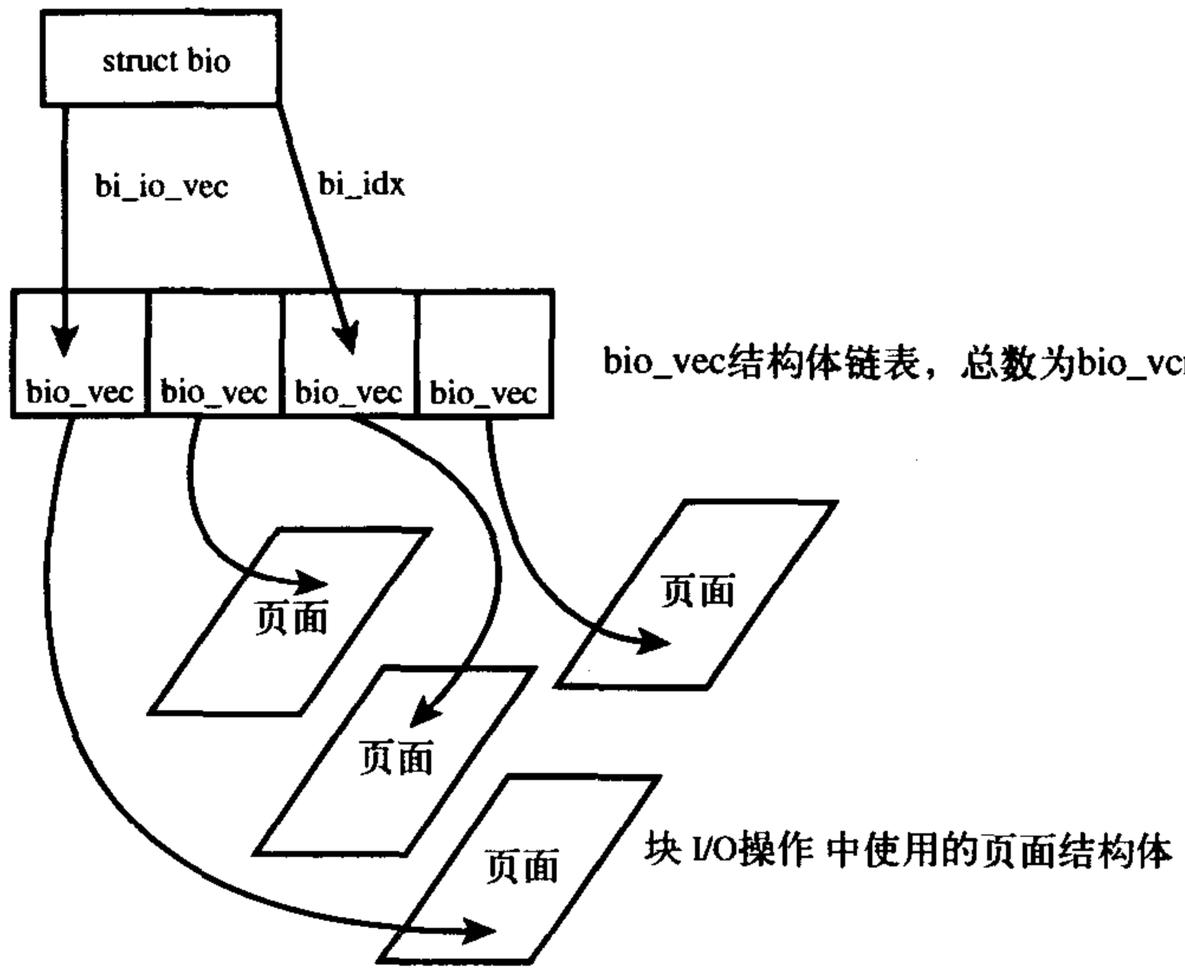


图13-2 bio结构体、bio_vec结构体和page结构体之间的关系

bi_io_vecs域指向一个bio_vec结构体数组，该结构体链表包含了一个特定I/O操作所需要使用到的所有片段。每个bio_vec结构都是一个形式为<page, offset, len>的向量，它描述的是一个特定的片段：片段所在的物理页、块在物理页中的偏移位置、从给定偏移量开始的块长度。整个bio_io_vec结构体数组表示了一个完整的缓冲区。bio_vec结构定义在<linux/bio.h>文件中：

```
struct bio_vec{
        /* 指向这个缓冲区所驻留的物理页*/
        struct page       *bv_page;

        /* 这个缓冲区以字节为单位的大小*/
        unsigned int      bv_len;

        /* 缓冲区所驻留的页中以字节为单位的偏移量*/
        unsigned int      bv_offset;
};
```

在每个给定的块I/O操作中，bi_vcnt域用来描述bi_io_vec所指向的bio_vec数组中的向量数目。当块I/O操作执行完毕后，bi_idx域指向数组的当前索引。

总而言之，每一个块I/O请求都通过一个 bio结构体表示。每个请求包含一个或多个块，这些块存储在bio_vec结构体数组中。这些结构体描述了每个片段在物理页中的实际位置，并且像向量一样被组织在一起。I/O操作的第一个片段由b_io_vec结构体所指向，其他的片段在其后依次放置，共有bi_vcnt个片段。当块I/O层开始执行请求、需要使用各个片段时，bi_idx域会不断更新，从而总指向当前片段。

bi_idx域指向数组中的当前bio_vec片段，块I/O层通过它可以跟踪块I/O操作的完成进度。但该域更重要的作用在于分割bio结构体。像RAID(冗余廉价磁盘阵列，出于提高性能和可靠性的目的，

将单个磁盘的卷扩展到多个磁盘上)这样的驱动器可以把单独的bio结构体——原本是为单个设备使用准备的——分割到RAID阵列中的各个硬盘上去。RAID设备驱动只需要拷贝这个bio结构体，再把bi_idx域设置为每个独立硬盘操作时需要的位置就可以了。

bi_cnt域记录bio结构体的使用计数，如果该域值减为0，就应该销毁该bio结构体，并释放它占用的内存。通过下面两个函数管理使用计数。

```
void bio_get(struct bio *bio)
void bio_put(struct bio *bio)
```

前者增加使用计数，后者减少使用计数（如果计数减到0，则销毁bio结构体）。在操作正在活动的bio结构体时，一定要首先增加它的使用计数，以免在操作过程中该bio结构体被释放；相反，在操作完毕后，要减少使用计数。

最后要说明的是bi_private域，这是一个属于拥有者（也就是创建者）的私有域，只有创建了bio结构的拥有者可以读写该域。

新老方法对比

缓冲区头和新的bio结构体之间存在明显差别。bio结构体代表的是I/O操作，它可以包括内存中的一个或多个页；而另一方面，buffer_head结构体代表的是一个缓冲区，它描述的仅仅是磁盘中的一个块。因为缓冲区头关联的是单独页中的单独磁盘块，所以它可能会引起不必要的分割，将请求按块为单位划分，只能靠以后才能再重新组合。由于bio结构是轻量级的，它描述的块可以不需要连续存储区，并且不需要分割I/O操作。

利用bio结构体代替buffer_head结构体还有以下好处：

- bio结构体很容易处理高端内存(参看第11章)，因为它处理的是物理页而不是直接指针。
- bio结构体既可以代表普通页I/O，同时也可以代表直接I/O（指那些不通过页高速缓存的I/O操作——请参考15章中对页高速缓存的讨论）
- bio结构体便于执行分散－集中（向量化的）块I/O操作，操作中的数据可取自多个物理页面。
- bio结构体相比缓冲区头属于轻量级的结构体。因为它只需要包含块I/O操作所需的信息就行了，不用包含与缓冲区本身相关的不必要信息。

但是还是需要缓冲区头这个概念，毕竟它还负责描述磁盘块到页面的映射。bio结构体不包含任何和缓冲区相关的状态信息——它仅仅是一个向量数组，描述一个或多个单独块I/O操作的数据片段和相关信息。在当前设置中，当bio结构体描述当前正在使用的I/O操作时， buffer_head结构体仍然需要包含缓冲区信息。内核通过这两种结构分别保存各自的信息，可以保证每种结构所含的信息量尽可能地少。

13.4 请求队列

块设备将它们挂起的块I/O请求保存在*请求队列*中，该队列由reques_queue结构体表示，定义在文件<linux/blkdev.h>中，包含一个双向请求链表以及相关控制信息。通过内核中像文件系统这样高层的代码将请求加入到队列中。请求队列只要不为空，队列对应的块设备驱动程序就会从队列头获取请求，然后将其送入对应的块设备上去。请求队列表中的每一项都是一个单独的请求，由request结构体表示。

请求

队列中的请求由结构体request表示，它定义在文件<linux/blkdev.h>中。因为一个请求可能要操作多个连续的磁盘块，所以每个请求可以由多个bio结构体组成。注意，虽然磁盘上的块必须连续，但是在内存中这些块并不一定要连续——每个bio结构体都可以描述多个片段（回忆一下，片段是内存中连续的小区域），而每个请求也可以包含多个bio结构体。

13.5 I/O调度程序

如果简单地以内核产生请求的次序直接将请求发向块设备的话，性能肯定让人难以接受。磁盘寻址是整个计算机中最慢的操作之一，每一次寻址——定位硬盘磁头到特定块上的某个位置——需要花费不少时间。所以尽量缩短寻址时间无疑是提高系统性能的关键。

为了优化寻址操作，内核既不会简单地按请求接收次序，也不会立即将其提交给磁盘。相反，它会在提交前，先执行名为合并与排序的预操作，这种预操作可以极大地提高系统的整体性能[⊖]。在内核中负责提交I/O请求的子系统被称为I/O调度程序。

I/O调度程序将磁盘I/O资源分配给系统中所有挂起的块I/O请求。具体的说，这种资源分配是通过将请求队列中挂起的请求合并和排序来完成的。注意不要将I/O调度程序和进程调度程序（请看第4章）混淆。进程调度程序的作用是将处理器资源分配给系统中的运行进程。这两种子系统看起来非常相似，但并不相同。进程调度程序和I/O调度程序都是将一个资源虚拟给多个对象，对进程调度程序来说，处理器被虚拟并被系统中的运行进程共享。这种虚拟提供给用户的就是多任务和分时操作系统，像Unix系统。相反，I/O调度程序虚拟块设备给多个磁盘请求，以便降低磁盘寻址时间，确保磁盘性能的最优化。

13.5.1 I/O调度程序的工作

I/O调度程序的工作是管理块设备的请求队列。它决定队列中的请求排列顺序以及在什么时刻派发请求到块设备。这样做有利于减少磁盘寻址时间，从而提高全局吞吐量。注意全局这个定语很重要，坦率地讲，一个I/O调度器可能为了提高系统整体性能，而对某些请求不公。

I/O调度程序通过两种方法减少磁盘寻址时间：合并与排序。合并指将两个或多个请求结合成一个新请求。考虑一下这种情况，文件系统提交请求到请求队列——从文件中读取一个数据区（当然，最终所有的操作都是针对扇区和块进行的，而不是文件，还假定请求的块都是来自文件块），如果这时队列中已经存在一个请求，它访问的磁盘扇区和当前请求访问的磁盘扇区相邻（比如，同一个文件中早些时候被读取的数据区），那么这两个请求就可以合并为一个对单个和多个相邻磁盘扇区操作的新请求。通过合并请求，I/O调度程序将多次请求的开销压缩成一次请求的开销。更重要的是，请求合并后只需要传递给磁盘一条寻址命令，就可以访问到请求合并前必须多次寻址才能访问完的磁盘区域了，因此合并请求显然能减少系统开销和磁盘寻址次数。

现在，假设在读请求被提交给请求队列的时候，队列中并没有其他请求需要操作相邻的扇区，此时就无法将当前请求与其他请求合并，当然，可以将其插入请求队列的尾部。但是如果有其他请

⊖ 这一点需要强调。如果一个系统没有这些功能，或者这些功能实现得很差劲，那么即使是数量不大的块I/O操作，执行性能也会很糟糕。

求需要操作磁盘上类似的位置呢？如果存在一个请求，它要操作的磁盘扇区位置与当前请求的比较接近，那么是不是该让这两个请求在请求队列上也相邻呢？事实上，I/O调度程序的确是这样处理上述情况的，整个请求队列将按扇区增长方向有序排列。使所有请求按硬盘上扇区的排列顺序有序排列（尽可能的）的目的不仅是为了缩短单独一次请求的寻址时间，更重要的优化在于，通过保持磁盘头以直线方向移动，缩短了所有请求的磁盘寻址时间。该排序算法类似于电梯调度——电梯不能随意的从一层跳到另一层，它应该向一个方向移动，当抵达了同一方向上的最后一层后，再掉头向另一个方向移动。出于这种相似性，所以I/O调度程序（或这种排序算法）被称作电梯调度。

13.5.2 Linus 电梯

下面看看Linux中实际使用的I/O调度程序。我们看到的第一个I/O调度程序被称为Linus电梯（没错，Linus确实是用他的名字命名了这个电梯！）。在2.4内核中，Linus电梯是默认的I/O调度程序。虽然后来在2.6内核中它被另外两种调度程序取替了，但是由于这个电梯比后来的调度程序简单，而且它们执行的许多功能都相似，所以仍然值得讨论一下。

Linus电梯能执行合并与排序预处理。当有新的请求加入队列时，它首先会检查其他每一个挂起的请求是否可以和新请求合并。Linus电梯I/O调度程序可以执行向前和向后合并，合并类型描述的是请求向前面还是向后面，这一点和已有请求相连。如果新请求正好连在一个现存的请求前，就是向前合并；相反如果新请求直接连在一个现存的请求后，就是向后合并。鉴于文件的分布（通常以扇区号的增长表现）特点和I/O操作执行方式具有典型性（一般都是从头读向尾，很少从反方向读），所以向前合并相比向后合并要少得多，但是Linus电梯还是会对两种合并类型都进行检查。

如果合并尝试失败，那么就需要寻找可能的插入点（新请求在队列中的位置必须符合请求以扇区方向有序排序的原则）。如果找到，新请求将被插入到该点；如果没有合适的位置，那么新请求就被加入到队列尾部。另外，如果发现队列中有驻留时间过长的请求，那么新请求也将被加入到队列尾部，即使插入后还要排序。这样做是为了避免由于访问相近磁盘位置的请求太多，从而造成访问磁盘其他位置的请求难以得到执行机会这一问题。不幸的是，这种“年龄”检测方法并不很有效，因为它并非是给等待了一段时间的请求提供实质性服务——它仅仅是在经过了一定时间后停止插入——排序请求，这改善了等待时间但最终还是会导致请求饥饿现象的发生，所以这是一个2.4内核I/O调度程序中必须要修改的缺陷。

总而言之，当一个请求加入到队列中时，有可能发生四种操作，它们依次是：

- 第一，如果队列中已存在一个对相邻磁盘扇区操作的请求，那么新请求将和这个已经存在的请求合并成一个请求。
- 第二，如果队列中存在一个驻留时间过长的请求，那么新请求将被插入到队列尾部，以防止其他旧的请求发生饥饿。
- 第三，如果队列中以扇区方向为序存在合适的插入位置，那么新的请求将被插入到该位置，保证队列中的请求是以被访问磁盘物理位置为序进行排列的。
- 第四，如果队列中不存在合适的请求插入位置，请求将被插入到队列尾部。

13.5.3 最终期限I/O调度程序

最终期限(deadline)I/O调度程序是为了解决Linus电梯所带来的饥饿问题而提出的。出于减少

磁盘寻址时间的考虑，对某个磁盘区域上的繁重操作，无疑会使得磁盘其他位置上的操作请求得不到运行机会。实际上，一个对磁盘同一位置操作的请求流可以造成较远位置的其他请求永远得不到运行机会，这是一种很不公平的饥饿现象。

更糟糕的是，普通的请求饥饿还会带来名为写－饥饿－读（writes-starving-reads）这种特殊问题。写操作通常是在内核有空时才将请求提交给磁盘的，写操作完全和提交它的应用程序异步执行；读操作则恰恰相反，通常当应用程序提交一个读请求时，应用程序会发生堵塞直到读请求被满足，也就是说，读操作是和提交它的应用程序同步执行的。所以虽然写反应时间（提交写请求花费的时间）不会给系统响应速度造成很大影响，但是读响应时间（提交读请求花费的时间）对系统响应时间来说却非同小可。虽然写请求时间对应用程序性能[⊖]带来的影响不大，但是应用程序却必须等待读请求完成后才能运行其他程序，所以读操作响应时间对系统的性能非常重要。

问题还可能更严重，这是因为读请求往往会相互依靠，比如，要读大量的文件，每次都是针对一块很小的缓冲区数据区进行读操作，而应用程序只有将上一个数据区域从磁盘中读取并返回之后，才能继续读取下一个数据区（或下一个文件）。糟糕的是，不管是读还是写，二者都需要读取像索引节点这样的元数据。从磁盘进一步读取这些块会使I/O操作串行化。所以如果每一次请求都发生饥饿现象，那么对读取文件的应用程序来说，全部延迟加起来会造成过长的等待时间，让用户无法忍受。综上所述，读操作具有同步性，并且彼此之间往往相互依靠，所以读请求响应时间直接影响系统性能，因此2.6版本内核新引入了最后期限I/O调度程序来减少请求饥饿现象，特别是读请求饥饿现象。

注意，减少请求饥饿必须以降低全局吞吐量为代价。Linus电梯调度程序虽然也做了这样的折衷，但显然不够——Linus电梯可以提供更好的系统吞吐量（通过最小化寻址），可是它总按照扇区顺序将请求插入到队列，从不检查驻留时间过长的请求，更不会将请求插入到列队尾部，所以它虽然能让寻址时间最短，但是却会带来同样不可取的请求饥饿问题。为了避免饥饿同时提供良好的全局吞吐量，最后期限I/O调度程序做了更多的努力。既要尽量提高全局吞吐量，又要使请求得到公平处理，这是很困难的。

在最后期限I/O调度程序中，每个请求都有一个超时时间。默认情况下，读请求的超时时间是500毫秒，写请求的超时时间是5秒。最后期限I/O调度请求类似于Linus电梯，也以磁盘物理位置为次序维护请求队列，这个队列被称为排序队列。当一个新请求递交给排序队列时，最后期限I/O调度程序类似于Linus电梯[⊜]，合并和插入请求，但是最后期限I/O调度程序同时也会以请求类型为依据将它们插入到额外队列中。读请求按次序被插入到特定的读FIFO队列中，写请求被插入到特定的写FIFO队列中。虽然普通队列以磁盘扇区为序进行排列，但是这些队列是以FIFO（很有效，以时间为基准排序）形式组织的，结果新队列总是被加入到队列尾部。对于普通操作来说，最后期限I/O调度程序将请求从排序队列的头部取下，再推入到派发队列中，派发队列然后将请求提交给磁盘驱动，从而保证了最小化的请求寻址。

如果在写FIFO队列头，或是在读FIFO队列头的请求超时（也就时，当前时间超过了请求指定的

⊖ 不过，我们还是不打算把写请求无限期地延迟下去，因为内核想确保数据最终能写到磁盘，以避免在内存缓冲区中的数据变得越来越多或者太陈旧。

⊜ 最后期限I/O排序执行向前合并是一个可选项。因为读操作请求通常很少需要向前合并，所以向前合并通常不必考虑。

超时时间），那么最后期限I/O调度程序便从FIFO队列中提取请求进行服务。依靠这种方法，最后期限I/O调度程序试图保证不会发生有请求在明显超期的情况下仍不能得到服务的现象。请见图13-3。

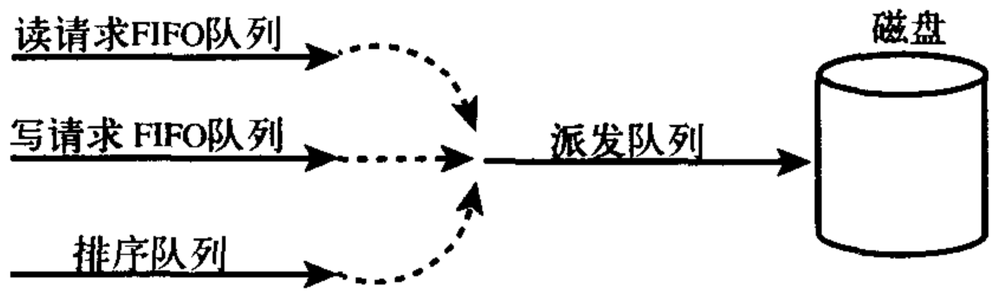


图13-3 最后期限I/O调度程序的三个队列

注意，最后期限I/O调度算法并不能严格保证请求的响应时间，但是通常情况下，可以在请求超时或超时前提交和执行，以防止请求饥饿现象的发生。由于读请求给定的超时时间要比写请求短许多，所以最后期限I/O调度器也确保了写请求不会因为堵塞读请求而使读请求发生饥饿。这种对读操作的照顾确保了读响应时间尽可能短。

最后期限I/O调度程序的实现在文件drivers/block/deadline-iosched.c中。

13.5.4 预测I/O调度程序

虽然最后期限I/O调度程序为降低读操作响应时间做了许多工作，但是它同时也降低了系统吞吐量。假设一个系统处于很繁重的写操作期间，每次提交读请求，I/O调度程序都会迅速处理读请求，所以磁盘首先为读操作进行寻址，执行读操作，然后返回再寻址进行写操作，并且对每个读请求都重复这个过程。这种做法对读请求来说是件好事，但是两次寻址操作（一次对读操作定位，一次返回来进行写操作定位）却损害了系统全局吞吐量。预测（Anticipatory）I/O调度程序的目标就是在保持良好的读响应的同时也能提供良好的全局吞吐量。

预测I/O调度的基础仍然是最后期限I/O调度程序，所以它们有很多相同之处。预测I/O调度程序也实现了三个队列（加上一个派发队列），并为每个请求设置了超时时间，这点与最后期限I/O调度程序一样。预测I/O调度程序最主要的改进是它增加了预测启发（anticipation-heuristic）能力。

预测I/O调度试图减少在进行I/O操作期间，处理新到的读请求所带来的寻址数量。和最后期限I/O调度程序一样，读请求通常会在超时前得到处理，但是预测I/O调度程序的不同之处在于，请求提交后并不直接返回处理其他请求，而是会有意空闲片刻（实际空闲时间可以设置，默认为6毫秒）。这几毫秒，对应用程序来说是个提交其他读请求的好机会——任何对相邻磁盘位置操作的请求都会立刻得到处理。在等待时间结束后，预测I/O调度程序重新返回原来的位置，继续执行以前剩下的请求。

要注意，如果等待可以减少读请求所带来的向后再向前（back-and-forth）寻址操作，那么完全值得花一些时间来等待更多的请求，如果一个相邻的I/O请求在等待期到来，那么I/O调度程序可以节省两次寻址操作。如果存在愈来愈多的访问同样区域的读请求到来，那么片刻等待无疑会避免大量的寻址操作。

当然，如果没有I/O请求在等待期到来，那么预测I/O调度程序会给系统性能带来轻微的损失，浪费掉几毫秒。预测I/O调度程序所能带来的优势取决于能否正确预测应用程序和文件系统的行为。这种预测依靠一系列的启发和统计工作。预测I/O调度程序跟踪并且统计每个应用程序块I/O操作的习惯行为，以便正确预测应用程序的未来行为。如果预测准确率足够高，那么预测调度程序便

可以大大减少服务读请求所需的寻址开销，而且同时仍能满足请求所需要的系统响应时间要求。这样的话，预测I/O调度程序既最小化了读响应时间，又能减少寻址次数和时间，所以说它既缩短了系统响应时间，又提高了系统吞吐量。

预测I/O调度程序的实现在文件内核源代码树的drivers/block/as-iosched.c中，它是Linux内核中默认的I/O调度程序，对大多数工作负荷来说都执行良好。对服务器也是理想的，不过，在某些非常见而又有严格工作负荷的服务器（包括数据库挖掘服务器）上，这个调度程序执行的效果不好。

13.5.5 完全公正的排队I/O调度程序

完全公正的排队I/O调度程序（Complete Fair Queuing，简称CFQ）是为专有工作负荷设计的，不过，在实际中，也为多种工作负荷提供了良好的性能。但是，它与前面介绍的I/O调度程序有根本的不同。

CFQ I/O调度程序把进入的I/O请求放入特定的队列中,这种队列是根据引起I/O请求的进程组织的。例如，来自foo进程的I/O请求进入foo队列，而来自bar进程的I/O请求进入bar队列。在每个队列中，刚进入的请求与相邻请求合并在一起，并进行插入分类。队列由此按扇区方式分类，这与其他I/O调度程序队列类似。CFQ I/O调度程序的差异在于每一个提交I/O的进程都有自己的队列。

CFQ I/O调度程序以时间片轮转调度队列，从每个队列中选取请求数（默认值为4，可以进行配置），然后进行下一轮调度。这就在进程级提供了公平，确保每个进程接收公平的磁盘带宽片断。预定的工作负荷是多媒体，在这种媒体中，这种公平的算法可以得到保证，比如，音频播放器总能够及时从磁盘再填满它的音频缓冲区。不过，实际上，CFQ I/O调度程序在很多场合都能很好地执行。

完全公正的排队I/O调度程序位于drivers/block/cfq-iosched.c。尽管这主要推荐给桌面工作负荷使用，但是，如果没有其他异常情况，它在几乎所有的工作负荷中都能很好地执行。

13.5.6 空操作的I/O调度程序

第四种也是最后一种I/O调度程序是空操作（Noop）I/O调度程序，之所以这样命名是因为它基本上是一个空操作，不做多少事情。空操作I/O调度程序不进行排序，或者也不进行什么其他形式的预寻道操作。依此类推，它也没必要实现那些老套的算法，也就是在以前的I/O调度程序中看到的为了最小化请求周期而采用的算法。

不过，空操作I/O调度程序忘不了执行合并，这就像它的家务事。当一个新的请求提交到队列时，就把它与任一相邻的请求合并。除了这一操作，空操作I/O调度程序的确再不做什么，只是维护请求队列以近乎FIFO的顺序排列，块设备驱动程序看可以从这种队列中摘取请求。

空操作I/O调度程序并不是一种懒散而毫无价值的I/O调度程序；它不勤奋工作是有道理的。因为它打算用在块设备，那是真正的随机访问设备，比如闪存卡。如果块设备只有一点或者没有“寻道”的负担，那么，就没有必要对进入的请求进行插入排序，因此，空操作I/O调度程序是理想的候选者。

空操作I/O调度程序位于drivers/block/noop-iosched.c，它是专为随机访问设备而设计的。

13.5.7 I/O调度程序的选择

你现在已经看到2.6内核中四种不同的I/O调度程序。其中的每一种I/O调度程序都可以被启用，

并内置在内核中。作为默认，块设备使用预测I/O调度程序。在启动时，可以通过命令行选项elevator =foo来覆盖默认，这里foo是一个有效而激活的I/O调度程序，参看表13-2。

表 13-2 给定 elevator选项的参数

参 数	I/O调度程序
as	预测
cfq	完全公正的排队
deadline	最终期限
noop	空操作

例如，内核命令行选项elevator=cfq会启用完全公正的I/O调度程序给所有的块设备。

13.6 小结

在本章，我们讨论了块设备的基本知识，并考察了块I/O层所用的数据结构：bio，表示活动的I/O操作；buffer_head，表示块到页的映射；还有请求结构，表示具体的I/O请求。我们追寻了I/O请求简单但重要的生命历程，其生命的重要点就是I/O调度程序。我们讨论了在调度I/O时如何在Linux内核目前的4种I/O调度程序，以及2.4的老式Linus电梯调度中做出选择。

下一章，我们将讨论进程地址空间。

第14章

进程地址空间

第11章介绍了内核如何管理物理内存。其实内核除了管理本身的内存外，还必须管理进程的地址空间——也就是系统中每个用户空间进程所看到的内存。Linux操作系统采用虚拟内存技术，因此，系统中的所有进程之间以虚拟方式共享内存。对每个进程来说，它们好像都可以访问整个系统的所有物理内存。更重要的是，即使单独一个进程，它拥有的地址空间也可以远远大于系统物理内存。本章将集中讨论内核如何管理进程地址空间。

进程地址空间由每个进程中的线性地址区组成，而且更为重要的特点是内核允许进程使用该空间中的地址。每个进程都有一个32或64位的平坦（flat）地址空间，空间的具体大小取决于体系结构。术语“平坦”描述的是地址空间范围是一个独立的连续区间（比如，地址从0扩展到429496729的32位地址空间）。一些操作系统提供了段地址空间，这种地址空间并非是一个独立的线性区域，而是被分段的，但现代采用虚拟内存的操作系统通常都使用平坦地址空间而不是分段式的内存模式。通常情况下，每个进程都有惟一的这种平坦地址空间，而且进程地址空间之间彼此互不相干。两个不同的进程可以在它们各自地址空间的相同地址内存放不同的数据。但是进程之间也可以选择共享地址空间，我们称这样的进程为线程。

内存地址是一个给定的值，它要在地址空间范围之内的，比如4021f000。这个值表示的是进程32位地址空间中的一个特定的字节。在地址空间中，我们更为关心的是进程有权访问的虚拟内存地址区间，比如08048000-0804c000。这些可被访问的合法地址区间被称为内存区域(memory area)，通过内核，进程可以给自己的地址空间动态地添加或减少内存区域。

进程只能访问有效范围内的内存地址。每个内存区域也具有相应进程必须遵循的特定访问属性，如只读、只写、可执行等属性。如果一个进程访问了不在有效范围中的地址，或以不正确的方式访问有效地址，那么内核就会终止该进程，并返回“段错误”信息。

内存区域可以包含各种内存对象，比如：

- 可执行文件代码的内存映射，称为代码段（text section）。
- 可执行文件的已初始化全局变量的内存映射，称为数据段（data section）。
- 包含未初始化全局变量，也就是bss段[⊖]的零页（页面中的信息全部为0值，所以可用于映射bss段等目的）的内存映射。
- 用于进程用户空间栈(不要和进程内核栈混淆，进程的内核栈独立存在并由内核维护)的零页

⊖ 术语“BBS”已经有些年头了，它是block started by symbol的缩写。因为未初始化的变量没有对应的值，所以并不需要存放在可执行对象中。但是因为C标准强制规定未初始化的全局变量要被赋予特殊的默认值（基本上是0值），所以内核要将变量（未赋值的）从可执行代码载入到内存中，然后将零页映射到该片内存上，于是这些未初始化变量就被赋予了0值。这样做避免了在目标文件中显式地进行初始化，减少空间浪费。

的内存映射。

- 每一个诸如C库或动态连接程序等共享库的代码段、数据段和bss也会被载入进程的地址空间。
- 任何内存映射文件。
- 任何共享内存段。
- 任何匿名的内存映射，比如由malloc()[⊖]分配的内存。

进程地址空间中的任何有效地址都只能位于惟一的区域，这些内存区域不能相互覆盖。可以看到，在执行的进程中，每个不同的内存片段都对应一个独立的内存区域：栈、对象代码、全局变量、被映射的文件等等。

14.1 内存描述符

内核使用内存描述符结构体表示进程的地址空间，该结构包含了和进程地址空间有关的全部信息。内存描述符由mm_struct结构体表示，定义在文件<linux/sched.h>[⊖]中。

下面给出内存描述符的结构和各个域的描述：

```
struct mm_struct {
        struct vm_area_struct    *mmap;              /*内核区域链表*/
        struct rb_root           mm_rb;              /* 虚拟内存区域红－黑树*/
        struct vm_area_struct    *mmap_cache;        /*最后使用的内存区域*/
        unsigned long            free_area_cache;    /*第一个地址空间洞*/
        pgd_t                    *pgd;               /*页全局目录*/
        atomic_t                 mm_users;           /*该地址空间的用户*/
        atomic_t                 mm_count;           /*主使用计数*/
        int                      map_count;          /*内存区域数目*/
        struct rw_semaphore      mmap_sem;           /*内存区域信号量*/
        spinlock_t               page_table_lock;    /*页表锁*/
        struct list_head         mmlist;             /*包含全部mm_struct的链表*/
        unsigned long            start_code;         /*代码段的开始地址*/
        unsigned long            end_code;           /*代码段的结束地址*/
        unsigned long            start_data;         /*数据的首地址*/
        unsigned long            end_data;           /*数据的尾地址*/
        unsigned long            start_brk;          /*堆的首地址*/
        unsigned long            brk;                /*堆的尾地址*/
        unsigned long            start_stack;        /*进程栈的首地址*/
        unsigned long            arg_start;          /*命令行参数的首地址*/
        unsigned long            arg_end;            /*命令行参数的尾地址*/
        unsigned long            env_start;          /*环境变量首地址*/
        unsigned long            env_end;            /*环境变量的尾地址*/
        unsigned long            rss;                /*所分配的物理页*/
        unsigned long            total_vm;           /*全部页面数目*/
        unsigned long            locked_vm;          /*上锁的页面数目*/
        unsigned long            def_flags;          /*默认的访问标志*/
        unsigned long            cpu_vm_mask;        /*懒惰（lazy）TLB交换掩码*/
```

⊖ 在最新版本的glibc中，通过mmap()和brk()来实现malloc()函数。

⊖ 因为进程描述符、内存描述符和它们相关函数相互之间紧密依存，所以mm_struct结构体被定义在文件shed.h中。

```
        unsigned long           swap_address;       /*最后被扫描的地址*/
        unsigned                dumpable:1;         /*是否可以产生内存信息转储*/
        int                     used_hugetlb;       /*是否使用了hugetlb?*/
        mm_context_t            context;            /*体系结构特殊数据*/
        int                     core_waiters;       /*内核转储等待线程*/
        struct completion       *core_startup_done; /* core 开始完成*/
        struct completion       core_done;          /* core结束完成*/
        rwlock_t                ioctx_list_lock;    /*AIO I/O链表锁*/
        struct kioctx           *ioctx_list;        /*AIO I/O链表*/
        struct kioctx           default_kioctx;     /*AIO默认I/O上下文*/
};
```

mm_users域记录正在使用该地址的进程数目。比如，如果两个进程共享该地址空间，那么mm_users的值便等于2；mm_count域是mm_struct结构体的主引用计数，只要mm_users不为0，那么mm_count值就等于1。当mm_users的值减为0（两个线程都退出）时，mm_count域的值才变为0。如果mm_count的值等于0，说明已经没有任何指向该mm_struct结构体的引用了，这时该结构体会被销毁。内核同时使用这两个计数器是为了区别主使用计数(mm_count)器和使用该地址空间的进程的数目（mm_users）。

mmap和mm_rb这两个不同数据结构体描述的对象是相同的：该地址空间中的全部内存区域。但是前者以链表形式存放而后者以红-黑树的形式存放。红-黑树是一种二叉树，与其他二叉树一样，搜索它的时间复杂度为O（log n）。在本章后续部分将进一步讨论红-黑树。

内核通常会避免使用两种数据结构组织同一种数据，但此处内核这样的冗余确实派得上用场。mmap结构体作为链表，利于简单、高效地遍历所有元素；而mm_rb结构体作为红-黑树，更合适搜索指定元素。对内存区域的具体操作将在本章的后续部分详细介绍。

所有的mm_struct结构体都通过自身的mmlist域连接在一个双向链表中，该链表的首元素是init_mm内存描述符，它代表init进程的地址空间。另外要注意，操作该链表的时候需要使用mmlist_lock锁来防止并发访问，该锁定义在文件kernel/fork.c中。内存描述符的总数存放在mmlist_nr全局变量中，该变量也定义在文件fork.c中。

14.1.1 分配内存描述符

在进程的进程描述符中，mm域存放着该进程使用的内存描述符，所以current-> mm便指向当前进程的内存描述符。fork()函数利用copy_mm()函数复制父进程的内存描述符，也就是current->mm域给其子进程，而子进程中的mm_struct结构体实际是通过文件kernel/fork.c中的allocate_mm()宏从mm_cachep slab缓存中分配得到的。通常，每个进程都有惟一的mm_struct结构体，即惟一的进程地址空间。

如果父进程希望和其子进程共享地址空间，可以在调用clone()时，设置CLONE_VM标志。我们把这样的进程称作线程。回忆第3章，是否共享地址空间，几乎是进程和Linux中所谓的线程间本质上的惟一区别。除此以外，Linux内核并不区别对待它们，线程对内核来说仅仅是一个共享特定资源的进程而已。

当CLONE_VM被指定后，内核就不再需要调用allocate_mm()函数了，而仅仅需要在调用copy_mm()函数中将mm域指向其父进程的内存描述符就可以了：

```
if(clone_flags & CLONE_VM){
        /*
         * current 是父进程而tsk在fork()执行期间是子进程
         */
        atomic_inc(&current->mm->mm_users);
        tsk->mm= current ->mm;
}
```

14.1.2 销毁内存描述符

当进程退出时，内核会调用exit_mm()函数，该函数执行一些常规的销毁工作，同时更新一些统计量。其中，该函数会调用mmput()函数减少内存描述符中的mm_users用户计数，如果用户计数降到零，继续调用mmdrop()函数，减少mm_count使用计数。如果使用计数也等于零了，说明该内存描述符不再有任何使用者了，那么调用free_mm()宏通过kmem_cache_free()函数将mm_struct结构体归还到mm_cachep slab 缓存中。

14.1.3 mm_struct 与内核线程

内核线程没有进程地址空间，也没有相关的内存描述符。所以内核线程对应的进程描述符中mm域为空。事实上，这也正是内核线程的真实含义——它们没有用户上下文。

省了进程地址空间再好不过了，因为内核线程并不需要访问任何用户空间的内存(那它们访问谁的呢？)。而且因为内核线程在用户空间中没有任何页，所以实际上它们并不需要有自己的内存描述符和页表(后面的小节将讲述页表)。尽管如此，即使访问内核内存，内核线程也还是需要使用一些数据的，比如页表。为了避免内核线程为内存描述符和页表浪费内存，也为了当新内核线程运行时，避免浪费处理器周期向新地址空间进行切换，内核线程将直接使用前一个进程的内存描述符。

当一个进程被调度时，该进程的mm域指向的地址空间被装载到内存，进程描述符中的active_mm域会被更新，指向新的地址空间。内核线程没有地址空间，所以mm域为NULL。于是，当一个内核线程被调度时，内核发现它的mm域为NULL，就会保留前一个进程的地址空间，随后内核更新内核线程对应的进程描述符中的active_mm域，使其指向前一个进程的内存描述符。所以在需要时，内核线程便可以使用前一个进程的页表。因为内核线程不访问用户空间的内存，所以它们仅仅使用地址空间中和内核内存相关的信息，这些信息的含义和普通进程完全相同。

14.2 内存区域

内存区域由vm_area_struct结构体描述，定义在文件<linux/mm.h>中，内存区域在内核中也经常被称做*虚拟内存区域*或VMA。

vm_area_struct结构体描述了指定地址空间内连续区间上的一个独立内存范围。内核将每个内存区域作为一个单独的内存对象管理，每个内存区域都拥有一致的属性，比如访问权限等，另外，相应的操作也都一致。这种管理方式类似于VFS层(请看第12章)，采用面向对象方法使VMA结构体可以代表多种类型的内存区域——比如内存映射文件或进程的用户空间栈等。下面给出该结构定义和各个域的描述：

```
struct vm_area_struct {
        struct mm_struct              *vm_mm;         /*相关的mm_struct结构体*/
        unsigned long                 vm_start;       /*区间的首地址*/
        unsigned long                 vm_end;         /*区间的尾地址*/
        struct vm_area_struct         *vm_next;       /*VMA链表*/
        pgprot_t                      vm_page_prot;   /*访问控制权限*/
        unsigned long                 vm_flags;       /*标志*/
        struct rb_node                vm_rb;          /* 树上该VMA的节点*/
        union {  /* 或者是关联于address_space->i_mmap字段，或者是关联于address_space-
                >i_mmap_nonlinear字段 */
           struct {
                   struct list_head        list;
                   void                    *parent;
                   struct vm_area_struct   *head;
           } vm_set;
           struct prio_tree_node prio_tree_node;
        } shared;
        struct list_head              anon_vma_node;  /*anon_vma目录项*/
        struct anon_vma               *anon_vma;      /*匿名的VMA对象*/
        struct vm_operations_struct   *vm_ops;        /*相关的操作表*/
        unsigned long                 vm_pgoff;       /*文件中的偏移量*/
        struct file                   *vm_file;       /*被映射的文件（如果存在）*/
        void                          *vm_private_data; /*私有数据*/
};
```

每个内存描述符都对应于进程地址空间中的惟一区间。vm_start域指向区间的首地址(最低地址)，vm_end域指向区间的尾地址(最高地址)之后的第一个字节，也就是说，vm_start 是内存区间的开始地址（它本身在区间内），而vm_end是内存区间的结束地址（它本身在区间外），因此，vm_end~vm_start的大小便是内存区间的长度，内存区域的位置就在[vm_start, vm_end]之中。注意，在同一个地址空间内的不同内存区间不能重叠。

vm_mm域指向和VMA相关的mm_struct结构体，注意每个VMA对其相关的mm_struct结构体来说都是惟一的，所以即使两个独立的进程将同一个文件映射到各自的地址空间，它们分别都会有一个vm_area_struct结构体来标志自己的内存区域；但是如果两个线程共享一个地址空间，那么它们也同时共享其中的所有vm_area_struct结构体。

14.2.1 VMA标志

VMA标志是一种位标志，其定义见<linux/mm.h>。它包含在vm_flags域内，标志了内存区域所包含的页面的行为和信息。和物理页的访问权限不同，VMA标志反映了内核处理页面所需要遵守的行为准则，而不是硬件要求。该标志同时也包含了内存区域中页面的信息，或内存区域的整体信息，表14-1列出了所有VMA标志。

让我们进一步看看其中有趣和重要的几种标志，VM_READ、VM_WRITE和VM_EXEC标志了内存区域中页面的读、写和执行权限。这些标志根据要求组合构成VMA的访问控制权限，当访问VMA时，需要查看其访问权限。比如进程的对象代码映射区域可能会标志为VM_READ和VM_EXEC，而没有标志为VM_ WRITE；另一方面，可执行对象数据段的映射区域被标志为

VM_READ和VM_WRITE，而VM_EXEC标志对它就毫无意义；只读文件数据段的映射区域仅可被标志为VM_READ。

表14-1 VMA标志

标 志	对VMA及其页面的影响
VM_READ	页面可读取
VM_WRITE	页面可写
VM_EXEC	页面可执行
VM_SHARED	页面可共享
VM_MAYREAD	VM_READ标志可被设置
VM_MAYWRITE	VM_WRITE标志可被设置
VM_MAYEXEC	VM_EXEC标志可被设置
VM_MAYSHARE	VM_SHARE标志可被设置
VM_GROWSDOWN	区域可向下增长
VM_GROWSUP	区域可向上增长
VM_SHM	区域可用作共享内存
VM_DENYWRITE	区域映射一个不可写文件
VM_EXECUTABLE	区域映射一个可执行文件
VM_LOCKED	区域中的页面被锁定
VM_IO	区域映射设备I/O空间
VM_SEQ_READ	页面可能会被连续访问
VM_RAND_READ	页面可能会被随机访问
VM_DONTCOPY	区域不能在fork()时被拷贝
VM_DONTEXPAND	区域不能通过mremap()增加
VM_RESERVED	区域不能被换出
VM_ACCOUNT	该区域是一个记账VM对象
VM_HUGETLB	区域使用了hugetlb页面
VM_NONLINEAR	该区域是非线性映射的

VM_SHARED指明了内存区域包含的映射是否可以在多进程间共享，如果该标志被设置，则我们称其为共享映射；如果未被设置，则只有一个进程可以使用该映射的内容，我们称它为私有映射。

VM_IO标志内存区域中包含对设备I/O空间的映射。该标志通常在设备驱动程序执行mmap()函数进行I/O空间映射时才被设置，同时该标志也表示该内存区域不能被包含在任何进程的存放转存（core dump）中。VM_RESERVED标志内存区域不能被换出，它也是在设备驱动程序进行映射时被设置。

VM_SEQ_READ标志暗示内核应用程序对映射内容执行有序的(线性和连续的)读操作；VM_RAND_READ标志的意义正好相反，暗示应用程序对映射内容执行随机的(非有序的)读操作。因此内核可以有意地减少或彻底取消文件预读，所以这两个标志可以通过系统调用madvise()设置，设置参数分别是MADV_SEQUENTIAL和MADV_RANDOM。文件预读是指在读数据时有意地按顺序多读取一些本次请求以外的数据——希望多读的数据能够很快就被需要。这种预读行为对那些顺序读取数据的应用程序有很大的好处，但是如果数据的访问是随机的，那么预读显然就多余了。

14.2.2 VMA 操作

vm_area_struct结构体中的vm_ops域指向与指定内存区域相关的操作函数表，内核使用表中的

方法操作VMA。vm_area_struct作为通用对象代表了任何类型的内存区域，而操作表描述针对特定的对象实例的特定方法。

操作函数表由vm_operations_struct结构体表示，定义在文件<linux/mm.h>中：

```
struct vm_operations_struct {
        void (*open)(struct vm_area_struct *);
        void (*close)(struct vm_area_struct *);
        struct page * (*nopage)(struct vm_area_struct *, unsigned long, int);
        int (*populate)(struct vm_area_struct *, unsigned long, unsigned long,
                        pgprot_t, unsigned long, int);
};
```

下面介绍具体方法：

- void open(struct vm_area_struct *area)

当指定的内存区域被加入到一个地址空间时，该函数被调用。

- void close(struct vm_area_struct *area)

当指定的内存区域从地址空间删除时，该函数被调用。

- struct page* nopage(struct vm_area_struct *area,
  unsigned long address,
  int unused)

当要访问的页不在物理内存中时，该函数被页错误处理程序调用。

- int populate(struct vm_area_struct *area,
  unsigned long address,
  unsigned long len,pgprot_t prot,
  unsigned long pgoff,int nonblock)

该函数被系统调用remap_pages()调用来为将要发生的缺页中断预映射一个新映射。

14.2.3 内存区域的树型结构和内存区域的链表结构

上文讨论过，可以通过内存描述符中的mmap和mm_rb域之一访问内存区域。这两个域各自独立地指向与内存描述符相关的全体内存区域对象。其实，它们包含完全相同的vm_area_ struct结构体的指针，仅仅是组织方法不同。

mmap域使用单独链表连接所有的内存区域对象。每一个vm_area_struct结构体通过自身的vm_next域被连入链表，所有的区域按地址增长的方向排序，mmap域指向链表中第一个内存区域，链中最后一个VMA结构体指针指向空。

mm_rb域使用红－黑树连接所有的内存区域对象。mm_rb域指向红－黑树的根节点，地址空间中每一个vm_area_struct结构体通过自身的vm_rb域连接到树中。红－黑树是一种二叉树，树中的每一个元素被称为一个节点，最初的节点被称为树根。红－黑树的多数节点都由两个子节点：一个左儿子节点和一个右儿子节点，不过也有节点只有一个子节点的情况。树末端的节点被称为叶子节点，它们没有子节点。红－黑树中的所有节点都遵从：左边节点值小于右边节点值；另外每个节点都被配以红色或黑色(要么红要么黑，所以叫做红－黑树)。分配的规则为：红节点的子节点为黑色，并且树中的任何一条从节点到叶子的路径必须包含同样数目的黑色节点。记住根节点总为红色。红－黑树的搜索、插入、删除等操作的复杂度都为O(log(n))。

链表用于需要遍历全部节点的时候，而红-黑树适用于在地址空间中定位特定内存区域的时候。内核为了内存区域上的各种不同操作都能获得高性能，所以同时使用了这两种数据结构。

14.2.4 实际使用中的内存区域

可以使用/proc文件系统和pmap (1)工具查看给定进程的内存空间和其中所含的内存区域。我们来看一个非常简单的用户空间程序的例子，它其实什么也不做，仅仅是为了做说明用：

```
int main(int argc,char *argv[])
{
        return 0;
}
```

下面列出该进程地址空间中包含的内存区域。其中有代码段、数据段和bss段等。假设该进程与C库动态连接，那么地址空间中还将分别包含libc.so和ld.so对应的上述三种内存区域。此外，地址空间中还要包含进程栈对应的内存区域。

/proc/<pid>/map的输出显示了该进程地址空间中的全部内存区域：

```
rml@phantasy:~$ cat /proc/1426/maps
00e80000 - 00faf000 r-xp 00000000 03:01 208530  /lib/tls/libc-2.3.2.so
00faf000 - 00fb2000 rw-p 0012f000 03:01 208530  /lib/tls/libc-2.3.2.so
00fb2000 - 00fb4000 rw-p 00000000 00:00 0
08048000 - 08049000 r-xp 00000000 03:03 439029  /home/rml/src/example
08049000 - 0804a000 rw-p 00000000 03:03 439029  /home/rml/src/example
40000000 - 40015000 r-xp 00000000 03:01 80276   /lib/ld-2.3.2.so
40015000 - 40016000 rw-p 00015000 03:01 80276   /lib/ld-2.3.2.so
4001e000 - 4001f000 rw-p 00000000 00:00 0
bfffe000 - c0000000 rwxp fffff000 00:00 0
```

每行数据格式如下：

```
开始－结束 访问权限  偏移  主设备号：次设备号 i节点  文件
```

pmap(1)工具[⊖]将上述信息以更方便阅读的形式输出：

```
rml@phantasy:~$ pmap 1426
example [1426]
00e80000(1212KB)        r-xp   (03:01 208530)       /lib/tls/libc-2.3.2.so
00faf000(12KB)          rw-p   (03:01 208530)       /lib/tls/libc-2.3.2.so
00fb2000(8KB)           rw-p   (00:00 0)
08048000(4KB)           r-xp   (03:03 439029)       /home/rml/src/example
08049000(4KB)           rw-p   (03:03 439029)       /home/rml/src/example
40000000(84KB)          r-xp   (03:01 80276)        /lib/ld-2.3.2.so
40015000 (4KB)          rw-p   (03:01 80276)        /lib/ld-2.3.2.so
4001e000(4KB)           rw-p   (00:00 0)
bfffe000(8KB)           rwxp   (00:00 0)           [stack]
mapped :1340KB          writable/private : 40KB    shared :0KB
```

⊖ pmap(1)工具将进程的内存区域格式化后显示，虽然结果中的信息和/proc中的是一样的信息，但它的输出形式比/proc的输出形式更具可读性。在新版本的procps包中可以找到这个工具。

前三行分别对应C库中libc.so的代码段、数据段和bss段，接下来的两行为可执行对象的代码段和数据段，再下来三行为动态连接程序ld.so的代码段，数据段和bss段，最后一行是进程的栈。

注意代码段具有我们所要求的可读且可执行权限；另一方面，数据段和bss（它们都包含全局数据变量）具有可读、可写但不可执行权限。而堆栈则可读、可写，甚至还可执行——虽然这点并不常用到。

该进程的全部地址空间大约为1340KB，但是只有大约40KB的内存区域是可写和私有的。如果一片内存范围是共享的或不可写的，那么内核只需要在内存中为文件（backing file）保留一份映射。对于共享映射来说，这样做没什么特别的，但是对于不可写内存区域也这样做，就有些让人奇怪了？如果考虑到映射区域不可写意味着该区域不可被改变(映射只用来读)，就应该清楚只把该映像读入一次是很安全的。所以C库在物理内存中仅仅需要占用1212KB空间，而不需要为每个使用C库的进程在内存中都保存一个1212KB的空间。进程访问了1340KB的数据和代码空间，然而紧紧消耗了40KB的物理内存，可以看出利用这种共享不可写内存的方法节约了大量的内存空间。

注意没有映射文件的内存区域的设备标志为00：00，索引节点标志也为0，这个区域就是零页——零页映射的内容全为零。如果将零页映射到可写的内存区域，那么该区域将被初始化为全0。这是零页的一个重要用处，而bss段需要的就是全0的内存区域。由于内存未被共享，所以只要一有进程写该处数据，那么该处数据就将被拷贝出来（就是我们所说的写时拷贝），然后才被更新。

每个和进程相关的内存区域都对应于一个vm_area_struct结构体。另外进程不同于线程，进程结构体task_struct包含惟一的mm_struct结构体引用。

14.3 操作内存区域

内核时常需要判断进程地址空间中的内存区域是否满足某些条件，比如某个指定地址是否包含在某个内存区域中。这类操作非常频繁，另外它们也是mmap()例程的基础——我们在下一节会讨论mmap()操作。为了方便执行这类对内存区域的操作，内核定义了许多的辅助函数，它们都声明在文件<linux/mm.h>中。

14.3.1 find_vma()

find_vma()函数定义在文件<mm/mmap.c>中。

该函数在指定的地址空间中搜索第一个vm_end大于addr的内存区域。换句话说，该函数寻找第一个包含addr或首地址大于addr的内存区域，如果没有发现这样的区域，该函数返回NULL；否则返回指向匹配的内存区域的vm_area_struct结构体指针。注意，由于返回的VMA首地址可能大于addr，所以指定的地址并不一定就包含在返回的VMA中。因为很有可能在对某个VMA执行操作后，还有其他更多的操作会对该VMA接着进行操作，所以find_vma()函数返回的结果被缓存在内存描述符的mmap_cache域中。实践证明被缓存的 VMA会有相当好的命中率(实践中大约30%～40%)，而且检查被缓存的VMA速度会很快，如果指定的地址不在缓存中，那么必须搜索和内存描述符相关的所有内存区域。这种搜索通过红－黑树进行：

```
struct vm_area_struct *find_vma(struct mm_struct*mm,unsigned long addr)
{
        struct vm_area_struct *vma = NULL;
```

```
if(mm){
        vma = mm->mmap_cache;
        if(!(vma && vma->vm_end>addr && vma->vm_start <=addr)){
                struct rb_node * rb_node;

                rb_node = mm->mm_rb.rb_node;
                vma  =  NULL;
                while(rb_node){
                        struct  vm_area_struct * vma_tmp;

                        vma_tmp = rb_entry(rb_node,
                                        struct vm_area_struct, vm_rb);
                        if(vma_tmp -> vm_end>addr){
                                vma = vma_tmp;
                                if(vma_tmp->vm_start<=addr)
                                        break;
                                rb_node = rb_node->rb_left;
                        }else
                                rb_node = rb_node ->rb_right;
                }
                if(vma)
                        mm->mmap_cache = vma;
        }
}
return vma;
}
```

首先，该函数检查mmap_cache，看看缓存的VMA是否包含了需要地址。注意简单地检查VMA的vm_end是否大于addr，并不能保证该VMA是第一个大于addr的内存区域，所以缓存要想发挥作用，那么就要求指定的地址必须包含在被缓存的VMA中——幸好，这也正是连续操作同——VMA必然发生的情况。

如果缓存中并未包含希望的VMA，那么该函数必须搜索红－黑树。如果当前VMA的vm_end大于addr，进入左子节点继续搜索；否则，沿右边子节点搜索，直到找到包含addr的VMA为止。如果没有包含addr 的VMA被找到，那么该函数继续搜索树，并且返回大于addr的第一个VMA。如果也不存在满足要求的VMA，那该函数返回NULL。

14.3.2 find_vma_prev()

find_vma_prev()函数和find_vma()工作方式相同，但是它返回第一个小于addr的VMA。该函数定义和声明分别在文件mm/mmap.c中和文件<linux/mm.h>中：

```
struct vm_area_struct * find_vma_prev(struct mm_struct *mm,unsigned long addr ,
                                      struct vm_area_struct **pprev)
```

pprev参数存放指向先于addr的VMA指针。

14.3.3 find_vma_intersection()

find_vma_intersection()函数返回第一个和指定地址区间相交的VMA。因为该函数是内联函数，

所以定义在文件<linux/mm.h>中：

```
static inline struct vm_area_struct *
find_vma_intersection(struct mm_struct * mm,
                      unsigned long start_addr,
                      unsigned long end_addr)
{
        struct vm_area_struct * vma;

        vma = find_vma(mm,start_addr);
        if (vma && end_addr <= vma->vm_start)
                vma = NULL;
        return vma;
}
```

第一个参数mm是要搜索的地址空间，start_addr是区间的开始首位置，end_addr是区间的尾位置。

显然，如果find_vma()返回NULL，那么find_vma_intersection()也会返回NULL。但是如果find_vma()返回有效的VMA，find_vma_intersection()只有在该VMA的起始位置于给定的地址区间结束位置之前，才将其返回。如果VMA的起始位置大于指定地址范围的结束位置，则该函数返回NULL。

14.4 mmap()和do_mmap()：创建地址区间

内核使用do_mmap()函数创建一个新的线性地址区间。但是说该函数创建了一个新VMA并不非常准确，因为如果创建的地址区间和一个已经存在的地址区间相邻，并且它们具有相同的访问权限的话，那么两个区间将合并为一个。如果不能合并，那么就确实需要创建一个新的VMA了。但无论哪种情况，do_mmap()函数都会将一个地址区间加入到进程的地址空间中——无论是扩展已存在的内存区域还是创建一个新的区域。

do_mmap()函数定义在文件<linux/mm.h>中。

```
unsigned long do_mmap(struct file *file, unsigned long addr,
                      unsigned long len, unsigned long prot,
                      unsigned long flag, unsigned long offset)
```

该函数映射由file指定的文件，具体映射的是文件中从偏移offset处开始，长度为len字节的范围内的数据。如果file参数是NULL并且offset参数也是0，那么就代表这次映射没有和文件相关，该情况被称作匿名映射（anonymous mapping）。如果指定了文件名和偏移量，那么该映射被称为文件映射（file-backed mapping）。

addr是可选参数，它指定搜索空闲区域的起始位置。

prot参数指定内存区域中页面的访问权限。访问权限标志定义在文件<asm/mman.h>中，不同体系结构标志的定义有所不同，但是对所有体系结构而言，都会包含表14-2中所列举的标志。flag参数指定了VMA标志，这些标志也定义在文件<asm/mman.h>中，请参看表14-3。

表14-2 页保护标志

标 志	对新建区间中页的要求
PROT_READ	对应于VM_READ
PROT_WRITE	对应于VM_WRITE
PROT_EXEC	对应于VM_EXEC
PROT_NONE	页不可被访问

表14-3 页保护标志

标 志	对新区间的要求
MAP_SHARED	映射可以被共享
MAP_PRIVATE	映射不能被共享
MAP_FIXED	新区间必须开始于指定的地址addr
MAP_ANONYMOUS	映射不是file-backed，而是匿名的
MAP_GROWSDOWN	对应于VM_GROWSDOWN
MAP_DENYWRITE	对应于VM_DENYWRITE
MAP_EXECUTABLE	对应于VM_EXECUTABLE
MAP_LOCKED	对应于VM_LOCKED
MAP_NORESERVE	不需要为映射保留空间
MAP_POPULATE	填充页表
MAP_NONBLOCK	在I/O操作上不堵塞

如果系统调用do_mmap()的参数中有无效参数，那么它返回一个负值；否则，它会在虚拟内存中分配一个合适的新内存区域。如果有可能的话，将新区域和临近区域进行合并，否则内核从vm_area_cachep长字节（slab)缓存中分配一个vm_area_struct结构体，并且使用vma_ link()函数将新分配的内存区域添加到地址空间的内存区域链表和红－黑树中，随后还要更新内存描述符中的total_vm域，然后才返回新分配的地址区间的初始地址。

mmap()系统调用

在用户空间可以通过mmap()系统调用获取内核函数do_mmap()的功能。mmap()系统调用定义如下：

```
void *mmap2(void *start,
            size_t length,
            int prot,
            int flags,
            int fd,
            off_t pgoff)
```

由于该系统调用是mmap()调用的第二种变种，所以起名为mmap2()。最原始的mmap()调用中最后一个参数是字节偏移量，而目前这个mmap2()使用页面偏移作最后一个参数。使用页面偏移量可以映射更大的文件和更大的偏移位置。原始的 mmap()调用由POSIX定义，仍然在C库中作为mmap()方法被使用，但是内核中已经没有对应的实现了，而实现的是新方法mmap2()。虽然C库仍然可以使用的原始版本的映射方法，但是它其实还是基于函数mmap2()进行的，因为对原始mmap()方法的调用是通过将字节偏移转化为页面偏移,从而转化为对mmap2()函数的调用。

14.5 munmap()和do_munmap()：删除地址区间

do_munmap()函数从特定的进程地址空间中删除指定地址区间，该函数定义在文件<linux/mm.h>中：

```
int do_munmap(struct mm_struct *mm,unsigned long start, size_t len)
```

第一个参数指定要删除区域所在的地址空间，删除从地址start开始，长度为len字节的地址区

间。如果成功，返回零。否则返回负的错误码。

munmap()系统调用

系统调用munmap()给用户空间程序提供了一种从自身地址空间中删除指定地址区间的方法，它和系统调用mmap()的作用相反:

```
int munmap(void *start, size_t length)
```

该系统调用定义在文件 mm/mmap.c中，它是对do_munmap()函数的一个简单的封装：

```
asmlinkage long sys_munmap(unsigned long addr, size_t len)
{
        int ret;
        struct mm_struct *mm;
        mm = current->mm;
        down_write(&mm->mmap_sem);
        ret = do_munmap(mm, addr, len);
        up_write(&mm->mmap_sem);
        return ret;
}
```

14.6 页表

虽然应用程序操作的对象是映射到物理内存之上的虚拟内存，但是处理器直接操作的却是物理内存。所以当用程序访问一个虚拟地址时，首先必须将虚拟地址转化成物理地址，然后处理器才能解析地址访问请求。地址的转换工作需要通过查询页表才能完成，概括地讲，地址转换需要将虚拟地址分段，使每段虚地址都作为一个索引指向页表，而页表项则指向下一级别的页表或者指向最终的物理页面。

Linux中使用三级页表完成地址转换。利用多级页表能够节约地址转换需占用的存放空间。如果利用三级页表转换地址，即使是64位机器，占用的空间也很有限。但是如果使用静态数组实现页表，那么即便在32位机器上，该数组也将占用巨大的存放空间。Linux对所有体系结构，包括对那些不支持三级页表的体系结构(比如，有些体系结构只使用两级页表或者使用散列表完成地址转换)都使用三级页表管理，因为使用三级页表结构可以利用“最大公约数”的思想——一种设计简单的体系结构，可以按照需要在编译时简化使用页表的三级结构，比如只使用两级。

顶级页表是页全局目录(PGD)。PGD包含了一个pgd_t类型数组，多数体系结构中pgd_t类型等同于无符号长整型类型。PGD中的表项指向二级页目录中的表项：PMD。

二级页表是中间页目录(PMD)。PMD是个pmd_t类型数组，其中的表项指向PTE中的表项。

最后一级的页表简称页表，其中包含了pte_t类型的页表项，该页表项指向物理页面。

多数体系结构中，搜索页表的工作是由硬件完成的（至少某种程度上）。虽然通常操作中，很多使用页表的工作都可以由硬件执行，但是只有在内核正确设置页表的前提下，硬件才能方便地操作它们。图14-1描述了虚拟地址通过页表找到物理地址的过程。

每个进程都有自己的页表(当然，线程会共享页表)。内存描述符的pgd域指向的就是进程的页全局目录。注意，操作和检索页表时必须使用page_table_lock锁，该锁在相应的进程的内存描述符

中，以防止竞争条件。

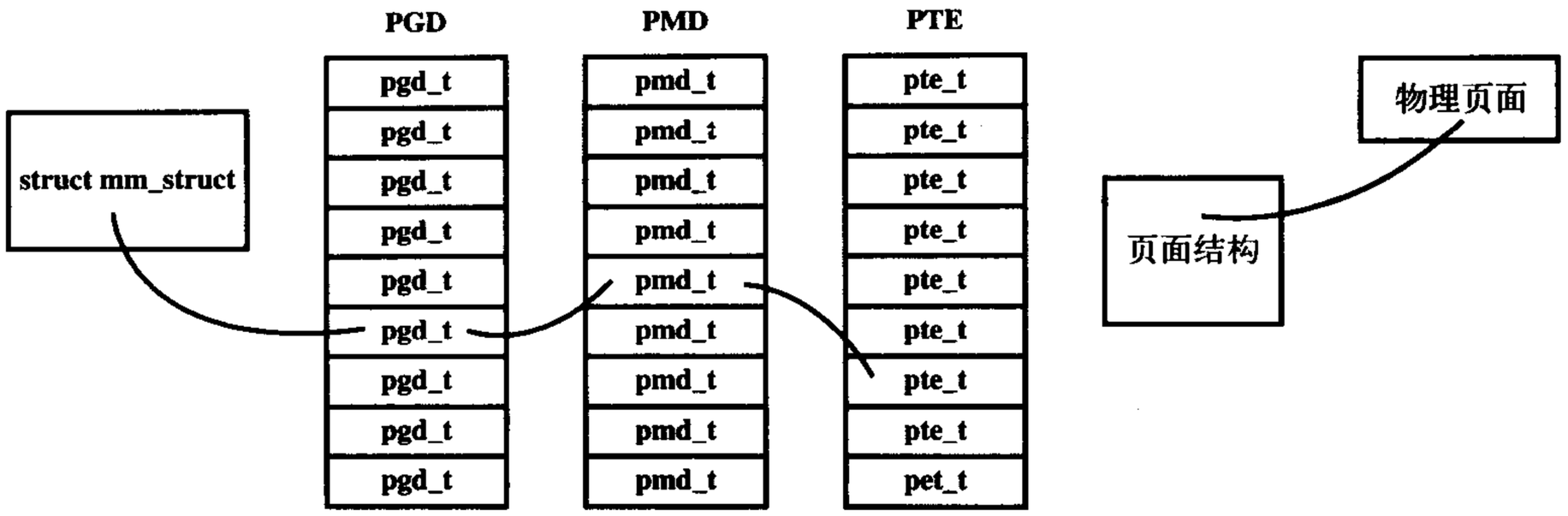


图14-1 页表

页表对应的结构体依赖于具体的体系结构，所以定义在文件<asm/page.h>中。

由于几乎每次对虚拟内存中的页面访问都必须先解析它，从而得到物理内存中的对应地址，所以页表操作的性能非常关键。但不幸的是，搜索内存中的物理地址速度很有限，因此为了加快搜索，多数体系结构都实现了一个翻译后缓冲器（translation lookaside buffer，TLB）。TLB作为一个将虚拟地址映射到物理地址的硬件缓存，当请求访问一个虚拟地址时，处理器将首先检查TLB中是否缓存了该虚拟地址到物理地址的映射，如果在缓存中直接命中，物理地址立刻返回，否则，如果未命中，那么就需要再通过页表搜索需要的物理地址。

虽然硬件完成了有关页表的部分工作，但是页表的管理仍然是内核的关键部分——而且在不断改进。2.6内核对页表管理的主要改进是：从高端内存分配部分页表。今后可能的改进包括通过在写时拷贝（copy-on-write）的方式共享页表。这种机制使得在fork()操作中可由父子进程共享页表。因为只有当子进程或父进程试图修改特定页表项时，内核才去创建该页表项的新拷贝，此后父子进程才不再共享该页表项。可以看到，利用共享页表可以消除fork()操作中页表拷贝所带来的消耗。

14.7 小结

这章的内容不能不说是很“难缠”啦。其中，我们看到了抽象出来的进程虚拟内存，看到了内核如何表示进程空间（通过mm_struct）以及内核如何表示该空间中的内存区域（通过结构体vm_area_struct）。除此以外，我们还了解了内核如何创建（通过mmap()）和销毁（通过munmap()）这些内存区域，最后我们还讨论了页表。因为Linux是一个基于虚拟内存的操作系统，所以这些概念对于系统运行来说都非常基础，一定要仔细领会。

下一章，我们要讨论页缓存——一种用于所有页I/O操作的内存数据缓存，而且还要涵盖内核执行基于页的数据回写。

大家振作点，坚持就是胜利。

第⑮章

页高速缓存和页回写

页高速缓存（cache）是 Linux内核实现的一种主要磁盘缓存。它主要用来减少对磁盘的I/O操作。具体地讲，是通过把磁盘中的数据缓存到物理内存中，把对磁盘的访问变为对物理内存的访问。这一章将讨论页高速缓存与页回写。

磁盘高速缓存的价值在于两个方面：第一，访问磁盘的速度要远远低于访问内存的速度，因此，从内存访问数据比从磁盘访问速度更快。第二，数据一旦被访问，就很有可能在短期内再次被访问到。这种在短时期内集中访问同一片数据的原理被称作*临时局部原理*（temporal locality）。临时局部原理能保证：如果在第一次访问数据时缓存它，那就极有可能在短期内再次被高速缓存命中（访问到高速缓存中的数据）。

页高速缓存是由RAM中的物理页组成的，缓存中每一页都对应着磁盘中的多个块。每当内核开始执行一个页I/O操作时（通常是对普通文件中页大小的块进行磁盘操作），首先会检查需要的数据是否在高速缓存中，如果在，那么内核就直接使用高速缓存中的数据，从而避免访问磁盘。

也可以通过块I/O缓冲区把独立的磁盘块与页高速缓存联系在一起。回顾第13章，一个缓冲(buffer)就是一个单独物理磁盘块在内存中的表示，缓冲就是内存到磁盘块的映射描述符，因此通过缓存磁盘块以及缓冲块I/O操作，页高速缓存同样也可以减少块I/O操作期间的磁盘访问量。这种缓存经常被称为“缓冲区高速缓存”，虽然它实际上并不是一个独立的缓存，而是页高速缓存中的一部分。

我们可以看一下页高速缓存封装的操作和数据，在页I/O操作中，比如执行read()和write()时，页缓存最常被用到。每次页I/O操作都要处理数据的全部页面，这就需要对一个以上的磁盘块进行操作，所以页高速缓存实际缓存的是页面大小的文件块。

块I/O操作每次操作一个单独的磁盘块，比如读写索引接点就是一个典型的块I/O操作。内核提供bread()底层函数从磁盘读单个块，通过缓冲，这些磁盘块被映射到内存中相应的页面上，这样，也就被缓存到页高速缓存里了。

举个例子，当使用文本编辑器打开一个源程序文件时，该文件的数据就被调入内存。编辑该文件的过程中，越来越多的数据会相继被调入内存页。最后，当你编译它的时候，内核可以直接使用页高速缓存中的页，而不需要重新从磁盘读取该文件了。因为用户往往会反复读取或操作同一个文件，所以页高速缓存能减少大量的磁盘操作。

15.1 页高速缓存

从它的名字就可以看出来，页高速缓存缓存的是页面。缓存中的页来自对正规文件、块设备文件和内存映射文件的读写，如此一来，页高速缓存内就包含了最近被访问过的文件的全部页面。

在执行I/O操作前，比如read()操作[⊖]，内核会检查数据是否已经在页高速缓存中了，如果所需数据确实在高速缓存中，那么内核就可以马上从缓存中得到所需的页，而不需要从磁盘读取数据。

address_space对象

一个物理页可能由多个不连续的物理磁盘块[⊜]组成。也正是由于页面中映射的磁盘块不一定连续，所以在页高速缓存中检测特定数据是否已被缓存是件非常困难的工作。因此不能用设备名称和块号来做页高速缓存中数据的索引，要不然这可是最简单的解决办法。

另外，Linux页高速缓存对被缓存页的范围定义非常宽。实际上，在最初SystemV Release 4引入页高速缓存时，仅仅只缓存文件系统的数据，所以SVR4的页高速缓存使用它的等价文件对象（被称为 vnode结构体）管理页高速缓存。Linux页高速缓存的目标是缓存任何基于页的对象，这包含各种类型的文件和各种类型的内存映射。

为了满足普遍性要求，Linux页高速缓存使用address_space结构体描述页高速缓存中的页面。该结构定义在文件<linux/fs.h>中，下面给出具体形式：

```
struct address_space {
   struct inode                    *host;          /*拥有节点*/
   struct radix_tree_root          page_tree;      /*包含全部页面的radix 树 */
   spinlock_t                      tree_lock;      /*保护page_tree的自旋锁*/
   unsigned int                    i_mmap_writable; /* VM_SHARED 计数*/
   struct prio_tree_root           i_mmap;         /*私有映射链表*/
   struct list_head                i_mmap_nonlinear; /* VM_NONLINEAR 链表*/
   spinlock_t                      i_mmap_lock;    /* 保护i_mmap的自旋锁 */
   atomic_t                        truncate_count; /* 截断计数*/
   unsigned long                   nrpages;        /* 页总数*/
   pgoff_t                         writeback_index; /* 回写的起始偏移*/
   struct address_space_operations *a_ops;         /* 操作表*/
   unsigned long                   flags;          /* gfp_mask 掩码与错误标识*/
   struct backing_dev_info         *backing_dev_info; /*预读信息*/
   spinlock_t                      private_lock;   /*私有address_space锁*/
   struct list_head                private_list;   /*私有address_space链表*/
   struct address_space            *assoc_mapping; /*相关的缓冲*/
};
```

其中i_mmap字段是一个优先搜索树，它的搜索范围包含了在address_space中所有共享的与私有的映射页面。优先搜索树是一种巧妙的将堆与radix树结合[⊝]的快速检索树。address space空间大小由nrpages字段描述。

address_space结构往往会和某些内核对象关联。通常情况下，会与一个索引节点（inode）关

⊖ 正如在第12章中看到的，并非是read()和write()系统调用执行实际的页I/O操作，而是通过特定文件系统提供的操作：file->f_op->read() 和 file->f_op->write()。

⊜ 比如，x86体系结构中一个物理页的大小是4KB，而大多数文件系统的磁盘块大小仅仅512字节，所以8个块才可以填满一个页面。另外，因为文件本身可能分布在磁盘各个位置，所以页面中映射的块也不需要连续。

⊝ 在内核中采用radix优先搜索树是由Edward M. McCreight 1995年5月于SIAM 计算机杂志第14期，第2集，257~276页中提出的。

联，这时host域就会指向该索引节点；如果关联对象不是一个索引节点的话，比如address_space和swapper关联时，host域会被置为NULL。

a_ops域指向地址空间对象中的操作函数表，这与VFS对象及其操作表关系类似，操作函数表定义在文件<linux/fs.h>中，由address_space_operations结构体来表示：

```
struct address_space_operations {
        int (*writepage)(struct page *, struct writeback_control *);
        int (*readpage)(struct file *, struct page *);
        int (*sync_page)(struct page *);
        int (*writepages)(struct address_space *, struct writeback_control *);
        int (*set_page_dirty)(struct page *);
        int (*readpages)(struct file *, struct address_space *,
                        struct list_head *, unsigned);
        int (*prepare_write)(struct file *, struct page *, unsigned, unsigned);
        int (*commit_write)(struct file *, struct page *, unsigned, unsigned);
        sector_t (*bmap)(struct address_space *, sector_t);
        int (*invalidatepage) (struct page *, unsigned long);
        int (*releasepage) (struct page *, int);
        int (*direct_IO)(int, struct kiocb *, const struct iovec *,
                        loff_t, unsigned long);
};
```

这里面readpage()和writepage()两个方法最为重要。我们下面就来看看一个页面的读操作会包含哪些步骤。

首先，一个address_space对象和一个偏移量会被传给readpage()方法，这两个参数用来在页高速缓存中搜索需要的数据：

```
page = find_get_page(mapping ,index);
```

mapping是指定的地址空间，index是文件中的指定位置。

如果搜索的页并没在高速缓存中，那么内核将分配一个新页面，然后将其加入到页高速缓存中。

```
struct page *cached_page;
int error;

cached_page = page_cache_alloc_cold(mapping);
if(!cached_page)
        /*内存分配出错*/
error = add_to_page_cache_lru(cached_page,mapping,index,GFP_KERNEL);
if(error)
        /*页面被加入到页面高速缓存时，出错*/
```

最后，需要的数据从磁盘被读入，再被加入页高速缓存，然后返回给用户：

```
error= mapping->a_ops->readpage(file,page);
```

写操作和读操作有少许不同。对于文件映射来说，当页被修改了，VM仅仅需要调用

```
SetPageDirty(page);
```

内核会在晚些时候通过 writepage()方法把页写出。对特定文件的写操作比较复杂——它的代码

在文件mm/filemap.c中，通常写操作路径基本上要包含以下各步：

```
page = __grab_cache_page(mapping,index,&cached_page,&lru_pvec);
status = a_ops->prepare_write(file,page,offset,offset+bytes);
page_fault = filemap_copy_from_user(page,offset,buf,bytes);
status = a_ops->commit_write(file,page,offset,offset+bytes);
```

首先，在页高速缓存中搜索需要的页，如果需要的页不在高速缓存中，那么内核在高速缓存中新分配一空闲项；下一步，prepare_write()方法被调用，创建一个写请求；接着数据被从用户空间拷贝到了内核缓冲；最后通过commit_write()函数将数据写入磁盘。

因为所有的页I/O操作都要执行上面这些步骤，这就保证了所有的页I/O操作必然都是通过页高速缓存进行的。因此，内核也总是试图先通过页高速缓存来满足所有的读请求。如果在页高速缓存中未搜索到需要的页，则内核将从磁盘读入需要的页，然后将该页加入到页高速缓存中；对于写操作，页高速缓存更像是一个存储平台，所有要被写出的页都要被加入页高速缓存中。

15.2 基树

因为在任何页I/O操作前内核都要检查页是否已经在页高速缓存中了，所以这种频繁进行的检查必须迅速、高效，否则搜索和检查页高速缓存的开销可能抵消页高速缓存带来的好处（至少在缓存命中率很低的时候——搜索的开销足以抵消以内存代替磁盘进行检索数据带来的好处）。

正如在前一小节所看到的，页高速缓存通过两个参数——address_space对象和一个偏移量——进行搜索。每个address_space对象都有惟一的基树（radix tree），它保存在page_tree结构体中。基树是一个二叉树，只要指定了文件偏移量，就可以在基树中迅速检索到希望的数据。页高速缓存的搜索函数find_get_page()要调用函数radix_tree_lookup()，该函数会在指定基树中搜索指定页面。

基树核心代码的通用形式可以在文件lib/radix-tree.c中找到。另外要想使用基树，需要包含头文件<linux/radix_tree.h>。

以前的页散列表

在2.6版本以前，内核页高速缓存不是通过基树检索，而是通过一个维护了系统中所有页的全局散列表进行检索。对于给定的同一个键值，该散列表会返回一个双向链表。如果需要的页驻存在缓存中，那么链表中的一项就会与其对应。否则，页就不在页面高速缓存中，散列函数返回NULL。

全局散列表主要存在四个问题：

- 由于使用单个的全局锁保护散列表，所以即使在中等规模的机器中，锁的争用情况也会相当严重，造成性能受损。
- 由于散列表需要包含所有页高速缓存中的页，可是搜索需要的只是和当前文件相关的那些页，所以散列表包含的页面相比搜索需要页面要大得多。
- 如果散列搜索失败（也就是，给定的页不在页高速缓存中），执行速度比希望的要慢得多，这是因为检索必须遍历指定散列键值对应的整个链表。
- 散列表比其他方法会消耗更多的内存。

2.6版本内核中引入基于基树的页高速缓存来解决这些问题。

15.3 缓冲区高速缓存

现在Linux系统中已经不再有独立的缓冲区高速缓存了。但在2.2版本的内核中，存在两个独立的磁盘缓存：页高速缓存和缓冲区高速缓存。前者缓存页，后者缓存缓冲。两种缓存并不统一：一个磁盘块可以在两种缓存中同时存在，因此需要对两个缓存中的同一拷贝进行很麻烦的同步操作——还不用说要消耗额外的内存。

该情况存在于2.2和更早版本的内核中，但是从2.4版本的内核开始，统一了这两种缓存，现在Linux只有惟一的磁盘缓存——页高速缓存。

虽然如此，内核仍然需要在内存中使用缓冲来表示磁盘块，幸好，缓冲是用页映射块的，所以它正好在页高速缓存中。

15.4 pdflush后台例程

由于页高速缓存的缓存作用，写操作实际上会被延迟。当页高速缓存中的数据比后台存储的数据更新时，那么该数据就被称做脏数据。在内存中累积起来的脏页最终必须被写回磁盘。在以下两种情况发生时，脏页被写回磁盘：

- 当空闲内存低于一个特定的阈值时，内核必须将脏页写回磁盘，以便释放内存。
- 当脏页在内存中驻留时间超过一个特定的阈值时，内核必须将超时的脏页写回磁盘，以确保脏页不会无限期地驻留在内存中。

上面两种工作的目的完全不同。实际上，在老内核中，这是由两个独立的内核线程（请看后面章节）分别完成的。但是在2.6内核中，由一群[⊖]内核线程——pdflush后台回写例程——统一执行两种工作。说pdflush是“dirty page flush”的缩写是不正确的，不用去管这个让人混淆的名称，我们来看看这两个目标是如何具体实现的。

首先，pdflush线程在系统中的空闲内存低于一个特定的阈值时，将脏页刷新回磁盘。该后台回写例程的目的在于在可用物理内存过低时，释放脏页以重新获得内存。特定的内存阈值可以通过dirty_background_ratio sysctl系统调用设置。当空闲内存比阈值：dirty_ background_ratio还低时，内核便会调用函数wakeup_bdflush()[⊜]唤醒一个pdflush线程，随后pdflush线程进一步调用函数background_writeout()开始将脏页写回磁盘。函数background_ writeout()需要一个长整型参数，该参数指定试图写回的页面数目。函数background_writeout()会连续地写出数据，直到满足以下两个条件：

- 已经有指定的最小数目的页被写出到磁盘。
- 空闲内存数已经回升，超过了阈值dirty_background_ratio。

上述条件确保了pdflush操作可以减轻系统中内存不足的压力。回写操作不会在达到这两个条件前停止，除非pdflush写回了所有的脏页，没有剩下的脏页可再被写回了。

为了满足第二个目标，pdflush后台例程会被周期性唤醒（和空闲内存是否过低无关），将那些

⊖ 术语“群”通常在计算机科学中指的是一组可以并行执行的事情。

⊜ 是的，它的确命名错了，它应该被称为wakeup_bdflush()。原因请看后面关于这个调用的继承部分。

在内存中驻留时间过长的脏页写出，确保内存中不会有长期存在的脏页。如果系统发生崩溃，由于内存处于混乱之中，所以那些在内存中还没来得及写回磁盘的脏页就会丢失，所以周期性同步页高速缓存和磁盘非常重要。在系统启动时，内核初始化一个定时器，让它周期地唤醒pdflush线程，随后使其运行函数wb_kupdate()。该函数将把所有驻留时间超过百分之dirty_expire_centisecs秒的脏页写回。然后定时器将再次被初始化为百分之dirty_expire_ centisecs秒后唤醒pdflush线程。总而言之，pdflush线程周期地被唤醒并且把超过特定期限的脏页写回磁盘。

系统管理员可以在/proc/sys/vm中设置回写相关的参数，也可以通过sysctl系统调用设置它们。表15-1列出了与pdflush相关的所有可设置变量。

表15-1 pdflush设置

变 量	描 述
dirty_background_ratio	占全部内存的百分比。当内存中空闲页达到这个比例时，pdflush线程开始回写脏页
dirty_expire_centisecs	该数值以百分之一秒为单位，它描述超时多久的数据将被周期性执行的pdflush线程写出
dirty_ratio	占全部内存百分比，当一个进程产生的脏页达到这个比例时，就开始被写出
dirty_writeback_centisecs	该数值以百分之一秒为单位，它描述pdflush线程的运行频率
laptop_mode	一个布尔值，用于控制膝上型电脑模式，具体请见后续章节

pdflush线程的实现代码在文件mm/pdflush.c中，回写机制的实现代码在文件mm/page-writeback.c和fs/fs-writeback.c中。

15.4.1 膝上型电脑模式

膝上型电脑模式是一种特殊的页回写策略，该策略主要意图是将硬盘转动的机械行为最小化，允许硬盘尽可能长时间的停滞，以此延长电池供电时间。该模式可通过/proc/sys/vm/laptop_mode文件进行配置，通常，上述配置文件内容为零，也就是说膝上型电脑模式关闭，如果需要启用膝上型电脑模式，则向配置文件中写入1。

膝上型电脑模式的页回写行为与传统方式相比只有一处变化。除了当缓存中的页面太旧时要执行回写脏页以外，pdflush还会找准磁盘运转的时机，把所有其他的物理磁盘I/O、刷新脏缓冲等统统写回到磁盘，以便保证不会专门为了写磁盘而去主动激活磁盘运行。

上述回写行为变化要求dirty_expire_centisecs和dirty_writeback_centisecs两阈值必须设置的更大，比如10分钟。因为磁盘运转并不很频繁，所以用这样长的回写延迟才能保证膝上型电脑模式可以等到磁盘运转机会写入数据。

多数Linux发布板会在电脑接上电池或拔掉电池时，自动开启或禁止膝上型电脑模式以及其他需要的pdflush可调节开关。因此机器可在使用电池电源时自动进入膝上型电脑模式；而在插上交流电源时恢复到常规的页回写模式。

15.4.2 bdflush和kupdated

在2.6版本前，pdflush线程的工作是分别由bdflush和kupdated两个线程共同完成。当可用内存过低时，bdflush内核线程在后台执行脏页回写操作。与pdflush一样，它也有一组阈值参数，当系统

中空闲内存消耗到特定阈值以下时，bdflush线程就被wakeup_bdflush()函数唤醒。

bdflush和pdflush之间有两个主要区别。第一个区别在下一节还会谈到，系统中只有一个bdflush后台线程，而pdflush线程的数目却可以动态变化；第二个区别是bdflush线程基于缓冲，它将脏缓冲写回磁盘，相反，pdflush基于页面，它将整个脏页写回磁盘。当然，页面可能包含缓冲，但是实际I/O操作对象是整页，而不是块。因为页在内存中是更普遍和普通的概念，所以管理页相比管理块要简单。

因为只有在内存过低和缓冲数量过大时，bdflush例程才刷新缓冲，所以kupdated例程被引入，以便周期地写回脏页。它和pdflush线程的wb_kupdate()函数提供同样的服务。bdflush和kupdate内核线程现在完全被pdflush线程取代了。

15.4.3 避免拥塞的方法：使用多线程

使用bdflush线程最主要的一个缺点就是：bdflush仅仅包含了一个线程，因此很有可能在页回写任务很重时，造成拥塞。这是因为单一的线程有可能堵塞在某个设备的已拥塞请求队列上（正在等待将请求提交给磁盘的I/O请求队列），而其他设备的请求队列却没法得到处理。如果系统有多个磁盘和较强的处理能力，内核应该能使得每个磁盘都处于忙状态。不幸的是，即使还有许多数据需要回写，单个的bdflush线程也可能会堵塞在某个队列的处理上，不能使所有磁盘都处于饱和的工作状态。原因在于磁盘的吞吐量是有限的——不幸的是它实在太小。正是因为磁盘的吞吐量很有限，所以如果只有惟一线程执行页回写操作，那么这个线程很容易苦苦等待对一个磁盘上的操作。为了避免出现这种情况，内核需要多个回写线程并发执行，这样单个设备队列的拥塞就不会成为系统瓶颈了。

2.6内核通过使用多个pdflush线程来解决上述问题。每个线程可以相互独立地将脏页刷新回磁盘，而且不同的pdflush线程处理不同的设备队列。

通过一个简单的算法，线程的数目可以根据系统的运行时间进行调整。如果所有已存在的pdflush线程都已持续工作1秒以上，内核就会创建一个新的pdflush线程。线程数量最多不能超过MAX_PDFLUSH_THREADS——默认值是8。相反，如果一个pdflush线程睡眠超过1秒，内核就会终止该线程。线程的数量最少不得小于MIN_PDFLUSH_THREADS——默认值是2。pdflush线程数量取决于页回写的数量和拥塞情况，动态调整。如果所有存在的pdflush线程都忙着写回数据，那么一个新线程就会被创建，确保不会出现一个设备队列处于拥塞状态，而其他设备队列却在等待——而不是接收——回写数据的情况。如果堵塞消除，pdflush线程的数量便会自动减少，以便节约内存。

这种方式看起来很理想，但是如果每一个pdflush线程都挂起在同一个堵塞的队列上会怎么样呢？在这种情况下，多个pdflush线程的性能并不会比单个线程提高多少，反而会造成严重的内存浪费。为了克服这种负面影响，pdflush线程利用了拥塞避免策略，它们会积极地试图写回那些不属于拥塞队列的页面。这样一来，pdflush线程通过分派回写工作，阻止多个线程在同一个忙设备上纠缠。所以pdflush线程很“忙”——此时一个新线程会被创建——它们是真正地繁忙。

由于页回写性能的提高，以及pdflush线程的引入，2.6内核相比以前的内核，能够使更多磁盘饱和工作，即使处于繁重负载的情况下，pdflush线程也可以让多个磁盘保持高吞吐量。

15.5 小结

本章我们讲述了页高速缓存和页回写。了解了内核如何通过页缓存执行页I/O操作，以及写操作如何在页缓存中被延迟，并最终通过pdflush内核线程被写出到磁盘上。

通过最近几章的学习，你应该已经对内存与文件系统有了深刻认识，那么接下来我们将进入模块专题，去学习Linux内核如何被模块化，以及在运行时插入和删除内核代码的动态机制。

第16章

模　块

尽管Linux是“单块内核”（monolithic）的操作系统——也是说整个系统内核都运行于一个单独的保护域中，但是Linux内核是模块化组成的，它允许内核在运行时动态地向其中插入或从中删除代码。这些代码——包括相关的子例程、数据、函数入口和函数出口被一并组合在一个单独的二进制镜像中，即所谓的可装载内核模块中，或被简称为模块。支持模块的好处是基本内核镜像可以尽可能的小，因为可选的功能和驱动程序可以利用模块形式再提供。模块允许我们方便的删除和重新载入内核代码，也方便了调试工作。而且当热插拔新设备时，可通过命令载入新的驱动程序。

本章我们将探询内核的模块的奥秘，同时学习如何编写自己的内核模块。

Hello World!

与开发我们已经讨论过的大多数内核核心子系统不同，模块开发更接近编写新的应用系统，因为至少要在模块文件中具有入口点和出口点。

虽然编写“hello world”程序作为实例实属陈词滥调了，但它的确很让人喜爱。

女士们先生们，内核模块Hello，World出场了：

```
/*
 * hello.c-Hello, World! 我们的第一个内核模块
 */

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

/*
 * hello_init——初始化函数，当模块装载时被调用，如果成功装载则返回零，否
 * 则返回非零值
 */
static int hello_init(void)
{
        printk(KERN_ALERT "I bear a charmed life.\n");
        return 0;
}

/*
 * hello_exit——退出函数,当模块卸载时被调用
 */
```

```
static void hello_exit(void)
{
        printk(KERN_ALERT "Out, out, brief candle!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Shakespeare");
```

这大概是我们所能见到的最简单的内核模块了，hello_init()函数是模块的入口点，它通过module_init()例程注册到系统中，在模块装载时被调用。调用module_init()实际上不是真正的函数调用，而是一个宏调用，它惟一的参数便是模块的初始化函数。模块的所有初始化函数必须符合下面的形式：

```
int my_init(void);
```

因为init函数通常不会被外部函数直接调用，所以不必导出该函数，故它可被标记为static类型。

init函数会返回一个int型数值，如果初始化（或你的init函数想作的事情）顺利完成，那么它的返回值为零，否则失败的话，则返回一个非零值。

Hello_exit()函数是模块的出口函数，它由module_exit()例程注册到系统。在模块从内存卸载时，内核便会调用hello_exit()。退出函数可能会在返回前负责清理资源，以保证硬件处于一致状态；或者做其他别的一些操作。在退出函数返回后，模块就被卸载了。

退出函数必须符合以下形式：

```
void my_exit(void);
```

与init函数的原因一样，你也可以标记其为static.

如果上述文件被静态地编译到内核映像中，那么退出函数将不被包含，而且永远都不会被调用（因为如果不是编译成模块的话，那么代码就不需从内核中卸载）。

MODULE_LICENSE()宏用于指定模块的版权。如果载入非GPL模块到系统内存,则会在内核中设置被污染标识——这个标识只起到记录信息的作用。不过如果开发者提交的bug报告中含有被污染标识，那么报告的信用无疑会降低。另外要补充的是,非GPL模块不能调用GPL-only符号(请看本章后续小节)。

最后还要说明MODULE_AUTHOR()宏指定了代码作者,它完全是用作信息记录目的。

16.1　构建模块

在2.6内核中，由于采用了新的“kbuild”构建系统，现在构建模块相比从前更加容易。构建过程中的第一步是决定在那里管理模块源码。你可以把模块源码加入到内核源代码树中——或者是作为一个补丁或者是最终把你的代码合并到正式的内核代码树中；另一种可行的方式是在内核源代码树之外维护和构建你的模块源码。

16.1.1　放在内核源代码树中

最理想的情况莫过于你的模块正式成为Linux内核的一部分，这样就会被存放在内核源代码树

中。把你的模块代码正确的置于内核中，开始的时候难免需要更多的维护，但这样通常是一劳永逸的解决之道。

首先你要清楚你的模块应在内核源代码树中处于何处。设备驱动程序存放在内核源码树根目录drivers/的子目录下，在其内部，设备驱动文件被进一步按照类别、类型或特殊驱动程序等更有序地组织起来。如字符设备存在于drivers/char/目录下，而块设备存放在drivers/block/目录下，USB设备则存放在drivers/usb/目录下。文件的具体组织规则并非需绝对墨守成规，不容打破，你可看到许多USB设备也属于字符设备。但是不管怎么样，这些组织关系对我们来说相当容易理解，而且也便于遵守。

假定你有一个字符设备,而且希望将它存放在drivers/char/目录下，那么你要注意，在该目录下同时会存在大量的C源代码文件和许多其他目录。所以对于仅仅只有一两个源文件的设备驱动程序，可以直接存放在该目录下，但如果驱动程序包含许多源文件和其他辅助文件，那么可以创建一个新子目录。这期间并没有什么金科玉律。

假设你想建立自己代码的子目录，你的驱动程序是一个钓鱼杆和计算机的接口，名为Fish Master XL 2000 Titanium，那么你应在/drivers/char/目录下建立一个名为fishing的子目录。

现在你需要向drivers/char/下的Makefile文件中添加一行。编辑drivers/char/Makefile加入:

```
obj-m += fishing/
```

这行编译指令告诉模块构建系统在编译模块时需要进入fishing/子目录中。更可能发生的情况是，你的驱动程序的编译取决于一个特殊配置选项；比如，可能的CONFIG_FISHING_POLE(请看本章后面的小节，它会告诉你如何加入一个新的编译选项的)。如果这样，你需要用下面的指令代替刚才那条指令：

```
obj-$(CONFIG_FISHING_POLE) += fishing/
```

最后，在drivers/char/fishing/下，需要添加一个新的Makefile文件，其中需要有下面这行：

```
obj-m += fishing.o
```

一切就绪了，此刻构建系统运行就将会进入fishing/目录下，并且将fishing.c 编译为fishing.ko模块。你没看错，虽然你写的扩展名是.o，但是模块被编译后的扩展名却是.ko。

再一个可能，要是你的钓鱼杆驱动程序编译时有编译选项，那么你可能需要这么来做：

```
obj-$(CONFIG_FISHING_POLE) += fishing.o
```

以后，假如你的钓鱼杆驱动程序需要更加智能化——它可以自动检测钓鱼线，这可是最新的鱼杆“必备要求”。这时驱动程序源文件可能就不再只有一个了。别怕，朋友，你只要把你的Makefile作如下修改就可搞定：

```
obj-$(CONFIG_FISHING_POLE) += fishing.o
fishing-objs := fishing-main.o fishing-line.o
```

这样fishing-main.c和fishing-line.c就一起被编译和连接到了fishing.ko模块内。

最后一个注意事项，你可能需要在构建文件时需要额外的编译标记，如果这样，只需在Makefile中添加如下指令：

```
EXTRA_CFLAGS += -DTITANIUM_POLE
```

如果喜欢把源文件置于drivers/char/目录下，并且不建立新目录。那么你要做的便是将前面提到的行（也就是原来处于drivers/char/fishing/下你自己的Makefile中的）都加入到drivers/char/Makefile中。

开始编译吧，运行内核构建过程和原来一样。如果模块编译取决于配置选项，比如有CONFIG_FISHING_POLE约束，那么在编译前首先要确保选项被允许。

16.1.2 放在内核代码外

如果你喜欢脱离内核源代码树来维护和构建你的模块，把自己作为一个圈外人，那你要做的就是在你自己的源代码树目录中建立一个Makefile文件，它只需要一行指令：

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

就可把fishing.c编译成fishing.ko。如果你有多个源文件，那么用两行就足够了：

```
obj-m := fishing.o
fishing-objs := fishing-main.o fishing-line.o
```

这样一来，fishing-main.c和fishing-line.c就一起被编译和连接到fishing.ko模块内。

模块在内核内或内核外构建的最大区别在于构建过程。当你的模块在内核源代码树外围时，你必须告诉make如何找到内核源代码文件和基础Makefile文件。不过要完成这个工作同样不难：

```
make -C /kernel/source/location SUBDIRS=$PWD modules
```

/ kernel/source/location 是你以配置的内核源代码树。回想一下，不要把要处理的内核源代码树放在/usr/src/linux下，而要移到你的home目录下某个方便访问的地方。

16.2 安装模块

编译后的模块将被装入到目录/lib/modules/*version*/kernel/下。比如，如果使用的是2.6.10内核，而且你将你的模块源代码直接放在drivers/char/下，那么编译后的钓鱼杆驱动程序的存放路径将是：/lib/modules/2.6.10/kernel/drivers/char/fishing.ko。

下面的构建命令用来安装编译的模块到合适的目录下：

```
make modules_install
```

通常需要以root权限运行。

16.3 产生模块依赖性

Linux模块之间存在依赖性，也就是说钓鱼模块依赖于鱼饵模块，那么当载入钓鱼模块时，鱼饵模块会被自动载入。这里需要的依赖信息必须事先生成。多数Linux发布版都能自动产生这些依赖关系信息，而且在每次启动时更新。若想产生内核依赖关系的信息，root用户可运行命令：

```
depmod
```

为了执行更快的更新操作，可以只为新模块生成依赖信息，而不是生成所有的依赖关系，这时root用户可运行命令：

```
depmod -A
```

模块依赖关系信息存放在/lib/modules/*version*/modules.dep文件中。

16.4 载入模块

载入模块最简单的方法是通过insmod命令。这是个功能很有限的命令，它的作用就是请求内

核载入你指定的模块。insmod程序不执行任何依赖性分析或进一步的错误检查。它用法简单，以root身份运行命令：

```
insmod module
```

需要载入的模块名称由参数module指定，比如装载钓鱼杆模块，那你就执行命令：

```
insmod fishing
```

类似地，卸载一个模块，你可使用rmmod命令，它同样需要以root身份运行：

```
rmmod module
```

比如，rmmod fishing将卸载钓鱼杆模块。

这两个命令简单是简单，但是它们一点也不智能。好在系统为我们提供了一个更先进的工具modprobe，它提供了模块依赖性分析，错误智能检查，错误报告以及许多其他功能和选项。强烈建议大家用这个命令。

```
modprobe module [ module parameters ]
```

其中,参数*module*指定了需要载入的模块名称，后面的参数将在模块加载时传入内核。(请看后面章节中对模块参数的讨论)。

modprobe命令不但会加载指定的模块，而且会自动加载任何它所依赖的有关模块。所以说它是加载模块的最佳技术。

modprobe命令也可用来从内核中卸载模块，当然这也需要以root身份运行。

```
modprobe -r modules
```

参数*modules*指定一个或多个需要卸载的模块。与rmmod命令不同，modprobe也会卸载给定模块所依赖的相关模块，其前提是这些相关模块没有被使用。

Linux用户手册第八部分提供了上述命令的使用参考，里面包括了命令选项和用法。

16.5 管理配置选项

在前面一节中我们看到只要设置了CONFIG_FISHING_POLE配置选项，钓鱼杆模块就将被自动编译。虽然配置选项在前面已经讨论过了，但这里我们将以钓鱼杆例子，再看看一个新的配置选项如何被加入。

拜2.6内核中新引入的“kbuild”系统所赐，加入一个新配置选项现在可以说是易如反掌。你所需做的全部就是向Kconfig文件中添加一项——用以对应内核源码树。对驱动程序而言，Kconfig通常和源代码处于同一目录。如果钓鱼杆驱动程序在目录drivers/char/下，那么你便会发现drivers/char/Kconfig同时存在。

如果你建立了一个新子目录，而且也希望Kconfig文件存在于该目录中的话，那么必须在一个已存在的Kconfig文件中将它引入——你需要加入下面一行指令：

```
source "drivers/char/fishing/Kconfig"
```

这里所谓存在的Kconfig文件可能是drivers/char/Kconfig。

可以很方便地在Kconfig文件中加入一个配置选型，请看钓鱼杆模块的选项如下所示:

```
config FISHING_POLE
        tristate "Fish Master XL support"
        default n
```

```
help
  If you say Y here, support for the Fish Master XL 2000 Titanium with
  computer interface will be compiled into the kernel and accessible via
  device node. You can also say M here and the driver will be built as a
  module named fishing.ko.

  If unsure, say N.
```

配置选项第一行定义了该选项所代表的配置目标。注意CONFIG_前缀并不需要写上。

第二行声明选项类型为tristate，也就是说可以被编译进内核（Y），也可作为模块编译(M)，或者干脆不编译它(N)。如果编译选项代表的是一个系统功能，而不是一个模块，那么编译选项将用bool指令代替tristate，这说明它不允许被编译成模块。处于指令之后的引号内文字为该选项指定了名称。

第三行指定了该选项的默认选择，这里默认操作是不编译它。

Help指令目的是为该选项提供帮助文档。各种配置工具都可以按要求显示这些帮助。因为这些帮助是面向编译内核的用户和开发者，所以帮助内容简洁扼要。一般的用户通常不会编译内核，但如果他们想试试，也往往能理解配置帮助的意思。

除了上述选项以外，还存在其他选项。比如depends指令规定了在该选项被设置前，首先要设置的选项。假如依赖性不满足，那么该选项就被禁止。比如，如果你加入指令

```
depends on FISH_TANK
```

到配置选项中,那么就意味着在CONFIG_FISH_TANK被选择前，我们的钓鱼竿模块是不能被使用的。

select指令和depends类似，不同之处在于，只要是select指定了谁，它就会强行将被指定的选项打开。所以这个指令可不能向depends那样滥用一通，因为它会自动的激活其他配置选项的。它的用法和depend一样。比如：

```
select BAIT
```

意味着当CONFIG_ FISHING_POLE被激活时，配置选项CONFIG_BAIT必然一起被激活。

如果select和depends同时指定多个选项，那就需要通过&&指令来进行多选。使用depends时，你还可以利用叹号前缀来指明禁止某个选项。比如：

```
depends on DUMB_DRIVERS && !NO_FISHING_ALLOWED
```

就指定驱动程序安装要求打开CONFIG_DUMB_DRIVERS选项，同时要禁止CONFIG_NO_FISHING_ALLOWED选项。

tristate和bool选项往往会结合if指令一起使用，这表示某个选项取决于另一个配置选项。如果条件不满足，那么配置选项不但会被禁止，而且甚至不会显示在配置工具中，比如，要求配置系统只有在CONFIG_X86配置选项设置时才显示某选项。请看下面指令:

```
bool "Deep Sea Mode" if OCEAN
```

意思就是“deep sea mode”只有在CONFIG_OCEAN选项打开时才可见且有效。

If指令也可与default指令结合使用，强制只有在条件满足时default选项才有效。

配置系统导出了一些元选项（meta-option）以简化生成配置文件。比如选项CONFIG_EMBEDDED是用于关闭那些用户想要禁止的关键功能（比如要在嵌入系统中节省珍贵的内存）；

选项CONFIG_BROKEN_ON_SMP用来表示驱动程序并非多处理器安全的。通常该项不应设置，标记它的目的是以确保用户能知道该驱动程序的弱点。当然，新的驱动程序不应该使用该标志。最后要说明CONFIG_EXPERIMENTAL选项，它是一个用于说明某项功能尚在试验或处于beta版阶段的标志选项。该选项默认情况下关闭，同样，标记它的目的是为了让用户在使用驱动程序前明白潜在风险。

16.6 模块参数

Linux提供了这样一个简单框架——它可允许驱动程序声明参数，从而用户可以在系统启动或者模块装载时再指定参数值，这些参数对于你的驱动程序属于全局变量。值得一提的是模块参数同时也将出现在sysfs文件系统中（请参见第17章），这样以来，无论是生成模块参数，或者是管理模块参数的方式都变的灵活多样了。

定义一个模块参数可通过宏module_param()完成：

```
module_param(name, type, perm);
```

参数name既是用户可见的参数名，也是模块中存放模块参数的变量名。参数type则存放了参数的类型，它可以是byte、short、ushort、int、uint、long、ulong、charp、bool或invbool，它们分别代表字节型、短整型、无符号短整形、整型、无符号整型、长整形、无符号长整型、字符指针、布尔型，以及应用户要求转换得来的布尔型。其中byte类型存放在char 类型变量中，boolean类型存放在int变量中，其余的类型都一致对应C语言的变量类型。 最后一个参数perm指定了模块在sysfs文件系统下对应文件的权限，该值可以是八进制的格式，比如0644（所有者可以读写，组内可以读，其他人可读），或是S_Ifoo的定义形式，比如 S_IRUGO | S_IWUSR （任何人可读，user可写），如果该值是零，则表示禁止所有的sysfs项。

上面的宏其实并没有定义变量，你必须在使用该宏前进行变量定义。通常使用类似下面的语句完成定义。

```
/* 在模块参数控制下，我们允许在钓鱼杆上用活鱼饵 */
static int allow_live_bait = 1;                /* 默认功能允许 */
module_param(allow_live_bait, bool, 0644);  /* 一个Boolean类型 */
```

这个值处于模块代码文件外部，换句话说，allow_live_bait是个全局变量。

有可能模块的外部参数名称不同于它对应的内部变量名称，这时就该使用宏module_param_named()定义了：

```
module_param_named(name, variable, type, perm);
```

参数name是外部可见的参数名称，参数variable是参数对应的内部全局变量名称。比如：

```
static unsigned int max_test = DEFAULT_MAX_LINE_TEST;
module_param_named(maximum_line_test, max_test, int, 0);
```

通常，需要用一个charp类型来定义模块参数（一个字符串），内核将用户提供的这个字符串拷贝到内存，而且将你的变量指向该字符串。比如：

```
static char *name;
module_param(name, charp, 0);
```

如果需要，也可使内核直接拷贝字符串到你指定的字符数组。宏module_param_string()可完成上述

任务：

```
module_param_string():
module_param_string(name, string, len, perm);
```

这里参数name为外部参数名称，参数string 是对应的内部变量名称，参数len是string命名缓冲区的长度（或更小的长度，但是没什么太大的意义），参数perm是sysfs文件系统的访问权限（如果为零，则表示完全禁止sysfs项），比如：

```
static char species[BUF_LEN];
module_param_string(specifies, species, BUF_LEN, 0);
```

你可接受逗号分割的参数序列，这些参数序列可通过宏module_param_array()在C数组中：

```
module_param_array_named():
module_param_array(name, type, nump, perm);
```

参数name仍然是外部参数以及对应内部变量名，参数type是数据类型，参数perm是sysfs文件系统访问权限，这里新参数是nump，它是一个整型指针，该整形存放数组项数。注意由参数name指定的数组必须是静态分配的，内核需要在编译时确定数组大小，从而保证不会造成溢出。该函数用法相当简单，比如：

```
static int fish[MAX_FISH];
static int nr_fish;
module_param_array(fish, int, &nr_fish, 0444);
```

你可以将内部参数数组命名区别于外部参数，这时需使用宏：

```
module_param_array_named():
module_param_array_named(name, array, type, nump, perm);
```

其中参数和其他宏一致。

最后，你可使用MODULE_PARM_DESC()描述你的参数：

```
static unsigned short size = 1;
module_param(size, ushort, 0644);
MODULE_PARM_DESC(size, "The size in inches of the fishing pole " \
                 "connected to this computer.");
```

上述所有宏被定义在<linux/moduleparam.h>文件中。

16.7　导出符号表

模块被载入后，就会动态连接到了内核。注意，它与用户空间中的动态连接库类似，只有当被显式导出后的外部函数，才可以被动态库调用。在内核中，导出内核函数需要使用特殊的指令：EXPORT_ SYMBOL() 和EXPORT_SYMBOL_GPL()。

导出的内核函数可以被模块调用，而未导出的函数模块则无法被调用。模块代码的链接和调用规则相比核心内核镜像中的代码而言，要更加严格。核心代码在内核中可以调用任意非静态接口，因为所有的核心源代码文件被链接成了同一个镜像。当然，被导出的符号表所含的函数必然也是非静态的。

导出的内核符号表被看作是导出的内核接口，甚至称为内核API。

导出符号相当简单，在声明函数后，紧跟上EXPORT_SYMBOL()指令就搞定了，比如：

```
/*
 * get_pirate_beard_color - 返回当前pirate胡须的颜色，
 * pirate是该函数的全局变量，颜色定义在文件<linux/beard_colors.h>中
 */
int get_pirate_beard_color(void)
{
        return pirate->beard->color;
}
EXPORT_SYMBOL(get_pirate_beard_color);
```

假定get_pirate_beard_color()同时也定义在一个可访问的头文件中，那么任何模块现在都可以访问它。

有一些开发者希望自己的接口仅仅对GPL兼容的模块可见，内核连接器使用MODULE_LICENSE()宏可满足这个要求。如果你希望先前的函数仅仅对标记为GPL协议的模块可见，那么你就需要用：

```
EXPORT_SYMBOL_GPL(get_pirate_beard_color);
```

如果你的代码被配置为模块，那么就必须确保当它被编译为模块时所用的全部接口都已被导出，否则就会产生连接错误（而且模块不能成功编译）。

16.8 小结

本章介绍了编写、编译、装载和卸载模块。我们不但揭示了模块是什么，还介绍了Linux——尽管它是个单块（monolithic）内核——如何可动态装载模块代码。另外又讨论了模块参数和导出符号表。作为例子，我们引用了一个钓鱼杆模块（一个很有想像力的设备，如果你真有的话）来诠释编写模块代码与利用参数为它添加额外属性。

在下一章中，我们将讲述kobjects和 sysfs文件系统，它们对于设备驱动和模块来说是最重要的一项新技术。

第17章

kobject与sysfs

2.6内核增加了一个引人注目的新特性——统一设备模型(device model)。设备模型提供了一个独立的机制专门来表示设备，并描述其在系统中的拓扑结构，从而使得系统具有以下优点：

- 代码重复最小化。
- 提供诸如引用计数这样的统一机制。
- 可以列举系统中所有的设备，观察它们的状态，并且查看它们连接的总线。
- 可以将系统中的全部设备结构以树的形式完整、有效地展现出来，包括所有的总线和内部连接。
- 可以将设备和其对应的驱动联系起来，反之亦然。
- 可以将设备按照类型加以归类，如分类为输入设备，而无需理解物理设备的拓扑结构。
- 可以沿设备树的叶子向其根的方向依次遍历，以保证能以正确顺序关闭各设备的电源。

最后一点是实现设备模型的最初动机。若想在内核中实现智能的电源管理，就需要来建立表示系统中设备拓扑关系的树结构。当在树上端的设备关闭电源时，内核必须首先关闭该设备节点以下的（处于叶子上的）设备电源。比如内核需要先关闭一个USB鼠标，然后才可关闭USB控制器；同样内核也必须在关闭PCI总线前先关闭USB控制器。简而言之，若要准确而又高效地完成上述电源管理目标，内核无疑需要一颗设备树。

17.1 kobject

设备模型的核心部分就是kobject，它由struct kobject结构体表示,定义于头文件〈linux/kobject.h〉中。kobject类似于C#或Java这些面向对象语言中的object(对象)类，提供了诸如引用计数、名称和父指针等字段，可以创建对象的层次结构。

不说那么多了，看下面具体结构：

```
struct kobject {
        char                    *k_name;
        char                    name[KOBJ_NAME_LEN];
        struct kref             kref;
        struct list_head        entry;
        struct kobject          *parent;
        struct kset             *kset;
        struct kobj_type        *ktype;
        struct dentry           *dentry;
};
```

k_name指针指向kobject名称的起始位置，如果名称长度小于KOBJ_NAME_LEN ——KOBJ_NAME_LEN当前为20个字节，那么该kobject的名称便存放到name数组中，k_name指向数

组头；如果名称长度大于KOBJ_NAME_LEN，则动态分配一个足够大的缓冲区来存放kobject的名称，这时k_name指向缓冲区。

parent指针指向kobject的父对象。因此，kobject就会在内核中构造一个对象层次结构，并且可以将多个对象间的关系表现出来。就如你将看到的，这便是sysfs的真正面目：一个用户空间的文件系统，用来表示内核中kobject对象的层次结构。

dentry指针指向dentry结构体，在sysfs中该结构体就表示这个kobject，当然假设该kobject已反映在sysfs中了。

kref、ktype和kset这些字段指向了kobject所需的其他结构体，entry字段与kset一起联合使用。这些结构体和它们的用法后面将作简单介绍。

kobject通常是嵌入到其他结构中的，其单独意义其实并不大。相反，那些更为重要的结构体，比如struct cdev中才真正需要用到kobject结构。

```
/* cdev structure - 该对象代表一个字符设备 */
struct cdev {
        struct kobject          kobj;
        struct module           *owner;
        struct file_operations  *ops;
        struct list_head        list;
        dev_t                   dev;
        unsigned int            count;
};
```

当kobject被嵌入到其他结构中时，该结构便拥有了kobject提供的标准功能。更重要的一点是，嵌入kobject的结构体可以成为对象层次架构中的一部分。比如cdev结构体就可通过其父指针cdev->kobj->parent 和链表cdev->kobj->entry 来插入到对象层次结构中。

17.2 ktype

kobject 对象被关联到一种特殊的类型，即ktype。ktype由kobj_type结构体表示，定义于头文件<linux/kobject.h>中：

```
struct kobj_type {
        void (*release)(struct kobject *);
        struct sysfs_ops        *sysfs_ops;
        struct attribute        **default_attrs;
};
```

ktype是为了描述一族kobject所具有的普遍特性。因此，不再需要每个kobject都分别定义自己的特性，而是将这些普遍的特性在ktype结构中一次定义，然后所有“同类”的kobject都能共享一样的特性。

release指针指向在kobject引用计数减至零时要被调用的析构函数。该函数负责释放所有kobject使用的内存和其他相关清理工作。

sysfs_ops变量指向sysfs_ops结构体。该结构体描述了sysfs文件读写时的特性。有关其细节参见17.8.2节。

最后，default_attrs指向一个attribute结构体数组。这些结构体定义了该kobject相关的默认属性。

属性描述了给定对象的特征，如果该kobject被导出到sysfs中，那么这些属性都将相应地作为文件而导出。数组中的最后一项必须为NULL。

17.3 kset

kset是kobject对象的集合体。把它看成是一个容器，可将所有相关的kobject对象，比如“全部的块设备”置于同一位置。似乎kset与ktypes非常类似，好像没有多少实质内容。那么“为什么会需要这两个类似的东西呢”。ksets可把kobject集中到一个集合中，而ktype描述相关类型kobject所共有的特性，它们之间的重要区别在于：具有相同ktype的kobject可以分组到不同的ksets中。

kobject的kset指针指向相应的kset集合。kset集合由kset结构体表示，定义于头文件<linux/kobject.h>中：

```
struct kset {
        struct subsystem          *subsys;
        struct kobj_type          *ktype;
        struct list_head          list;
        struct kobject            kobj;
        struct kset_hotplug_ops   *hotplug_ops;
};
```

其中ktype指针指向集合（kset）中kobject对象的类型（ktype），list连接该集合（kset）中所有的kobject对象。kobj指向的koject对象代表了该集合的基类。hotplug_ops指向一个用于处理集合中kobject对象的热插拔操作的结构体。

最后，subsys指针指向该结构体相关的struct subsystem结构体。

17.4 subsystem

subsystem 在内核中代表高层概念，它是一个或多个ksets的大集合。虽然ksets包含kobject，而subsystems包含ksets,但是在subsystem中ksets之间的关联相比kset中kobject之间的关联要弱很多。要知道，在subsystem中的ksets只可以共享一些较宏观的普遍属性。

尽管subsystem有重要的作用，但相应的结构体却相当简单——struct subsystem：

```
struct subsystem {
        struct kset               kset;
        struct rw_semaphore       rwsem;
};
```

虽然subsystem结构体只指向一个kset，但是多个ksets可以通过其subsys指针指向一个subsystem。这种单向关系意味着不可能仅仅通过一个subsystem结构体就找到所有的ksets。

subsystem中的kset字段指向的是subsystem中的默认kset，它在对象层次结构中起到了粘合剂的作用。

subsystem结构体的rwsem字段是一个读写信号量（参见第9章），该信号量用来对subsystem和它的所有ksets进行并发访问保护。所有的ksets必须属于subsystem，因为它们使用该读写信号量去同步访问它们的内部数据。

17.5 别混淆了这些结构体

上文反复讨论的这一组结构体很容易混淆，这可不是因为它们数量繁多（其实只有四个），也不是它们太复杂（它们都相当简单），而是由于它们内部相互交织。要了解kobject，很难只讨论其中一个结构而不涉及其他相关结构。然而在这些结构的相互作用下，会更有助于深刻理解它们之间的关系。

这里最重要的是kobject，它由struct kobject表示。kobject为我们引入了诸如引用计数(reference counting)、父子关系和对象名称等基本对象道具，并且是以一个统一方式提供这些功能。不过kobject本身意义并不大，通常情况下它需要被嵌入到其他数据结构中。

kobject与一个特别的ktype对象关联，ktype由struct kobj_type结构体表示，在kobject中ktype字段指向该对象。ktype定义了一些kobject相关的默认特性：析构行为（反构造功能）、sysfs行为(sysfs的操作表）以及其他一些默认属性。

kobject又被归入了称作kset的集合，kset集合由struct kset结构体表示。kset提供了两个功能。第一，其中嵌入的kobject作为kobject组的基类。第二，kset将相关的kobject集合在一起。在sysfs中，这些相关的kobject将以独立的目录出现在文件系统中。这些相关的目录，也许是给定目录的所有子目录，它们可能处于同一个kset中。

subsystem表示得更为宏观，它是包含kset的集合，由struct subsystem结构体表示。sysfs中的根级目录映射的便是subsystem。

图17-1描述了这些数据结构的内在关系。

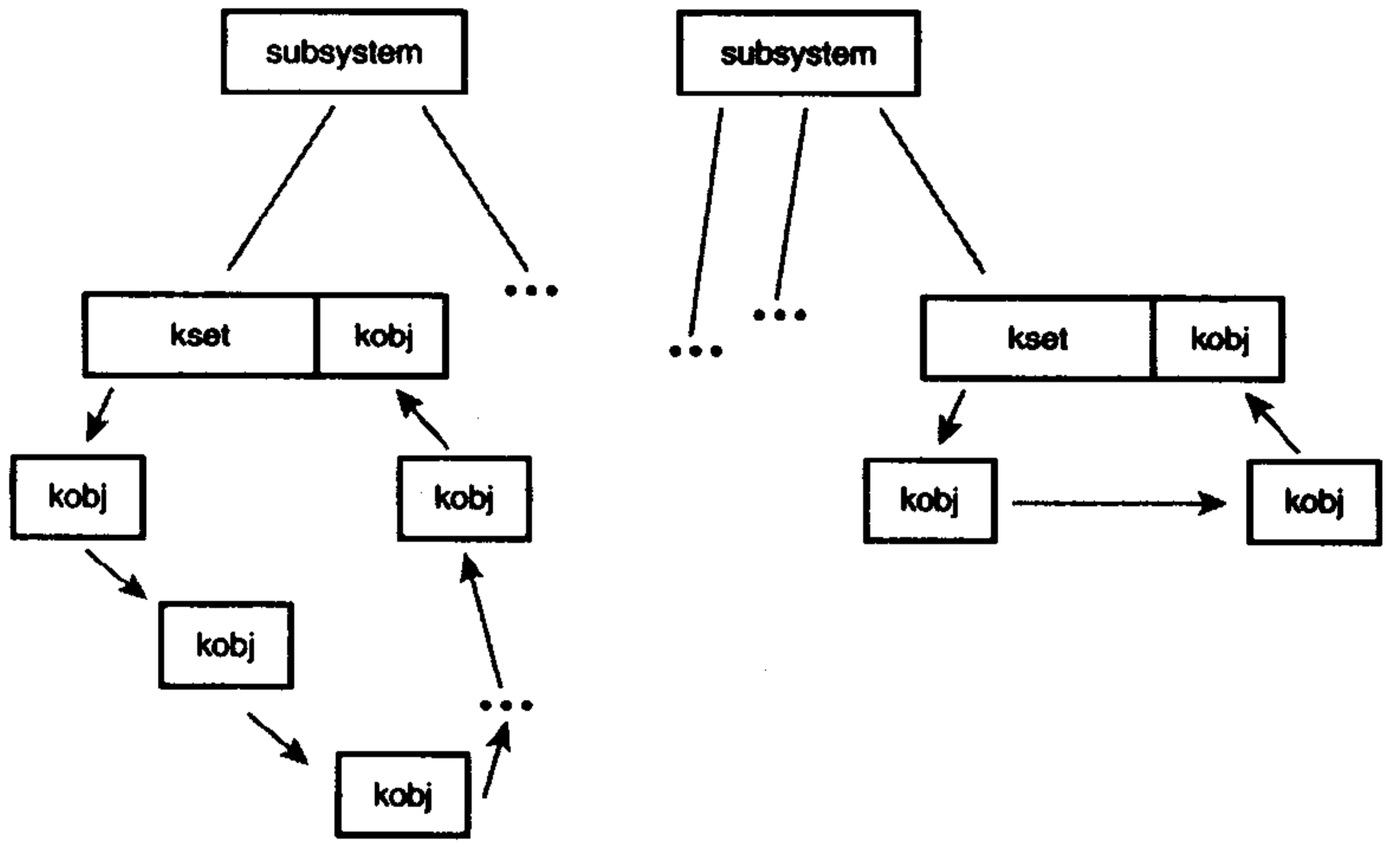


图17-1 kobject、kset和subsystem的内在关系

17.6 管理和操作kobject

当了解了kobject的内部基本细节后，我们来看管理和操作它的外部接口了。多数时候，驱动程序开发者并不必直接处理kobject，因为kobject是被嵌入到一些特殊类型结构体中的(就如在字符设备结构体中看到的情形)，而且会由相关的设备驱动程序在“幕后”管理。即便如此，kobject并不是有意在隐藏自己，它可以出现在设备驱动代码中，或者你可以在设备驱动子系统本身中使用它。

使用kobject的第一步需要先来声明和初始化它。kobject通过函数kobject_init进行初始化，该函数定义在文件<linux/kobject.h>中：

```
void kobject_init(struct kobject *kobj);
```

该函数的惟一参数就是需要初始化的kobject对象，在调用初始化函数前，kobject必须清空。这个工作往往会在kobject所在的上层结构体初始化时完成。如果kobject未被清空，那么只需要调用memset()即可：

```
memset(kobj, 0, sizeof (*kobj));
```

在清零后，就可以安全地初始化parent和kset字段。例如，

```
kobj = kmalloc(sizeof (*kobj), GFP_kERNEL);
if (!kobj)
        return -ENOMEM;
memset(kobj, 0, sizeof (*kobj));
kobj->kset = kset;
kobj->parent = parent_kobj;
kobject_init(kobj);
```

初始化后，必须采用kobject_set_name()函数为kobject设置名称：

```
int kobject_set_name(struct kobject * kobj, const char * fmt, ...);
```

该函数利用了类似printf()和printk()风格的可变参数列队来为kobject设置名称。正如你看到的，k_name指向了kobject的名称。如果名称字符长度够小的话，那么该名称存放于静态数组name，所以要注意不要毫无必要地为kobject设置很长的名字。

初始化了kobject并设置名称后，还需要为它设置kset字段以及可能的ktype字段（可选）。如果kset没有被提供，那么仅需要设置ktype，否则kset中的ktype字段将优先被使用。要是你疑惑为什么kobject也需要ktype字段，那就请加入社区吧。

17.7 引用计数

kobject的主要功能之一就是为我们提供了一个统一的引用计数系统。初始化后，kobject的引用计数设置为1。只要引用计数不为零，那么该对象就会继续保留在内存中，也可以说是被“钉住”了。任何包含对象引用的代码首先要增加该对象的引用计数，当代码结束后则减少它的引用计数。增加引用计数称为获得（getting）对象的引用，减少引用计数称为释放（putting）对象的引用。当引用计数减少到零时，对象便可以被销毁，同时相关内存也都被释放。

增加一个引用计数可通过kobject_get()函数完成：

```
struct kobject * kobject_get(struct kobject *kobj);
```

该函数正常情况下将返回一个指向kobject的指针，如果失败，则返回NULL指针。

减少引用计数通过kobject_put()完成：

```
void kobject_put(struct kobject *kobj);
```

如果对应的kobject的引用计数减少到零，则与该kobject关联的ktype中的析构函数将被调用。

kref

我们深入到引用计数系统的内部去看，会发现kobject的引用计数是通过kref结构体实现的，该

结构体定义在头文件<linux/kref.h>中：

```
struct kref {
        atomic_t refcount;
};
```

其中惟一的字段是用来存放引用计数的原子变量。那为什么采用结构体？这是为了便于进行类型检测。在使用kref前，必须先通过kref_init()函数来初始化它：

```
void kref_init(struct kref *kref)
{
        atomic_set(&kref->refcount, 1);
}
```

正如你所看到的，这个函数简单地将原子变量置1，所以kref一但被初始化，它表示的引用计数便固定为1。这点和kobject中的计数行为一致。

要获得对kref的引用，需要调用kref_get()函数:

```
void kref_get(struct kref *kref)
{
        WARN_ON(!atomic_read(&kref->refcount));
        atomic_inc(&kref->refcount);
}
```

该函数增加引用计数值，它没有返回值。减少对kref的引用，调用函数kref_put():

```
void kref_put(struct kref *kref, void (*release) (struct kref *kref))
{
        WARN_ON(release == NULL);
        WARN_ON(release == (void (*)(struct kref *))kfree);

        if (atomic_dec_and_test(&kref->refcount))
                release(kref);
}
```

该函数将使得引用计数减1，如果计数减少到零，则要调用作为参数提供的release()函数。注意WARN_ON()声明，提供的release()函数不能简单地采用kfree()，它必须是一个仅接收一个kref结构体作为参数的特有函数，而且还没有返回值。

开发者现在不必在内核代码中利用atmoic_t类型和简单的“get”、“put”这些封装函数来实现自己的引用计数。对开发者而言，在内核代码中最好的方法是利用kref类型和它相应的辅助函数，为自己提供一个通用的、正确的引用计数机制。

上述的所有函数定义与声明分别在文件lib/kref.c和文件<linux/kref.h>中。

17.8 sysfs

sysfs文件系统是一个处于内存中的虚拟文件系统，它为我们提供了kobject对象层次结构的视图。帮助用户能以一个简单文件系统的方式来观察系统中各种设备的拓扑结构。借助属性对象，kobject可以用导出文件的方式，将内核变量提供给用户读取或写入（可选）。

虽然设备模型的初衷是为了方便电源管理而提供的一种设备拓扑结构，但是sysfs是颇为意外

的收获。为了方便调试，设备模型的开发者决定将设备结构树导出为一个文件系统。这个举措很快被证明是非常明智的，首先sysfs代替了先前处于/proc下的设备相关文件；另外它为系统对象提供了一个很有效的视图。实际上，sysfs起初被称为driverfs，它早于kobject出现。最终sysfs使得我们认识到一个全新的对象模型非常有利于系统，于是kobject应运而生。今天所有2.6内核的系统都拥有sysfs文件系统，而且几乎都毫无例外地将其挂载。

sysfs的诀窍是把kobject对象与目录项（directory entry）紧密联系起来，这点是通过kobject对象中的dentry字段实现的。回忆第12章，dentry结构体表示目录项，通过连接kobject到指定的目录项上，无疑方便地将kobject映射到该目录上。从此，把kobject导出形成文件系统就变得如同在内存中构建目录项一样简单。 好了，kobjects其实已经形成了一棵树了——就是我们心爱的对象模型体系。由于kobject被映射到目录项，同时对象层次结构也已经在内存中形成了一棵树，因此sysfs的生成便水到渠成般的简单了。

图17-2是挂载于/sys目录下的sysfs文件系统的局部视图。

```
/sys
├── block/
│   ├── fd0
│   ├── hda
│   ├── dev
│   ├── device -> ../../devices/pci0000:00/0000:00:1f.1/ide0/0.0
│   ├── hda1
│   │   ├── dev
│   │   ├── size
│   │   ├── start
│   │   └── stat
│   ├── hda2
│   ├── hda3
│   ├── hda4
│   ├── hda5
│   ├── hda6
│   ├── queue
│   ├── hdc
│   ├── hdd
│   ├── loop0
│   ├── loop1
│   ├── loop2
│   ├── loop3
│   ├── loop4
│   ├── loop5
│   ├── loop6
│   ├── loop7
│   └── md0
├── bus/
├── class/
├── devices/
├── firmware/
└── power/
```

图17-2 /sys目录下的局部视图

sysfs的根目录下包含了七个目录：block、bus、class、devices、firmware、module和 power。block目录下的每个子目录都对应着系统中的一个块设备。反过来，每个目录下又都包含了该块设备的所有分区。bus目录提供了一个系统总线视图。class目录包含了以高层功能逻辑组织起来的系统设备视图。devices目录是系统中设备拓扑结构视图，它直接映射出了内核中设备结构体的组织层次。firmware目录包含了一些诸如ACPI、EDD、EFI等低层子系统的特殊树。power目录包含了系统范围的电源管理数据。

其中最重要的目录是devices，该目录将设备模型导出到用户空间。目录结构就是系统中实际的设备拓扑。其他目录中的很多数据都是将devices目录下的数据加以转换加工而得。比如，/sys/class/net/目录是以注册网络接口这一高层概念来组织设备关系的，在这个目中可能会有目录eth0，它里面包含的devices文件其实就是一个指回到devices下实际设备目录的符号连接。

随便看看任何你可访问到的Linux系统，这种系统设备视图相当准确和漂亮，而且可以看到class中的高层概念与devices中的低层物理设备，以及bus中的实际驱动程序之间互相联络是非常广泛的。可见，这种数据是开放的，换句话说，这是内核中维持系统的很好表示方式[⊖]时，认识到这点非常重要。

17.8.1 sysfs中添加和删除kobject

仅仅初始化kobject是不能自动将其导出到sysfs中的，想要把kobject导入sysfs，需要用到函数kobject_add():

```
int kobject_add(struct kobject *kobj);
```

kobject在sysfs中的位置取决于kobject在对象层次结构中的位置。如果kobject的父指针被设置，那么在sysfs中kobject将被映射为其父目录下的子目录，如果parent没有设置，那么kobject将被映射为kset->kobj中的子目录。如果给定的kobject中parent或kset字段都没有设置，那么就认为kobject没有父对象，所以就会被映射成sysfs下的根级目录。这往往不是你所需要的，所以在调用kobject_add()前parent或kset字段应该显式地设置。不管怎么样，sysfs中代表kobject的目录名字是由kobj->k_name指定的。

你不必分别调用kobject_init() 和kobject_add()，因为系统提供函数kobject_register()可帮助你一步到位:

```
int kobject_register(struct kobject *kobj)
```

该函数既初始化了给定的kobject对象，同时又将其加入到对象层次结构中。

从sysfs中删除一个kobject对应文件目录，需使用函数kobject_del():

```
void kobject_del(struct kobject *kobj);
```

类似地，函数kobject_unregister()包含了kobject_del()和kobject_put()两者的功能:

```
void kobject_unregister(struct kobject * kobj)
```

上述四个函数都定义于文件lib/kobject.c中，声明于头文件〈linux/kobject.h〉中。

17.8.2 向sysfs中添加文件

我们已经看到kobject被映射为文件目录，而且所有的对象层次结构都优雅的、一个不少的映射成sys下的目录结构。但是里面的文件是什么？sysfs仅仅是一个漂亮的树，但是没有提供实际数据的文件。

默认属性

默认的文件集合是通过kobject和kset中的ktype字段提供的。因此所有具有相同类型的kobject在

⊖ 如果你对sysfs感兴趣，可能也会对HAL感兴趣，它是一个硬件抽象层，可以在http://hal.freedesktop.org/找到它。HAL基于sysfs中的数据建立起了一个内存数据库，将class概念、设备概念和驱动概念联系到一起。在这些数据之上，HAL提供了丰富的API以使得应用程序更灵活。

它们对应的sysfs目录下都拥有相同的默认文件集合。kobj_type字段含有一个字段——default_attrs，它是一个attribute结构体数组。这些属性负责将内核数据映射成sysfs中的文件。

attribute结构体定义在文件<linux/sysfs.h>中：

```
/* attribute 结构体，内核数据映射成sysfs中的文件 */
struct attribute {
        char                *name;      /* 属性名称*/
        struct module       *owner;     /* 所属模块，如果存在*/
        mode_t              mode;       /* 权限*/
};
```

其中名称字段提供了该属性的名称，最终出现在sysfs中的文件名就是它。owner字段在存在所属模块的情况下指向其所属module结构体。如果一个模块没有该属性，那么该字段为 NULL。mode字段类型为mode_t，它表示了sysfs中该文件的权限。对于只读属性而言，如果是所有人都可读它，那么该字段设为S_IRUGO；如果只限于所有者可读，则该字段设置为S_IRUSR。同样对于可写属性，可能会设置该字段为S_IRUGO | S_IWUSR。sysfs中的所有文件和目录的uid与gid标志均为零。

虽然default_attrs列出了默认的属性，sysfs_ops字段则描述了如何使用它们。sysfs_ops字段指向了一个定义于文件<linux/sysfs.h>的同名的结构体：

```
struct sysfs_ops {
        /* 在读sysfs文件时该方法被调用 */
        ssize_t (*show) (struct kobject *kobj,
                         struct attribute *attr,
                         char *buffer);

        /*在写sysfs文件时该方法被调用*/
        ssize_t (*store) (struct kobject *kobj,
                          struct attribute *attr,
                          const char *buffer,
                          size_t size);
};
```

show()方法在读操作时被调用。它会拷贝由attr提供的属性值到buffer指定的缓冲区中，缓冲区大小为PAGE_SIZE字节；在x86体系中，PAGE_SIZE为4096字节。该函数如果执行成功，则将返回实际写入buffer的字节数；如果失败，则返回负的错误码。

store()方法在写操作时调用，它会从buffer中读取size大小的字节，并将其存放入attr表示的属性结构体变量中。缓冲区的大小总是为PAGE_SIZE和更小些。该函数如果执行成功，则将返回实际从buffer中读取的字节数；如果失败，则返回负数的错误码。

由于这组函数必须对所有的属性都进行文件I/O请求处理，所以它们通常需要维护某些通用映射来调用每个属性所特有的处理函数。

创建新属性

通常来讲，由kobject 相关ktype所提供的默认属性是充足的，事实上，因为所有具有相同ktype的kobject，如果本质上区别不大的情况下，都应是相互接近的。也就是说，比如对于所有的分区而言，它们完全可以具有同样的属性集合。这不但可以让事情简单，有助于代码合并，还使类似对象在sysfs目录中外观一致。

但是，有时在一些特别情况下会碰到特殊的kobject实例。它希望（甚至是必须）有自己的属性——也许是通用属性没包含那些需要的数据或者函数。为此，内核为能在默认集合之上，再添加新属性而提供了sysfs_create_file()接口：

```
int sysfs_create_file(struct kobject *kobj, const struct attribute *attr);
```

这个接口通过attr参数指向相应的attribute结构体，而参数kobj则指定了属性所在的kobject对象。在该函数被调用前，给定的属性将被赋值，如果成功，该函数返回零，否则返回负的错误码。

注意，kobject中ktype所对应的sysfs_ops操作将负责处理新属性。现有的show()和store()方法必须能够处理新属性。

除了添加文件外，还有可能需要创建符号连接。在sysfs中创建一个符号连接相当简单：

```
int sysfs_create_link(struct kobject *kobj, struct kobject *target, char *name);
```

该函数创建的符号连接名由name指定，连接则由kobj对应的目录映射到target指定的目录。如果成功，该函数返回零，如果失败，返回负的错误码。

删除新属性

删除一个属性需通过函数sysfs_remove_file()完成:

```
void sysfs_remove_file(struct kobject *kobj, const struct attribute *attr);
```

一旦调用返回，给定的属性将不复存在于给定的kobject目录中。

另外由sysfs_create_link()创建的符号连接可通过函数sysfs_remove_link()删除:

```
void sysfs_remove_link(struct kobject *kobj, char *name);
```

调用一旦返回，在kobj对应目录中的名为name的符号连接将不复存在。

上述的四个函数在文件<linux/kobject.h>中声明；sysfs_create_file() 和 sysfs_remove_file() 函数定义于文件fs/sysfs/file.c.中；sysfs_create_link()和sysfs_remove_link() 函数定义于文件fs/sysfs/symlink.c中。

sysfs约定

当前sysfs文件系统代替了以前需要由ioctl()（作用于设备节点）和procfs文件系统完成的功能。目前，在合适目录下实现sysfs属性这样的功能的确别具一格。比如利用在设备映射的sysfs目录中添加一个sysfs属性，代替在设备节点上实现一新的ioctl()。采用这种方法避免了在调用ioctl()时使用类型不正确的参数和弄乱/proc目录结构。

但是为了保持sysfs整齐和直观，开发者必须遵从以下约定。

首先sysfs属性应该保证每个文件只导出一个值，该值应该是文本形式而且被映射为简单C类型。其目的是为了避免数据的过度结构化和太凌乱，现在/proc中就混乱而不具有可读性。每个文件提供一个值，这使得从命令行读写变得简洁，同时也使C语言程序轻易地将内核数据从sysfs导入到自身的变量中去。但有些时候，每值一文件的规则不能很有效地表示数据，那么可以将同一类型的多个值放入在一个文件中。不过这时需要合理地表述它们，比如利用一个空格也许就可使其意义清晰、明了。总的来讲，应考虑sysfs属性要映射到独立的内核变量（正如通常所作），而且要记住应保证从用户空间操作简单，尤其是从shell操作简单。

其次，在sysfs中要以一个清晰的层次组织数据。父子关系要正确才能将kobject层次结构直观地映射到sysfs树中。另外，kobject相关属性同样需要正确，并且要记住kobject层次结构不仅仅存在于内核，而且也要作为一个树导出到用户空间，所以要保证 sysfs树健全无误。

最后，记住sysfs提供内核到用户空间的服务，这多少有些用户空间的ABI（应用程序二进制接口）的作用。用户程序可以检测和获得其存在性、位置、取值以及sysfs目录和文件的行为。任何情况下都不应改变现有的文件，另外更改给定属性，但保留其名称和位置不变无疑是在自找麻烦。

这些简单的约定保证sysfs可为用户空间提供丰富和直观的接口。正确使用sysfs，其他应用程序的开发者绝不会对你的代码报有微辞，相反定会赞美它的。

17.9 内核事件层

你可能会猜到，内核事件层实现了内核到用户的消息通知系统，就是建立在上文一直讨论的kobject基础之上。在2.6.0版本以后，显而易见，系统确实需要一种机制来帮助将事件传出内核输送到用户空间，特别是对桌面系统而言，因为它需要更完整和异步的系统。为此就要让内核将其事件压到堆栈：硬盘满了！处理器过热了！分区挂载了！海盗船来了！

最后一点是开玩笑啦。

早期的事件层没有采用kobject和sysfs，它们如过眼烟云，没有存在多久。现在的事件层借助kobjects和sysfs得以实现是相当理想的。内核事件层把事件模拟为信号——从明确的kobjects对象发出，所以每个事件源都是一个sysfs路径。如果请求的事件与你的第一个硬盘相关，那么/sys/block/had便是源树。实质上，在内核中我们认为事件都是从幕后的kobject对象产生的。

每个事件都被赋予了一个动词或动作字符串表示信号。该字符串会以“被修改过”或“未挂载”等词语来描述事件。

最后，每个事件都有一个可选的负载（payload）。相比传递任意一个表示负载的字符串到用户空间而言，内核事件层使用sysfs属性代表负载。

从内部实现来讲，内核事件由内核空间传递到用户空间需要经过netlink。netlink是一个用于传送网络信息的多点传送套接字。使用netlink意味着从用户空间获取内核事件就如同在套接字上堵塞一样易如反掌。方法就是用户空间实现一个系统后台服务用于监听套接字，处理任何读到的信息，并将事件传送到系统栈里。对于这种用户后台服务来说，一个潜在的目的就是将事件融入D-BUS系统[⊖]。D-BUS系统已经实现了一套系统范围的消息总线，这种总线可帮助内核如同系统中其他组件一样地发出信号。

在内核代码中向用户空间发送信号使用函数kobject_uevent():

```
int kobject_uevent(struct kobject *kobj,
                   enum kobject_action action,
                   struct attribute *attr);
```

第一个参数指定发送该信号的kobject对象。实际的内核事件将包含该kobject映射到sysfs的路径。

第二个参数指定了描述该信号的“动作”或“动词”。实际的内核事件将包含一个映射成枚举类型kobject_action的字符串。该函数不是直接提供一个字符串，而是利用一个枚举变量来提高可重用性和保证类型安全，而且也消除了打字错误或别的其他错误。该枚举变量定义于文件<linux/kobject_uevent.c>中，其形式为KOBJ_*foo*。当前值包含KOBJ_MOUNT、KOBJ_UNMOUNT、KOBJ_ADD、KOBJ_REMOVE和KOBJ_CHANGE等，这些值分别映射为字符串“mount”、

⊖ 想了解D-BUS更多的信息，请看http://dbus.freedesktop.org/.

“unmount”、“add”、“remove”和“change”等。当这些现有的值不够用时，允许添加新动作。

最后一个参数是指向attribute结构体的可选指针。它可看作为一个事件的“负载”。如果事件值不足以描述该事件，那么事件可以指向一个特殊文件，由此文件提供更详细的信息。

该函数分配内存，所以可以睡眠。但在内核中实现了原子和非原子操作两个版本，其不同在于原子操作使用GFP_ATOMIC标志分配内存。

```
int kobject_uevent_atomic(struct kobject *kobj,
                          enum kobject_action action,
                          struct attribute *attr);
```

在可能的情况下，建议使用标准的非原子接口，原子函数与非原子函数两者的参数和行为是一致的。

使用kobject和属性不但有利于很好地实现基于sysfs的事件，同时也有利于创建新kobjects对象和属性来表示新对象和数据——它们尚未出现在sysfs中。

这两个函数分别定义和声明于文件lib/kobject_uevent.c与文件<linux/kobject_uevent.h>中。

17.10 小结

本章介绍了设备模型、sysfs、kobject和内核事件层。但是讨论这些主题时又不得不提及它们大大小小的许多朋友：我们不得不接触了ksets、subsystems、attributes、ktypes、krefs以及别的一些东西。这些结构体对于不同的人有不同的用途。设备驱动程序开发者需要的仅仅是上述主题中的外设视图。多数驱动程序子系统巧妙地隐藏了kobject及其伙伴间的内部细节。理解基本和可能的接口，比如sysfs_create_file()对于设备驱动程序开发者来说就足够了。但是对于核心内核开发者，也许需要更深一步了解kobject。目前kobject已经变得越发重要，以致于使用的范围已不再仅仅局限于驱动程序和驱动程序子系统了。

这章讨论了内核中我们要学习的最后一个子系统，从下面开始要介绍一些普遍的但却重要的主题，这些主题是任何一个内核开发者都需要了解的，首先要讲的就是调试！

第18章 调 试

调试工作艰苦是内核级的开发区别于用户级开发的一个显著特点。相对于用户级开发，内核开发确实要艰苦得多。更要命的是，它带来的风险相比用户级开发更高。内核的一个错误往往马上就能让系统崩溃——一点情面都不会留。

驾驭内核调试的能力（当然，最终是为了能够成功地开发内核）很大程度上取决于经验和对整个操作系统的把握。没错，玉树临风可能会对别的事情有帮助，但是调试内核的关键还是在于你对内核的深刻理解。然而我们必须找到可以开始着手的地方，所以，在这一章里我们从调试内核的一种可能步骤开始。

18.1 调试前需要准备什么

你确定要开始了吗？这可能会是一个漫长而又困难重重的苦旅。不少bug已经让整个开发社区几个月都食不甘味了。幸运的是，虽然确实有许多这样难以对付的bug，但也有不少比较简单，而且容易消灭的小bug。运气好时，你可能面对的是些简单的小bug。你开始做一些调查之前，不会清楚到底面对的是什么。现在，你需要的只是：

- 一个bug。听起来很可笑，但你确实需要一个定义精确的bug。如果错误总是能够再现的话，那对我们会很有帮助（而且有一部分错误确实如此）。然而不幸的是，大部分bug通常并非是行为可靠而又定义清楚的。
- 藏匿bug的内核的版本（一般可以假设是在最新版本的内核里，否则谁又会去理睬它呢？）如果你能搞清楚这个bug最早出现在哪个版本中就再理想不过了。如果你不知道，我们待会儿会解释该怎么办。
- 一点好运气、一些内核骇客技巧，或者都需要那么一些。

如果你没办法让bug重现出来，下面要讲的这些步骤就毫无意义。问题的关键在于你是否能够让这些错误重复发生。如果你不能，消灭bug就只能通过抽象出问题，再从代码中搜索蛛丝马迹来进行了。虽然有时也得这么做（了解了吧，内核开发者其实也就是这么棒了），但如果你能够让错误重复出现，成功的机会要大许多。

有一个bug存在而有人没办法让它重现，这听起来可能感觉挺奇怪。在用户级的程序里，bug常常表现得很直截了当——比如，执行foo就会让程序立即产生核心信息转储（core dump）。但内核全然不是这样。内核、用户程序和硬件之间的交互非常微妙。一个竞争条件往往把它狰狞的面目隐藏在某个算法百万次的迭代之中。设计不佳的甚至是包含错误的代码可能在某些系统上表现得相当不错，而在其他系统上却让人无法忍受。在跟踪bug的时候，对于一些特定的配置、一些特定的机器，付出额外的努力是司空见惯的，要不然的话它可能压根就不会显露原型。在跟踪bug的

时候，你掌握的信息越多越好。许多时候，当你可以精确地重现一个bug的时候，你就已经成功了一大半了。

18.2 内核中的bug

内核中的bug和用户空间应用程序中的bug一样多种多样。它们的产生可以有无数的原因，同时它们的表象也变化多端。从明白无误的错误代码（比如，没有把正确的值存放在恰当的位置）到同步时发生的错误（比如，共享变量锁定不当），都是bug的温床。从降低所有东西的运行性能到毁坏数据，都可能是bug发作时的症状。

从隐藏在源代码中的错误到展现在目击者面前的bug，其发作往往是一系列连锁反应的事件才可能触发的。举个例子，一个被共享的结构体，如果它没有引用计数，那么它就有可能会引发竞争条件。因为没有引用计数的话，一个进程可以在另外一个进程仍然需要使用该结构的时候就释放掉它。继而，第二个进程就有可能试图通过无效的指针去使用一个不存在的数据结构。这样做可能导致引用一个空指针，也可能会读出一些垃圾数据，还可能并不产生什么恶果（如果该数据并没有被其他什么覆盖的话）。引用空指针会导致产生一个oops，而垃圾数据可能会导致系统崩溃（这种情形比oops还坏）。用户报告了oops或系统的错误现象之后，开发者回过头来观察错误情形，发现在释放数据之后还会对它进行读写，存在着一个竞争条件，于是就会进行修正，给这个共享的结构加上适当的引用计数。多说一句，对它的访问大多还需要由锁来保证。

调试内核听起来很难，但事实上Linux内核与其他大型的软件项目也没有什么太大的不同。内核确实有一些独特的问题需要考虑，像定时限制和竞争条件等，它们都是允许多个线程在内核中同时运行产生的结果。我相信你有一定的理解能力也能够付出些努力，所以你应该可以调试内核中的存在的问题（说不定你还会喜欢上这种挑战呢）。

18.3 printk()

内核提供的打印函数printk()，和C库提供的printf()函数功能几乎相同。实际上，在整个这本书中我们都没有用到这两个函数的不同部分。从它实现的大部分功能来说，这个名字很不错，printk()就是内核的格式化打印函数。但是，差异确实也还是存在的。

18.3.1 printk()函数的健壮性

健壮性是printk()函数最容易让人们接受的一个特质。任何时候，任何地方都能调用它，内核中的printk()比比皆是。它可以在中断上下文和进程上下文中调用；它可以在持有锁时调用；它可以在多处理器上同时调用，而且调用者连锁都不必使用。

它是一个弹性极佳的函数。这一点相当重要，printk()之所以这么有用，就在于它随时都在那里而且随时都能用。

printk()的脆弱之处

printk()函数的健壮躯壳下也难免会有漏洞。在系统启动过程中，终端还没有初始化之前，在某些地方不能使用它。不过说实在的，如果终端没有初始化，你又能输出到什么地方去呢？

这一般不是一个什么问题，除非你要调试的是启动过程最开始的那些步骤（比如说在负责执行硬件体系结构相关的初始化动作的setup_arch()函数中）。着手进行这样的调试挑战性很强——没

有任何打印函数能用确实让问题更加棘手。

不过还是有一些指望的，虽然不多。核心硬件部分的黑客依靠此时能够工作的硬件设备（比如说一个串口）与外界通信。相信我，绝大部分人对此都不会感兴趣。一些被支持的体系结构确实靠这种方式实现了健全的解决方案，但是——其他系统（包括i386）都通过提供可用的补丁来扭转局面。

解决的办法是提供一个printk()的变体函数:early_printk()，在启动过程的初期就具有在终端上打印的能力。它的功能与printk()完全相同，区别仅仅在于名字和它能够更早地工作。不过，由于该函数在一些内核支持的硬件体系结构上无法实现，所以这种办法缺少可移植性。可是，如果它能够工作的话，它就是你最好的助手。

除非你在启动过程的初期就要在终端上输出，否则可以认为printk()在什么情况下都能工作。

18.3.2 记录等级

printk()和printf()一个最主要的区别就是前者可以指定一个记录级别。内核根据这个级别来判断是否在终端上打印消息。内核把级别比某个特定值低的所有消息显示在终端上。

可以通过下面这种方式指定一个记录级别：

```
printk(KERN_WARNING "This is a warning!\n");
printk(KERN_DEBUG "This is a debug notice!\n");
printk("I did not specify a loglevel!\n");
```

KERN_WARNING和KERN_DEBUG都是<linux/kernel.h>中的简单宏定义。它们扩展开是像“<4>”或“<7>”这样的字符串，加进printk()函数要打印的消息的开头。内核用这个指定的记录等级和当前终端的记录等级console_loglevel来决定是不是向终端上打印。表18-1列举了所有可供使用的记录等级。

表18-1　可供使用的记录等级

记录等级	描　述
KERN_EMERG	紧急情况
KERN_ALERT	需要立即被注意到的错误
KERN_CRIT	临界情况
KERN_ERR	错误
KERN_WARNING	警告
KERN_NOTICE	普通的，不过也有可能需要注意的情况
KERN_INFO	非正式的消息
KERN_DEBUG	调试信息——一般是冗余信息

如果你没有特别指定一个记录等级，函数会选用默认的DEFAULT_MESSAGE_ LOGLEVEL，现在默认等级是KERN_WARNING。由于这个默认值将来存在变化的可能性，所以还是应该给你自己的消息指定一个记录等级。

内核将最重要的记录等级KERN_EMERG定为“<0>”，将无关紧要的记录等级“KERN_DEBUG”定为“<7>”。举例来说，当编译预处理完成之后，前例中的代码实际被编译成如下格式：

```
printk("<4> This is a warning!\n");
```

```
printk("<7>This is a debug notice!\n");
printk("<4> did not specify a loglevel!\n");
```

怎样给你调用的printk()赋记录等级完全取决于你自己。那些正式的、需要保持的消息应该有合适的记录等级。但是那些当你试图解决一个问题时加得到处都是的调试信息——必须承认，我们都这么干而且也确实行得通——可以按照你的想法赋给记录等级。一种选择是保持终端的默认记录等级不变，给所有调试信息KERN_CRIT或更低的等级。相反，可以给你的所有调试信息KERN_DEBUG等级，而调整终端的默认记录等级。两种方法各有利弊，自己拿主意吧。

记录等级（loglevel）定义在文件<linux/kernel.h>中。

18.3.3 记录缓冲区

内核消息都被保存在一个LOG_BUF_LEN大小的环形队列中。该缓冲区大小可以在编译时通过CONFIG_LOG_BUF_SHIFT进行调整。在单处理器的系统上其默认值是16Kb。换句话说，就是内核在同一时间只能保存16Kb的内核消息。在消息被读出到用户空间时就会从队列中被清除掉。如果消息队列已经达到最大值，那么如果再有printk()调用时，新消息将覆盖队列中的老消息。这个记录缓冲区之所以称为环形是因为它的读写都是按照环形队列方式进行操作的。

使用环形队列有许多好处。由于读写环形队列时同步问题非常容易解决，所以即使在中断上下文中也可以方便地使用printk()。此外，它使记录维护起来也更容易。如果有大量的消息同时产生，新消息只需覆盖掉旧消息即可。在某个问题引发大量消息的时候，记录只会覆盖掉它本身，而不会因为失控而消耗掉大量内存。而环形缓冲区的惟一缺点——可能会丢失消息，与简单性和健壮性的好处相比，这种缺点完全可以忽略不计。

18.3.4 syslogd和klogd

在标准的Linux系统上，用户空间的守护进程klogd从记录缓冲区中获取内核消息，再通过syslogd守护进程将它们保存在系统日志文件中。klogd程序既可以从/proc/kmsg文件中也可以通过syslog()系统调用读取这些消息。默认情况下，它选择读取/proc方式实现。不管是哪种方法，直到有新的内核消息可供读出，klogd都会阻塞。在被唤醒之后，它会读取出新的内核消息并进行处理。默认方式下，处理例程就是把消息传给syslogd守护进程。

syslogd守护进程把它接收到的所有消息加进一个文件中，该文件默认是/var/log/messages。你也可以通过/etc/syslog.conf配置文件重新指定。

在载入klogd的时候，可以通过指定-c标志来改变终端的记录等级。

18.3.5 printk()和内核开发时需要留意的一点

当你刚开始开发内核代码的时候，往往会把printk()敲成printf()。这很正常，你没法抗拒多年来在用户级程序中使用printf()的习惯。幸而这种错误不会持续很长时间，反复出现的链接错误很快就会让你在心烦意乱中开始培养新的习惯。

终于有一天，在编写用户级程序的时候，你敲printf()的时候不小心敲成了printk()。恭喜你，你成为一个真正的内核黑客的时刻终于到来了。

18.4 oops

oops是内核告知用户有不幸发生的最常用的方式。由于内核是整个系统的管理者，所以它不能采取像在用户空间出现运行错误时使用的那些简单手段，因为它很难自行修复，它也不能将自己杀死。内核只能发布oops。这个过程包括向终端上输出错误消息，输出寄存器中保存的信息并输出可供跟踪的回溯线索。内核中出现的故障很难处理，所以内核往往要经历严峻的考验才能发送出oops和靠它自己完成的一些清理工作。通常，发送完oops之后，内核会处于一种不稳定状态。举例来说，oops发生的时候内核可能正在处理非常重要的数据。它可能持有一把锁或正在和硬件设备交互。内核必须适当地从当前的上下文环境中退出并尝试恢复对系统的控制。多数时候，这种尝试都会失败。因为如果oops在中断上下文时发生，内核根本无法继续，它会陷入混乱。混乱的结果就是系统死机。如果oops在idle进程（pid为0）或init进程（pid为1）时发生，结果同样是系统陷入混乱，因为内核缺了这两个重要的进程根本就没法工作。不过，要是oops在其他进程运行时发生，内核就会杀死该进程并尝试着继续执行。

oops的产生有很多可能原因，其中包括内存访问越界或者非法的指令等。作为一个内核开发者，你将会经常处理（毫无疑问，也将导致）oops。

紧接着的是一个oops的实例，它是在一台PPC机器上的tulip网卡的定时器处理函数运行时发生的：

```
Oops: Exception in kernel mode, sig : 4
Unable to handle kernel NULL pointer dereference at virtual address 00000001
NIP: C013A7F0 LR:C013A7F0 SP:C0685E00 REGS: c0905d10 TRAP: 0700
Not tainted
MSR: 00089037 EE:1 PR: 0 FP:0 ME: 1 IR/DR: 11
TASK = c0712530[0] 'swapper' Last syscall: 120
GPR00:C013A7C0 C0295E00 C0231530 0000002F 00000001 C0380CB8 C0291B80 C02D0000
GPR08:000012A0 00000000 00000000 C0292AA0 4020A088 00000000 00000000 00000000
GPR16:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR24:00000000 00000005 00000000 00001032 C3F7C000 00000032 FFFFFFFF C3F7C1C0
Call trace:
[c013ab30] tulip_timer+0x128/0x1c4
[c0020744] run_timer_softirq+0x10c/0x164
[c001b864] do_softirq+0x88/0x104
[c0007e80] timer_interrupt+0x284/0x298
[c00033c4] ret_from_except+0x0/0x34
[c0007b84] default_idle+0x20/0x60
[c0007bf8] cpu_idle+0x34/0x38
[c0003ae8] rest_init+0x24/0x34
```

使用PC的读者可能对这么多的寄存器感到惊奇（居然有32个之多）。读者可能对x86系统更熟悉一些，在这种系统上，oops会简单一点。但是，oops中包含的重要信息对于所有体系结构都是完全相同的：寄存器上下文和回溯线索。

回溯线索显示了导致错误发生的函数调用链。这样我们就可以观察究竟发生了什么：机器处于空闲状态，正在执行idle循环，由cpu_idle()反复调用default_idle()。此时定时器中断产生了，它引起了对定时器的处理。tulip_timer()这个定时器处理函数被调用，而就是它引用了空指针。你甚至可以通过偏移量（像0x128/0x1c4这些出现在函数左侧的数字）找出导致问题的语句。

寄存器上下文信息可能同样有用，尽管使用起来不那么方便。如果你有函数的汇编代码，这些寄存器数据可以帮助你重建引发问题的现场。在寄存器中发现一个本不应该出现的数值可能会在黑暗中给你带来第一丝光明。在上面的例子中，我们可以查看是哪个寄存器包含了NULL(一个所有位都为零的数值)进而找出是函数的哪个变量的值不正常。一般在这种情况下问题往往是竞争引起的，在本例中，是定时器和这块网卡驱动的其他部分之间的竞争。调试一个竞争条件往往很有挑战性。

18.4.1 ksymoops

前面列举的oops可以说是一个经过解码的oops，因为内存地址都已经被转换成了它们对应的函数。下面是其未解码版本：

```
NIP:C013A7F0 LR:C013A7F0 SP:C0685E00 REGS: c0905d10 TRAP 0700
Not tainted
MSR:00089037 EE:1 PR: 0 FP:0 ME: 1 IR/DR: 11
TASK = c0712530[0] 'swapper' Last syscall:120
GPR00:C013A7C0 C0295E00 C0231530 0000002F 00000001 C0380CB8 C0291B80 C02D0000
GPR08:000012A0 00000000 00000000 C0292AA0 4020A088 00000000 00000000 00000000
GPR16:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR24:00000000 00000005 00000000 00001032 C3F7C000 00000032 FFFFFFFF C3F7C1C0
call trace: [c013ab30] [c0020744] [c001b864] [c0007e80] [c00061c4]
[c0007b84] [c0007bf8] [c0003ae8]
```

回溯线索中的地址需要被转化成有意义的符号名称才方便使用。这需要调用ksymoops命令，并且还必须提供编译内核时产生的System.map。如果你使用的是模块，你还需要一些模块信息。ksymoops通常会自行解析这些信息，所以一般你可以这样调用它：

```
ksymoops saved_oops.txt
```

然后该程序就会吐出解码版的oops。如果ksymoops无法找到默认位置上的信息，或者你想提供不同信息，该程序可以接受许多参数。它的使用手册上提供了许多说明信息，使用之前你最好先行查阅。

ksymoops一般会随Linux发行提供。

18.4.2 kallsyms

谢天谢地，现在已经无须使用ksymoops工具了，这是一个了不起的工作。因为尽管开发者使用它的时候一般很少出现问题，但是最终用户常常会错误地匹配system.map文件或错误地对oops进行解码。

开发版的2.5内核引入了kallsyms特性，它可以通过定义CONFIG_KALLSYMS配置选项启用。该选项可以载入内核镜像对应的内存地址的符号的名称，所以内核可以打印解码好的跟踪线索。相应地，解码oops也不再需要system.map或者ksymoops工具了。另外，这样做会使内核变大一些，因为地址对应的符号名称必须始终驻留在内核所在的内存上。占用这些内存是值得的，至少在开发过程中如此。

18.5 内核调试配置选项

在编译的时候，为了方便调试和测试代码，内核提供了许多配置选项。这些选项都在内核配

置编辑器的内核开发（Kernel Hacking）菜单项中，它们都依赖于CONFIG_ DEBUG_KERNEL。当你开发内核的时候，作为一种练习，不妨打开所有这些选项。

有些选项确实有用，你应该启用slab layer debugging（slab层调试选项），high-memory debugging（高端内存调试选项），I/O mapping debugging（I/O映射调试选项），spin-lock debugging（自旋锁调试选项）和stack-overflow checking（栈溢出检查选项）。其中最有用的一个是sleep-inside-spinlock checking（自旋锁内睡眠选项），这些选项确实能完成不少调试工作。

调试原子操作

从2.5版开始，为了检查各类由原子操作引发的问题，内核提供了极佳的工具。回忆一下第8章，原子操作指那些能够不分隔执行的东西；在执行时不会被中断否则就不完成的代码。正在使用一个自旋锁或禁止抢占的代码进行的就是原子操作。在进行此类操作的时候，代码不能睡眠—使用锁时睡眠是引发死锁的元凶。

托内核抢占的福，内核提供了一个原子操作计数器。它可以被配置成一旦在原子操作过程中进程进入睡眠或者做了一些可能引起睡眠的操作，就打印警告信息并提供追踪线索。所以，包括正使用锁的时候调用schedule()，正使用锁的时候以阻塞方式请求分配内存和在引用单CPU数据时睡眠在内，各种潜在的bug都能够被探测到。

下面这些选项可以最大限度地利用该特性：

```
CONFIG_PREEMPT=y
CONFIG_DEBUG_KERNEL=y
CONFIG_KALLSYMS=y
CONFIG_SPINLOCK_SLEEP=y
```

18.6　引发bug并打印信息

一些内核调用可以用来方便标记bug，提供断言并输出信息。最常用的两个是BUG()和BUG_ON()。当被调用的时候，它们会引发oops，导致栈的回溯和错误信息的打印。为什么这些声明会导致oops跟硬件的体系结构是相关的。大部分体系结构把BUG()和BUG_ON()定义成某种非法操作，这样自然会产生需要的oops。你可以把这些调用当作断言使用，想要断言某种情况不该发生：

```
if (bad_thing)
        BUG();
```

或者使用更好的形式：

```
BUG_ON(bad_thing);
```

多数内核开发者相信BUG_ON() 比BUG()更清晰、更可读，而且BUG_ON()会将其声明作为一个语句放入unlikely()中。请注意，有些开发者在讨论是否能用一个编译选项将BUG_ON()声明在编译时剔除，以便能在嵌入内核中节约空间。这就意味着在BUG_ON()内的声明有可能带来任何“不良反映”都可以避免了。

可以用panic()引发更严重的错误。调用panic()不但会打印错误消息，而且还会挂起整个系统。显然，你只应该在极端恶劣的情况下使用它：

```
if (terrible_thing)
        panic("foo is %ld!\n", foo);
```

有些时候，只需要在终端上打印一下栈的回溯信息来帮助你测试。此时可以使用dump_stack()。它只在终端上打印寄存器上下文和函数的跟踪线索：

```
if (!debug_check) {
        printk(KERN_DEBUG "provide some information...\n");
        dump_stack();
}
```

18.7 神奇的SysRq

神奇系统请求键是另外一根救命稻草，该功能可以通过定义CONFIG_MAGIC_SYSRQ配置选项来启用。SysRq(系统请求)键都是键盘上的标准键。在i386和PPC上，它可以通过ALT-PrintScreen访问。当该功能被启用的时候，无论内核处于什么状态，你都可以通过特殊的组合键跟内核进行通信。这种功能可以让你面对一台奄奄一息的系统时能完成一些有用的工作。

除了配置选项以外，还要通过一个sysctl用来标记该特性的开或关。需要启用它时用下命令：

```
echo 1 > /proc/sys/kernel/sysrq
```

从终端上，你可以键入SysRq-h获取一份可用的选项列表。SysRq-s将“脏”缓冲区跟硬盘交换分区同步，SysRq-u卸载所有的文件系统，SysRq-b重启设备。在一行内发送这三个键的组合可以重新启动濒临死亡的系统，这比直接按下机器的reset键要安全一些。

如果机器已经完全锁死了，它也可能不会再响应神奇系统请求键，或者无法完成给定的命令。不过如果运气稍好的话，这些选项或许可以保存你的数据或者帮你进行调试。表18-2列举了所有支持的系统请求命令。

表 18-2 支持SysRq 的命令

主要命令	描 述
SysRq-b	重新启动机器
SysRq-e	向init以外的所有进程发送SIGTERM信号
SysRq-h	在控制台显示SysRq
SysRq-i	向init以外的所有进程发送SIGKILL信号
SysRq-k	安全访问键：杀死这个控制台上的所有程序
SysRq-l	向包括init的所有进程发送SIGKILL信号
SysRq-m	把内存信息输出到控制台
SysRq-o	关闭机器
SysRq-p	把寄存器的信息输出到控制台
SysRq-r	关闭键盘原始模式
SysRq-s	把所有已安装文件系统都刷新到磁盘
SysRq-t	把任务信息输出到控制台
SysRq-u	卸载所有已安装文件系统

内核代码树中文件Documentation/sysrq.txt有此方面更详细的信息。实际的实现在drivers/char/sysrq.c中。神奇系统请求健是调试和挽救垂危系统所必需的一种工具。由于该功能对终端上的任何用户都提供服务，所以在重要的机器上启用它你需要三思而行。可是对于你自己用于开发的机器，启用它确实帮助很大。

18.8　内核调试器的传奇

很多内核开发者一直以来都希望能拥有一个用于内核的调试器。不幸的是，Linus不愿意在他的内核源代码树中加入一个调试器。他认为调试器会误导开发者，从而导致引入不良的修正。没有人能对他的逻辑提出异议，从真正理解代码出发，确实更能保证修正的正确性。然而，许多内核开发者们还是希望有一个官方发布的，用于内核的调试器。因为这个要求看起来不会马上满足，所以许多补丁应运而生了，它们为标准内核附加上了内核调试的支持。虽然这都是一些不被官方认可的附加补丁，但它们确实功能完善，十分强大。在我们深入了解这些解决方案之前，最好先看看标准的Linux调试器gdb能够给我们一些什么帮助。

18.8.1　gdb

可以使用标准的GNU调试器对正在运行的内核进行查看。针对内核启动调试器的方法与针对进程的方法大致相同：

```
gdb vmlinux /proc/kcore
```

其中vmlinux文件是未经压缩的内核映像，不是压缩过的zImage或bzImage，它存放在源代码树的根目录上。

/proc/kcore作为一个参数选项，是作为core文件来用的，通过它你能够访问到内核驻留的高端内存。只有超级用户才能读取此文件的数据。

可以使用gdb的所有命令来获取信息。举个例子，为了打印一个变量的值，你可以：

```
p global_variable
```

反汇编一个函数：

```
disassemble function
```

如果你编译内核的时候使用了-g参数（在内核的Makefile文件的CFLAGS变量中加入-g），gdb还可以提供更多的信息。比如，你可以打印出结构体中存放的信息或是跟踪一个指针。当然，你编译出的内核会大很多，所以不要把编译带调试信息的内核当作一种习惯。

接下来，就要说不幸的那一面了，gdb还是有很多局限性的。它没有任何办法修改内核数据。它也不能单步执行内核代码，不能加断点。不能修改内核数据是个非常大的缺陷。尽管在必要时反汇编函数无疑是个非常有用的功能，但是能够修改数据的却更为有用。

18.8.2　kgdb

kgdb是一个补丁，它可以让我们在远端主机上通过串口利用gdb的所有功能对内核进行调试。这需要两台计算机。第一台运行带有kgdb补丁的内核。第二台通过串行线（不通过modem，直接连接两台机器的电缆）使用gdb对第一台进行调试。通过kgdb，gdb的所有功能都能使用：读取或修改变量值，设置断点，设置关注变量，单步执行等等。某些版本的gdb甚至允许执行函数。

设置kgdb和连接串行线比较麻烦，但是一旦做完了，调试就变得很简单了。该补丁会在Documentation/目录下安装很多说明文件，你可以把它们挑出来研究一下。

不同体系结构、不同内核版本使用的kgdb由不同的人员维护，为了给需要调试的内核找到合适的补丁，还是在网上搜索一下比较好。

18.8.3 kdb

kdb是kgdb的一种替代品。不象kgdb，它不是一个远端调试器。kdb这个补丁对内核源代码进行了很多修改，使调试内核在本地主机上就可以进行。它提供了变量修改、设置断点、单步执行等许多功能。运行调试器非常简单，在终端上敲一下break键就可以了。内核执行oops的时候，它也会自动投入运行。安装kdb之后，可以在Documentation/kdb中找到有关它的详细文档。

在http://oss.sgi.com/可以找到kdb补丁。

18.9 刺探系统

伴随着你调试内核的经验越来越丰富，你刺探内核以获取答案的技巧也会越来越多。由于调试内核是一项复杂度非常高的工作，所以掌握好任何技巧和窍门都会有所帮助。下面就介绍一些。

18.9.1 用UID作为选择条件

如果你开发的是进程相关的部分，有些时候，你可以在提供替代物的同时不打破原有代码的可执行性。这在你开发重要系统调用的时候，或者在你希望进行调试时系统功能依旧健全的情况下非常有用。

举个例子，假设为了加入一个激动人心的新特性，你重写了fork()系统调用。除非你第一次的尝试就完美无缺，否则系统调试就是一场噩梦。如果fork()系统调用不正常的话，你压根就不用指望整个系统还能正常工作。当然，和任何时候一样，希望总是存在的。

一般情况下，只要保留原有的算法而把你的新算法加入到其他位置上，基本就能保证安全。你可以利用把用户id（UID）作为选择条件来实现这种功能，通过这种选择条件，你可以安排到底执行哪种算法：

```
if (current->uid != 7777) {
    /* 老算法.. */
} else {
    /* 新算法.. */
}
```

除了uid为7777以外，其他所有的用户都用的是老算法。你可以创建一个UID为7777的用户，专门来测试新算法。对于要求很严格的进程相关部分的代码来说，这种方法使得测试变得容易了许多。

18.9.2 使用条件变量

如果代码与进程无关，或者你希望有一个针对所有情况都能使用的机制来控制某个特性，你可以使用条件变量。这比使用UID还来得简单。你只需要创建一个全局变量作为一个条件选择开关。如果该变量为零，你就是用一个分支上的代码。如果它不为零，你就选择另外一个分支。你可以通过某种界面提供对这个变量的操控，也可以直接通过调试器进行操控。

18.9.3 使用统计量

有些时候你需要掌握某个特定事件的发生规律。有些时候你需要比较多个事件并从中得出规律。通过创建统计量并提供某种机制访问其统计结果，很容易就能满足这种需求。

举个例子，假设我们希望得到foo和bar的发生频率，那么在某个文件中，当然最好是在事件所发生的那个文件里，定义两个全局变量：

```
unsigned long foo_stat = 0;
unsigned long bar_stat = 0;
```

每当事件发生的时候，就让相应的变量加1。然后按照你喜欢的方式提供这两个变量的访问方法。比如，你可以在/proc目录中创建一个文件，还可以新创建一个系统调用。最简单的办法当然还是通过调试器直接访问它们。

注意，这种实现并非是SMP安全的。理想的办法是通过原子操作进行实现。但是仅仅对于一个简单的每次加1的调试统计量，一般无需搞得这么麻烦。

18.9.4 重复频率限制

为了发现一个错误，开发者们往往在代码的某个部分加入很多错误检查语句（多数对应的都是一些print语句）。在内核中，有些函数每秒都要被调用很多次。如果你在这样的函数中加入了prink()，那么系统马上就会被显示调试信息这一个任务压得喘不过气来，很快就什么也干不成了。

有两种相关的技巧可以用来防止此类问题的发生。第一种是重复频率限制，如果某种事件发生的非常频繁，而你又需要观察它的整体进展情况，你就可以让这种技巧施展身手了。为了避免调试信息发生井喷，你可以每隔几秒执行一次打印（或者是其他任何你想完成的操作）。举个例子：

```
static unsigned long prev_jiffy = jiffies;      /* 频率限制 */

if (time_after(jiffies, prev_jiffy + 2*HZ)) {
    prev_jiffy = jiffies;
    printk(KERN_ERR "blah blah blah\n");
}
```

此例中，调试信息最多两秒打印一次。这可以让你的终端不至于被汹涌而至的调试信息洪流充塞，也保证你的系统依旧能用。你完全可以根据自己的需要，或低或高的调整这种重复频率。

在你观察某种频繁发生的时间时，还有另一种棘手的问题。与前面的例子不同，你想观察的不是整个事件在内核中进展过程，你只是想在某个事件发生时得到一个通知。可能只要得到一次或是两次就足够了。如果这种通知在被触发一次之后依旧不停的到来，那就比较麻烦了。下面这种技巧针对的就不再是如何限制重复频率了，它要实现的是发生次数限制。

```
static unsigned long limit = 0;

if (limit < 5) {
    limit++;
    printk(KERN_ERR "blah blah blah\n");
}
```

此例中，调试信息输出五次就封顶了。五次之后，打印条件总是不能成立。

不管是哪种技巧，用到的变量都应该是静态的（static），并且应该限制在函数的局部以内。像例子中展示的那样，这样才能保证在函数的多次调用中变量的值能够保留。

这些例子的代码都不是SMP或抢占安全的，不过，只需要用原子操作改造一下就没问题了。可是，这只不过是在调试中才会用到的代码，没有必要搞得那么复杂吧？

18.10 用二分查找法找出引发灾难的变更

了解bug是什么时候引入内核源代码的很有用。如果你知道2.6.10中出现了一个bug，而能肯定2.4.9中没有，那么你就能够很容易地确定引发这个bug的代码变更。消灭bug变得唾手可得，要么取消这个变更，要么对其进行修正。

可是，很多时候你并不知道到底是哪个内核版本引入了bug。你知道当前版本里bug是确确实实存在的，不过，它好像就是存在于当前版本中。这就需要做一些调查取证了，不过只需要花一点点力气，你就能找出引发问题的代码变更了。元凶在手，消灭bug就指日可待了。

一开始，你需要一个可靠的可复制的错误。最好是一个系统一启动就能查证的bug。下面，你需要一个能确保没问题的内核。你应该能够找到。举个例子，你知道几个月前的内核没有这种错误，那么就从那时使用的内核中选取一个。如果发现问题那时就存在了，那么就找更早的。这不会太难（除非bug是Linux与生俱来的）一定要找到不含bug的内核。

接下来需要一个肯定有问题的内核。为了简单起见，你应该从已知最早出现该问题的内核开始。

现在，你就可以在问题内核和良好的内核之间使用二分法了。举个例子，假定确保没有问题的内核版本是2.4.11，有问题的内核版本是2.4.20。从二者的正中选取一个内核版本，像2.4.15。检查2.4.15是否包含此bug。如果2.4.15没有问题，那么你就知道错误是发生在此版本之后了。所以，再从2.4.15开始，在它和2.4.20正中选取下一个版本，比如说2.4.17进行检查。如果2.4.15有问题，那么错误就可能发生在此版本之前了，那么你就该选2.4.13作为下一个待查目标了。就这样重复筛选。

最终你肯定能把问题局限在两个版本之间——一个包含错误而另外一个不包含。你就能够很容易地对引发这个bug的代码变更进行定位了。

这种方式比依次对每个版本的内核进行核查要好得多了。

18.11 当所有的努力都失败时

或许你已经做完了所有你能想到的尝试。你在键盘上呕心沥血了几个小时，实际上，可能是无数日子，答案依旧没有眷顾你。此时，如果bug是在Linux内核的主流部分中，你可以在内核开发社区中寻求其他开发者的帮助。

你应该向内核邮件列表发送一份电子邮件，对bug进行完整而又简洁的描述，你的发现可能会对找到最终的答案起到帮助作用。毕竟，没人希望bug存在。

第20章会重点推荐社区和它最重要的论坛，Linux内核邮件列表（LKML）。

第19章

可移植性

Linux是一个可移植性非常好的操作系统，它广泛支持了许多不同体系结构的计算机。可移植性是指代码从一种体系结构移植到另外一种不同的体系结构上的方便程度。我们都知道Linux是可移植的，因为它已经能够在各种不同的体系结构上运行了。但这种可移植性不是凭空得来的一它需要在做设计时就为此付出诸多努力。现在，这种努力已经开始得到回报了，移植Linux到新的系统上就很容易（相对来说）完成。本章中我们将讨论的是如何编写可移植的代码—在编写内核代码和驱动程序时，你必须时刻牢记这个问题。

有些操作系统在设计时把可移植性作为头等大事之一。尽可能少地涉及与机器相关的代码。汇编代码用得少之又少，为了支持各种不同类别的体系结构，界面和功能在定义时都尽最大可能地具有普适性和抽象性。这么做最显著的回报就是需要支持新的体系结构时，所需完成的工作要相对容易许多。一些移植性非常高而本身又比较简单的操作系统在支持新的体系结构时，可能只需要为此体系结构编写几百行专门的代码就行了。问题在于，体系结构相关的一些特性往往无法被支持，也不能对特定的机器进行手动优化。选择这种设计，就是利用代码的性能优化能力换取代码的可移植性。Minix、NetBSD和许多研究用的系统就是这种高度可移植操作系统的实例。

与之相反，还有一种操作系统完全不顾及可移植性，它们尽最大的可能追求代码的性能表现，尽可能多地使用汇编代码，压根就是为在一种硬件体系结构使用。内核的特性都是围绕硬件提供的特性设计的。因此，将其移植到其他体系结构就等于从头再重新编写一个新的操作内核。选择这种设计，就是用代码的可移植性换取代码的性能优化能力。这样的系统往往比移植性好的系统难于维护（DOS和Windows 9x就是这样的典型）。虽然目前看来这些系统对性能的要求不见得比对可移植性要求更强，不过它们还是愿意牺牲可移植性，而不乐意让设计打折扣。

Linux在可移植性这个方面走的是中间路线。差不多所有的接口和核心代码都是独立于硬件体系结构的C语言代码。但是，在对性能要求很严格的部分，内核的特性会根据不同的硬件体系进行调整。举例来说，需要快速执行和底层的代码都是与硬件相关并且是用汇编语言写成的。这种实现方式使Linux在保持可移植性的同时兼顾对性能的优化。当可移植性妨碍性能发挥的时候，往往性能会被优先考虑。除此之外，代码就一定要保证可移植性。

一般来说，暴露在外的内核接口往往是与硬件体系结构无关的。如果函数的任何部分需要针对特殊的体系结构（无论是出于优化的目的还是作为一种必需的选择）提供支持的时候，这些部分都会被安置在独立的函数中，等待调用。每种被支持的体系结构都实现了一个与体系结构相关的函数，而且会被链接到内核映像之中。

调度程序就是一个好例子。调度程序的主体程序存放在kernel/sched.c文件中，用C语言编写的，与体系结构无关。可是，调度程序需要进行的一些工作，比如说切换处理器上下文和切换地址空

间等，却不得不依靠相应的体系结构完成。于是，内核用C语言编写了函数context_switch()用于实现进程切换，而在它的内部，会调用switch_to()和switch_mm()分别完成处理器上下文和地址空间的切换。

而对于Linux支持的每种体系结构，它们的switch_to()和switch_mm()实现各不相同。所以，当Linux需要移植到新的体系结构上的时候，只需要重新编写和提供这样的函数就可以了。

与体系结构相关的代码都存放在arch/architecture/和include/asm-architecture/目录中，architecture是Linux支持的体系结构的简称。比如说，Intel x86体系结构对应的简称是i386。与这种体系结构相关的代码都存放在arch/i386/以及include/asm-i386/目录下。2.6系列内核支持的体系结构包括alpha、arm、cris、h8300、i386、ia64、m32r、m68k、m68knommu、mips、mips64、parisc、ppc、ppc64、s390、sh、sparc、sparc64、um、v850和x86-64。本章稍后给出的表19-1是一份更详尽的清单。

19.1 Linux的可移植性

当Linus最初把Linux带到这个无法预测的大千世界的时候，它只能在i386上运行。尽管这个操作系统通用性很强，代码也写得不错，可是可移植性在那时算不上是一个关注焦点。实际上，Linus一度还建议让Linux只在i386体系结构上驰骋。不过，人们还是在1993年开始把Linux向Digital Alpha体系结构上移植了。Digital Alpha是一种高性能现代计算机体系结构，它支持RISC和64位寻址。这与Linus最初选的i386无疑是天壤之别。虽然如此，最初的这次移植工作最终花了将近一年终于完成了，Alpha机成为了i386后第一个被官方支持的体系结构。万事开头难，这次移植的挑战性是最大的，为了提高可移植性，内核中不少代码都被重写了[⊖]。尽管这给整个移植带来了不小的工作量，可是效果是显著的，自此以后，移植变得简单轻松多了。

尽管第一个发行版只支持Intel x86，但1.2版的内核就可以支持Digital Alpha、Intel x86、MIPS和SPARC—虽然支持的不是很完善，带些试验性质。

在2.0版内核中，加入了对Motorola 68k和PowerPC的官方支持，而原1.2版支持的体系结构也纳入了官方支持的范畴，并且稳定下来了。

2.2版内核加入了对更多体系结构的支持，新增了对ARMS、IBMS390和UltraSPARC的支持。没过几年，2.4版内核支持的体系结构就达到了15个，像CRIS、IA-64、64位MIPS、HP PA-RISC、64位IBM S390和Hitachi SH都被加进来了。

当前的2.6内核把这个数字进一步提高到了20，有不含MMU的Motorola 68k、M32xxx、H8/300、IBM POWR、v850、x86-64，甚至还提供了用户模式（Usermode） Linux——一个在Linux虚拟机上运行的内核版本。对64位S390的支持和32位S390的支持放在了一起，移去了重复之处。

值得注意的是，每一种体系结构本身就可以支持不同的芯片和机型。像被支持的ARM和PowerPC等体系结构，它们就可以支持很多不同的芯片和机型。所以说，尽管Linux移植到了20中基本体系结构上，但实际上可以运行它的机器的数目要大得多。

⊖ 在内核开发中这很普遍。如果打算做一件事，那么就要把它做好。为了追求完美，内核开发者们是决不会介意重写大段代码的。

19.2 字长和数据类型

能够由机器一次就完成处理的数据被称为字。这和我们在文档中用字符（8位）和页（许多字，通常是4K或8K）来计量数据是相似的。字是指位的整数数目——比如说，1、2、4或8等。但人们说某个机器是多少“位”的时候，他们其实说的就是该机的字长。比如说，当人们说奔腾是32位芯片时，他们的意思是奔腾的字长为32位，也就是4字节。

处理器通用寄存器（GPR's）的大小和它的字长是相同的。一般来说，对于一个的体系结构来说，它各个部件的宽度——比如说内存总线——最少要和它的字长一样大。而一般来说，地址空间的大小也等于字长，至少Linux支持的体系结构中都是这样的[⊖]。此外，C语言定义的long类型总对等于机器的字长，而int类型有时会比字长小。比如说，Alpha是64位机。所以它的寄存器、指针和long类型都是64位长度的，而int类型是32位的。Alpha机每一次可以访问和操作一个64位长的数据。

字、双字的含义

有些操作系统和处理器不把它们的标准字长称作字，相反，出于历史原因和某种主观的命名习惯，它们用字来代表一些固定长度的数据类型。比如说，一些系统根据长度把数据划分为字节（byte，8位），字（word，16位），双字（double words 32位）和四字（quad words 64位），而实际上该机是32位的。在本书中——在Linux中一般也是这样——像我们前面所讨论的那样，一个字就是代表处理器的字长。

对于支持的每一种体系结构，Linux都要将<asm/types.h>中的BITS_PER_LONG定义为C long类型的长度，也就是系统的字长。表19-1是Linux支持的体系结构和它们的字长的对照表。

表19-1 Linux支持的体系结构

体系结构	描　述	字　长
alpha	Digital Alpha	64位
arm	ARM and StrongARM	32位
cris	CRIS	32位
h8300	H8/300	32位
i386	Intel x86	32位
ia64	IA-64	64位
m32r	M32xxx	32位
m68k	Motorola 68k	32位
m68knommu	m68k without MMU	32位
mips	MIPS	32位
mips64	64位MIPS	64位
parisc	HP PA_RISC	32位或64位
ppc	PowerPC	32位
ppc64	POWER	64位
s390	IBM S/390	32位或64位

⊖ 不过实际上可寻址的内存空间也可能会比字长小一些。比如，一个64位的体系结构虽然可能会提供64位的指针，但可能只会用48位来寻址。此外，如果支持Intel的PAE，那么实际的物理内存也有比字长还大的可能。

（续）

体系结构	描　述	字　长
sh	Hitachi SH	32位
sparc	SPARC	32位
sparc64	UltraSPARC	64位
um	Usermode Linux	32位或64位
v850	v850	32位
x86_64	x86-64	64位

C语言虽然规定了变量的最小长度，但是没有规定变量具体的标准长度，它们可以根据实现变化[⊖]。C语言的标准数据类型长度随体系结构变化这一特性不断引起争议。好的一面是标准数据类型可以充分利用不同体系结构变化的字长而无需明确定义长度。C语言中long类型的长度就被确定为机器的字长。不好的一面是在编程时不能对标准的C数据类型进行大小的假定，没有什么能够保障int一定和long的长度是相同的[⊜]。

情况其实还会更加复杂，因为用户空间使用的数据类型和内核空间的数据类型不一定要相互关联。sparc64体系结构就提供了32位的用户空间，其中指针、int和long的长度都是32位。而在内核空间，它的int长度是32位，指针和long的长度却是64。没有什么标准来规范这些。

牢记下述准则：

- ANSI C标准规定，一个char的长度一定是8位。
- 尽管没有规定int类型的长度是32位，但在Linux当前所有支持的体系结构中，它都是32位的。
- short类型也类似，在当前所有支持的体系结构中，虽然没有明文规定，但是它都是16位的。
- 决不应该假定指针和long的长度，在Linux当前支持的体系结构中，它们就可以在32位和64位中变化。
- 由于不同的体系结构long的长度不同，决不应该假设sizeof(int) == sizeof(long)。
- 类似地，也不要假设指针和int长度相等。

19.2.1 不透明类型

不透明数据类型隐藏了它们内部格式或结构。在C语言中，它们就像黑盒一样。支持它们的语言不是很多。作为替代，开发者们利用typedef声明一个类型，把它叫做不透明类型，希望其他人别去把它重新转化回对应的那个标准C类型。通常开发者们在定义一套特别的接口时才会用到它们。比如说用来保存进程标识符的pid_t类型。该类型的实际长度被隐藏起来了—尽管任何人都可以偷偷撩开它的面纱，发现它就是一个int。如果所有代码都不显式地利用它的长度（显式利用长度这里指直接使用int类型的长度，比如说在编程时使用sizeof(int)而不是sizeof(pid_t)—译者注），那么改变时就不会引起什么争议，这种改变确实可能会出现：在老版本的Unix系统中，pid_t的定义是short类型。

另外一个不透明数据类型的例子是atomic_t。在第9章“内核同步方法”中介绍过，它放置的

⊖ 惟一的例外是char，它的长度总是8位。

⊜ 事实上对于Linux支持的64位体系结构来说，long和int长度是不同的，int是32位的，而long是64位的。但对于我们所熟悉的32位体系结构而言，两种数据类型都是32位的。

是一个可以进行原子操作的整型值。尽管这种类型就是一个int，但利用不透明类型可以帮助确保这些数据只在特殊的有关原子操作的函数中才会被使用。不透明类型还帮助我们隐藏了类型的长度，但是该类型也并不总是完整的32位，比如在32位SPARC体系下长度被限制。

内核还用到了其他一些不透明类型，包括dev_t、gid_t和uid_t等等。处理不透明类型时的原则是：

- 不要假设该类型的长度。
- 不要将该类型转化回其对应的C标准类型使用。
- 编程时要保证在该类型实际存储空间和格式发生变化时代码不受影响。

19.2.2 指定数据类型

内核中还有一些数据虽然无需用不透明的类型表示，但它们被定义成了指定的数据类型。jiffy数目和在中断控制时用到的flags参数就是两个例子，它们都应该被存放在unsigned long类型中。

当存放和处理这些特别的数据时，一定要搞清楚它们对应的类型后再使用。把它们存放在其他如unsigned int等类型中是一种常见错误。在32位机上这没什么问题，可是64位机上就会捅娄子了。

19.2.3 长度明确的类型

作为一个程序员，你往往需要在程序中使用长度明确的数据。像操作硬件设备，进行网络通信和操作二进制文件时，通常都必须满足它们明确的内部要求。比如说，一块声卡可能用的是32位寄存器，一个网络包有一个16位字段，一个可执行文件有8位的cookie。在这些情况下，数据对应的类型应该长度明确。

内核在<asm/typs.h>中定义了这些长度明确的类型，而该文件又被包含在文件<linux/types.h>中。表19-2有完整的清单。

表19-2 长度明确的数据类型

类 型	描 述
s8	带符号字节
u8	无符号字节
s16	带符号16位整数
u16	无符号16位整数
s32	带符号32位整数
u32	无符号32位整数
s64	带符号64位整数
u64	无符号64位整数

其中带符号的变量用的比较少。

这些长度明确的类型大部分都是通过typedef对标准的C类型进行映射得到的。在一个64位机上，它们看起来像：

```
typedef signed char s8;
typedef unsigned char u8;
typedef signed short s16;
typedef unsigned short u16;
typedef signed int s32;
```

```
typedef unsigned int u32;
typedef signed long s64;
typedef unsigned long u64;
```

而在32位机上，它们可能定义成：

```
typedef signed char s8;
typedef unsigned char u8;
typedef signed short s16;
typedef unsigned short u16;
typedef signed int s32;
typedef unsigned int u32;
typedef signed long long s64;
typedef unsigned long long u64;
```

上述的这些类型只能在内核中使用，不可以在用户空间中出现（比如，在头文件中的某个用户可见结构中出现）。这个限制是为了保护命名空间。不过内核对应这些不可见变量同时也定义了对应的用户可见的变量类型，这些类型与上面类型所不同的是增加了两个下画线前缀。比如，无符号32位整型对应的用户空间可见类型就是__u32。该类型除了名字有区别外，与u32相同。在内核中你可以任意使用这两个名字，但是如果是用户可见的类型，则必须使用带下画线前缀的名字，防止污染用户空间的命名空间。

19.2.4 char型的符号问题

C标准表示char类型可以带符号也可以不带符号，由具体的编译器、处理器或由它们两者共同决定到底char是带符号合适还是不带符号合适。

大部分体系结构上，char默认是带符号的，它可以自-128到127之间取值。而也有一些例外，比如ARM体系结构上，char就是不带符号的，它的取值范围是0～255。

举例来说，在默认char不带符号，下面的代码实际会把255而不是-1赋予i：

```
char i = -1;
```

而另一种机器上，默认char带符号，就会确切地把-1赋予i。如果程序员本意是把－1保存在i中，那么前面的代码就该修改成：

```
signed char i = -1;
```

另外，如果程序员确实希望存储255，那么代码应该如下：

```
unsigned char = 255;
```

如果你在自己的代码中使用了char类型，那么你要保证在带符号和不带符号的情况下代码都没问题。如果你能明确要用的是哪一个，那么就直接声明它。

19.3 数据对齐

对齐是跟数据块在内存中的位置相关的话题。如果一个变量的内存地址正好是它长度的整数倍，它就被称作是自然对齐的。举例来说，对于一个32位类型的数据，如果它在内存中的地址刚好可以被4整除（也就最低两位为0），那它就是自然对齐的。也就是说，一个大小为2n字节的数据类型n，它地址的最低有效位的后n位都应该为0。

一些体系结构对对齐的要求非常严格。通常像基于RISC的系统，载入未对齐的数据会导致处理器陷入（一种可处理的错误）。还有一些系统可以访问没有对齐的数据，只不过性能会下降。编写可移植性高的代码要避免对齐问题，保证所有的类型都能够自然对齐。

19.3.1 避免对齐引发的问题

编译器通常会通过让所有的数据自然对齐来避免引发对齐问题。实际上，内核开发者在对齐上不用花费太大心思—只有搞gcc的那些老兄才应该为此犯愁呢。可是，当程序员使用指针太多，对数据的访问方式超出编译器的预期时，就会引发问题了。

一个数据类型长度较小，它本来是对齐的，如果你用一个指针进行类型转换，并且转换后的类型长度较大，那么通过改指针进行数据访问时就会引发对齐问题（无论如何，对于某些体系结构会存在这种问题）。也就是说，下面的代码是错误的：

```
char dog[10];
char *p = &dog[1];
unsigned long l = *(unsigned long *)p;
```

这个例子将一个指向char型的指针当作指向unsigned long型的指针来用，这会引起问题，因为此时会试图从一个并不能被4整除的内存地址上载入32位的unsigned long型数据。

如果你能想到“我会在现实中这么做吗？”，你基本上就不会有问题了。无论如何，这种错误出现了，并且它还会发生，所以应该小心。实际编程时错误可能不会像这个例子中这么明显。

19.3.2 非标准类型的对齐

前面提到了，对于标准数据类型来说，它的地址只要是其长度的整数倍就对齐了。而非标准的（复合的）C数据类型按照下列原则对齐：

- 对于数组，只要按照基本数据类型进行对齐就可了（其实随后的所有元素自然能够对齐）。
- 对于联合，只要它包含的长度最大的数据类型能够对齐就可以了。
- 对于结构体，只要它包含的长度最大的数据类型能够对齐就可以了。

结构体还要引入填补机制，这会引出下一个问题。

19.3.3 结构体填补

为了保证结构体中每一个成员都能够自然对齐，结构体要被填补。这点确保了当处理器访问结构中一个给定元素时，元素本身是对齐的。举个例子，下面是一个在32位机上的结构体：

```
struct animal_struct {
        char dog;               /* 1字节*/
        unsigned long cat;      /* 4字节*/
        unsigned short pig;     /* 2字节*/
        char fox;               /* 1字节*/
};
```

由于该结构不能准确地满足各个成员自然对齐，所以它在内存中可不是按照原样存放的。编译器会在内存中创建一个类似下面给出的结构体：

```
struct animal _struct {
        char dog;                   /* 1字节*/
        u8 __pad0[3];               /* 3字节*/
        unsigned long cat;          /* 4字节*/
        unsigned short pig;         /* 2字节*/
        char fox;                   /* 1字节*/
        u8 __pad1;                  /* 1字节*/
};
```

填补的变量都是为了能够让数据自然对齐而加入的。第一个填充物占用了3个字节的空间，保证cat可以按照4字节对齐。这也自动使其他小的对象都被对齐了，因为它们长度都比cat要小。第二个也是最后的填充是为了填补struct本身的大小。额外的这个填补使结构体的长度能够被4整除，这样，在由该结构体构成的数组中，每个数组项也就会自然对齐了。

注意，在大部分32位系统上，对于任何一个这样的结构体，sizeof(animal_struct)都会返回12。C编译器自动进行填补以保证自然对齐。

通常你可以通过重新排列结构中的对象来避免填充。这样既可以得到一个较小的结构体，又能保证无需填补它也是自然对齐的。

```
struct animal_struct {
        unsigned long cat;          /* 4字节*/
        unsigned short pig;         /* 2字节 */
        char dog;                   /* 1字节*/
        char fox;                   /* 1字节*/
};
```

现在这个结构体只有8字节大小了。不过，不是任何时候都可以这样对结构体进行调整的。举个例子，如果该结构体是为某个标准的一部分，或者它是现有代码的一部分，那么它的成员次序就已经被定死了，虽然在内核中（缺少一个正式的ABI）相比用户空间来说，这种需求要少得多。还有些时候，你因为一些原因必须使用某种固定的次序—比如说，为了提高高速缓存的命中率进行优化时设定的变量次序。注意，ANSI C明确规定不允许编译器改变结构体内成员对象的次序[⊖]——它总是由你，程序员来决定。虽然编译器可以帮助你做填充，但是，如果使用-Wpadded flag 标志，那么将使gcc在发现结构体被填充时产生警告。

内核开发者需要注意结构体填补问题，特别是在整体使用时—这是指当需要通过网络发送它们或需要将它们写入文件的时候，因为不同体系结构之间所需要的填补也不尽相同。这也是为什么C没有提供一个内建的结构体比较操作符的原因之一。结构体内的填充字节中可能会包含垃圾信息，所以在结构体之间进行一字节一字节地比较就不大可能实现了。C语言的设计者（正确的）感觉到最好还是由程序员自己为不同的情况编写比较函数，这样才能利用到结构体次序信息。

19.4 字节顺序

字节顺序是指在一个字中各个字节的顺序。处理器在对字取值时既可能将最低有效位所在的

⊖ 如果让编译器随心所欲地改变结构体中各个对象的位置的话，现存的程序大部分都会崩溃。在C语言中，函数往往通过在结构体地址上加上偏移量来计算变量的位置。

字节当作第一个字节（最左边的字节），也可能将其当作最后一个字节（最右边的字节）。如果最高有效位所在的字节放在最高字节位置上，其他字节依次放在低字节位置上，那么该字节顺序称作高位优先（big-endian）。如果最低有效位所在的字节放在最高字节位置上，其他字节依次放在低字节位置上，那么就称作低位优先（little-endian）。

编写内核代码时不应该假设字节顺序是给定的哪一种（当然，如果你编写的是与体系结构相关的那部分代码就另当别论了）。Linux内核支持的机器中使用哪一种字节顺序的都有—甚至包括一些可以在启动的时候选择字节顺序的机器—适用性强的代码应该两种字节顺序都支持。

图19-1是高位优先字节顺序的一个实例。图19-2是低位优先字节顺序的一个实例。

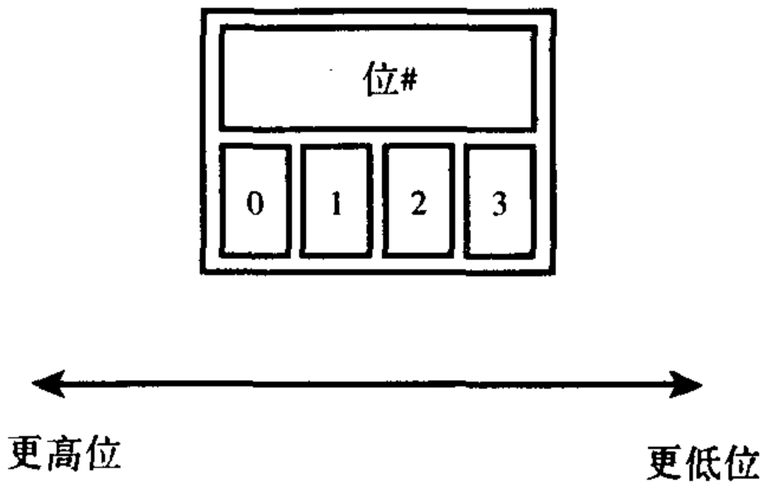


图19-1 高位优先字节顺序

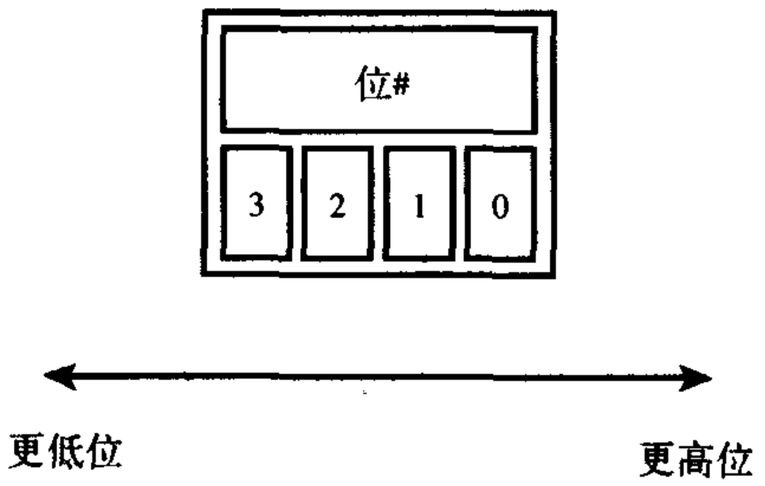


图19-2 低位优先字节顺序

i386体系结构使用的是低位优先字节顺序。而其他系统大多使用高位优先字节顺序。

让我们看看在实际编程时这些概念有什么意义。让我们考察一下存放在一个四字节的整型中的二进制数，它的十进制对应值是1027：

```
00000000 00000000 00000100 00000011
```

在内存中用高位优先和低位优先两种不同字节顺序存放时的比较如表19-3所示。

表19-3 字节顺序比较

地 址	高位优先	低位优先
0	00000000	00000011
1	00000000	00000100
2	00000100	00000000
3	00000011	00000000

注意使用高位优先的体系结构把最高字节位存放在最小的内存地址上的。这和低位优先形成了鲜明的对照。

最后一个例子，我们提供了如何判断给定的机器使用是高位优先还是低位优先字节顺序的代码：

```
int x = 1;

if (*(char *) &x ==1)
   /* 低位优先 */
else
   /* 高位优先 */
```

这段代码在用户空间和内核空间都能用。

19.4.1 高位优先和低位优先的历史

高位优先和低位优先源于乔纳森·斯威夫特写于1726年的讽刺小说《格列佛游记》。在小说中，虚构的小人国里最重要的政治问题就是应该把鸡蛋从大头敲开还是从小头敲开。那些支持从大头敲开的就是高位优先，而那些支持从小头敲开的，就是低位优先。

高位优先与低位优先的比较就好像小人国中的政治争论一样，与其说是技术问题，倒不如说是政治问题啦。

19.4.2 内核中的字节顺序

对于Linux支持的每一种体系结构，相应的内核都会根据机器使用的字节顺序在它的<asm/byteorder.h>中定义__BIG_ENDIAN或__LITTLE_ENDIAN中的一个。

这个头文件还从include/linux/byteorder/中包含了一组宏命令用于完成字节顺序之间的相互转换。最常用的宏命令有：

```
u32 __cpu_to_be32(u32);   /* 把cpu字节顺序转换为高位优先字节顺序 */
u32 __cpu_to_le32(u32);   /* 把cpu字节顺序转换为低位优先字节顺序 */
u32 __be32_to_cpu(u32);   /* 把高位优先字节顺序转换为cpu字节顺序 */
u32 __le32_to_cpus(u32);  /* 把低位优先字节顺序转换为cpu字节顺序 */
```

这些转换能够把一种字节顺序变为另一种字节顺序。如果两种字节顺序本来就相同（比如，希望从本地字节顺序转化为高位优先字节顺序，而处理器本身使用的就是高位优先字节顺序），那么宏就什么都不做。否则它们就进行转换。

19.5 时间

绝对不要假定时钟中断发生的频率，也就是每秒产生的jiffies数目。相反，应该使用Hz来正确计量时间。这一点至关重要，因为不但不同的体系结构之间定时中断的频率不同，即使是在同一种体系机构上，两个不同版本的内核之间这种频率也不尽相同。

举个例子，在x86系统上，Hz设定为1000。也就是说，定时中断每秒发生1000次，也就是每毫秒一次。可是在2.6版以前，x86上Hz定为100。而其他体系机构上的数值各不相同：alpha的Hz是1024而ARM的Hz是100。

绝对不要用jiffies直接去和1000这样的数值去比较，认为这样做大体上不会出问题是要不得的。计量时间的正确方法是乘以或除以Hz。比如：

```
HZ                  /* 1秒 */
(2*HZ)              /* 2秒 */
(HZ/2)              /* 半秒 */
(HZ/100)            /* 10毫秒 */
(2*HZ/100)          /* 20毫秒 */
```

HZ定义在文件<asm/param.h>中，在前面的第10章“定时器与定时管理”中曾经讨论过。

19.6 页长度

当处理用页管理的内存时，绝对不要假设页的长度。在x86下编程的程序员往往错误地认为一

页的大小就是4KB。尽管x86机器上使用的页确实是4KB，但是其他不同的体系结构使用的页长度可能不同。实际上有些体系结构还同时支持多种不同长度的页。表19-4列举了各种体系结构使用的页的长度。

当处理用页组织管理的内存时，通过PAGE_SIZE来使用以字节数来表示的页长度。而PAGE_SHIFT这个值定义了从最右端屏蔽多少位能够得到该地址对应的页的页号。举例来说，在页长为4KB的x86机上，PAGE_SIZE为4096而PAGE_SHIFT为12。它们都定义于<ams/ page.h>中。

表19-4 不同体系结构的页长度

体系结构	PAGE_SHIFT	PAGE_SIZE
alpha	13	8KB
arm	12,14,15	4KB,16KB,32KB
cris	13	8KB
h8300	12	4KB
i386	12	4KB
ia64	12,13,14,16	4KB, 8KB, 16KB, 64KB
m32r	12	4KB
m68k	12,13	4KB, 8KB
m68knommu	12	4KB
mips	12	4KB
mips64	12	4KB
parisc	12	4KB
ppc	12	4KB
ppc64	12	4KB
s390	12	4KB
sh	12	4KB
sparc	12,13	4KB, 8KB
sparc64	13	8KB
v850	12	4KB
x86_64	12	4KB

19.7 处理器排序

回忆第9章，其中讨论过体系结构对指令序列的排序问题。有些处理器严格限制指令排序，代码指定的所有装载或存储指令都不能被重新排序；而另外一些体系结构对排序要求则很弱，可以自行排序指令序列。

在代码中，如果在对排序要求最弱的体系结构上，要保证指令执行顺序。那么就必须使用诸如rmb()和wmb()等恰当的内存屏障来确保处理器以正确顺序提交装载和存储指令。详情请参见第9章。

19.8 SMP、内核抢占、高端内存

在讨论可移植性的地方加入有关并发处理、内核抢占和高端内存的部分看起来似乎不太恰当。毕竟，这些都不是会影响到操作系统的硬件之间有所差异的那些特性，恰恰相反，它们都是Linux内核本身的一些功能，硬件体系结构根本感知不到它们的存在。但是，它们代表的其实都是可配

置的重要选项，而你的代码应该充分考虑到对它们的支持。就是说，只有在编程时就针对SMP/内核抢占/高端内存进行了考虑，你的代码才会无论内核怎样配置，都能身处安全之中。再在前面那些保证可移植性的规范下加上这几条：

- 假设你的代码会在SMP系统上运行，要正确地选择和使用锁。
- 假设你的代码会在支持内核抢占的情况下运行，要正确地选择和使用锁和内核抢占语句。
- 假设你的代码会运行在使用高端内存（非永久映射内存）的系统上，必要时使用kmap()。

19.9 小结

简而言之，要想写出可移植性好、简洁、合适的代码，要注意以下两点：

- 编码尽量选取最大公因子：假定任何事情都可能发生，任何潜在的约束也都存在。
- 编码尽量选取最小公约数：不要假定给定的内核特性是可用的，且仅仅需要最小的体系结构功能。

编写可移植的代码需要考虑许多问题：字长、数据类型、对齐、字节次序、页大小、处理器排序等等。对于绝大多数内核开发来说，可能主要考虑的问题就是保证正确使用数据类型，虽然如此，说不定有朝一日，还是会有些与古老的体系结构有关的问题突然跳出来开始困扰你。所以说理解移植性的重要性，并且在开发内核过程中时刻注意编写简洁、可移植的代码是非常重要的。

第20章

补丁、开发和社区

Linux最大的一个优势就是它有一个紧密团结了众多使用者和开发者的社区。社区能帮你检查代码，社区能帮你进行测试，社区还能向你反馈存在的问题。此外，什么样的代码可以加入内核也是由社区做出决定的。因而了解这些到底是怎么运作的就显得尤为重要了。

20.1 社区

如果一定要让Linux内核社区在现实世界中找到它的位置，那它也许会叫做内核邮件列表（Linux Kernel Mailing List）之家。内核邮件列表（或者简写成lkml）是对内核进行发布、讨论、争辩和打口水仗的主战场。在做任何实际的动作之前，新特性会在此处被讨论，新代码的大部分也会在此处张贴。这个列表每天发布的消息超过300条，所以决不适合心血来潮的顽主。任何想踏踏实实研究，认认真真开发内核的人都应该订阅它（至少要订阅它的摘要或者是它的归档资料）。单单看看这些奇才们使出的一招一式，也能让你受益匪浅了。

你可以通过发送下面的消息进行订阅：

```
subscribe linux-kernel <your@email.address>
```

只要用纯文本向majordomo@vger.kernel.org 发送就可以了。关于这方面更为详细的信息可以在http://vger.kernel.org/中找到，此外在http://www.tux.org/lkml/，还有一个专门的FAQ。

网上还有无数与内核相关或与普通的Linux使用相关的资源。http://www.kernelnew-bies.org是一方适合内核开发初级黑客的乐土—该网站几乎能够满足所有磨刀霍霍向内核的新手的需求。还有两个网站也是不错的资源，包括http://www.lwn.net/，Linux新闻周刊，它有一个专区报道有关内核的重要新闻；http://www.kerneltraffic.org，内核直通车，它有前一周内核邮件列表的摘要，并且伴有对其一针见血的评论。

20.2 Linux编码风格

像所有其他大型软件项目一样，Linux制定了一套编码风格，对代码的格式、风格和布局做出了规定。这么做不是因为Linux内核的风格有多么出众（可能确实还不错）或是你自己原来的风格有多么拙劣（不过这也说不定），而是因为保持编码风格的一致有助于提高编程效率。然而对规定编码风格还是存在一些争议，有人认为这其实无关紧要，因为无论如何，最终编译出来的目标码不会受影响。在像内核这样的大型软件项目中，涉及许许多多的开发者，编码的一致性变得至关重要。一致意味着相似和熟悉，也就意味着容易读懂，不含歧义，并且以后的代码仍旧会保持这种风格。这种做法能让更多人读懂你的代码，也让你读懂更多其他人编写的代码。在开源项目中，眼球自然是越多越好。

跟选择一个惟一确定的风格相比，到底选择什么样的风格反而显得不是那么重要了。好在Linus早就展示出了该用什么风格，而且绝大部分代码都照这么做了。编码风格的主要规范伴随着Linus一贯的幽默，都记录在内核源代码树的Documentation/CodingStyle中了。

20.2.1 缩进

内核的缩进风格是用制表符（tab）每次缩进八个字符长度。这不是说用八个、四个或者其他什么数目的空格就行了。这里的规定很明确，每次缩进通过制表符进行，每个制表符八个字节长度。不知为什么，虽然违反它会对可读性带来非常大的冲击，但这个规定还是最容易被人们违反。八个字符长度的缩进能让不同的代码块看起来一目了然，特别是在连续几个小时的开发之后，效果更加明显。

20.2.2 括号

括号的使用不存在技术上的差异，完全是个人喜好问题，但我们还是必须宣传一致的风格。内核选定的风格是左括号紧跟在语句的最后，与语句在相同的一行。而右括号要新起一行，作为该行的第一个字符。例如：

```
if (fox) {
        dog();
        cat();
}
```

注意，如果接下来的部分是相同语句的一部分，那么右括号就不单独占一行，例如：

```
if (fox) {
        ant();
        pig();
} else {
        dog();
        cat();
}
```

还有，

```
do {
        dog();
        cat();
} while (fox);
```

函数不采用这样的书写方式，因为函数不会在内部嵌套定义：

```
unsigned long func(void)
{
  /* ... */
}
```

最后，不需要一定使用括号的语句可以忽略它：

```
if (foo)
        bar();
```

所有这些方法原理都源自K&R[⊖]。

20.2.3 每行代码的长度

在内核中编码，要尽可能地保证每行代码长度不超过80个字符，因为这样做可使代码最合适在标准的80x24的终端上显示。事实上，并不存在一个广泛接受的标准——如果代码行超过80应该折到下一行。有些开发者也许根本不理会代码跨行问题，而是让编辑器以可读的方式处理代码的显示；而也有些开发者会手动插入断行符来分割代码行，他们也许会在新行头插入两个tab键以便和原先行错开。

类似的，有些开发者会在圆括弧内来分行，以便对齐排列函数参数。比如

```
static void get_pirate_parrot(const char *name,
                              unsigned long disposition,
                              unsigned long feather_quality)
```

而另一些开发者虽然也会将参数分行输入，但却不会把它们对齐排列，而是在开头简单地加入两个标准tab。比如：

```
int find_pirate_flag_by_color(const char *color,
                              const char *name, int len)
```

因为分行没有确定的规则，所以开发者在这点上可采取自由行动。

20.2.4 命名规范

名称中不允许使用混合的大小写字符。局部变量如果能够清楚地表明它的用途，那么选取idx甚至是i这样的名称都是可行的。而像theLoopIndex这样冗长繁复的名字不在接受之列。匈牙利命名法（在变量名称中加入变量的类别）危害极大，绝对不允许使用—要知道这里是C，不是Java，用的是Unix，不是Windows。

而全局变量和函数应该选择包含描述性内容的名称。给一个全局函数起名为atty()会使人迷惑；而像get_active_tty()这样就比较容易让人接受了。这里是Linux，不是BSD。

20.2.5 函数

根据经验，函数的代码长度不应该超过两屏，局部变量不应超过十个。一个函数应该功能单一并且实现精准。将一个函数分解成一些更短小的函数的组合不会带来危害。如果你担心函数调用导致的开销，可以使用inline关键字。

20.2.6 注释

代码的注释非常重要，但注释必须按照正确的方式进行。一般情况下，你应该描述的是你的代码要做什么和为什么要做，而不是具体通过什么方式实现的。怎么实现应该由代码本身展现。如果你不是这样做的，那么你应该回过头去考虑一下你写的东西了。此外，注释不应该包含谁写

⊖ 《C语言程序设计（第二版 新版）》由Brian Kernighan和Dennis Ritchie著，这两位作者简称K&R，该书是C语言圣经，由C语言的发明者和他的同事合著。

了那个函数，修改日期和其他那些琐碎而无实际意义的内容。这些信息应该集中在文件最开头的地方。

虽然gcc也支持C++风格的注释符号，但内核只使用C风格的注释符号。内核中一条注释看起来如下：

```
/*
 * get_ship_speed() - return the current speed of the pirate ship
 * We need this to calculate the ship coordinates. As this function can sleep,
 * do not call while holding a lock.
 */
```

在注释中，重要信息常常以“XXX:”开头，而bug通常以“FIXME”开头，就像：

```
/*
 * FIXME: We assume dog == cat which may not be true in the future
 */
```

内核包含一套自动文档生成工具。它源自GNOME-doc，略加修改后被命名为Kernel-doc。如果想要生成独立的HTML格式文档，运行：

```
make htmldocs
```

如果想要postscript格式的话，用下列命令：

```
make psdocs
```

你也可以按照特定的格式对你的函数进行注解，这样该工具也可以为你的函数服务：

```
/**
 * find_treasure-find 'X marks the spot'
 * @map-treasue map
 * @time - time the  treasure was hidden
 *
 * Must call while holding the pirate_ship_lock.
 */
void find_treasure(int map, struct timeval *time)
{
  /* ... */
}
```

有关此方面更多的细节请参看Documentation/kernel-doc-nano-HOWTO.txt文件。

20.2.7 typedef

内核开发者们出于某些考虑，常常反对使用typedef语句。他们的理由是：

- typedef掩盖了数据的真实类型。
- 由于数据类型被隐藏起来了，所以很容易因此而犯错误，比如以传值的方式向栈中推入结构。
- 使用typedef往往是因为想要偷懒。（有些程序员往往是为了少敲打几次键盘而使用typedef，比如typedef unsigned char uchar。而这种缩写可能会引发理解和一致性上的问题，所以仅仅出于此目的而使用typedef被作者视为懒惰行为。——译者注）

无论如何，就算是为了别惹人耻笑吧，尽量少用typedef。

当然，typedef也有它施展身手的时候：当需要隐藏变量与体系结构相关的实现细节的时候，当某种类型将来有可能发生变化，而现有程序必须要考虑到向后兼容问题的时候，都需要typedef。使用typedef要谨慎，只有在确实需要的时候再用它，如果仅仅是为了少敲打几下键盘，别使用它。

20.2.8 多用现成的东西

请勿闭门造车。内核本身就提供了字符串操作函数、压缩函数和一个链表接口，所以请使用它们。

不要为了使现存接口更通用化而对它们进行新的封装。你经常会发现，当把一段代码从某个操作系统移植到Linux上的时候，表面好像看起来根本没什么问题，可是隐藏在接口下面复杂的函数调用却往往是与它原有的内核相关的。没人愿意面对这些问题，所以请直接使用内核提供的接口。

20.2.9 在源码中不要使用ifdef

我们不赞成在源码中使用ifdef预处理指令。你绝不应该在自己的函数中使用如下的实现方法：

```
    ...
#ifdef CONFIG_FOO
    foo();
#endif
    ...
```

相反，应该采取的方法是在CONFIG_FOO没定义的时候让foo()函数为空。

```
#ifdef CONFIG_FOO
static int foo(void)
{
    /* .. */
}
#else
static inline int foo(void) { }
#endif
```

这样，你在任何情况下都能调用foo()了。让编译器去做这些工作好了。

20.2.10 结构初始化

结构初始化的时候必须在它的成员前加上结构标识符。这种初始化能避免错误地使用其他结构而引发一个初始化错误。它也支持使用忽略值。不幸的是，C99标准改用了一种丑陋的格式来表示这种标识符，于是gcc就再也不支持原来GNU风格的标识符了，尽管它看起来确实要更帅一些。结果，内核代码现在必须都要使用新的C99标识符格式了，不管它有多难看：

```
struct foo my_foo = {
.a      = INITIAL_A,
.b      = INITIAL_B,
};
```

其中a和b是结构体foo的成员而INITIAL_A和INITIAL_B是它们对应的初始值。如果一个字段没有给初始值，那么它就会被设置为ANSI C规定的默认值（如指针被设为NULL，整型被设为0，

浮点数被设置为0.0)。举例来说，如果foo结构体还有一个int型的c成员，那么上面的初始化语句执行之后c会被设置为0。

诚然，它很丑陋；然而，我们别无选择。

20.2.11 代码的事后修正

即使你得到了一段与内核编码风格风马牛不相及的代码，也不用发愁。只稍微抬抬手，indent工具就能帮你解决它。indent是一个在大多数Linux系统中都能找到的好工具，它可以按照指定的方式对源代码进行格式化。默认情况下它按照不怎么好看的GNU编码风格格式化代码。想要用Linux内核编码风格，只要

```
indent -kr -i8 -ts8 -sob -l80 -ss -bs -psl <file>
```

这样就能调用该工具按内核编码风格对你的代码进行格式化了。此外，还可以通过scripts/ Lindent自动按照所需的格式调用indent。

20.3 管理系统

内核黑客就是那些从事内核开发工作的人。做这些工作有些人是因为钱，有些人是因为嗜好，但几乎所有人都是为了从中找到快乐。所有做出卓越贡献的黑客都能在源代码树根目录上的CREDITS文件中留名。

内核中几乎每个部分都对应一个维护者。维护者是指一个或几个对内核特定部分负责的人。比如，每个单独的驱动程序都对应一个维护者。每个内核子系统——如网络——也有一个维护者。驱动程序和子系统的维护者也能在源代码树根目录上的MAINTAINERS文件中找到。

还有一类特殊的维护者被称作内核维护者。这些人负责维护的实际上就是代码树本身。以前，由Linus自己负责维护开发版的内核（乐趣尽在此中），稳定版最开始的一段时间也会由他来维护。等到该内核稳定下来了，他就会把火炬传递给最好的内核开发者中的一些人手上。由这些人负责维护该代码树，而Linus会转身启动下一开发版本的内核开发工作。通过这种模式，2.0、2.2和2.4版的内核现在还仍旧有人维护。

别信那些谣言，根本没有什么内核小团体在搞什么阴谋。

20.4 提交错误报告

如果你碰到了一个bug，最理想的应对无疑是写出修正代码，创建补丁，测试后提交它，这个流程在下一小节会仔细介绍。当然，你也可以报告这个问题，然后让其他人替你解决。

提交一个错误报告最重要的莫过于对问题进行清楚的描述。要讲清楚症状、系统输出信息、完整并经过解码的oops（如果有的话）。更重要的是，你应该尽可能地提供能够准确地重现这个错误的步骤，并提供你的机器的硬件配置基本信息。

然后再来考虑把这个错误报告发送给谁。MAINTAINERS文件中列举出了所有相关的设备驱动程序和子系统的维护者——这些人应当接收并解决关于其所维护的代码的所有问题。如果你找不到对此问题感兴趣的人，那么就把它报告给位于linux-kernel@vger.kernel.org的内核邮件列表。即使你已经找到维护者了，贴一份副本在那里也不会有什么坏处。

文档REPORTING-BUGS和Documentation/oops-tracing.txt中有更多相关信息。

20.5 创建补丁

所有对Linux内核的修正都是通过补丁这种方式完成的，它其实是GNU diff(1)程序的一种特定格式的输出，该格式的信息能够被patch(1)程序所能理解。创建补丁最简单的办法是通过两份内核源代码进行的，一份码，另一份是加进了你所修改部分的源代码。一般会给原来的内核代码起名linux-x.y.z（其实就是把源代码包解压缩后所得到的文件夹），而你修改过的就起名为linux。然后利用下面的命令通过这两份代码创建补丁：

```
diff -urN linux-x.y.z/ linux/ > my-patch
```

你可以在自己的目录下运行该命令，一般都是在你的home目录下，而不是在/usr/src/linux目录下进行这种操作，所以你不一定必须具备超级用户权限。通过-u参数指定使用特殊的diff输出格式。否则得到的patch格式怪异，一般人都没法看懂。-r参数保证会遍历所有子目录进行操作，而-N参数指明做出修改的源代码中所有新加入的文件在diff操作时会被包含在内。另外，如果你希望得到一个单独的文件，你也可以这么做：

```
diff -u linux-x.y.z/some/file linux/some/file > my-patch
```

注意，在你自己代码所在的目录下执行diff很重要。这样创建的补丁别人用起来更方便，哪怕他们的目录名字叫differ也没问题。执行一个这样生成的补丁，只需要在你自己代码树的根部：

```
patch -p1 < ../my-patch
```

就可以了。在这个例子中，补丁的名字叫my-patch，它位于当前目录的上一级目录中。-p1参数用来剥去补丁中头一个目录的名称。这么做的好处是可以在打补丁的时候忽略创建补丁的人的目录命名习惯。

diffstat是一个很有用的工具，它可以列出补丁所引起的变更的统计（加入或移去的代码行）。输出关于你的补丁的信息，执行

```
diffstat -p1 my-patch
```

在你向lkml贴出自己的补丁时，附带上这份信息往往会很有用。由于patch(1)会忽略第一个diff之前的所有内容，所以你甚至可以在patch的最前面直接加上简短的说明。

20.6 提交补丁

补丁可以按照前一小节描述的方式创建。如果补丁涉及了某个特定的驱动程序或子系统，应该把它发给MAINTAINER中列举的相关部分的维护者。此外，还应该向Linux内核邮件列表linux-kernel@vger.kernel.org发送一份拷贝。只有在经过广泛的讨论之后，或者是补丁所做的修改很细微并且很容易就能保证正确的时候，才应该向内核维护者（比如说Linus）提交。

一般包含一份补丁的邮件，它的主题一栏内容应该以“[PATCH] 简要说明”的格式写出。邮件的主体部分应该描述所做的改变的技术细节，以及为什么要做这些的原因。越详细越准确越好。在email中还要注明补丁对应的内核版本。

内核开发者们都希望能通过邮件阅读补丁，并且能够将其保存为一个单独文件。因此，最好把你的补丁直接插入邮件，放在所有信息的最后。还要小心一些，差劲的邮件客户端工具，它们会加入信息或者改变邮件的格式；这会导致补丁出错，从而引起其他开发者的不满。如果你用的邮件客户端工具也有类似表现，就检查一下，看它是否有“Insert Inline”、“Preformat”或类似功

能。如果有的话，就用纯文本方式把你的补丁贴到邮件上作为附件，不要对它做什么编码工作。

如果你的补丁很大或者包含对几个不同的逻辑的修改，那么应该将你的补丁分成几块，每块对应一个逻辑。比如你在补丁中引入了一个新的API，并且同时对几个驱动程序进行了修改以便利用它，那么你应该把该补丁一分为二（先是新的API，然后是对驱动程序的修改），邮件也写成两份。如果任何一个部分需要其他的补丁先行，要明确地注明这一点。

提交之后，保持耐心，等待答复。别因为某些反对言论而灰心—至少你还是得到回应了嘛！和其他人讨论这个问题并且在需要的时候应该提交修正过的新补丁。如果你压根就没听到回声，想想是什么出了问题，然后着手解决它。多请邮件列表和维护者们提出宝贵意见。运气好的话，下个版本的内核发行时，你可能就会看到自己做出的修改了——那可就真的该恭喜你了！

20.7 小结

对于骇客而言，最可贵的品质便是自发行动——就如身上痒痒，不抓不快一般的自发行动。本书讲述了Linux内核的主要部分，讨论了接口、数据结构、算法和原理。从实践出发，以内在的视角洞悉内核，既可以满足你的好奇心，也可以帮助你开始学习内核。

不过，正如我前面所说，你上路的惟一方法是去自己读、写代码。Linux社区不但创造了这样的条件，而且很欢迎大家这么做。好了，不管你追求什么，现在就开始去做吧。

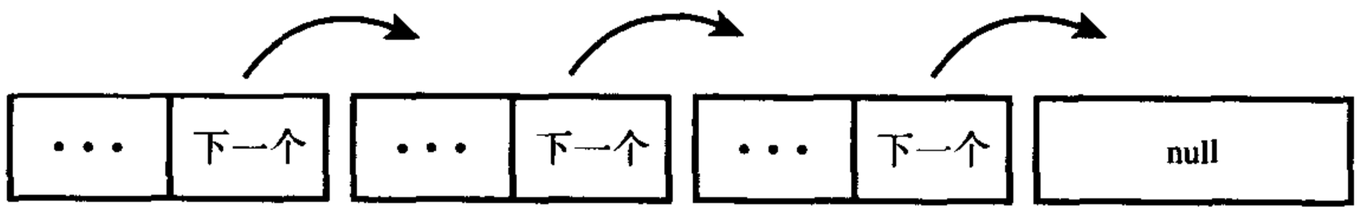
附录 A
链　表

链表是一种存放和操作可变数量元素（常称为节点）的数据结构。链表和数组不同之处在于，它所包含的元素都是动态创建的，在编译时不必知道具体需要创建多少个元素；另外也因为链表中的每个元素的创建时间各不相同，所以它们在内存中无需占用连续内存区。也正是因为元素不连续地存放，所以各元素需要通过某种方式被连接在一起。于是每个元素都包含一个指向下一个元素的指针，当有元素加入链表或从链表中删除元素时，简单的调整指向下一个节点的指针就可以了。

可以用一种最简单的数据结构来表示这样一个链表：

```
/*链表中只有一个元素 */
struct list_element {
        void *data;                         /* 有效数据 */
        struct list_element *next;          /* 指向下一个元素的指针 */
};
```

图A-1　描述一个链表结构体：



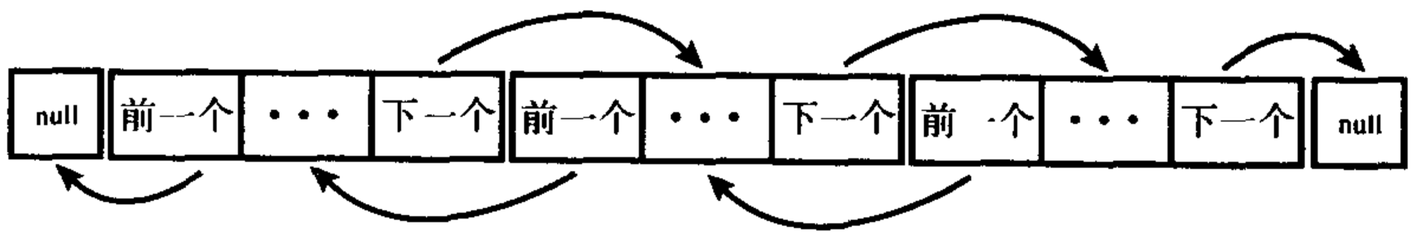
图A-1　一个简单链表

在有些链表中，每个元素还包含一个指向前一个元素的指针，因为它们可以同时向前和向后相互连接，所以这种链表被称作双向链表。而类似于图A-1所示的那种只能向前连接的链表被称作单向链表。

表示双向链表的一种数据结构如下：

```
/* 链表中只有一个元素 */
struct list_element {
        void *data;                         /* 有效数据 */
        struct list_element *next;          /* 指向下一个元素的指针 */
        struct list_element *prev;          /* 指向前一个元素的指针 */
};
```

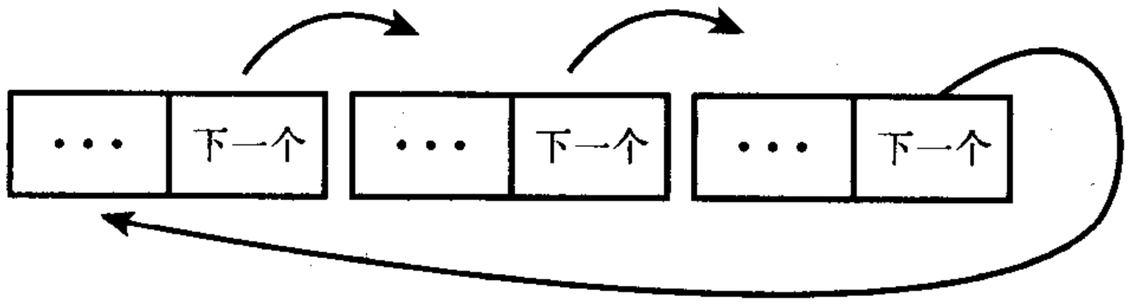
图A-2描述了一个双向链表。



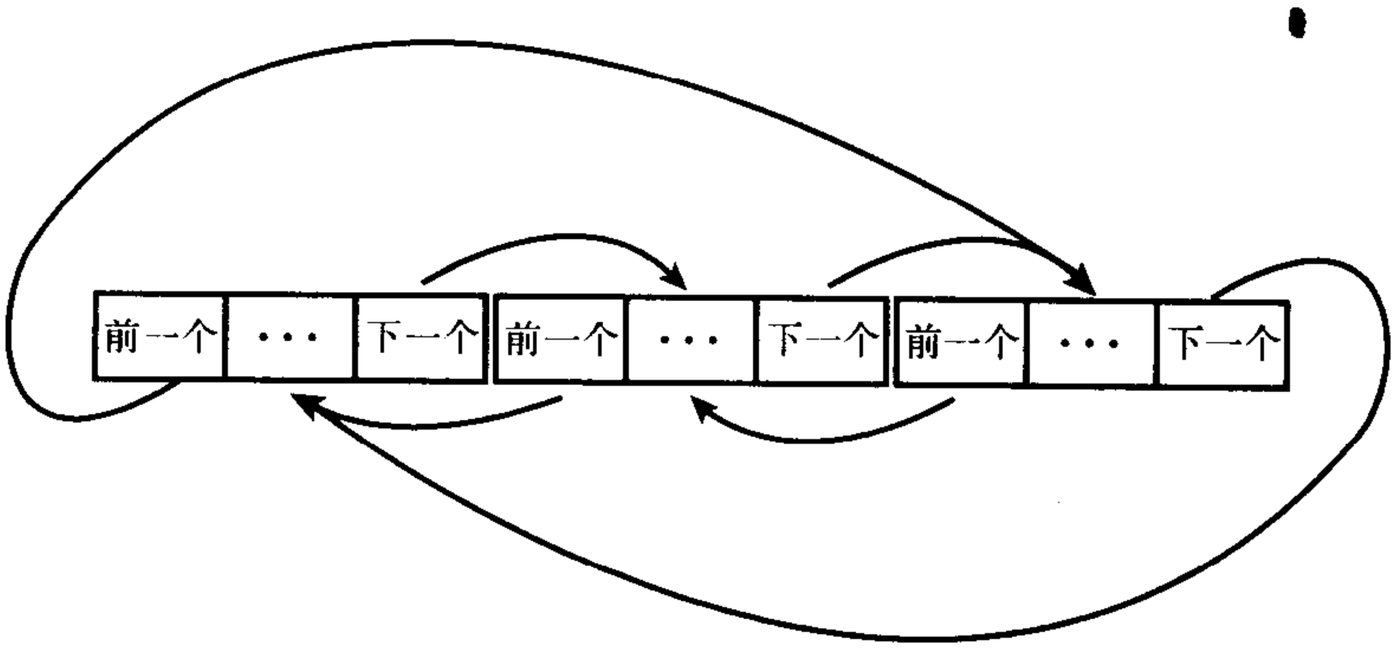
图A-2　一个双向链表

A.1 环形链表

通常情况下，因为链表中最后一个元素不再有下一个元素，所以将链表尾元素中指向后的指针设置为NULL，以此表明它是链表中的最后一个元素。但在有些链表中，末尾元素并不指向特殊值，相反，它指回链表的首元素。这种链表因为首尾相连，所以被称为是环形链表。环形链表也存在双向链表和单向链表两种形式。在环形双向链表中，首节点的向前指针指向尾节点。图A-3和A-4分别表示单向和双向环形链表。



图A-3 环形单向链表



图A-4 环形双向链表

因为环行链表提供了最大的灵活性，所以Linux内核的标准链表就是采用环形双向链表形式实现的。

沿链表移动

沿链表移动只能是线性移动。先访问某个元素，然后沿该元素的向后指针访问下一个元素，不断重复这个过程，就可以沿链表向后移动了。这是一种最简单的沿链表移动方法，也是最适合访问链表的方法。如果需要随机访问数据，一般不使用链表。使用链表的存放数据的理想情况是，需要遍历所有数据或需要动态加入和删除数据时。

有时，首元素会用一个特殊指针表示——该指针被称为头指针，利用头指针可方便、快速地找到链表。在非环形链表里，向后指针指向NULL的元素是尾元素，而在环形链表里向后指针指向头元素的元素是尾元素。遍历一个链表需要线性地访问从第一个元素到最后一个元素之间的所有元素。对于双向链表来说，也可以向后遍历链表，可以从最后一个元素线性访问到第一个元素。当然还可以从链表中的指定元素开始向前和向后访问数个元素，并不一定要访问整个链表。

A.2 Linux 内核中链表的实现

Linux内核使用了一种独一无二的方法遍历链表。在遍历链表时，如果没有要求访问顺序的话，

那么是否从链表头元素开始访问无关紧要。事实上，从任何元素开始访问都可以！重要的是，要遍历每一个元素。其实，我们甚至可以抛开头节点和尾节点这些概念，如果环形链表包含的是一些无序数据的话，那么任何一个元素都可作为头节点。遍历链表，仅仅需要从某个节点开始，沿指针访问下一个节点，直到又重新回到最初这个节点就行了，因此并不需要有一个特殊的头节点，这同时也使得链表的操作例程变得很简单。每个例程仅需要一个指向链表中某个元素的指针——任何元素，就可操作链表了。为了这种巧妙设计，内核骇客们还是颇有点自豪的。

链表在内核中使用非常频繁，其实任何其他复杂程序也都会用到它。比如内核就是用链表来存放任务队列的（每个进程的task_struct结构体都作为链表的一个元素）。

链表结构体

过去，内核中存在许多种链表的实现方法，所以需要一种功能完善的链表实现来统一它们，避免代码冗余。自2.1内核开发版本以来，就引入了正式的链表实现。目前所有链表应用都已经使用了正式的链表实现，并且所有的新开发者都应该使用这些已有的链表接口，别螳臂挡车，阻挡潮流了。

链表结构体定义在文件<linux/list.h>中，形式很简单：

```
struct list_head{
        struct list_head  *next;
        struct list_head  *prev;
};
```

注意list_head这个奇怪的名字，这个名字暗示了链表不存在头节点。但是由于可以从链表中的任何一个节点开始遍历，所以任何一个节点都可起到头节点的效果，因此每个独立节点都可以被称作是链表头。

向后指针指向链表中的下一个元素，向前指针指向链表中的前一个元素。如果是链表中的尾元素，那么它的向后指针指向首节点；如果是首元素，那么向前指针指向尾节点。但是由于内核链表的实现方法没有用到链表头的概念，所以我们可以忽略首尾节点的概念，将链表看作一个没有开始和结束的大循环结构。

一个list_head结构体本身没有什么意义；通常需要把它嵌入到你自己的结构体中：

```
struct my_struct {
        struct list_head list;
        unsigned long dog;
        void *cat;
};
```

链表在使用前需要先初始化。由于多数元素都是被动态创建（或许这也是你需要使用链表的原因），所以最常见的情况是在运行时初始化链表。

```
struct my_struct *p;
/* 分配my_struct..*/
p->dog  =  0;
p->cat  =  NULL;
INIT_LIST_HEAD(&p->list);
```

如果在编译时静态创建链表，并且直接引用它，那么可以使用如下方法：

```
struct my_struct mine = {
  .list = LIST_HEAD_INIT(mine.list),
  .dog = 0,
  .cat = NULL
};
```

直接声明和初始化一个静态链表：

```
static LIST_HEAD(fox);
```

上述语句声明并初始化名为fox的静态链表。

不需要与任何链表的内部成员打交道，你仅仅将链表结构插入到你自己的数据中就可以了。此后，你就可以使用链表提供的接口方便地操作和检索你的数据了。

A.3 操作链表

内核提供了一组函数来操作链表，这些函数都要使用一个或多个list_head结构体指针做参数。因为函数都是用C语言以内联函数形式实现的，所以它们的原形在文件include/linux/list.h中。

有趣的是，所有这些函数的复杂度都为O(1)[⊖]。这意味着，无论这些函数操作的链表大小如何，无论它们得到的参数如何，它们都在恒定时间内完成。比如，不管是对于包含三个元素的链表还是对于包含3000个元素的链表，从链表中删除一项或加入一项花费的时间都是相同的。这点可能没什么让人惊奇的，但你最好还是搞清楚其中的原因。

• 给链表增加一个节点

```
list_add(struct list_head *new, struct list_head *head)
```

该函数向指定链表的head节点后插入new节点。因为链表是循环的，而且通常没有首尾节点的概念，所以你可以将任何节点传递给head。但是如果传递最后一个元素传给head，那么该函数可以用来实现一个栈。

• 把节点增加到链表尾

```
list_add_tail(struct list_head *new, struct list_head *head)
```

该函数向指定链表的head节点前插入new节点。和list_add()函数类似，因为链表是环形的，而且可以将任何节点传递给head。但是如果传递第一个元素给head那么，该函数可以用来实现一个队列。

• 从链表删除一个节点

```
list_del(struct list_head *entry)
```

该函数从链表中删除entry元素。注意，该操作并不会释放entry或释放包含entry的数据结构体所占用的内存；该函数仅仅是将entry元素从链表中移走，所以该函数被调用后，通常还需要再销毁包含 entry的数据结构体和其中的list_head项。

• 从链表删除一个节点并对其重新初始化

```
list_del_init(struct list_head *entry)
```

该函数除了还需要再次初始化entry以外，和list_del()函数类似。这样做是因为：虽然链表不

⊖ 请看附录C，其中简要介绍了算法复杂度。

再需要entry项，但是还可以再次使用包含entry的数据结构体。

• 把节点从一个链表移到另一个链表

```
list_move(struct list_head *list, struct list_head *head)
```

该函数从一个链表中摘除list项，然后将其加入另一链表的head节点后。

• 把节点从一个链表移到另一个链表的末尾

```
list_move_tail(struct list_head *list, struct list_head *head)
```

该函数和list_move()函数一样，惟一的不同是将list项插入到head项前。

• 检查链表是否为空

```
list_empty(struct list_head *head)
```

如果指定的链表为空，该函数返回非0值；否则返回 0。

• 把两个未连接的链表合并在一起

```
list_splice(struct list_head *list,struct list_head *head)
```

该函数合并两个链表，它将list指向的链表插入到指定链表中head元素后。

• 把两个未连接的链表合并在一起，并对重新初始化原来的链表

```
list_splice_init(struct list_head *list,struct list_head *head)
```

该函数和list_splice_init()函数一样，惟一的不同是由list指向的链表要被重新初始化。

节约两次提领（dereference）

如果你碰巧已经得到了next和prev 指针，你可以直接调用内部链表操作函数，从而省下一点时间（其实就是提领指针的时间）。前面讨论的所有函数其实没有做什么其他特别的操作，它仅仅是找到next和prev指针，再去调用内部函数而已。内部函数和它们的外部包装函数同名，仅仅在前面加了两条下划线。比如，可以调用__list_del(prev, next)函数代替调用list_del(list)函数。但这只有在向前和向后指针确实已经被提领过的情况下才有意义。否则，你只是在画蛇添足。请看文件include/linux/list.h中具体的接口。

A.4　遍历链表

现在，你已经知道了如何在内核中声明、初始化和操作一个链表。这很了不起，但如果无法访问自己的数据，这些又有什么意义。链表仅仅是个能够包含你重要数据的容器；我们必须利用链表移动并访问包含我们数据的结构体。幸好，内核为我们提供了一组非常棒的接口，可以用来遍历链表和引用链表中的数据结构体。

注意，和链表操作函数不同，遍历链表的复杂度为O(n)——n就是链表所包含的元素数目。遍历链表最简单的方法是使用 list_for_each()宏，该宏使用两个list_head类型的参数，第一个参数用来指向当前项，第二个参数是需要检索的链表。每次遍历中，第一个参数在链表中不断移动，直到链表中的所有元素都被访问为止。用法如下：

```
struct list_head *p;

list_for_each(p, list) {
        /*p指向链表中的元素*/
}
```

虽然能够遍历链表，但仅仅能得到指向链表结构体的指针是没有用的，我们所需要的是一个能指向包含链表的结构体的指针。比如前面例子中的my_struct结构体，我们需要的是一个能指向每个my_struct的指针，而不是一个指向其中链表元素的指针。宏list_entry()可以帮助我们取得包含list_head的结构体，它包含三个参数：一个是指向给定的链表元素的指针，一个是其中嵌入了链表的结构体类型引用，另一个是结构体中链表成员的名称。

比如：

```
struct list_head *p;
struct my_struct *my;

list_for_each(p, &mine->list) {
        /* my指针指向嵌入链表的结构体 */
        my = list_entry(p, struct my_struct , list);
}
```

list_for_each()宏扩展开来是一个简单的循环，比如上述语句扩展开来就是：

```
for (p = mine->list->next; p !=mine->list; p=p->next)
```

惟一和上面展开循环不同的地方在于，宏list_for_each()还会使用处理器预取功能，如果处理器支持预取功能，那么该宏会预取链表中的后续的项存入内存。如果不希望执行预取操作，可以使用__list_for_each()宏只实现简单的循环。但是除非你能确定链表很小或是空的，否则都应该使用有预取功能的宏。还要注意，决不要使用循环硬编码，一定要使用内核提供的宏。

如果需要向前遍历链表，可以使用list_for_each_prev()宏，该宏沿向前指针代替向后指针执行遍历。

注意，在遍历链表时，没有什么特别的东西来阻止对链表执行删除操作。通常链表都需要用一些能避免并发访问的锁来保护。宏list_for_each_safe()使用临时存放的方法使得检索链表时不被删除操作破坏。

```
struct list_head  *p , *n;
struct my_struct *my;
list_for_each_safe(p, n, &mine->list) {
        /* my指针指向链表中的my_struct结构体 */
        my = list_entry(p, struct my_struct, list);
}
```

注意，这个宏只能防止链表删除操作，为了防止并发访问实际的链表数据，应该使用其他的锁。

附录 B

内核随机数产生器

内核实现了一个功能强大的随机数产生器，从理论上讲它能产生真随机数。随机数产生器从设备驱动收集环境噪音放入熵池。内核或用户空间进程都可以访问熵池，熵池中的数据不但是随机数，而且对外部攻击者来说是非确定的（non-deterministic）。这些随机数可被用在各种各样的程序中，尤其是加密程序。

真随机数与由函数——比如C库中的随机数产生函数——产生的伪随机数不同。伪随机数由可确定的(deterministic)函数产生，虽然随机函数可以产生有随机特征的数字序列，但是这些数字并不具备真随机数的一些特性，并非统计意义上的随机数。伪随机数是可确定的——知道序列中的一个数就可以获得其他剩下数字的有关信息。事实上，如果知道了序列的初始值（种子）通常可以确定整个序列。对于那些需要真随机和非确定数的应用程序，比如加密程序，一般不能使用伪随机数。

和伪随机数相反，真随机数的生成过程必须独立于它的生成函数。所以即使知道了真随机数序列中的一些数，外界也无法推算序列中的其他数——产生器是非确定的。

熵是一个物理学概念，这里被当作系统中不规则性和随机性的量度。熵的本意是用来测量每单位温度的能量（焦耳/卡Joules/Kelvin）。当Claude Shannon[⊖] ——信息论的奠基者——寻找一个能描述信息随机性的术语时，数学家John von Neumann[⊜]建议使用熵这个词，因为没有人真正理解它到底代表什么意思。Shannon采纳了这个建议，以至在今天有些时候也把熵也称为shannon 熵。那些后见之明的科学家发现这个词的两种意义容易混淆，更喜欢使用简单的术语不确定性uncertainty描述随机性。但是内核骇客们认为熵这个词听起来更酷，所以积极使用它。

在讨论随即数产生器期间，Shannon熵是个很重要的描述工具。通过每个符号进行测量，来决定其大小。高熵意味着字符序列中有用数据很少（但有大量随机垃圾）。内核维护着一个熵池，其中的数据取自非确定性的设备事件。理想情况下，该池中的数据是完全随机的。为了跟踪池中的熵值，内核不断测量池中数据的不确定性。每次当内核向池里填充数据，都会估计新被加入的数据的随机性。相反地当从熵池中取出数据时，内核会减少熵的估计值。这种计算被称为熵估计。如果熵估计值等于0，那么内核可以拒绝对随机数的请求。

自1.3.30版本以来，内核就引入了随机数产生器，它的实现代码在文件drivers/char/random.c中。

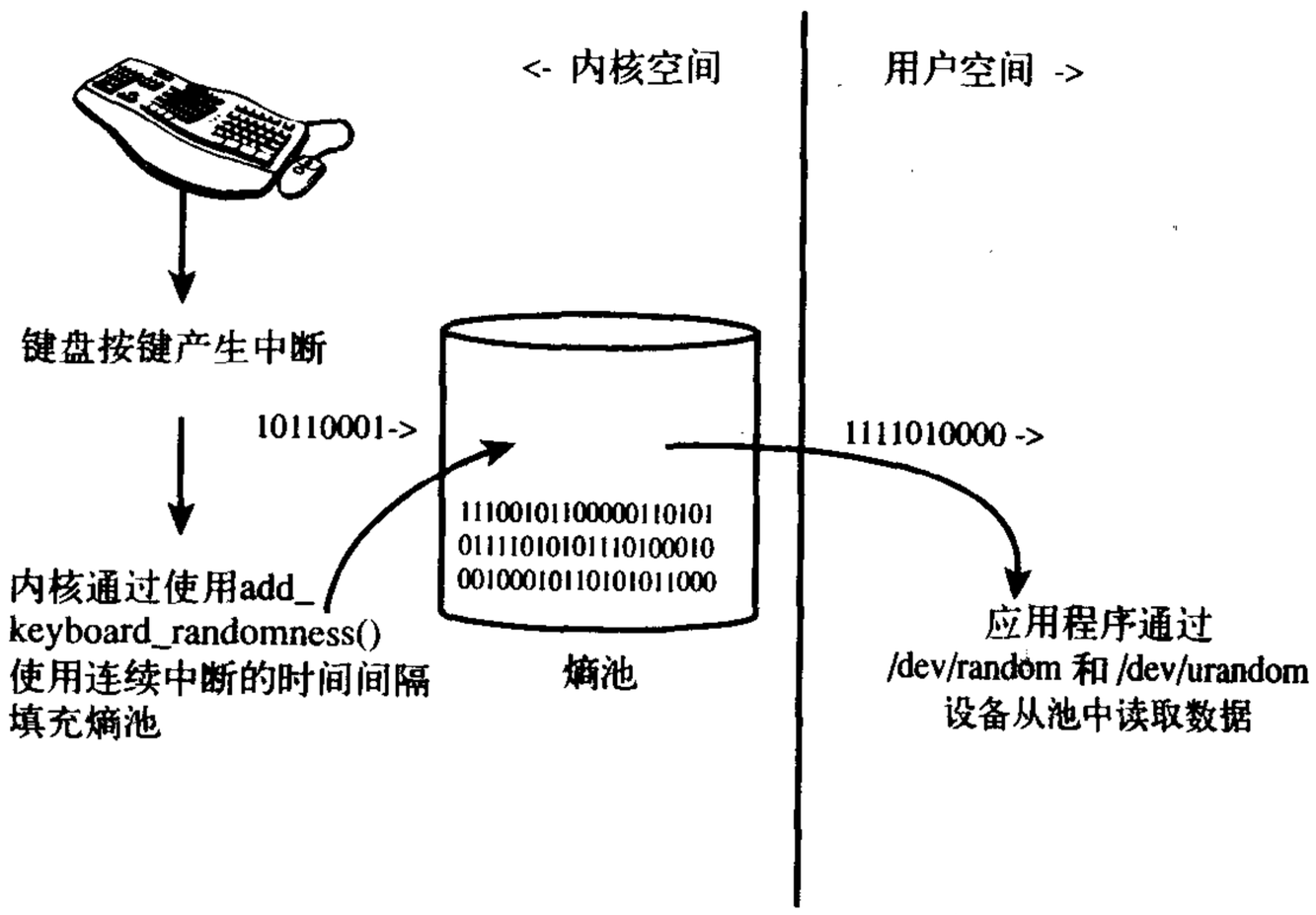
⊖ Claude E. Shannon（1916年4月30日—2001年2月24日）是贝尔试验室的一名工程师，他的最著名的贡献是于1948年出版了*Mathematical Theory of Communication*一书，该书介绍了信息理论概念和Shannon熵。另外Shannon也很喜欢骑他的独轮车。

⊜ John von Neumann（1903年12月28日—1957年2月8日）是普林斯顿高级进修学院的一员，他一生中为数学、经济学和计算机科学作出了无数贡献。其中游戏理论、诺埃曼几何和诺埃曼瓶颈最为著名。

B.1 设计和实现

计算机其实是一种可预测设备，因为它的行为完全被编程控制，所以很难在系统中找到随机数。然而机器的环境却充满了噪音，这些噪音不但易于获得而且还很大的不确定性。噪声源包括各种硬件设备的运行时速和用户与计算机交互的时速。比如敲击按键的间隔时间，鼠标的移动速度，特定中断的时间间隔和块I/O请求的响应时间等。这些对外界都属于非确定的和不可测量的噪声。因为在有新噪声被加入熵池时，内核都要估计熵池中数据的随机性并将随机性的值保存起来，所以能保证内核可以随时跟踪池中的熵值。图B-1描述了熵流如何进出熵池。

内核提供了一组接口访问熵池。利用这些接口，即可从内核中也可从用户程序访问熵池。当访问到这些接口时，内核首先计算熵池的SHA哈希值。SHA（安全哈希算法）是一种由美国国家安全局开发的消息摘要算法，并且被NIST制定为美国国家联邦标准(FIPS 186)。该摘要算法接收一个可变长度（可长可短）输入，但输出为固定长度的哈希值（通常128或160位）——从原始输入得到的摘要值。从输出的哈希值，不可重构输入消息；另外对输入的任何微小的改变都会造成完全不同得哈希值。值得一提的是，摘要算法还可用在其他不同地方，比如数据验证和指纹等。除了SHA以外，摘要算法还有MD4和MD5等。内核将熵池的SHA哈希值返回给用户，而不会将原始熵池数据返回——熵池的内容不能直接被访问。假定从返回的SHA哈希值无法推断出熵池的任何状态，所以即便知道了熵池中的一些值也无法获得以前和以后的任何值。但是如果熵池的熵为零，那么内核可以拒绝返回任何随机数。当熵被从池中读出时，熵估计就会随池中信息量下降而减少。



图B-1 进和出内核熵池的熵流

当熵估计减少到0时，内核仍然可以提供随机数，但是理论上讲，攻击者这时可以利用前面的输出推得后面的输出。当然这要求攻击者获得熵池中几乎所有先前的输出，而且要精通SHA密码分析。可是要想杜绝任何风险，就要利用判断熵估计值可以确保随机数的强度。当然对于绝大多数用户来说，并不需要这种额外的保证。

为什么要在内核中完成

在Linux内核中实现的功能都必满足一个准则——它们在用户空间无法轻易地实现。把所有事情都交给内核实现显然是不可以的。乍看随机数产生器和熵池不必在内核中实现，但是有三个条件限制，使它们必须在内核中实现。首先，随机数产生器需要得到系统间隔时间，比如中断和用户输入的间隔时间。如果不让内核导出这些接口和钩子来通知用户程序这些间隔时间，用户空间是无法得到这些的。另外即使这些数据能被导出，用户也不能简单迅速地得到它们。其次，随机数产生器必须足够安全，虽然熵池系统可以以root身份运行而且采取各种安全措施，但是内核能为熵池提供更安全地环境。最后，内核自身也需要使用随机数，从用户空间代理的服务器中获得这些数既不方便也不现实。所以随机数产生器在内核中实现最理想。

系统建立时的窘境

当内核第一次启动时，系统完成一系列几乎完全可以预测的动作。所以攻击者在启动时可以较准确地推测出熵池的状态。更糟的是，每一次启动都和下一次启动几乎相同，所以熵池每次初始化后的内容也都大致相同。这样必定会降低熵估计的准确度，因而也就没有办法确定在本次启动过程中所得到的熵的不确定性比其他启动时的不确定性高。

为了消除这个缺陷，多数Linux系统都会在关闭系统时从熵池中存储一些信息留到下次用，在下一启动时，这些数据会被读取而且被填入到熵池中。利用这种将前次熵池内容装入到当前熵池的方法，不会增加熵的可估计性。

这样，攻击者在不知道系统当前状态和系统以前状态的情况下，就不能预测熵池的状态了。

B.2 熵的输入接口

内核导出了一组接口向熵池填充数据，这些接口由某些内核子系统和驱动调用。它们是：

```
void add_interrupt_randomness(int irq)
void add_keyboard_randomness(unsigned char scancode)
void add_mouse_randomness(__u32 mouse_data)
```

如果一个中断的处理程序注册时标志为SA_SAMPLE_RANDOM，那么当该中断发生时，add_interrupt_randomness()会被中断系统调用。其中参数irq为中断号，随机数产生器利用连续中断之间的时间间隔作为噪声源，注意并非所有的设备都适合这样做；如果设备产生中断有规律可循（如时钟中断）或可能受外部攻击者影响（比如，网络设备），那么这些设备就不能用来填充熵池。相反硬盘是种可以利用的设备，因为它产生中断频率无规律可循。

add_keyboard_randomness()使用扫描码和敲击按键的时间间隔来填充熵池，有趣的是，该例程可以聪明地忽略自动重复（当用户一直按住按键时）。因为这时不管是扫描码还是按键间隔时间都是恒量，所以对熵的贡献很小。该函数惟一的参数为按键的扫描码。

add_mouse_randomness()函数使用鼠标的位置和中断间隔时间填充熵池，参数mouse_data是硬件报告的鼠标位置。

这三个例程都会给熵池提供充足的数据，计算给定数据的熵估计，而且将熵估计累加起来。

所有这些导出的接口都要使用内部函数add_timer_randomness()填充熵池。该函数计算连续两

次同一类行事件的间隔时间，并且将其加入到熵池中。比如两次连续的存盘中断间隔时间就有很大的随机性——特别是当测量很精确时，数据的低位经常就是电子噪音。在该函数填充熵池后，还要计算目前熵池中数据有多大随机性。从上次时间间隔及1阶和2阶Δ值计算这次时间间隔的1阶、2阶和3阶Δ值，从而得出随机数的大小。对这些Δ值的最大值取其最低12位则为熵估计。

B.3 熵的输出接口

内核导出了一个访问内核中的随机数的接口：

```
void get_random_bytes(void *buf,int nbytes)
```

该函数返回长度为nbytes字节的缓冲区buf。该函数不管熵估计是否为0都将返回数据，因为内核相比用户空间的加密应用程序来说，不太关心是否熵估计是否为0。大量内核任务都需要用到随机数，特别是在网络中需要用随机数为连接提供TCP初始序列号。

内核代码可以使用下列语句接收一个字长的随机数：

```
unsigned long rand;
get_random_bytes(&rand,sizeof(unsigned long));
```

内核提供了两个字符设备：/dev/random和/dev/urandom供用户空间程序使用。/dev/random适用于对随机性有很高要求的情况，比如高安全加密应用程序。它仅仅返回随机数中由熵估计标记的随机数中的最大限度的位数。如果熵估计跌到零,那么/dev/random会被堵塞，不再返回剩余数据，直到熵估计重新变为足够大的正数。相反设备/dev/urandom没有这种限制，但是通常情况下也是足够安全的。两种设备都从同一个池中返回数据。

从这两种设备文件中读取数据非常简单，下面是个用户空间函数的例子，应用程序使用该函数抽取一个字长的熵:

```
unsigned long get_random(void)
{
        unsigned long seed;
        int fd;

        fd = open("/dev/urandom",O_RDONLY);
        if(fd==-1){
              perror("open");
              return0;
        }
        if(read(fd,&seed,sizeof(seed))<0){
              perror("read");
              seed = 0;
        }
        if(close(fd))
              perror("close");

        return seed ;
}
```

另外，也可以使用dd(1)程序将bytes长的字节写入file文件中：

```
dd if=/dev/urandom of = file count = 1 bs = bytes
```

附录 C
复杂度算法

在计算机科学和相关的学科中，很有必要将算法的复杂度(或伸缩度)作为一个有意义的值进行描述（这是相对描述性不太强的术语，比如总量而言的）。虽然有着各种各样表示伸缩度的方法，但最常用的技术还是研究算法的渐近行为（asymptotic behavior），这就是当算法的输入变得非常大或接近于无限大时，算法的行为随之发生变化。渐近行为充分显示了当一个算法的输入逐渐变大时，该算法的伸缩度如何。研究算法的伸缩度（当输入增大时算法执行的变化）可以帮助我们以特定基准抽象出算法模型，从而更好的理解算法的行为。

C.1 算法

算法就是一系列的指令，它可能有一个或多个输入，最后产生一个结果或输出。比如计算一个房间中人数的步骤就是一个算法，它的输入是人，计数结果是输出。在Linux内核中，页换出和进程调度都是算法的例子。从数学角度讲，一个算法好比一个函数（或至少我们可将它抽象为一个函数）。比如，我们称人数统计算法为f，要统计的人数为x，可以写成下面形式：

$$y = f(x) \quad \text{人数统计的函数}$$

这里y是统计x个人所需的时间。

C.2 大O符号

一种很有用的渐近表示法就是上限——它是一个函数，其行为总是超过我们所研究的函数，也就是说上限增长快于我们研究的函数。一个特殊符号，大O（发音为大O）符号用来描述这种增长率。函数$f(x)$可写作$O(g(x))$，读为g的大O。数学定义形式为：如果$f(x)$是$O(g(x))$, 那么

$$\exists c, x' \text{ 满足 } f(x) \leqslant c \cdot g(x), \forall x > x'$$

换成自然语言就是，完成$f(x)$的时间总是短于完成$g(x)$时间和任意常量（至少，只要当输入的x值大于某个初始值x'）的乘积。

从根本上讲，我们需要寻找一个函数，它的行为和我们的算法一样差或更差。这样一来我们就可以通过给该函数送入非常大的输入，然后观察该函数的结果，从而了解我们算法的执行上限。

C.3 大θ符号

当大多数人谈论大O符号时，更准确的讲它们谈论的更接近Donald Knuth所描述的大θ符号。从技术角度讲，大O符号适合描述上限，比如 7是6的上限，同样道理9、12和65也都是6的上限。

但在后来多数讨论函数增长率时，更多说的是最小上限，或一个抽象出具有上限和下限[⊖]的函数。Knuth教授将其描述为大θ符号，并给出了下面的定义：

如果$f(x)$是$g(x)$的大θ，那么$g(x)$既是$f(x)$的上限也是$f(x)$的下限

那么，我们也可以说$f(x)$是$g(x)$级（order）。级或大θ，是理解内核中算法的最重要的数学工具之一。

所以，当人们谈到大O符号时，他们往往是在谈论大θ。当然你不用为此担心，除非你想讨Knuth教授欢心。

C.4 把时间复杂度放在一起

比如，再次考虑计算房间里的人数，假设你一秒钟数一个人，那么如果有7个人在房间里，你需要花7秒钟数他们。显然如果有n个人，需要花n秒来数他们。我们称该算法复杂度为$O(n)$。如果任务是在房间里的所有人面前跳舞呢？因为不管房间里有5个人还是有5千人，跳舞花费的时间都是相同的，所以该任务的复杂度为$O(1)$。请看表C-1给出的通常的复杂度。

表C-1 时间复杂度表

$O(g(x))$	名 称
1	恒量（理想的伸缩度）
$\log n$	对数的
n	线性的
n^2	平方的
n^3	立方的
2^n	指数的（可怕）
$n!$	阶乘（可怕）

将房间里的所有人相互介绍的复杂度是多少呢？有什么函数抽象这种算法呢？如果介绍一个人需要花费30秒，那么相互介绍10个人花多久呢？介绍100个人又需要花多久呢？

C.5 时间复杂度的危险

显然，应避免使用复杂度为$O(n!)$或$O(2^n)$的算法，另外，用复杂度为$O(1)$的函数代替复杂度为$O(n)$的函数通常都能提高执行性能。但是情况并非总是如此，不能仅仅依靠算法复杂度（大O符号）来作判断那种算法在实际使用中性能更高。回忆一下，指定的$O(g(x))$有一个恒量c和$g(x)$相乘，所以有可能复杂度为$O(1)$的算法需要花费3个小时才能完成任务，而且无论输入多大，总是要花3个小时。这样的话很可能要比复杂度为$O(n)$，但输入很少的算法费时还长。因此我们在比较算法性能时，还需要考虑输入长度。

不要沉醉于复杂度算法，要记住算法的开销还与输入长度有关。不要盲目对某些随机情况进行优化！

⊖ 如果你好奇，下限使用大omega符号模仿，其定义除了$g(x)$总是小于或等于而不是大于或等于$f(x)$外，和大O相同。大omega表示没有大O表示有趣，因为查找小于你的函数的函数，这就趣味性不大了。

参考资料

这里按主题分类列举了许多不但有趣而且很实用的书籍，这些书籍可以作为本书的补充资料。

所有这些书籍都经受了时间考验，其中有的书籍在其相关领域被冠以“圣碑”的美誉，有的书籍曾给我带来一些有趣、深刻或娱乐性的提示，我希望它们也能帮助你。

但你应该清楚，本书最好的“课外阅读”参考资料就是内核源代码。作为Linux的使用者，我们被赋予无限制地获得全部操作系统源代码的权利。别把这当作是理所当然的，不要暴殄天物，要潜心钻研它。

操作系统设计书籍

这些书覆盖了操作系统设计领域，常作为大学教科书。它们包含设计一个功能健全的操作系统所需要的概念、算法、问题和解决方法。

H.Deitel, P.Deitel, and D.Choffines: *Operating Systems*. 操作系统理论书籍；包括从理论到实践的精彩实例研究和绝技。笔者曾对该书做过技术编辑，但愿提升了它的价值。

Tanenbaum,Andrew. *Operating Systems ： Design and Implementation*.Prentice Hall ,1997。该书精辟地介绍了如何设计和实现一个类Unix操作系统——Minix。

Tanenbaum,Andrew .*Modern Operating Systems*. Prentice Hall,2001。该书深刻揭示了标准操作系统设计精髓，其中也不乏讨论许多今天流行的现代操作系统，如Unix和Windows的概念。

A.Silberschatz, P.Galvin,和GGagne .*Operating System Concepts*. John Wiley and Sons,2001 由于该书封面看起来和恐龙有关，所以也称为恐龙宝典。这是一本介绍操作系统设计的好书，而且频繁再版，版版经典。

Unix内核书籍

这些书针对Unix内核设计与实现，前三本讨论Unix特定版本，后两本关注所有Unix版本的通用特点。

Bach,Maurice.*The Design of the Unix Operating System*[⊖]. Prentice Hall,1986.该书讨论了V2版本的Unix系统。

M.McKusick,K.Bostic,M.Karels和J.Quarterman. *The Design and Implementation of the 4.4BSD Operating System* .Addison-Wesley,1996.操作系统设计者自己编写的关于设计4.4BSD系统的好书。

M.Mckusick and G. Neville-Neil: *The Design and Implementation of the Free BSD Operating System*.详细讨论了Free BSD5的设计与实现。

⊖ 中译本为《UNIX操作系统设计》，机械工业出版社出版。

J.Mauro和R.McDougall.*Solaris Internals:Core Kernel Architecture*.Prentice Hall,2000.该书对Solaris内核的核心子系统和算法进行了精彩的讨论。

C.Cooper and C.Moore: HP-UX lli *Internals*. 详细介绍HP-UX和PA-RISC内部结构。

Vahalia,Uresh.*Unix Internals:The New Frontiers* ,Prentice Hall,1995.一本关于现代Unix系统特色的优秀书籍，介绍了线程序管理和内核抢占等功能。

Schimmel,Curt. *UNIX System for Modern Architectures:Symmetric Multiprocessing and Caching for Kernel Programmers*. Addison-Wesley,1994.该书揭示了在现代体系结构上实现现代Unix操作系统可能要面对的种种危险。

Linux内核书籍

以下书籍讨论Linux内核，遗憾的是，这方面的好书不多，以下两本很好：

A.Rubini and J.Corbet. *Linux Device Drivers* . O' Reilly and Associates ,2001. 这本书是学习在2.4内核上编写驱动程序的良师益友。

D.Mosberger and S.Eranian.*IA-64 Linux Kernel:Design and Implementation* .Prentice Hall,2002.该书探讨了Intel Itanium 体系结构，也分析Linux 2.4内核在其上的实现方法。

关于其他系统内核的书

了解你的敌人——错误和竞争对手——才能处于不败之地。这些书讨论非Linux系统的设计与实现。你可以从中获得经验或教训。

M.Kogan 和 H.Deitel.*The Design of OS/2*. Addison-Wesley,1996. 该书介绍OS/2 2.0的设计。

D.Solomon 和 M.Russinovich. *Inside Windows 2000* .Microsoft Press,2000.典型的非Unix系统设计书籍。

Richter,Jeff. *Advanced Windows* .Microsoft Press ,1997.介绍Windows的系统编程和底层编程书籍。

关于Unix API的书籍应用程序

深入探索Unix系统的API函数不但对编写优秀的用户空间应用程序很关键，同时也对理解内核功能大有帮助。

Stevens W.Richard.*Advanced Programming in the UNIX Environment* .Addison-Wesley,1992.讨论Unix系统调用接口的权威书籍。

Stevens W.Richard.*UNIX Network Programming Volume 1*.Prentice Hall ,1998. 关于Unix系统套接字API的经典教材。

M.Johnson 和 E.Troan. *Linux Application Development* .Addison -Wesley.1998.关于Linux系统和其系统调用的书籍。

其他书籍

下面的书籍严格说可能和操作系统无关，但是无疑这些书从某些层面却影响着操作系统。

Knuth,Donald. *The Art of Computer Programming ,Volume 1*. Addison-Wesley,1997.一本无价的计算机科学算法基础书籍，其中介绍了内存管理中的最佳适应算法和最坏适应算法。

Hofstadter, Douglas. *Gödel, Escher, Bach: An Eternal Golden Braid* ,Basic Books, 1999。一本对人类思想进行深刻研究的必备书籍，它的内容覆盖众多主题，其中也包含计算机科学。

关于C语言的书籍

B.Kernighan 和D.Ritchie.*The C Programming Language*⊖.Prentice Hall ,1988. C语言编程的权威书籍。

Peter vom der Linden. *Expert C. Programming*. 对C语言中的难点进行了详细讨论，剖析了错误作法，文字幽默，我喜欢。

网站

以下这些网站为我们提供了及时新闻和相关Linux的资料下载，其中也包含我们挚爱的内核。

Kernel Traffic.是每周Linux内核邮件列表总结，强烈推荐。http://www.kerneltraffic.org/

Linux Weekly News. 非常好的新闻网站，评论每周内核发生的事，强烈推荐。http://www.lwn.net/

Kernel Newbies.帮助内核黑客的社区项目。http://www.kernelnewbies.org/

Kernel.org.内核官方资源库，包括大量核心内核黑客补丁。http://www.kernel.org/

kernelTrap.包括所有内核相关的内容，尤其重点包括Linux内核的内容。包括大量Linux内核开发的新闻和总结，还有大量与顶极内核开发者的访谈录。http://www.kerneltrap.org/

OS News. Operating System News，包括许多原创文章、访谈、见闻等。http://www.osnews.com/

本书的网站，包括勘误、新消息等。http://tech9.net/rml/kernel_book/

⊖ 中译本为《C程序设计语言（第2版 · 新版）》，机械工业出版社出版。

[General Information]
书名=LINUX内核设计与实现 （第2版）
作者=BEXP
SS号=
加密地址=
下载位置=http://hn9.5read.com/300-1/disknsjs/nsjs209/03/!00001.pdg