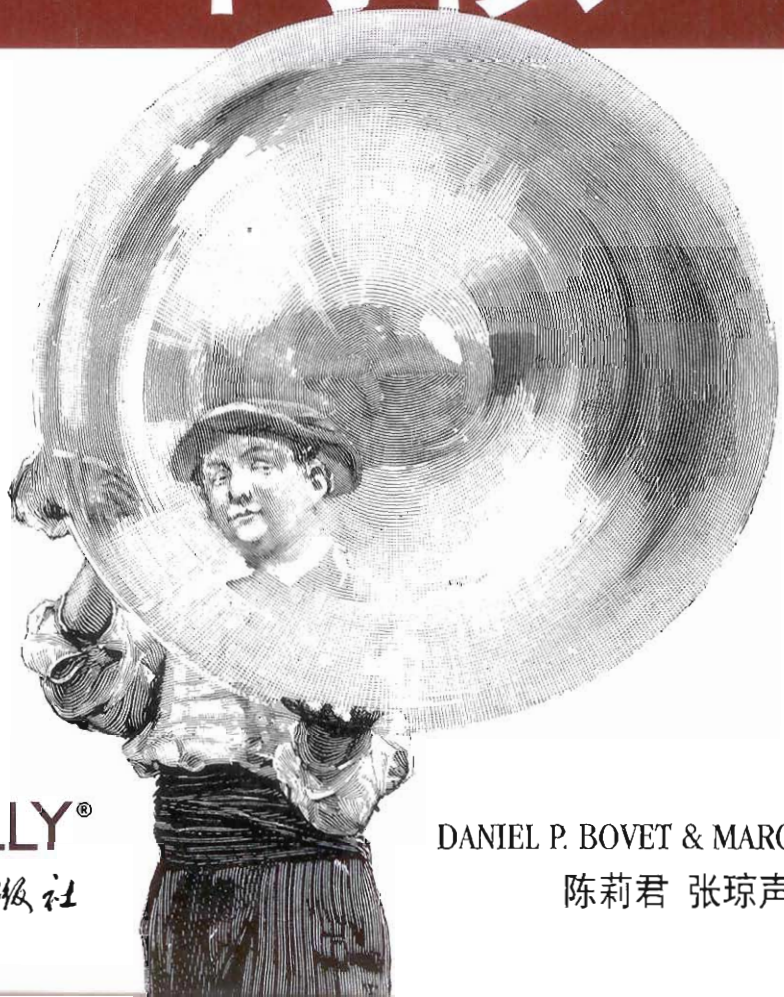
UNDERSTANDING THE LINUX KERNEL

第三版

涵盖2.6版

深入理解 LINUX 内核



O'REILLY®
中国电力出版社

DANIEL P. BOVET & MARCO CESATI 著
陈莉君 张琼声 张宏伟 译

深入理解
LINUX
内核

深入理解 LINUX 内核

第三版

Daniel P. Bovet & Marco Cesati 著

陈莉君 张琼声 张宏伟 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

深入理解Linux内核（第三版）/（美）博韦（Bovet, D. P.），西斯特（Cesati, M.）著；陈莉君，张琼声，张宏伟译．－北京：中国电力出版社，2007

书名原文：Understanding the Linux Kernel, Third Edition

ISBN 978-7-5083-5394-4

I. 深… II. ①博…②西…③陈…④张…⑤张… III. Linux操作系统 IV. TP316.89

中国版本图书馆CIP数据核字（2007）第031334号

北京市版权局著作权合同登记

图字：01-2007-1766号

©2005 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2007. Authorized translation of the English edition, 2005 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc. 出版2005。

简体中文版由中国电力出版社出版2007。英文原版的翻译得到O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

书　　名/ 深入理解Linux内核（第三版）
书　　号/ ISBN 978-7-5083-5394-4
责任编辑/ 夏平，白立军
封面设计/ Edie Freedman，张健
出版发行/ 中国电力出版社（www.infopower.com.cn）
地　　址/ 北京三里河路6号（邮政编码100044）
经　　销/ 全国新华书店
印　　刷/ 北京市丰源印刷厂
开　　本/ 787毫米×1092毫米　16开本　57印张　1299千字
版　　次/ 2007年9月第一版　2007年9月第一次印刷
印　　数/ 0001-4000册
定　　价/ 98.00元（册）

敬告读者

本书封面贴有防伪标签，加热后中心图案消失

本书如有印装质量问题，我社发行部负责退换

版权所有 翻印必究

O'Reilly Media, Inc. 介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc. 授权中国电力出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的 *The Whole Internet User's Guide & Catalog*（被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商 —— 每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体 —— 他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以O'Reilly Media, Inc. 知道市场上真正需要什么图书。

译者序

当着手翻译第三版时，我不由得回想起开始接触Linux的那段日子。

几年前，当我们拿到Linux内核代码开始研究时，可以说茫然无措。其规模之大，叫“履压三百余里，隔离天日”似乎不为过；其关系错综复杂，叫“廊腰缦回，檐牙高啄，各抱地势，勾心斗角”也非言过其实。阿房宫在规模和结构上给人的震撼可能与Linux有异曲同工之妙。“楚人一炬，可怜焦土”，可能正是因为它的结构和规模，阿房宫在中国两千多年盛极的封建历史中终于没有再现，只能叫后人扼腕叹息；但是，Linux却实实在在地矗立在我们面前，当我们徘徊在这宏伟宫殿之前时，或许，我们也需要火炬——不是用来毁灭，而是为了照亮勇者脚下的征途。

Linus Torvalds在我们面前展现的Linux魔法卷轴，让我们的视野进入一个自由而开放的新世界。自由意味着自我价值的实现，开放代表着团结协作的理想，这对于从没把握过核心操作系统的中国人来说，无疑燃起了心中的梦想。于是，许多人毫不犹豫地走进来了，希望深入到那散发自由光彩、由众人团结协力搭造起的殿堂。但是很快，不少人退缩了。面对这样一个汪洋大海，有的人迷惑了，出海的航道在哪里？有的人倒下了，漫漫征途何时是尽头？我常常想，如果那时他们手中就有这本书的话……

Daniel P.Bovet和Marco Cesati携手为我们打造了这本鸿篇巨著，自此我们有了火把，有了航海图，于是我们就有了彼岸，有了航道，也有了补给码头。不是吗？中断虽繁，但第四、六两章切中肯綮的剖析，肯定能让你神清气爽；内存管理虽难，但多达三章细致入微的说理一定会让你茅塞顿开。内容的组织更是别具匠心，每章开始部分一般性原理的描述打破了知识的局限，将每个部分的全景展现在你面前。而针对每个知识点落到实处的独到分析，又会使你沉迷于知识的融会贯通之中。第三版对Linux 2.6的全面描述会使你为2.4与2.6之间的沟壑而感叹，但请放心，你曾从Linux旧版本中获取的点滴依然是你前进的基石。总之，你面对的不再是赤裸裸的代码，而是真正能雅俗共赏的艺术。

对整个Linux社区来说，这绝不是微末的贡献而已，连Andre Morton都已经指出：“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，虽然现在这一拨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”。而这本书的目的无疑是为了弥合这个断层。按照这本书指明的道路，我们可以躲过暗礁，绕过险滩，穿过逆流，勇往直前。这也是为什么这本书总在Linux书籍排行榜中稳居前列的原因之一。

不过，除非行动，否则地图再好也不会让人向自己的目标迈进半步。所以，在读书的同时，你还一定要亲身实践：理解内核某部分的捷径就是对它做些修改，这样你才能越过代码本身看到内核的深层机理。

Linux是一个全新的世界，世界意味着博大精深，而新或许代表对旧的割舍和扬弃，加在一起，就是要我们在割舍和扬弃的同时还要积累知识到博大精深的地步，这容易做到吗？是的，这不容易做到。Gerald M. Weinberg在《Becoming a Technical Leader: An Organic Problem-Solving Approach》一书中将成长总结为高原－低谷模式：“成长是跳跃式的，要经过量的积累，在积累的过程中，往往要伴随着扬弃，所以常常会跌入低谷”。面对Linux这个需要长期孜孜以求的学习对象，无疑这种震荡会加重我们的疑虑，降低我们的信心，消磨我们的意志，使我们轻易地认为达到了自己的成长上限。

根据我们的经验，这需要系统的思考来改变心智模式，最好有一个学习型组织来提供帮助：团队是学习的最佳单位。（可以参看彼得·圣吉的《第五项修炼》，这本书值得有心改变自己并进而改善周围世界的人一读再读。）所以，我们希望结合这本《深入理解Linux内核》创造这样的一个氛围，一种环境。为此我们在www.KernelTravel.net建立了中文网站“内核之旅”，不但有一些有价值的资料，而且我们会把这些资料按照学习路径组织起来，让它们真正伴随内核学习者前进。

翻译组陈莉君、张琼声、张宏伟、黄亭宇、夏守姬、周冲为本书的翻译竭尽全力，但我们所做毕竟有限，正是你的加入，才会让我们大家共同成长。

阅读本书需要一份耐心，更需要一份执着。当你闯过一道道难关阅读到本书的最后一章时，会有“蓦然回首，那人却在灯火阑珊处”的感觉！

感谢电力出版社给了我们翻译这样一本好书的机会，感谢编辑为本书出版所做的细致入微的工作。

本书的第一、三～五、七、九～十一、十五章由张琼声翻译，第十六～二十章由张宏伟翻译，黄亭宇、夏守姬、周冲为第二版和第三版之间的差异做了大量校对工作，同时还参与部分翻译工作。全书由陈莉君审校并统稿。

译 者

2006年9月

目录

前言 1

第一章 绪论 7

Linux 与其他类 Unix 内核的比较 8

硬件的依赖性 12

Linux 版本 13

操作系统基本概念 14

Unix 文件系统概述 18

Unix 内核概述 25

第二章 内存寻址 40

内存地址 40

硬件中的分段 41

Linux 中的分段 46

硬件中的分页 50

Linux 中的分页 62

第三章 进程 84

进程、轻量级进程和线程 84
进程描述符 85
进程切换 107
创建进程 118
撤消进程 130

第四章 中断和异常 135

中断信号的作用 136
中断和异常 137
中断和异常处理程序的嵌套执行 146
初始化中断描述符表 148
异常处理 151
中断处理 154
软中断及 tasklet 174
工作队列 182
从中断和异常返回 186

第五章 内核同步 192

内核如何为不同的请求提供服务 192
同步原语 197
对内核数据结构的同步访问 219
避免竞争条件的实例 224

第六章 定时测量 228

时钟和定时器电路 229
Linux 计时体系结构 233
更新时间和日期 240
更新系统统计数 241

软定时器和延迟函数 244
与定时测量相关的系统调用 252

第七章 进程调度 258
调度策略 258
调度算法 261
调度程序所使用的数据结构 266
调度程序所使用的函数 270
多处理器系统中运行队列的平衡 284
与调度相关的系统调用 289

第八章 内存管理 294
页框管理 294
内存区管理 323
非连续内存区管理 343

第九章 进程地址空间 351
进程的地址空间 352
内存描述符 354
线性区 358
缺页异常处理程序 376
创建和删除进程的地址空间 391
堆的管理 394

第十章 系统调用 397
POSIX API 和系统调用 397
系统调用处理程序及服务例程 398
进入和退出系统调用 400
参数传递 408
内核封装例程 416

第十一章 信号 418

信号的作用 418
产生信号 431
传递信号 437
与信号处理相关的系统调用 448

第十二章 虚拟文件系统 454

虚拟文件系统（VFS）的作用 454
VFS 的数据结构 461
文件系统类型 480
文件系统处理 483
路径名查找 495
VFS 系统调用的实现 505
文件加锁 509

第十三章 I/O 体系结构和设备驱动程序 518

I/O 体系结构 518
设备驱动程序模型 525
设备文件 535
设备驱动程序 539
字符设备驱动程序 550

第十四章 块设备驱动程序 557

块设备的处理 557
通用块层 562
I/O 调度程序 568
块设备驱动程序 582
打开块设备文件 591

第十五章 页高速缓存 595

页高速缓存 596
把块存放在页高速缓存中 607
把脏页写入磁盘 618
sync()、fsync()和fdatasync()系统调用 625

第十六章 访问文件 627

读写文件 628
内存映射 652
直接I/O传送 662
异步I/O 665

第十七章 回收页框 670

页框回收算法 670
反向映射 674
PFRA实现 682
交换 704

第十八章 Ext2和Ext3文件系统 729

Ext2的一般特征 729
Ext2磁盘数据结构 732
Ext2的内存数据结构 741
创建Ext2文件系统 744
Ext2的方法 746
管理Ext2磁盘空间 749
Ext3文件系统 757

第十九章 进程通信 766

管道 767
FIFO 778

System V IPC 780
POSIX 消息队列 796

第二十章 程序的执行 799
可执行文件 800
可执行格式 814
执行域 817
exec 函数 818

附录一 系统启动 825

附录二 模块 832

参考文献 842

源代码索引 847



前言

在1997年春季的那一学期，我们讲授了基于Linux 2.0操作系统这门课程。其主导思想是鼓励学生阅读源代码。为了达到这一目的，我们按小组分配项目，这些项目对内核进行修改并对所修改的版本进行测试。对于诸如任务切换和任务调度这样一些Linux的主要特点，我们也为学生写下了课程笔记。

除了这些工作，还有来自O'Reilly编辑Andy Oram的很多支持，这就促成了《深入理解Linux内核》这本书的第一版，那时是2000年底，该版涵盖了Linux 2.2以及对Linux 2.4的一些展望。这本书的成功鼓励我们继续沿这一思路走下去，在2002年底，我们完成了涵盖Linux 2.4的第二版。现在你看到的第三版则涵盖了Linux 2.6。

与以往所经历的一样，我们这次又阅读了数千行的代码，并努力搞清其含义。在做了所有这些工作以后，可以说我们的努力是完全值得的。我们学到很多你无法从书本中找到的东西，因此我们希望自己已经成功地在后面的内容中涵盖了这些信息。

本书的读者对象

如果你对Linux如何工作、其性能又为什么会如此之高怀有强烈的好奇心，你将会从这里找到答案。阅读本书之后，你会通过上千行代码找到自己的方式来区别重要数据结构和次要数据结构的不同，简而言之，你将成为一名真正的Linux高手。

可以把我们的工作看作是畅游Linux内核的向导：我们讨论了在内核中使用的很多重要的数据结构、算法和编程技巧。在很多例子中，我们逐行讨论了有关代码片段。当然，你手头应当备有Linux源代码，你还应当乐于花一些功夫去解读那些为简洁起见而未完整描述的函数。

另一方面，如果你想更多地了解现代操作系统中的主要设计问题，那么本书将提供颇有价值的见解。本书不是专门针对系统管理员或编程人员的，而是主要针对那些想探究机器内部到底是如何工作的人们的！与任何好向导一样，我们试图透过现象看其本质。我们还提供了背景材料，例如主要特点的历史及使用它们的理由。

材料的组织

开始写这本书时，我们面临重大的抉择：是应该涉及特定的硬件平台，还是跳过与硬件相关的细节而集中于纯粹与硬件无关的内核部分？

有关Linux内核内幕的其他书选择后一种方式；因为下述理由，我们决定采用前一种方式：

- 高效率的内核充分利用硬件可利用的特点，诸如寻址技术、高速缓存（cache）、处理器异常（exception）、专用指令、处理器控制寄存器等等。如果我们想使你相信，内核在执行一个特殊的任务时确实工作得相当好，那我们必须首先告诉你内核工作在一个什么样的硬件平台上。
- 即使Unix内核大部分源代码是独立于处理器的，并且用C语言编写，但也有少数重要的部分是用汇编语言编写的。因此，为了充分理解内核，就需要学习一些与硬件打交道的汇编语言片段。

当涉及硬件特征时，我们的策略非常简单：对全部由硬件驱动的特征给予简单描述，而对需要软件支持的特征给予详细描述。事实上，我们感兴趣的是内核的设计而不是计算机的体系结构。

我们下一步就是选择所描述的计算机系统。尽管Linux目前已运行在很多种类的个人计算机（PC）和工作站上，但我们决定把主要精力放在非常流行且便宜的IBM PC兼容机上，其中微处理器是Intel 80x86及PC中所支持的一些芯片。在以后的章节中，术语“Intel 80x86微处理器”将表示Intel 80386、80486、Pentium、Pentium Pro、Pentium II、Pentium III、Pentium 4微处理器或兼容模型。在少数情况下，对于特殊的模型会给出明确的说明。

在研究Linux各组件时，我们还必须对所遵循的顺序做出选择。我们尝试的是一种自底向上的方式：从硬件相关的主题开始，以完全与硬件无关的主题结束。事实上，在本书的初始部分我们将多次引用Intel 80x86微处理器，而其他部分相对来说与硬件无关。不过，第十三章和第十四章是一种例外。实际上，遵循自底向上的方法并不像看起来那样简单，这是因为存储器管理、进程管理和文件系统这几部分相互渗透；少数向前引用（即引用还待解释的主题）是不可避免的。

每章以所涵盖内容的理论概述开始，然后按自底向上的方式组织材料。我们以描述每章内容所需要的数据结构开始，然后，我们通常从描述最低级功能移到描述较高级功能，最后说明用户应用程序所发出的系统调用是如何得到支持的。

描述级别

支持各种体系结构的Linux源代码包含在14000多个C语言和汇编语言的文件中，这些文件存放在大约1000个子目录中。源代码大约由六百万行代码组成，占230MB以上的磁盘空间。当然，这本书只能涵盖源代码非常少的一部分。考虑一下你所读的书的全部源代码只占不到3MB的磁盘空间，就能想像出Linux源代码有多么庞大了。因此，即使不对源代码进行解释，只列出所有的代码，75本书也写不完!

因此，我们必须对要阐述的内容做出选择，我们的决策大致情况如下：

- 我们相当全面地描述了进程管理和内存管理。
- 我们涵盖了虚拟文件系统以及Ext2和Ext3文件系统，不过，很多功能仅仅是提及而已，并没有对其代码进行详尽描述；我们不讨论Linux所支持的其他文件系统。
- 我们描述了占内核50%左右的设备驱动程序；但仅涉及有关的内核接口，而并不试图分析每个具体的驱动程序。

本书描述的是Linux内核2.6.11的正式版，可以从*http://www.kernel.org*站点下载。

注意，很多GNU/Linux发布版都对正式内核进行了修改，以实现新的特点或提高其效率。在少数情况下，由你喜爱的发布版所提供的源代码可能与本书所描述的源代码有很大的不同。

在很多实例中，我们展示了以易读但低效的方式重写的原始代码的片段。这些代码出现在关键时间点上，在这些点上，程序片段是用手工优化的C语言和汇编代码混合在一起编写的。再次声明，我们的目的是为研究Linux原始代码的人提供一些帮助。

在讨论内核代码时，我们常常同时描述Unix程序员熟悉的很多基础知识（共享内存和映射内存、信号、管道、符号链等等），也许他们听说过这些内容，但可能还想进一步了解。

本书概述

为了对全书有一个大体了解，在第一章“绪论”中对Unix内核内部结构给出了一般性描述，并说明Linux如何与其他著名的Unix系统展开竞争。

任何Unix内核的核心都是内存管理。第二章“内存寻址”说明Intel 80x86处理器包含有对内存数据进行寻址的特殊电路，并解释Linux如何充分利用它们。

进程是Linux所提供的一种基本抽象，这在第三章“进程”中进行介绍。在这一章，我们也解释了每个进程如何在非特权的用户态下运行，又如何在有特权的内核态下运行。用户态与内核态之间的转换只能通过已建立的所谓中断和异常处理硬件机制实现，这些内容将在第四章“中断和异常”中介绍。

在很多情况下，内核必须处理来自不同设备和处理器的突发性中断。因此，就需要同步机制，以便所有这些请求能由内核以交错方式去处理：这些将在第五章“内核同步”中进行讨论，其中既涉及单处理器系统，也涉及多处理器系统。

定时中断使Linux能够处理已经经历的时间，是一种重要的中断类型；更详细的内容将在第六章“定时测量”中介绍。

第七章“进程调度”说明Linux如何轮流执行系统中的每个活动进程，以便所有的进程都能顺利地执行完。

接下来我们再一次集中讨论内存。第八章“内存管理”描述用来处理系统中最宝贵的资源——可用内存（当然除了处理器）——所需要的复杂技术。这种资源必须同时满足Linux内核和用户应用程序的需要。第九章“进程地址空间”讲述内核如何处理应用程序对内存发出的“贪婪（greedy）”请求。

第十章“系统调用”说明在用户态下运行的进程如何对内核发出请求;而第十一章“信号”描述进程如何给其他进程发送同步信号。现在我们准备进入另一个实质性的主题，即Linux如何实现文件系统。很多章节涉及到这个主题。第十二章“虚拟文件系统”介绍了支持很多种不同文件系统的通用层。某些Linux文件比较特殊，这是因为它们能提供到达硬件设备的陷阱门；第十三章“I/O体系结构和设备驱动程序”以及第十四章“块设备驱动程序”进一步考察了这些特殊的文件和相应的硬件设备驱动程序。

另一个值得考虑的问题是磁盘访问时间，第十五章“页高速缓存”说明灵活地利用RAM可以减少磁盘的访问时间，因而极大地提高系统的性能。在前几章内容的基础上，我们将在第十六章“访问文件”中讨论用户应用程序如何访问常规文件。在第十七章“回收页框”完成对Linux内存管理的讨论，并说明Linux确保总是有足够的内存所使用的技术。第十八章“Ext2和Ext3文件系统”是讨论文件系统的最后一章，阐述了Linux最常用的文件系统，即Ext2及最新改进的Ext3。

最后两章结束我们对Linux内核的详细游览：第十九章“进程通信”介绍通信机制而不是用户态进程使用的信号；第二十章“程序的执行”说明用户应用程序是如何开始执行的。

最后，但必不可少的，就是附录：附录一“系统启动”大致描述了Linux如何启动；而附录二“模块”描述怎样动态地重新配置正在运行的内核，即按需增加或删除有关功能。源代码索引（Source Code Index）包含了在本书中引用的所有Linux符号；你将在这里找到定义每个符号的Linux文件名，以及对这个符号进行解释的正文所在的页码。我们认为你会发现它非常方便实用。

背景知识

除了一些C语言编程技巧和汇编语言的知识外，理解这些内容不需要任何先决条件。

排版约定

下面是本书在英文字体上的两个约定：

等宽字体（Constant Width）
　用来说明代码文件的内容或命令输出的内容，也表示出现在代码中的源代码关键字。

斜体（*Italic*）
　用来说明文件名、目录名、程序名、命令名、命令行选项和URL。

如何与我们联系

请把有关本书的评论和问题告知出版社：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市海淀区知春路49号希格玛公寓B座809室（100080）
奥莱理软件（北京）有限公司

本书的网页上列出了勘误表、示例和任何额外的信息。可登录以下网址查询：

http://www.oreilly.com/catalog/understandlk/
http://www.oreilly.com.cn/book.php?bn=978-7-5083-5394-4

如果想要发表关于本书的评论或咨询有关技术问题，请发邮件至：

bookquestions@oreilly.com
info@mail.oreilly.com.cn

关于图书、会议、资源中心和O'Reilly网络的更多信息，请查看我们的站点：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

如果没有罗马大学Tor Vergata分校工程学院很多学生的尽力帮助，这本书不可能完成，他们不但上了这门课，还试着解读了Linux内核的讲稿。他们不懈的努力紧紧抓住了源代码的真正含义，使得我们对讲稿不断改进，并改正了很多错误。

Andy Oram是O'Reilly Media的优秀编辑，非常值得信任。他是O'Reilly第一位对此项目给予信任的人，他花费了很多时间和精力阅读我们的初稿。他提出的很多建议使本书的可读性更强，同时他还写出了不少出色的介绍性段落。

还有一些颇具声望的技术审校，他们非常认真地阅读了本书的内容。第一版的审校人为（名字按字母顺序排列）：Alan Cox、Michael Kerrisk、Paul Kinzelman、Raph Levien和Rik van Riel。

第二版的审校人为Erez Zadok，Jerry Cooperstein，John Goerzen，Michael Kerrisk，Paul Kinzelman，Rik van Riel和Walt Smith。

第三版的审校人为Charles P. Wright，Clemens Buchacher，Erez Zadok，Raphael Finkel，Rik van Riel和Robert P. J. Day。他们的建议，加之世界各地很多读者的参与，帮助我们去掉了很多错误和不准确的地方，使得本书更具有说服力。

—Daniel P. Bovet
Marco Cesati
July 2005



第一章

绪论

Linux（注1）是类Unix（Unix-like）操作系统大家族中的一名成员。从20世纪90年代末开始，Linux这位相对较新的成员突然变得非常流行，并且跻身于那些知名的商用Unix操作系统之列，这些Unix系统包括AT&T公司（现在由SCO公司拥有）开发的（System V Release 4）SRV4、加利福尼亚大学伯克利分校发布的4.4 BSD、DEC公司（现在属于HP）的Digital Unix、IBM公司的AIX、HP公司的HP-UX、Sun公司的Solaris，以及Apple公司的Mac OS X。除了Linux之外，还有一些其他的类Unix操作系统也是开放源代码的，如FreeBSD、NetBSD以及OpenBSD.

1991年，Linus Torvalds开发出最初的Linux，它作为一个适用于基于Intel 80386微处理器的IBM PC兼容机的操作系统。现在，Linus依然不遗余力地改进Linux，使它保持与各种硬件平台同步发展，并协调世界各地数百名Linux开发者的开发工作。几年来，开发者已经使Linux可以在其他平台上运行，包括HP的Alpha、Intel的Itanium，AMD的AMD64、Power PC及IBM的zSeries。

Linux最吸引人的一个优点就在于它不是商业操作系统：它的源代码在GNU公共许可证（*General Pwblic License*，*GPL*）（注2）下是开放的，任何人都可以获得源代码并研究它（就像我们在本书中那样研究）；只要你下载源代码（可以下载源代码的官方站点是*http://www.kernel.org/*），或者在Linux光盘上找到源代码，你就可以从用户接口层到与

注1：　Linux ®是Linus Torvaeds注册的商标。

注2：　GNU项目是由自由软件基金会（*http://www.gun.org*）所协调的，其目的是实现一个完整的操作系统，供所有人免费使用。GNU C编译器对Linux项目的成功必不可少。

硬件密切相关的操作系统核心层，对这个最成功而又最现代的操作系统进行由表及里的研究。事实上本书假定你有源代码，而且你能把我们介绍的方法应用到自己的探索中。

从技术角度来说，Linux是一个真正的Unix内核，但它不是一个完全的Unix操作系统，这是因为它不包含全部的Unix应用程序，诸如文件系统实用程序、窗口系统及图形化桌面、系统管理员命令、文本编辑程序、编译程序等等。不过，因为以上大部分应用程序都可在GNU许可证下免费获得，因此，可以把它们安装在任何一个基于Linux内核的系统中。

因为Linux内核需要大量运行其他软件才能为用户提供一个有用的环境，因此很多Linux用户更喜欢依赖从CD-ROM获得的商业发布版，以得到包含在标准Unix系统中的应用层代码。另外，这些支持应用的源代码也可以从几个不同的网站获得，例如*http://www.kernel.org*。一些发布版将Linux源代码安装在*/usr/src/linux*目录下。在本书的其余部分，所有文件的目录都暗指这一目录。

Linux与其他类Unix内核的比较

市场上各种类Unix系统在很多重要的方面有所不同，其中有些系统已经有很长的历史，并且显得有点过时。所有商业版本都是SVR4或4.4BSD的变体，并且都趋向于遵循某些通用标准，诸如IEEE的POSIX （Portable Operating Systems based on Unix 基于Unix的可移植操作系统)和X/Open的CAE （Common Applications Environment，公共应用环境)。

现有标准仅仅指定了应用程序编程接口（application programming interfale，API）——也就是说，指定了用户程序应当运行的一个已定义好的环境。因此，这些标准并没有对内核的内部设计施加任何限制（注3)。

为了定义一个通用用户接口，类Unix内核通常采用相同的设计思想和特征。在这一点上，Linux和其他的类Unix操作系统是一样的。因此，阅读本书并研读Linux内核也有助于你理解其他Unix变体。

Linux内核2.6版的目标是遵循IEEE POSIX标准。这意味着在Linux系统下，很容易编译和运行目前现有的大多数Unix程序，只需少许或根本无需为源代码打补丁。此外，Linux包括了现代Unix操作系统的全部特点，诸如虚拟存储、虚拟文件系统、轻量级进程、Unix信号量、SVR4进程间通信、支持对称多处理器（Symmetric Multiprocessor，SMP）系统等。

注3： 实际上，一些非Unix操作系统（诸如Windows NT及其后续系统）也遵循POSIX标准。

Linus Torvalds 在写第一个内核的时候，参考了 Unix 内幕方面一些经典的书，比如 Maurice Bach 的《The Design of the Unix Operating System》(Prentice Hall, 1986)。实际上，Linux 始终对 Bach 的书（即 SVR4）中所描述的 Unix 基准有些偏爱。但是，Linux 没有拘泥于任何一个特定的变体，相反，它尝试采纳了几种不同 Unix 内核中最好的特征和设计选择。

Linux 与一些著名的商用 Unix 内核到底如何竞争，下面给予描述：

单块结构的内核（Monolithic kernel）

它是一个庞大、复杂的自我完善（do-it-yourself）程序，由几个逻辑上独立的成分构成。在这一点上，它是相当传统的，大多数商用 Unix 变体也是单块结构。（一个显著的例外是 Apple 的 Mac Os X 和 GNU 的 Hurd 操作系统，它们都是从卡耐基-梅隆大学的 Mach 演变而来的，都遵循微内核的方法 。）

编译并静态连接的传统 Unix 内核

大部分现代操作系统内核可以动态地装载和卸载部分内核代码（典型的例子如设备驱动程序），通常把这部分代码称做模块（module）。Linux 对模块的支持是很好的，因为它能自动按需装载或卸载模块。在主要的商用 Unix 变体中，只有 SVR4.2 和 Solaris 内核有类似的特点。

内核线程

一些 Unix 内核，如 Solaris 和 SVR4.2/MP，被组织成一组内核线程（kernel thread）。内核线程是一个能被独立调度的执行环境（context）；也许它与用户程序有关，也许仅仅执行一些内核函数。线程之间的上下文切换比普通进程之间的上下文切换花费的代价要少得多，因为前者通常在同一个地址空间执行。Linux 以一种十分有限的方式使用内核线程来周期性地执行几个内核函数；但是，它们并不代表基本的执行上下文的抽象（这就是下面要讨论的议题）。

多线程应用程序支持

大多数现代操作系统在某种程度上都支持多线程应用程序，也就是说，这些用户程序是根据很多相对独立的执行流来设计的，而这些执行流之间共享应用程序的大部分数据结构。一个多线程用户程序由很多轻量级进程（lightweight process，LWP）组成，这些进程可能对共同的地址空间、共同的物理内存页、共同的打开文件等等进行操作。Linux 定义了自己的轻量级进程版本，这与 SVR4、Solaris 等其他系统上所使用的类型有所不同。当 LWP 的所有商用 Unix 变体都基于内核线程时，Linux 却把轻量级进程当作基本的执行上下文，通过非标准的 `clone()` 系统调用来处理它们。

抢占式（preemptive）内核

当采用“可抢占的内核”选项来编译内核时，Linux2.6可以随意交错执行处于特权模式的执行流。除了Linux 2.6，还有其他一些传统的、通用的Unix系统（如Solaris和Mach3.0）是完全的抢占式内核。SVR4.2/MP通过引入一些固定抢占点（fixed preemption point）的方法获得有限的抢占能力。

多处理器支持

几种Unix内核变体都利用了多处理器系统。Linux 2.6支持不同存储模式的对称多处理（SMP），包括NUMA: 系统不仅可以使用多处理器，而且每个处理器可以毫无区别地处理任何一个任务。尽管通过一个单独的“大内核锁”使得内核中的少数代码依然串行执行，但公平地说，Linux 2.6以几乎最优化的方式使用SMP。

文件系统

Linux标准文件系统呈现出多种风格。如果你没有特殊需要，就可以使用普通的Ext2文件系统。如果你想避免系统崩溃后冗长的文件系统检查，就可以切换到Ext3。如果你不得不处理很多小文件，ReiserFS文件系统可能就是最好的选择。除了Ext3和ReiserFS，还可以在Linux中使用另外几个日志文件系统；这些文件系统包括IBM AIX的日志文件系统（Journaling File System，JFS）和SGI公司IRIX系统上的XFS文件系统。有了强大的面向对象虚拟文件系统技术（为Solaris和SVR4所采用），把外部文件系统移植到Linux比移植到其他内核相对要容易。

STREAMS

尽管现在大部分的Unix内核内包含了SRV4引入的STREAMS I/O子系统，并且已变成编写设备驱动程序、终端驱动程序及网络协议的首选接口，但是Linux并没有与此类似的子系统。

对Linux的评价充分说明，与商业化的操作系统相比，Linux已经具备足够的竞争力。而且，Linux一些独具特色的特点使其成为一种趣味盎然的操作系统。商业化的Unix内核为了赢得更大的市场份额通常也引入了新特征，但这些特征本是可有可无，其稳定性和效率都值得商榷。事实上，现代Unix内核有向更臃肿变化的倾向，而Linux以及其他开放源代码的操作系统不受市场因素的制约，因此可以根据设计者的想法（主要是Linus Torvalds的想法）自由地演进。尤其是，与商用竞争对手相比，Linux有如下优势：

Linux是免费的。除硬件之外，你无需任何花费就能安装一套完整的Linux系统。

Linux的所有成分都可以充分地定制。通过内核编译选项，你可以选择自己真正需要的

特征来定制内核。而且有了通用公共许可证（GPL），你就可以自由地阅读、修改内核和所有系统程序的源代码（注4）。

Linux可以运行在低档、便宜的硬件平台上。你可以用一个4MB内存的旧Intel 80386系统构建网络服务器。

Linux是强大的。由于充分挖掘了硬件部分的特点，使得Linux系统速度非常快。Linux的主要目标是效率，所以，商用系统的许多设计选择由于有降低性能的隐患而被Linus舍弃，如STREAMSI/O子系统。

Linux的开发者都是非常出色的程序员。Linux系统非常稳定，有非常低的故障率和非常少系统维护时间。

Linux内核非常小，而且紧凑。我们甚至可以把一个内核映像和一些系统程序放在一张1.4MB的软盘上！据我们所知，没有一个商用Unix变体能从一张软盘上启动。

Linux与很多通用操作系统高度兼容。Linux可以让你直接安装以下文件系统的所有版本：MS-DOS和MS Windows、SVR4、OS/2、Mac OS X、Solaris、SunOS、NEXTSTEP，还有很多BSD变体等等。另外，Linux也能对很多网络层进行操作，这些网络层如以太网[如：快速以太网和高速（Gbit/s及10Gbit/s）以太网]、光纤分布式数据接口（Fiber Distributed Data Interface，FDDI）、高性能并行接口（High Performance Parallel Interface，HIPPI）、IEEE 802.11（无线局域网）和IEEE802.15（蓝牙）。。通过使用适当的库函数，Linux系统甚至能直接运行为其他操作系统所编写的程序。例如，Linux能执行为以下操作系统所编写的应用程序：MS-DOS、MS Windows、SVR3及SV R4、4.4BSD、SCO Unix、Xenix，以及其他在Intel 80x86平台上运行的操作系统。

Linux有很好的技术支持。不管你信不信，Linux比任何有版权的操作系统更容易获得补丁和更新！如果你把遇到的难题发给一些新闻组或邮件列表，经常在几个小时内就会得到回应。此外，当新的硬件产品投放市场以后，其Linux驱动程序通常在几周内就可得到。与此相反，硬件厂商仅仅给少数商业操作系统发布设备驱动程序，通常只有微软一家。因此，所有商用Unix变体只能运行在有限的硬件上。

因为有了数千万台安装Linux的基础，那些习惯了其他操作系统某些标准特征的用户开始期望Linux也具有相同的特征。在这种情况下，对Linux开发者的需求也在不断增加。值得庆幸的是，在Linus的密切指导下，Linux始终在不断发展以满足如此众多的需求。

注4：许多商业公司已经开始在Linux下支持他们的产品，但其大部产品并不是在GNU许可证下发布的，因此，可能不允许你阅读或修改他们的源代码。

硬件的依赖性

Linux试图在硬件无关的源代码与硬件相关的源代码之间保持清晰的界限。为了做到这点，在*arch*和*include*目录下包含了23个子目录，以对应Linux所支持的不同硬件平台。这些平台的标准名字如下：

alpha

HP的Alpha工作站，最早属于Digital公司，后来属于Compag公司，现在不再生产）。

arm，*arm26*

基于ARM处理器的计算机（如PDA）和嵌入式设备。

cris

Axis在它的瘦服务器中使用的“代码精简指令集(Code Reduced Instruction Set)”CPU，用在诸如Web摄像机或开发主板中。

frv

基于Fujitsu FR-V系列微处理器的嵌入式系统。

h8300

Hitachi h8/300和h8S的8位和16位RISC微处理器。

i386

基于80x86微处理器的IBM兼容个人计算机。

ia64

基于64位Itanium微处理器的工作站。

m32r

基于Renesas M32R系列微处理器的计算机。

m68k，*m68knommu*

基于Motorola MC680x0微处理器的个人计算机。

mips

基于MIPS微处理器的工作站，如Silicon Graphics公司销售的那些工作站。

parisc

基于HP公司HP 9000 PA-RISC微处理器的工作站。

ppc,ppc64

基于Motorola-IBM PowerPC32位和64位微处理器的工作站。

s390

IBM ESA/390 及 zSeries 大型机。

sh, sh64

基于 Hitachi 和 STMicroelectronics 联合开发的 SuperH 微处理器的嵌入式系统。

sparc, sparc64

基于 Sun 公司 SPARC 和 64 位 Ultra SPARC 微处理器的工作站。

um

用户态的 Linux —— 一个允许开发者在用户态下运行内核的虚拟平台。

v850

集成了基于 Harvard 体系结构的 32 位 RISC 核心的 NEC V850 微控制器。

x86_64

基于AMD的64位微处理器的工作站，如Athlon和Opteron，以及基于Intel的 ia32e/EM64T64 位微处理器的工作站。

Linux 版本

一直到2.5版本的内核，Linux都通过简单的编号来区别内核的稳定版和开发版。每个版本号用三个数字描述，由圆点分隔。前两个数字用来表示版本号，第三个数字表示发布号。第一位版本号2从1996年开始就没有变过。第二位版本号表示内核的类型：如果为偶数，表示稳定的内核；否则，表示开发中的内核。

正如内核版本名字所表示的，稳定版本的内核由Linux的发布者和内核黑客彻底检查过，一个稳定版的新发布主要用来纠正用户所报告的错误或者增加新的驱动程序。另一方面，开发版的不同版本之间可能有非常明显的差异。内核开发者可以自由地采用不同方案进行实验，但这些实验可能导致内核有很大变化。用开发版运行应用程序的用户，当把内核升级到新版时，也许会遇到一些不那么令人愉快的意外。

然而，在Linux内核2.6版的开发过程中，内核版本的编号方式发生了很大的变化。主要变化在于第二个数字已经不再用于表示一个内核是稳定版本还是正在开发的版本。因此，现在内核开发者都在当前的2.6版本中对内核进行大幅改进。只有在内核开发者必须对内核的重大修改进行测试时，才会采用一个新的内核分支2.7。这种2.7的分支要么产生一个新的内核版本，要么干脆丢弃所修改的部分而回退到2.6版。

Linux这种新的开发模式意味着两种内核具有相同的版本号，但却有不同的发布号，如2.6.10和2.6.11内核就可能在核心部件和基本算法上有很大的差别。这样一来，具有新

发布号的内核可能潜藏着不稳定性和各种错误。为了解决这个问题，内核开发者可能发布带有补丁程序的内核版本，并且用第四位数字表示带有不同补丁的内核版本。例如，在写本段文字时，最新的稳定内核版本是2.6.11.12。

必须强调的是本书描述的是Linux2.6.11版的内核。

操作系统基本概念

任何计算机系统都包含一个名为操作系统的基本程序集合。在这个集合里，最重要的程序称为内核（*kernel*）。当操作系统启动时，内核被装入到RAM中，内核中包含了系统运行所必不可少的很多核心过程（*procedure*）。其他程序是一些不太重要的实用程序，尽管这些程序为用户提供了与计算机进行广泛交流的经验(以及用户买计算机要做的所有工作)，但系统根本的样子和能力还是由内核决定。内核也为系统中所有事情提供了主要功能，并决定高层软件的很多特性。因此，我们将经常使用术语“操作系统”作为“内核”的同义词。

操作系统必须完成两个主要目标：

- 与硬件部分交互，为包含在硬件平台上的所有低层可编程部件提供服务。
- 为运行在计算机系统上的应用程序（即所谓用户程序）提供执行环境。

一些操作系统允许所有的用户程序都直接与硬件部分进行交互（典型的例子是MS-DOS）。与此相反，类Unix操作系统把与计算机物理组织相关的所有低层细节都对用户运行的程序隐藏起来。当程序想使用硬件资源时，必须向操作系统发出一个请求。内核对这个请求进行评估，如果允许使用这个资源，那么，内核代表应用程序与相关的硬件部分进行交互。

为了实施这种机制，现代操作系统依靠特殊的硬件特性来禁止用户程序直接与低层硬件部分进行交互，或者禁止直接访问任意的物理地址。特别是，硬件为CPU引入了至少两种不同的执行模式：用户程序的非特权模式和内核的特权模式。Unix把它们分别称为用户态（*User Mode*）和内核态（*Kernel Mode*）。

我们将在本章剩余部分介绍一些基本概念，在过去的20多年里，这些概念推动了Unix、Linux和其他操作系统的设计。作为Linux用户，你也许已熟悉了这些概念，但为了说明这些概念对Linux内核的必要性，下面试图对其作更深一步的研究。这些广泛的考虑事实上涉及到全部类Unix系统。希望本书的其他章节能帮助你理解Linux内核内幕。

多用户系统

多用户系统（*multiuser system*）就是一台能并发和独立地执行分别属于两个或多个用户的若干应用程序的计算机。“并发”（*concurrently*）意味着几个应用程序能同时处于活动状态并竞争各种资源，如CPU、内存、硬盘等等。“独立”（*independently*）意味着每个应用程序能执行自己的任务，而无需考虑其他用户的应用程序在干些什么。当然，从一个应用程序切换到另一个会使每个应用程序的速度有所减慢，从而影响用户看到的响应时间。现代操作系统内核提供的许多复杂特性（我们将在本书中考察这些特性）减少了强加在每个程序上的延迟时间，给用户提供了尽可能快的响应时间。

多用户操作系统必须包含以下几个特点：

- 核实用户身份的认证机制。
- 防止有错误的用户程序防碍其他应用程序在系统中运行的保护机制。
- 防止有恶意的用户程序干涉或窥视其他用户的活动的保护机制。
- 限制分配给每个用户的资源数的计账机制。

为了确保能实现这些安全保护机制，操作系统必须利用与CPU特权模式相关的硬件保护机制，否则，用户程序将能直接访问系统电路并克服强加于它的这些限制。Unix是实施系统资源硬件保护的多用户系统。

用户和组

在多用户系统中，每个用户在机器上都有私用空间；典型地，他拥有一定数量的磁盘空间来存储文件、接收私人邮件信息等等。操作系统必须保证用户空间的私有部分仅仅对其拥有者是可见的。特别是必须能保证，没有用户能够开发一个用于侵犯其他用户私有空间的系统应用程序。

所有的用户由一个惟一的数字来标识，这个数字叫用户标识符（*User ID*，*UID*）。通常一个计算机系统只能由有限的人使用。当其中的某个用户开始一个工作会话时，操作系统要求输入一个登录名和口令，如果用户输入的信息无效，则系统拒绝访问。因为口令是不公开的，所以用户的保密性得到了保证。

为了和其他用户有选择地共享资料，每个用户是一个或多个用户组的一名成员，组由唯一的用户组标识符（*user group ID*）标识。每个文件也恰好与一个组相对应。例如，可以设置这样的访问权限，拥有文件的用户具有对文件的读写权限，同组用户仅有只读权限，而系统中的其他用户没有对文件的任何访问权限。

任何类Unix操作系统都有一个特殊的用户，叫做*root*，即超级用户（*superuser*）。系统管理员必须以root的身份登录，以便处理用户账号，完成诸如系统备份、程序升级等维护任务。root用户几乎无所不能，因为操作系统对她不使用通常的保护机制。尤其是，root用户能访问系统中的每一个文件，能干涉每一个正在执行的用户程序的活动。

进程

所有的操作系统都使用一种基本的抽象：进程（*process*）。一个进程可以定义为："程序执行时的一个实例"，或者一个运行程序的"执行上下文"。在传统的操作系统中，一个进程在地址空间（*address space*）中执行一个单独的指令序列。地址空间是允许进程引用的内存地址集合。现代操作系统允许具有多个执行流的进程，也就是说，在相同的地址空间可执行多个指令序列。

多用户系统必须实施一种执行环境，在这种环境里，几个进程能并发活动，并能竞争系统资源（主要是CPU）。允许进程并发活动的系统称为多道程序系统（*multiprogramming*）或多处理系统（*multiprocessing*）（注5）。区分程序和进程是非常重要的：几个进程能并发地执行同一程序，而同一个进程能顺序地执行几个程序。

在单处理器系统上，只有一个进程能占用CPU，因此，在某一时刻只能有一个执行流。一般来说，CPU的个数总是有限的，因而只有少数几个进程能同时执行。操作系统中叫做调度程序（*scheduler*）的部分决定哪个进程能执行。一些操作系统只允许有非抢占式（*nonpreemptable*）进程，这就意味着，只有当进程自愿放弃CPU时，调度程序才被调用。但是，多用户系统中的进程必须是抢占式的（*preemptable*）；操作系统记录下每个进程占有的CPU时间，并周期性地激活调度程序。

Unix是具有抢占式进程的多处理操作系统。即使没有用户登录，没有程序运行，也还是有几个系统进程在监视外围设备。尤其是，有几个进程在监听系统终端等待用户登录。当用户输入一个登录名，监听进程就运行一个程序来验证用户的口令。如果用户身份得到证实，那么监听进程就创建另一个进程来执行shell，此时在shell下可以输入命令。当一个图形化界面被激活时，有一个进程就运行窗口管理器，界面上的每个窗口通常都由一个单独的进程来执行。如果用户创建了一个图形化shell，那么，一个进程运行图形化窗口，而第二个进程运行用户可以输入命令的shell。对每一个用户命令，shell进程都创建执行相应程序的另一个进程。

类Unix操作系统采用进程/内核模式。每个进程都自以为它是系统中唯一的进程，可以独占操作系统所提供的服务。只要进程发出系统调用（即对内核提出请求，参见第10

注5： 一些多处理操作系统不是多用户的，其中一个例子就是微软公司的Windows 98。

章），硬件就会把特权模式由用户态变成内核态，然后进程以非常有限的目的开始一个内核过程的执行。这样，操作系统在进程的执行上下文中起作用，以满足进程的请求。一旦这个请求完全得到满足，内核过程将迫使硬件返回到用户态，然后进程从系统调用的下一条指令继续执行。

内核体系结构

如前所述，大部分Unix内核是单块结构：每一个内核层都被集成到整个内核程序中，并代表当前进程在内核态下运行。相反，微内核（*microkernel*）操作系统只需要内核有一个很小的函数集，通常包括几个同步原语、一个简单的调度程序和进程间通信机制。运行在微内核之上的几个系统进程实现从前操作系统级实现的功能，如内存分配程序、设备驱动程序、系统调用处理程序等等。

尽管关于操作系统的学术研究都是面向微内核的，但这样的操作系统一般比单块内核的效率低，因为操作系统不同层次之间显式的消息传递要花费一定的代价。不过，微内核操作系统比单块内核有一定的理论优势。微内核操作系统迫使系统程序员采用模块化的方法，因为任何操作系统层都是一个相对独立的程序，这种程序必须通过定义明确而清晰的软件接口与其他层交互。此外，已有的微内核操作系统可以很容易地移植到其他的体系结构上，因为所有与硬件相关的部分都被封装进微内核代码中。最后，微内核操作系统比单块内核更加充分地利用了RAM，因为暂且不需要执行的系统进程可以被调出或撤消。

为了达到微内核理论上的很多优点而又不影响性能，Linux内核提供了模块（*module*）。模块是一个目标文件，其代码可以在运行时链接到内核或从内核解除链接。这种目标代码通常由一组函数组成，用来实现文件系统、驱动程序或其他内核上层功能。与微内核操作系统的外层不同，模块不是作为一个特殊的进程执行的。相反，与任何其他静态链接的内核函数一样，它代表当前进程在内核态下执行。

使用模块的主要优点包括：

模块化方法

因为任何模块都可以在运行时被链接或解除链接，因此，系统程序员必须提出良定义的软件接口以访问由模块处理的数据结构。这使得开发新模块变得容易。

平台无关性

即使模块依赖于某些特殊的硬件特点，但它不依赖于某个固定的硬件平台。例如，符合SCSI标准的磁盘驱动程序模块，在IBM兼容PC与HP的Alpha机上都能很好地工作。

节省内存使用

当需要模块功能时，把它链接到正在运行的内核中，否则，将该模块解除链接。这种机制对于小型嵌入式系统是非常有用的。

无性能损失

模块的目标代码一旦被链接到内核，其作用与静态链接的内核的目标代码完全等价。因此，当模块的函数被调用时，无需显式地进行消息传递（注 6）。

Unix 文件系统概述

Unix操作系统的设计集中反映在其文件系统上，文件系统有几个有趣的特点。因为在后面的章节中会反复提到这些特点，所以我们先回顾最重要的几个特点。

文件

Unix 文件是以字节序列组成的信息载体（*container*），内核不解释文件的内容。很多编程的库函数实现了更高级的抽象，例如，由字段构成的记录以及基于关键字编址的记录。然而，这些库中的程序必须依靠内核提供的系统调用。从用户的观点来看，文件被组织在一个树结构的命名空间中，如图 1-1 所示。

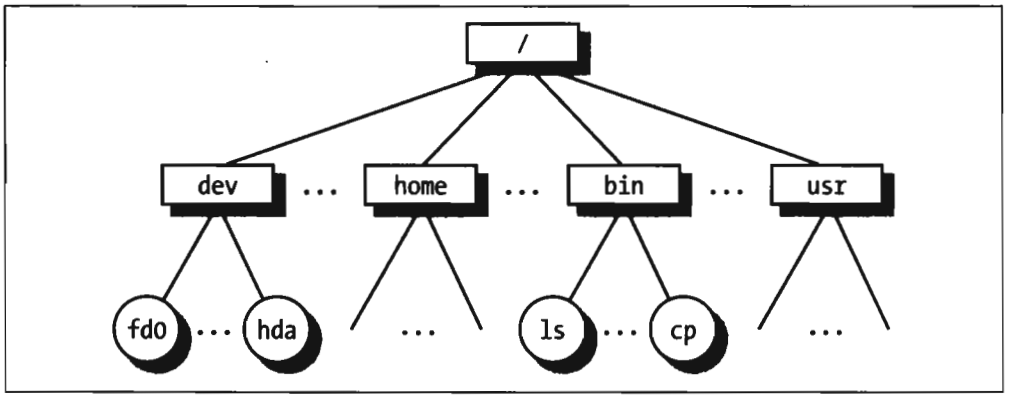


图 1-1：目录树示例

除了叶节点之外，树的所有节点都表示目录名。目录节点包含它下面文件及目录的所有

注 6： 当模块被链接或被解除链接时，性能稍有下降。但是在微内核操作系统中，系统进程的创建和删除也是这样的。

信息。文件或目录名由除“/”和空字符“\0”之外的任意ASCII字符序列组成（注7）。大多数文件系统对文件名的长度都有限制，通常不能超过255个字符。与树的根相对应的目录被称为根目录（*root directory*）。按照惯例，它的名字是“/”。在同一目录中的文件名不能相同，而在不同目录中的文件名可以相同。

Unix的每个进程都有一个当前工作目录（参见本章后面的“进程/内核模式”一节），它属于进程执行上下文（*execution context*），标识出进程所用的当前目录。为了标识一个特定的文件，进程使用路径名（*pcothname*），路径名由斜杠及一列指向文件的目录名交替组成。如果路径名的第一个字符是斜杠，那么这个路径就是所谓的绝对路径，因为它的起点是根目录。否则，如果第一项是目录名或文件名，那么这个路径就是所谓的相对路径，因为它的起点是进程的当前目录。

当标识文件名时，也用符号“.”和“..”。它们分别标识当前工作目录和父目录。如果当前工作目录是根目录，“.”和“..”就是一致的。

硬链接和软链接

包含在目录中的文件名就是一个文件的硬链接（*hard link*），或简称链接（*Link*）。在同一目录或不同的目录中，同一文件可以有几个链接，因此对应几个文件名。

Unix命令：

```
$ ln P1 P2
```

用来创建一个新的硬链接，即为由路径P1标识的文件创建一个路径名为P2的硬链接。

硬链接有两方面的限制：

- 不允许用户给目录创建硬链接。因为这可能把目录树变为环形图，从而就不可能通过名字定位一个文件。
- 只有在同一文件系统中的文件之间才能创建链接。这带来比较大的限制，因为现代Unix系统可能包含了多种文件系统，这些文件系统位于不同的磁盘和/或分区，用户也许无法知道它们之间的物理划分。

为了克服这些限制，引入了软链接（*soft link*）[也称符号链接（*symbolic link*）]。符号链接是短文件，这些文件包含有另一个文件的任意一个路径名。路径名可以指向位于任意一个文件系统的任意文件或目录，甚至可以指向一个不存在的文件。

注7：一些操作系统允许以多种字符表来表示文件名，例如Unicode，基于16位图形字符的扩展编码。

Unix 命令：

```
$ ln -s P1 P2
```

创建一个路径名为P2的新软链接，P2指向路径名P1。当这个命令执行时，文件系统抽出P2的目录部分，并在那个目录下创建一个名为P2的符号链接类型的新项。这个新文件包含路径名P1。这样，任何对P2的引用都可以被自动转换成指向P1的一个引用。

文件类型

Unix 文件可以是下列类型之一：

- 普通文件（regular file）
- 目录
- 符号链接
- 面向块的设备文件（block-oriented device file）
- 面向字符的设备文件（character-oriented device file）
- 管道（pipe）和命名管道（named pipe）（也叫FIFO）
- 套接字（socket）

前三种文件类型是所有 Unix 文件系统的基本类型。其实现将在第十八章详细讨论。

设备文件与I/O设备以及集成到内核中的设备驱动程序相关。例如，当程序访问设备文件时，它直接访问与那个文件相关的I/O设备（参见第十三章）。

管道和套接字是用于进程间通信的特殊文件（参见本章后面的“同步和临界区”一节以及第十九章）。

文件描述符与索引节点

Unix对文件的内容和描述文件的信息给出了清楚的区分。除了设备文件和特殊文件系统文件外，每个文件都由字符序列组成。文件内容不包含任何控制信息，如文件长度或文件结束（end-of-file,EOF）符。

文件系统处理文件需要的所有信息包含在一个名为索引节点（*inode*）的数据结构中。每个文件都有自己的索引节点，文件系统用索引节点来标识文件。

虽然文件系统及内核函数对索引节点的处理可能随Unix系统的不同有很大的差异，但它们必须至少提供在 POSIX 标准中指定的如下属性：

- 文件类型（参见前一节）
- 与文件相关的硬链接个数
- 以字节为单位的文件长度
- 设备标识符（即包含文件的设备的标识符）
- 在文件系统中标识文件的索引节点号
- 文件拥有者的UID
- 文件的用户组ID
- 几个时间戳，表示索引节点状态改变的时间、最后访问时间及最后修改时间
- 访问权限和文件模式（参见下一节）

访问权限和文件模式

文件的潜在用户分为三种类型：

- 作为文件所有者的用户
- 同组用户，不包括所有者
- 所有剩下的用户（其他）

有三种类型的访问权限——读、写及执行每组用户都有这三种权限。因此，文件访问权限的组合就用九种不同的二进制来标记。还有三种附加的标记，即 *suid* (*Set User ID*)，*sgid* (*Set Group ID*)，及 *sticky* 用来定义文件的模式。当这些标记应用到可执行文件时有如下含义：

`suid`

进程执行一个文件时通常保持进程拥有者的UID。然而，如果设置了可执行文件`suid`的标志位，进程就获得了该文件拥有者的UID。

`sgid`

进程执行一个文件时保持进程组的用户组ID。然而，如果设置了可执行文件`sgid`的标志位，进程就获得了该文件用户组的ID。

`sticky`

设置了`sticky`标志位的可执行文件相当于向内核发出一个请求，当程序执行结束以后，依然将它保留在内存（注8）。

注8：　这个标志已经过时，现在使用基于代码页共享的其他方法（参见第九章）。

当文件由一个进程创建时，文件拥有者的ID就是该进程的UID。而其用户组ID可以是进程创建者的ID，也可以是父目录的ID，这取决于父目录sgid标志位的值。

文件操作的系统调用

当用户访问一个普通文件或目录文件的内容时，他实际上是访问存储在硬件块设备上的一些数据。从这个意义上说，文件系统是硬盘分区物理组织的用户级视图。因为处于用户态的进程不能直接与低层硬件交互，所以每个实际的文件操作必须在内核态下进行。因此，Unix操作系统定义了几个与文件操作有关的系统调用。

所有Unix内核都对硬件块设备的处理效率给予极大关注，其目的是为了获得非常好的系统整体性能。在后面的章节中，我们将描述Linux与文件操作相关的主题，尤其是讨论内核如何对文件相关的系统调用作出反应。为了理解这些内容，你需要知道如何使用文件操作的主要系统调用。下面对此给予描述。

打开文件

进程只能访问“打开的”文件。为了打开一个文件，进程调用系统调用：

```
fd=open(path, flag, mode)
```

其中的三个参数具有以下含义：

`path`

表示被打开文件的（相对或绝对）路径。

`flag`

指定文件打开的方式（例如，读、写、读/写、追加）。它也指定是否应当创建一个不存在的文件。

`mode`

指定新创建文件的访问权限。

这个系统调用创建一个“打开文件”对象，并返回所谓文件描述符（*file descriptor*）的标识符。一个打开文件对象包括：

- 文件操作的一些数据结构，如指定文件打开方式的一组标志；表示文件当前位置的`offset`字段，从这个位置开始将进行下一个操作（即所谓的文件指针），等等。
- 进程可以调用的一些内核函数指针。这组允许调用的函数集合由参数`flag`的值决定。

我们将在第十二章中详细讨论打开文件对象。在这里，我们仅描述一些POSIX语义所指定的一般特性：

- 文件描述符表示进程与打开文件之间的交互，而打开文件对象包含了与这种交互相关的数据。同一打开文件对象也许由同一个进程中的几个文件描述符标识。
- 几个进程也许同时打开同一文件。在这种情况下，文件系统给每个文件分配一个单独的打开文件对象以及单独的文件描述符。当这种情况发生时，Unix 文件系统对进程在同一文件上发出的I/O操作之间不提供任何形式的同步机制。然而，有几个系统调用，如 `flock()`，可用来让进程在整个文件或部分文件上对I/O操作实施同步（参见第十二章）。

为了创建一个新的文件，进程也可以调用 `create()` 系统调用，它与 `open()` 非常相似，都是由内核来处理。

访问打开的文件

对普通 Unix 文件，可以顺序地访问，也可以随机地访问，而对设备文件和命名管道文件，通常只能顺序地访问。在这两种访问方式中，内核把文件指针存放在打开文件对象中，也就是说，当前位置就是下一次进行读或写操作的位置。

顺序访问是文件的默认访问方式，即 `read()` 和 `write()` 系统调用总是从文件指针的当前位置开始读或写。为了修改文件指针的值，必须在程序中显式地调用 `lseek()` 系统调用。当打开文件时，内核让文件指针指向文件的第一个字节（偏移量为0）。

`lseek()` 系统调用需要下列参数：

```
newoffset=lseek(fd, offset, whence);
```

其参数含义如下：

`fd`

表示打开文件的文件描述符。

`offset`

指定一个有符号整数值，用来计算文件指针的新位置。

`whence`

指定文件指针新位置的计算方式：可以是offset加0，表示文件指针从文件头移动；也可以是offset加文件指针的当前位置，表示文件指针从当前位置移动；还可以是offset加文件最后一个字节的位置，表示文件指针从文件末尾开始移动。

read()系统调用需要以下参数：

```
nread= read(fd, buf, count);
```

其参数含义如下：

fd

表示打开文件的文件描述符。

buf

指定在进程地址空间中缓冲区的地址，所读的数据就放在这个缓冲区。

count

表示所读的字节数。

当处理这样的系统调用时，内核会尝试从拥有文件描述符fd的文件中读count个字节，其起始位置为打开文件的offset字段的当前值。在某些情况下可能遇到文件结束、空管道等等，因此内核无法成功地读出全部count个字节。返回的nread值就是实际所读的字节数。给原来的值加上nread就会更新文件指针。write()的参数与read()相似。

关闭文件

当进程无需再访问文件的内容时，就调用系统调用：

```
res=close(fd);
```

释放与文件描述符fd相对应的打开文件对象。当一个进程终止时，内核会关闭其所有仍然打开着的文件。

更名及删除文件

要重新命名或删除一个文件时，进程不需要打开它。实际上，这样的操作并没有对这个文件的内容起作用，而是对一个或多个目录的内容起作用。例如，系统调用：

```
res= rename(oldpath, newpath);
```

改变了文件链接的名字，而系统调用：

```
res= unlink(pathname);
```

减少了文件链接数，删除了相应的目录项。只有当链接数为0时，文件才被真正删除。

Unix 内核概述

Unix内核提供了应用程序可以运行的执行环境。因此，内核必须实现一组服务及相应的接口。应用程序使用这些接口，而且通常不会与硬件资源直接交互。

进程 / 内核模式

如前所述，CPU既可以运行在用户态下，也可以运行在内核态下。实际上，一些CPU可以有两种以上的执行状态。例如，Intel 80x86微处理器有四种不同的执行状态。但是，所有标准的Unix内核都仅仅利用了内核态和用户态。

当一个程序在用户态下执行时，它不能直接访问内核数据结构或内核的程序。然而，当应用程序在内核态下运行时，这些限制不再有效。每种CPU模型都为从用户态到内核态的转换提供了特殊的指令，反之亦然。一个程序执行时，大部分时间都处在用户态下，只有需要内核所提供的服务时才切换到内核态。当内核满足了用户程序的请求后，它让程序又回到用户态下。

进程是动态的实体，在系统内通常只有有限的生存期。创建、撤消及同步现有进程的任务都委托给内核中的一组例程来完成。

内核本身并不是一个进程，而是进程的管理者。进程 / 内核模式假定：请求内核服务的进程使用所谓系统调用（*system call*）的特殊编程机制。每个系统调用都设置了一组识别进程请求的参数，然后执行与硬件相关的CPU指令完成从用户态到内核态的转换。

除用户进程之外，Unix系统还包括几个所谓内核线程（*kernel thread*）的特权进程（被赋予特殊权限的进程），它们具有以下特点：

- 它们以内核态运行在内核地址空间。
- 它们不与用户直接交互，因此不需要终端设备。
- 它们通常在系统启动时创建，然后一直处于活跃状态直到系统关闭。

在单处理器系统中，任何时候只有一个进程在运行，它要么处于用户态，要么处于内核态。如果进程运行在内核态，处理器就执行一些内核例程。图1-2举例说明了用户态与内核态之间的相互转换。处于用户态的进程1发出系统调用之后，进程切换到内核态，系统调用被执行。然后，直到发生定时中断且调度程序在内核态被激活，进程1才恢复在用户态下执行。进程切换发生，进程2在用户态开始执行，直到硬件设备发出中断请求。中断的结果是，进程2切换到内核态并处理中断。

Unix 内核做的工作远不止处理系统调用。实际上，可以有几种方式激活内核例程：

- 进程调用系统调用。
- 正在执行进程的CPU发出一个异常（*exception*）信号，异常是一些反常情况，例如一个无效的指令。内核代表产生异常的进程处理异常。

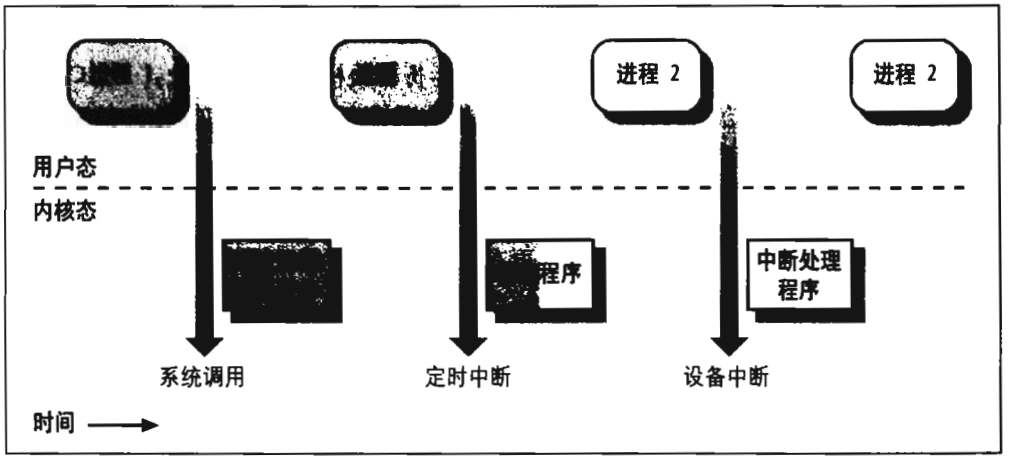


图 1-2：用户态与内核态之间的转换

- 外围设备向CPU发出一个中断（*interrupt*）信号以通知一个事件的发生，如一个要求注意的请求、一个状态的变化或一个I/O操作已经完成等。每个中断信号都是由内核中的中断处理程序（*interrupt handler*）来处理的。因为外围设备与CPU异步操作，因此，中断在不可预知的时间发生。
- 内核线程被执行。因为内核线程运行在内核态，因此必须认为其相应程序是内核的一部分。

进程实现

为了让内核管理进程，每个进程由一个进程描述符（*process descriptor*）表示，这个描述符包含有关进程当前状态的信息。

当内核暂停一个进程的执行时，就把几个相关处理器寄存器的内容保存在进程描述符中。这些寄存器包括：

- 程序计数器（PC）和栈指针（SP）寄存器
- 通用寄存器

- 浮点寄存器
- 包含CPU状态信息的处理器控制寄存器（处理器状态字，Processor Status Word）
- 用来跟踪进程对RAM访问的内存管理寄存器

当内核决定恢复执行一个进程时，它用进程描述符中合适的字段来装载CPU寄存器。因为程序计数器中所存的值指向下一条将要执行的指令，所以进程从它停止的地方恢复执行。

当一个进程不在CPU上执行时，它正在等待某一事件。Unix内核可以区分很多等待状态，这些等待状态通常由进程描述符队列实现。每个（可能为空）队列对应一组等待特定事件的进程。

可重入内核

所有的Unix内核都是可重入的（*reentrant*），这意味着若干个进程可以同时在内核态下执行。当然，在单处理器系统上只有一个进程在真正运行，但是有许多进程可能在等待CPU或某一I/O操作完成时在内核态下被阻塞。例如，当内核代表某一进程发出一个读磁盘请求后，就让磁盘控制器处理这个请求并且恢复执行其他进程。当设备满足了读请求时，有一个中断就会通知内核，从而以前的进程可以恢复执行。

提供可重入的一种方式是编写函数，以便这些函数只能修改局部变量，而不能改变全局数据结构，这样的函数叫可重入函数。但是可重入内核不仅仅局限于这样的可重入函数（尽管一些实时内核正是如此实现的）。相反，可重入内核可以包含非重入函数，并且利用锁机制保证一次只有一个进程执行一个非重入函数。

如果一个硬件中断发生，可重入内核能挂起当前正在执行的进程，即使这个进程处于内核态。这种能力是非常重要的，因为这能提高发出中断的设备控制器的吞吐量。一旦设备已发出一个中断，它就一直等待直到CPU应答它为止。如果内核能够快速应答，设备控制器在CPU处理中断时就能执行其他任务。

现在，让我们看一下内核的可重入性及它对内核组织的影响。内核控制路径（*kernel control path*）表示内核处理系统调用、异常或中断所执行的指令序列。

在最简单的情况下，CPU从第一条指令到最后一条指令顺序地执行内核控制路径。然而，当下述事件之一发生时，CPU交错执行内核控制路径：

- 运行在用户态下的进程调用一个系统调用，而相应的内核控制路径证实这个请求无法立即得到满足；然后，内核控制路径调用调度程序选择一个新的进程投入运行。

结果，进程切换发生。第一个内核控制路径还没完成，而CPU又重新开始执行其他的内核控制路径。在这种情况下，两条控制路径代表两个不同的进程在执行。

- 当运行一个内核控制路径时，CPU检测到一个异常（例如，访问一个不在RAM中的页）。第一个控制路径被挂起，而CPU开始执行合适的过程。在我们的例子中，这种过程能给进程分配一个新页，并从磁盘读它的内容。当这个过程结束时，第一个控制路径可以恢复执行。在这种情况下，两个控制路径代表同一个进程在执行。
- 当CPU正在运行一个启用了中断的内核控制路径时，一个硬件中断发生。第一个内核控制路径还没执行完，CPU开始执行另一个内核控制路径来处理这个中断。当这个中断处理程序终止时，第一个内核控制路径恢复。在这种情况下，两个内核控制路径运行在同一进程的可执行上下文中，所花费的系统CPU时间都算给这个进程。然而，中断处理程序无需代表这个进程运行。
- 在支持抢占式调度的内核中，CPU正在运行，而一个更高优先级的进程加入就绪队列，则中断发生。在这种情况下，第一个内核控制路径还没有执行完，CPU代表高优先级进程又开始执行另一个内核控制路径。只有把内核编译成支持抢占式调度之后，才可能出现这种情况。

图1-3显示了非交错的和交错的内核控制路径的几个例子。考虑以下三种不同的CPU状态：

- 在用户态下运行一个进程（User）
- 运行一个异常处理程序或系统调用处理程序（Excp）
- 运行一个中断处理程序（Intr）

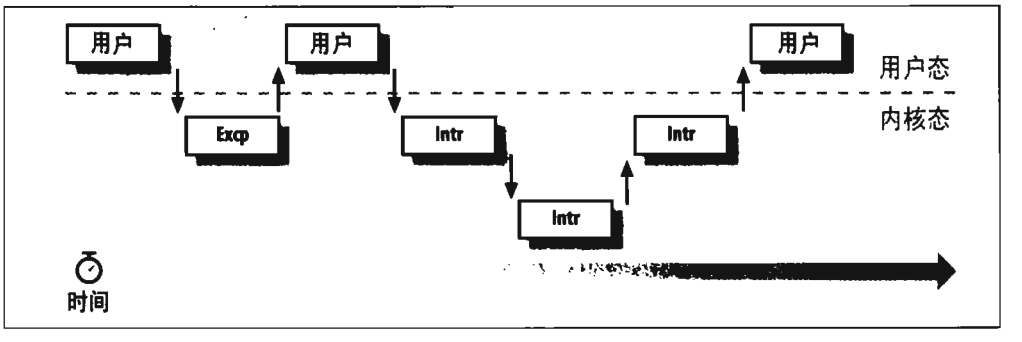


图1-3：内核控制路径的交错执行

进程地址空间

每个进程运行在它的私有地址空间。在用户态下运行的进程涉及到私有栈、数据区和代码区。当在内核态运行时，进程访问内核的数据区和代码区，但使用另外的私有栈。

因为内核是可重入的，因此几个内核控制路径（每个都与不同的进程相关）可以轮流执行。在这种情况下，每个内核控制路径都引用它自己的私有内核栈。

尽管看起来每个进程访问一个私有地址空间，但有时进程之间也共享部分地址空间。在一些情况下，这种共享由进程显式地提出；在另外一些情况下，由内核自动完成共享以节约内存。

如果同一个程序（比如说编辑程序）由几个用户同时使用，则这个程序只被装入内存一次，其指令由所有需要它的用户共享。当然，其数据不被共享，因为每个用户将有独立的数据。这种共享的地址空间由内核自动完成以节省内存。

进程间也能共享部分地址空间，以实现一种进程间通信，这就是由 System V 引入并且已经被 Linux 支持的“共享内存”技术。

最后，Linux 支持 `mmap()` 系统调用，该系统调用允许存放在块设备上的文件或信息的一部分映射到进程的部分地址空间。内存映射为正常的读写传送数据方式提供了另一种选择。如果同一文件由几个进程共享，那么共享它的每个进程地址空间都包含有它的内存映射。

同步和临界区

实现可重入内核需要利用同步机制：如果内核控制路径对某个内核数据结构进行操作时被挂起，那么，其他的内核控制路径就不应当再对该数据结构进行操作，除非它已被重新设置成一致性（consistent）状态。否则，两个控制路径的交互作用将破坏所存储的信息。

例如，假设全局变量 V 包含某个系统资源的可用项数。第一个内核控制路径 A 读这个变量，并且确定仅有一个可用资源项。这时，另一个内核控制路径 B 被激活，并读同一个变量 V，V 的值仍为 1。因此，B 对 V 减 1，并开始用这个资源项。然后，A 恢复执行。因为 A 已经读到 V 的值，于是它假定自己可以对 V 减 1 并获取 B 已经在使用的这个资源项。结果，V 的值变为 -1，两个内核控制路径使用相同的资源项有可能导致灾难性的后果。

当某个计算结果取决于如何调度两个或多个进程时，相关代码就是不正确的。我们说存在一种竞争条件（*race condition*）。

一般来说，对全局变量的安全访问通过原子操作（*atomic operation*）来保证。在前面的例子中，如果两个控制路径读V并减1是一个单独的、不可中断的操作，那么，就不可能出现数据讹误。然而，内核包含的很多数据结构是无法用单一操作访问的。例如，用单一的操作从链表中删除一个元素是不可能的，因为内核一次至少访问两个指针。临界区（*critical region*）是这样的一段代码，进入这段代码的进程必须完成，之后另一个进程才能进入（注9）。

这些问题不仅出现在内核控制路径之间，也出现在共享公共数据的进程之间。几种同步技术已经被采用。以下将集中讨论怎样同步内核控制路径。

非抢占式内核

在寻找彻底、简单地解决同步问题的方案中，大多数传统的Unix内核都是非抢占式的：当进程在内核态执行时，它不能被任意挂起，也不能被另一个进程代替。因此，在单处理器系统上，中断或异常处理程序不能修改的所有内核数据结构，内核对它们的访问都是安全的。

当然，内核态的进程能自愿放弃CPU，但是在这种情况下，它必须确保所有的数据结构都处于一致性状态。此外，当这种进程恢复执行时，它必须重新检查以前访问过的数据结构的值，因为这些数据结构有可能被改变。

如果内核支持抢占，那么在应用同步机制时，确保进入临界区前禁止抢占，退出临界区时启用抢占。

非抢占能力在多处理器系统上是低效的，因为运行在不同CPU上的两个内核控制路径本可以并发地访问相同的数据结构。

禁止中断

单处理器系统上的另一种同步机制是：在进入一个临界区之前禁止所有硬件中断，离开时再重新启用中断。这种机制尽管简单，但远不是最佳的。如果临界区比较大，那么在一个相对较长的时间内持续禁止中断就可能使所有的硬件活动处于冻结状态。

此外，由于在多处理器系统中禁止本地CPU上的中断是不够的，所以必须使用其他的同步技术。

注9：同步问题已在其他著作中进行了详细描述。有兴趣的读者可以参考有关Unix操作系统方面的书（参见本书末尾的“参考书目”）。

信号量

广泛使用的一种机制是信号量(*semaphore*)，它在单处理器系统和多处理器系统上都有效。信号量仅仅是与一个数据结构相关的计数器。所有内核线程在试图访问这个数据结构之前，都要检查这个信号量。可以把每个信号量看成一个对象，其组成如下：

- 一个整数变量
- 一个等待进程的链表
- 两个原子方法：down()和up()

down()方法对信号量的值减1，如果这个新值小于0，该方法就把正在运行的进程加入到这个信号量链表，然后阻塞该进程(即调用调度程序)。up()方法对信号量的值加1，如果这个新值大于或等于0，则激活这个信号量链表中的一个或多个进程。

每个要保护的数据结构都有它自己的信号量，其初始值为1。当内核控制路径希望访问这个数据结构时，它在相应的信号量上执行down()方法。如果信号量的当前值不是负数，则允许访问这个数据结构。否则，把执行内核控制路径的进程加入到这个信号量的链表并阻塞该进程。当另一个进程在那个信号量上执行up()方法时，允许信号量链表上的一个进程继续执行。

自旋锁

在多处理器系统中，信号量并不总是解决同步问题的最佳方案。系统不允许在不同CPU上运行的内核控制路径同时访问某些内核数据结构，在这种情况下，如果修改数据结构所需的时间比较短，那么，信号量可能是很低效的。为了检查信号量，内核必须把进程插入到信号量链表中，然后挂起它。因为这两种操作比较费时，完成这些操作时，其他的内核控制路径可能已经释放了信号量。

在这些情况下，多处理器操作系统使用了自旋锁(*spin lock*)。自旋锁与信号量非常相似，但没有进程链表；当一个进程发现锁被另一个进程锁着时，它就不停地“旋转”，执行一个紧凑的循环指令直到锁打开。

当然，自旋锁在单处理器环境下是无效的。当内核控制路径试图访问一个上锁的数据结构时，它开始无休止循环。因此，内核控制路径可能因为正在修改受保护的数据结构而没有机会继续执行，也没有机会释放这个自旋锁。最后的结果可能是系统挂起。

避免死锁

与其他控制路径同步的进程或内核控制路径很容易进入死锁(*deadlock*)状态。举一个

最简单的死锁的例子，进程*p1*获得访问数据结构*a*的权限，进程*p2*获得访问*b*的权限，但是*p1*在等待*b*，而*p2*在等待*a*。进程之间其他更复杂的循环等待的情况也可能发生。显然，死锁情形会导致受影响的进程或内核控制路径完全处于冻结状态。

只要涉及到内核设计，当所用内核信号量的数量较多时，死锁就成为一个突出问题。在这种情况下，很难保证内核控制路径在各种可能方式下的交错执行不出现死锁状态。有几种操作系统（包括Linux）通过按规定的顺序请求信号量来避免死锁。

信号和进程间通信

Unix信号（*signal*）提供了把系统事件报告给进程的一种机制。每种事件都有自己的信号编号，通常用一个符号常量来表示，例如SIGTERM。有两种系统事件：

异步通告

例如，当用户在终端按下中断键（通常为CTRL-C）时，即向前台进程发出中断信号SIGINT。

同步错误或异常

例如，当进程访问内存非法地址时，内核向这个进程发送一个SIGSEGV信号。

POSIX标准定义了大约20种不同的信号，其中，有两种是用户自定义的，可以当作用户态下进程通信和同步的原语机制。一般来说，进程可以以两种方式对接收到的信号做出反应：

- 忽略该信号。
- 异步地执行一个指定的过程（信号处理程序）。

如果进程不指定选择何种方式，内核就根据信号的编号执行一个默认操作。五种可能的默认操作是：

- 终止进程。
- 将执行上下文和进程地址空间的内容写入一个文件（核心转储，core dump），并终止进程。
- 忽略信号。
- 挂起进程。
- 如果进程曾被暂停，则恢复它的执行。

因为POSIX语义允许进程暂时阻塞信号，因此内核信号的处理相当精细。此外，SIGKILL和SIGSTOP信号不能直接由进程处理，也不能由进程忽略。

AT&T的Unix System V引入了在用户态下其他种类的进程间通信机制，很多Unix内核也采用了这些机制：信号量、消息队列及共享内存。它们被统称为*System V IPC*。

内核把它们作为*IPC*资源来实现：进程要获得一个资源，可以调用shmget()、semget()或msgget()系统调用。与文件一样，IPC资源是持久不变的，进程创建者、进程拥有者或超级用户进程必须显式地释放这些资源。

这里的信号量与本章“同步和临界区”一节中所描述的信号量是相似的，只是它们用在用户态下的进程中。消息队列允许进程利用msgsnd()及msgget()系统调用交换消息，msgsnd()表示把消息插入到指定的队列中，msgget()表示从队列中提取消息。

POSIX标准（IEEE Std 1003.1-2001）定义了一种基于消息队列的IPC机制，这就是所谓的POSIX消息队列。它们和System V IPC消息队列是相似的，但是，它们对应用程序提供一个更简单的基于文件的接口。

共享内存为进程之间交换和共享数据提供了最快的方式。通过调用shmget()系统调用来创建一个新的共享内存，其大小按需设置。在获得IPC资源标识符后，进程调用shmat()系统调用，其返回值是进程的地址空间中新区域的起始地址。当进程希望把共享内存从其地址空间分离出去时，就调用shmdt()系统调用。共享内存的实现依赖于内核对进程地址空间的实现。

进程管理

Unix在进程和它正在执行的程序之间做出一个清晰的划分。fork()和_exit()系统调用分别用来创建一个新进程和终止一个进程，而调用exec()类系统调用则是装入一个新程序。当这样一个系统调用执行以后，进程就在所装入程序的全新地址空间恢复运行。

调用fork()的进程是父进程，而新进程是它的子进程。父子进程能互相找到对方，因为描述每个进程的数据结构都包含有两个指针，一个直接指向它的父进程，另一个直接指向它的子进程。

- 实现fork()一种天真的方式就是将父进程的数据与代码都复制，并把这个拷贝赋予子进程。这会相当费时。当前依赖硬件分页单元的内核采用写时复制（Copy-On-Write）技术，即把页的复制延迟到最后一刻（也就是说，直到父或子进程需要时才写进页）。我们将在第九章“写时复制”一节中描述Linux是如何实现这一技术的。

_exit()系统调用终止一个进程。内核对这个系统调用的处理是通过释放进程所拥有的资源并向父进程发送SIGCHILD信号（默认操作为忽略）来实现的。

僵死进程（zombie process）

父进程如何查询其子进程是否终止了呢？wait4()系统调用允许进程等待，直到其中的一个子进程结束；它返回已终止子进程的进程标识符（Process ID，PID）。

内核在执行这个系统调用时，检查子进程是否已经终止。引入僵死进程的特殊状态是为了表示终止的进程：父进程执行完wait4()系统调用之前，进程就一直停留在那种状态。系统调用处理程序从进程描述符字段中获取有关资源使用的一些数据；一旦得到数据，就可以释放进程描述符。当进程执行wait4()系统调用时如果没有子进程结束，内核就通常把该进程设置成等待状态，一直到子进程结束。

很多内核也实现了waitpid()系统调用，它允许进程等待一个特殊的子进程。其他wait4()系统调用的变体也是相当通用的。

在父进程发出wait4()调用之前，让内核保存子进程的有关信息是一个良好的习惯，但是，假设父进程终止而没有发出wait4()调用呢？这些信息占用了一些内存中非常有用的位置，而这些位置本来可以用来为活动着的进程提供服务。例如，很多shell允许用户在后台启动一个命令然后退出。正在运行这个shell命令的进程终止，但它的子进程继续运行。

解决的办法是使用一个名为*init*的特殊系统进程，它在系统初始化的时候被创建。当一个进程终止时，内核改变其所有现有子进程的进程描述符指针，使这些子进程成为*init*的孩子。*init*监控所有子进程的执行，并且按常规发布wait4()系统调用，其副作用就是除掉所有僵死的进程。

进程组和登录会话

现代Unix操作系统引入了进程组（*process group*）的概念，以表示一种“作业（job）”的抽象。例如，为了执行命令行：

```
$ ls | sort | more
```

Shell支持进程组，例如bash，为三个相应的进程ls、sort及more创建了一个新的组。shell以这种方式作用于这三个进程，就好像它们是一个单独的实体（更准确地说是作业）。每个进程描述符包括一个包含进程组ID的字段。每一进程组可以有一个领头进

程（即其PID与这个进程组的ID相同的进程）。新创建的进程最初被插入到其父进程的进程组中。

现代Unix内核也引入了登录会话（*login session*）。非正式地说，一个登录会话包含在指定终端已经开始工作会话的那个进程的所有后代进程——通常情况下，登录会话就是shell进程为用户创建的第一条命令。进程组中的所有进程必须在同一登录会话中。一个登录会话可以让几个进程组同时处于活动状态，其中，只有一个进程组一直处于前台，这意味着该进程组可以访问终端，而其他活动着的进程组在后台。当一个后台进程试图访问终端时，它将收到SIGTTIN或SIGTTOUT信号。在很多shell命令中，用内部命令bg和fg把一个进程组放在后台或者前台。

内存管理

内存管理是迄今为止Unix内核中最复杂的活动。在本书中，我们将用超过三分之一的篇幅来描述Linux是如何实现它的。本节只说明一些与内存管理相关的主要问题。

虚拟内存

所有新近的Unix系统都提供了一种有用的抽象，叫虚拟内存（*virtual memory*）。虚拟内存作为一种逻辑层，处于应用程序的内存请求与硬件内存管理单元（Memory Management Unit，MMU）之间。虚拟内存有很多用途和优点：

- 若干个进程可以并发地执行。
- 应用程序所需内存大于可用物理内存时也可以运行。
- 程序只有部分代码装入内存时进程可以执行它。
- 允许每个进程访问可用物理内存的子集。
- 进程可以共享库函数或程序的一个单独内存映像。
- 程序是可重定位的，也就是说，可以把程序放在物理内存的任何地方。
- 程序员可以编写与机器无关的代码，因为他们不必关心有关物理内存的组织结构。

虚拟内存子系统的主要成分是虚拟地址空间（*virtual address space*）的概念。进程所用的一组内存地址不同于物理内存地址。当进程使用一个虚拟地址时（注10），内核和MMU协同定位其在内存中的实际物理位置。

注10：这些地址的叫法在不同的计算机体系结构中是不一样的。正如我们在第二章中将会看到的一样，Intel使用手册把它们叫做“逻辑地址”。

现在的CPU包含了能自动把虚拟地址转换成物理地址的硬件电路。为了达到这个目标，把可用RAM划分成长度为4KB或8KB的页框（page frame），并且引入一组页表来指定虚拟地址与物理地址之间的对应关系。这些电路使内存分配变得简单，因为一块连续的虚拟地址请求可以通过分配一组非连续的物理地址页框而得到满足。

随机访问存储器（RAM）的使用

所有的Unix操作系统都将RAM毫无疑义地划分为两部分，其中若干兆字节专门用于存放内核映像（也就是内核代码和内核静态数据结构）。RAM的其余部分通常由虚拟内存系统来处理，并且用在以下三种可能的方面：

- 满足内核对缓冲区、描述符及其他动态内核数据结构的请求。
- 满足进程对一般内存区的请求及对文件内存映射的请求。
- 借助于高速缓存从磁盘及其他缓冲设备获得较好的性能。

每种请求类型都是重要的。但从另一方面来说，因为可用RAM是有限的，所以必须在请求类型之间做出平衡，尤其是当可用内存没有剩下多少时。此外，当可用内存达到临界阈值时，可以调用页框回收（page-frame-reclaiming）算法释放其他内存，那么哪些页框是最适合回收的页框呢？正如我们将在第十七章中看到的一样，对这个问题既没有简单的答案，也没有多少理论的支持，惟一可用的解决方法是开发经过仔细调节的经验算法。

虚拟内存系统必须解决的一个主要问题是内存碎片。理想情况下，只有当空闲页框数太少时，内存请求才失败。然而，通常要求内核使用物理上连续的内存区域，因此，即使有足够的可用内存，但它不能作为一个连续的大块使用时，内存的请求也会失败。

内核内存分配器

内核内存分配器（*Kernel Memory Allocator*，*KMA*）是一个子系统，它试图满足系统中所有部分对内存的请求。其中一些请求来自内核其他子系统，它们需要一些内核使用的内存，还有一些请求来自于用户程序的系统调用，用来增加用户进程的地址空间。一个好的KMA应该具有下列特点：

- 必须快。实际上，这是最重要的属性，因为它由所有的内核子系统（包括中断处理程序）调用。
- 必须把内存的浪费减到最少。
- 必须努力减轻内存的碎片（fragmentation）问题。

- 必须能与其他内存管理子系统合作，以便借用和释放页框。

基于各种不同的算法技术，已经提出了几种KMA，包括：

- 资源图分配算法（allocator）
- 2的幂次方空闲链表
- McKusick-Karels分配算法
- 伙伴（Buddy）系统
- Mach的区域（Zone）分配算法
- Dynix分配算法
- Solaris的Slab分配算法

我们将在第八章中看到，Linux的KMA在伙伴系统之上采用了Slab分配算法。

进程虚拟地址空间处理

进程的虚拟地址空间包括了进程可以引用的所有虚拟内存地址。内核通常用一组内存区描述符描述进程虚拟地址空间。例如，当进程通过`exec()`类系统调用开始某个程序的执行时，内核分配给进程的虚拟地址空间由以下内存区组成：

- 程序的可执行代码
- 程序的初始化数据
- 程序的未初始化数据
- 初始程序栈（即用户态栈）
- 所需共享库的可执行代码和数据
- 堆（由程序动态请求的内存）

所有现代Unix操作系统都采用了所谓请求调页（*demand paging*）的内存分配策略。有了请求调页，进程可以在它的页还没有在内存时就开始执行。当进程访问一个不存在的页时，MMU产生一个异常；异常处理程序找到受影响的内存区，分配一个空闲的页，并用适当的数据把它初始化。同理，当进程通过调用`malloc()`或`brk()`（由`malloc()`在内部调用）系统调用动态地请求内存时，内核仅仅修改进程的堆内存区的大小。只有试图引用进程的虚拟内存地址而产生异常时，才给进程分配页框。

虚拟地址空间也采用其他更有效的策略，如前面提到的写时复制策略。例如，当一个新

进程被创建时，内核仅仅把父进程的页框赋给子进程的地址空间，但是把这些页框标记为只读。一旦父或子进程试图修改页中的内容时，一个异常就会产生。异常处理程序把新页框赋给受影响的进程，并用原来页中的内容初始化新页框。

高速缓存

物理内存的一大优势就是用作磁盘和其他块设备的高速缓存。这是因为硬盘非常慢：磁盘的访问需要数毫秒，与RAM的访问时间相比，这太长了。因此，磁盘通常是影响系统性能的瓶颈。通常，在最早的Unix系统中就已经实现的一个策略是：尽可能地推迟写磁盘的时间，因此，从磁盘读入内存的数据即使任何进程都不再使用它们，它们也继续留在RAM中。

这一策略的前题是有好机会摆在面前：新进程请求从磁盘读或写的数据，就是被撤消进程曾拥有的数据。当一个进程请求访问磁盘时，内核首先检查进程请求的数据是否在缓存中，如果在（把这种情况叫做缓存命中），内核就可以为进程请求提供服务而不用访问磁盘。

`sync()`系统调用把所有“脏”的缓冲区（即缓冲区的内容与对应磁盘块的内容不一样）写入磁盘来强制磁盘同步。为了避免数据丢失，所有的操作系统都会注意周期性地把脏缓冲区写回磁盘。

设备驱动程序

内核通过设备驱动程序（*device driver*）与I/O设备交互。设备驱动程序包含在内核中，由控制一个或多个设备的数据结构和函数组成，这些设备包括硬盘、键盘、鼠标、监视器、网络接口及连接到SCSI总线上的设备。通过特定的接口，每个驱动程序与内核中的其余部分（甚至与其他驱动程序）相互作用这种方式具有以下优点：

- 可以把特定设备的代码封装在特定的模块中。
- 厂商可以在不了解内核源代码而只知道接口规范的情况下，就能增加新的设备。
- 内核以统一的方式对待所有的设备，并且通过相同的接口访问这些设备。
- 可以把设备驱动程序写成模块，并动态地把它们装进内核而不需要重新启动系统。不再需要时，也可以动态地卸下模块，以减少存储在RAM中的内核映像的大小。

图1-4说明了设备驱动程序与内核其他部分及进程之间的接口。

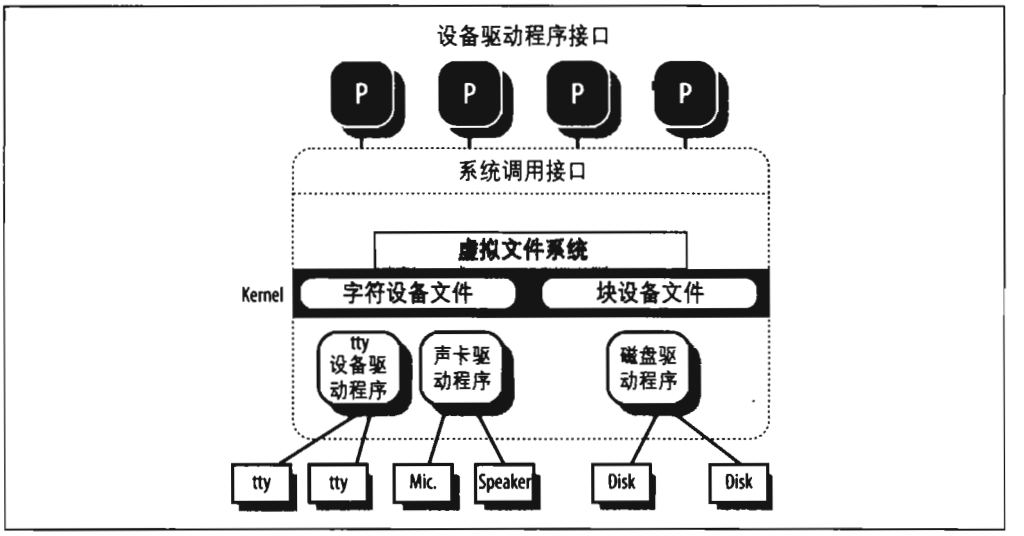


图 1-4：设备驱动程序接口

一些用户程序（P）希望操作硬件设备。这些程序就利用常用的、与文件相关的系统调用及在*/dev*目录下能找到的设备文件向内核发出请求。实际上，设备文件是设备驱动程序接口中用户可见的部分。每个设备文件都有专门的设备驱动程序，它们由内核调用以执行对硬件设备的请求操作。

这里值得一提的是，在Unix刚出现的时候，图形终端是罕见而且昂贵的，因此Unix内核只直接处理字符终端。当图形终端变得非常普遍时，一些如X Window系统那样的特别的应用就出现了，它们以标准进程的身份运行，并且能直接访问图形界面的I/O端口和RAM的视频区域。一些新近的Unix内核，例如Linux 2.6，对图形卡的帧缓冲提供了一种抽象，从而允许应用软件无需了解图形界面的I/O端口的任何知识就能对其进行访问（参见第十三章“内核支持的级别”一节）。

第二章



内存寻址

本章介绍寻址技术。值得庆幸的是，操作系统自身不必完全了解物理内存；如今的微处理器包含的硬件线路使内存管理既高效又健壮，所以编程错误就不会对该程序之外的内存产生非法访问。

作为本书的一部分，本章将详细描述80x86微处理器怎样进行芯片级的内存寻址，Linux又是如何利用寻址硬件的。我们希望当你学习内存寻址技术在Linux最流行的硬件平台上的详细实现方法时，既能够更好地理解分页单元的一般原理，又能更好地研究内存寻址技术在其他平台上是如何实现的。

关于内存管理有三章，这是其中的第一章；还有第八章，讨论内核怎样给自己分配主存；以及第九章，考虑怎样给进程分配线性地址。

内存地址

程序员偶尔会引用内存地址（*memory address*）作为访问内存单元内容的一种方式，但是，当使用80x86微处理器时，我们必须区分以下三种不同的地址：

逻辑地址（logical address）

包含在机器语言指令中用来指定一个操作数或一条指令的地址。这种寻址方式在80x86著名的分段结构中表现得尤为具体，它促使MS-DOS或Windows程序员把程序分成若干段。每一个逻辑地址都由一个段（*segment*）和偏移量（*offset*或*displacement*）组成，偏移量指明了从段开始的地方到实际地址之间的距离。

线性地址（linear address）(也称虚拟地址 virtual address)

是一个32位无符号整数，可以用来表示高达4GB的地址，也就是，高达4 294 967 296个内存单元。线性地址通常用十六进制数字表示，值的范围从0x00000000到0xffffffff。

物理地址（physical address）

用于内存芯片级内存单元寻址。它们与从微处理器的地址引脚发送到内存总线上的电信号相对应。物理地址由32位或36位无符号整数表示。

内存控制单元（MMU）通过一种称为分段单元（segmentation unit）的硬件电路把一个逻辑地址转换成线性地址；接着，第二个称为分页单元（paging unit）的硬件电路把线性地址转换成一个物理地址（见图2-1）。

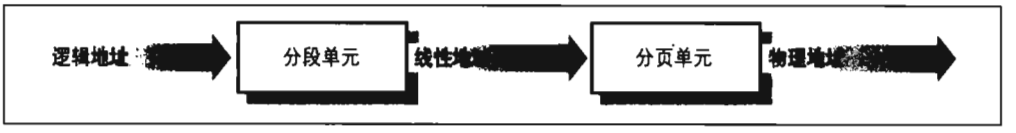


图2-1：逻辑地址转换

在多处理器系统中，所有CPU都共享同一内存；这意味着RAM芯片可以由独立的CPU并发地访问。因为在RAM芯片上的读或写操作必须串行地执行，因此一种所谓内存仲裁器（*memory arbiter*）的硬件电路插在总线和每个RAM芯片之间。其作用是如果某个RAM芯片空闲，就准予一个CPU访问，如果该芯片忙于为另一个处理器提出的请求服务，就延迟这个CPU的访问。即使在单处理器上也使用内存仲裁器，因为单处理器系统中包含一个叫做DMA控制器的特殊处理器，而DMA控制器与CPU并发操作［参见第十三章“直接内存访问（DMA）”一节”］。在多处理器系统的情况下，因为仲裁器有多个输入端口，所以其结构更加复杂。例如，双Pentium在每个芯片的入口维持一个两端口仲裁器，并在试图使用公用总线前请求两个CPU交换同步信息。从编程观点看，因为仲裁器由硬件电路管理，因此它是隐藏的。

硬件中的分段

从80286模型开始，Intel微处理器以两种不同的方式执行地址转换，这两种方式分别称为实模式（*real mode*）和保护模式（*protected mode*）。我们将从下一节开始描述保护模式下的地址转换。实模式存在的主要原因是要维持处理器与早期模型兼容，并让操作系统自举（参阅附录一中针对实模式的简短描述）。

段选择符和段寄存器

一个逻辑地址由两部分组成：一个段标识符和一个指定段内相对地址的偏移量。段标识符是一个16位长的字段，称为段选择符（*Segment Selector*）如图2-2所示，而偏移量是一个32位长的字段。我们将在本章“快速访问段描述符”一节描述段选择符字段。

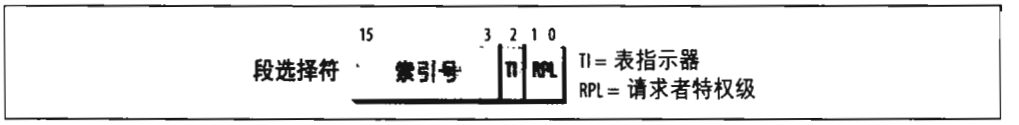


图2-2：段描述符格式

为了快速方便地找到段选择符，处理器提供段寄存器，段寄存器的唯一目的是存放段选择符。这些段寄存器称为cs，ss，ds，es，fs和gs。尽管只有6个段寄存器，但程序可以把同一个段寄存器用于不同的目的，方法是先将其值保存在内存中，用完后再恢复。

6个寄存器中3个有专门的用途：

`cs` 代码段寄存器，指向包含程序指令的段。

`ss` 栈段寄存器，指向包含当前程序栈的段。

`ds` 数据段寄存器，指向包含静态数据或者全局数据段。

其他3个段寄存器作一般用途，可以指向任意的数据段。

cs寄存器还有一个很重要的功能：它含有一个两位的字段，用以指明CPU的当前特权级（Current Privilege Level，CPL）。值为0代表最高优先级，而值为3代表最低优先级。Linux只用0级和3级，分别称之为内核态和用户态。

段描述符

每个段由一个8字节的段描述符（*Segment Descriptor*）表示，它描述了段的特征。段描述符放在全局描述符表（*Global Descriptor Table*,*GDT*）或局部描述符表（*Local Descriptor Table*，*LDT*）中。

通常只定义一个GDT，而每个进程除了存放在GDT中的段之外如果还需要创建附加的段，就可以有自己的LDT。GDT在主存中的地址和大小存放在`gdtr`控制寄存器中，当前正被使用的LDT地址和大小放在`ldtr`控制寄存器中。

图2-3阐明了段描述符的格式；表2-1解释了图中各个字段的含义

表 2-1：段描述符字段

字段名	描述
Base	包含段的首字节的线性地址
G	粒度标志：如果该位清 0，则段大小以字节为单位，否则以 4096 字节的倍数计
Limit	存放段中最后一个内存单元的偏移量，从而决定段的长度。如果 G 被置为 0，则一个段的大小在 1 个字节到 1MB 之间变化；否则，将在 4KB 到 4GB 之间变化
S	系统标志：如果它被清 0，则这是一个系统段，存储诸如 LDT 这种关键的数据结构，否则它是一个普通的代码段或数据段
Type	描述了段的类型特征和它的存取权限（请看表下面的描述）
DPL	描述符特权级（*Descriptor Privilege Level*）字段：用于限制对这个段的存取。它表示为访问这个段而要求的 CPU 最小的优先级。因此，DPL 设为 0 的段只能当 CPL 为 0 时（即在内核态）才是可访问的，而 DPL 设为 3 的段对任何 CPL 值都是可访问的
P	*Segment-Present* 标志：等于 0 表示段当前不在主存中。Linux 总是把这个标志（第 47 位）设为 1，因为它从来不把整个段交换到磁盘上去
D 或 B	称为 D 或 B 的标志，取决于是代码段还是数据段。D 或 B 的含义在两种情况下稍微有所区别，但是如果段偏移量的地址是 32 位长，就基本上把它置为 1，如果这个偏移量是 16 位长，它被清 0（更详细的描述参见 Intel 使用手册）
AVL 标志	可以由操作系统使用，但是被 Linux 忽略

有几种不同类型的段以及和它们对应的的段描述符。下面列出了 Linux 中被广泛采用的类型：

代码段描述符

表示这个段描述符代表一个代码段，它可以放在 GDT 或 LDT 中。该描述符置 S 标志为 1（非系统段）。

数据段描述符

表示这个段描述符代表一个数据段，它可以放在 GDT 或 LDT 中。该描述符置 S 标志为 1。栈段是通过一般的数据段实现的。

任务状态段描述符（TSSD）

表示这个段描述符代表一个任务状态段（Task State Segment，TSS），也就是说这个段用于保存处理器寄存器的内容（参见第三章中的“任务状态段”一节）。它只

能出现在GDT中。根据相应的进程是否正在CPU上运行，其Type字段的值分别为11或9。这个描述符的S标志置为0。

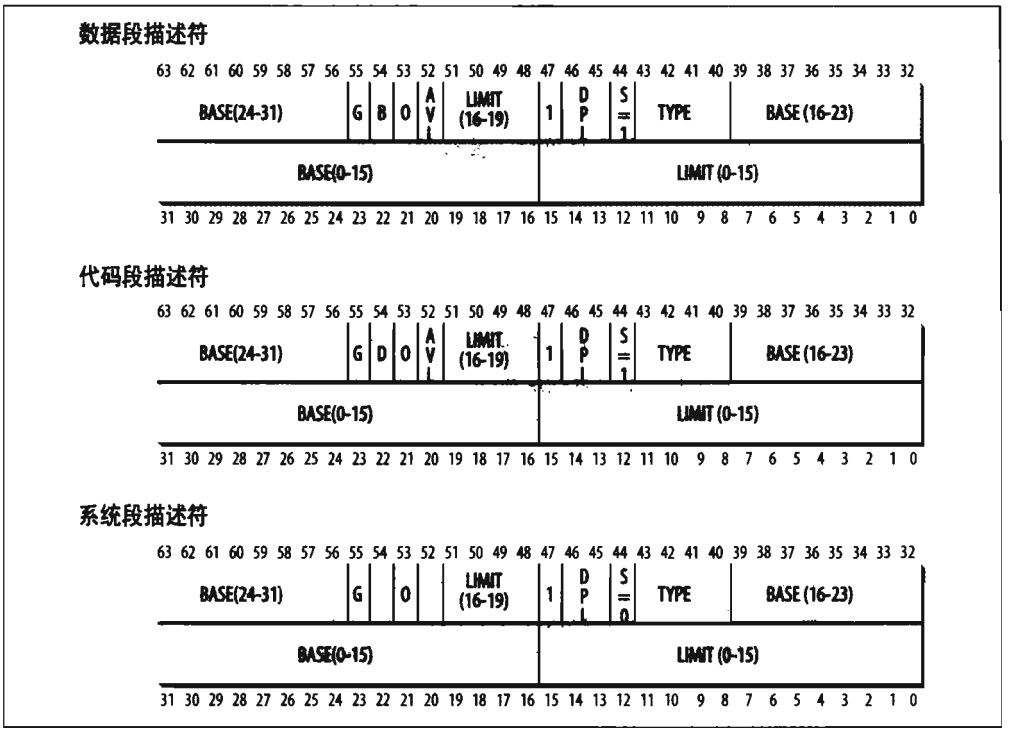


图2-3：段描述符格式

局部描述符表描述符（LDTD）

表示这个段描述符代表一个包含LDT的段，它只出现在GDT中。相应的Type字段的值为2，S标志置为0。下一节说明80x86处理器如何决定一个段描述符是存放在GDT中还是存放在进程的LDT中。

快速访问段描述符

我们回忆一下：逻辑地址由16位段选择符和32位偏移量组成，段寄存器仅仅存放段选择符。

为了加速逻辑地址到线性地址的转换，80x86处理器提供一种附加的非编程的寄存器（一个不能被程序员所设置的寄存器），供6个可编程的段寄存器使用。每一个非编程的寄存器含有8个字节的段描述符（在前一节已讲述），由相应的段寄存器中的段选择符来指定。每当一个段选择符被装入段寄存器时，相应的段描述符就由内存装入到对应的非

编程CPU寄存器。从那时起，针对那个段的逻辑地址转换就可以不访问主存中的GDT或LDT，处理器只需直接引用存放段描述符的CPU寄存器即可。仅当段寄存器的内容改变时，才有必要访问GDT或LDT（参见图2-4）。

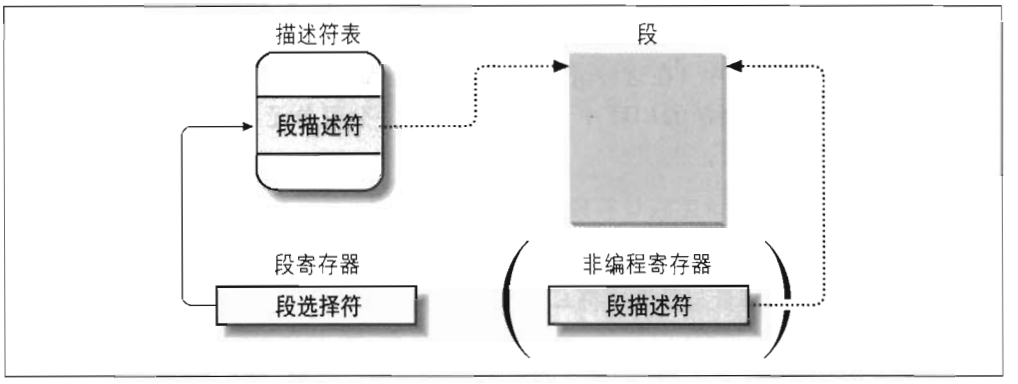


图2-4：段选择符和段描述符

表2-2描述了任意段选择符所包含的3个字段。

表2-2：段选择符字段

字段名	描述
index	指定了放在GDT或LDT中的相应段描述符的入口（在下面将作进一步的讲述）
TI	TI（(*Table Indicator*)标志：指明段描述符是在GDT中（TI=0）或在LDT中（TI=1）
RPL	请求者特权级：当相应的段选择符装入到cs寄存器中时指示出CPU当前的特权级；它还可以用于在访问数据段时有选择地削弱处理器的特权级（详情请参见Intel文档）

由于一个段描述符是8字节长，因此它在GDT或LDT内的相对地址是由段选择符的最高13位的值乘以8得到的。例如：如果GDT在0x00020000（这个值保存在gdtr寄存器中），且由段选择符所指定的索引号为2，那么相应的段描述符地址是0x00020000 + (2 × 8)，或0x00020010。

GDT的第一项总是设为0。这就确保空段选择符的逻辑地址会被认为是无效的，因此引起一个处理器异常。能够保存在GDT中的段描述符的最大数目是8191，即$2^{13}-1$。

分段单元

图2-5详细显示了一个逻辑地址是怎样转换成相应的线性地址的。分段单元(*segmentation unit*)执行以下操作：

- 先检查段选择符的TI字段，以决定段描述符保存在哪一个描述符表中。TI字段指明描述符是在GDT中（在这种情况下，分段单元从gdtr寄存器中得到GDT的线性基地址）还是在激活的LDT中（在这种情况下，分段单元从ldtr寄存器中得到LDT的线性基地址）。
- 从段选择符的index字段计算段描述符的地址，index字段的值乘以8（一个段描述符的大小），这个结果与gdtr或ldtr寄存器中的内容相加。
- 把逻辑地址的偏移量与段描述符Base字段的值相加就得到了线性地址。

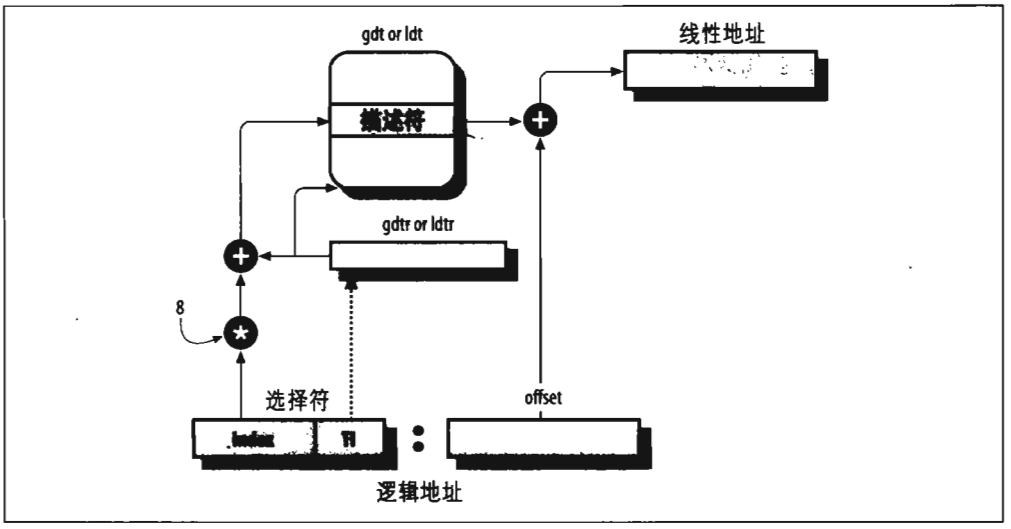


图2-5：逻辑地址的转换

请注意，有了与段寄存器相关的不可编程寄存器，只有当段寄存器的内容被改变时才需要执行前两个操作。

Linux中的分段

80x86微处理器中的分段鼓励程序员把他们的程序化分成逻辑上相关的实体，例如子程序或者全局与局部数据区。然而，Linux以非常有限的方式使用分段。实际上，分段和分页在某种程度上有点多余，因为它们都可以划分进程的物理地址空间：分段可以给每

一个进程分配不同的线性地址空间，而分页可以把同一线性地址空间映射到不同的物理空间。与分段相比，Linux 更喜欢使用分页方式，因为：

- 当所有进程使用相同的段寄存器值时，内存管理变得更简单，也就是说它们能共享同样的一组线性地址。
- Linux设计目标之一是可以把它移植到绝大多数流行的处理器平台上。然而，RISC 体系结构对分段的支持很有限。

2.6 版的 Linux 只有在 80x86 结构下才需要使用分段。

运行在用户态的所有Linux进程都使用一对相同的段来对指令和数据寻址。这两个段就是所谓的用户代码段和用户数据段。类似地，运行在内核态的所有 Linux 进程都使用一对相同的段对指令和数据寻址：它们分别叫做内核代码段和内核数据段。表 2-3 显示了这四个重要段的段描述符字段的值。

表 2-3：四个主要的 Linux 段的段描述符字段的值

段	Base	G	Limit	S	Type	DPL	D/B	P
用户代码段	0x00000000	1	0xfffff	1	10	3	1	1
用户数据段	0x00000000	1	0xfffff	1	2	3	1	1
内核代码段	0x00000000	1	0xfffff	1	10	0	1	1
内核数据段	0x00000000	1	0xfffff	1	2	0	1	1

相应的段选择符由宏 __USER_CS，__USER_DS，__KERNEL_CS，和 __KERNEL_DS 分别定义。例如，为了对内核代码段寻址，内核只需要把 __KERNEL_CS 宏产生的值装进 cs 段寄存器即可。

注意，与段相关的线性地址从 0 开始，达到 2^{32}-1 的寻址限长。这就意味着在用户态或内核态下的所有进程可以使用相同的逻辑地址。

所有段都从 0x00000000 开始，这可以得出另一个重要结论，那就是在 Linux 下逻辑地址与线性地址是一致的，即逻辑地址的偏移量字段的值与相应的线性地址的值总是一致的。

如前所述，CPU 的当前特权级（CPL）反映了进程是在用户态还是内核态，并由存放在 cs 寄存器中的段选择符的 RPL 字段指定。只要当前特权级被改变，一些段寄存器必须相应地更新。例如，当 CPL=3 时（用户态），ds 寄存器必须含有用户数据段的段选择符，而当 CPL=0 时，ds 寄存器必须含有内核数据段的段选择符。

类似的情况也出现在 ss 寄存器中。当 CPL 为 3 时，它必须指向一个用户数据段中的用

户栈，而当CPL为0时，它必须指向内核数据段中的一个内核栈。当从用户态切换到内核态时，Linux总是确保ss寄存器装有内核数据段的段选择符。

当对指向指令或者数据结构的指针进行保存时，内核根本不需要为其设置逻辑地址的段选择符，因为cs寄存器就含有当前的段选择符。例如，当内核调用一个函数时，它执行一条call汇编语言指令，该指令仅指定其逻辑地址的偏移量部分，而段选择符不用设置，它已经隐含在cs寄存器中了。因为“在内核态执行”的段只有一种，叫做代码段，由宏__KERNEL_CS定义，所以只要当CPU切换到内核态时将__KERNEL_CS装载进cs就足够了。同样的道理也适用于指向内核数据结构的指针（隐含地使用ds寄存器）以及指向用户数据结构的指针（内核显式地使用es寄存器）。

除了刚才描述的4个段以外，Linux还使用了其他几个专门的段。我们将在下一节讲述Linux GDT的时候介绍它们。

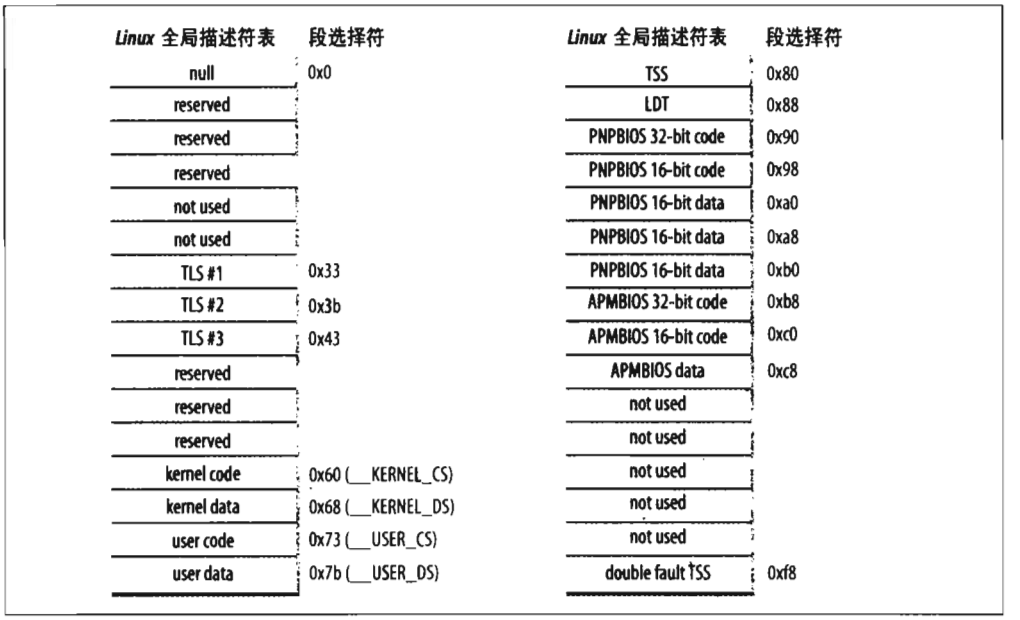
Linux GDT

在单处理器系统中只有一个GDT，而在多处理器系统中每个CPU对应一个GDT。所有的GDT都存放在cpu_gdt_table数组中，而所有GDT的地址和它们的大小（当初始化gdtr寄存器时使用）被存放在cpu_gdt_descr数组中。如果你到源代码索引中查看，可以看到这些符号都在文件 *arch/i386/kernel/head.S* 中被定义。本书中的每一个宏、函数和其他符号都被列在源代码索引中，所以能在源代码中很方便地找到它们。

图2-6是GDT的布局示意图。每个GDT包含18个段描述符和14个空的，未使用的，或保留的项。插入未使用的项的目的是为了使经常一起访问的描述符能够处于同一个32字节的硬件高速缓存行中（参见本章后面“硬件高速缓存”一节）。

每一个GDT中包含的18个段描述符指向下列的段：

- 用户态和内核态下的代码段和数据段共4个（参见前面一节）。
- 任务状态段（TSS），每个处理器有1个。每个TSS相应的线性地址空间都是内核数据段相应线性地址空间的一个小子集。所有的任务状态段都顺序地存放在init_tss数组中；值得特别说明的是，第*n*个CPU的TSS描述符的Base字段指向init_tss数组的第n个元素。G（粒度）标志被清0，而Limit字段置为0xeb，因为TSS段是236字节长。Type字段置为9或11（可用的32位TSS），且DPL置为0，因为不允许用户态下的进程访问TSS段。在第三章“任务状态段”一节你可以找到Linux是如何使用TSS的细节。



Linux 全局描述符表	段选择符
null	0x0
reserved	
reserved	
reserved	
not used	
not used	
TLS #1	0x33
TLS #2	0x3b
TLS #3	0x43
reserved	
reserved	
reserved	
kernel code	0x60 (__KERNEL_CS)
kernel data	0x68 (__KERNEL_DS)
user code	0x73 (__USER_CS)
user data	0x7b (__USER_DS)

Linux 全局描述符表	段选择符
TSS	0x80
LDT	0x88
PNPBIOS 32-bit code	0x90
PNPBIOS 16-bit code	0x98
PNPBIOS 16-bit data	0xa0
PNPBIOS 16-bit data	0xa8
PNPBIOS 16-bit data	0xb0
APMBIOS 32-bit code	0xb8
APMBIOS 16-bit code	0xc0
APMBIOS data	0xc8
not used	
not used	
not used	
not used	
not used	
double fault TSS	0xf8

图 2-6：全局描述符表

- 1个包括缺省局部描述符表的段，这个段通常是被所有进程共享的段（参见下一节）。
- 3个局部线程存储（*Thread-Local Storage*，*TLS*）段：这种机制允许多线程应用程序使用最多3个局部于线程的数据段。系统调用`set_thread_area()`和`get_thread_area()`分别为正在执行的进程创建和撤消一个TLS段。
- 与高级电源管理（AMP）相关的3个段：由于BIOS代码使用段，所以当Linux APM驱动程序调用BIOS函数来获取或者设置APM设备的状态时，就可以使用自定义的代码段和数据段。
- 与支持即插即用（PnP）功能的BIOS服务程序相关的5个段：在前一种情况下，就像前述与AMP相关的3个段的情况一样，由于BIOS例程使用段，所以当Linux的PnP设备驱动程序调用BIOS函数来检测PnP设备使用的资源时，就可以使用自定义的代码段和数据段。
- 被内核用来处理“双重错误”（译注1）异常的特殊TSS段（参见第四章的“异常”一节）。

译注1：处理一个异常时可能会引发另一个异常，在这种情况下产生双重错误。

如前所述，系统中每个处理器都有一个GDT副本。除少数几种情况以外，所有GDT的副本都存放相同的表项。首先，每个处理器都有它自己的TSS段，因此其对应的GDT项不同。其次，GDT中只有少数项可能依赖于CPU正在执行的进程（LDT和TLS段描述符）。最后，在某些情况下，处理器可能临时修改GDT副本里的某个项；例如，当调用APM的BIOS例程时就会发生这种情况。

Linux LDT

大多数用户态下的Linux程序不使用局部描述符表，这样内核就定义了一个缺省的LDT供大多数进程共享。缺省的局部描述符表存放在`default_ldt`数组中。它包含5个项，但内核仅仅有效地使用了其中的两个项：用于iBCS执行文件的调用门和Solaris/x86可执行文件的调用门（参见第二十章的“执行域”一节）。调用门是80x86微处理器提供的一种机制，用于在调用预定义函数时改变CPU的特权级，由于我们不会再更深入地讨论它们，所以请参考Intel文档以获取更多详情。

在某些情况下，进程仍然需要创建自己的局部描述符表。这对有些应用程序很有用，像Wine那样的程序，它们执行面向段的微软Windows应用程序。`modify_ldt()`系统调用允许进程创建自己的局部描述符表。

任何被`modify_ldt()`创建的自定义局部描述符表仍然需要它自己的段。当处理器开始执行拥有自定义局部描述符表的进程时，该CPU的GDT副本中的LDT表项相应地就被修改了。

用户态下的程序同样也利用`modify_ldt()`来分配新的段，但内核却从不使用这些段，它也不需要了解相应的段描述符，因为这些段描述符被包含在进程自定义的局部描述符表中了。

硬件中的分页

分页单元（*paging unit*）把线性地址转换成物理地址。其中的一个关键任务是把所请求的访问类型与线性地址的访问权限相比较，如果这次内存访问是无效的，就产生一个缺页异常（参见第四章和第八章）。

为了效率起见，线性地址被分成以固定长度为单位的组，称为页（*page*）。页内部连续的线性地址被映射到连续的物理地址中。这样，内核可以指定一个页的物理地址和其存取权限，而不用指定页所包含的全部线性地址的存取权限。我们遵循通常习惯，使用术语“页”既指一组线性地址，又指包含在这组地址中的数据。

分页单元把所有的RAM分成固定长度的页框（*page frame*）（有时叫做物理页）。每一个页框包含一个页（*page*），也就是说一个页框的长度与一个页的长度一致。页框是主存的一部分，因此也是一个存储区域。区分一页和一个页框是很重要的，前者只是一个数据块，可以存放在任何页框或磁盘中。

把线性地址映射到物理地址的数据结构称为页表（*page table*）。页表存放在主存中，并在启用分页单元之前必须由内核对页表进行适当的初始化。

从80386开始，所有的80x86处理器都支持分页，它通过设置cr0寄存器的PG标志启用。当PG=0时，线性地址就被解释成物理地址。

常规分页

从80386起，Intel处理器的分页单元处理4KB的页。

32位的线性地址被分成3个域：

Directory（目录）
: 最高10位

Table（页表）
: 中间10位

Offset（偏移量）
: 最低12位

线性地址的转换分两步完成，每一步都基于一种转换表，第一种转换表称为页目录表（*page directory*），第二种转换表称为页表（*page table*）（注1）。

使用这种二级模式的目的在于减少每个进程页表所需RAM的数量。如果使用简单的一级页表，那将需要高达2^{20}个表项（也就是，在每项4个字节时，需要4MB RAM）来表示每个进程的页表（如果进程使用全部4GB线性地址空间），即使一个进程并不使用那个范围内的所有地址。二级模式通过只为进程实际使用的那些虚拟内存区请求页表来减少内存容量。

每个活动进程必须有一个分配给它的页目录。不过，没有必要马上为进程的所有页表都分配RAM。只有在进程实际需要一个页表时才给该页表分配RAM会更为有效率。

注1： 在接下来的讨论中，小写的“page table”表示保存线性地址和物理地址之间映射的页，而利用“Page Table”表示在上层页表中的页。

正在使用的页目录的物理地址存放在控制寄存器 cr3 中。线性地址内的 Directory 字段决定页目录中的目录项，而目录项指向适当的页表。地址的 Table 字段依次又决定页表中的表项，而表项含有页所在页框的物理地址。Offset 字段决定页框内的相对位置（见图 2-7）。由于它是 12 位长，故每一页含有 4096 字节的数据。

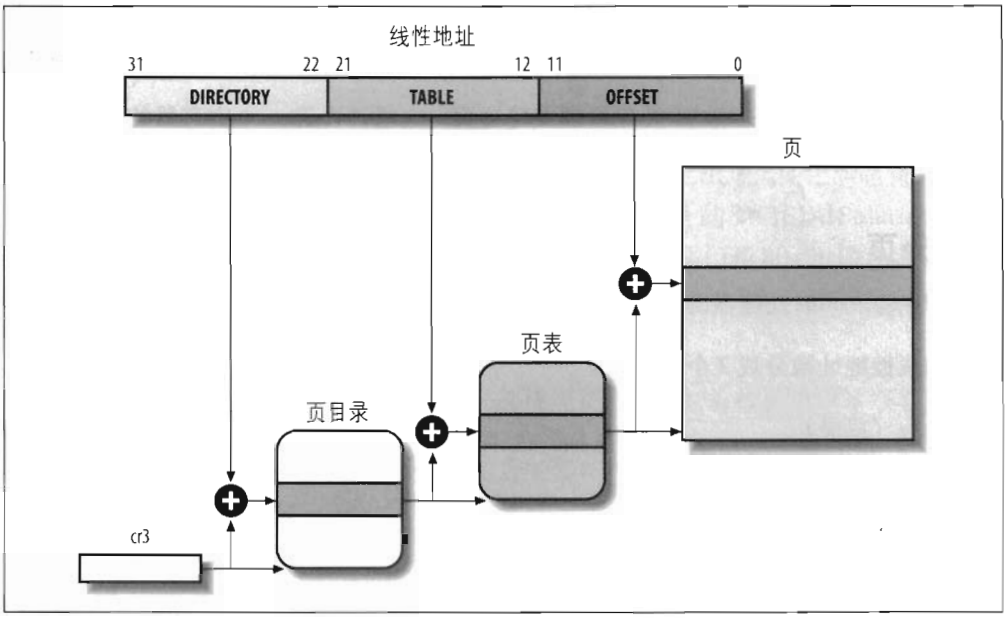


图 2-7：80x86 处理器的分页

Directory 字段和 Table 字段都是 10 位长，因此页目录和页表都可以多达 1024 项。那么一个页目录可以寻址到高达 $1024 \times 1024 \times 4096 = 2^{32}$ 个存储单元，这和你对 32 位地址所期望的一样。

页目录项和页表项有同样的结构，每项都包含下面的字段：

Present 标志

如果被置为 1，所指的页（或页表）就在主存中；如果该标志为 0，则这一页不在主存中，此时这个表项剩余的位可由操作系统用于自己的目的。如果执行一个地址转换所需的页表项或页目录项中 Present 标志被清 0，那么分页单元就把该线性地址存放在控制寄存器 cr2 中，并产生 14 号异常：缺页异常。（我们将在第十七章中看到 Linux 如何使用这个字段。）

包含页框物理地址最高 20 位的字段

由于每一个页框有 4KB 的容量，它的物理地址必须是 4096 的倍数，因此物理地址

的最低12位总是为0。如果这个字段指向一个页目录，相应的页框就含有一个页表；如果它指向一个页表，相应的页框就含有一页数据。

Accessed *标志*

每当分页单元对相应页框进行寻址时就设置这个标志。当选中的页被交换出去时，这一标志就可以由操作系统使用。分页单元从来不重置这个标志，而是必须由操作系统去做。

Dirty *标志*

只应用于页表项中。每当对一个页框进行写操作时就设置这个标志。与Accessed标志一样，当选中的页被交换出去时，这一标志就可以由操作系统使用。分页单元从来不重置这个标志，而是必须由操作系统去做。

Read/Write *标志*

含有页或页表的存取权限（Read/Write或Read）（参阅本章后面“硬件保护方案”一节）。

User/Supervisor *标志*

含有访问页或页表所需的特权级（参见后面的“硬件保护方案”一节）。

PCD *和* PWT *标志*

控制硬件高速缓存处理页或页表的方式（参见本章后面“硬件高速缓存”一节）。

Page Size *标志*

只应用于页目录项。如果设置为1，则页目录项指的是2MB或4MB的页框（参见下一节）。

Global *标志*

只应用于页表项。这个标志是在Pentium Pro中引入的，用来防止常用页从TLB（译注2）高速缓存中刷新出去［参阅本章后面“转换后援缓冲器（TLB）“一节］。只有在cr4寄存器的页全局启用（Page Global Enable，PGE）标志置位时这个标志才起作用。

扩展分页

从Pentium模型开始，80x86微处理器引入了扩展分页（*extended paging*），它允许页框大小为4MB而不是4KB（见图2-8）。扩展分页用于把大段连续的线性地址转换成相应

译注2：TLB的全称为Translation Lookaside Buffer，这是IBM的叫法，有时也叫联想内存（Associative Memory），俗称“快表”。

的物理地址，在这些情况下，内核可以不用中间页表进行地址转换，从而节省内存并保留TLB项［参阅“转换后援缓冲器（LTB）”一节］。

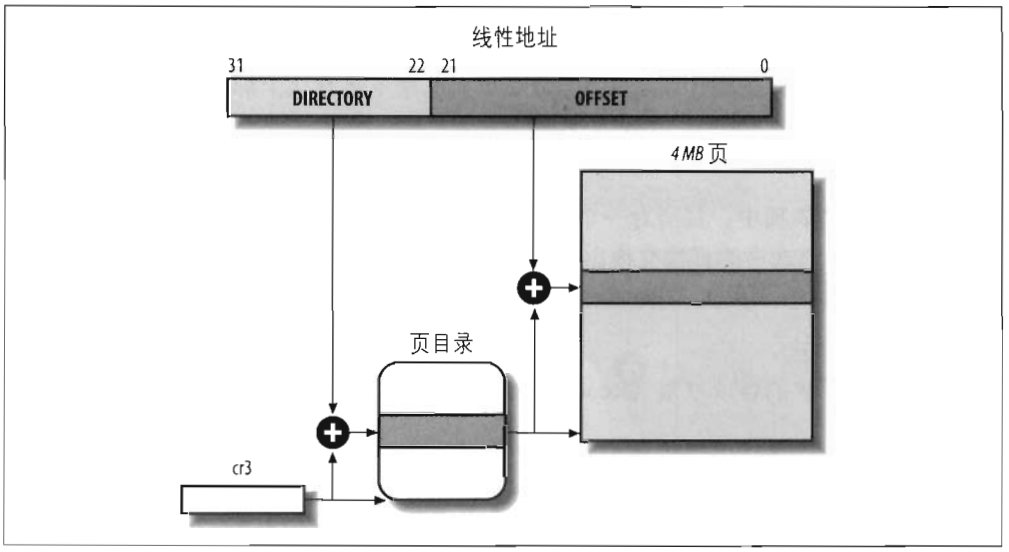


图2-8：扩展分页

正如前面所述，通过设置页目录项的Page Size标志启用扩展分页功能。在这种情况下，分页单元把32位线性地址分成两个字段：

Directory
　　最高10位

Offset
　　其余22位

扩展分页和正常分页的页目录项基本相同，除了：

- Page Size标志必须被设置。
- 20位物理地址字段只有最高10位是有意义的。这是因为每一个物理地址都是在以4MB为边界的地方开始的，故这个地址的最低22位为0。

通过设置cr4处理器寄存器的PSE标志能使扩展分页与常规分页共存。

硬件保护方案

分页单元和分段单元的保护方案不同。尽管80x86处理器允许一个段使用4种可能的特

权级别，但与页和页表相关的特权级只有两个，因为特权由前面“常规分页”一节中所提到的User/Supervisor标志所控制。若这个标志为0，只有当CPL小于3（这意味着对于Linux而言，处理器处于内核态）时才能对页寻址；若该标志为1，则总能对页寻址。

此外，与段的3种存取权限（读，写，执行）不同的是，页的存取权限只有两种（读，写）。如果页目录项或页表项的Read/Write标志等于0，说明相应的页表或页是只读的，否则是可读写的（注2）。

常规分页举例

这个简单的例子将有助于阐明常规分页是如何工作的。我们假定内核已给一个正在运行的进程分配的线性地址空间范围是0x20000000到0x2003ffff（注3）。这个空间正好由64页组成。我们不必关心包含这些页的页框的物理地址，事实上，其中的一些页甚至可能不在主存中。我们只关注页表项中剩余的字段。

让我们从分配给进程的线性地址的最高10位（分页单元解释为Directory字段）开始。这两个地址都以2开头后面跟着0，因此高10位有相同的值，即0x080或十进制的128。因此，这两个地址的Directory字段都指向进程页目录的第129项。相应的目录项中必须包含分配给该进程的页表的物理地址（见图2-9）。如果没有给这个进程分配其它的线性地址，则页目录的其余1023项都填为0。

中间10位的值（即Table字段的值）范围从0到0x03f，或十进制的从0到63。因而只有页表的前64个表项是有意义的，其余960个表项都填0。

假设进程需要读线性地址0x20021406中的字节。这个地址由分页单元按下面的方法处理：

1. Directory字段的0x80用于选择页目录的第0x80目录项，此目录项指向和该进程的页相关的页表。
2. Table字段0x21用于选择页表的第0x21表项，此表项指向包含所需页的页框。
3. 最后，Offet字段0x406用于在目标页框中读偏移量为0x406中的字节。

注2：新的Intel Pentium 4处理器在每个64位页表项中增加了一个NX（No eXecute）标志[必须激活PAE，参见本章后面的“物理地址扩展（PAE）分页机制”一节]。Linux 2.6.11支持这个硬件特性。

注3：正如我们在后面章节所看到的那样，3GB线性地址空间是一个上限，但是用户态进程只允许引用其中的一个子集。

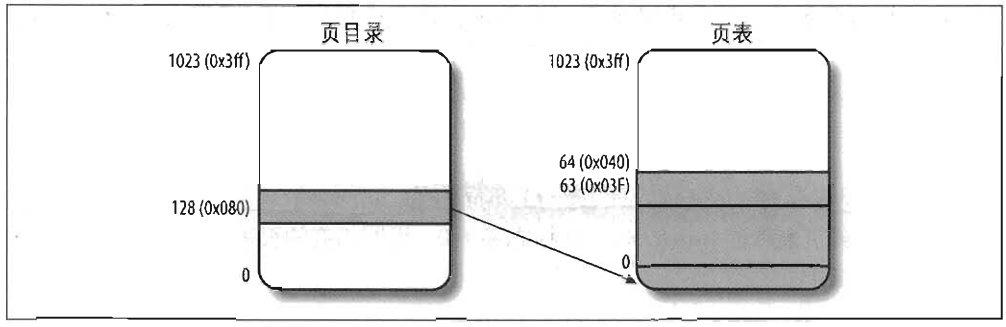


图 2-9：分页的例子

如果页表第 0x21 表项的 Present 标志为 0，则此页就不在主存中；在这种情况下，分页单元在线性地址转换的同时产生一个缺页异常。无论何时，当进程试图访问限定在 0x20000000 到 0x2003ffff 范围之外的线性地址时，都将产生一个缺页异常，因为这些页表项都填充了 0，尤其是它们的 Present 标志都被清 0。

物理地址扩展（PAE）分页机制

处理器所支持的RAM容量受连接到地址总线上的地址管脚数限制。早期Intel处理器从80386到Pentium使用32位物理地址。从理论上讲，这样的系统上可以安装高达4GB的RAM；而实际上，由于用户进程线性地址空间的需要，内核不能直接对1GB以上的RAM进行寻址，我们将会在后面“Linux 中的分页”一节中看到这一点。

然而，大型服务器需要大于4GB的RAM来同时运行数以千计的进程，近几年这对Intel造成了压力，所以必须扩展32位80x86结构所支持的RAM容量。

Intel通过在它的处理器上把管脚数从32增加到36已经满足了这些需求。从Pentium Pro开始，Intel所有处理器现在寻址能力达2^{36} = 64GB。不过，只有引入一种新的分页机制把32位线性地址转换为36位物理地址才能使用所增加的物理地址。

从 Pentium Pro 处理器开始、Intel 引入一种叫做物理地址扩展（*Physical Address Extension*，*PAE*）的机制。另外一种叫做页大小扩展 [Page Size Extension (PSE-36)] 的机制在Pentium III处理器中引入，但是Linux并没有采用这种机制，因而我们在本书中不做进一步讨论。

通过设置 cr4 控制寄存器中的物理地址扩展（PAE）标志激活PAE。页目录项中的页大小标志 PS 启用大尺寸页（在 PAE 启用时为 2MB）。

Intel 为了支持 PAE 已经改变了分页机制。

- 64GB 的 RAM 被分为 2^{24} 个页框，页表项的物理地址字段从 20 位扩展到了 24 位。因为 PAE 页表项必须包含 12 个标志位（在前面“常规分页”一节已描述）和 24 个物理地址位，总数之和为 36，页表项大小从 32 位变为 64 位增加了一倍。结果，一个 4KB 的页表包含 512 个表项而不是 1024 个表项。
- 引入一个叫做页目录指针表（Page Directory Pointer Table，PDPT）的页表新级别，它由 4 个 64 位表项组成。
- `cr3` 控制寄存器包含一个 27 位的页目录指针表（PDPT）基地址字段。因为 PDPT 存放在 RAM 的前 4GB 中，并在 32 字节（2^5）的倍数上对齐，因此 27 位足以表示这种表的基地址。
- 当把线性地址映射到 4 KB 的页时（页目录项中的 PS 标志清 0），32 位线性地址按下列方式解释：

  `cr3`
  : 指向一个 PDPT

  位 31－30
  : 指向 PDPT 中 4 个项中的一个

  位 29－21
  : 指向页目录中 512 个项中的一个

  位 20－12
  : 指向页表中 512 项中的一个

  位 11－0
  : 4KB 页中的偏移量

- 当把线性地址映射到 2MB 的页时（页目录项中的 PS 标志置为 1），32 位线性地址按下列方式解释：

  `cr3`
  : 指向一个 PDPT

  位 31－30
  : 指向 PDPT 中 4 个项中的一个

  位 29－21
  : 指向页目录中 512 个项中的一个

位 20－0

2MB 页中的偏移量

总之，一旦 cr3 被设置，就可能寻址高达 4GB RAM。如果我们希望对更多的 RAM 寻址，就必须在 cr3 中放置一个新值，或改变 PDPT 的内容。然而，使用 PAE 的主要问题是线性地址仍然是32位长。这就迫使内核编程人员用同一线性地址映射不同的RAM区。在后面的“当 RAM 大于 4096MB 时的最终内核页表”一节中，我们将描述启用 PAE 时 Linux 如何初始化页表。很明显，PAE 并没有扩大进程的线性地址空间，因为它只处理物理地址。此外，只有内核能够修改进程的页表，所以在用户态下运行的进程不能使用大于 4GB 的物理地址空间。另一方面，PAE 允许内核使用容量高达 64GB 的 RAM，从而显著增加了系统中的进程数量。

64 位系统中的分页

我们在前面几节已经看到，32位微处理器普遍采用两级分页（注 4）。然而两级分页并不适用于采用 64 位系统的计算机。让我们用一种思维实验来解释为什么：

首先假设一个大小为 4KB 的标准页。因为 1KB 覆盖 2^{10} 个地址的范围，4KB 覆盖 2^{12} 个地址，所以 offset 字段是 12 位。这样线性地址就剩下 52 位分配给 Table 和 Directory 字段。如果我们现在决定仅仅使用 64 位中的 48 位来寻址（这个限制仍然使我们自在地拥有 256TB 的寻址空间！），剩下的 48-12 = 36 位将被分配给 Table 和 Directory 字段。如果我们现在决定为两个字段各预留 18 位，那么每个进程的页目录和页表都含有 2^{18} 个项，即超过 256000 个项。

由于这个原因，所有 64 位处理器的硬件分页系统都使用了额外的分页级别。使用的级别数量取决于处理器的类型。表 2-4 总结了一些 Linux 所支持 64 位平台使用的硬件分页系统的主要特征。对于与平台名称相关的硬件的简要描述请参见第一章的“硬件的依赖性”一节。

表 2-4：一些 64 位系统的分页级别

平台名称	页大小	寻址使用的位数	分页级别数	线性地址分级
alpha	8 KB[a]	43	3	10 + 10 + 10 + 13
ia64	4 KB[a]	39	3	9 + 9 + 9 + 12

注 4：80x86处理器中引入的第三级分页在PAE激活时只是将页目录项和页表项的数目从1024个减少到了 512 个。这样就将页表项从 32 位扩大到了 64 位，于是它们能够存放物理地址的最高 24 位。

表2-4：一些64位系统的分页级别（续）

平台名称	页大小	寻址使用的位数	分页级别数	线性地址分级
ppc64	4 KB	41	3	10 + 10 + 9 + 12
sh64	4 KB	41	3	10 + 10 + 9 + 12
x86_64	4 KB	48	4	9 + 9 + 9 + 9 + 12

a. 该体系结构支持不同的页大小；我们选择了一种Linux支持的典型页大小。

稍后我们将会在本章的“Linux中的分页”一节看到，Linux成功地提供了一种通用分页模型，它适合于绝大多数所支持的硬件分页系统。

硬件高速缓存

当今的微处理器时钟频率接近几个GHz，而动态RAM（DRAM）芯片的存取时间是时钟周期的数百倍。这意味着，当从RAM中取操作数或向RAM中存放结果这样的指令执行时，CPU可能等待很长时间。

为了缩小CPU和RAM之间的速度不匹配，引入了硬件高速缓存内存（hardware cache memory）。硬件高速缓存基于著名的局部性原理（*locality principle*），该原理既适用程序结构和也适用数据结构。这表明由于程序的循环结构及相关数组可以组织成线性数组，最近最常用的相邻地址在最近的将来又被用到的可能性极大。因此，引入小而快的内存来存放最近最常使用的代码和数据变得很有意义。为此，80x86体系结构中引入了一个叫行（*line*）的新单位。行由几十个连续的字节组成，它们以脉冲突发模式（burst mode）在慢速DRAM和快速的用来实现高速缓存的片上静态RAM（SRAM）之间传送，用来实现高速缓存。

高速缓存再被细分为行的子集。在一种极端的情况下，高速缓存可以是直接映射的（*direct mapped*），这时主存中的一个行总是存放在高速缓存中完全相同的位置。在另一种极端情况下，高速缓存是充分关联的（*fully associative*），这意味着主存中的任意一个行可以存放在高速缓存中的任意位置。但是大多数高速缓存在某种程度上是N-路组关联的（*N-way set associative*），意味着主存中的任意一个行可以存放在高速缓存N行中的任意一行中。例如，内存中的一个行可以存放到一个2路组关联高速缓存两个不同的行中。

如图2-10所示，高速缓存单元插在分页单元和主内存之间。它包含一个硬件高速缓存内存（*hardware cache memory*）和一个高速缓存控制器（*cache controller*）。高速缓存内存存放内存中真正的行。高速缓存控制器存放一个表项数组，每个表项对应高速缓存

内存中的一个行。每个表项有一个标签（*tag*）和描述高速缓存行状态的几个标志(*flag*)。这个标签由一些位组成，这些位让高速缓存控制器能够辨别由这个行当前所映射的内存单元。这种内存物理地址通常分为3组：最高几位对应标签，中间几位对应高速缓存控制器的子集索引，最低几位对应行内的偏移量。

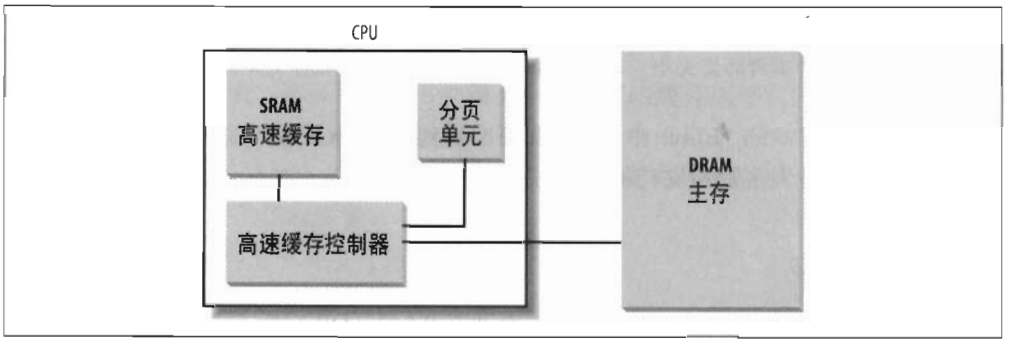


图2-10：处理器硬件高速缓存

当访问一个RAM存储单元时，CPU从物理地址中提取出子集的索引号并把子集中所有行的标签与物理地址的高几位相比较。如果发现某一个行的标签与这个物理地址的高位相同，则CPU命中一个高速缓存（*cache hit*）；否则，高速缓存没有命中（*cache miss*）。

当命中一个高速缓存时，高速缓存控制器进行不同的操作，具体取决于存取类型。对于读操作，控制器从高速缓存行中选择数据并送到CPU寄存器；不需要访问RAM因而节约了CPU时间，因此，高速缓存系统起到了其应有的作用。对于写操作，控制器可能采用以下两个基本策略之一，分别称之为通写（*write-through*）和回写（*write-back*）。在通写中，控制器总是既写RAM也写高速缓存行，为了提高写操作的效率关闭高速缓存。回写方式只更新高速缓存行，不改变RAM的内容，提供了更快的功效。当然，回写结束以后，RAM最终必须被更新。只有当CPU执行一条要求刷新高速缓存表项的指令时，或者当一个FLUSH硬件信号产生时（通常在高速缓存不命中之后），高速缓存控制器才把高速缓存行写回到RAM中。

当高速缓存没有命中时，高速缓存行被写回到内存中，如果有必要的话，把正确的行从RAM中取出放到高速缓存的表项中。

多处理器系统的每一个处理器都有一个单独的硬件高速缓存，因此它们需要额外的硬件电路用于保持高速缓存内容的同步。如图2-11所示，每个CPU都有自己的本地硬件高速缓存。但是，现在更新变得更耗时：只要一个CPU修改了它的硬件高速缓存，它就必须检查同样的数据是否包含在其他的硬件高速缓存中；如果是，它必须通知其他CPU用

适当的值对其更新。常把这种活动叫做高速缓存侦听（*cache snooping*）。值得庆幸的是，所有这一切都在硬件级处理，内核无需关心。

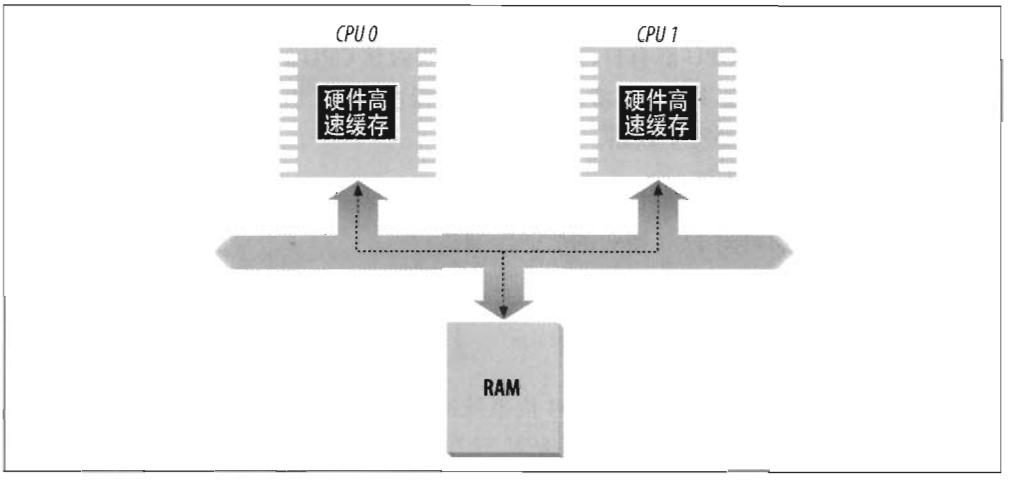


图 2-11：双处理器中的高速缓存

高速缓存技术正在快速向前发展。例如，第一代Pentium芯片包含一颗称为*L1-cache*的片上高速缓存。近期的芯片又包含另外的容量更大、速度较慢，称之为*L2-cache*，*L3-cache*等的片上高速缓存。多级高速缓存之间的一致性是由硬件实现的。Linux忽略这些硬件细节并假定只有一个单独的高速缓存。

处理器的cr0寄存器的CD标志位用来启用或禁用高速缓存电路。这个寄存器中的NW标志指明高速缓存是使用通写还是回写策略。

Pentium处理器高速缓存的另一个有趣的特点是，让操作系统把不同的高速缓存管理策略与每一个页框相关联。为此，每一个页目录项和每一个页表项都包含两个标志：PCD（Page Cache Disablt）标志指明当访问包含在这个页框中的数据时，高速缓存功能必须被启用还是禁用。PWT（page Write-Through）标志指明当把数据写到页框时，必须使用的策略是回写策略还是通写策略。Linux清除了所有页目录项和页表项中的PCD和PWT标志；结果是：对于所有的页框都启用高速缓存，对于写操作总是采用回写策略。

转换后援缓冲器（TLB）

除了通用硬件高速缓存之外，80x86处理器还包含了另一个称为转换后援缓冲器或TLB（*Translation Lookaside Buffer*）的高速缓存用于加快线性地址的转换。当一个线性地

址被第一次使用时，通过慢速访问RAM中的页表计算出相应的物理地址。同时，物理地址被存放在一个TLB表项（TLB entry）中，以便以后对同一个线性地址的引用可以快速地得到转换。

在多处理系统中，每个CPU都有自己的TLB，这叫做该CPU的本地TLB。与硬件高速缓存相反，TLB中的对应项不必同步，这是因为运行在现有CPU上的进程可以使同一线性地址与不同的物理地址发生联系。

当CPU的cr3控制寄存器被修改时，硬件自动使本地TLB中的所有项都无效，这是因为新的一组页表被启用而TLB指向的是旧数据。

Linux中的分页

Linux采用了一种同时适用于32位和64位系统的普通分页模型。正像前面“64位系统中的分页”一节所解释的那样，两级页表对32位系统来说已经足够了，但64位系统需要更多数量的分页级别。直到2.6.10版本，Linux采用三级分页的模型。从2.6.11版本开始，采用了四级分页模型（注5）。图2-12中展示的4种页表分别被为：

- 页全局目录（Page Global Directory）
- 页上级目录（Page Upper Directory）
- 页中间目录（Page Middle Directory）
- 页表（Page Table）

页全局目录包含若干页上级目录的地址，页上级目录又依次包含若干页中间目录的地址，而页中间目录又包含若干页表的地址。每一个页表项指向一个页框。线性地址因此被分成五个部分。图2-12没有显示位数，因为每一部分的大小与具体的计算机体系结构有关。

对于没有启用物理地址扩展的32位系统，两级页表已经足够了。Linux通过使“页上级目录”位和“页中间目录”位全为0，从根本上取消了页上级目录和页中间目录字段。不过，页上级目录和页中间目录在指针序列中的位置被保留，以便同样的代码在32位系统和64位系统下都能使用。内核为页上级目录和页中间目录保留了一个位置，这是通过把它们的页目录项数设置为1，并把这两个目录项映射到页全局目录的一个适当的目录项而实现的。

注5： 这个变化用来全力支持x86_64平台使用的对线性地址的位的划分（参见表2-4）。

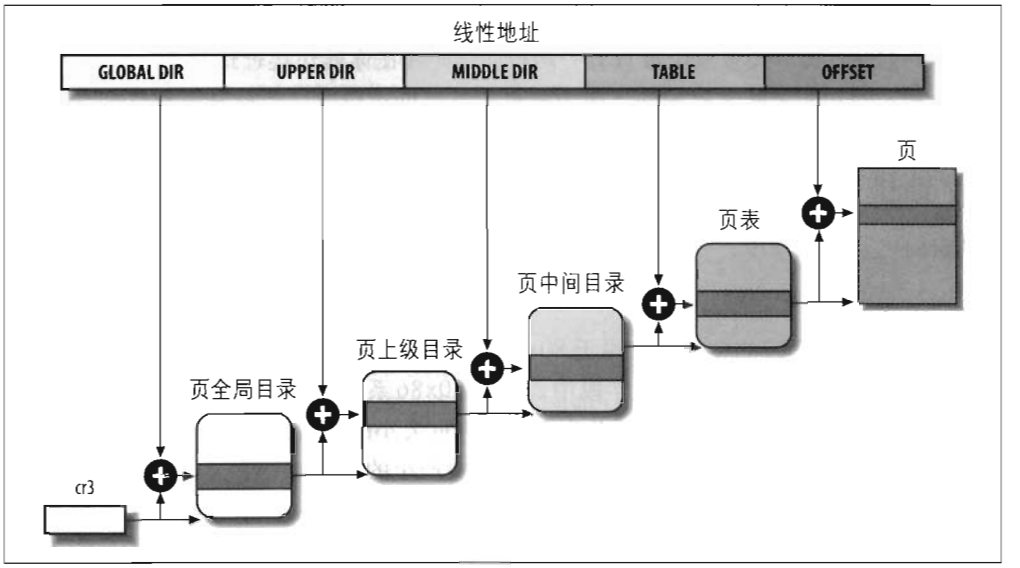


图 2-12：Linux 分页模式

启用了物理地址扩展的32位系统使用了三级页表。Linux的页全局目录对应80x86的页目录指针表（PDPT），取消了页上级目录，页中间目录对应80x86的页目录，Linux的页表对应80x86的页表。

最后，64位系统使用三级还是四级分页取决于硬件对线性地址的位的划分（见表2-4）。

Linux的进程处理很大程度上依赖于分页。事实上，线性地址到物理地址的自动转换使下面的设计目标变得可行：

- 给每一个进程分配一块不同的物理地址空间，这确保了可以有效地防止寻址错误。
- 区别页（即一组数据）和页框（即主存中的物理地址）之不同。这就允许存放在某个页框中的一个页，然后保存到磁盘上，以后重新装入这同一页时又可以被装在不同的页框中。这就是虚拟内存机制的基本要素（参见第十七章）。

在本章剩余的部分，为了具体起见，我们将涉及80x86处理器使用的分页机制。

我们将在第九章看到，每一个进程有它自己的页全局目录和自己的页表集。当发生进程切换时（参见第三章“进程切换”一节），Linux把cr3控制寄存器的内容保存在前一个执行进程的描述符中，然后把下一个要执行进程的描述符的值装入cr3寄存器中。因此，当新进程重新开始在CPU上执行时，分页单元指向一组正确的页表。

把线性地址映射到物理地址虽然有点复杂，但现在已经成了一种机械式的任务。本章下

面的几节中列举了一些比较单调乏味的函数和宏，它们检索内核为了查找地址和管理表格所需的信息；其中大多数函数只有一两行。也许现在你就想跳过这部分，但是知道这些函数和宏的功能是非常有用的，因为在贯穿本书的讨论中你会经常看到它们。

线性地址字段

下列宏简化了页表处理：

PAGE_SHIFT

指定Offset字段的位数；当用于80x86处理器时，它产生的值为12。由于页内所有地址都必须能放到Offset字段中，因此80x86系统的页的大小是2^{12}=4096字节。PAGE_SHIFT的值为12可以看作以2为底的页大小的对数。这个宏由PAGE_SIZE使用以返回页的大小。最后，PAGE_MASK宏产生的值为0xfffff000，用以屏蔽Offset字段的所有位。

PMD_SHIFT

指定线性地址的Offset字段和Table字段的总位数；换句话说，是页中间目录项可以映射的区域大小的对数。PMD_SIZE宏用于计算由页中间目录的一个单独表项所映射的区域大小，也就是一个页表的大小。PMD_MASK宏用于屏蔽Offset字段与Table字段的所有位。

当PAE被禁用时，PMD_SHIFT产生的值为22（来自Offset的12位加上来自Table的10位），PMD_SIZE产生的值为2^{22}或4 MB，PMD_MASK产生的值为0xffc00000。相反，当PAE被激活时，PMD_SHIFT产生的值为21（来自Offset的12位加上来自Table的9位），PMD_SIZE产生的值为2^{21}或2 MB，PMD_MASK产生的值为0xffe00000。

大型页不使用最后一级页表，所以产生大型页尺寸的LARGE_PAGE_SIZE宏等于PMD_SIZE (2PMD_SHIFT)，而在大型页地址中用于屏蔽Offset字段和Table字段的所有位的LARGE_PAGE_MASK宏，就等于PMD_MASK。

PUD_SHIFT

确定页上级目录项能映射的区域大小的对数。PUD_SIZE宏用于计算页全局目录中的一个单独表项所能映射的区域大小。PUD_MASK宏用于屏蔽Offset字段、Table字段、Middle Air字段和Upper Air字段的所有位。

在80x86处理器上，PUD_SHIFT总是等价于PMD_SHIFT，而PUD_SIZE则等于4MB或2MB。

PGDIR_SHIFT

确定页全局目录项能映射的区域大小的对数。PGDIR_SIZE宏用于计算页全局目

录中一个单独表项所能映射区域的大小。PGDIR_MASK宏用于屏蔽Offset、Table、Middle Air及Upper Air字段的所有位。

当PAE被禁止时，PGDIR_SHIFT产生的值为22（与PMD_SHIFT和PUD_SHIFT产生的值相同），PGDIR_SIZE产生的值为2^{22}或4 MB，以及PGDIR_MASK产生的值为0xffc00000。相反，当PAE被激活时，PGDIR_SHIFT产生的值为30（12位Offset加9位Table再加9位Middle Air），PGDIR_SIZE产生的值为2^{30}或1 GB以及PGDIR_MASK产生的值为0xc0000000。

PTRS_PER_PTE，PTRS_PER_PMD，PTRS_PER_PUD以及PTRS_PER_PGD

用于计算页表、页中间目录、页上级目录和页全局目录表中表项的个数。当PAE被禁止时，它们产生的值分别为1024，1，1和1024。当PAE被激活时，产生的值分别为512，512，1和4。

页表处理

pte_t、pmd_t、pud_t和pgd_t分别描述页表项、页中间目录项、页上级目录和页全局目录项的格式。当PAE被激活时它们都是64位的数据类型，否则都是32位数据类型。pgprot_t是另一个64位（PAE激活时）或32位（PAE禁用时）的数据类型，它表示与一个单独表项相关的保护标志。

五个类型转换宏（__pte、__pmd、__pud、__pgd和__pgprot）把一个无符号整数转换成所需的类型。另外的五个类型转换宏（pte_val、pmd_val、pud_val、pgd_val和pgprot_val）执行相反的转换，即把上面提到的四种特殊的类型转换成一个无符号整数。

内核还提供了许多宏和函数用于读或修改页表表项：

- 如果相应的表项值为0，那么，宏pte_none、pmd_none、pud_none和pgd_none产生的值为1，否则产生的值为0。
- 宏pte_clear、pmd_clear、pud_clear和pgd_clear清除相应页表的一个表项，由此禁止进程使用由该页表项映射的线性地址。ptep_get_and_clear()函数清除一个页表项并返回前一个值。
- set_pte、set_pmd、set_pud和set_pgd向一个页表项中写入指定的值。set_pte_atomic与set_pte的作用相同，但是当PAE被激活时它同样能保证64位的值被原子地写入。
- 如果a和b两个页表项指向同一页并且指定相同的访问优先级，那么pte_same(a,b)返回1，否则返回0。

- 如果页中间目录项e指向一个大型页（2MB或4MB），那么pmd_large(e)返回1，否则返回0。

宏pmd_bad由函数使用并通过输入参数传递来检查页中间目录项。如果目录项指向一个不能使用的页表，也就是说，如果至少出现以下条件中的一个，则这个宏产生的值为1：

- 页不在主存中（Present标志被清除）。
- 页只允许读访问(Read/Write标志被清除)。
- Acessed或者Dirty位被清除（对于每个现有的页表，Linux总是强制设置这些标志）。

pud_bad宏和pgd_bad宏总是产生0。没有定义pte_bad宏，因为页表项引用一个不在主存中的页、一个不可写的页或一个根本无法访问的页都是合法的。

如果一个页表项的Present标志或者Page Size标志等于1，则pte_present宏产生的值为1，否则为0。前面讲过页表项的Page Size标志对微处理器的分页单元来讲没有意义，然而，对于当前在主存中却又没有读、写或执行权限的页，内核将其Present和Page Size分别标记为0和1。这样，任何试图对此类页的访问都会引起一个缺页异常，因为页的Present标志被清0，而内核可以通过检查Page Size的值来检测到产生异常并不是因为缺页。

如果相应表项的Present标志等于1，也就是说，如果对应的页或页表被载入主存，pmd_present宏产生的值为1。pud_present宏和pgd_present宏产生的值总是1。

表2-5中列出的函数用来查询页表项中任意一个标志的当前值；除了pte_file()外，其他函数只有在pte_present返回1的时候，才能正常返回页表项中任意一个标志。

表2-5：读页标志的函数

函数名称	说明
pte_user()	读User/Supervisor标志
pte_read()	读User/Supervisor标志（表示80x86处理器上的页不受读的保护）
pte_write()	读Read/Write标志
pte_exec()	读User/Supervisor标志（80x86处理器上的页不受代码执行的保护）
pte_dirty()	读Dirty标志
pte_young()	读Accessed标志
pte_file()	读Dirty标志（当Present标志被清除而Dirty标志被设置时，页属于一个非线性磁盘文件映射，参见第十六章）

表 2-6 列出的另一组函数用于设置页表项中各标志的值。

表 2-6：设置页标志的函数

函数名称	说明
mk_pte_huge()	设置页表项中的 Page Size 和 Present 标志
pte_wrprotect()	清除 Read/Write 标志
pte_rdprotect()	清除 User/Supervisor 标志
pte_exprotect()	清除 User/Supervisor 标志
pte_mkwrite()	设置 Read/Write 标志
pte_mkread()	设置 User/Supervisor 标志
pte_mkexec()	设置 User/Supervisor 标志
pte_mkclean()	清除 Dirty 标志
pte_mkdirty()	设置 Dirty 标志
pte_mkold()	清除 Accessed 标志（把此页标记为未访问）
pte_mkyoung()	设置 Accessed 标志（把此页标记为访问过）
pte_modify(p,v)	把页表项 p 的所有访问权限设置为指定的值 v
ptep_set_wrprotect()	与 pte_wrprotect()类似，但作用于指向页表项的指针
ptep_set_access_flags()	如果 Dirty 标志被设置为 1，则将页的存取权限设置为指定的值，并调用 flush_tlb_page()函数［参见本章“转换后援缓冲器（TLB）”一节］
ptep_mkdirty()	与 pte_mkdirty()类似，但作用于指向页表项的指针
ptep_test_and_clear_dirty()	与 pte_mkclean()类似，但作用于指向页表项的指针并返回 Dirty 标志的旧值
ptep_test_and_clear_young()	与 pte_mkold()类似，但作用于指向页表项的指针并返回 Accessed 标志的旧值

现在，我们来讨论表 2-7 中列出的宏，它们把一个页地址和一组保护标志组合成页表项，或者执行相反的操作，从一个页表项中提取出页地址。请注意这其中的一些宏对页的引用是通过“页描述符”的线性地址（参见第八章“页描述符”一节），而不是通过该页本身的线性地址。

表 2-7：对页表项操作的宏

宏名称	说明
pgd_index(addr)	找到线性地址 addr 对应的目录项在页全局目录中的索引（相对位置）

表2-7：对页表项操作的宏（续）

宏名称	说明
pgd_offset(mm, addr)	接收内存描述符地址mm（参见第九章）和线性地址addr作为参数。这个宏产生地址addr在页全局目录中相应表项的线性地址；通过内存描述符mm内的一个指针可以找到这个页全局目录
pgd_offset_k(addr)	产生主内核页全局目录中的某个项的线性地址，该项对应于地址addr（参见稍后“内核页表”一节）
pgd_page(pgd)	通过页全局目录项pgd产生页上级目录所在页框的页描述符地址。在两级或三级分页系统中，该宏等价于pud_page()，后者应用于页上级目录项
pud_offset(pgd, addr)	接收指向页全局目录项的指针pgd和线性地址addr作为参数。这个宏产生页上级目录中目录项addr对应的线性地址。在两级或三级分页系统中，该宏产生pgd，即一个页全局目录项的地址
pud_page(pud)	通过页上级目录项pud产生相应的页中间目录的线性地址。在两级分页系统中，该宏等价于pmd_page()，后者应用于页中间目录项
pmd_index(addr)	产生线性地址addr在页中间目录中所对应目录项的索引（相对位置）
pmd_offset(pud, addr)	接收指向页上级目录项的指针pud和线性地址addr作为参数。这个宏产生目录项addr在页中间目录中的偏移地址。在两级或三级分页系统中，它产生pud，即页全局目录项的地址
pmd_page(pmd)	通过页中间目录项pmd产生相应页表的页描述符地址。在两级或三级分页系统中，pmd实际上是页全局目录中的一项
mk_pte(p,prot)	接收页描述符地址p和一组存取权限prot作为参数，并创建相应的页表项
pte_index(addr)	产生线性地址addr对应的表项在页表中的索引（相对位置）
pte_offset_kernel(dir,addr)	线性地址addr在页中间目录dir中有一个对应的项，该宏就产生这个对应项，即页表的线性地址。另外，该宏只在主内核页表上使用（参见稍后的“内核页表”一节）

表2-7：对页表项操作的宏（续）

宏名称	说明
pte_offset_map(dir,addr)	接收指向一个页中间目录项的指针dir和线性地址addr作为参数，它产生与线性地址addr相对应的页表项的线性地址。如果页表被保存在高端内存中，那么内核建立一个临时内核映射（参见第八章“高端内存页框的内核映射”一节），并用pte_unmap对它进行释放。pte_offset_map_nested宏和pte_unmap_nested宏是相同的，但它们使用不同的临时内核映射
pte_page(x)	返回页表项x所引用页的描述符地址
pte_to_pgoff(pte)	从一个页表项的pte字段内容中提取出文件偏移量，这个偏移量对应着一个非线性文件内存映射所在的页（参见第十六章“非线性内存映射”一节）
pgoff_to_pte(offset)	为非线性文件内存映射所在的页创建对应页表项的内容

这里罗列最后一组函数来简化页表项的创建和撤消。

当使用两级页表时，创建或删除一个页中间目录项是不重要的。如本节前部分所述，页中间目录仅含有一个指向下属页表的目录项。所以，页中间目录项只是页全局目录中的一项而已。然而当处理页表时，创建一个页表项可能很复杂，因为包含页表项的那个页表可能就不存在。在这样的情况下，有必要分配一个新页框，把它填写为0，并把这个表项加入。

如果PAE被激活，内核使用三级页表。当内核创建一个新的页全局目录时，同时也分配四个相应的页中间目录，只有当父页全局目录被释放时，这四个页中间目录才得以释放。

当使用两级或三级分页时，页上级目录项总是被映射为页全局目录中的一个单独项。

与以往一样，表2-8中列出的函数描述是针对80x86体系结构的。

表2-8：页分配函数

函数名称	说明
pgd_alloc(mm)	分配一个新的页全局目录。如果PAE被激活，它还分配三个对应用户态线性地址的子页中间目录。参数mm（内存描述符的地址）在80x86体系结构上被忽略

表 2-8：页分配函数

函数名称	说明
pgd_free(pgd)	释放页全局目录中地址为 pgd 的项。如果 PAE 被激活，它还将释放用户态线性地址对应的三个页中间目录
pud_alloc(mm,pgd,addr)	在两级或三级分页系统下，这个函数什么也不做：它仅仅返回页全局目录项 pgd 的线性地址
pud_free(x)	在两级或三级分页系统下，这个宏什么也不做
pmd_alloc(mm,pud,addr)	定义这个函数以使普通三级分页系统可以为线性地址 addr 分配一个新的页中间目录。如果 PAE 未被激活，这个函数只是返回输入参数 pud 的值，也就是说，返回页全局目录中目录项的地址。如果 PAE 被激活，该函数返回线性地址 addr 对应的页中间目录项的线性地址。参数 mm 被忽略
pmd_free(x)	该函数什么也不做，因为页中间目录的分配和释放是随同它们的父全局目录一同进行的
pte_alloc_map(mm,pmd,addr)	接收页中间目录项的地址 pmd 和线性地址 addr 作为参数，并返回与 addr 对应的页表项的地址。如果页中间目录项为空，该函数通过调用函数 pte_alloc_one()分配一个新页表。如果分配了一个新页表，addr 对应的项就被创建，同时 User/Supervisor 标志被设置为 1。如果页表被保存在高端内存，则内核建立一个临时内核映射（参见第八章“高端内存页框的内核映射”一节），并用 pte_unmap 对它进行释放
pte_alloc_kernel(mm,pmd,addr)	如果与地址 addr 相关的页中间目录项 pmd 为空，该函数分配一个新页表。然后返回与 addr 相关的页表项的线性地址。该函数仅被主内核页表使用（参见稍后“内核页表”一节）
pte_free(pte)	释放与页描述符指针 pte 相关的页表
pte_free_kernel(pte)	等价于 pte_free()，但由主内核页表使用
clear_page_range(mmu,start,end)	从线性地址 start 到 end 通过反复释放页表和清除页中间目录项来清除进程页表的内容

物理内存布局

在初始化阶段，内核必须建立一个物理地址映射来指定哪些物理地址范围对内核可用而

哪些不可用（或者因为它们映射硬件设备I/O的共享内存，或者因为相应的页框含有BIOS数据）。

内核将下列页框记为保留：

- 在不可用的物理地址范围内的页框。
- 含有内核代码和已初始化的数据结构的页框。

保留页框中的页绝不能被动态分配或交换到磁盘上。

一般来说，Linux内核安装在RAM中从物理地址0x00100000开始的地方，也就是说，从第二个MB开始。所需页框总数依赖于内核的配置方案：典型的配置所得到的内核可以被安装在小于3MB的RAM中。

为什么内核没有安装在RAM第一个MB开始的地方？因为PC体系结构有几个独特的地方必须考虑到。例如：

- 页框0由BIOS使用，存放加电自检（*Power-On Self-Test*，*POST*）期间检查到的系统硬件配置。因此，很多膝上型电脑的BIOS甚至在系统初始化后还将数据写到该页框。
- 物理地址从0x000a0000到 0x000fffff的范围通常留给BIOS例程，并且映射ISA图形卡上的内部内存。这个区域就是所有IBM兼容PC上从640KB到1MB之间著名的洞：物理地址存在但被保留，对应的页框不能由操作系统使用。
- 第一个MB内的其他页框可能由特定计算机模型保留。例如，IBM Thinkpnd把0xa0页框映射到0x9f页框。

在启动过程的早期阶段（参看附录一），内核询问BIOS并了解物理内存的大小。在新近的计算机中，内核也调用BIOS过程建立一组物理地址范围和其对应的内存类型。

随后，内核执行machine_specific_memory_setup()函数，该函数建立物理地址映射（见表2-9中的例子）。当然，如果这张表是可获取的，那是内核在BIOS列表的基础上构建的；否则，内核按保守的缺省设置构建这张表：从0x9f (LOWMEMSIZE()) 到0x100 (HIGH_MEMORY)号的所有页框都标记为保留。

表2-9：BIOS提供的物理地址映射举例

开始	结束	类型
0x00000000	0x0009ffff	Usable
0x000f0000	0x000fffff	Reserved

表2-9：BIOS提供的物理地址映射举例（续）

开始	结束	类型
0x00100000	0x07feffff	Usable
0x07ff0000	0x07ff2fff	ACPI data
0x07ff3000	0x07ffffff	ACPI NVS
0xffff0000	0xffffffff	Reserved

表2-9显示了具有128MB RAM计算机的典型配置。从0x07ff0000到0x07ff2fff的物理地址范围中存有加电自检（POST）阶段由BIOS写入的系统硬件设备信息；在初始化阶段，内核把这些信息拷贝到一个合适的内核数据结构中，然后认为这些页框是可用的。相反，从0x07ff3000到0x07ffffff的物理地址范围被映射到硬件设备的ROM芯片。从0xffff0000开始的物理地址范围标记为保留，因为它由硬件映射到BIOS的ROM芯片（参见附录一）。注意BIOS也许并不提供一些物理地址范围的信息（在上述表中，范围是0x000a0000到0x000effff）。为安全可靠起见，Linux假定这样的范围是不可用的。

内核可能不会见到BIOS报告的所有物理内存：例如，如果未使用PAE支持来编译，即使有更大的物理内存可供使用，内核也只能寻址4GB大小的RAM。setup_memory()函数在machine_specific_memory_setup()执行后被调用：它分析物理内存区域表并初始化一些变量来描述内核的物理内存布局，这些变量如表2-10所示。

表2-10：描述内核物理内存布局的变量

变量名称	说明
num_physpages	最高可用页框的页框号
totalram_pages	可用页框的总数量
min_low_pfn	RAM中在内核映像后第一个可用页框的页框号
max_pfn	最后一个可用页框的页框号
max_low_pfn	被内核直接映射的最后一个页框的页框号（低地址内存）
totalhigh_pages	内核非直接映射的页框的总数（高地址内存）
highstart_pfn	内核非直接映射的第一个页框的页框号
highend_pfn	内核非直接映射的最后一个页框的页框号

为了避免把内核装入一组不连续的页框里，Linux更愿跳过RAM的第一个MB。明确地说，Linux用PC体系结构未保留的页框来动态存放所分配的页。

图 2-13 显示 Linux 怎样填充前 3MB 的 RAM。我们假设内核需要小于 3MB 的 RAM。

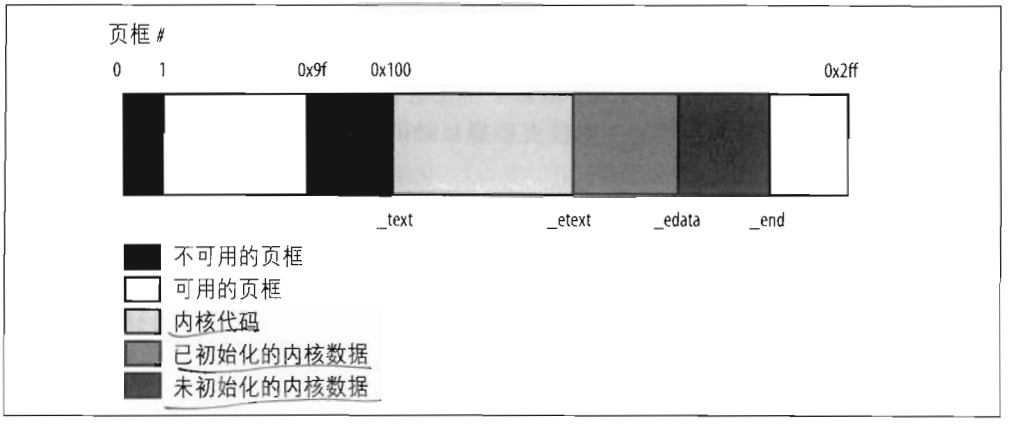


图 2-13：Linux 2.6 的前 768 个页框（3MB）

符号 _text 对应于物理地址 0x00100000，表示内核代码第一个字节的地址。内核代码的结束位置由另外一个类似的符号 _etext 表示。内核数据分为两组：初始化过的数据的和没有初始化的数据。初始化过的数据在 _etext 后开始，在 _edata 处结束。紧接着是未初始化的数据并以 _end 结束。

图中出现的符号并没有在 Linux 源代码中定义，它们是编译内核时产生的（注 6）。

进程页表

进程的线性地址空间分成两部分：

- 从 0x00000000 到 0xbfffffff 的线性地址，无论进程运行在用户态还是内核态都可以寻址。
- 从 0xc0000000 到 0xffffffff 的线性地址，只有内核态的进程才能寻址。

当进程运行在用户态时，它产生的线性地址小于 0xc0000000；当进程运行在内核态时，它执行内核代码，所产生的地址大于等于 0xc0000000。但是，在某些情况下，内核为了检索或存放数据必须访问用户态线性地址空间。

注 6：你可以在 System.map 文件中找到这些符号的线性地址，System.map 是编译内核以后所创建的。

宏PAGE_OFFSET产生的值是0xc0000000，这就是进程在线性地址空间中的偏移量，也是内核生存空间的开始之处。在本书中，我们常常直接引用0xc0000000这个数。

页全局目录的第一部分表项映射的线性地址小于0xc0000000(在PAE未启用时是前768项，PAE启用时是前3项)，具体大小依赖于特定进程。相反，剩余的表项对所有进程来说都应该是相同的，它们等于主内核页全局目录的相应表项(参见下一节)。

内核页表

内核维持着一组自己使用的页表，驻留在所谓的主内核页全局目录(*master kernel Page Global Directory*)中。系统初始化后，这组页表还从未被任何进程或任何内核线程直接使用；更确切地说，主内核页全局目录的最高目录项部分作为参考模型，为系统中每个普通进程对应的页全局目录项提供参考模型。

我们在第八章“非连续内存区的线性地址“一节将会解释，内核如何确保对主内核页全局目录的修改能传递到由进程实际使用的页全局目录中。

我们现在描述内核如何初始化自己的页表。这个过程分为两个阶段。事实上，内核映像刚刚被装入内存后，CPU仍然运行于实模式，所以分页功能没有被启用。

第一个阶段，内核创建一个有限的地址空间，包括内核的代码段和数据段、初始页表和用于存放动态数据结构的共128KB大小的空间。这个最小限度的地址空间仅够将内核装入RAM和对其初始化的核心数据结构。

第二个阶段，内核充分利用剩余的RAM并适当地建立分页表。下一节解释这个方案是怎样实施的。

临时内核页表

临时页全局目录是在内核编译过程中静态地初始化的，而临时页表是由startup_32()汇编语言函数(定义于*arch/i386/kernel/head.S*)初始化的。我们不再过多提及页上级目录和页中间目录，因为它们相当于页全局目录项。在这个阶段PAE支持并未激活。

临时页全局目录放在swapper_pg_dir变量中。临时页表在pg0变量处开始存放，紧接在内核未初始化的数据段(图2-13中的_end符号)后面。为简单起见，我们假设内核使用的段、临时页表和128KB的内存范围能容纳于RAM前8MB空间里。为了映射RAM前8MB的空间，需要用到两个页表。

分页第一个阶段的目标是允许在实模式下和保护模式下都能很容易地对这8MB寻址。因此，内核必须创建一个映射，把从0x00000000到0x007fffff的线性地址和从0xc0000000

到0xc07fffff的线性地址映射到从0x00000000到0x007fffff的物理地址。换句话说，内核在初始化的第一阶段，可以通过与物理地址相同的线性地址或者通过从0xc0000000开始的8MB线性地址对RAM的前8MB进行寻址。

内核通过把swapper_pg_dir所有项都填充为0来创建期望的映射，不过，0、1、0x300（十进制768）和0x301（十进制769）这四项除外；后两项包含了从0xc0000000到0xc07fffff间的所有线性地址。0、1、0x300和0x301按以下方式初始化：

- 0项和0x300项的地址字段置为pg0的物理地址，而1项和0x301项的地址字段置为紧随pg0后的页框的物理地址。
- 把这四个项中的Present、Read/Write和User/Supervisor标志置位。
- 把这四个项中的Accessed、Dirty、PCD、PWD和Page Size标志清0。

汇编语言函数startup_32()也启用分页单元，通过向cr3控制寄存器装入swapper_pg_dir的地址及设置cr0控制寄存器的PG标志来达到这一目的。下面是等价的代码片段：

```
movl $swapper_pg_dir-0xc0000000,%eax
movl %eax,%cr3          /* 设置页表指针……*/
movl %cr0,%eax
orl $0x80000000,%eax
movl %eax,%cr0           /*……设置分页（PG）位*/
```

当RAM小于896MB时的最终内核页表

由内核页表所提供的最终映射必须把从0xc0000000开始的线性地址转化为从0开始的物理地址。

宏__pa用于把从PAGE_OFFSET开始的线性地址转换成相应的物理地址，而宏__va做相反的转化。

主内核页全局目录仍然保存在swapper_pg_dir变量中。它由paging_init()函数初始化。该函数进行如下操作：

1. 调用pagetable_init()适当地建立页表项。
2. 把swapper_pg_dir的物理地址写入cr3控制寄存器中。
3. 如果CPU支持PAE并且如果内核编译时支持PAE，则将cr4控制寄存器的PAE标志置位。
4. 调用__flush_tlb_all()使TLB的所有项无效。

pagetable_init()执行的操作既依赖于现有RAM的容量，也依赖于CPU模型。让

我们从最简单的情况开始。我们的计算机有小于896MB（注7）的RAM，32位物理地址足以对所有可用RAM进行寻址，因而没有必要激活PAE机制［参见前面“物理地址扩展（PAE）分页机制”一节］。

swapper_pg_dir页全局目录由如下等价的循环重新初始化：

```
pgd = swapper_pg_dir + pgd_index(PAGE_OFFSET); /* 768 */
phys_addr = 0x00000000;
while (phys_addr < (max_low_pfn * PAGE_SIZE)) {
    pmd = one_md_table_init(pgd); /*返回pgd*/
    set_pmd(pmd, _ _pmd(phys_addr | pgprot_val(_ _pgprot(0x1e3))));
    /* 0x1e3 == Present, Accessed, Dirty, Read/Write,
                Page Size, Global */
    phys_addr += PTRS_PER_PTE * PAGE_SIZE; /* 0x400000 */
    ++pgd;
}
```

我们假定CPU是支持4MB页和“全局（global）”TLB表项的最新80x86微处理器。注意如果页全局目录项对应的是0xc0000000之上的线性地址，则把所有这些项的User/Supervisor标志清0，由此拒绝用户态进程访问内核地址空间。还要注意Page Size被置位使得内核可以通过使用大型页来对RAM进行寻址（参见本章先前的“扩展分页”一节）。

由startup_32()函数创建的物理内存前8MB的恒等映射用来完成内核的初始化阶段。当这种映射不再必要时，内核调用zap_low_mappings()函数清除对应的页表项。

实际上，这种描述并未说明全部事实。我们将在后面“固定映射的线性地址”一节看到，内核也调整与“固定映射的线性地址”对应的页表项。

当RAM大小在896MB和4096MB之间时的最终内核页表

在这种情况下，并不把RAM全部映射到内核地址空间。Linux在初始化阶段可以做的最好的事是把一个具有896MB的RAM窗口（window）映射到内核线性地址空间。如果一个程序需要对现有RAM的其余部分寻址，那就必须把某些其他的线性地址间隔映射到所需的RAM。这意味着修改某些页表项的值。我们将在第八章讨论这种动态重映射是如何进行的。

内核使用与前一种情况相同的代码来初始化页全局目录。

注7：线性地址的最高128MB留给几种映射去用（参见本章后面“固定映射的线性地址”一节和第八章“非连续内存区的线性地址”一节）。因此映射RAM所剩空间为1GB－128MB＝896MB。

当RAM大于4096MB时的最终内核页表

现在让我们考虑RAM大于4GB计算机的内核页表初始化；更确切地说，我们处理以下发生的情况：

- CPU模型支持物理地址扩展（PAE）
- RAM容量大于4GB
- 内核以PAE支持来编译

尽管PAE处理36位物理地址，但是线性地址依然是32位地址。如前所述，Linux映射一个896 MB的RAM窗口到内核线性地址空间；剩余RAM留着不映射，并由动态重映射来处理，第八章将对此进行描述。与前一种情况的主要差异是使用三级分页模型，因此页全局目录按以下循环代码来初始化：

```
pgd_idx = pgd_index(PAGE_OFFSET); /* 3 */
for (i=0; i<pgd_idx; i++)
    set_pgd(swapper_pg_dir + i, __pgd(__pa(empty_zero_page) + 0x001));
    /* 0x001 == Present */
pgd = swapper_pg_dir + pgd_idx;
phys_addr = 0x00000000;
for (; i<PTRS_PER_PGD; ++i, ++pgd) {
    pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
    set_pgd(pgd, __pgd(__pa(pmd) | 0x001)); /* 0x001 == Present */
    if (phys_addr < max_low_pfn * PAGE_SIZE)
        for (j=0; j < PTRS_PER_PMD /* 512 */
                && phys_addr < max_low_pfn*PAGE_SIZE; ++j) {
            set_pmd(pmd, __pmd(phys_addr |
                               pgprot_val(__pgprot(0x1e3))));
            /* 0x1e3 == Present, Accessed, Dirty, Read/Write,
                        Page Size, Global */
            phys_addr += PTRS_PER_PTE * PAGE_SIZE; /* 0x200000 */
    }
}
swapper_pg_dir[0] = swapper_pg_dir[pgd_idx];
```

页全局目录中的前三项与用户线性地址空间相对应，内核用一个空页(empty_zero_page)的地址对这三项进行初始化。第四项用页中间目录(pmd)的地址初始化，该页中间目录是通过调用alloc_bootmem_low_pages()分配的。页中间目录中的前448项（有512项，但后64项留给非连续内存分配；参见第八章的“非连续内存区管理”一节）用RAM前896MB的物理地址填充。

注意，支持PAE的所有CPU模型也支持大型2MB页和全局页。正如前一种情况一样，只要可能，Linux使用大型页来减少页表数。

然后页全局目录的第四项被拷贝到第一项中，这样好为线性地址空间的前896MB中的低物理内存映射作镜像。为了完成对SMP系统的初始化，这个映射是必需的：当这个映

射不再必要时，内核通过调用zap_low_mappings()函数来清除对应的页表项，正如先前的情况一样。

固定映射的线性地址

我们看到内核线性地址第四个GB的初始部分映射系统的物理内存。但是，至少128MB的线性地址总是留作他用，因为内核使用这些线性地址实现非连续内存分配和固定映射的线性地址。

非连续内存分配仅仅是动态分配和释放内存页的一种特殊方式，将在第八章“非连续内存区的线性地址”一节描述。本节我们集中讨论固定映射的线性地址。

固定映射的线性地址（*fix-mapped linear address*）基本上是一种类似于0xffffc000这样的常量线性地址，其对应的物理地址不必等于线性地址减去0xc000000，而是可以以任意方式建立。因此，每个固定映射的线性地址都映射一个物理内存的页框。我们将会在后面的章节看到，内核使用固定映射的线性地址来代替指针变量，因为这些指针变量的值从不改变。

固定映射的线性地址概念上类似于对RAM前896MB映射的线性地址。不过，固定映射的线性地址可以映射任何物理地址，而由第4GB初始部分的线性地址所建立的映射是线性的（线性地址X映射物理地址*X*-PAGE_OFFSET）。

就指针变量而言，固定映射的线性地址更有效。事实上，间接引用一个指针变量比间接引用一个立即常量地址要多一次内存访问。此外，在间接引用一个指针变量之前对其值进行检查是一个良好的编程习惯；相反，对一个常量线性地址的检查则是没有必要的。

每个固定映射的线性地址都由定义于enum fixed_addresses数据结构中的整型索引来表示：

```
enum fixed_addresses {
    FIX_HOLE,
    FIX_VSYSCALL,
    FIX_APIC_BASE,
    FIX_IO_APIC_BASE_0,
    [...]
    __end_of_fixed_addresses
};
```

每个固定映射的线性地址都存放在线性地址第四个GB的末端。fix_to_virt()函数计算从给定索引开始的常量线性地址：

```
inline unsigned long fix_to_virt(const unsigned int idx)
{
```

```
        if (idx >= _  _end_of_fixed_addresses)
            __this_fixmap_does_not_exist();
        return (0xfffff000UL - (idx << PAGE_SHIFT));
    }
```

让我们假定某个内核函数调用fix_to_virt(FIX_IO_APIC_BASE_0)。因为该函数声明为"inline"，所以C编译程序不调用fix_to_virt()，而是仅仅把它的代码插入到调用函数中。此外，运行时从不对这个索引值执行检查。事实上，FIX_IO_APIC_BASE_0是个等于3的常量，因此编译程序可以去掉if语句，因为它的条件在编译时为假。相反，如果条件为真，或者fix_to_virt()的参数不是一个常量，则编译程序在连接阶段产生一个错误，因为符号_ _this_fixmap_does_not_exist在别处没有定义。最后，编译程序计算0xfffff000-(3<<PAGE_SHIFT)，并用常量线性地址0xffffc000代替fix_to_virt()函数调用。

为了把一个物理地址与固定映射的线性地址关联起来，内核使用set_fixmap(idx,phys)和set_fixmap_nocache(idx,phys)宏。这两个函数都把fix_to_virt(idx)线性地址对应的一个页表项初始化为物理地址phys；不过，第二个函数也把页表项的PCD标志置位，因此，当访问这个页框中的数据时禁用硬件高速缓存（参见本章前面"硬件高速缓存"一节）。反过来，clear_fixmap(idx)用来撤消固定映射线性地址idx和物理地址之间的连接。

处理硬件高速缓存和TLB

内存寻址的最后一个主题是关于内核如何使用硬件高速缓存来达到最佳效果。硬件高速缓存和转换后援缓冲器（TLB）在提高现代计算机体系结构的性能上扮演着重要角色。内核开发者采用一些技术来减少高速缓存和TLB的未命中次数。

处理硬件高速缓存

如前所述，硬件高速缓存是通过高速缓存行（cache line）寻址的。L1_CACHE_BYTES宏产生以字节为单位的高速缓存行的大小。在早于Pentium 4的Intel模型中，这个宏产生的值为32；在Pentium 4上，它产生的值为128。

为了使高速缓存的命中率达到最优化，内核在下列决策中考虑体系结构：

- 一个数据结构中最常使用的字段放在该数据结构内的低偏移部分，以便它们能够处于高速缓存的同一行中。
- 当为一大组数据结构分配空间时，内核试图把它们都存放在内存中，以便所有高速缓存行按同一方式使用。

80x86微处理器自动处理高速缓存的同步，所以应用于这种处理器的Linux内核并不处理任何硬件高速缓存的刷新。不过内核却为不能同步高速缓存的处理器提供了高速缓存刷新接口。

处理TLB

处理器不能自动同步它们自己的TLB高速缓存，因为决定线性地址和物理地址之间映射何时不再有效的是内核，而不是硬件。

Linux 2.6提供了几种在合适时机应当运用的TLB刷新方法，这取决于页表更换的类型（见表2-11）。

表2-11：独立于系统的使TLB表项无效的方法

方法名称	说明	典型的应用时机
flush_tlb_all	刷新所有TLB表项（包括那些全局页对应的TLB表项，即那些Global标志被置位的页）	改变内核页表项时
flush_tlb_kernel_range	刷新给定线性地址范围内的所有TLB表项（包括那些全局页对应的TLB表项）	更换一个范围内的内核页表项时
flush_tlb	刷新当前进程拥有的非全局页相关的所有TLB表项	执行进程切换时
flush_tlb_mm	刷新指定进程拥有的非全局页相关的所有TLB表项	创建一个新的子进程时 释放某个进程的线性
flush_tlb_range	刷新指定进程的线性地址间隔对应的TLB表项	地址间隔时
flush_tlb_pgtables	刷新指定进程中特定的相临页表集相关的TLB表项	释放进程的一些页表时
flush_tlb_page	刷新指定进程中单个页表项相关的TLB表项	处理缺页异常时

尽管普通Linux内核提供了丰富的TLB方法，但通常每个微处理器都提供了更受限制的一组使TLB无效的汇编语言指令。在这个方面，一个更为灵活的硬件平台就是Sun的UltraSPARC。与之相比，Intel微处理器只提供了两种使TLB无效的技术：

- 在向cr3寄存器写入值时所有Pentium处理器自动刷新相对于非全局页的TLB表项。
- 在Pentium Pro及以后的处理器中，invlpg汇编语言指令使映射指定线性地址的单个TLB表项无效。

表2-12列出了采用这种硬件技术的Linux宏；这些宏是实现独立于系统的方法（表2-11）的基本要素。

表2-12：Intel Pentium Pro及以后的处理器上使用的使TLB无效的宏

宏名称	描述	使用对象
__flush_tlb()	将cr3寄存器的当前值重新写回cr3	flush_tlb, flush_tlb_m, flush_tlb_range
__flush_tlb_global()	通过清除cr4的PGE标志禁用全局页，将cr3寄存器的当前值重新写回cr3，并再次设置PGE标志	flush_tlb_all, flush_tlb_kernel_range
__flush_tlb_single(addr)	以addr为参数执行invlpg汇编语言指令	flush_tlb_page

注意表2-12中没有flush_tlb_pgtables方法：在80x86系统中，当页表与父页表解除链接时什么也不需要做，所以实现这个方法的函数为空。

独立于体系结构的使TLB无效的方法非常简单地扩展到了多处理器系统上。在一个CPU上运行的函数发送一个处理器间中断（参见第四章的“处理器间中断处理”）给其他的CPU来强制它们执行适当的函数使TLB无效。

一般来说，任何进程切换都会暗示着更换活动页表集。相对于过期页表，本地TLB表项必须被刷新；这个过程在内核把新的页全局目录的地址写入cr3控制寄存器时会自动完成。不过内核在下列情况下将避免TLB被刷新：

- 当两个使用相同页表集的普通进程之间执行进程切换时（参见第七章的“schedule()函数”一节）。
- 当在一个普通进程和一个内核线程间执行进程切换时。事实上，我们将在第九章的“内核线程的内存描述符”一节看到，内核线程并不拥有自己的页表集；更确切地说，它们使用刚在CPU上执行过的普通进程的页表集。

除了进程切换以外，还有其他几种情况下内核需要刷新TLB中的一些表项。例如，当内核为某个用户态进程分配页框并将它的物理地址存入页表项时，它必须刷新与相应线性地址对应的任何本地TLB表项。在多处理器系统中，如果有多个CPU在使用相同的页表集，那么内核还必须刷新这些CPU上使用相同页表集的TLB表项。

为了避免多处理器系统上无用的TLB刷新，内核使用一种叫做懒惰TLB（*lazy TLB*）模式的技术。其基本思想是，如果几个CPU正在使用相同的页表，而且必须对这些CPU上的一个TLB表项刷新，那么，在某些情况下，正在运行内核线程的那些CPU上的刷新就可以延迟。

事实上，每个内核线程并不拥有自己的页表集；更确切地说，它使用一个普通进程的页表集。不过，没有必要使一个用户态线性地址对应的TLB表项无效，因为内核线程不访问内核态地址空间（注8）。

当某个CPU开始运行一个内核线程时，内核把它置为懒惰TLB模式。当发出清除TLB表项的请求时，处于懒惰TLB模式的每个CPU都不刷新相应的表项；但是，CPU记住它的当前进程正运行在一组页表上，而这组页表的TLB表项对用户态地址是无效的。只要处于懒惰TLB模式的CPU用一个不同的页表集切换到一个普通进程，硬件就自动刷新TLB表项，同时内核把CPU设置为非懒惰TLB模式。然而，如果处于懒惰TLB模式的CPU切换到的进程与刚才运行的内核线程拥有相同的页表集，那么，任何使TLB无效的延迟操作必须由内核有效地实施；这种使TLB无效的“懒惰”操作可以通过刷新CPU的所有非全局TLB项来有效地获取。

为了实现懒惰TLB模式，需要一些额外的数据结构。`cpu_tlbstate`变量是一个具有`NR_CPUS`个结构的静态数组（这个宏的默认值是32，它代表了系统中CPU的最大数量），这个结构有两个字段，一个是指向当前进程内存描述符的`active_mm`字段（参见第九章），一个是具有两个状态值的`state`字段：`TLBSTATE_OK`（非懒惰TLB模式）或`TLBSTATE_LAZY`（懒惰TLB模式）。此外，每个内存描述符中包含一个`cpu_vm_mask`字段，该字段存放的是CPU（这些CPU将要接收与TLB刷新相关的处理器间中断）下标；只有当内存描述符属于当前运行的一个进程时这个字段才有意义。

当一个CPU开始执行内核线程时，内核把该CPU的`cpu_tlbstate`元素的`state`字段置为`TLBSTATE_LAZY`；此外，活动（active）内存描述符的`cpu_vm_mask`字段存放系统中所有CPU（包括进入懒惰TLB模式的CPU）的下标。对于与给定页表集相关的所

注8：顺便说一句，`flush_tlb_all`方法并不使用懒惰TLB模式机制；通常只有在内核修改与内核态地址空间相关的一个页表项时才调用这个方法。

有CPU的TLB表项，当另外一个CPU想使这些表项无效时，该CPU就把一个处理器间中断发送给下标处于对应内存描述符的cpu_vm_mask字段中的那些CPU。

当CPU接受到一个与TLB刷新相关的处理器间中断，并验证它影响了其当前进程的页表集时，它就检查它的cpu_tlbstate元素的state字段是否等于TLBSTATE_LAZY；如果等于，内核就拒绝使TLB表项无效，并从内存描述符的cpu_vm_mask字段删除该CPU下标。这有两种结果：

- 只要CPU还处于懒惰TLB模式，它将不接受其他与TLB刷新相关的处理器间中断。
- 如果CPU切换到另一个进程，而这个进程与刚被替换的内核线程使用相同的页表集，那么内核调用__flush_tlb()使该CPU的所有非全局TLB表项无效。

第三章



进程

进程是任何多道程序设计的操作系统中的基本概念。通常把进程定义为程序执行的一个实例，因此，如果16个用户同时运行*vi*，那么就有16个独立的进程（尽管它们共享同一个可执行代码）。在Linux源代码中，常把进程称为任务（*task*）或线程（*thread*）。

在这一章，我们将首先讨论进程的静态特性，然后描述内核如何进行进程切换。最后两节研究如何创建和撤消进程。这一章还将讲述Linux对多线程应用程序的支持，正如第一章中所提到的，它依赖所谓的轻量级进程（LWP）。

进程、轻量级进程和线程

术语“进程”在使用中常有几个不同的含义。在本书中，我们遵循OS教科书中的通常定义：进程是程序执行时的一个实例。你可以把它看作充分描述程序已经执行到何种程度的数据结构的汇集。

进程类似于人类：它们被产生，有或多或少有效的生命，可以产生一个或多个子进程，最终都要死亡。一个微小的差异是进程之间没有性别差异 —— 每个进程都只有一个父亲。

从内核观点看，进程的目的就是担当分配系统资源（CPU时间、内存等）的实体。

当一个进程创建时，它几乎与父进程相同。它接受父进程地址空间的一个（逻辑）拷贝，并从进程创建系统调用的下一条指令开始执行与父进程相同的代码。尽管父子进程可以共享含有程序代码（正文）的页，但是它们各自有独立的数据拷贝（栈和堆），因此子进程对一个内存单元的修改对父进程是不可见的（反之亦然）。

尽管早期Unix内核使用了这种简单模式，但是，现代Unix系统并没有如此使用。它们支持多线程应用程序——拥有很多相对独立执行流的用户程序共享应用程序的大部分数据结构。在这样的系统中，一个进程由几个用户线程（或简单地说，线程）组成，每个线程都代表进程的一个执行流。现在，大部分多线程应用程序都是用*pthread*（*POSIX thread*）库的标准库函数集编写的。

Linux内核的早期版本没有提供多线程应用的支持。从内核观点看，多线程应用程序仅仅是一个普通进程。多线程应用程序多个执行流的创建、处理、调度整个都是在用户态进行的（通常使用POSIX兼容的*pthread*库）。

但是，这种多线程应用程序的实现方式不那么令人满意。例如，假设一个象棋程序使用两个线程：其中一个控制图形化棋盘，等待人类选手的移动并显示计算机的移动，而另一个思考棋的下一步移动。尽管第一个线程等待选手移动时，第二个线程应当继续运行，以此利用选手的思考时间。但是，如果象棋程序仅是一个单独的进程，第一个线程就不能简单地发出等待用户行为的阻塞系统调用；否则，第二个线程也被阻塞。相反，第一个线程必须使用复杂的非阻塞技术来确保进程仍然是可运行的。

Linux使用轻量级进程（*lightwetght process*）对多线程应用程序提供更好的支持。两个轻量级进程基本上可以共享一些资源，诸如地址空间、打开的文件等等。只要其中一个修改共享资源，另一个就立即查看这种修改。当然，当两个线程访问共享资源时就必须同步它们自己。

实现多线程应用程序的一个简单方式就是把轻量级进程与每个线程关联起来。这样，线程之间就可以通过简单地共享同一内存地址空间、同一打开文件集等来访问相同的应用程序数据结构集；同时，每个线程都可以由内核独立调度，以便一个睡眠的同时另一个仍然是可运行的。POSIX兼容的*pthread*库使用Linux轻量级进程有3个例子，它们是*LinuxThreads*、*Native Posix Thread Library*(*NPTL*)和IBM的下一代Posix线程包NGPT（*Next Generation Posix Threading Package*）。

POSIX兼容的多线程应用程序由支持“线程组”的内核来处理最好不过。在Linux中，一个线程组基本上就是实现了多线程应用的一组轻量级进程，对于像`getpid()`、`kill()`、和`_exit()`这样的一些系统调用，它像一个组织，起整体的作用。在本章随后我们将对其进行详细描述。

进程描述符

为了管理进程，内核必须对每个进程所做的事情进行清楚的描述。例如，内核必须知道进程的优先级，它是正在CPU上运行还是因某些事件而被阻塞，给它分配了什么样的地

址空间，允许它访问哪个文件等等。这正是进程描述符（*process descriptor*）的作用——进程描述符都是task_struct类型结构，它的字段包含了与一个进程相关的所有信息（注1）。因为进程描述符中存放了那么多信息，所以它是相当复杂的。它不仅包含了很多进程属性的字段，而且一些字段还包括了指向其他数据结构的指针，依此类推。图3-1示意性地描述了Linux的进程描述符。

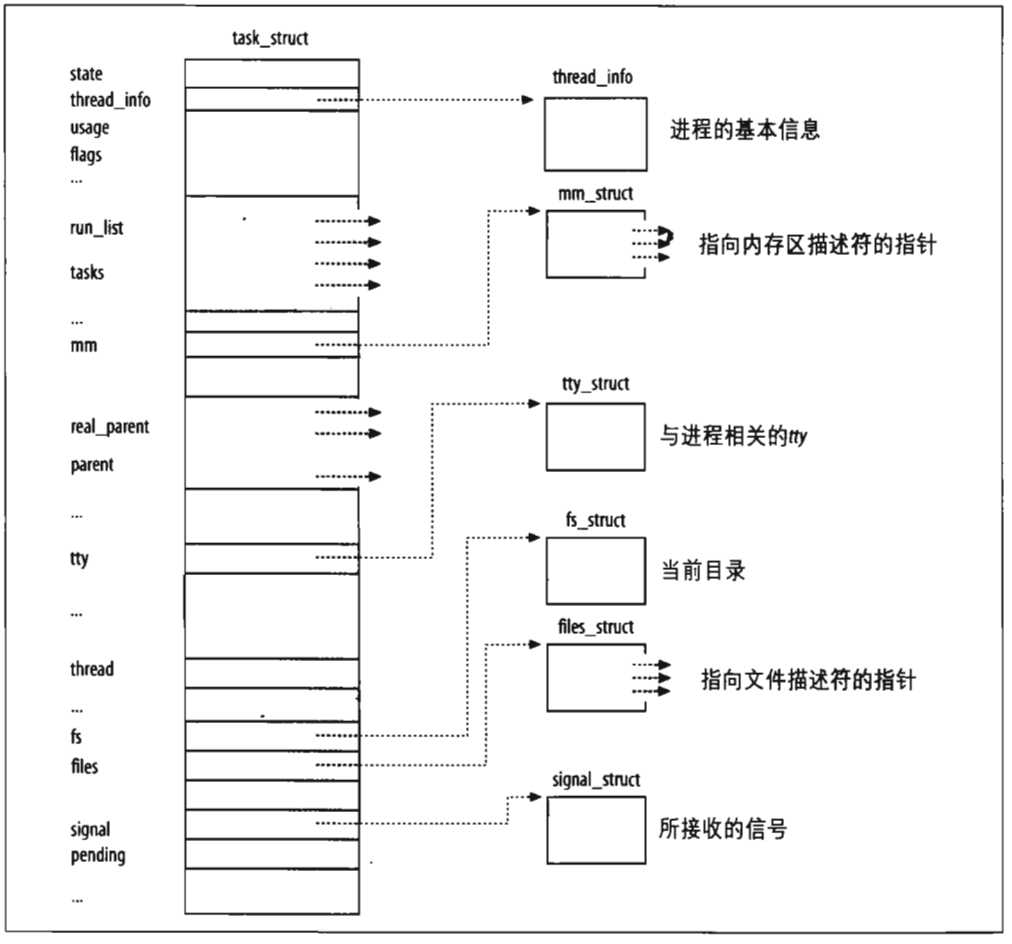


图3-1：Linux进程描述符

图右边的六个数据结构涉及进程所拥有的特殊资源，这些资源将在以后的章节中涉及到。本章集中讨论两种字段：进程的状态和进程的父/子间关系。

注1： 内核还定义了task_t数据类型来等同于struct task_struct。

进程状态

顾名思义，进程描述符中的state字段描述了进程当前所处的状态。它由一组标志组成，其中每个标志描述一种可能的进程状态。在当前的Linux版本中，这些状态是互斥的，因此，严格意义上说，只能设置一种状态；其余的标志将被清除。下面是进程可能的状态：

可运行状态（TASK_RUNNING）

进程要么在CPU上执行，要么准备执行。

可中断的等待状态（TASK_INTERRUPTIBLE）

进程被挂起（睡眠），直到某个条件变为真。产生一个硬件中断，释放进程正等待的系统资源，或传递一个信号都是可以唤醒进程的条件（把进程的状态放回到TASK_RUNNING）。

不可中断的等待状态（TASK_UNINTERRUPTIBLE）

与可中断的等待状态类似，但有一个例外，把信号传递到睡眠进程不能改变它的状态。这种状态很少用到，但在一些特定的情况下（进程必须等待，直到一个不能被中断的事件发生），这种状态是很有用的。例如，当进程打开一个设备文件，其相应的设备驱动程序开始探测相应的硬件设备时会用到这种状态。探测完成以前，设备驱动程序不能被中断，否则，硬件设备会处于不可预知的状态。

暂停状态（TASK_STOPPED）

进程的执行被暂停。当进程接收到SIGSTOP、SIGTSTP、SIGTTIN或SIGTTOU信号后，进入暂停状态。

跟踪状态（TASK_TRACED）

进程的执行已由debugger程序暂停。当一个进程被另一个进程监控时（例如debugger执行ptrace()系统调用监控一个测试程序），任何信号都可以把这个进程置于TASK_TRACED状态。

还有两个进程状态是既可以存放在进程描述符的state字段中，也可以存放在exit_state字段中。从这两个字段的名称可以看出，只有当进程的执行被终止时，进程的状态才会变为这两种状态中的一种：

僵死状态（EXIT_ZOMBIE）

进程的执行被终止，但是，父进程还没有发布wait4()或waitpid()系统调用来返回有关死亡进程的信息（注2）。发布wait()类系统调用前，内核不能丢弃

注2：还有其他wait()类的库函数，如wait3()和wait()，但在Linux中它们是依靠wait4()和waitpid()系统调用来实现的。

包含在死进程描述符中的数据，因为父进程可能还需要它（参见本章结尾的“进程删除”一节）。

僵死撤消状态（EXIT_DEAD）

最终状态：由于父进程刚发出wait4()或waitpid()系统调用，因而进程由系统删除。为了防止其他执行线程在同一个进程上也执行wait()类系统调用（这是一种竞争条件），而把进程的状态由僵死（EXIT_ZOMBIE）状态改为僵死撤消状态（EXIT_DEAD）(参见第五章)。

state字段的值通常用一个简单的赋值语句设置。例如：

```
p->state = TASK_RUNNING;
```

内核也使用set_task_state和set_current_state宏：它们分别设置指定进程的状态和当前执行进程的状态。此外，这些宏确保编译程序或CPU控制单元不把赋值操作与其他指令混合。混合指令的顺序有时会导致灾难性的后果（参见第五章）。

标识一个进程

一般来说，能被独立调度的每个执行上下文都必须拥有它自己的进程描述符；因此，即使共享内核大部分数据结构的轻量级进程，也有它们自己的task_struct结构。

进程和进程描述符之间有非常严格的一一对应关系，这使得用32位进程描述符地址（注3）标识进程成为一种方便的方式。进程描述符指针指向这些地址，内核对进程的大部分引用是通过进程描述符指针进行的。

另一方面，类Unix操作系统允许用户使用一个叫做进程标识符*process ID*（或*PID*）的数来标识进程，PID存放在进程描述符的pid字段中。PID被顺序编号，新创建进程的PID通常是前一个进程的PID加1。不过，PID的值有一个上限，当内核使用的PID达到这个上限值的时候就必须开始循环使用已闲置的小PID号。在缺省情况下，最大的PID号是32767(PID_MAX_DEFAULT-1)；系统管理员可以通过往*/proc/sys/kernel/pid_max*这个文件中写入一个更小的值来减小PID的上限值，使PID的上限小于32767。(*/proc*是一个特殊文件系统的安装点，参看第十二章“特殊文件系统”一节。）在64位体系结构中，系统管理员可以把PID的上限扩大到4194303。

由于循环使用PID编号，内核必须通过管理一个pidmap-array位图来表示当前已分配

注3： 正如已经在第二章的“Linux中的分段”一节中说明的那样，尽管从技术上说，这32位仅仅是一个逻辑地址的偏移量部分，但它们与线性地址相一致。

的PID号和闲置的PID号。因为一个页框包含32768个位，所以在32位体系结构中pidmap-array位图存放在一个单独的页中。然而，在64位体系结构中，当内核分配了超过当前位图大小的PID号时，需要为PID位图增加更多的页。系统会一直保存这些页不被释放。

Linux把不同的PID与系统中每个进程或轻量级进程相关联（本章后面我们会看到，在多处理器系统上稍有例外）。这种方式能提供最大的灵活性，因为系统中每个执行上下文都可以被唯一地识别。

另一方面，Unix程序员希望同一组中的线程有共同的PID。例如，把指定PID的信号发送给组中的所有线程。事实上，POSIX 1003.1c标准规定一个多线程应用程序中的所有线程都必须有相同的PID。

遵照这个标准，Linux引入线程组的表示。一个线程组中的所有线程使用和该线程组的领头线程（*thread group leader*）相同的PID，也就是该组中第一个轻量级进程的PID，它被存入进程描述符的tgid字段中。getpid()系统调用返回当前进程的tgid值而不是pid的值，因此，一个多线程应用的所有线程共享相同的PID。绝大多数进程都属于一个线程组，包含单一的成员；线程组的领头线程其tgid的值与pid的值相同，因而getpid()系统调用对这类进程所起的作用和一般进程是一样的。

下面，我们将向你说明如何从进程的PID中有效地导出它的描述符指针。效率至关重要，因为像kill()这样的很多系统调用使用PID表示所操作的进程。

进程描述符处理

进程是动态实体，其生命周期范围从几毫秒到几个月。因此，内核必须能够同时处理很多进程，并把进程描述符存放在动态内存中，而不是放在永久分配给内核的内存区（译注1）。对每个进程来说，Linux都把两个不同的数据结构紧凑地存放在一个单独为进程分配的存储区域内：一个是内核态的进程堆栈，另一个是紧挨进程描述符的小数据结构thread_info，叫做线程描述符，这块存储区域的大小通常为8192个字节（两个页框）。考虑到效率的因素，内核让这8K空间占据连续的两个页框并让第一个页框的起始地址是2^{13}的倍数。当几乎没有可用的动态内存空间时，就会很难找到这样的两个连续页框，因为空闲空间可能存在大量碎片（见第八章“伙伴系统算法”一节）。因此，在80x86体系结构中，在编译时可以进行设置，以使内核栈和线程描述符跨越一个单独的页框（4096个字节）。

译注1：这里的内存区是指线性地址空间中的一个区域，分配给内核的线性地址空间在3GB之上。

在第二章“Linux 中的分段”一节中我们已经知道，内核态的进程访问处于内核数据段的栈，这个栈不同于用户态的进程所用的栈。因为内核控制路径使用很少的栈，因此只需要几千个字节的内核态堆栈。所以，对栈和thread_info结构来说，8KB 足够了。不过，当使用一个页框存放内核态堆栈和thread_info结构时，内核要采用一些额外的栈以防止中断和异常的深度嵌套而引起的溢出（见第四章）。

图 3-2 显示了在 2 页（8KB）内存区中存放两种数据结构的方式。线程描述符驻留于这个内存区的开始，而栈从末端向下增长。该图还显示了分别通过task和thread_info字段使thread_info结构与task_struct结构互相关联。

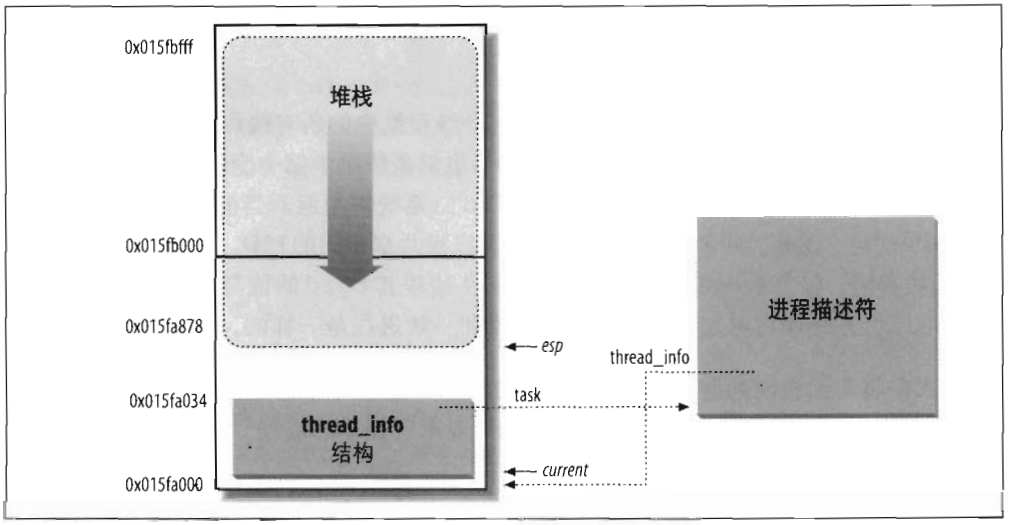


图 3-2：thread_info 结构和进程内核栈存放在两个连续的页框中

esp 寄存器是 CPU 栈指针，用来存放栈顶单元的地址。在 80x86 系统中，栈起始于末端，并朝这个内存区开始的方向增长。从用户态刚切换到内核态以后，进程的内核栈总是空的，因此，esp 寄存器指向这个栈的顶端。

一旦数据写入堆栈，esp 的值就递减。因为thread_info结构是 52 个字节长，因此内核栈能扩展到 8140 个字节。

C 语言使用下列的联合结构方便地表示一个进程的线程描述符和内核栈：

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[2048]; /* 对4K的栈数组下标是1024 */
};
```

如图3-2所示，thread_info结构从0x015fa000地址处开始存放，而栈从0x015fc000地址处开始存放。esp寄存器的值指向地址为0x015fa878的当前栈顶。

内核使用alloc_thread_info和free_thread_info宏分配和释放存储thread_info结构和内核栈的内存区。

标识当前进程

从效率的观点来看，刚才所讲的thread_info结构与内核态堆栈之间的紧密结合提供的主要好处是：内核很容易从esp寄存器的值获得当前在CPU上正在运行进程的thread_info结构的地址。事实上，如果thread_union结构长度是8K（2^{13}字节），则内核屏蔽掉esp的低13位有效位就可以获得thread_info结构的基地址；而如果thread_union结构长度是4K，内核需要屏蔽掉esp的低12位有效位。这项工作由current_thread_info()函数来完成，它产生如下一些汇编指令：

```
movl $0xfffe000, %ecx /* 或者是用于4K堆栈的0xfffff000*/
andl %esp, %ecx
movl %ecx, p
```

这三条指令执行以后，p就包含在执行指令的CPU上运行的进程的thread_info结构的指针。

进程最常用的是进程描述符的地址而不是thread_info结构的地址。为了获得当前在CPU上运行进程的描述符指针，内核要调用current宏，该宏本质上等价于current_thread_info()->task，它产生如下汇编语言指令：

```
movl $0xfffe000,%ecx /* 或者是用于4K堆栈的0xfffff000*/
andl %esp,%ecx
movl (%ecx),p
```

因为task字段在thread_info结构中的偏移量为0，所以执行完这三条指令之后，p就包含在CPU上运行进程的描述符指针。

current宏经常作为进程描述符字段的前缀出现在内核代码中，例如，current->pid返回在CPU上正在执行的进程的PID。

用栈存放进程描述符的另一个优点体现在多处理器系统上：如前所述，对于每个硬件处理器，仅通过检查栈就可以获得当前正确的进程。早先的Linux版本没有把内核栈与进程描述符存放在一起，而是强制引入全局静态变量current来标识正在运行进程的描述符。在多处理器系统上，有必要把current定义为一个数组，每一个元素对应一个可用CPU。

双向链表

在继续阐述内核跟踪系统中各种进程的细节之前，先着重说明实现双向链表的特殊数据结构的作用。

对每个链表，必须实现一组原语操作：初始化链表，插入和删除一个元素，扫描链表等等。这可能既浪费开发人员的精力，也因为对每个不同的链表都要重复相同的原语操作而造成存储空间的浪费。

因此，Linux 内核定义了 list_head 数据结构，字段 next 和 prev 分别表示通用双向链表向前和向后的指针元素。不过，值得特别关注的是，list_head 字段的指针中存放的是另一个 list_head 字段的地址，而不是含有 list_head 结构的整个数据结构地址[参见图 3-3 (a)]。

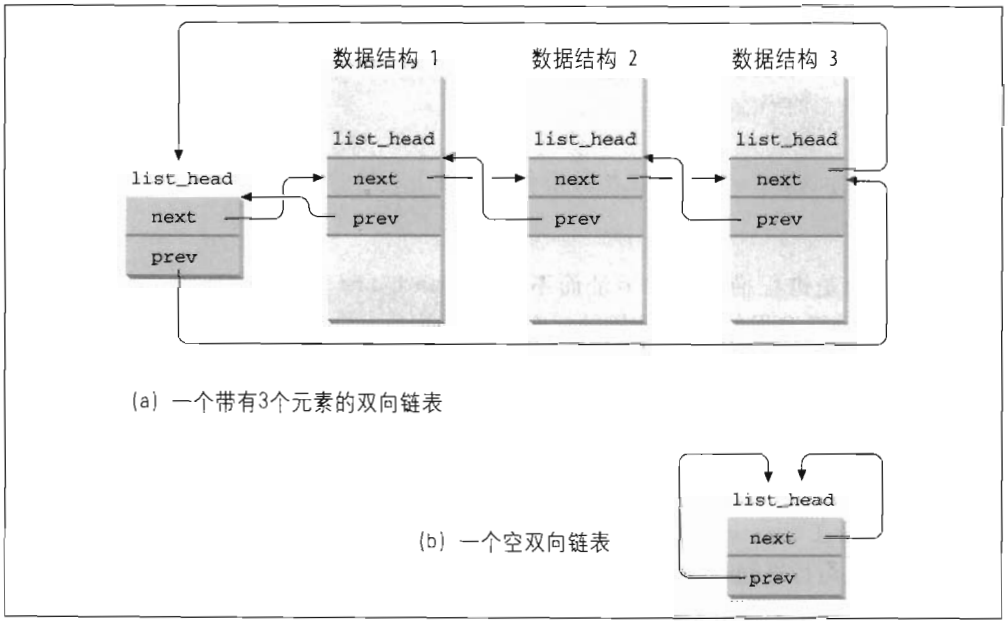


图 3-3：用 list_head 数据结构构造的一个双向链表

新链表是用 LIST_HEAD(list_name)宏创建的。它申明类型为 list_head 的新变量 list_name，该变量作为新链表头的占位符，是一个哑元素。LIST_HEAD(list_name)宏还初始化 list_head 数据结构的 prev 和 next 字段，让它们指向 list_name 变量本身。见图 3-3 (b)。

有几个实现原语的函数和宏，如表 3-1 所示。

表 3-1：处理函数和宏

名称	说明
list_add(n,p)	把 n 指向的元素插入 p 所指向的特定元素之后（为了把 n 插入在链表的开始，就设置 p 为第一个元素的地址）
list_add_tail(n,p)	把 n 指向的元素插到 p 所指向的特定元素之前（为了把 n 插入在链表的尾部，就设置 p 为第一个元素的地址）
list_del(p)	删除 p 所指向的元素（没有必要指定链表的第一个元素）
list_empty(p)	检查由第一个元素的地址 p 指定的链表是否为空
list_entry(p,t,m)	返回类型为 t 的数据结构的地址，其中类型 t 中含有 list_head 字段，而 list_head 字段中含有名字 m 和地址 p
list_for_each(p,h)	对表头地址 h 指定的链表进行扫描，在每次循环时，通过 p 返回指向链表元素的 list_head 结构的指针
list_for_each_entry(p,h,m)	与 list_for_each 类似，但是返回包含了 list_head 结构的数据结构的地址，而不是 list_head 结构本身的地址

Linux 2.6 内核支持另一种双向链表，其与 list_head 有着明显的区别，因为它不是循环链表，主要用于散列表，对散列表而言重要的是空间而不是在固定的时间内找到表中的最后一个元素。表头存放在 hlist_head 数据结构中，该结构只不过是指向表的第一个元素的指针（如果链表为空，那么这个指针为 NULL）。每个元素都是 hlist_node 类型的数据结构，它的 next 指针指向下一个元素，pprev 指针指向前一个元素的 next 字段。因为不是循环链表，所以第一个元素的 pprev 字段和最后一个元素的 next 字段都置为 NULL。对这种表可以用类似表 3-1 中的函数和宏（hlist_add_head(), hlist_del(), hlist_empty(), hlist_entry, hlist_for_each_entry）来操纵。

进程链表

我们首先介绍双向链表的第一个例子 —— 进程链表，进程链表把所有进程的描述符链接起来。每个 task_struct 结构都包含一个 list_head 类型的 tasks 字段，这个类型的 prev 和 next 字段分别指向前面和后面的 task_struct 元素。

进程链表的头是 init_task 描述符，它是所谓的 0 进程（*process 0*）或 *swapper* 进程的进程描述符（参见本章“内核线程”一节）。init_task 的 tasks.prev 字段指向链表中最后插入的进程描述符的 tasks 字段。

SET_LINKS和REMOVE_LINKS宏分别用于从进程链表中插入和删除一个进程描述符。这些宏考虑了进程间的父子关系（见本章后面“如何组织进程”一节）。

还有一个很有用的宏就是for_each_process，它的功能是扫描整个进程链表，其定义如下：

```
#define for_each_process(p) \
   for (p=&init_task; (p=list_entry((p)->tasks.next, \
                                    struct task_struct, tasks) \
                                  ) != &init_task; )
```

这个宏是循环控制语句，内核开发者利用它提供循环。注意init_task进程描述符是如何起到链表头作用的。这个宏从指向init_task的指针开始，把指针移到下一个任务，然后继续，直到又到init_task为止（感谢链表的循环性）。在每一次循环时，传递给这个宏的参变量中存放的是当前被扫描进程描述符的地址，这与list_entry宏的返回值一样。

TASK_RUNNING状态的进程链表

当内核寻找一个新进程在CPU上运行时，必须只考虑可运行进程（即处在TASK_RUNNING状态的进程）。

早先的Linux版本把所有的可运行进程都放在同一个叫做运行队列（*runqueue*）的链表中，由于维持链表中的进程按优先级排序开销过大，因此，早期的调度程序不得不为选择“最佳”可运行进程而扫描整个队列。

Linux 2.6实现的运行队列有所不同。其目的是让调度程序能在固定的时间内选出“最佳”可运行进程，与队列中可运行的进程数无关。我们仅在此提供一些基本信息，第七章会详细描述这种新的运行队列。

提高调度程序运行速度的诀窍是建立多个可运行进程链表，每种进程优先权对应一个不同的链表。每个task_struct描述符包含一个list_head类型的字段run_list。如果进程的优先权等于*k*（其取值范围是0到139），run_list字段把该进程链入优先权为k的可运行进程的链表中。此外，在多处理器系统中，每个CPU都有它自己的运行队列，即它自己的进程链表集。这是一个通过使数据结构更复杂来改善性能的典型例子：调度程序的操作效率的确更高了，但运行队列的链表却为此而被拆分成140个不同的队列！

正如我们将看到的，内核必须为系统中每个运行队列保存大量的数据，不过运行队列的主要数据结构还是组成运行队列的进程描述符链表，所有这些链表都由一个单独的prio_array_t数据结构来实现，其字段说明如表3-2所示。

表3-2：prio_array_t数据结构的字段

类型	字段	描述
int	nr_active	链表中进程描述符的数量
unsigned long [5]	bitmap	优先权位图：当且仅当某个优先权的进程链表不为空时设置相应的位标志
struct list_head [140]	queue	140个优先权队列的头结点

enqueue_task(p,array)函数把进程描述符插入某个运行队列的链表，其代码本质上等同于：

```
list_add_tail(&p->run_list, &array->queue[p->prio]);
__set_bit(p->prio, array->bitmap);
array->nr_active++;
p->array = array;
```

进程描述符的prio字段存放进程的动态优先权，而array字段是一个指针，指向当前运行队列的prio_array_t数据结构。类似地，dequeue_task(p,array)函数从运行队列的链表中删除一个进程的描述符。

进程间的关系

程序创建的进程具有父/子关系。如果一个进程创建多个子进程时，则子进程之间具有兄弟关系。在进程描述符中引入几个字段来表示这些关系，表示给定进程P的这些字段列在表3-3中。进程0和进程1是由内核创建的；稍后我们将看到，进程1（*init*）是所有进程的祖先。

表3-3：进程描述符中表示进程亲属关系的字段的描述

字段名	说明
real_parent	指向创建了P的进程的描述符，如果P的父进程不再存在，就指向进程1（*init*）的描述符（因此，如果用户运行一个后台进程而且退出了shell，后台进程就会成为*init*的子进程）
parent	指向P的当前父进程(这种进程的子进程终止时，必须向父进程发信号)。它的值通常与real_parent一致，但偶尔也可以不同，例如，当另一个进程发出监控P的ptrace()系统调用请求时（参见第二十章中“执行跟踪”一节）
children	链表的头部，链表中的所有元素都是P创建的子进程
sibling	指向兄弟进程链表中的下一个元素或前一个元素的指针，这些兄弟进程的父进程都是P

图 3-4 显示了一组进程间的亲属关系。进程 P0 接连创建了 P1， P2， 和 P3。进程 P3 又创建了 P4。

特别要说明的是，进程之间还存在其他关系：一个进程可能是一个进程组或登录会话的领头进程（参见第一章“进程管理”一节），也可能是一个线程组的领头进程（参见本章前面“标识一个进程”一节），它还可能跟踪其他进程的执行（参见第二十章“执行跟踪”一节）。表 3-4 列出了进程描述符中的一些字段，这些字段建立起了进程 P 和其他进程之间的关系。

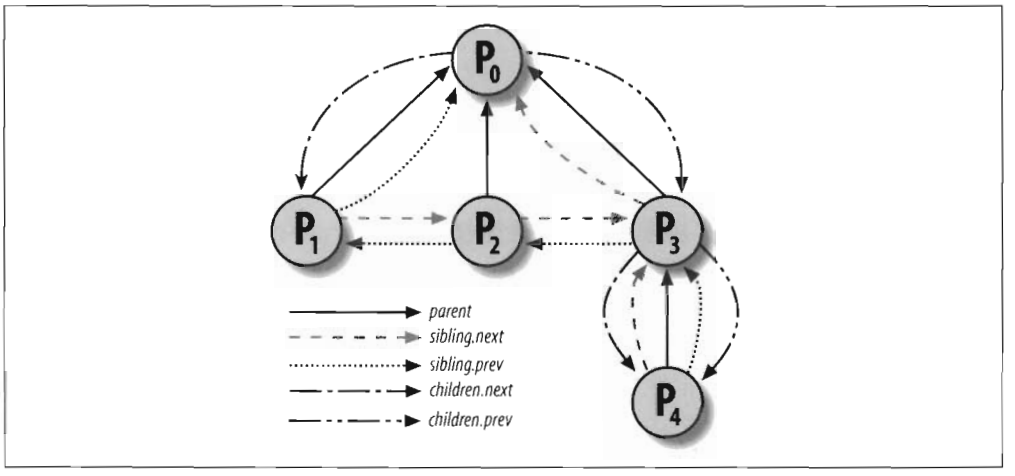


图 3-4：五个进程间的亲属关系

表 3-4：建立非亲属关系的进程描述符字段

字段名	说明
group_leader	P 所在进程组的领头进程的描述符指针
signal->pgrp	P 所在进程组的领头进程的 PID
tgid	P 所在线程组的领头进程的 PID
signal->session	P 的登录会话领头进程的 PID
ptrace_children	链表的头，该链表包含所有被 debugger 程序跟踪的 P 的子进程
ptrace_list	指向所跟踪进程其实际父进程链表的前一个和下一个元素（用于 P 被跟踪的时候）

pidhash 表及链表

在几种情况下，内核必须能从进程的PID导出对应的进程描述符指针。例如，为kill()

系统调用提供服务时就会发生这种情况：当进程P1希望向另一个进程P2发送一个信号时，P1调用kill()系统调用，其参数为P2的PID，内核从这个PID导出其对应的进程描述符，然后从P2的进程描述符中取出记录挂起信号的数据结构指针。

顺序扫描进程链表并检查进程描述符的pid字段是可行但相当低效的。为了加速查找，引入了4个散列表。需要4个散列表是因为进程描述符包含了表示不同类型PID的字段（见表3-5），而且每种类型的PID需要它自己的散列表。

表3-5：4个散列表和进程描述符中的相关字段

Hash表的类型	字段名	说明
PIDTYPE_PID	pid	进程的PID
PIDTYPE_TGID	tgid	线程组领头进程的PID
PIDTYPE_PGID	pgrp	进程组领头进程的PID
PIDTYPE_SID	session	会话领头进程的PID

内核初始化期间动态地为4个散列表分配空间，并把它们的地址存入pid_hash数组。一个散列表的长度依赖于可用RAM的容量，例如：一个系统拥有512 MB的RAM，那么每个散列表就被存在4个页框中，可以拥有2048个表项。

用pid_hashfn宏把PID转化为表索引，pid_hashfn宏展开为：

```
#define pid_hashfn(x) hash_long((unsigned long) x, pidhash_shift)
```

变量pidhash_shift用来存放表索引的长度（以位为单位的长度，在我们的例子里是11位）。很多散列函数都使用hash_long()，在32位体系结构中它基本等价于：

```
unsigned long hash_long(unsigned long val, unsigned int bits)
{
    unsigned long hash = val * 0x9e370001UL;
    return hash >> (32 - bits);
}
```

因为在我们的例子中pidhash_shift等于11，所以pid_hashfn的取值范围是0到$2^{11} - 1 = 2047$。

正如计算机科学的基础课程所阐述的那样，散列（hash）函数并不总能确保PID与表的索引一一对应。两个不同的PID散列（hash）到相同的表索引称为冲突（*colliding*）。

魔数常量

也许你会想常量0x9e370001(= 2 654 404 609)究竟是怎么得出的。这种散列函数是基于表索引乘以一个适当的大数，于是结果溢出，就把留在32位变量中的值作为模数操作的结果。Knuth建议，要得到满意的结果，这个大乘数就应当是接近黄金比例的2^{32}的一个素数（32位是80x86寄存器的大小）。这里，2 654 404 609就是接近$2^{32} \times (\sqrt{5}-1)/2$的一个素数，这个数可以方便地通过加运算和位移运算得到，因为它等于：$2^{31}+2^{29}-2^{25}+2^{22}-2^{19}-2^{16}+1$。

Linux利用链表来处理冲突的PID：每一个表项是由冲突的进程描述符组成的双向链表。图3-5显示了具有两个链表的PID散列表。进程号（PID）为2 890和29 384的两个进程散列到这个表的第200个元素，而进程号（PID）为29 385的进程散列到这个表的第1 466个元素。

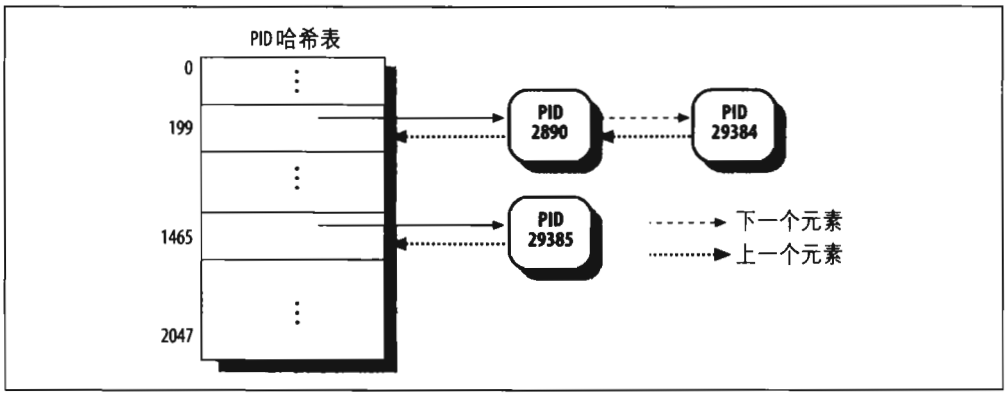


图3-5：pidhash表及链表

具有链表的散列法比从PID到表索引的线性转换更优越，这是因为在任何给定的实例中，系统中的进程数总是远远小于32 768（所允许的进程PID的最大数）。如果在任何给定的实例中大部分表项都不使用的话，那么把表定义为32 768项会是一种存储浪费。

由于需要跟踪进程间的关系，PID散列表中使用的数据结构非常复杂。看一个例子：假设内核必须回收一个指定线程组中的所有进程，这意味着这些进程的`tgid`的值是相同的，都等于一个给定值。如果根据线程组号查找散列表，只能返回一个进程描述符，就是线程组领头进程的描述符。为了能快速返回组中其他所有进程，内核就必须为每个线程组保留一个进程链表。在查找给定登录会话或进程组的进程时也会有同样的情形。

PID散列表的数据结构解决了所有这些难题，因为它们可以为包含在一个散列表中的任

何PID号定义进程链表。最主要的数据结构是四个pid结构的数组，它在进程描述符的pid字段中，表3-6显示了pid结构的字段。

表3-6：pid结构的字段

类型	名称	描述
int	nr	pid的数值
struct hlist_node	pid_chain	链接散列链表的下一个和前一个元素
struct list_head	pid_list	每个pid的进程链表头

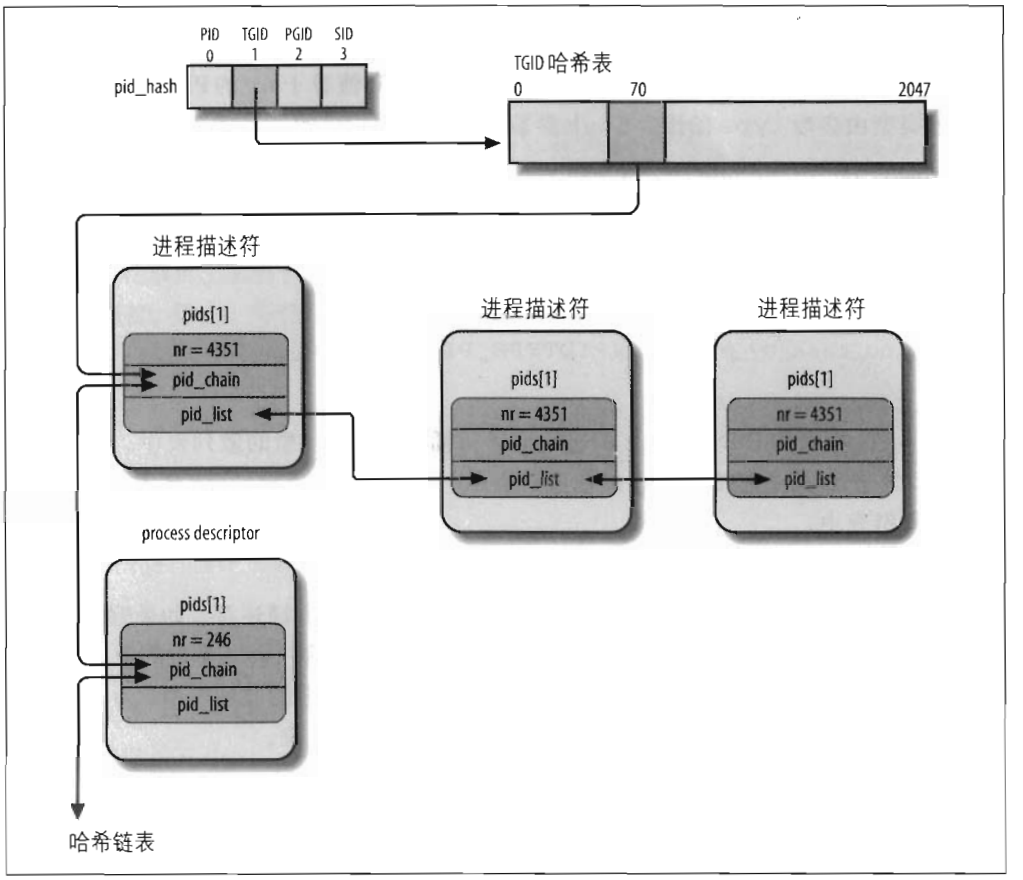


图3-6：PID散列表

图3-6给出了PIDTYPE_TGID类型散列表的例子。pid_hash数组的第二个元素存放

散列表的地址，也就是用hlist_head结构的数组表示链表的头。在散列表第71项为起点形成的链表中，有两个PID号为246和4351的进程描述符（双箭头线表示一对向前和向后的指针）。PID的值存放在pid结构的nr字段中，而pid结构在进程描述符中。（顺便提一下，由于线程组的号和它的首创者的PID相同，因此这些PID值也存在进程描述符的pid字段中。）我们考虑线程组4351的PID链表：散列表中的进程描述符的pid_list字段中存放链表的头，同时每个PID链表中指向前一个元素和后一个元素的指针也存放在每个链表元素的pid_list字段中。

下面是处理PID散列表的函数和宏：

do_each_task_pid(nr,type,task)
while_each_task_pid(nr,type,task)

标记do-while循环的开始和结束，循环作用在PID值等于nr的PID链表上，链表的类型由参数type给出，task参数指向当前被扫描的元素的进程描述符。

find_task_by_pid_type(type,nr)

在type类型的散列表中查找PID等于nr的进程。该函数返回所匹配的进程描述符指针，若没有匹配的进程，函数返回NULL。

find_task_by_pid(nr)

与find_task_by_pid_type(PIDTYPE_PID, nr)相同。

attach_pid(task,type,nr)

把task指向的PID等于nr的进程描述符插入type类型的散列表中。如果一个PID等于nr的进程描述符已经在散列表中，这个函数就只把task插入已有的PID进程链表中。

detach_pid(task,type)

从type类型的PID进程链表中删除task所指向的进程描述符。如果删除后PID进程链表没有变为空，则函数终止，否则，该函数还要从type类型的散列表中删除进程描述符。最后，如果PID的值没有出现在任何其他的散列表中，为了这个值能够被反复使用，该函数还必须清除PID位图中的相应位。

next_thread(task)

返回PIDTYPE_TGID类型的散列表链表中task指示的下一个轻量级进程的进程描述符。由于散列链表是循环的，若应用于传统的进程，那么该宏返回进程本身的描述符地址。

如何组织进程

运行队列链表把处于TASK_RUNNING状态的所有进程组织在一起。当要把其他状态的进程分组时，不同的状态要求不同的处理，Linux选择了下列方式之一：

- 没有为处于TASK_STOPPED、EXIT_ZOMBIE或EXIT_DEAD状态的进程建立专门的链表。由于对处于暂停、僵死、死亡状态进程的访问比较简单，或者通过PID，或者通过特定父进程的子进程链表，所以不必对这三种状态进程分组。
- 没有为处于、状态的进程建立专门的链表。由于对处于暂停、僵死、死亡状态进程的访问比较简单，或者通过PID，或者通过特定父进程的子进程链表，所以不必对这三种状态进程分组。

等待队列

等待队列在内核中有很多用途，尤其用在中断处理、进程同步及定时。因为这些主题将在以后的章节中讨论，所以我们只在这里说明，进程必须经常等待某些事件的发生，例如，等待一个磁盘操作的终止，等待释放系统资源，或等待时间经过固定的间隔。等待队列实现了在事件上的条件等待：希望等待特定事件的进程把自己放进合适的等待队列，并放弃控制权。因此，等待队列表示一组睡眠的进程，当某一条件变为真时，由内核唤醒它们。

等待队列由双向链表实现，其元素包括指向进程描述符的指针。每个等待队列都有一个等待队列头（*wait queue head*），等待队列头是一个类型为wait_queue_head_t的数据结构：

```
struct _ _wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct _ _wait_queue_head wait_queue_head_t;
```

因为等待队列是由中断处理程序和主要内核函数修改的，因此必须对其双向链表进行保护以免对其进行同时访问，因为同时访问会导致不可预测的后果（参见第五章）。同步是通过等待队列头中的lock自旋锁达到的。task_list字段是等待进程链表的头。

等待队列链表中的元素类型为wait_queue_t：

```
struct _ _wait_queue {
    unsigned int flags;
    struct task_struct * task;
    wait_queue_func_t func;
    struct list_head task_list;
};
typedef struct _ _wait_queue wait_queue_t;
```

等待队列链表中的每个元素代表一个睡眠进程，该进程等待某一事件的发生；它的描述符地址存放在task字段中。task_list字段中包含的是指针，由这个指针把一个元素链接到等待相同事件的进程链表中。

然而，要唤醒等待队列中所有睡眠的进程有时并不方便。例如，如果两个或多个进程正在等待互斥访问某一要释放的资源，仅唤醒等待队列中的一个进程才有意义。这个进程占有资源，而其他进程继续睡眠。(这就避免了所谓“雷鸣般兽群”问题，即唤醒多个进程只为了竞争一个资源，而这个资源只能有一个进程访问，结果是其他进程必须再次回去睡眠。)

因此，有两种睡眠进程：互斥进程（等待队列元素的flags字段为1）由内核有选择地唤醒，而非互斥进程（falgs值为0）总是由内核在事件发生时唤醒。等待访问临界资源的进程就是互斥进程的典型例子。等待相关事件的进程是非互斥的。例如，我们考虑等待磁盘传输结束的一组进程：一但磁盘传输完成，所有等待的进程都会被唤醒。正如我们将在下面所看到的那样，等待队列元素的func字段用来表示等待队列中睡眠进程应该用什么方式唤醒。

等待队列的操作

可以用DECLARE_WAIT_QUEUE_HEAD(name)宏定义一个新等待队列的头，它静态地声明一个叫name的等待队列的头变量并对该变量的lock和task_list字段进行初始化。函数init_waitqueue_head()可以用来初始化动态分配的等待队列的头变量。

函数init_waitqueue_entry(q,p)如下所示初始化wait_queue_t结构的变量q：

```
q->flags = 0;
q->task = p;
q->func = default_wake_function;
```

非互斥进程p将由default_wake_function()唤醒，default_wake_function()是在第七章中要讨论的try_to_wake_up()函数的一个简单的封装。

也可以选择DEFINE_WAIT宏声明一个wait_queue_t类型的新变量，并用CPU上运行的当前进程的描述符和唤醒函数autoremove_wake_function()的地址初始化这个新变量。这个函数调用default_wake_function()来唤醒睡眠进程，然后从等待队列的链表中删除对应的元素(每个等待队列链表中的一个元素其实就是指向睡眠进程描述符的指针)。最后，内核开发者可以通过init_waitqueue_func_entry()函数来自定义唤醒函数，该函数负责初始化等待队列的元素。

一但定义了一个元素，必须把它插入等待队列。add_wait_queue()函数把一个非互斥

进程插入等待队列链表的第一个位置。add_wait_queue_exclusive()函数把一个互斥进程插入等待队列链表的最后一个位置。remove_wait_queue()函数从等待队列链表中删除一个进程。waitqueue_active()函数检查一个给定的等待队列是否为空。

要等待特定条件的进程可以调用如下列表中的任何一个函数。

- sleep_on()对当前进程进行操作:

```
void sleep_on(wait_queue_head_t *wq)
{
   wait_queue_t wait;
   init_waitqueue_entry(&wait, current);
   current->state = TASK_UNINTERRUPTIBLE;
   add_wait_queue(wq,&wait); /*  wq指向当前队列的头  */
   schedule();
   remove_wait_queue(wq, &wait);
}
```

  该函数把当前进程的状态设置为TASK_UNINTERRUPTIBLE，并把它插入到特定的等待队列。然后，它调用调度程序，而调度程序重新开始另一个程序的执行。当睡眠进程被唤醒时，调度程序重新开始执行sleep_on()函数，把该进程从等待队列中删除。

- interruptible_sleep_on()与sleep_on()函数是一样的，但稍有不同，前者把当前进程的状态设置为TASK_INTERRUPTIBLE而不是TASK_UNINTERRUPTIBLE，因此，接受一个信号就可以唤醒当前进程。

- sleep_on_timeout()和interruptible_sleep_on_timeout()与前面函数类似，但它们允许调用者定义一个时间间隔，过了这个间隔以后，进程将由内核唤醒。为了做到这点，它们调用schedule_timeout()函数而不是schedule()函数（参见第六章中“动态定时器的应用”一节）。

- 在Linux 2.6中引入的prepare_to_wait()、prepare_to_wait_exclusive()和finish_wait()函数提供了另外一种途径来使当前进程在一个等待队列中睡眠。它们的典型应用如下：

```
DEFINE_WAIT(wait);
prepare_to_wait_exclusive(&wq, &wait, TASK_INTERRUPTIBLE);
                              /* wq 是等待队列的头 */
...
if (!condition)
    schedule();
finish_wait(&wq, &wait);
```

  函数prepare_to_wait()和prepare_to_wait_exclusive()用传递的第三个参

数设置进程的状态，然后把等待队列元素的互斥标志flag分别设置为0（非互斥）或1（互斥），最后，把等待元素wait插入到以wq为头的等待队列的链表中。

进程一但被唤醒就执行finish_wait()函数，它把进程的状态再次设置为TASK_RUNNING（仅发生在调用schedule()之前，唤醒条件变为真的情况下），并从等待队列中删除等待元素（除非这个工作已经由唤醒函数完成）。

- wait_event和wait_event_interruptible宏使它们的调用进程在等待队列上睡眠，一直到修改了给定条件为止。例如，宏wait_event(wq,condition)本质上实现下面的功能：

```
DEFINE_WAIT(_  _wait);
for (;;) {
    prepare_to_wait(&wq, &_  _wait, TASK_UNINTERRUPTIBLE);
    if (condition)
       break;
    schedule();
}
finish_wait(&wq, &_  _wait);
```

对上面列出的函数做一些说明：sleep_on()类函数在以下条件下不能使用，那就是必须测试条件并且当条件还没有得到验证时又紧接着让进程去睡眠，由于那些条件是众所周知的竞争条件产生的根源，所以不鼓励这样使用。此外，为了把一个互斥进程插入等待队列，内核必须使用prepare_to_wait_exclusive()函数［或者只是直接调用add_wait_queue_exclusive()］。所有其他的相关函数把进程当作非互斥进程来插入。最后，除非使用DEFINE_WAIT或finish_wait()，否则内核必须在唤醒等待进程后从等待队列中删除对应的等待队列元素。

内核通过下面的任何一个宏唤醒等待队列中的进程并把它们的状态置为TASK_RUNNING：wake_up，wake_up_nr，wake_up_all，wake_up_interruptible，wake_up_interruptible_nr，wake_up_interruptible_all，wake_up_interruptible_sync和wake_up_locked。从每个宏的名字我们可以明白其功能：

- 所有宏都考虑到处于TASK_INTERRUPTIBLE状态的睡眠进程；如果宏的名字中不含字符串"interruptible"，那么处于TASK_UNINTERRUPTIBLE状态的睡眠进程也被考虑到。
- 所有宏都唤醒具有请求状态的所有非互斥进程（参见上一项）。
- 名字中含有“nr”字符串的宏唤醒给定数的具有请求状态的互斥进程；这个数字是宏的一个参数。名字中含有“all”字符串的宏唤醒具有请求状态的所有互斥进程。最后，名字中不含“nr”或“all”字符串的宏只唤醒具有请求状态的一个互斥进程。
- 名字中不含有“sync”字符串的宏检查被唤醒进程的优先级是否高于系统中正在运

行进程的优先级，并在必要时调用schedule()。这些检查并不是由名字中含有“sync”字符串的宏进行的，造成的结果是高优先级进程的执行稍有延迟。

- wake_up_locked宏和wake_up宏相类似，仅有的不同是当wait_queue_head_t中的自旋锁已经被持有时要调用wake_up_locked。

例如，wake_up宏等价于下列代码片段：

```
void wake_up(wait_queue_head_t *q)
{
   struct list_head *tmp;
   wait_queue_t *curr;

   list_for_each(tmp, &q->task_list) {
       curr = list_entry(tmp, wait_queue_t, task_list);
       if (curr->func(curr, TASK_INTERRUPTIBLE|TASK_UNINTERRUPTIBLE,
                      0, NULL) && curr->flags)
           break;
    }
}
```

list_for_each宏扫描双向链表q->task_list中的所有项，即等待队列中的所有进程。对每一项，list_entry宏都计算wait_queue_t变量对应的地址。这个变量的func字段存放唤醒函数的地址，它试图唤醒由等待队列元素的task字段标识的进程。如果一个进程已经被有效地唤醒（函数返回1）并且进程是互斥的（curr->flags等于1），循环结束。因为所有的非互斥进程总是在双向链表的开始位置，而所有的互斥进程在双向链表的尾部，所以函数总是先唤醒非互斥进程然后再唤醒互斥进程，如果有进程存在的话（注4）。

进程资源限制

每个进程都有一组相关的资源限制（*resource limit*），限制指定了进程能使用的系统资源数量。这些限制避免用户过分使用系统资源（CPU、磁盘空间等）。Linux承认以下表3-7中的资源限制。

对当前进程的资源限制存放在current->signal->rlim字段，即进程的信号描述符的一个字段（参见第十一章“与信号相关的数据结构”一节）。该字段是类型为rlimit结构的数组，每个资源限制对应一个元素：

```
struct rlimit {
    unsigned long rlim_cur;
    unsigned long rlim_max;
};
```

注4：顺便提一下，一个等待队列中同时包含互斥进程和非互斥进程的情况是非常罕见的。

表 3-7：资源限制

字段名	说明
RLIMIT_AS	进程地址空间的最大数（以字节为单位）。当进程使用 malloc()或相关函数扩大它的地址空间时，内核检查这个值（参见第九章“进程的地址空间”一节）
RLIMIT_CORE	内存信息转储文件的大小（以字节为单位）。当一个进程异常终止时，内核在进程的当前目录下创建内存信息转储文件之前检查这个值（参见第十一章的“传递信号之前所执行的操作”一节）。如果这个限制为0，那么，内核就不创建这个文件
RLIMIT_CPU	进程使用CPU的最长时间（以秒为单位）。如果进程超过了这个限制，内核就向它发一个SIGXCPU信号，然后如果进程还不终止，再发一个SIGKILL信号（参见第十一章）
RLIMIT_DATA	堆大小的最大值（以字节为单位）。在扩充进程的堆之前，内核检查这个值（参见第九章中“堆的管理”一节）
RLIMIT_FSIZE	文件大小的最大值（以字节为单位）。如果进程试图把一个文件的大小扩充到大于这个值，内核就给这个进程发SIGXFSZ信号
RLIMIT_LOCKS	文件锁的最大值（目前是非强制的）
RLIMIT_MEMLOCK	非交换内存的最大值（以字节为单位）。当进程试图通过 mlock()或mlockall()系统调用锁住一个页框时，内核检查这个值（参见第九章“分配线性地址区间”一节）
RLIMIT_MSGQUEUE	POSIX消息队列中的最大字节数（参见第十九章“POSIX消息队列”一节）
RLIMIT_NOFILE	打开文件描述符的最大数。当打开一个新文件或复制一个文件描述符时，内核检查这个值（参见第十二章）
RLIMIT_NPROC	用户能拥有的进程最大数（参见本章“clone(), fork()及vfork() 系统调用”一节）
RLIMIT_RSS	进程所拥有的页框最大数（目前是非强制的）
RLIMIT_SIGPENDING	进程挂起信号的最大数（参见第十一章）
RLIMIT_STACK	栈大小的最大值（以字节为单位）。内核在扩充进程的用户态堆栈之前检查这个值（参见第九章“异常处理”一节）

rlim_cur字段是资源的当前资源限制。例如，current->signal->rlim[RLIMIT_CPU].rlim_cur表示正运行进程所占用CPU时间的当前限制。

rlim_max字段是资源限制所允许的最大值。利用getrlimit()和setrlimit()系统调用，用户总能把一些资源的rlim_cur限制增加到rlim_max。然而，只有超级用户（或更确切地说，具有CAP_SYS_RESOURCE权能的用户）才能改变rlim_max字段，或把rlim_cur字段设置成大于相应rlim_max字段的一个值。

大多数资源限制包含值RLIMIT_INFINITY(0xffffffff)，它意味着没有对相应的资源施加用户限制（当然，由于内核设计上的限制，可用RAM、可用磁盘空间等，实际的限制还是存在的）。然而，系统管理员可以给一些资源选择施加更强的限制。只要用户注册进系统，内核就创建一个由超级用户拥有的进程，超级用户能调用setrlimit()以减少一个资源rlim_max和rlim_cur字段的值。随后，同一进程执行一个login shell，该进程就变为由用户拥有。由用户创建的每个新进程都继承其父进程rlim数组的内容，因此，用户不能忽略系统强加的限制。

进程切换

为了控制进程的执行，内核必须有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行。这种行为被称为进程切换（*process switch*）、任务切换（*task switch*）或上下文切换（*context switch*）。下面几节描述在Linux中进行进程切换的主要内容。

硬件上下文

尽管每个进程可以拥有属于自己的地址空间，但所有进程必须共享CPU寄存器。因此，在恢复一个进程的执行之前，内核必须确保每个寄存器装入了挂起进程时的值。

进程恢复执行前必须装入寄存器的一组数据称为硬件上下文（*hardware context*）。硬件上下文是进程可执行上下文的一个子集，因为可执行上下文包含进程执行时需要的所有信息。在Linux中，进程硬件上下文的一部分存放在TSS段，而剩余部分存放在内核态堆栈中。

在下面的描述中，我们假定用prev局部变量表示切换出的进程的描述符，next表示切换进的进程的描述符。因此，我们把进程切换定义为这样的行为：保存prev硬件上下文，用next硬件上下文代替prev。因为进程切换经常发生，因此减少保存和装入硬件上下文所花费的时间是非常重要的。

早期的Linux版本利用80x86体系结构所提供的硬件支持，并通过far jmp指令（注5）跳到next进程TSS描述符的选择符来执行进程切换。当执行这条指令时，CPU通

注5：far jmp指令既修改cs寄存器，也修改eip寄存器，而简单的jmp指令只修改eip寄存器。

过自动保存原来的硬件上下文，装入新的硬件上下文来执行硬件上下文切换。但基于以下原因，Linux 2.6使用软件执行进程切换：

- 通过一组mov指令逐步执行切换，这样能较好地控制所装入数据的合法性。尤其是，这使检查ds和es段寄存器的值成为可能，这些值有可能被恶意用户伪造。当用单独的far jmp指令时，不可能进行这类检查。
- 旧方法和新方法所需时间大致相同。然而，尽管当前的切换代码还有改进的余地，却不能对硬件上下文切换进行优化。

进程切换只发生在内核态。在执行进程切换之前，用户态进程使用的所有寄存器内容都已保存在内核态堆栈上（参见第四章），这也包括ss和esp这对寄存器的内容（存储用户态堆栈指针的地址）。

任务状态段

80x86体系结构包括了一个特殊的段类型，叫任务状态段（*Task State Segment*，TSS）来存放硬件上下文。尽管Linux并不使用硬件上下文切换，但是强制它为系统中每个不同的CPU创建一个TSS。这样做的两个主要理由为：

- 当80x86的一个CPU从用户态切换到内核态时，它就从TSS中获取内核态堆栈的地址（参见第四章“中断和异常的硬件处理”一节和第十章“通过sysenter指令发送系统调用”一节）。
- 当用户态进程试图通过in或out指令访问一个I/O端口时，CPU需要访问存放在TSS中的I/O许可权位图（Permission Bitmap）以检查该进程是否有访问端口的权力。

  更确切地说，当进程在用户态下执行in或out指令时，控制单元执行下列操作：

  1. 它检查eflags寄存器中的2位IOPL字段。如果该字段值为3，控制单元就执行I/O指令。否则，执行下一个检查。
  2. 访问tr寄存器以确定当前的TSS和相应的I/O许可权位图。
  3. 检查I/O指令中指定的I/O端口在I/O许可权位图中对应的位。如果该位清0，这条I/O指令就执行，否则控制单元产生一个“General protection”异常。

tss_struct结构描述TSS的格式。正如第二章所提到的，init_tss数组为系统上每个不同的CPU存放一个TSS。在每次进程切换时，内核都更新TSS的某些字段以便相应的CPU控制单元可以安全地检索到它需要的信息。因此，TSS反映了CPU上的当前进程的特权级，但不必为没有在运行的进程保留TSS。

每个TSS有它自己 8字节的任务状态段描述符(*Task State Segment Descriptor*,TSSD)。这个描述符包括指向TSS起始地址的32位Base字段，20位Limit字段。TSSD的S标志位被清0，以表示相应的TSS是系统段的事实（参见第二章“段描述符”一节）。

Type字段置为11或9以表示这个段实际上是一个TSS。在Intel的原始设计中，系统中的每个进程都应当指向自己的TSS；Type字段的第二个有效位叫做Busy位；如果进程正由CPU执行，则该位置1，否则置0。在Linux的设计中，每个CPU只有一个TSS，因此，Busy位总置为1。

由Linux创建的TSSD存放在全局描述符表（GDT）中，GDT的基地址存放在每个CPU的gdtr寄存器中。每个CPU的tr寄存器包含相应TSS的TSSD选择符，也包含了两个隐藏的非编程字段：TSSD的Base字段和Limit字段。这样，处理器就能直接对TSS寻址而不用从GDT中检索TSS的地址。

thread字段

在每次进程切换时，被替换进程的硬件上下文必须保存在别处。不能像Intel原始设计那样把它保存在TSS中，因为Linux为每个处理器而不是为每个进程使用TSS。

因此，每个进程描述符包含一个类型为thread_struct的thread字段，只要进程被切换出去，内核就把其硬件上下文保存在这个结构中。随后我们会看到，这个数据结构包含的字段涉及大部分CPU寄存器，但不包括诸如eax、ebx等等这些通用寄存器，它们的值保留在内核堆栈中。

执行进程切换

进程切换可能只发生在精心定义的点：schedule()函数(在第七章会用很长的篇幅来讨论)。这里，我们仅关注内核如何执行一个进程切换。

从本质上说，每个进程切换由两步组成：

1. 切换页全局目录以安装一个新的地址空间；我们将在第九章描述这一步。
2. 切换内核态堆栈和硬件上下文，因为硬件上下文提供了内核执行新进程所需要的所有信息，包含CPU寄存器。

我们又一次假定prev指向被替换进程的描述符，而next指向被激活进程的描述符。我们在第七章会看到，prev和next是schedule()函数的局部变量。

switch_to 宏

进程切换的第二步由switch_to宏执行。它是内核中与硬件关系最密切的例程之一，要理解它到底做了些什么我们必须下些功夫。

首先，该宏有三个参数，它们是prev,next和last。你可能很容易猜到prev和next的作用：它们仅是局部变量prev和next的占位符，即它们是输入参数，分别表示被替换进程和新进程描述符的地址在内存中的位置。

那第三个参数last呢？在任何进程切换中，涉及到三个进程而不是两个。假设内核决定暂停进程A而激活进程B。在schedule()函数中，prev指向A的描述符而next指向B的描述符。switch_to宏一但使A暂停，A的执行流就冻结。

随后，当内核想再次此激活A，就必须暂停另一个进程C（这通常不同于B），于是就要用prev指向C而next指向A来执行另一个switch_to宏。当A恢复它的执行流时，就会找到它原来的内核栈，于是prev局部变量还是指向A的描述符而next指向B的描述符。此时，代表进程A执行的内核就失去了对C的任何引用。但是，事实表明这个引用对于完成进程切换是很有用的（更多细节参见第七章）。

switch_to宏的最后一个参数是输出参数，它表示宏把进程C的描述符地址写在内存的什么位置了（当然，这是在A恢复执行之后完成的）。在进程切换之前，宏把第一个输入参数prev（即在A的内核堆栈中分配的prev局部变量）表示的变量的内容存入CPU的eax寄存器。在完成进程切换，A已经恢复执行时，宏把CPU的eax寄存器的内容写入由第三个输出参数——last所指示的A在内存中的位置。因为CPU寄存器不会在切换点发生变化，所以C的描述符地址也存在内存的这个位置。在schedule()执行过程中，参数last指向A的局部变量prev，所以prev被C的地址覆盖。

图3-7显示了进程A,B,C内核堆栈的内容以及eax寄存器的内容。必须注意的是：图中显示的是在被eax寄存器的内容覆盖以前的prev局部变量的值。

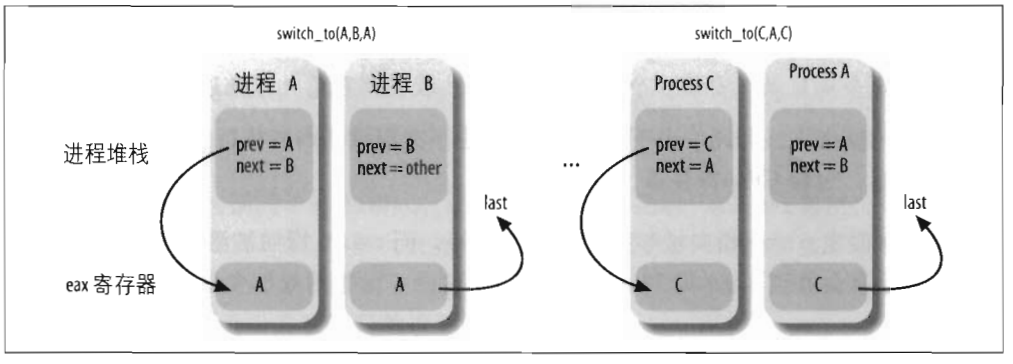


图3-7：通过一个进程切换保留对进程C的引用

由于switch_to宏采用扩展的内联汇编语言编码，所以可读性比较差：实际上这段代码通过特殊位置记数法使用寄存器，而实际使用的通用寄存器由编译器自由选择。我们将采用标准汇编语言而不是麻烦的内联汇编语言来描述switch_to宏在80x86微处理器上所完成的典型工作。

1. 在eax和edx寄存器中分别保存prev和next的值：

```
movl prev,%eax
movl next,%edx
```

2. 把eflags和ebp寄存器的内容保存在prev内核栈中。必须保存它们的原因是编译器认为在switch_to结束之前它们的值应当保持不变。

```
pushfl
pushl %ebp
```

3. 把esp的内容保存到prev->thread.esp中以使该字段指向prev内核栈的栈顶：

```
movl %esp,484(%eax)
```

484（%eax）操作数表示内存单元的地址为eax内容加上484。

4. 把next->thread.esp装入esp。此时，内核开始在next的内核栈上操作，因此这条指令实际上完成了从prev到next的切换。由于进程描述符的地址和内核栈的地址紧挨着（就像我们在本章前面“标识一个进程”一节所解释的），所以改变内核栈意味着改变当前进程。

```
movl 484(%edx), %esp
```

5. 把标记为1的地址（本节后面所示）存入prev->thread.eip。当被替换的进程重新恢复执行时，进程执行被标记为1的那条指令：

```
movl $1f, 480(%eax)
```

6. 宏把next->thread.eip的值（绝大多数情况下是一个被标记为1的地址）压入next的内核栈：

```
pushl 480(%edx)
```

7. 跳到__switch_to()C函数（见下面）：

```
jmp__switch_to
```

8. 这里被进程B替换的进程A再次获得CPU：它执行一些保存eflags和ebp寄存器内容的指令，这两条指令的第一条指令被标记为1。

```
1:
     popl %ebp
     popfl
```

注意这些pop指令是怎样引用prev进程的内核栈的。当进程调度程序选择了prev作为新进程在CPU上运行时，将执行这些指令。于是，以prev作为第二个参数调用switch_to。因此，esp寄存器指向prev的内核栈。

9. 拷贝eax寄存器(上面步骤1中被装载)的内容到switch_to宏的第三个参数last标识的内存区域中：

```
movl %eax, last
```

正如先前讨论的，eax寄存器指向刚被替换的进程的描述符（注6)。

__switch_to()函数

__switch_to()函数执行大多数开始于switch_to()宏的进程切换。这个函数作用于prev_p和next_p参数，这两个参数表示前一个进程和新进程。这个函数的调用不同于一般函数的调用，因为__switch_to()从eax和edx取参数prev_p和next_p(我们在前面已看到这些参数就是保存在那里)，而不像大多数函数一样从栈中取参数。为了强迫函数从寄存器取它的参数，内核利用__attribute__和regparm关键字，这两个关键字是C语言非标准的扩展名，由gcc编译程序实现。在 *include /asm-i386 / system.h* 头文件中，__switch_to()函数的声明如下：

```
__switch_to(struct task_struct *prev_p,
            struct task_struct *next_p)
__attribute__(regparm(3));
```

函数执行的步骤如下：

1. 执行由__unlazy_fpu()宏产生的代码（参见本章稍后“保存和加载FPU、MMX及XMM寄存器”一节)，以有选择地保存prev_p进程的FPU、MMX及XMM寄存器的内容。

```
__unlazy_fpu(prev_p);
```

2. 执行smp_processor_id()宏获得本地 (*local*) *CPU*的下标，即执行代码的CPU。该宏从当前进程的thread_info结构的cpu字段获得下标并将它保存到cpu局部变量。

3. 把next_p->thread.esp0装入对应于本地CPU的TSS的esp0字段；我们将在第十章的“通过sysenter指令发生系统调用”一节看到，以后任何由sysenter汇编指令产生的从用户态到内核态的特权级转换将把这个地址拷贝到esp寄存器中：

```
init_tss[cpu].esp0 = next_p->thread.esp0;
```

注6： 正如本节前面所叙述的，当前执行的schedule()函数重新使用了prev局部变量，于是汇编语言指令就是：movl %eax,prev。

4. 把next_p进程使用的线程局部存储（TLS）段装入本地CPU的全局描述符表；三个段选择符保存在进程描述符内的tls_array数组中（参见第二章的“Linux中的分段”一节）。

```
cpu_gdt_table[cpu][6] = next_p->thread.tls_array[0];
cpu_gdt_table[cpu][7] = next_p->thread.tls_array[1];
cpu_gdt_table[cpu][8] = next_p->thread.tls_array[2];
```

5. 把fs和gs段寄存器的内容分别存放在prev-P->thread.fs和prev-P->thread.gs中，对应的汇编语言指令是：

```
movl %fs, 40(%esi)
movl %gs, 44(%esi)
```

esi寄存器指向prev_p->thread结构。

6. 如果fs或gs段寄存器已经被prev_p或next_p进程中的任意一个使用（也就是说如果它们有一个非0的值），则将next_p进程的thread_struct描述符中保存的值装入这些寄存器中。这一步在逻辑上补充了前一步中执行的操作。主要的汇编语言指令如下：

```
movl 40(%ebx),%fs
movl 44(%ebx),%gs
```

ebx寄存器指向next_p->thread结构。代码实际上更复杂，因为当它检测到一个无效的段寄存器值时，CPU可能产生一个异常。代码采用一种“修正（fix-up）”途径来考虑这种可能性（参见第十章“动态地址检查：修正代码”一节）。

7. 用next_p->thread.debugreg数组的内容装载dr0，…，dr7中的6个调试寄存器（注7）。只有在next_p被挂起时正在使用调试寄存器（也就是说，next_p->thread.debugreg[7]字段不为0），这种操作才能进行。这些寄存器不需要被保存，因为只有当一个调试器想要监控prev时prev_p->thread.debugreg才会被修改。

```
if (next_p->thread.debugreg[7]){
    loaddebug(&next_p->thread, 0);
    loaddebug(&next_p->thread, 1);
    loaddebug(&next_p->thread, 2);
    loaddebug(&next_p->thread, 3);
    /* 没有4和5*/
    loaddebug(&next_p->thread, 6);
    loaddebug(&next_p->thread, 7);
}
```

注7： 80×86调试寄存器允许进程被硬件监控。最多可定义4个断点区域。一个被监控的进程只要产生的一个线性地址位于个断点区域中之一，就会产生一个异常。

8. 如果必要，更新TSS中的I/O位图。当next_p或prev_p有其自己的定制I/O权限位图时必须这么做：

```
if (prev_p->thread.io_bitmap_ptr || next_p->thread.io_bitmap_ptr)
   handle_io_bitmap(&next_p->thread, &init_tss[cpu]);
```

因为进程很少修改I/O权限位图，所以该位图在“懒”模式中被处理：当且仅当一个进程在当前时间片内实际访问I/O端口时，真实位图才被拷贝到本地CPU的TSS中。进程的定制I/O权限位图被保存在thread_info结构的io_bitmap_ptr字段指向的缓冲区中。handle_io_bitmap()函数为next_p进程设置本地CPU使用的TSS的io_bitmap字段如下：

- 如果next_p进程不拥有自己的I/O权限位图，则TSS的io_bitmap字段被设为0x8000。
- 如果next_p进程拥有自己的I/O权限位图，则TSS的io_bitmap字段被设为0x9000。

TSS的io_bitmap字段应当包含一个在TSS中的偏移量，其中存放实际位图。无论何时用户态进程试图访问一个I/O端口，0x8000和0x9000指向TSS界限之外并将因此引起“General protection”异常(参见第四章的“异常”一节)。do_general_protection()异常处理程序将检查保存在io_bitmap字段的值：如果是0x8000，函数发送一个SIGSEGV信号给用户态进程；如果是0x9000，函数把进程位图（由thread_info结构中的io_bitmap_ptr字段指示）拷贝到本地CPU的TSS中，把io_bitmap字段设为实际位图的偏移（104），并强制再一次执行有缺陷的汇编语言指令。

9. 终止。_ _switch_to()C函数通过使用下列声明结束：

```
return prev_p;
```

由编译器产生的相应汇编语言指令是：

```
movl %edi,%eax
ret
```

prev_p参数（现在在edi中）被拷贝到eax，因为缺省情况下任何C函数的返回值被传递给eax寄存器。注意eax的值因此在调用__switch_to()的过程中被保护起来；这非常重要，因为调用switch_to宏时会假定eax总是用来存放将被替换的进程描述符的地址。

汇编语言指令ret把栈顶保存的返回地址装入eip程序计数器。不过，通过简单地跳转到__switch_to()函数来调用该函数。因此，ret汇编指令在栈中找到标号为1的指令的地址，其中标号为1的地址是由switch_to()宏推入栈中的。如果因为

next_p第一次执行而以前从未被挂起，__switch_to()就找到ret_from_fork()函数的起始地址（参见本章后面“clone()，fork()和vfork()系统调用一节”）。

保存和加载FPU、MMX及XMM寄存器

从Intel 80486DX开始，算术浮点单元（floating-point unit，FPU）已被集成到CPU中。数学协处理这个名词使人想起使用昂贵的专用芯片执行浮点计算的岁月。然而，为了维持与旧模式的兼容，浮点算术函数用ESCAPE指令来执行，这个指令的一些前缀字节在0xd8和0xdf之间。这些指令作用于包含在CPU中的浮点寄存器集。显然，如果一个进程正在使用ESCAPE指令，那么，浮点寄存器的内容就属于它的硬件上下文，并且应该被保存。

在最近的Pentium模型中，Intel在它的微处理器中引入一个新的汇编指令集，叫做MMX指令，用来加速多媒体应用程序的执行。MMX指令作用于FPU的浮点寄存器。选择这种体系结构的明显缺点是编程者不能把浮点指令与MMX指令混在一起使用。优点是操作系统设计者能忽视新指令集，因为保存浮点单元状态的任务切换代码可以不加修改地应用到保存MMX状态。

MMX指令加速了多媒体应用程序的执行，因为它们在处理器内部引入了单指令多数据（single-instruction multiple-data，SIMD）流水线。Pentium III模型扩展了这种SIMD能力：它引入SSE扩展（Streaming SIMD Extensions），该扩展为处理包含在8个128位寄存器（叫做XMM寄存器）的浮点值增加了功能。这样的寄存器不与FPU和MMX寄存器重叠，因此SSE和FPU/MMX指令可以随意地混合。Pentium 4模型指令还引入另一种特点：SSE2扩展，该扩展基本上是SSE的一个扩展，支持高精度浮点值。SSE2与SSE使用同一XMM寄存器集。

80x86微处理器并不在TSS中自动保存FPU、MMX和XMM寄存器。不过，它们包含某种硬件支持，能在需要时保存这些寄存器的值。硬件支持由cr0寄存器中的一个TS（Task-Switching）标志组成，遵循以下规则：

- 每当执行硬件上下文切换时，设置TS标志。
- 每当TS标志被设置时执行ESCAPE、MMX、SSE或SSE2指令，控制单元就产生一个“Device not available”异常（参见第四章）。

TS标志使得内核只有在真正需要时才保存和恢复FPU、MMX和XMM寄存器。为了说明它如何工作，让我们假设进程A使用数学协处理器。当发生上下文切换时，内核置TS标志并把浮点寄存器保存在进程A的TSS中。如果新进程B不利用协处理器，内核就不必恢复浮点寄存器的内容。但是，只要B打算执行ESCAPE或MMX指令，CPU就产生

一个“Device not available”异常，并且相应的异常处理程序用保存在进程B中的TSS的值装载浮点寄存器。

现在，让我们描述为处理FPU、MMX和XMM寄存器的选择性装入而引入的数据结构。它们存放在进程描述符的thread.i387子字段中，其格式由i387_union联合体描述：

```
union i387_union {
    struct i387_fsave_struct    fsave;
    struct i387_fxsave_struct   fxsave;
    struct i387_soft_struct     soft;
};
```

正如你看到的，这个字段只可以存放三种不同数据结构中的一种。i387_soft_struct结构由无数学协处理器的CPU模型使用；Linux内核通过软件模拟协处理器来支持这些老式芯片。不过，我们不打算进一步讨论这种遗留问题。i387_fsave_struct结构由具有数学协处理器、也可能有MMX单元的CPU模型使用。最后，i387_fxsave_struct结构由具有SSE和SSE2扩展功能的CPU模型使用。

进程描述符包含两个附加的标志：

- 包含在thread_info描述符的status字段中的TS_USEDFPU标志。它表示进程在当前执行的过程中是否使用过FPU、MMX和XMM寄存器。
- 包含在task_struct描述符的flags字段中的PF_USED_MATH标志。这个标志表示thread.i387子字段的内容是否有意义。该标志在两种情况下被清0（没有意义），如下所示：
  - 当进程调用execve()系统调用（参见第二十章）开始执行一个新程序时。因为控制权将不再返回到前一个程序，所以当前存放在thread.i387中的数据也不再使用。
  - 当在用户态下执行一个程序的进程开始执行一个信号处理程序时（参见第十一章）。因为信号处理程序与程序的执行流是异步的，因此，浮点寄存器对信号处理程序来说可能是毫无意义的。不过，内核开始执行信号处理程序之前在thread.i387中保存浮点寄存器，处理程序结束以后恢复它们。因此，信号处理程序可以使用数学协处理器。

保存FPU寄存器

如前所述，__switch_to()函数把被替换进程prev的描述符作为参数传递给__unlazy_fpu宏，并执行该宏。这个宏检查prev的TS_USEDFPU标志值。如果该标志被设

置，说明prev在这次执行中使用了FPU、MMX、SSE或SSE2指令；因此内核必须保存相关的硬件上下文：

```
if (prev->thread_info->status & TS_USEDFPU)
    save_init_fpu(prev);
```

save_init_fpu()函数依次执行下列操作：

1. 把FPU寄存器的内容转储到prev进程描述符中，然后重新初始化FPU。如果CPU使用SSE/SSE2扩展，则还应该转储XMM寄存器的内容，并重新初始化SSE/SSE2单元。一对功能强大的嵌入式汇编语言指令处理每件事情，如果CPU使用SSE/SSE2扩展，则：

   ```
   asm volatile( "fxsave %0 ; fnclex"
       : "=m" (tsk->thread.i387.fxsave) );
   ```

   否则：

   ```
   asm volatile( "fnsave %0 ; fwait"
       : "=m" (tsk->thread.i387.fsave) );
   ```

2. 重置prev的TS_USEDFPU标志：

   ```
   prev->thread_info->status &= ~TS_USEDFPU;
   ```

3. 用stts()宏设置cr0的TS标志，实际上，该宏产生下面的汇编语言指令：

   ```
   movl %cr0, %eax
   orl $8,%eax
   movl %eax, %cr0
   ```

装载FPU寄存器

当next进程刚恢复执行时，浮点寄存器的内容还没有被恢复，不过，cr0的TS标志位已由_ _unlazy_fpu()设置。因此，next进程第一次试图执行ESCAPE、MMX或SSE/SSE2指令时，控制单元产生一个“Device not available”异常，内核（更确切地说，由异常调用的异常处理程序）运行math_state_restore()函数。处理程序把next进程当作current进程。

```
void math_state_restore()
{
    asm volatile ("clts"); /* clear the TS flag of cr0 */
    if (!(current->flags & PF_USED_MATH))
        init_fpu(current);
    restore_fpu(current);
    current->thread.status |= TS_USEDFPU;
}
```

这个函数清cr0的TS标志，以便进程以后执行FPU、MMX或SSE/SSE2指令时不再触发“设备不可用”的异常。如果thread.i387子字段中的内容是无效的，也就是说，如果PF_USED_MATH标志等于0，就调用init_fpu()重新设置thread.i387子字段，并把PF_USED_MATH标志的当前值置为1。restore_fpu()函数把保存在thread.i387子字段中的适当值载入FPU寄存器。为此，根据CPU是否支持SSE/SSE2扩展来使用fxrstor或frstor汇编语言指令。最后，math_state_restore()设置TS_USEDFPU标志。

在内核态使用FPU、MMX和SSE/SSE2单元

内核也可以使用FPU、MMX和SSE/SSE2单元。当然，这样做的时候，应该避免干扰用户态进程所进行的任何计算。因此：

- 在使用协处理器之前，如果用户态进程使用了FPU（TS_USEDFPU标志），内核必须调用kernel_fpu_begin()，其本质就是调用save_init_fpu()来保存寄存器的内容，然后重新设置cr0寄存器的TS标志。
- 在使用完协处理器之后，内核必须调用kernel_fpu_end()设置cr0寄存器的TS标志。

稍后，当用户态进程执行协处理器指令时，math_state_restore()函数将恢复寄存器的内容（就像处理进程切换那样）。

但是，应该注意，当前用户态进程正在使用协处理器时，kernel_fpu_begin()的执行时间相当长，以至于无法通过使用FPU、MMX或SSE/SSE2单元达到加速的目的。实际上，内核只在有限的场合使用FPU、MMX或SSE/SSE2单元，典型的情况有：当移动或清除大内存区字段时，或者当计算校验和函数时。

创建进程

Unix操作系统紧紧依赖进程创建来满足用户的需求。例如，只要用户输入一条命令，shell进程就创建一个新进程，新进程执行shell的另一个拷贝。

传统的Unix操作系统以统一的方式对待所有的进程：子进程复制父进程所拥有的资源。这种方法使进程的创建非常慢且效率低，因为子进程需要拷贝父进程的整个地址空间。实际上，子进程几乎不必读或修改父进程拥有的所有资源，在很多情况下，子进程立即调用execve()，并清除父进程仔细拷贝过来的地址空间。

现代Unix内核通过引入三种不同的机制解决了这个问题：

- 写时复制技术允许父子进程读相同的物理页。只要两者中有一个试图写一个物理页，内核就把这个页的内容拷贝到一个新的物理页，并把这个新的物理页分配给正在写的进程。第九章将全面地解释这种技术在Linux中的实现。
- 轻量级进程允许父子进程共享每进程在内核的很多数据结构，如页表（也就是整个用户态地址空间）、打开文件表及信号处理。
- vfork()系统调用创建的进程能共享其父进程的内存地址空间。为了防止父进程重写子进程需要的数据，阻塞父进程的执行，一直到子进程退出或执行一个新的程序为止。我们将在下一节了解有关vfork()系统调用更多的知识。

clone()、fork()及vfork()系统调用

在Linux中，轻量级进程是由名为clone()的函数创建的，这个函数使用下列参数：

fn

指定一个由新进程执行的函数。当这个函数返回时，子进程终止。函数返回一个整数，表示子进程的退出代码。

arg

指向传递给fn()函数的数据。

flags

各种各样的信息。低字节指定子进程结束时发送到父进程的信号代码，通常选择SIGCHLD信号。剩余的3个字节给一clone标志组用于编码，如表3-8所示。

child_stack

表示把用户态堆栈指针赋给子进程的esp寄存器。调用进程（指调用clone()的父进程）应该总是为子进程分配新的堆栈。

tls

表示线程局部存储段（TLS）数据结构的地址，该结构是为新轻量级进程定义的（参见第二章“Linux GDT”一节）。只有在CLONE_SETTLS标志被设置时才有意义。

ptid

表示父进程的用户态变量地址，该父进程具有与新轻量级进程相同的PID。只有在CLONE_PARENT_SETTID标志被设置时才有意义。

ctid

表示新轻量级进程的用户态变量地址，该进程具有这一类进程的PID。只有在CLONE_CHILD_SETTID标志被设置时才有意义。

表3-8：clone 标志

标志名称	说明
CLONE_VM	共享内存描述符和所有的页表（参见第九章）
CLONE_FS	共享根目录和当前工作目录所在的表，以及用于屏蔽新文件初始许可权的位掩码值（所谓文件的 umask）
CLONE_FILES	共享打开文件表（参见第十二章）
CLONE_SIGHAND	共享信号处理程序的表、阻塞信号表和挂起信号表（参见第十一章）。如果这个标志为 true，就必须设置 CLONE_VM 标志
CLONE_PTRACE	如果父进程被跟踪，那么，子进程也被跟踪。尤其是，debugger 程序可能希望以自己作为父进程来跟踪子进程，在这种情况下，内核把该标志强置为 1
CLONE_VFORK	在发出 vfork() 系统调用时设置（参见本节后面）
CLONE_PARENT	设置子进程的父进程（进程描述符中的 parent 和 real_parent 字段）为调用进程的父进程
CLONE_THREAD	把子进程插入到父进程的同一线程组中，并迫使子进程共享父进程的信号描述符。因此也设置子进程的 tgid 字段和 group_leader 字段。如果这个标志位为 true，就必须设置 CLONE_SIGHAND 标志
CLONE_NEWNS	当 clone 需要自己的命名空间时（即它自己的已挂载文件系统视图）设置这个标志（参见第十二章）。不能同时设置 CLONE_NEWNS 和 CLONE_FS
CLONE_SYSVSEM	共享 System V IPC 取消信号量的操作（参见第十九章“IPC 信号量”一节）
CLONE_SETTLS	为轻量级进程创建新的线程局部存储段(TLS)，该段由参数 tls 所指向的结构进行描述
CLONE_PARENT_SETTID	把子进程的 PID 写入由 ptid 参数所指向的父进程的用户态变量
CLONE_CHILD_CLEARTID	如果该标志被设置，则内核建立一种触发机制，用在子进程要退出或要开始执行新程序时。在这些情况下，内核将清除由参数 ctid 所指向的用户态变量，并唤醒等待这个事件的任何进程
CLONE_DETACHED	遗留标志，内核会忽略它
CLONE_UNTRACED	内核设置这个标志以使 CLONE_PTRACE 标志失去作用（用来禁止内核线程跟踪进程，参见本章稍后的“内核线程”一节）

表3-8：clone 标志（续）

标志名称	说明
CLONE_CHILD_SETTID	把子进程的PID写入由ctid参数所指向的子进程的用户态变量中
CLONE_STOPPED	强迫子进程开始于TASK_STOPPED状态

实际上，clone()是在C语言库中定义的一个封装(wrapper)函数(参见第十章"POSIX API 和系统调用"一节)，它负责建立新轻量级进程的堆栈并且调用对编程者隐藏的clone()系统调用。实现clone()系统调用的sys_clone()服务例程没有fn和arg参数。实际上，封装函数把fn指针存放在子进程堆栈的某个位置处，该位置就是该封装函数本身返回地址存放的位置。arg指针正好存放在子进程堆栈中fn的下面。当封装函数结束时，CPU从堆栈中取出返回地址，然后执行fn(arg)函数。

传统的fork()系统调用在Linux中是用clone()实现的，其中clone()的flags参数指定为SIGCHLD信号及所有清0的clone标志，而它的child_stack参数是父进程当前的堆栈指针。因此，父进程和子进程暂时共享同一个用户态堆栈。但是，要感谢写时复制机制，通常只要父子进程中有一个试图去改变栈，则立即各自得到用户态堆栈的一份拷贝。

前一节描述的vfork()系统调用在Linux中也是用clone()实现的，其中clone()的参数flags指定为SIGCHLD信号和CLONE_VM及CLONE_VFORK标志，clone()的参数child_stack等于父进程当前的栈指针。

do_fork()函数

do_fork()函数负责处理clone()、fork()和vfork()系统调用，执行时使用下列参数：

clone_flags
: 与clone()的参数flags相同。

stack_start
: 与clone()的参数child_stack相同。

regs
: 指向通用寄存器值的指针，通用寄用器的值是在从用户态切换到内核态时被保存到内核态堆栈中的（参见第四章"do_IRQ()函数"一节）。

stack_size
: 未使用（总是被设置为0）。

parent_tidptr,child_tidptr

与clone()中的对应参数ptid和ctid相同。

do_fork()利用辅助函数copy_process()来创建进程描述符以及子进程执行所需要的所有其他内核数据结构。下面是do_fork()执行的主要步骤：

1. 通过查找pidmap_array位图，为子进程分配新的PID（参见本章前面“标识一个进程”一节）。
2. 检查父进程的ptrace字段(current->ptrace)：如果它的值不等于0，说明有另外一个进程正在跟踪父进程，因而，do_fork()检查debugger程序是否自己想跟踪子进程（独立于由父进程指定的CLONE_PTRACE标志的值）。在这种情况下，如果子进程不是内核线程（CLONE_UNTRACED标志被清0），那么do_fork()函数设置CLONE_PTRACE标志。
3. 调用copy_process()复制进程描述符。如果所有必须的资源都是可用的，该函数返回刚创建的task_struct描述符的地址。这是创建过程的关键步骤，我们将在do_fork()之后描述它。
4. 如果设置了CLONE_STOPPED标志，或者必须跟踪子进程，即在p->ptrace中设置了PT_PTRACED标志，那么子进程的状态被设置成TASK_STOPPED，并为子进程增加挂起的SIGSTOP信号（参见第十一章“信号的作用”一节）。在另外一个进程（不妨假设是跟踪进程或是父进程）把子进程的状态恢复为TASK_RUNNING之前（通常是通过发送SIGCONT信号），子进程将一直保持TASK_STOPPED状态。
5. 如果没有设置CLONE_STOPPED标志，则调用wake_up_new_task()函数以执行下述操作：
   a. 调整父进程和子进程的调度参数（参见第七章“调度算法”一节）
   b. 如果子进程将和父进程运行在同一个CPU上（注8），而且父进程和子进程不能共享同一组页表（CLONE_VM标志被清0），那么，就把子进程插入父进程运行队列，插入时让子进程恰好在父进程前面，因此而迫使子进程先于父进程运行。如果子进程刷新其地址空间，并在创建之后执行新程序，那么这种简单的处理会产生较好的性能。而如果我们让父进程先运行，那么写时复制机制将会执行一系列不必要的页面复制。
   c. 否则，如果子进程与父进程运行在不同的CPU上，或者父进程和子进程共享同一组页表（CLONE_VM标志被设置），就把子进程插入父进程运行队列的队尾。
6. 如果CLONE_STOPPED标志被设置，则把子进程置为TASK_STOPPED状态。

注8： 当内核创建一个新进程时父进程有可能会被转移到另一个CPU上执行。

7. 如果父进程被跟踪，则把子进程的PID存入current的ptrace_message字段并调用ptrace_notify()。ptrace_notify()使当前进程停止运行，并向当前进程的父进程发送SIGCHLD信号。子进程的祖父进程是跟踪父进程的debugger进程。SIGCHLD信号通知debugger进程：current已经创建了一个子进程，可以通过查找current->ptrace_message字段获得子进程的PID。
8. 如果设置了CLONE_VFORK标志，则把父进程插入等待队列，并挂起父进程直到子进程释放自己的内存地址空间（也就是说，直到子进程结束或执行新的程序）。
9. 结束并返回子进程的PID。

copy_process()函数

copy_process()创建进程描述符以及子进程执行所需要的所有其他数据结构。它的参数与do_fork()的参数相同，外加子进程的PID。下面描述copy_process()的最重要的步骤：

1. 检查参数clone_flags所传递标志的一致性。尤其是，在下列情况下，它返回错误代号：
   a. CLONE_NEWNS和CLONE_FS标志都被设置。
   b. CLONE_THREAD标志被设置，但CLONE_SIGHAND标志被清0（同一线程组中的轻量级进程必须共享信号）。
   c. CLONE_SIGHAND标志被设置，但CLONE_VM被清0（共享信号处理程序的轻量级进程也必须共享内存描述符）。
2. 通过调用security_task_create()以及稍后调用的security_task_alloc()执行所有附加的安全检查。Linux 2.6提供扩展安全性的钩子函数，与传统Unix相比，它具有更加强壮的安全模型。详情参见第二十章。
3. 调用dup_task_struct()为子进程获取进程描述符。该函数执行如下操作：
   a. 如果需要，则在当前进程中调用__unlazy_fpu()，把FPU、MMX和SSE/SSE2寄存器的内容保存到父进程的thread_info结构中。稍后，dup_task_struct()将把这些值复制到子进程的thread_info结构中。
   b. 执行alloc_task_struct()宏，为新进程获取进程描述符（task_struct结构），并将描述符地址保存在tsk局部变量中。
   c. 执行alloc_thread_info宏以获取一块空闲内存区，用来存放新进程的thread_info结构和内核栈，并将这块内存区字段的地址存在局部变量ti中。正如在本章前面“标识一个进程”一节中所述：这块内存区字段的大小是8KB或4KB。

d. 将current进程描述符的内容复制到tsk所指向的task_struct结构中，然后把tsk->thread_info置为ti。

e. 把current进程的thread_info描述符的内容复制到ti所指向的结构中，然后把ti->task置为tsk。

f. 把新进程描述符的使用计数器（tsk->usage）置为2，用来表示进程描述符正在被使用而且其相应的进程处于活动状态（进程状态即不是EXIT_ZOMBIE，也不是EXIT_DEAD）。

g. 返回新进程的进程描述符指针（tsk）。

4. 检查存放在current->signal->rlim[RLIMIT_NPROC].rlim_cur变量中的值是否小于或等于用户所拥有的进程数。如果是，则返回错误码，除非进程没有root权限。该函数从每用户数据结构user_struct中获取用户所拥有的进程数。通过进程描述符user字段的指针可以找到这个数据结构。

5. 递增user_struct结构的使用计数器（tsk->user->__count字段）和用户所拥有的进程的计数器（tsk->user->processes）。

6. 检查系统中的进程数量（存放在nr_threads变量中）是否超过max_threads变量的值。这个变量的缺省值取决于系统内存容量的大小。总的原则是：所有thread_info描述符和内核栈所占用的空间不能超过物理内存大小的1/8。不过，系统管理员可以通过写*/proc/sys/kernel/threads-max*文件来改变这个值。

7. 如果实现新进程的执行域和可执行格式的内核函数（参见第二十章）都包含在内核模块中，则递增它们的使用计数器（参见附录二）。

8. 设置与进程状态相关的几个关键字段：

a. 把大内核锁计数器tsk->lock_depth初始化为-1（参见第五章“大内核锁”一节）。

b. 把tsk->did_exec字段初始化为0；它记录了进程发出的execve()系统调用的次数。

c. 更新从父进程复制到tsk->flags字段中的一些标志：首先清除PF_SUPERPRIV标志，该标志表示进程是否使用了某种超级用户权限。然后设置PF_FORKNOEXEC标志，它表示子进程还没有发出execve()系统调用。

9. 把新进程的PID存入tsk->pid字段。

10. 如果clone_flags参数中的CLONE_PARENT_SETTID标志被设置，就把子进程的PID复制到参数parent_tidptr指向的用户态变量中。

11. 初始化子进程描述符中的list_head数据结构和自旋锁，并为与挂起信号、定时器及时间统计表相关的几个字段赋初值。

12. 调用copy_semundo()，copy_files()，copy_fs()，copy_sighand()，copy_signal()，copy_mm()和copy_namespace()来创建新的数据结构，并把父进程相应数据结构的值复制到新数据结构中，除非clone_flags参数指出它们有不同的值。

13. 调用copy_thread()，用发出clone()系统调用时CPU寄存器的值（正如第十章所述，这些值已经被保存在父进程的内核栈中）来初始化子进程的内核栈。不过，copy_thread()把eax寄存器对应字段的值［这是fork()和clone()系统调用在子进程中的返回值］字段强行置为0。子进程描述符的thread.esp字段初始化为子进程内核栈的基地址，汇编语言函数（ret_from_fork()）的地址存放在thread.eip字段中。如果父进程使用I/O权限位图，则子进程获取该位图的一个拷贝。最后，如果CLONE_SETTLS标志被设置，则子进程获取由clone()系统调用的参数tls指向的用户态数据结构所表示的TLS段（注9）。

14. 如果clone_flags参数的值被置为CLONE_CHILD_SETTID或CLONE_CHILD_CLEARTID，就把child_tidptr参数的值分别复制到tsk->set_chid_tid 或tsk->clear_child_tid字段。这些标志说明：必须改变子进程用户态地址空间的child_tidptr所指向的变量的值，不过实际的写操作要稍后再执行。

15. 清除子进程thread_info结构的TIF_SYSCALL_TRACE标志，以使ret_from_fork()函数不会把系统调用结束的消息通知给调试进程（参见第十章“进入和退出系统调用”一节）。（因为对子进程的跟踪是由tsk->ptrace中的PTRACE_SYSCALL标志来控制的，所以子进程的系统调用跟踪不会被禁用。）

16. 用clone_flags参数低位的信号数字编码初始化tsk->exit_signal字段，如果CLONE_THREAD标志被置位，就把tsk->exit_signal字段初始化为-1。正如我们将在本章稍后“进程终止”一节所看见的，只有当线程组的最后一个成员（通常是线程组的领头）“死亡”，才会产生一个信号，以通知线程组的领头进程的父进程。

17. 调用sched_fork()完成对新进程调度程序数据结构的初始化。该函数把新进程的状态设置为TASK_RUNNING，并把thread_info结构的preempt_count字段设置为

注9：细心的读者可能想知道copy_thread()怎样获得clone()的tls参数的值，因为tls并不被传递给do_fork()和嵌套函数。我们将在第十章看到，通常通过拷贝系统调用的参数的值到某个CPU寄存器来把它们传递给内核；因此，这些值与其他寄存器一起被保存在内核态堆栈中。copy_thread()函数只查看esi的值在内核堆栈中对应的位置保存的地址。

1，从而禁止内核抢占（参见第五章“内核抢占”一节）。此外，为了保证公平的进程调度，该函数在父子进程之间共享父进程的时间片（参见第七章“scheduler_tick()函数”一节）。

18. 把新进程的thread_info结构的cpu字段设置为由smp_processor_id()所返回的本地CPU号。

19. 初始化表示亲子关系的字段。尤其是，如果CLONE_PARENT或CLONE_THREAD被设置，就用current->real_parent的值初始化 tsk->real_parent和tsk->parent，因此，子进程的父进程似乎是当前进程的父进程。否则，把tsk->real_parent和tsk->parent 置为当前进程。

20. 如果不需要跟踪子进程（没有设置CLONE_PTRAC标志），就把tsk->ptrace字段设置为0。tsk->ptrace字段会存放一些标志，而这些标志是在一个进程被另外一个进程跟踪时才会用到的。采用这种方式，即使当前进程被跟踪，子进程也不会被跟踪。

21. 执行SET_LINKS宏，把新进程描述符插入进程链表。

22. 如果子进程必须被跟踪（tsk->ptrace字段的PT_PTRACED标志被设置），就把current->parent赋给tsk->parent，并将子进程插入调试程序的跟踪链表中。

23. 调用attach_pid()把新进程描述符的PID插入pidhash[PIDTYPE_PID]散列表。

24. 如果子进程是线程组的领头进程（CLONE_THREAD标志被清0）：

    a. 把tsk->tgid的初值置为tsk->pid。

    b. 把tsk->group_leader的初值置为tsk。

    c. 调用三次attach_pid()，把子进程分别插入PIDTYPE_TGID、PIDTYPE_PGID和PIDTYPE_SID类型的PID散列表。

25. 否则，如果子进程属于它的父进程的线程组（CLONE_THREAD标志被设置）：

    a. 把tsk->tgid的初值置为tsk->current->tgid。

    b. 把tsk->group_leader的初值置为current->group_leader的值。

    c. 调用attach_pid()，把子进程插入PIDTYPE_TGID类型的散列表中（更具体地说，插入current->group_leader进程的每个PID链表）。

26. 现在，新进程已经被加入进程集合：递增nr_threads变量的值。

27. 递增total_forks变量以记录被创建的进程的数量。

28. 终止并返回子进程描述符指针（tsk）。

让我们回头看看在do_fork()结束之后都发生了什么。现在，我们有了处于可运行状态的完整的子进程。但是，它还没有实际运行，调度程序要决定何时把CPU交给这个子进程。在以后的进程切换中，调度程序继续完善子进程：把子进程描述符thread字段的值装入几个CPU寄存器。特别是把thread.esp（即把子进程内核态堆栈的地址）装入esp寄存器，把函数ret_from_fork()的地址装入eip寄存器。这个汇编语言函数调用schedule_tail()函数（它依次调用finish_task_switch()来完成进程切换，参见第七章“schedule()函数”一节），用存放在栈中的值再装载所有的寄存器，并强迫CPU返回到用户态。然后，在fork()、vfork()或clone()系统调用结束时，新进程将开始执行。系统调用的返回值放在eax寄存器中：返回给子进程的值是0，返回给父进程的值是子进程的PID。回顾copy_thread()对子进程的eax寄存器所执行的操作（copy_process()的第13步），就能理解这是如何实现的。

除非fork系统调用返回0，否则，子进程将与父进程执行相同的代码（参见copy_process()的第13步）。应用程序的开发者可以按照Unix编程者熟悉的方式利用这一事实，在基于PID值的程序中插入一个条件语句使子进程与父进程有不同的行为。

内核线程

传统的Unix系统把一些重要的任务委托给周期性执行的进程，这些任务包括刷新磁盘高速缓存，交换出不用的页框，维护网络连接等等。事实上，以严格线性的方式执行这些任务的确效率不高，如果把它们放在后台调度，不管是对它们的函数还是对终端用户进程都能得到较好的响应。因为一些系统进程只运行在内核态，所以现代操作系统把它们的函数委托给内核线程（*kernel thread*），内核线程不受不必要的用户态上下文的拖累。在Linux中，内核线程在以下几方面不同于普通进程：

- 内核线程只运行在内核态，而普通进程既可以运行在内核态，也可以运行在用户态。
- 因为内核线程只运行在内核态，它们只使用大于PAGE_OFFSET的线性地址空间。另一方面，不管在用户态还是在内核态，普通进程可以用4GB的线性地址空间。

创建一个内核线程

kernel_thread()函数创建一个新的内核线程，它接受的参数有：所要执行的内核函数的地址（fn）、要传递给函数的参数（arg）、一组clone标志（flags）。该函数本质上以下面的方式调用do_fork()：

```
do_fork(flags|CLONE_VM|CLONE_UNTRACED, 0, pregs, 0, NULL, NULL);
```

CLONE_VM标志避免复制调用进程的页表：由于新内核线程无论如何都不会访问用户态地址空间，所以这种复制无疑会造成时间和空间的浪费。CLONE_UNTRACED标志保证不会有任何进程跟踪新内核线程，即使调用进程被跟踪。

传递给do_fork()的参数pregs表示内核栈的地址，copy_thread()函数将从这里找到为新线程初始化CPU寄存器的值。kernel_thread()函数在这个栈中保留寄存器值的目的是：

- 通过copy_thread()把ebx和edx分别设置为参数fn和arg的值。
- 把eip寄存器的值设置为下面汇编语言代码段的地址：

```
movl %edx,%eax
pushl %edx
call *%ebx
pushl %eax
call do_exit
```

因此，新的内核线程开始执行fn(arg)函数，如果该函数结束，内核线程执行系统调用_exit()，并把fn()的返回值传递给它（参见本章稍后“撤消进程”一节）。

进程0

所有进程的祖先叫做进程0，*idle*进程 或因为历史的原因叫做*swapper*进程，它是在Linux的初始化阶段从无到有创建的一个内核线程（参见附录一）。这个祖先进程使用下列静态分配的数据结构（所有其他进程的数据结构都是动态分配的）：

- 存放在init_task变量中的进程描述符，由INIT_TASK宏完成对它的初始化。
- 存放在init_thread_union变量中的thread_info描述符和内核堆栈，由INIT_THREAD_INFO宏完成对它们的初始化。
- 由进程描述符指向的下列表：
  - init_mm
  - init_fs
  - init_files
  - init_signals
  - init_sighand

  这些表分别由下列宏初始化：
  - INIT_MM

— INIT_FS

— INIT_FILES

— INIT_SIGNALS

— INIT_SIGHAND

- 主内核页全局目录存放在 swapper_pg_dir 中（参见第二章“内核页表”一节）。

start_kernel()函数初始化内核需要的所有数据结构，激活中断，创建另一个叫进程1的内核线程（一般叫做 *init* 进程）：

```
kernel_thread(init, NULL, CLONE_FS|CLONE_SIGHAND);
```

新创建内核线程的PID为1，并与进程0共享每进程所有的内核数据结构。此外，当调度程序选择到它时，*init* 进程开始执行 init()函数。

创建 *init* 进程后，进程0执行 cpu_idle()函数，该函数本质上是在开中断的情况下重复执行 hlt 汇编语言指令（参见第四章）。只有当没有其他进程处于 TASK_RUNNING 状态时，调度程序才选择进程0。

在多处理器系统中，每个CPU都有一个进程0。只要打开机器电源，计算机的BIOS就启动某一个CPU，同时禁用其他CPU。运行在CPU 0上的swapper进程初始化内核数据结构，然后激活其他的CPU，并通过 copy_process()函数创建另外的swapper进程，把0传递给新创建的swapper进程作为它们的新PID。此外，内核把适当的CPU索引赋给内核所创建的每个进程的 thread_info 描述符的cpu字段。

进程 1

由进程0创建的内核线程执行 init()函数，init()依次完成内核初始化。init()调用 execve()系统调用装入可执行程序 *init*。结果，*init* 内核线程变为一个普通进程，且拥有自己的每进程（per-process）内核数据结构（参见第二十章）。在系统关闭之前，*init* 进程一直存活，因为它创建和监控在操作系统外层执行的所有进程的活动。

其他内核线程

Linux使用很多其他内核线程。其中一些在初始化阶段创建，一直运行到系统关闭；而其他一些在内核必须执行一个任务时“按需”创建，这种任务在内核的执行上下文中得到很好的执行。

一些内核线程的例子（除了进程0和进程1）是：

keventd（也被称为事件）

执行keventd_wq工作队列（参见第四章）中的函数。

kapmd

处理与高级电源管理(APM)相关的事件。

kswapd

执行内存回收，在第十七章“周期回收”一节将进行描述。

pdflush

刷新“脏”缓冲区中的内容到磁盘以回收内存，在第十五章“pdflush内核线程”一节将进行描述。

kblockd

执行kblockd_workqueue工作队列中的函数。实质上，它周期性地激活块设备驱动程序，将在第十四章“激活块设备驱动程序”一节给予描述。

ksoftirqd

运行tasklet（参看第四章“软中断及tasklet”一节）；系统中每个CPU都有这样一个内核线程。

撤消进程

很多进程终止了它们本该执行的代码，从这种意义上说，这些进程“死”了。当这种情况发生时，必须通知内核以便内核释放进程所拥有的资源，包括内存、打开文件及其他我们在本书中讲到的零碎东西，如信号量。

进程终止的一般方式是调用exit()库函数，该函数释放C函数库所分配的资源，执行编程者所注册的每个函数，并结束从系统回收进程的那个系统调用。exit()函数可能由编程者显式地插入。另外，C编译程序总是把exit()函数插入到main()函数的最后一条语句之后。

内核可以有选择地强迫整个线程组死掉。这发生在以下两种典型情况下：当进程接收到一个不能处理或忽视的信号时（参见十一章），或者当内核正在代表进程运行时在内核态产生一个不可恢复的CPU异常时（参见第四章）。

进程终止

在Linux 2.6中有两个终止用户态应用的系统调用：

- exit_group()系统调用，它终止整个线程组，即整个基于多线程的应用。do_group_exit()是实现这个系统调用的主要内核函数。这是C库函数exit()应该调用的系统调用。
- exit()系统调用，它终止某一个线程，而不管该线程所属线程组中的所有其他进程。do_exit()是实现这个系统调用的主要内核函数。这是被诸如pthread_exit()的Linux线程库的函数所调用的系统调用。

do_group_exit()函数

do_group_exit()函数杀死属于current线程组的所有进程。它接受进程终止代号作为参数，进程终止代号可能是系统调用exit_group()(正常结束）指定的一个值，也可能是内核提供的一个错误代号（异常结束）。该函数执行下述操作：

1. 检查退出进程的SIGNAL_GROUP_EXIT标志是否不为0，如果不为0，说明内核已经开始为线程组执行退出的过程。在这种情况下，就把存放在current->signal->group_exit_code中的值当作退出码，然后跳转到第4步。
2. 否则，设置进程的SIGNAL_GROUP_EXIT标志并把终止代号存放到current->signal->group_exit_code字段。
3. 调用zap_other_threads()函数杀死current线程组中的其他进程(如果有的话)。为了完成这个步骤，函数扫描与current->tgid对应的PIDTYPE_TGID类型的散列表中的每个PID链表，向表中所有不同于current的进程发送SIGKILL信号(参见第十一章)，结果，所有这样的进程都将执行do_exit()函数，从而被杀死。
4. 调用do_exit()函数，把进程的终止代号传递给它。正如我们将在下面看到的，do_exit()杀死进程而且不再返回。

do_exit()函数

所有进程的终止都是由do_exit()函数来处理的，这个函数从内核数据结构中删除对终止进程的大部分引用。do_exit()函数接受进程的终止代号作为参数并执行下列操作：

1. 把进程描述符的flag字段设置为PF_EXITING标志，以表示进程正在被删除。
2. 如果需要，通过函数del_timer_sync()(参见第六章）从动态定时器队列中删除进程描述符。
3. 分别调用exit_mm()、exit_sem()、__exit_files()、__exit_fs()、exit_namespace()和exit_thread()函数从进程描述符中分离出与分页、信号量、文件系统、打开文件描述符、命名空间以及I/O权限位图相关的数据结构。如果没有其他进程共享这些数据结构，那么这些函数还删除所有这些数据结构中。

4. 如果实现了被杀死进程的执行域和可执行格式(参见第二十章)的内核函数包含在内核模块中，则函数递减它们的使用计数器。

5. 把进程描述符的exit_code字段设置成进程的终止代号，这个值要么是_exit()或exit_group()系统调用参数（正常终止），要么是由内核提供的一个错误代号（异常终止）。

6. 调用exit_notify()函数执行下面的操作：

   a. 更新父进程和子进程的亲属关系。如果同一线程组中有正在运行的进程，就让终止进程所创建的所有子进程都变成同一线程组中另外一个进程的子进程，否则让它们成为*init*的子进程。

   b. 检查被终止进程其进程描述符的exit_signal字段是否不等于-1，并检查进程是否是其所属进程组的最后一个成员（注意：正常进程都会具有这些条件，参见前面“clone()、fork()和vfork()系统调用”一节中对copy_process()的描述，第16步）。在这种情况下，函数通过给正被终止进程的父进程发送一个信号（通常是SIGCHLD），以通知父进程子进程死亡。

   c. 否则，也就是exit_signal字段等于-1，或者线程组中还有其他进程，那么只要进程正在被跟踪，就向父进程发送一个SIGCHLD信号（在这种情况下，父进程是调试程序，因而，向它报告轻量级进程死亡的信息）。

   d. 如果进程描述符的exit_signal字段等于-1，而且进程没有被跟踪，就把进程描述符的exit_state字段置为EXIT_DEAD，然后调用release_task()回收进程的其他数据结构占用的内存，并递减进程描述符的使用计数器（见下一节）。使用记数器变为1（参见copy_process()函数的第3f步），以使进程描述符本身正好不会被释放。

   e. 否则，如果进程描述符的exit_signal字段不等于-1，或进程正在被跟踪，就把exit_state字段置为EXIT_ZOMBIE。在下一节我们将看到如何处理僵死进程。

   f. 把进程描述符的flags字段设置为PF_DEAD标志（参见第七章“schedule()函数”一节）。

7. 调用schedule()函数（参见第七章）选择一个新进程运行。调度程序忽略处于EXIT_ZOMBIE状态的进程，所以这种进程正好在schedule()中的宏switch_to被调用之后停止执行。正如在第七章我们将看到的：调度程序将检查被替换的僵死进程描述符的PF_DEAD标志并递减使用计数器，从而说明进程不再存活的事实。

进程删除

Unix允许进程查询内核以获得其父进程的PID,或者其任何子进程的执行状态。例如,进程可以创建一个子进程来执行特定的任务,然后调用诸如wait()这样的一些库函数检查子进程是否终止。如果子进程已经终止,那么,它的终止代号将告诉父进程这个任务是否已成功地完成。

为了遵循这些设计选择,不允许Unix内核在进程一终止后就丢弃包含在进程描述符字段中的数据。只有父进程发出了与被终止的进程相关的wait()类系统调用之后,才允许这样做。这就是引入僵死状态的原因:尽管从技术上来说进程已死,但必须保存它的描述符,直到父进程得到通知。

如果父进程在子进程结束之前结束会发生什么情况呢?在这种情况下,系统中会到处是僵死的进程,而且它们的进程描述符永久占据着RAM。如前所述,必须强迫所有的孤儿进程成为*init*进程的子进程来解决这个问题。这样,*init*进程在用wait()类系统调用检查其合法的子进程终止时,就会撤消僵死的进程。

release_task()函数从僵死进程的描述符中分离出最后的数据结构;对僵死进程的处理有两种可能的方式:如果父进程不需要接收来自子进程的信号,就调用do_exit();如果已经给父进程发送了一个信号,就调用wait4()或waitpid()系统调用。在后一种情况下,函数还将回收进程描述符所占用的内存空间,而在前一种情况下,内存的回收将由进程调度程序来完成(参见第七章)。该函数执行下述步骤:

1. 递减终止进程拥有者的进程个数。这个值存放在本章前面提到的user_struct结构中(参见copy_process()的第4步)。
2. 如果进程正在被跟踪,函数将它从调试程序的ptrace_children链表中删除,并让该进程重新属于初始的父进程。
3. 调用__exit_signal()删除所有的挂起信号并释放进程的signal_struct描述符。如果该描述符不再被其他的轻量级进程使用,函数进一步删除这个数据结构。此外,函数调用exit_itimers()从进程中剥离掉所有的POSIX时间间隔定时器。
4. 调用__exit_sighand()删除信号处理函数。
5. 调用__unhash_process(),该函数依次执行下面的操作:
   a. 变量nr_threads减1。
   b. 两次调用detach_pid(),分别从PIDTYPE_PID和PIDTYPE_TGID类型的PID散列表中删除进程描述符。

c. 如果进程是线程组的领头进程，那么再调用两次detach_pid()，从PIDTYPE_PGID和PIDTYPE_SID类型的散列表中删除进程描述符。

d. 用宏REMOVE_LINKS从进程链表中解除进程描述符的链接。

6. 如果进程不是线程组的领头进程，领头进程处于僵死状态，而且进程是线程组的最后一个成员，则该函数向领头进程的父进程发送一个信号，通知它进程已死亡。

7. 调用sched_exit()函数来调整父进程的时间片（这一步在逻辑上作为对copy_process()第17步的补充）。

8. 调用put_task_struct()递减进程描述符的使用计数器，如果计数器变为0，则函数终止所有残留的对进程的引用。

a. 递减进程所有者的user_struct数据结构的使用计数器（__count字段）（参见copy_process()的第5步），如果使用计数器变为0，就释放该数据结构。

b. 释放进程描述符以及thread_info描述符和内核态堆栈所占用的内存区域。



第四章

中断和异常

中断(*interrupt*)通常被定义为一个事件，该事件改变处理器执行的指令顺序。这样的事件与CPU芯片内外部硬件电路产生的电信号相对应。

中断通常分为同步（*synchronous*）中断和异步（*asynchronous*）中断：

- 同步中断是当指令执行时由CPU控制单元产生的，之所以称为同步，是因为只有在一条指令终止执行后CPU才会发出中断。
- 异步中断是由其他硬件设备依照CPU时钟信号随机产生的。

在Intel微处理器手册中，把同步和异步中断分别称为异常(*exception*)和中断(*interrupt*)。我们也采用这种分类，当然有时我们也用术语“中断信号”指这两种类型（同步及异步）。

中断是由间隔定时器和I/O设备产生的，例如，用户的一次按键会引起一个中断。

另一方面，异常是由程序的错误产生的，或者是由内核必须处理的异常条件产生的。第一种情况下，内核通过发送一个每个Unix程序员都熟悉的信号来处理异常。第二种情况下，内核执行恢复异常需要的所有步骤，例如缺页，或对内核服务的一个请求（通过一条`int`或`sysenter`指令）。

我们在下一节描述引入信号的动机，以此开始进行学习。然后，说明由I/O设备产生的著名IRQ（Interrupt ReQuest）如何引起中断，我们将详细讨论80x86微处理器如何在硬件级处理中断和异常。接下来，我们将在“初始化中断描述符表”一节阐明Linux如何初始化Intel中断结构必需的所有数据结构。剩余的3节描述Linux如何在软件级处理中断信号。

继续进行学习之前，需要值得注意的是：我们在本章仅涉及对所有PC都通用的“经典”中断，而不涉及一些体系结构的非标准中断。

中断信号的作用

顾名思义，中断信号提供了一种特殊的方式，使处理器转而去运行正常控制流之外的代码。当一个中断信号达到时，CPU必须停止它当前正在做的事情，并且切换到一个新的活动。为了做到这一点，就要在内核态堆栈保存程序计数器的当前值（即eip和cs寄存器的内容），并把与中断类型相关的一个地址放进程序计数器。

在本章，有些事情会使你想起在前一章描述的上下文切换，这发生在内核用一个进程替换另一个进程时。但是，中断处理与进程切换有一个明显的差异：由中断或异常处理程序执行的代码不是一个进程。更确切地说，它是一个内核控制路径，代表中断发生时正在运行的进程执行（参见本章“中断和异常处理程序的嵌套执行”一节）。作为一个内核控制路径，中断处理程序比一个进程要“轻”（light）（中断的上下文很少，建立或终止中断处理需要的时间很少）。

中断处理是由内核执行的最敏感的任务之一，因为它必须满足下列约束：

- 当内核正打算去完成一些别的事情时，中断随时会到来。因此，内核的目标就是让中断尽可能快地处理完，尽其所能把更多的处理向后推迟。例如，假设一个数据块已到达了网线，当硬件中断内核时，内核只简单地标志数据到来了，让处理器恢复到它以前运行的状态，其余的处理稍后再进行（如把数据移入一个缓冲区，它的接收进程可以在缓冲区找到数据并恢复这个进程的执行）。因此，内核响应中断后需要进行的操作分为两部分：关键而紧急的部分，内核立即执行；其余推迟的部分，内核随后执行。
- 因为中断随时会到来，所以内核可能正在处理其中的一个中断时，另一个中断（不同类型）又发生了。应该尽可能多地允许这种情况发生，因为这能维持更多的I/O设备处于忙状态（参见“中断和异常处理程序的嵌套执行”一节）。因此，中断处理程序必须编写成使相应的内核控制路径能以嵌套的方式执行。当最后一个内核控制路径终止时，内核必须能恢复被中断进程的执行，或者，如果中断信号已导致了重新调度，内核能切换到另外的进程。
- 尽管内核在处理前一个中断时可以接受一个新的中断，但在内核代码中还是存在一些临界区，在临界区中，中断必须被禁止。必须尽可能地限制这样的临界区，因为根据以前的要求，内核，尤其是中断处理程序，应该在大部分时间内以开中断的方式运行。

中断和异常

Intel 文档把中断和异常分为以下几类：

- 中断：

  可屏蔽中断（maskable interrupt）

  I/O设备发出的所有中断请求（IRQ）都产生可屏蔽中断。可屏蔽中断可以处于两种状态：屏蔽的（masked）或非屏蔽的（unmasked）；一个屏蔽的中断只要还是屏蔽的，控制单元就忽略它。

  非屏蔽中断（nonmaskable Interrupt）

  只有几个危急事件（如硬件故障）才引起非屏蔽中断。非屏蔽中断总是由CPU辨认。

- 异常：

  处理器探测异常（processor-detected exception）

  当CPU执行指令时探测到的一个反常条件所产生的异常。可以进一步分为三组，这取决于CPU控制单元产生异常时保存在内核态堆栈eip寄存器中的值。

  故障（fault）

  通常可以纠正；一旦纠正，程序就可以在不失连贯性的情况下重新开始。保存在eip中的值是引起故障的指令地址。因此，当异常处理程序终止时，那条指令会被重新执行。我们将在第九章的“缺页异常处理程序”一节中看到，只要处理程序能纠正引起异常的反常条件，重新执行同一指令就是必要的。

  陷阱（trap）

  在陷阱指令执行后立即报告；内核把控制权返回给程序后就可以继续它的执行而不失连贯性。保存在eip中的值是一个随后要执行的指令地址。只有当没有必要重新执行已终止的指令时，才触发陷阱。陷阱的主要用途是为了调试程序。在这种情况下，中断信号的作用是通知调试程序一条特殊指令已被执行（例如到了一个程序内的断点）。一旦用户检查到调试程序所提供的数据，她就可能要求被调试程序从下一条指令重新开始执行。

  异常中止（abort）

  发生一个严重的错误，控制单元出了问题，不能在eip寄存器中保存引起异常的指令所在的确切位置。异常中止用于报告严重的错误，如硬件故障或系统表中无效的值或不一致的值。由控制单元发送的这个中断信号是紧急信号，用来把控制权切换到相应的异常中止处理程序，这个异常中止处理程序除了强制受影响的进程终止外，没有别的选择。

编程异常（programmed exception）

在编程者发出请求时发生。是由 int 或 int3 指令触发的；当 into(检查溢出)和 bound（检查地址出界）指令检查的条件不为真时，也引起编程异常。控制单元把编程异常作为陷阱来处理。编程异常通常也叫做软中断(*software interrupt*)。这样的异常有两种常用的用途：执行系统调用及给调试程序通报一个特定的事件（参见第十章）。

每个中断和异常是由0～255之间的一个数来标识。因为一些未知的原因，Intel把这个8位的无符号整数叫做一个向量（*vector*）。非屏蔽中断的向量和异常的向量是固定的，而可屏蔽中断的向量可以通过对中断控制器的编程来改变（参见下一节）。

IRQ和中断

每个能够发出中断请求的硬件设备控制器都有一条名为*IRQ*（Interrupt ReQuest）的输出线（注1）。所有现有的IRQ线（*IRQ* line）都与一个名为可编程中断控制器（*Programmable Interrpt Controuer*,PIC）的硬件电路的输入引脚相连。可编程中断控制器执行下列动作：

1. 监视IRQ线，检查产生的信号（raised signal）。如果有条或两条以上的IRQ线上产生信号，就选择引脚编号较小的IRQ线。
2. 如果一个引发信号出现在IRQ线上：
   a. 把接收到的引发信号转换成对应的向量。
   b. 把这个向量存放在中断控制器的一个I/O端口，从而允许CPU通过数据总线读此向量。
   c. 把引发信号发送到处理器的INTR引脚，即产生一个中断。
   d. 等待，直到CPU通过把这个中断信号写进可编程中断控制器的一个I/O端口来确认它；当这种情况发生时，清INTR线。
3. 返回到第1步。

IRQ线是从0开始顺序编号的，因此，第一条IRQ线通常表示成IRQ0。与IRQ*n*关联的Intel的缺省向量是*n*+32。如前所述，通过向中断控制器端口发布合适的指令，就可以修改IRQ和向量之间的映射。

可以有选择地禁止每条IRQ线。因此，可以对PIC编程从而禁止IRQ，也就是说，可以告诉PIC停止对给定的IRQ线发布中断，或者激活它们。禁止的中断是丢失不了的，它

注1： 复杂一些的设备有几条IRQ线，例如，PCI卡可能使用多达4条IRQ线。

们一旦被激活，PIC就又把它们发送到CPU。这个特点被大多数中断处理程序使用，因为这允许中断处理程序逐次地处理同一类型的IRQ。

有选择地激活/禁止IRQ线不同于可屏蔽中断的全局屏蔽/非屏蔽。当eflags寄存器的IF标志被清0时，由PIC发布的每个可屏蔽中断都由CPU暂时忽略。cli和sti汇编指令分别清除和设置该标志。

传统的PIC是由两片8259A风格的外部芯片以“级联”的方式连接在一起的。每个芯片可以处理多达8个不同的IRQ输入线。因为从PIC的INT输出线连接到主PIC的IRQ2引脚，因此，可用IRQ线的个数限制为15。

高级可编程中断控制器

以前的描述仅涉及为单处理器系统设计的PIC。如果系统只有一个单独的CPU，那么主PIC的输出线以直截了当的方式连接到CPU的INTR引脚。然而，如果系统中包含两个或多个CPU，那么这种方式不再有效，因而需要更复杂的PIC。

为了充分发挥SMP体系结构的并行性，能够把中断传递给系统中的每个CPU至关重要。基于此理由，Intel从Pentiun III开始引入了一种名为I/O高级可编程控制器（*I/OAdvanced Programmable Interrupt Controller,I/O APIC*）的新组件，用以代替老式的8259A可编程中断控制器。新近的主板为了支持以前的操作系统都包括两种芯片。此外，80x86微处理器当前所有的CPU都含有一个本地APIC。每个本地APIC都有32位的寄存器、一个内部时钟、一个本地定时设备及为本地APIC中断保留的两条额外的IRQ线LINT0和LINT1。所有本地APIC都连接到一个外部I/O APIC，形成一个多APIC的系统。

图4-1以示意图的方式显示了一个多APIC系统的结构。一条APIC总线把“前端”I/O APIC连接到本地APIC。来自设备的IRQ线连接到I/O APIC，因此，相对于本地APIC，I/O APIC起路由器的作用。在Pentium III和早期处理器的母板上，APIC总线是一个串行三线总线；从Pentium 4开始，APIC总线通过系统总线来实现。不过，因为APIC总线及其信息对软件是不可见的，因此，我们不做进一步的详细讨论。

I/O APIC的组成为：一组24条IRQ线、一张24项的中断重定向表（*Interrupt Redirection Table*）、可编程寄存器，以及通过APIC总线发送和接收APIC信息的一个信息单元。与8259A的IRQ引脚不同，中断优先级并不与引脚号相关联：中断重定向表中的每一项都可以被单独编程以指明中断向量和优先级、目标处理器及选择处理器的方式。重定向表中的信息用于把每个外部IRQ信号转换为一条消息，然后，通过APIC总线把消息发送给一个或多个本地APIC单元。

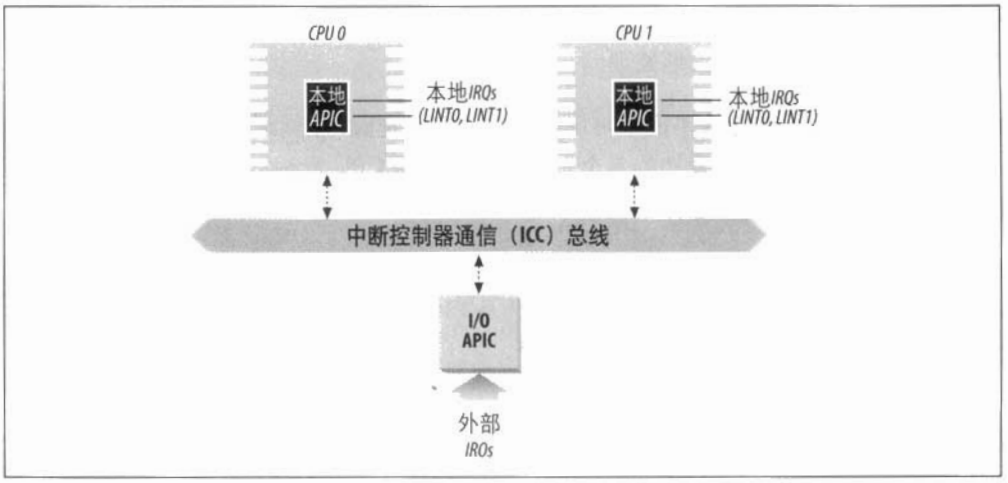


图4-1：多APIC系统

来自外部硬件设备的中断请求以两种方式在可用CPU之间分发：

静态分发

IRQ信号传递给重定向表相应项中所列出的本地APIC。中断立即传递给一个特定的CPU，或一组CPU，或所有CPU（广播方式）。

动态分发

如果处理器正在执行最低优先级的进程，IRQ信号就传递给这种处理器的本地APIC。

每个本地APIC都有一个可编程任务优先级寄存器（*task priority register*，TPR），TPR用来计算当前运行进程的优先级。Intel希望在操作系统内核中通过每次进程切换对这个寄存器进行修改。

如果两个或多个CPU共享最低优先级，就利用仲裁（*arbitration*）技术在这些CPU之间分配负荷。在本地APIC的仲裁优先级寄存器中，给每个CPU都分配一个0（最低）~15（最高）范围内的值。

每当中断传递给一个CPU时，其相应的仲裁优先级就自动置为0，而其他每个CPU的仲裁优先级都增加1。当仲裁优先级寄存器大于15时，就把它置为获胜CPU的前一个仲裁优先级加1。因此，中断以轮转方式在CPU之间分发，且具有相同的任务优先级（注2）。

注2：Pentium 4本地APIC没有仲裁优先级寄存器；仲裁机制隐藏在总线仲裁电路中。Intel手册中声明，如果操作系统内核不能有规律地更新任务优先级寄存器，那么性能可能就达不到最优，因为中断有可能总是由同一个CPU处理。

除了在处理器之间分发中断外，多APIC系统还允许CPU产生处理器间中断(*interprocessor interrupt*)。当一个CPU希望把中断发给另一个CPU时，它就在自己本地APIC的中断指令寄存器（Interrupt Command Register，ICR)中存放这个中断向量和目标本地APIC的标识符。然后，通过APIC总线向目标本地APIC发送一条消息，从而向自己的CPU发出一个相应的中断。

处理器间中断（简称IPI）是SMP体系结构至关重要的组成部分，并由Linux有效地用来在CPU之间交换信息（参见本章后面)。

目前大部分单处理器系统都包含一个I/O APIC芯片，可以用以下两种方式对这种芯片进行配置：

- 作为一种标准8259A方式的外部PIC连接到CPU。本地APIC被禁止，两条LINT0和LINT1本地IRQ线分别配置为INTR和NMI引脚。
- 作为一种标准外部I/O APIC。本地APIC被激活，且所有的外部中断都通过I/O APIC接收。

异常

80x86微处理器发布了大约20种不同的异常（注3)。内核必须为每种异常提供一个专门的异常处理程序。对于某些异常，CPU控制单元在开始执行异常处理程序前会产生一个硬件出错码（*hardware error code*)，并且压入内核态堆栈。

下面的列表给出了在80x86处理器中可以找到的异常的向量、名字、类型及其简单描述。更多的信息可以在Intel的技术文挡中找到。

0 - "Divide error"（故障)
当一个程序试图执行整数被0除操作时产生。

1- "Debug"（陷阱或故障)
产生于：①设置eflags的TF标志时（对于实现调试程序的单步执行是相当有用的)，②一条指令或操作数的地址落在一个活动debug寄存器的范围之内（参见第三章的"硬件上下文"一节)。

2 - 未用
为非屏蔽中断保留（利用NMI引脚的那些中断)。

3 - "Breakpoint"（陷阱)
由int3（断点）指令（通常由debugger插入）引起。

注3： 精确的数字依赖于处理器模型。

4 - *"Overflow"（陷阱）*

当eflags的OF (overflow)标志被设置时，into(检查溢出)指令被执行。

5 - *"Bounds check"（故障）*

对于有效地址范围之外的操作数，bound(检查地址边界)指令被执行。

6 - *"Invalid opcode"（故障）*

CPU执行单元检测到一个无效的操作码（决定执行操作的机器指令部分）。

7- *"Device not available"（故障）*

随着cr0的TS标志被设置，ESCAPE、MMX或XMM指令被执行（参见第三章的"保存和加载FPU、MMX及XMM寄存器"一节）。

8 - *"Double fault"（异常中止）*

正常情况下，当CPU正试图为前一个异常调用处理程序时，同时又检测到一个异常，两个异常能被串行地处理。然而，在少数情况下，处理器不能串行地处理它们，因而产生这种异常。

9 - *"Coprocessor segment overrun"（异常中止）*

因外部的数学协处理器引起的问题（仅用于80386微处理器）。

10 - *"Invalid TSS"（故障）*

CPU试图让一个上下文切换到有无效的TSS的进程。

11 - *"Segment not present"（故障）*

引用一个不存在的内存段（段描述符的Segment-Present标志被清0）。

12 - *"Stack segment fault"（故障）*

试图超过栈段界限的指令，或者由ss标识的段不在内存。

13 - *"General protection"（故障）*

违反了80x86保护模式下的保护规则之一。

14 - *"Page fault"（故障）*

寻址的页不在内存，相应的页表项为空，或者违反了一种分页保护机制。

15 - *由Intel保留*

16 - *"Floating point error"（故障）*

集成到CPU芯片中的浮点单元用信号通知一个错误情形，如数字溢出，或被0除（注4）。

注4： 80x86微处理器也产生这个异常，这发生在执行一个带符号的除法运算，而运算结果不能以带符号整数存放的时候(例如-2 147 483 648到-1之间的一个除法运算)。

17 - "Alignment check" (故障)
操作数的地址没有被正确地对齐（例如，一个长整数的地址不是4的倍数）。

18 - "Machine check" (异常中止)
机器检查机制检测到一个CPU错误或总线错误。

19 - "SIMD floating point exception" (故障)
集成到CPU芯片中的SSE或SSE2单元对浮点操作用信号通知一个错误情形。

20~31这些值由Intel留作将来开发。如表4-1所示，每个异常都由专门的异常处理程序来处理（参见本章后面的"异常处理"一节），它们通常把一个Unix信号发送到引起异常的进程。

表4-1：由异常处理程序发送的信号

编号	异常	异常处理程序	信号
0	Divide error	divide_error()	SIGFPE
1	Debug	debug()	SIGTRAP
2	NMI	nmi()	None
3	Breakpoint	int3()	SIGTRAP
4	Overflow	overflow()	SIGSEGV
5	Bounds check	bounds()	SIGSEGV
6	Invalid opcode	invalid_op()	SIGILL
7	Device not available	device_not_available()	None
8	Double fault	doublefault_fn()	None
9	Coprocessor segment overrun	coprocessor_segment_overrun()	SIGFPE
10	Invalid TSS	invalid_tss()	SIGSEGV
11	Segment not present	segment_not_present()	SIGBUS
12	Stack exception	stack_segment()	SIGBUS
13	General protection	general_protection()	SIGSEGV
14	Page fault	page_fault()	SIGSEGV
15	Intel reserved	None	None
16	Floating point error	coprocessor_error()	SIGFPE
17	Alignment check	alignment_check()	SIGSEGV
18	Machine check	machine_check()	None
19	SIMD floating point	simd_coprocessor_error()	SIGFPE

中断描述符表

中断描述符表（*Interrupt Descriptor Table*，IDT）是一个系统表，它与每一个中断或异常向量相联系，每一个向量在表中有相应的中断或异常处理程序的入口地址。内核在允许中断发生前，必须适当地初始化IDT。

在第二章中，我们介绍了GDT和LDT，IDT的格式与这两种表的格式非常相似，表中的每一项对应一个中断或异常向量，每个向量由8个字节组成。因此，最多需要256 × 8=2048字节来存放IDT。

`idtr` CPU寄存器使IDT可以位于内存的任何地方，它指定IDT的线性基地址及其限制（最大长度）。在允许中断之前，必须用`lidt`汇编指令初始化`idtr`。

IDT包含三种类型的描述符，图4-2显示了每种描述符中的64位的含义。尤其值得注意的是，在40~43位的`Type`字段的值表示描述符的类型。

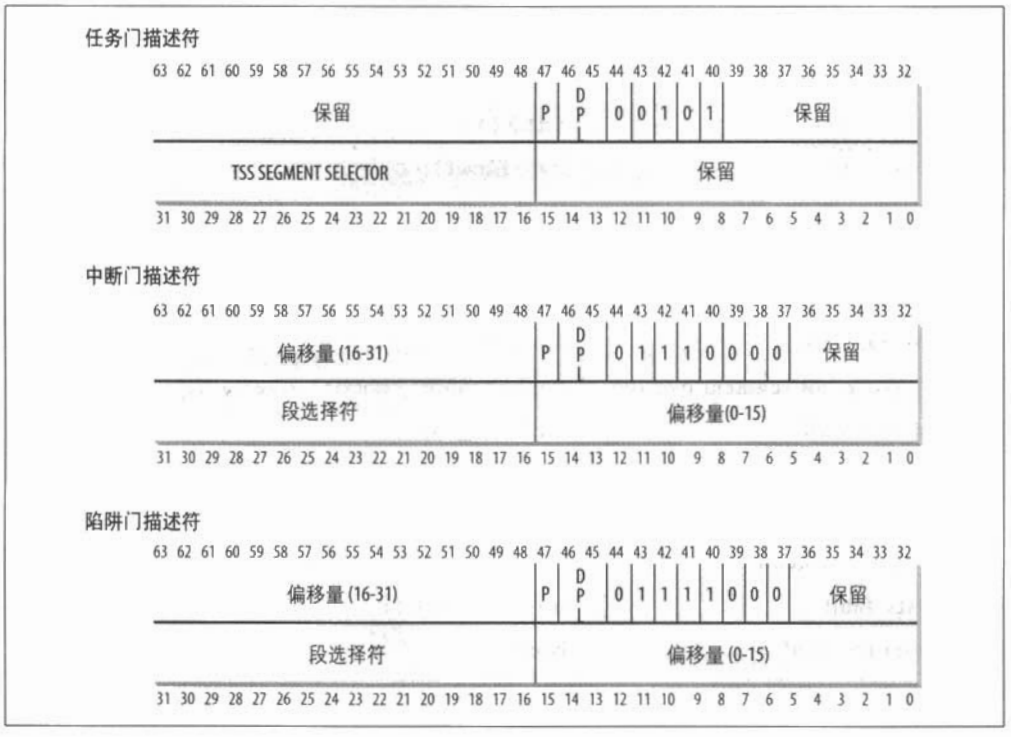


图4-2：门描述符的格式

这些描述符是：

任务门（task gate）

当中断信号发生时，必须取代当前进程的那个进程的TSS选择符存放在任务门中。

中断门（interrupt gate）

包含段选择符和中断或异常处理程序的段内偏移量。当控制权转移到一个适当的段时，处理器清IF标志，从而关闭将来会发生的可屏蔽中断。

陷阱门（Trap gate）

与中断门相似，只是控制权传递到一个适当的段时处理器不修改IF标志。

正如我们将在“中断门、陷阱门及系统门”一节中所看到的那样，Linux利用中断门处理中断，利用陷阱门处理异常（注5）。

中断和异常的硬件处理

我们现在描述CPU控制单元如何处理中断和异常。我们假定内核已被初始化，因此，CPU在保护模式下运行。

当执行了一条指令后，cs和eip这对寄存器包含下一条将要执行的指令的逻辑地址。在处理那条指令之前，控制单元会检查在运行前一条指令时是否已经发生了一个中断或异常。如果发生了一个中断或异常，那么控制单元执行下列操作：

1. 确定与中断或异常关联的向量i ($0 \leqslant i \leqslant 255$)。
2. 读由idtr寄存器指向的IDT表中的第i项（在下面的描述中，我们假定IDT表项中包含的是一个中断门或一个陷阱门）。
3. 从gdtr寄存器获得GDT的基地址，并在GDT中查找，以读取IDT表项中的选择符所标识的段描述符。这个描述符指定中断或异常处理程序所在段的基地址。
4. 确信中断是由授权的（中断）发生源发出的。首先将当前特权级CPL（存放在cs寄存器的低两位）与段描述符（存放在GDT中）的描述符特权级DPL比较，如果CPL小于DPL，就产生一个“General protection”异常，因为中断处理程序的特权不能低于引起中断的程序的特权。对于编程异常，则做进一步的安全检查：比较CPL与处于IDT中的门描述符的DPL，如果DPL小于CPL，就产生一个“General protection”异常。这最后一个检查可以避免用户应用程序访问特殊的陷阱门或中断门。

注5：“Double fault”异常是唯一由任务门处理的异常，它表示一种内核错误（参见本章稍后“异常处理”一节）。

5. 检查是否发生了特权级的变化，也就是说，CPL是否不同于所选择的段描述符的DPL。如果是，控制单元必须开始使用与新的特权级相关的栈。通过执行以下步骤来做到这点：
   a. 读tr寄存器，以访问运行进程的TSS段。
   b. 用与新特权级相关的栈段和栈指针的正确值装载ss和esp寄存器。这些值可以在TSS中找到（参见第三章的“任务状态段”一节）
   c. 在新的栈中保存ss和esp以前的值，这些值定义了与旧特权级相关的栈的逻辑地址。
6. 如果故障已发生，用引起异常的指令地址装载cs和eip寄存器，从而使得这条指令能再次被执行。
7. 在栈中保存eflags、cs及eip的内容。
8. 如果异常产生了一个硬件出错码，则将它保存在栈中。
9. 装载cs和eip寄存器，其值分别是IDT表中第*i*项门描述符的段选择符和偏移量字段。这些值给出了中断或者异常处理程序的第一条指令的逻辑地址。

控制单元所执行的最后一步就是跳转到中断或者异常处理程序。换句话说，处理完中断信号后，控制单元所执行的指令就是被选中处理程序的第一条指令。

中断或异常被处理完后，相应的处理程序必须产生一条iret指令，把控制权转交给被中断的进程，这将迫使控制单元：

1. 用保存在栈中的值装载cs、eip或eflags寄存器。如果一个硬件出错码曾被压入栈中，并且在eip内容的上面，那么，执行iret指令前必须先弹出这个硬件出错码。
2. 检查处理程序的CPL是否等于cs中最低两位的值（这意味着被中断的进程与处理程序运行在同一特权级）。如果是，iret终止执行；否则，转入下一步。
3. 从栈中装载ss和esp寄存器，因此，返回到与旧特权级相关的栈。
4. 检查ds、es、fs及gs段寄存器的内容，如果其中一个寄存器包含的选择符是一个段描述符，并且其DPL值小于CPL，那么，清相应的段寄存器。控制单元这么做是为了禁止用户态的程序（CPL=3）利用内核以前所用的段寄存器（DPL=0）。如果不清这些寄存器，怀有恶意的用户态程序就可能利用它们来访问内核地址空间。

中断和异常处理程序的嵌套执行

每个中断或异常都会引起一个内核控制路径，或者说代表当前进程在内核态执行单独的

指令序列。例如：当I/O设备发出一个中断时，相应的内核控制路径的第一部分指令就是那些把寄存器的内容保存在内核堆栈的指令，而最后一部分指令就是恢复寄存器内容并让CPU返回到用户态的那些指令。

内核控制路径可以任意嵌套；一个中断处理程序可以被另一个中断处理程序“中断”，因此引起内核控制路径的嵌套执行，如图4-3所示。其结果是，对中断进行处理的内核控制路径，其最后一部分指令并不总能使当前进程返回到用户态：如果嵌套深度大于1，这些指令将执行上次被打断的内核控制路径，此时的CPU依然运行在内核态。

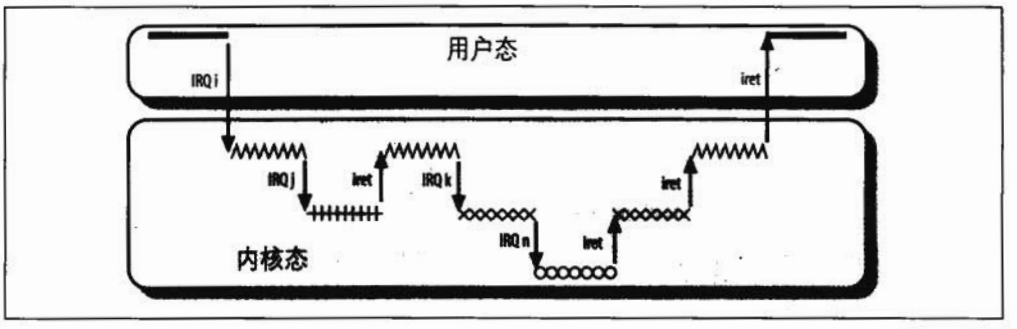


图4-3：内核控制路径嵌套执行的例子

允许内核控制路径嵌套执行必须付出代价，那就是中断处理程序必须永不阻塞，换句话说，中断处理程序运行期间不能发生进程切换。事实上，嵌套的内核控制路径恢复执行时需要的所有数据都存放在内核态堆栈中，这个栈毫无疑义的属于当前进程。

假定内核没有bug，那么大多数异常就只在CPU处于用户态时发生。事实上，异常要么是由编程错误引起，要么是由调试程序触发。然而，“Page Fault（缺页）”异常发生在内核态。这发生在当进程试图对属于其地址空间的页进行寻址，而该页现在不在RAM中时。当处理这样的一个异常时，内核可以挂起当前进程，并用另一个进程代替它，直到请求的页可以使用为止。只要被挂起的进程又获得处理器，处理缺页异常的内核控制路径就恢复执行。

因为“Page Fault”异常处理程序从不进一步引起异常，所以与异常相关的至多两个内核控制路径（第一个由系统调用引起，第二个由缺页引起）会堆叠在一起，一个在另一个之上。

与异常形成对照的是，尽管处理中断的内核控制路径代表当前进程运行，但由I/O设备产生的中断并不引用当前进程的专有数据结构。事实上，当一个给定的中断发生时，要预测哪个进程将会运行是不可能的。

一个中断处理程序既可以抢占其他的中断处理程序，也可以抢占异常处理程序。相反，异常处理程序从不抢占中断处理程序。在内核态能触发的唯一异常就是刚刚描述的缺页异常。但是，中断处理程序从不执行可以导致缺页（因此意味着进程切换）的操作。

基于以下两个主要原因，Linux 交错执行内核控制路径：

- 为了提高可编程中断控制器和设备控制器的吞吐量。假定设备控制器在一条IRQ线上产生了一个信号，PIC把这个信号转换成一个外部中断，然后PIC和设备控制器保持阻塞，一直到PIC从CPU处接收到一条应答信息。由于内核控制路径的交错执行，内核即使正在处理前一个中断，也能发送应答。
- 为了实现一种没有优先级的中断模型。因为每个中断处理程序都可以被另一个中断处理程序延缓，因此，在硬件设备之间没必要建立预定义优先级。这就简化了内核代码，提高了内核的可移植性。

在多处理器系统上，几个内核控制路径可以并发执行。此外，与异常相关的内核控制路径可以开始在一个CPU上执行，并且由于进程切换而移往另一个CPU上执行。

初始化中断描述符表

现在，我们知道了80x86微处理器在硬件级对中断和异常做了些什么，接下来，我们可以继续描述如何初始化中断描述符表。

内核启用中断以前，必须把IDT表的初始地址装到`idtr`寄存器，并初始化表中的每一项。这项工作是在初始化系统时完成的（参见附录一）。

`int`指令允许用户态进程发出一个中断信号，其值可以是0～255的任意一个向量。因此，为了防止用户通过`int`指令模拟非法的中断和异常，IDT的初始化必须非常小心。这可以通过把中断或陷阱门描述符的DPL字段设置成0来实现。如果进程试图发出其中的一个中断信号，控制单元将检查出CPL的值与DPL字段有冲突，并且产生一个“General protection”异常。

然而，在少数情况下，用户态进程必须能发出一个编程异常。为此，只要把中断或陷阱门描述符的DPL字段设置成3，即特权级尽可能一样高就足够了。

现在，让我们来看一下Linux是如何实现这种策略的。

中断门、陷阱门及系统门

与在前面“中断描述符表”中所提到的一样，Intel提供了三种类型的中断描述符：任务门、中断门及陷阱门描述符。Linux使用与Intel稍有不同的细目分类和术语，把它们如下进行分类：

中断门（interrupt gate）

用户态的进程不能访问的一个Intel中断门（门的DPL字段为0）。所有的Linux中断处理程序都通过中断门激活，并全部限制在内核态。

系统门（system gate）

用户态的进程可以访问的一个Intel陷阱门（门的DPL字段为3）。通过系统门来激活三个Linux异常处理程序，它们的向量是4，5及128，因此，在用户态下，可以发布into、bound及int $0x80三条汇编语言指令。

系统中断门（system interrupt gate）

能够被用户态进程访问的Intel中断门（门的DPL字段为3）。与向量3相关的异常处理程序是由系统中断门激活的，因此，在用户态可以使用汇编语言指令int3。

陷阱门（trap gate）

用户态的进程不能访问的一个Intel陷阱门（门的DPL字段为0）。大部分Linux异常处理程序都通过陷阱门来激活。

任务门（task gate）

不能被用户态进程访问的Intel任务门（门的DPL字段为0）。Linux对"Double fault"异常的处理程序是由任务门激活的。

下列体系结构相关的函数用来在IDT中插入门：

`set_intr_gate(n,addr)`

在IDT的第*n*个表项插入一个中断门。门中的段选择符设置成内核代码的段选择符，偏移量设置为中断处理程序的地址addr，DPL字段设置为0。

`set_system_gate(n,addr)`

在IDT的第*n*个表项插入一个陷阱门。门中的段选择符设置成内核代码的段选择符，偏移量设置为异常处理程序的地址addr，DPL字段设置为3。

`set_system_intr_gate(n,addr)`

在IDT的第*n*个表项插入一个中断门。门中的段选择符设置成内核代码的段选择符，偏移量设置为异常处理程序的地址addr，DPL字段设置为3。

`set_trap_gate(n,addr)`

与前一个函数类似，只不过DPL的字段设置成0。

set_task_gate(n,gdt)

在IDT的第*n*个表项中插入一个中断门。门中的段选择符中存放一个TSS的全局描述符表的指针，该TSS中包含要被激活的函数。偏移量设置为0，而DPL字段设置为3。

IDT的初步初始化

当计算机还运行在实模式时，IDT被初始化并由BIOS例程使用。然而，一旦Linux接管，IDT就被移到RAM的另一个区域，并进行第二次初始化，因为Linux没有利用任何BIOS例程（参见附录一）。

IDT存放在idt_table表中，有256个表项。6字节的idt_descr变量指定了IDT的大小和它的地址，只有当内核用lidt汇编指令初始化idtr寄存器时才用到这个变量（注6）。

在内核初始化过程中，setup_idt()汇编语言函数用同一个中断门(即指向ignore_int()中断处理程序)来填充所有这256个idt_table表项：

```
setup_idt:
    lea ignore_int, %edx
    movl $(_ _KERNEL_CS << 16), %eax
    movw %dx, %ax          /* selector = 0x0010 = cs */
    movw $0x8e00, %dx      /* interrupt gate, dpl=0, present */
    lea idt_table, %edi
    mov $256, %ecx
rp_sidt:
    movl %eax, (%edi)
    movl %edx, 4(%edi)
    addl $8, %edi
    dec %ecx
    jne rp_sidt
    ret
```

用汇编语言写成的ignore_int()中断处理程序，可以看作一个空的处理程序，它执行下列动作：

1. 在栈中保存一些寄存器的内容。
2. 调用printk()函数打印“Unknown interrupt”系统消息。
3. 从栈恢复寄存器的内容。
4. 执行iret指令以恢复被中断的程序。

注6： 一些旧的Pentium模式有声名狼藉的“f00f” bug，能让用户态程序冻结系统。当Linux在这样的CPU上执行时，就使用工作区，而该工作区基于用指向实际IDT的只读固定映射线性地址初始化idtr寄存器（参见第二章“固定映射的线性地址”一节）。

ignore_int()处理程序应该从不被执行，在控制台或日志文件中出现的“Unknown interrupt”消息标志着要么是出现了一个硬件的问题(一个I/O设备正在产生没有预料到的中断)，要么就是出现了一个内核的问题（一个中断或异常未被适当地处理)。

紧接着这个预初始化，内核将在IDT中进行第二遍初始化，用有意义的陷阱和中断处理程序替换这个空处理程序。一旦这个过程完成，对控制单元产生的每个不同的异常，IDT都有一个专门的陷阱或系统门，而对于可编程中断控制器确认的每一个IRQ，IDT都将包含一个专门的中断门。

在接下来的两节中，将分别针对异常和中断来详细地说明这个工作是如何完成的。

异常处理

CPU产生的大部分异常都由Linux解释为出错条件。当其中一个异常发生时，内核就向引起异常的进程发送一个信号向它通知一个反常条件。例如，如果进程执行了一个被0除的操作，CPU就产生一个“Divide error”异常，并由相应的异常处理程序向当前进程发送一个SIGFPE信号，这个进程将采取若干必要的步骤来（从出错中）恢复或者中止运行（如果没有为这个信号设置处理程序的话)。

但是，在两种情况下，Linux利用CPU异常更有效地管理硬件资源。第一种情况已经在第三章“保存和加载FPU、MMX及XMM寄存器”一节描述过，“Device not availeble”异常与cr0寄存器的TS标志一起用来把新值装入浮点寄存器。第二种情况指的是“Page Fault”异常，该异常推迟给进程分配新的页框，直到不能再推迟为止。相应的处理程序比较复杂，因为异常可能表示一个错误条件，也可能不表示一个错误条件（参见第九章“缺页异常处理程序”一节)。

异常处理程序有一个标准的结构，由以下三部分组成：

1. 在内核堆栈中保存大多数寄存器的内容（这部分用汇编语言实现)。
2. 用高级的C函数处理异常。
3. 通过ret_from_exception()函数从异常处理程序退出。

为了利用异常，必须对IDT进行适当的初始化，使得每个被确认的异常都有一个异常处理程序。trap_init()函数的工作是将一些最终值（即处理异常的函数）插入到IDT的非屏蔽中断及异常表项中。这是由函数set_trap_gate()、set_intr_gate()、set_system_gate()、set_system_intr_gate()和set_task_gate()来完成的。

```
set_trap_gate(0,&divide_error);
set_trap_gate(1,&debug);
```

```
set_intr_gate(2,&nmi);
set_system_intr_gate(3,&int3);
set_system_gate(4,&overflow);
set_system_gate(5,&bounds);
set_trap_gate(6,&invalid_op);
set_trap_gate(7,&device_not_available);
set_task_gate(8,31);
set_trap_gate(9,&coprocessor_segment_overrun);
set_trap_gate(10,&invalid_TSS);
set_trap_gate(11,&segment_not_present);
set_trap_gate(12,&stack_segment);
set_trap_gate(13,&general_protection);
set_intr_gate(14,&page_fault);
set_trap_gate(16,&coprocessor_error);
set_trap_gate(17,&alignment_check);
set_trap_gate(18,&machine_check);
set_trap_gate(19,&simd_coprocessor_error);
set_system_gate(128,&system_call);
```

由于“Double fault”异常表示内核有严重的非法操作，其处理是通过任务门而不是陷阱门或系统门来完成的，因而，试图显示寄存器值的异常处理程序并不确定esp寄存器的值是否正确。产生这种异常的时候，CPU取出存放在IDT第8项中的任务门描述符，该描述符指向存放在GDT表第32项中的TSS段描述符。然后，CPU用TSS段中的相关值装载eip和esp寄存器，结果是：处理器在自己的私有栈上执行doublefault_fn()异常处理函数。

现在我们要考察一旦一个典型的异常处理程序被调用，它会做些什么。由于篇幅所限，我们对异常处理仅做粗略的描述，尤其是我们不涉及下面的内容：

1. 由一些处理函数发送给用户态进程的信号码（见第十一章中的表11-8）。
2. 内核运行在MS-DOS虚拟模式（VM86模式）时产生的异常，它们的处理是不同的。
3. “Debug”异常。

为异常处理程序保存寄存器的值

让我们用handler_name来表示一个通用的异常处理程序的名字。(所有异常处理程序的实际名字都出现在前一部分的宏列表中。）每一个异常处理程序都以下列的汇编指令开始：

```
handler_name:
    pushl $0 /* only for some exceptions */
    pushl $do_handler_name
    jmp error_code
```

当异常发生时，如果控制单元没有自动地把一个硬件出错代码插入到栈中，相应的汇编语言片段会包含一条pushl $0指令，在栈中垫上一个空值。然后，把高级C函数的地址压进栈中，它的名字由异常处理程序名与do_前缀组成。

标号为error_code的汇编语言片段对所有的异常处理程序都是相同的，除了“Device not available”这一个异常（参见第三章的“保存和加载FPU、MMX及XMM寄存器”一节）。这段代码执行以下步骤：

1. 把高级C函数可能用到的寄存器保存在栈中。
2. 产生一条cld指令来清eflags的方向标志DF，以确保调用字符串指令（注7）时会自动增加edi和esi寄存器的值。
3. 把栈中位于esp+36处的硬件出错码拷贝到edx中，给栈中这一位置存上值-1，正如我们将在第十一章的“系统调用的重新执行”一节中所看到的那样，这个值用来把0x80异常与其他异常隔离开。
4. 把保存在栈中esp+32位置的do_handler_name()高级C函数的地址装入edi寄存器中，然后，在栈的这个位置写入es的值。
5. 把内核栈的当前栈顶拷贝到eax寄存器。这个地址表示内存单元的地址，在这个单元中存放的是第1步所保存的最后一个寄存器的值。
6. 把用户数据段的选择符拷贝到ds和es寄存器中。
7. 调用地址在edi中的高级C函数。

被调用的函数从eax和edx寄存器而不是从栈中接收参数。我们已经遇见过一个从CPU寄存器获取参数的函数_ _switch_to()，在第三章“执行进程切换”一节我们讨论过这个函数。

进入和离开异常处理程序

如前所述，执行异常处理程序的C函数名总是由do_前缀和处理程序名组成。其中的大部分函数把硬件出错码和异常向量保存在当前进程的描述符中，然后，向当前进程发送一个适当的信号。用代码描述如下：

```
current->thread.error_code = error_code;
current->thread.trap_no = vector;
force_sig(sig_number, current);
```

异常处理程序刚一终止，当前进程就关注这个信号。该信号要么在用户态由进程自己的信号处理程序（如果存在的话）来处理，要么由内核来处理。在后面这种情况下，内核一般会杀死这个进程（参见第十一章）。异常处理程序发送的信号已在表4-1中列出。

注7：一条诸如rep;movsb这样的汇编语言“字符串指令”能够作用于整个（字符串）块。

异常处理程序总是检查异常是发生在用户态还是在内核态，在后一种情况下，还要检查是否由系统调用的无效参数引起。我们将在第十章“动态地址检查：修正代码”一节描述内核如何防御自己受无效的系统调用参数攻击。出现在内核态的任何其他异常都是由于内核的bug引起的。在这种情况下，异常处理程序认为是内核行为失常了。为了避免硬盘上的数据崩溃，处理程序调用die()函数，该函数在控制台上打印出所有CPU寄存器的内容（这种转储就叫做*kernel oops*），并调用do_exit()来终止当前进程（参见第三章“进程终止”一节）。

当执行异常处理的C函数终止时，程序执行一条jmp指令以跳转到ret_from_exception()函数。这个函数将在后面的“从中断和异常返回”一节中进行描述。

中断处理

正如前面解释的那样，内核只要给引起异常的进程发送一个Unix信号就能处理大多数异常。因此，要采取的行动被延迟，直到进程接收到这个信号。所以，内核能很快地处理异常。

这种方法并不适合中断，因为经常会出现一个进程（例如，一个请求数据传输的进程）被挂起好久后中断才到达的情况，因此，一个完全无关的进程可能正在运行。所以，给当前进程发送一个Unix信号是毫无意义的。

中断处理依赖于中断类型。就我们的目的而言，我们将讨论三种主要的中断类型：

I/O中断

某些I/O设备需要关注；相应的中断处理程序必须查询设备以确定适当的操作过程。我们在后面“I/O中断处理”一节将描述这种中断。

时钟中断

某种时钟（或者是一个本地APIC时钟，或者是一个外部时钟）产生一个中断；这种中断告诉内核一个固定的时间间隔已经过去。这些中断大部分是作为I/O中断来处理的；我们将在第六章讨论时钟中断的具体特征。

处理器间中断

多处理器系统中一个CPU对另一个CPU发出一个中断。我们在后面“处理器间中断处理”一节将讨论这种中断。

I/O中断处理

一般而言，I/O中断处理程序必须足够灵活以给多个设备同时提供服务。例如在PCI总

线的体系结构中，几个设备可以共享同一个IRQ线。这就意味着仅仅中断向量不能说明所有问题。在表4-3所示的例子中，同一个向量43既分配给USB端口，也分配给声卡。然而，在老式PC体系结构（像ISA）中发现的一些硬件设备，当它们的IRQ与其他设备共享时，就不能可靠地运转。

中断处理程序的灵活性是以两种不同的方式实现的，讨论如下：

IRQ共享

中断处理程序执行多个中断服务例程（*interrupt service routine*，*ISR*）。每个ISR是一个与单独设备（共享IRQ线）相关的函数。因为不可能预先知道哪个特定的设备产生IRQ，因此，每个ISR都被执行，以验证它的设备是否需要关注；如果是，当设备产生中断时，就执行需要执行的所有操作。

IRQ动态分配

一条IRQ线在可能的最后时刻才与一个设备驱动程序相关联；例如，软盘设备的IRQ线只有在用户访问软盘设备时才被分配。这样，即使几个硬件设备并不共享IRQ线，同一个IRQ向量也可以由这几个设备在不同时刻使用（见本节最后一部分的讨论）。

当一个中断发生时，并不是所有的操作都具有相同的急迫性。事实上，把所有的操作都放进中断处理程序本身并不合适。需要时间长的、非重要的操作应该推后，因为当一个中断处理程序正在运行时，相应的IRQ线上发出的信号就被暂时忽略。更重要的是，中断处理程序是代表进程执行的，它所代表的进程必须总处于TASK_RUNNING状态，否则，就可能出现系统僵死情形。因此，中断处理程序不能执行任何阻塞过程，如磁盘I/O操作。因此，Linux把紧随中断要执行的操作分为三类：

紧急的（Critical）

这样的操作诸如：对PIC应答中断，对PIC或设备控制器重编程，或者修改由设备和处理器同时访问的数据结构。这些都能被很快地执行，而之所以说它们是紧急的是因为它们必须被尽快地执行。紧急操作要在一个中断处理程序内立即执行，而且是在禁止可屏蔽中断的情况下。

非紧急的（Noncritical）

这样的操作诸如：修改那些只有处理器才会访问的数据结构（例如，按下一个键后读扫描码）。这些操作也要很快地完成，因此，它们由中断处理程序立即执行，但必须是在开中断的情况下。

非紧急可延迟的（Noncritical deferrable）

这样的操作诸如：把缓冲区的内容拷贝到某个进程的地址空间（例如，把键盘行缓

冲区的内容发送到终端处理程序进程)。这些操作可能被延迟较长的时间间隔而不影响内核操作，有兴趣的进程将会等待数据。非紧急可延迟的操作由独立的函数来执行，我们将在“软中断及 tasklet”一节讨论。

不管引起中断的电路种类如何，所有的 I/O 中断处理程序都执行四个相同的基本操作：

1. 在内核态堆栈中保存 IRQ 的值和寄存器的内容。
2. 为正在给 IRQ 线服务的 PIC 发送一个应答，这将允许 PIC 进一步发出中断。
3. 执行共享这个 IRQ 的所有设备的中断服务例程（ISR）。
4. 跳到 `ret_from_intr()` 的地址后终止。

当中断发生时，需要用几个描述符来表示 IRQ 线的状态和需要执行的函数。图 4-4 以示意图的方式展示了处理一个中断的硬件电路和软件函数。下面几节会讨论这些函数。

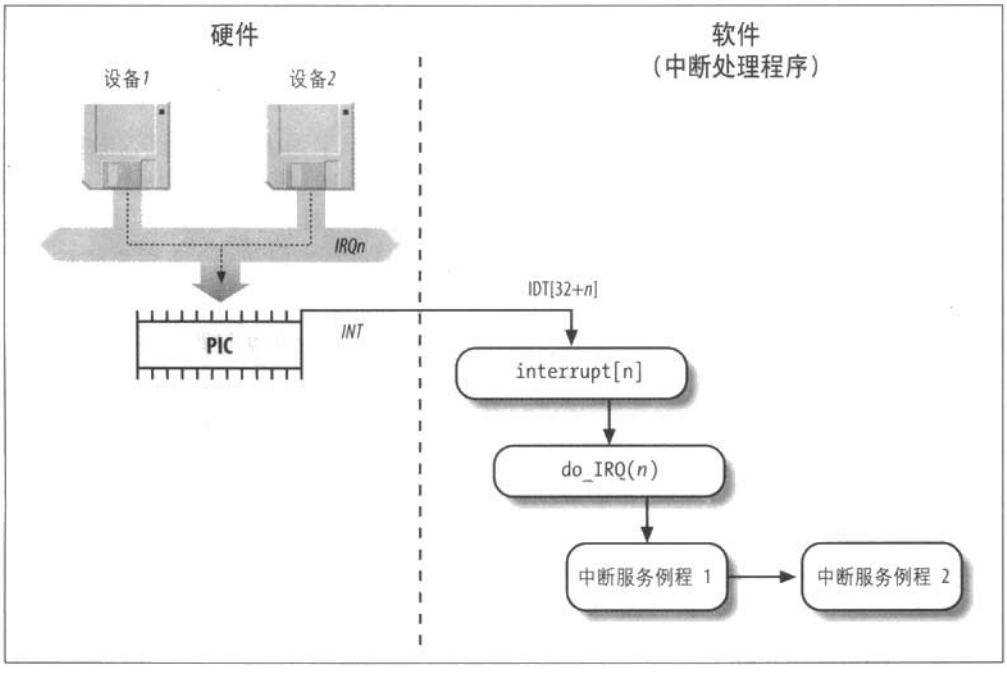


图 4-4：I/O 中断处理

中断向量

如表 4-2 所示，物理 IRQ 可以分配给 32~238 范围内的任何向量。不过，Linux 使用向量 128 实现系统调用。

IBM PC 兼容的体系结构要求，一些设备必须被静态地连接到指定的 IRQ 线。尤其是：

- 间隔定时设备必须连到 IRQ0 线（参见第六章）。
- 从 8259A PIC 必须与 IRQ2 线相连（尽管现在有了更高级的 PIC，Linux 还是支持 8259A 风格的 PIC）。
- 必须把外部数学协处理器连接到 IRQ13 线（尽管最近的 80x86 处理器不再使用这样的设备，但 Linux 仍然支持历史悠久的 80386 模型）。
- 一般而言，一个 I/O 设备可以连接到有限个 IRQ 线。（事实上，当玩一个老式 PC 时，IRQ 的共享是不可能的，由于 IRQ 与其他已经存在的硬件设备冲突，因此你不可能成功地安装一个新卡。）

表 4-2：Linux 中的中断向量

向量范围	用途
0～19 (0x0～0x13)	非屏蔽中断和异常
20～31 (0x14～0x1f)	Intel 保留
32～127 (0x20～0x7f)	外部中断（IRQ）
128 (0x80)	用于系统调用的可编程异常（参见第十章）
129～238 (0x81～0xee)	外部中断（IRQ）
239 (0xef)	本地 APIC 时钟中断（参见第六章）
240 (0xf0)	本地 APIC 高温中断（在 Pentium 4 模型中引入）
241～250 (0xf0～0xfa)	由 Linux 留作将来使用
251～253(0xfb～0xff)	处理器间中断（参见本章后面“处理器间中断处理”一节）
254 (0xfe)	本地 APIC 错误中断（当本地 APIC 检测到一个错误条件时产生）
255 (0xff)	本地 APIC 伪中断（CPU 屏蔽某个中断时产生）

为 IRQ 可配置设备选择一条线有三种方式：

- 设置一些硬件跳接器（仅适用于旧式设备卡）。
- 安装设备时执行一个实用程序。这样的程序可以让用户选择一个可用的 IRQ 号，或者探测系统自身以确定一个可用的 IRQ 号。
- 在系统启动时执行一个硬件协议。外设宣布它们准备使用哪些中断线，然后协商一个最终的值以尽可能减少冲突。该过程一旦完成，每个中断处理程序都通过访问设备某个 I/O 端口的函数，来读取所分配的 IRQ。例如，遵循外设部件互连

(Peripheral Component Interconnect，PCI) 标准的设备的驱动程序利用一组函数，如pci_read_config_byte()访问设备的配置空间。

表4-3显示了设备和IRQ之间一种相当随意的安排，你或许能在某个PC中找到同样的排列。

表4-3：把IRQ分配给I/O设备的一个例子

IRQ	INT	硬件设备
0	32	时钟
1	33	键盘
2	34	PIC级联
3	35	第二串口
4	36	第一串口
6	38	软盘
8	40	系统时钟
10	42	网络接口
11	43	USB端口、声卡
12	44	PS/2鼠标
13	45	数学协处理器
14	46	EIDE磁盘控制器的一级链接
15	47	EIDE磁盘控制器的二级链接

内核必须在启用中断前发现IRQ号与I/O设备之间的对应，否则，内核在不知道哪个向量对应哪个设备（如SCSI硬盘）的情况下，怎么能处理来自这个设备的信号呢？IRQ号与I/O设备之间的对应是在初始化每个设备驱动程序时建立的（参见第十三章）。

IRQ数据结构

当讨论到涉及状态转换的复杂操作时，首先了解关键数据存放在什么地方总是有益的。因此，本节将解释支持中断处理的数据结构以及怎样把它们放在各种描述符中。图4-5示意性地显示了几个主要描述符之间的关系，这些描述符表示IRQ线的状态（该图没有显示处理软中断及tasklet所需的数据结构，后面将对它们进行讨论。）

每个中断向量都有它自己的irq_desc_t描述符，其字段在表4-4中列出。所有的这些描述符组织在一起形成irq_desc数组。

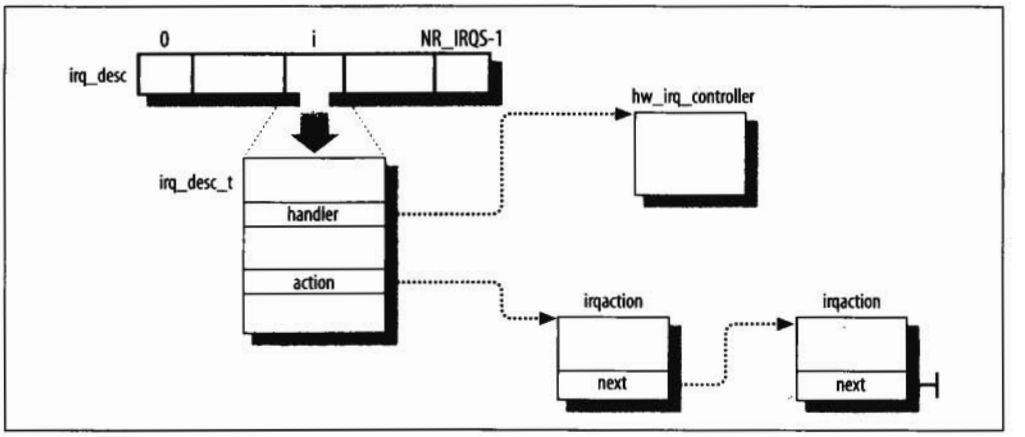


图4-5：IRQ描述符

表4-4：irq_desc_t描述符

字段	说明
handler	指向PIC对象（hw_irq_controller描述符），它服务于IRQ线
handler_data	指向PIC方法所使用的数据
action	标识当出现IRQ时要调用的中断服务例程。该字段指向IRQ的irqaction描述符链表的第一个元素。在本章后面将描述irqaction描述符。
status	描述IRQ线状态的一组标志（见表4-5）
depth	如果IRQ线被激活，则显示0；如果IRQ线被禁止了不止一次，则显示一个正数
irq_count	中断计数器，统计IRQ线上发生中断的次数（仅在诊断时使用）
irqs_unhandled	对在IRQ线上发生的无法处理的中断进行计数（仅在诊断时使用）
lock	用于串行访问IRQ描述符和PIC的自旋锁（参见第五章）

如果一个中断内核没有处理，那么这个中断就是意外中断，也就是说，与某个IRQ线相关的中断处理例程(ISR)不存在，或者与某个中断线相关的所有例程都识别不出是否是自己的硬件设备发出的中断。通常，内核检查从IRQ线接收的意外中断的数量，当这条IRQ线连接的有故障设备没完没了地发中断时，就禁用这条IRQ线。由于几个设备可能共享IRQ线，内核不会在每检测到一个意外中断时就立刻禁用IRQ线，更合适的办法是：内核把中断和意外中断的总次数分别存放在irq_desc_t描述符的irq_count和irqs_unhandled字段中，当第100 000次中断产生时，如果意外中断的次数超过99 900，内核才禁用这条IRQ线（即来自共享IRQ线的硬件设备的意外中断，比最近接收的100000次正常中断少101次。）

描述IRQ线状态的标志列在表4-5中。

表4-5：描述IRQ线状态的一组标志

标志名	描述
IRQ _INPROGRESS	IRQ的一个处理程序正在执行
IRQ _DISABLED	由一个设备驱动程序故意地禁用IRQ线
IRQ _PENDING	一个IRQ已经出现在线上，它的出现也已对PIC做出应答，但是内核还没有为它提供服务
IRQ _REPLAY	IRQ线已被禁用，但是前一个出现的IRQ还没有对PIC做出应答
IRQ _AUTODETECT	内核在执行硬件设备探测时使用IRQ线
IRQ _WAITING	内核在执行硬件设备探测时使用IRQ线；此外，相应的中断还没有产生
IRQ_LEVEL	在80x86结构上没有使用
IRQ_MASKED	未使用
IRQ_PER_CPU	在80x86结构上没有使用

irq_desc_t描述符的depth字段和IRQ_DISABLED标志表示IRQ线是否被禁用。每次调用disable_irq()或disable_irq_nosync()函数，depth字段的值增加，如果depth等于0，函数禁用IRQ线并设置它的IRQ_DISABLED标志（注8）相反，每当调用enable_irq()函数，depth字段的值减少，如果depth变为0，函数激活IRQ线并清除IRQ_DISABLED标志。

在系统初始化期间，init_IRQ()函数把每个IRQ主描述符的status字段设置成IRQ_DISABLED。此外，init_IRQ()通过替换由setup_idt()所建立的中断门(见“IDT的初步初始化”一节)来更新IDT。这是通过下列语句实现的：

```
for (i = 0; i < NR_IRQS; i++)
   if (i+32 != 128)
       set_intr_gate(i+32,interrupt[i]);
```

这段代码在interrupt数组中找到用于建立中断门的中断处理程序地址。interrupt数组中的第*n*项中存放IRQ*n*的中断处理程序的地址（见后面“为中断处理程序保存寄存器的值”一节）。注意：这里不包括与128号中断向量相关的中断门，因为它用于系统调用的编程异常。

注8：与disable_irq_nosync()相反，disable_irq(n)一直等待，直到在其他CPU上为IRQ*n*运行的所有中断处理程序都完成才返回。

Linux除了支持本章前面已提到的8259A芯片外，也支持其他的几个PIC电路，如SMP IO-APIC、Intel PIIX4的内部8259 PIC及SGI的Visual Workstation Cobalt (IO-)APIC。为了以统一的方式处理所有这样的设备，Linux用了一个“*PIC*对象”，由PIC名字和7个PIC标准方法组成。这种面向对象方法的优点是，驱动程序不必关注安装在系统中的PIC种类。每个驱动程序可见的中断源透明地连接到适当的控制器。定义PIC对象的数据结构叫做hw_interrupt_type（也叫做hw_irq_controller）。

为了简单起见，让我们假定我们的计算机是有两片8259A PIC的单处理机，它提供16个标准的IRQ。在这种情况下，有16个irq_desc_t描述符，其中每个描述符的handler字段指向描述8259A PIC的i8259A_irq _type变量。这个变量被初始化为：

```
struct hw_interrupt_type i8259A_irq_type = {
    .typename     = "XT-PIC",
    .startup      = startup_8259A_irq,
    .shutdown     = shutdown_8259A_irq,
    .enable       = enable_8259A_irq,
    .disable      = disable_8259A_irq,
    .ack          = mask_and_ack_8259A,
    .end          = end_8259A_irq,
    .set_affinity = NULL
};
```

这个结构中的第一个字段“XT-PIC”是PIC的名字。接下来就是用于对PIC编程的六个不同的函数指针。前两个函数分别启动和关闭芯片的IRQ线。但是，在使用8259A芯片的情况下，这两个函数的作用与第三、四个函数是一样的，第三、四个函数是启用和禁用IRQ线。mask_and_ack_8259A()函数通过把适当的字节发往8259A I/O端口来应答所接收的IRQ。end_8259A_irq()函数在IRQ的中断处理程序终止时被调用。最后一个set_affinity ()方法置为空：它用在多处理器系统中以声明特定IRQ所在CPU的“亲和力”——也就是说，那些CPU被启用来处理特定的IRQ。

如前所述，多个设备能共享一个单独的IRQ。因此，内核要维护多个irqaction描述符（见图4-5），其中的每个描述符涉及一个特定的硬件设备和一个特定的中断。包含在这个描述符中的字段如表4-6所示，标志如表4-7所示。

表4-6：irqaction描述符的字段

字段名	说明
handler	指向一个I/O设备的中断服务例程。这是允许多个设备共享同一IRQ的关键字段
flags	描述IRQ与I/O设备之间的关系（参见表4-7）
mask	未使用

表 4-6：irqaction 描述符的字段（续）

字段名	说明
name	I/O 设备名（通过读 */proc/interrupts* 文件，在列出所服务的 IRQ 时也显示设备名）
dev_id	I/O 设备的私有字段。典型情况下，它标识 I/O 设备本身（例如，它可能等于其主设备号和次设备号；参见第十三章中的“设备文件”一节），或者它指向设备驱动程序的数据
next	指向 irqaction 描述符链表的下一个元素。链表中的元素指向共享同一 IRQ 的硬件设备
irq	IRQ 线
dir	指向与 IRQ*n* 相关的 */proc/irq/n* 目录的描述符

表 4-7：irqaction 描述符的标志

标志名	说明
SA_INTERRUPT	处理程序必须以禁止中断执行
SA_SHIRQ	设备允许它的 IRQ 线与其他设备共享
SA_SAMPLE_RANDOM	设备可以被看作是事件随机的发生源，因此，内核可以用它做随机数产生器（用户可以从 */dev/random* 和 */dev/urandom* 设备文件中取得随机数而访问这种特征）

最后，irq_stat 数组包含 NR_CPUS 个元素，系统中的每个 CPU 对应一个元素。每个元素的类型为 irq_cpustat_t，该类型包含几个计数器和内核记录 CPU 正在做什么的标志（见表 4-8）。

表 4-8：irq_cpustat_t 结构的字段

字段名	描述
__softirq_pending	表示挂起的软中断（见本章后面“软中断”一节），为一组标志
idle_timestamp	CPU 变为空闲的时间（只是在 CPU 正空闲的时候才有意义）
__nmi_count	NMI 中断发生的次数
apic_timer_irqs	本地 APIC 时钟中断发生的次数（参见第六章）

IRQ 在多处理器系统上的分发

Linux 遵循对称多处理模型（SMP）；这就意味着，内核从本质上对任何一个 CPU 都不

应该有偏爱。因而，内核试图以轮转的方式把来自硬件设备的IRQ信号在所有CPU之间分发。因此，所有CPU服务于I/O中断的执行时间片几乎相同。

在前面“高级可编程中断控制器”一节我们已提到，多APIC系统有复杂的机制在CPU之间动态分发IRQ信号。

在系统启动的过程中，引导CPU执行setup_IO_APIC_irqs()函数来初始化I/O APIC芯片。芯片的中断重定向表的24项被填充，以便根据“最低优先级”模式把来自I/O硬件设备的所有信号都传递给系统中的每个CPU（见前面“IRQ和中断”一节）。此外，在系统启动期间，所有的CPU都执行setup_local_APIC()函数，该函数处理本地APIC的初始化。特别是，每个芯片的任务优先级寄存器（TPR）都初始化为一个固定的值，这就意味着CPU愿意处理任何类型的IRQ信号，而不管其优先级。Linux内核启动以后再也不修改这个值。

因为所有的任务优先级寄存器都包含相同的值，因此，所有CPU总是具有相同的优先级。为了突破这种约束，正如前面所解释的那样，多APIC系统使用本地APIC仲裁优先级寄存器中的值。因为这样的值在每次中断后都自动改变，因此，IRQ信号就公平地在所有CPU之间分发（注9）。

简而言之，当硬件设备产生了一个中断信号时，多APIC系统就选择其中的一个CPU，并把该信号传递给相应的本地APIC，本地APIC又依次中断它的CPU。这个事件不通报给其他所有的CPU。

所有这些都由硬件神奇地完成，因此，多APIC系统初始化后无需内核费心。遗憾的是在有些情况下，硬件不能以公平的方式在微处理器之间成功地分配中断（如，一些Pentium 4基于对称多处理的主板存在这种问题）。因此，在必要的时候，Linux2.6利用叫做*kirqd*的特殊内核线程来纠正对CPU进行的IRQ的自动分配。

内核线程为多APIC系统开发了一种优良特性，叫做CPU的IRQ亲和力：通过修改I/O APIC的中断重定向表表项，可以把中断信号发送到某个特定的CPU上。set_ioapic_affinity_irq()函数用来实现这一功能，该函数有两个参数：被重定向的IRQ向量和一个32位掩码（表示可以接收这个IRQ的CPU）。系统管理员通过向文

注9：不过，有一个例外，Linux通常以这样的方式建立本地APIC以对焦点处理器（*focus processor*）给予关注（如果它存在）。一个已经接收了某种类型IRQ信号的焦点进程，只要还没有执行完中断处理程序，它就接收所有同样类型的IRQ信号。然而，Intel在Pentium 4模型中已经取消了对焦点处理器的支持。

件 */proc/irq/n/smp_affinity*（*n* 是中断向量）中写入新的CPU位图掩码也可以改变指定中断IRQ的亲和力。

kirqd 内核线程周期性地执行 do_irq_balance () 函数，该函数跟踪在最近时间间隔内每个CPU接收的中断次数。如果该函数发现负荷最重的CPU和负荷最轻的CPU之间IRQ负载不平衡的问题太严重，它要么把IRQ从一个CPU转移到另一个CPU，要么让所有的IRQ在所有CPU之间“轮转”。

多种类型的内核栈

就像在第三章“标识一个进程”一节所提到的，每个进程的 thread_info 描述符与 thread_union结构中的内核栈紧邻，而根据内核编译时的选项不同，thread_union结构可能占一个页框或两个页框。如果 thread_union结构的大小为8KB，那么当前进程的内核栈被用于所有类型的内核控制路径：异常、中断和可延迟的函数（见后面“软中断及tasklet”一节）。相反，如果 thread_union结构的大小为4KB，内核就使用三种类型的内核栈：

- 异常栈，用于处理异常（包括系统调用）。这个栈包含在每个进程的thread_union数据结构中，因此对系统中的每个进程，内核使用不同的异常栈。
- 硬中断请求栈，用于处理中断。系统中的每个CPU都有一个硬中断请求栈，而且每个栈占用一个单独的页框。
- 软中断请求栈，用于处理可延迟的函数（软中断或tasklet；见后面“软中断及tasklet”一节）。系统中的每个CPU都有一个软中断请求栈，而且每个栈占用一个单独的页框。

所有的硬中断请求存放在hardirq_stack数组中，而所有的软中断请求存放在softirq_stack数组中，每个数组元素都是跨越一个单独页框的 irq_ctx 类型的联合体。thread_info结构存放在这个页的底部，栈使用其余的内存空间，注意每个栈向低地址方向增长。所以，硬中断请求栈和软中断请求栈都与第三章“标识一个进程”一节所描述的异常栈很相似，唯一的区别是与每个栈相连的 thread_info结构不是与进程而是与CPU相关联的。

hardirq_ctx和 softirq_ctx数组使内核能快速确定指定CPU的硬中断请求栈和软中断请求栈，它们包含的指针分别指向相应的 irq_ctx 元素。

为中断处理程序保存寄存器的值

当CPU接收一个中断时，就开始执行相应的中断处理程序代码，该代码的地址存放在IDT的相应门中（参见前面“中断和异常的硬件处理”一节）。

与其他上下文切换一样，需要保存寄存器这一点给内核开发者留下有点杂乱的编码工作，因为寄存器的保存和恢复必须用汇编语言代码，但是，在这些操作内部，又期望处理器从C函数调用和返回。在这一节，我们将描述处理寄存器的汇编语言任务，而下一节，我们将讨论在随后调用的C函数中所需的一些技巧。

保存寄存器是中断处理程序做的第一件事情。如前所述，IRQ*n*中断处理程序的地址开始存在interrupt[n]中，然后复制到IDT相应表项的中断门中。

通过文件*arch/i386/kernel/entry.S*中的几条汇编语言指令建立interrupt数组，数组包括NR_IRQS个元素，这里NR_IRQS宏产生的数为224或16，当内核支持新近的I/O APIC芯片时（注10），NR_IRQS宏产生的数为224；而当内核支持旧的8259A可编程控制器芯片时，NR_IRQS宏产生数16。数组中索引为*n*的元素中存放下面两条汇编语言指令的地址：

```
pushl $n-256
jmp common_interrupt
```

结果是把中断号减256的结果保存在栈中。内核用负数表示所有的中断，因为正数用来表示系统调用（见第十章）。当引用这个数时，可以对所有的中断处理程序都执行相同的代码。这段通用代码开始于标签common_interrupt处，包括下面的汇编语言宏和指令。

```
common_interrupt:
    SAVE_ALL
    movl %esp,%eax
    call do_IRQ
    jmp ret_from_intr
```

SAVE_ALL宏依次展开成下列片段：

```
cld
push %es
push %ds
pushl %eax
pushl %ebp
pushl %edi
pushl %esi
pushl %edx
pushl %ecx
pushl %ebx
movl $ _  _USER_DS,%edx
movl %edx,%ds
movl %edx,%es
```

注10：80x86体系结构限制了只能使用256个向量。其中32个留给CPU，因此可用向量空间有224个向量。

SAVE_ALL可以在栈中保存中断处理程序可能会使用的所有CPU寄存器，但eflags、cs、eip、ss及esp除外，因为这几个寄存器已经由控制单元自动保存了（参见前面“中断和异常的硬件处理”一节）。然后，这个宏把用户数据段的选择符装到ds和es寄存器。

保存寄存器的值以后，栈顶的地址被存放到eax寄存器中，然后中断处理程序调用do_IRQ()函数。执行do_IRQ()的ret指令时（即函数结束时），控制转到ret_from_intr()(见后面“从中断和异常返回”一节)。

do_IRQ()函数

调用do_IRQ()函数执行与一个中断相关的所有中断服务例程。该函数声明为：

```
__attribute__((regparm(3))) unsigned int do_IRQ(struct pt_regs *regs)
```

关键字regparm表示函数到eax寄存器中去找到参数regs的值。如上所见，eax指向被SAVE_ALL最后压入栈的那个寄存器在栈中的位置。

do_IRQ()函数执行下面的操作：

1. 执行irq_enter()宏，它使表示中断处理程序嵌套数量的计数器递增。计数器保存在当前进程thread_info结构的preempt_count字段中（见本章后面的表4-10）。
2. 如果thread_union结构的大小为4KB，函数切换到硬中断请求栈，并执行下面这些特殊的步骤：
   a. 执行current_thread_info()函数以获取与内核栈(地址在esp中)相连的thread_info描述符的地址（见第三章“标识一个进程”一节）。
   b. 把上一步获取的thread_info描述符的地址与存放在hardirq_ctx[smp_processor_id()]中的地址（与本地CPU相关的thread_info描述符的地址）相比较。如果两个地址相等，说明内核已经在使用硬中断请求栈，因此跳转到第3步，这种情况发生在内核处理另外一个中断时又产生了中断请求的时候。
   c. 这一步必须切换内核栈。保存当前进程描述符指针，该指针在本地CPU的irq_ctx联合体中的thread_info描述符的task字段中。完成这一步操作就能在内核使用硬中断请求栈时使当前宏按预先的期望工作（参见第三章“标识一个进程”一节）。
   d. 把esp栈指针寄存器的当前值存入本地CPU的irq_ctx联合体的thread_info描述符的previous_esp字段中（仅当为内核oop准备函数调用跟踪时使用该字段）。

e. 把本地CPU硬中断请求栈的栈顶（其值等于hardirq_ctx[smp_processor_id()]加上4096）装入esp寄存器;以前esp的值存入ebx寄存器。

3. 调用__do_IRQ()函数，把指针regs和regs->orig_eax字段中的中断号传递给该函数（见下面一节）。

4. 如果在上面的第2e步已经成功地切换到硬中断请求栈，函数把ebx寄存器中的原始栈指针拷贝到esp寄存器，从而回到以前在用的异常栈或软中断请求栈。

5. 执行宏irq_exit(),该宏递减中断计数器并检查是否有可延迟函数正等待执行(见本章稍后“软中断及tasklet”一节)。

6. 结束：控制转向ret_from_intr()函数（见后面“从中断和异常返回”一节)。

__do_IRQ()函数

__do_IRQ()函数接受IRQ号（通过eax寄存器）和指向pt_regs结构的指针（通过edx寄存器，用户态寄存器的值已经存在其中）作为它的参数。

函数相当于下面的代码段：

```
spin_lock(&(irq_desc[irq].lock));
irq_desc[irq].handler->ack(irq);
irq_desc[irq].status &= ~(IRQ_REPLAY | IRQ_WAITING);
irq_desc[irq].status |= IRQ_PENDING;
if (!(irq_desc[irq].status & (IRQ_DISABLED | IRQ_INPROGRESS))
        && irq_desc[irq].action) {
    irq_desc[irq].status |= IRQ_INPROGRESS;
    do {
        irq_desc[irq].status &= ~IRQ_PENDING;
        spin_unlock(&(irq_desc[irq].lock));
        handle_IRQ_event(irq, regs, irq_desc[irq].action);
        spin_lock(&(irq_desc[irq].lock));
    } while (irq_desc[irq].status & IRQ_PENDING);
    irq_desc[irq].status &= ~IRQ_INPROGRESS;
}
irq_desc[irq].handler->end(irq);
spin_unlock(&(irq_desc[irq].lock));
```

在访问主IRQ描述符之前，内核获得相应的自旋锁。在第五章我们会看到，自旋锁保护不同CPU的并发访问（在单处理器系统上，spin_lock()函数无所事事)。在多处理器系统上，这个锁是必要的，因为同种类型的其他中断可能产生，其他CPU可能关注新中断的出现。没有自旋锁，主IRQ描述符会被几个CPU同时访问。正如我们会看到的那样，这种情况必须绝对避免。

获得自旋锁后，函数就调用主IRQ描述符的ack方法。如果使用旧的8259A PIC，相

应的mask_and_ack_8259A()函数应答PIC上的中断，并禁用这条IRQ线。屏蔽IRQ线是为了确保在这个中断处理程序结束前，CPU不进一步接受这种中断的出现。请记住，__do_IRQ()函数是以禁止本地中断运行的；事实上，CPU控制单元自动清eflags寄存器的IF标志，因为中断处理程序是通过IDT中断门调用的。不过，我们立即会看到，内核在执行这个中断的中断服务例程之前可能会重新激活本地中断。

然而，在使用I/O高级可编程中断控制器（APIC）时，事情更为复杂。应答中断依赖于中断类型，可能是由ack方法做，也可能延迟到中断处理程序结束（也就是应答由end方法去做）。在任何一种情况下，我们都认为中断处理程序结束前，本地APIC不进一步接收这种中断，尽管这种中断的进一步出现可能被其他的CPU接受。

然后，__do_IRQ()函数初始化主IRQ描述符的几个标志。设置IRQ_PENDING标志，是因为中断已被应答（在一定程度上），但是还没有真正地处理；也清除IRQ_WAITING和IRQ_REPLAY标志（但我们现在不必关注它们）。

现在，__do_IRQ()检查是否必须真正地处理中断。在三种情况下什么也不做，这在下面给予讨论：

IRQ_DISABLED *被设置*

即使相应的IRQ线被禁止，CPU也可能执行__do_IRQ()函数；在后面“挽救丢失的中断”一节你会找到对这种非直觉情况的解释。此外，即使PIC上的IRQ线被禁用，有问题的主板也可能产生伪中断。

IRQ_INPROGRESS *被设置*

在多处理器系统中，另一个CPU可能处理同一个中断的前一次出现。为什么不把这次出现的中断推迟到那个CPU（处理前一次中断）上去处理呢？这正是Linux所做的事情。这就导致了较简单的内核结构，因为设备驱动程序的中断服务例程不必是可重入的（它们的执行是串行的）。此外，释放的CPU很快又返回到它正在做的事上而没有弄脏它的硬件高速缓存；这对系统性能是有益的。只要一个CPU用来执行中断的中断服务例程，IRQ_INPROGRESS标志就被设置；因此，__do_IRQ()函数在开始真正工作之前对这个标志进行检查。

irc_desc[irq].action *为*NULL

当中断没有相关的中断服务例程时出现这种情况。通常情况下，只有在内核正在探测一个硬件设备时这才会发生。

让我们假定三种情况没有一种成立，因此中断必须被处理。__do_IRQ()设置IRQ_INPROGRESS标志并开始一个循环。在每次循环中，函数清IRQ_PENDING标志，释放中断自旋锁，并调用handle_IRQ_event()执行中断服务例程（在后面“中断服务

例程”一节中描述）。当handle_IRQ_event()终止时，__do_IRQ()再次获得自旋锁，并检查IRQ_PENDING标志的值。如果该标志清0，那么，中断的进一步出现不传递给另一个CPU，因此，循环结束。相反，如果IRQ_PENDING被设置，当这个CPU正在执行handle_IRQ_event()时，另一个CPU已经在为这种中断执行do_IRQ()函数。因此，do_IRQ()执行循环的另一次反复，为新出现的中断提供服务（注11）。

我们的__do_IRQ()函数现在准备终止，或者是因为已经执行了中断服务例程，或者是因为无事可做。函数调用主IRQ描述符的end方法。当使用旧的8259A PIC时，相应的end_8259A_irq()函数重新激活IRQ线（除非出现伪中断）。当使用I/O APIC时，end方法应答中断（如果ack方法还没有去做）。

最后，__do_IRQ()释放自旋锁：艰难的工作已经完成！

挽救丢失的中断

__do_IRQ()函数小而简单，但在大多数情况下它都能正常工作。的确，IRQ_PENDING、IRQ_INPROGRESS和IRQ_DISABLED标志确保中断能被正确地处理，即使硬件失常也不例外。然而，在多处理器系统上事情可能不会这么顺利。

假定CPU有一条激活的IRQ线。一个硬件设备出现在这条IRQ线程上，且多APIC系统选择我们的CPU处理中断。在CPU应答中断前，这条IRQ线被另一个CPU屏蔽掉；结果，IRQ_DISABLED标志被设置。随后，我们的CPU开始处理挂起的中断；因此，do_IRQ()函数应答这个中断，然后返回，但没有执行中断服务例程，因为它发现IRQ_DISABLED标志被设置了。因此，在IRQ线禁用之前出现的中断丢失了。

为了应付这种局面，内核用来激活IRQ线的enable_irq()函数先检查是否发生了中断丢失，如果是，该函数就强迫硬件让丢失的中断再产生一次：

```
spin_lock_irqsave(&(irq_desc[irq].lock), flags);
if (--irq_desc[irq].depth == 0) {
    irq_desc[irq].status &= ~IRQ_DISABLED;
    if (irq_desc[irq].status & (IRQ_PENDING | IRQ_REPLAY))
            == IRQ_PENDING) {
        irq_desc[irq].status |= IRQ_REPLAY;
        hw_resend_irq(irq_desc[irq].handler,irq);
    }
    irq_desc[irq].handler->enable(irq);
}
spin_lock_irqrestore(&(irq_desc[irq].lock), flags);
```

注11：由于IRQ_PENDING是一个标志而不是计数器，因此只有第二次出现的中断才能被识别，而且do_IRQ()在每次循环中都只是丢弃再次出现的中断。

函数通过检查IRQ_PENDING标志的值检测到一个中断被丢失了。当离开中断处理程序时，这个标志总置为0；因此，如果IRQ线被禁止且该标志被设置，那么，中断的一个出现已经被应答但还没有处理。在这种情况下，hw_resend_irg()函数产生一个新中断。这可以通过强制本地APIC产生一个自我中断（self-interrupt）来达到（参看后面“处理器间中断处理”一节）。IRQ_REPLAY标志的作用是确保只产生一个自我中断。请记住，__do_IRQ()函数在开始处理中断时清除那个标志。

中断服务例程

如前所述，一个中断服务例程（ISR）实现一种特定设备的操作。当中断处理程序必须执行ISR时，它就调用handle_IRQ_event()函数。这个函数本质上执行如下步骤：

1. 如果SA_INTERRUPT标志清0，就用sti汇编语言指令激活本地中断。
2. 通过下列代码执行每个中断的中断服务例程：

```
retval = 0;
do {
    retval |= action->handler(irq, action->dev_id, regs);
    action = action->next;
} while (action);
```

   在循环的开始，action指向irqaction数据结构链表的开始，而irqaction表示接受中断后要采取的操作（参见本章前面的图4-5）。
3. 用cli汇编语言指令禁止本地中断。
4. 通过返回局部变量retval的值而终止，也就是说，如果没有与中断对应的中断服务例程，返回0；否则返回1（见下面）。

所有的中断服务例程都作用于相同的参数（它们分别又一次通过eax、edx和ecx寄存器来传递）：

irq
: IRQ号

dev_id
: 设备标识符

regs
: 指向内核（异常）栈的pt_regs结构的指针，栈中含有中断发生后随即保存的寄存器。pt_regs结构包括15个字段：

  - 开始的9个字段是被SAVE_ALL压入栈中的寄存器的值。

- 第10个字段为IRQ号编码，通过orig_eax字段被引用。
- 其余的字段对应由控制单元自动压入栈中的寄存器的值。

第一个参数允许一个单独的ISR处理几条IRQ线，第二个参数允许一个单独的ISR照顾几个同类型的设备，第三个参数允许ISR访问被中断的内核控制路径的执行上下文。实际上，大多数ISR不使用这些参数。

每个中断服务例程在成功处理完中断后都返回1，也就是说，当中断服务例程所处理的硬件设备（而不是共享相同IRQ的其他设备）发出信号时；否则返回0。这个返回码使内核可以更新在本章前面“IRQ数据结构”一节描述过的伪中断计数器。

当do_IRQ()函数调用一个ISR时，主IRQ描述符的SA_INTERRUPT标志决定是开中断还是关中断。通过中断调用的ISR可以由一种状态转换成相反的状态。在单处理器系统上，这是通过cli（关中断）和sti（开中断汇编语言指令实现的）。

ISR的结构依赖于所处理设备的特点。我们将在第六章和第十三章给出几个ISR的例子。

IRQ线的动态分配

在前面“中断向量”一节已经看到，有几个向量留给特定的设备，而其余的向量都被动态地处理。因此有一种方式，在该方式下同一条IRQ线可以让几个硬件设备使用，即使这些设备不允许IRQ共享。技巧就在于使这些硬件设备的活动串行化，以便一次只能有一个设备拥有这个IRQ线。

在激活一个准备利用IRQ线的设备之前，其相应的驱动程序调用request_irq()。这个函数建立一个新的irqaction描述符，并用参数值初始化它。然后调用setup _irq()函数把这个描述符插入到合适的IRQ链表。如果setup _irq()返回一个出错码，设备驱动程序中止操作，这意味着IRQ线已由另一个设备所使用，而这个设备不允许中断共享。当设备操作结束时，驱动程序调用free_irq()函数从IRQ链表中删除这个描述符，并释放相应的内存区。

让我们用一个简单的例子看一下这种方案是怎么工作的。假定一个程序想访问*/dev/fd0*设备文件对应于系统中的第一个软盘（注12）。程序要做到这点，可以通过直接访问*/dev/fd0*，也可以通过在系统上安装一个文件系统。通常将IRQ6分配给软盘控制器，给定这个号，软盘驱动程序发出下列请求：

注12：软盘是通常不允许IRQ共享的“旧设备”。

```
request_irq(6, floppy_interrupt, SA_INTERRUPT|SA_SAMPLE_RANDOM, "floppy", NULL);
```

我们可以观察到，floppy_interrupt()中断服务例程必须以关中断(设置SA_INTERRUPT)的方式来执行，并且不共享这个IRQ（清SA_SHIRQ标志)。设置SA-SAMPLE-RANDOM标志意味着对软盘的访问是内核用于产生随机数的一个较好的随机事件源。当软盘的操作被终止时(要么终止对*/dev/fd0*的I/O操作，要么卸载这个文件系统)，驱动程序就释放IRQ6：

```
free_irq(6, NULL);
```

为了把一个irqaction描述符插入到适当的链表中，内核调用setup _irq()函数，传递给这个函数的参数为irq_nr（即IRQ号）和new（即刚分配的irqaction描述符的地址)。这个函数将：

1. 检查另一个设备是否已经在用irq_nr这个IRQ，如果是，检查两个设备的irqaction描述符中的SA_SHIRQ标志是否都指定了IRQ线能被共享。如果不能使用这个IRQ线，则返回一个出错码。
2. 把*new(由new指向的新irqaction描述符)加到由irq_desc[irq_nr]->action指向的链表的末尾。
3. 如果没有其他设备共享同一个IRQ，清*new的flags字段的IRQ _DISABLED、IRQ_AUTODETECT、IRQ_WAITING和IRQ _INPROGRESS标志，并调用irq_desc[irq_nr]->handler PIC对象的startup方法以确保IRQ信号被激活。

举一个如何使用setup _irq()的例子，它是从系统初始化的代码中抽出的。内核通过执行time_init()函数中的下列指令，初始化间隔定时器设备的irq0描述符(参见第六章)。

```
struct irqaction irq0  =
    {timer_interrupt, SA_INTERRUPT, 0, "timer", NULL, NULL};
setup_irq(0, &irq0);
```

首先，类型irqaction的irq0变量被初始化：把handler字段设置成timer_interrupt()函数的地址，flags字段设置成SA_INTERRUPT，name字段设置成“*timer*”，最后一个字段设置成NULL以表示没有用dev_id值。接下来，内核调用setup _irq()把irq0插入到与IRQ0相关的irqaction描述符链表中。

处理器间中断处理

处理器间中断允许一个CPU向系统中的其他CPU发送中断信号。如本章前面“高级可编程中断控制器”一节所述，处理器间中断（IPI）不是通过IRQ线传输的，而是作为信号直接放在连接所有CPU本地APIC的总线上（在较老的主板上是一条专门的总线，而在基于Pentium 4的主板上就是系统总线)。

在多处理器系统上，Linux定义了下列三种处理器间中断（参看表4-2）：

CALL_FUNCTION_VECTOR（*向量*0xfb）

发往所有的CPU（不包括发送者），强制这些CPU运行发送者传递过来的函数。相应的中断处理程序叫做call_function_interrupt()。例如，地址存放在全局变量call_data中来传递的函数，可能强制其他所有CPU都停止，也可能强制它们设置内存类型范围寄存器（Memory Type Range Register，MTRR)(注13）的内容。通常，这种中断发往所有的CPU，但通过smp_call_function()执行调用函数的CPU除外。

RESCHEDULE_VECTOR（*向量*0xfc）

当一个CPU接收这种类型的中断时，相应的处理程序(叫做reschedule_interrupt())限定自己来应答中断。当从中断返回时，所有的重新调度都自动进行（参见本章后面“从中断和异常返回”一节)。

INVALIDATE_TLB_VECTOR（*向量*0xfd）

发往所有的CPU（不包括发送者），强制它们的转换后援缓冲器（TLB）变为无效。相应的处理程序（叫做invalidate_interrupt()）刷新处理器的某些TLB表项，正如在第二章“处理硬件高速缓存和TLB”一节所描述的那样。

处理器间中断处理程序的汇编语言代码是由BUILD_ INTERRUPT宏产生的：它保存寄存器，从栈顶压入向量号减256的值，然后调用高级C函数，其名字就是低级处理程序的名字加前缀smp_。例如，CALL_FUNCTION_VECTOR类型的处理器间中断的低级处理程序是call_function_interrupt()，它调用名为smp_call_function_interrupt()的高级处理程序。每个高级处理程序应答本地APIC上的处理器间中断，然后执行由中断触发的特定操作。

由于下列的一组函数，使得产生处理器间中断（IPI）变为一件容易的事：

send_IPI_all()

发送一个IPI到所有的CPU（包括发送者)。

send_IPI_allbutself()

发送一个IPI到所有的CPU（不包括发送者)。

注13： 从Pentium Pro模型开始，Intel微处理器包含这些附加的寄存器以易于定制高速缓存的操作。例如，Linux可以使用这些寄存器禁止硬件高速缓存对PCI/AGP图形卡的帧缓冲区进行地址映射，同时维持“操作的写组合模式：“在把写传送数据（write transfer）拷贝到帧缓冲区之前，分页单元把它们组合成较大的块。

`send_IPI_self()`

发送一个IPI到发送者的CPU。

`send_IPI_mask()`

发送一个IPI到位掩码指定的一组CPU。

软中断及tasklet

我们前面在“中断处理”一节提到，在由内核执行的几个任务之间有些不是紧急的：在必要情况下它们可以延迟一段时间。回忆一下，一个中断处理程序的几个中断服务例程之间是串行执行的，并且通常在一个中断的处理程序结束前，不应该再次出现这个中断。相反，可延迟中断可以在开中断的情况下执行。把可延迟中断从中断处理程序中抽出来有助于使内核保持较短的响应时间。这对于那些期望它们的中断能在几毫秒内得到处理的“急迫”应用来说是非常重要的。

Linux 2.6迎接这种挑战是通过两种非紧迫、可中断内核函数：所谓的可延迟函数（注14）（包括软中断与*tasklets*）和通过工作队列来执行的函数（我们将在本章后面“工作队列”一节描述它们）。

软中断和tasklet有密切的关系，tasklet是在软中断之上实现。事实上，出现在内核代码中的术语“软中断（softirq）”常常表示可延迟函数的所有种类。另外一种被广泛使用的术语是“中断上下文”：表示内核当前正在执行一个中断处理程序或一个可延迟的函数。

软中断的分配是静态的（即在编译时定义），而tasklet的分配和初始化可以在运行时进行（例如：安装一个内核模块时）。软中断（即便是同一种类型的软中断）可以并发地运行在多个CPU上。因此，软中断是可重入函数而且必须明确地使用自旋锁保护其数据结构。tasklet不必担心这些问题，因为内核对tasklet的执行进行了更加严格的控制。相同类型的tasklet总是被串行地执行，换句话说就是：不能在两个CPU上同时运行相同类型的tasklet。但是，类型不同的tasklet可以在几个CPU上并发执行。tasklet的串行化使tasklet函数不必是可重入的，因此简化了设备驱动程序开发者的工作。

一般而言，在可延迟函数上可以执行四种操作：

初始化（initialization）

定义一个新的可延迟函数；这个操作通常在内核自身初始化或加载模块时进行。

注14： 它们也称软中断（software interrupt），我们称它们为“可延迟函数“是为了避免与编程异常相混淆，在Intel手册中，编程异常被称为软中断。

激活（activation）

标记一个可延迟函数为“挂起”（在可延迟函数的下一轮调度中执行）。激活可以在任何时候进行（即使正在处理中断）。

屏蔽（masking）

有选择地屏蔽一个可延迟函数，这样，即使它被激活，内核也不执行它。我们会在第五章“禁止和激活可延迟函数”一节看到，禁止可延迟函数有时是必要的。

执行（execution）

执行一个挂起的可延迟函数和同类型的其他所有挂起的可延迟函数；执行是在特定的时间进行的，这将在后面“软中断”一节解释。

激活和执行不知何故总是捆绑在一起：由给定CPU激活的一个可延迟函数必须在同一个CPU上执行。没有什么明显的理由说明这条规则对系统性能是有益的。把可延迟函数绑定在激活CPU上从理论上说可以更好地利用CPU的硬件高速缓存。毕竟，可以想象，激活的内核线程访问的一些数据结构，可延迟函数也可能会使用。然而，当可延迟函数运行时，因为它的执行可以延迟一段时间，因此相关高速缓存行很可能就不再在高速缓存中了。此外，把一个函数绑定在一个CPU上总是一种有潜在“危险的”操作，因为一个CPU可能忙死而其他CPU又无所事事。

软中断

Linux 2.6使用有限个软中断。在很多场合，tasklet是足够用的，且更容易编写，因为tasklet不必是可重入的。

事实上，如表4-9所示，目前只定义了六种软中断。

表4-9：Linux 2.6中使用的软中断

软中断	下标（优先级）	说明
HI_SOFTIRQ	0	处理高优先级的tasklet
TIMER_SOFTIRQ	1	和时钟中断相关的tasklet
NET_TX_SOFTIRQ	2	把数据包传送到网卡
NET_RX_SOFTIRQ	3	从网卡接收数据包
SCSI_SOFTIRQ	4	SCSI命令的后台中断处理
TASKLET_SOFTIRQ	5	处理常规tasklet

一个软中断的下标决定了它的优先级：低下标意味着高优先级，因为软中断函数将从下标0开始执行。

软中断所使用的数据结构

表示软中断的主要数据结构是softirq_vec数组，该数组包含类型为softirq_action的32个元素。一个软中断的优先级是相应的softirq_action元素在数组内的下标。如表4-9所示，只有数组的前六个元素被有效地使用。softirq_action数据结构包括两个字段：指向软中断函数的一个action指针和指向软中断函数需要的通用数据结构的data指针。

另外一个关键的字段是32位的preempt_count字段，用它来跟踪内核抢占和内核控制路径的嵌套，该字段存放在每个进程描述符的thread_info字段中（见第三章“标识一个进程”一节）。如表4-10所示，preempt_count字段的编码表示三个不同的计数器和一个标志。

表4-10：preempt_count的字段

位	描述
0～7	抢占计数器(max value = 255)
8～15	软中断计数器 (max value = 255)
16～27	硬中断计数器 (max value = 4096)
28	PREEMPT_ACTIVE 标志

第一个计数器记录显式禁用本地CPU内核抢占的次数，值等于0表示允许内核抢占。第二个计数器表示可延迟函数被禁用的程度(值为0表示可延迟函数处于激活状态)。第三个计数器表示在本地CPU上中断处理程序的嵌套数（irq_enter()宏递增它的值，irq_exit()宏递减它的值；见本章前面“I/O中断处理”一节）。

给preempt_count字段起这个名字的理由是很充分的：当内核代码明确不允许发生抢占(抢占计数器不等于0）或当内核正在中断上下文中运行时，必须禁用内核的抢占功能。因此，为了确定是否能够抢占当前进程，内核快速检查preempt_count字段中的相应值是否等于0。在第五章“内核抢占”一节将深入讨论内核抢占。

宏in_interrupt()检查current_thread_info()->preempt_count字段的硬中断计数器和软中断计数器，只要这两个计数器中的一个值为正数，该宏就产生一个非零值，否则产生一个零值。如果内核不使用多内核栈，则该宏只检查当前进程的thread_info描述符的preempt_count字段。但是，如果内核使用多内核栈，则该宏可能还要检查本地CPU的irq_ctx联合体中thread_info描述符的preempt_count字段。在这种情况下，由于该字段总是正数值，所以宏返回非零值。

实现软中断的最后一个关键的数据结构是每个CPU都有的32位掩码（描述挂起的软中断），它存放在irq_cpustat_t数据结构（回忆一下，在系统中每个CPU有一个这样的数据结构，见表4-8）的_ _softirq_pending字段中。为了获取或设置位掩码的值，内核使用宏local_softirq_pending()，它选择本地CPU的软中断位掩码。

处理软中断

open_softirq()函数处理软中断的初始化。它使用三个参数：软中断下标、指向要执行的软中断函数的指针及指向可能由软中断函数使用的数据结构的指针。open_softirq()限制自己初始化softirq_vec数组中适当的元素。

raise_softirq()函数用来激活软中断，它接受软中断下标nr作为参数，执行下面的操作：

1. 执行local_irq_save宏以保存eflags寄存器IF标志的状态值并禁用本地CPU上的中断。
2. 把软中断标记为挂起状态，这是通过设置本地CPU的软中断掩码中与下标nr相关的位来实现的。
3. 如果in_interrupt()产生为1的值，则跳转到第5步。这种情况说明：要么已经在中断上下文中调用了raise_softirq()，要么当前禁用了软中断。
4. 否则，就在需要的时候去调用wakeup_softirqd()以唤醒本地CPU的*ksoftirqd*内核线程（见后面）。
5. 执行local_irq_restore宏，恢复在第1步保存的IF标志的状态值。

应该周期性地（但又不能太频繁地）检查活动（挂起）的软中断，检查是在内核代码的几个点上进行的。这在下列几种情况下进行（注意，检查点的个数和位置随内核版本和所支持的硬件结构而变化）：

- 当内核调用local_bh_enable（）函数（注15）激活本地CPU的软中断时。
- 当do_IRQ()完成了I/O中断的处理时或调用irq_exit()宏时。
- 如果系统使用I/O APIC，则当smp_apic_timer_interrupt()函数处理完本地定时器中断时（见第六章“多处理器系统上的计时体系结构”一节）。
- 在多处理器系统中，当CPU处理完被CALL_FUNCTION_VECTOR处理器间中断所触发的函数时。

注15： local_bh_enable()这个名称表示叫做“后半部分”的特殊类型的可延迟函数，在Linux 2.6内核中已经不存在“后半部分”类型的可延迟函数。

- 当一个特殊的 *ksoftirqd/n* 内核线程被唤醒时（见后面）。

do_softirq()函数

如果在这样的一个检查点（local_softirq_pending()不为0）检测到挂起的软中断，内核就调用 do_softirq()来处理它们。这个函数执行下面的操作：

1. 如果 in_interrupt()产生值1，则函数返回。这种情况说明要么在中断上下文中调用了 do_softirq()函数，要么当前禁用软中断。
2. 执行 local_irq_save以保存 IF 标志的状态值，并禁用本地 CPU 上的中断。
3. 如果 thread_union的结构大小为4KB，那么在需要的情况下，它切换到软中断请求栈。这一步与前面“I/O 中断处理”一节 do_IRQ()的第2步很相似，当然这里使用的是数组 softirq_ctx而不是 hardirq_ctx。
4. 调用 __do_softirq()函数（参见下面一节）。
5. 如果在上面第3步成功切换到软中断请求栈，则把最初的栈指针恢复到esp寄存器中，这样就切换回到以前使用的异常栈。
6. 执行 local_irq_restore以恢复在第2步保存的 IF 标志（表示本地是关中断还是开中断）的状态值并返回。

__do_softirq()函数

__do_softirq()函数读取本地CPU的软中断掩码并执行与每个设置位相关的可延迟函数。由于正在执行一个软中断函数时可能出现新挂起的软中断，所以为了保证可延迟函数的低延迟性，__do_softirq()一直运行到执行完所有挂起的软中断。但是，这种机制可能迫使__do_softirq()运行很长一段时间，因而大大延迟用户态进程的执行。因此，__do_softirq()只做固定次数的循环，然后就返回。如果还有其余挂起的软中断，那么下一节要描述的内核线程 *ksoftirqd* 将会在预期的时间内处理它们。下面简单描述 __do_softirq()函数执行的操作：

1. 把循环计数器的值初始化为10。
2. 把本地CPU（被 local_softirq_pending()选中的）软中断的位掩码复制到局部变量 pending 中。
3. 调用 local_bh_disable()增加软中断计数器的值。在可延迟函数开始执行之前应该禁用它们，这似乎有点违反直觉，但确实极有意义。因为在绝大多数情况下可延迟函数是在开中断的状态下运行的，所以在执行__do_softirq()的过程中可能会产生新的中断。当 do_IRQ()执行 irq_exit()宏时，可能有另外一个

__do_softirq()函数的实例开始执行。这种情况是应该避免的，因为可延迟函数必须以串行的方式在CPU上运行。因此，__do_softirq()函数的第一个实例禁用可延迟函数，以使每个新的函数实例将会在do_softirq()函数的第1步就退出。

4. 清除本地CPU的软中断位图，以便可以激活新的软中断（在第2步，已经把位图保存在pending局部变量中）。
5. 执行local_irq_enable()来激活本地中断。
6. 根据局部变量pending每一位的设置，执行对应的软中断处理函数。回忆一下，下标为n的软中断函数的地址存放在softirq_vec[n]->action变量中。
7. 执行local_irq_disable()以禁用本地中断。
8. 把本地CPU的软中断位掩码复制到局部变量pending中，并且再次递减循环计数器。
9. 如果pending 不为0，那么从最后一次循环开始，至少有一个软中断被激活，而且循环计数器仍然是正数，跳转回到第4步。
10. 如果还有更多的挂起软中断，则调用wakeup_softirqd()唤醒内核线程来处理本地CPU的软中断（见下一节）。
11. 软中断计数器减1，因而重新激活可延迟函数。

ksoftirqd内核线程

在最近的内核版本中，每个CPU都有自己的*ksoftirqd/n*内核线程（这里，*n*为CPU的逻辑号）。每个*ksoftirqd/n*内核线程都运行ksoftirqd()函数，该函数实际上执行下列的循环：

```
for(;;) {
    set_current_state(TASK_INTERRUPTIBLE );
    schedule();
    /* now in TASK_RUNNING state */
    while (local_softirq_pending()) {
        preempt_disable();
        do_softirq();
        preempt_enable();
        cond_resched();
    }
```

当内核线程被唤醒时，就检查local_softirq_pending()中的软中断位掩码并在必要时调用do_softirq()。如果没有挂起的软中断，函数把当前进程状态置为TASK_INTERRUPTIBLE，随后，如果当前进程需要（当前thread_info的TIF_NEED_RESCHED标志被设置）就调用cond_resched()函数来实现进程切换。

ksoftirqd/n 内核线程为重要而难以平衡的问题提供了解决方案。

软中断函数可以重新激活自己；实际上，网络软中断和 tasklet 软中断都可以这么做。此外，像网卡上数据包泛滥这样的外部事件可能以高频率激活软中断。

软中断的连续高流量可能会产生问题，该问题就是由引入的内核线程来解决的。没有内核线程，开发者实际上就面临两种选择策略。

第一种策略就是忽略do_softirq()运行时新出现的软中断。换句话说，do_softirq()函数开始执行时，确定哪些软中断是挂起的，然后执行这些软中断的函数。接下来，do_softirq()不再重新检查挂起的软中断就终止。这种解决方法不是很好。假设一个软中断函数在 do_softirq()执行期间被重新激活。在最坏的情况下，即使机器空闲，也只有在下一次时钟中断到来时，该软中断才被再执行。结果，对网络开发者来说，软中断的等待时间是不可接受的。

第二种策略在于不断地重新检查挂起的软中断。do_softirq()函数一直检查挂起的软中断，只有在没有挂起的软中断时才终止。尽管这种解决方法可能满足了网络开发者的愿望，但是，它肯定会使系统中的普通用户感到恼怒：如果网卡接收高频率的数据包流，或者如果一个软中断函数总是激活自己，那么，do_softirq()函数就会永不返回，用户态程序实际上就停止执行。

*ksoftirqd/n*内核线程试图解决这种很难平衡的问题。do_softirq()函数确定哪些软中断是挂起的，并执行它们的函数。如果已经执行的软中断又被激活，do_softirq()则唤醒内核线程并终止（_ _do_softirq()的第 10 步）。内核线程有较低的优先级，因此用户程序就有机会运行；但是，如果机器空闲，挂起的软中断就很快被执行。

tasklet

tasklet 是 I/O 驱动程序中实现可延迟函数的首选方法。如前所述，tasklet 建立在两个叫做HI_SOFTIRQ和TASKLET_SOFTIRQ的软中断之上。几个tasklet可以与同一个软中断相关联，每个 tasklet 执行自己的函数。两个软中断之间没有真正的区别，只不过do_softirq()先执行HI_SOFTIRQ的 tasklet，后执行TASKLET_SOFTIRQ的tasklet。

tasklet 和高优先级的 tasklet 分别存放在 tasklet_vec 和 tasklet_hi_vec 数组中。二者都包含类型为tasklet_head 的NR_CPUS个元素，每个元素都由一个指向 tasklet 描述符链表的指针组成。tasklet 描述符是一个 tasklet_struct 类型的数据结构，其字段如表 4-11 所示。

表4-11：tasklet描述符的字段

字段名	描述
next	指向链表中下一个描述符的指针
state	tasklet的状态
count	锁计数器
func	指向tasklet函数的指针
data	一个无符号长整数，可以由tasklet函数来使用

tasklet描述符的state字段含有两个标志：

TASKLET_STATE_SCHED

该标志被设置时，表示tasklet是挂起的（曾被调度执行），也意味着tasklet描述符被插入到tasklet_vec和 tasklet_hi_vec数组的其中一个链表中。

TASKLET_STATE_RUN

该标志被设置时，表示tasklet正在被执行；在单处理器系统上不使用这个标志，因为没有必要检查特定的tasklet是否在运行。

让我们假定，你正在写一个设备驱动程序，且想使用tasklet，应该做些什么呢？首先，你应该分配一个新的tasklet_struct数据结构，并调用tasklet_init()初始化它；该函数接收的参数为tasklet描述符的地址、tasklet函数的地址和它的可选整型参数。

调用tasklet_disable_nosync()或tasklet_disable()可以选择性地禁止tasklet。这两个函数都增加tasklet描述符的count字段，但是后一个函数只有在tasklet函数已经运行的实例结束后才返回。为了重新激活你的tasklet，调用tasklet_enable()。

为了激活tasklet，你应该根据自己tasklet需要的优先级，调用tasklet_schedule()函数或tasklet_hi_schedule()函数。这两个函数非常类似，其中每个都执行下列操作：

1. 检查TASKLET_STATE_SCHED标志；如果设置则返回（tasklet已经被调度）。
2. 调用local_irq_save保存IF标志的状态并禁用本地中断。
3. 在tasklet_vec[n]或 tasklet_hi_vec[n]指向的链表的起始处增加tasklet描述符（n表示本地CPU的逻辑号）。
4. 调用raise_softirq_irqoff()激活TASKLET_SOFTIRQ或HI_SOFTIRQ类型的软中断。（这个函数与raise_softirq()函数类似，只是raise_softirq_irqoff()函数假设已经禁用了本地中断。）
5. 调用local_irq_restore恢复IF标志的状态。

最后，让我们看一下tasklet如何被执行。我们从前一节知道，软中断函数一旦被激活，就由do_softirq()函数执行。与HI_SOFTIRQ软中断相关的软中断函数叫做tasklet_hi_action()，而与TASKLET_SOFTIRQ相关的函数叫做tasklet_action()。这两个函数非常相似，它们都执行下列操作：

1. 禁用本地中断。
2. 获得本地CPU的逻辑号n。
3. 把tasklet_vec[n]或tasklet_hi_vec[n]指向的链表的地址存入局部变量list。
4. 把tasklet_vec[n]或tasklet_hi_vec[n]的值赋为NULL，因此，已调度的tasklet描述符的链表被清空。
5. 打开本地中断。
6. 对于list指向的链表中的每个tasklet描述符：
   a. 在多处理器系统上，检查tasklet的TASKLET_STATE_RUN标志。
      - 如果该标志被设置，说明同类型的一个tasklet正在另一个CPU上运行，因此，就把任务描述符重新插入到由tasklet_vec[n]或tasklet_hi_vec[n]指向的链表中，并再次激活TASKLET_SOFTIRQ或HI_SOFTIRQ软中断。这样，当同类型的其他tasklet在其他CPU上运行时，这个tasklet就被延迟。
      - 如果TASKLET_STATE_RUN标志未被设置，tasklet就没有在其他CPU上运行，就需要设置这个标志，以便tasklet函数不能在其他CPU上执行。
   b. 通过查看tasklet描述符的count字段，检查tasklet是否被禁止。如果是，就清TASKLET-STATE-RUN标志，并把任务描述符重新插入到由tasklet_vec[n]或tasklet_hi_vec[n]指向的链表中，然后函数再次激活TASKLET_SOFTIRQ或HI_SOFTIRQ软中断。
   c. 如果tasklet被激活，清TASKLET_STATE_SCHED标志，并执行tasklet函数。

注意，除非tasklet函数重新激活自己，否则，tasklet的每次激活至多触发tasklet函数的一次执行。

工作队列

在Linux 2.6中引入了工作队列，它与Linux 2.4中的任务队列具有相似的构造，用来代替任务队列。它们允许内核函数（非常像可延迟函数）被激活，而且稍后由一种叫做工作者线程（*worker thread*）的特殊内核线程来执行。

尽管可延迟函数和工作队列非常相似，但是它们的区别还是很大的。主要区别在于：可延迟函数运行在中断上下文中，而工作队列中的函数运行在进程上下文中。执行可阻塞函数（例如：需要访问磁盘数据块的函数）的唯一方式是在进程上下文中运行。因为，正如本章前面“中断和异常处理程序的嵌套执行”一节所见，在中断上下文中不可能发生进程切换。可延迟函数和工作队列中的函数都不能访问进程的用户态地址空间。事实上，可延迟函数被执行时不可能有任何正在运行的进程。另一方面，工作队列中的函数是由内核线程来执行的，因此，根本不存在它要访问的用户态地址空间。

工作队列的数据结构

与工作队列相关的主要数据结构是名为workqueue_struct的描述符，它包括一个有NR_CPUS个元素的数组，NR_CPUS是系统中CPU的最大数量（注16）。每个元素都是cpu_workqueue_struct类型的描述符，有关数据结构的字段如表4-12所示。

表4-12：cpu_workqueue_struct结构的字段

字段名	描述
lock	保护该数据结构的自旋锁
remove_sequence	flush_workqueue()使用的序列号
insert_sequence	flush_workqueue()使用的序列号
worklist	挂起链表的头结点
more_work	等待队列，其中的工作者线程因等待更多的工作而处于睡眠状态
work_done	等待队列，其中的进程由于等待工作队列被刷新而处于睡眠状态
wq	指向workqueue_struct结构的指针，其中包含该描述符
thread	指向结构中工作者线程的进程描述符指针
run_depth	run_workqueue()当前的执行深度（当工作队列链表中的函数阻塞时，这个字段的值会变得比1大）

cpu_workqueue_struct结构的worklist字段是双向链表的头，链表集中了工作队列中的所有挂起函数。work_struct数据结构用来表示每一个挂起函数，它的字段如表4-13所示。

注16：在多处理器系统中复制工作队列数据结构的原因是每CPU本地数据结构产生更有效的代码（参见第五章“每CPU变量”一节）。

表 4-13：work_struct 结构的字段

字段名	描述
pending	如果函数已经在工作队列链表中，该字段值设为 1，否则设为 0
entry	指向挂起函数链表前一个或后一个元素的指针
func	挂起函数的地址
data	传递给挂起函数的参数，是一个指针
wq_data	通常是指向 cpu_workqueue_struct 描述符的父结点的指针
timer	用于延迟挂起函数执行的软定时器

工作队列函数

create_workqueue("foo")函数接收一个字符串作为参数，返回新创建工作队列的workqueue_struct描述符的地址。该函数还创建*n*个工作者线程（*n*是当前系统中有效运行的CPU的个数），并根据传递给函数的字符串为工作者线程命名，如：*foo/0*，*foo/1*等等。create_singlethread_workqueue()函数与之相似，但不管系统中有多少个CPU，create_singlethread_workqueue()函数都只创建一个工作者线程。内核调用destroy_workqueue()函数撤消工作队列，它接收指向workqueue_struct数组的指针作为参数。

queue_work()（封装在work_struct描述符中）把函数插入工作队列，它接收wq和work两个指针。wq指向workqueue_struct描述符，work 指向work_struct描述符。queue_work()主要执行下面的步骤：

1. 检查要插入的函数是否已经在工作队列中（work->pending字段等于1），如果是就结束。
2. 把work_struct描述符加到工作队列链表中，然后把work->pending置为1。
3. 如果工作者线程在本地CPU的cpu_workqueue_struct描述符的more_work等待队列上睡眠，该函数唤醒这个线程。

queue_delayed_work()函数和queue_work()几乎是相同的，只是queue_delayed_work()函数多接收一个以系统滴答数（参见第六章）来表示时间延迟的参数，它用于确保挂起函数在执行前的等待时间尽可能短。事实上，queue_delayed_work()依靠软定时器（work_struct描述符的timer字段）把work_struct描述符插入工作队列链表的实际操作向后推迟了。如果相应的work_struct描述符还没有插入工作队列链表，cancel_delayed_work()就删除曾被调度过的工作队列函数。

每个工作者线程在worker_thread()函数内部不断地执行循环操作，因而，线程在绝大多数时间里处于睡眠状态并等待某些工作被插入队列。工作线程一旦被唤醒就调用run_workqueue()函数，该函数从工作者线程的工作队列链表中删除所有work_struct描述符并执行相应的挂起函数。由于工作队列函数可以阻塞，因此，可以让工作者线程睡眠，甚至可以让它迁移到另一个CPU上恢复执行（注17）。

有些时候，内核必须等待工作队列中的所有挂起函数执行完毕。flush_workqueue()函数接收workqueue_struct描述符的地址，并且在工作队列中的所有挂起函数结束之前使调用进程一直处于阻塞状态。但是该函数不会等待在调用flush_workqueue()之后新加入工作队列的挂起函数，每个cpu_workqueue_struct描述符的remove_sequence字段和insert_sequence字段用于识别新增加的挂起函数。

预定义工作队列

在绝大多数情况下，为了运行一个函数而创建整个工作者线程开销太大了。因此，内核引入叫做*events*的预定义工作队列，所有的内核开发者都可以随意使用它。预定义工作队列只是一个包括不同内核层函数和I/O驱动程序的标准工作队列，它的workqueue_struct描述符存放在keventd_wq数组中。为了使用预定义工作队列，内核提供表4-14中列出的函数。

表4-14：预定义工作队列的支持函数

预定义工作队列函数	等价的标准工作队列函数
schedule_work(w)	queue_work(keventd_wq,w)
schedule_delayed_work(w,d)	queue_delayed_work(keventd_wq,w,d)（在任何CPU上）
schedule_delayed_work_on(cpu,w,d)	queue_delayed_work(keventd_wq,w,d)（在某个指定的CPU上）
flush_scheduled_work()	flush_workqueue(keventd_wq)

当函数很少被调用时，预定义工作队列节省了重要的系统资源。另一方面，不应该使在预定义工作队列中执行的函数长时间处于阻塞状态。因为工作队列链表中的挂起函数是在每个CPU上以串行的方式执行的，而太长的延迟对预定义工作队列的其他用户会产生不良影响。

注17：非常奇怪，一个工作者线程不仅仅被与cpu_workqueue_struct描述符（工作者线程属于该描述符）相关的CPU执行，而且能被所有CPU执行。因此，虽然queue_work()把函数插入本地CPU队列，但系统中的所有CPU都可以执行这个函数。

除了一般的*events*队列，在Linux2.6中你还会发现一些专用的工作队列。其中最重要的是块设备层使用的*kblockd*工作队列(参见第十四章)。

从中断和异常返回

我们通过考察中断和异常处理程序的终止阶段来结束本章（从系统调用返回是个特例，我们将在第十章给予描述)。尽管终止阶段的主要目的很清楚，即恢复某个程序的执行，但是，在这样做之前，还需要考虑几个问题：

内核控制路径并发执行的数量

如果仅仅只有一个，那么CPU必须切换到用户态。

挂起进程的切换请求

如果有任何请求，内核就必须执行进程调度；否则，把控制权还给当前进程。

挂起的信号

如果一个信号发送到当前进程，就必须处理它。

单步执行模式

如果调试程序正在跟踪当前进程的执行，就必须在进程切换回到用户态之前恢复单步执行。

Virtual-8086模式

如果CPU处于virtual-8086模式，当前进程正在执行原来的实模式程序，因而必须以特殊的方式处理这种情况。

需要使用一些标志来记录挂起进程切换的请求、挂起信号和单步执行，这些标志被存放在thread_info描述符的flags字段中，这个字段也存放其他与从中断和异常返回无关的标志。表4-15完整地列出了与中断和异常返回相关的标志。

表4-15：thread_info描述符的flags字段

标志名	描述
TIF_SYSCALL_TRACE	正在跟踪系统调用
TIF_NOTIFY_RESUME	在80x86平台上不使用
TIF_SIGPENDING	进程有挂起信号
TIF_NEED_RESCHED	必须执行调度程序
TIF_SINGLESTEP	临返回用户态前恢复单步执行
TIF_IRET	通过iret而不是sysexit从系统调用强行返回
TIF_SYSCALL_AUDIT	系统调用正在被审计

表 4-15：thread_info描述符的flags字段（续）

标志名	描述
TIF_POLLING_NRFLAG	空闲进程正在轮询TIF_NEED_RESCHED标志
TIF_MEMDIE	正在撤消进程以回收内存（参见第十七章“内存不足删除程序”一节）

从技术上说，完成所有这些事情的内核汇编语言代码并不是一个函数，因为控制权从不返回到调用它的函数。它只是一个代码片段，有两个不同的入口点，分别叫做ret_from_intr()和ret_from_exception()。正如其名所暗示的，中断处理程序结束时，内核进入ret_from_intr(),而当异常处理程序结束时，它进入ret_from_exception()。为了描述起来更容易一些，我们将把这两个入口点当作函数来讨论。

图4-6是关于两个入口点的完整的流程图。灰色的框图涉及实现内核抢占（参见第五章）的汇编语言指令，如果你只想了解不支持抢占的内核都做了些什么，就可以忽略这些灰色的框图。在流程图中，入口点ret_from_exception()和ret_from_intr()看起来非常相似，它们的唯一区别是：如果内核在编译时选择了支持内核抢占，那么从异常返回时要立刻禁用本地中断。

流程图大致给出了恢复执行被中断的程序所必需的步骤。现在，我们要通过讨论汇编语言代码来详细描述这个过程。

入口点

ret_from_intr()和ret_from_exception()入口点本质上相当于下面这段汇编语言代码：

```
ret_from_exception:
    cli ; missing if kernel preemption is not supported
ret_from_intr:
    movl $-8192, %ebp ; -4096 if multiple Kernel Mode stacks are used
    andl %esp, %ebp
    movl 0x30(%esp), %eax
    movb 0x2c(%esp), %al
    testl $0x00020003, %eax
    jnz resume_userspace
    jpm resume_kernel
```

回忆前面对handle_IRQ_event()描述的第3步，在中断返回时，本地中断是被禁用的。因此，只有在从异常返回时才使用cli这条汇编语言指令。

内核把当前thread_info描述符的地址装载到ebp寄存器（参见第三章“标识一个进程”一节）。

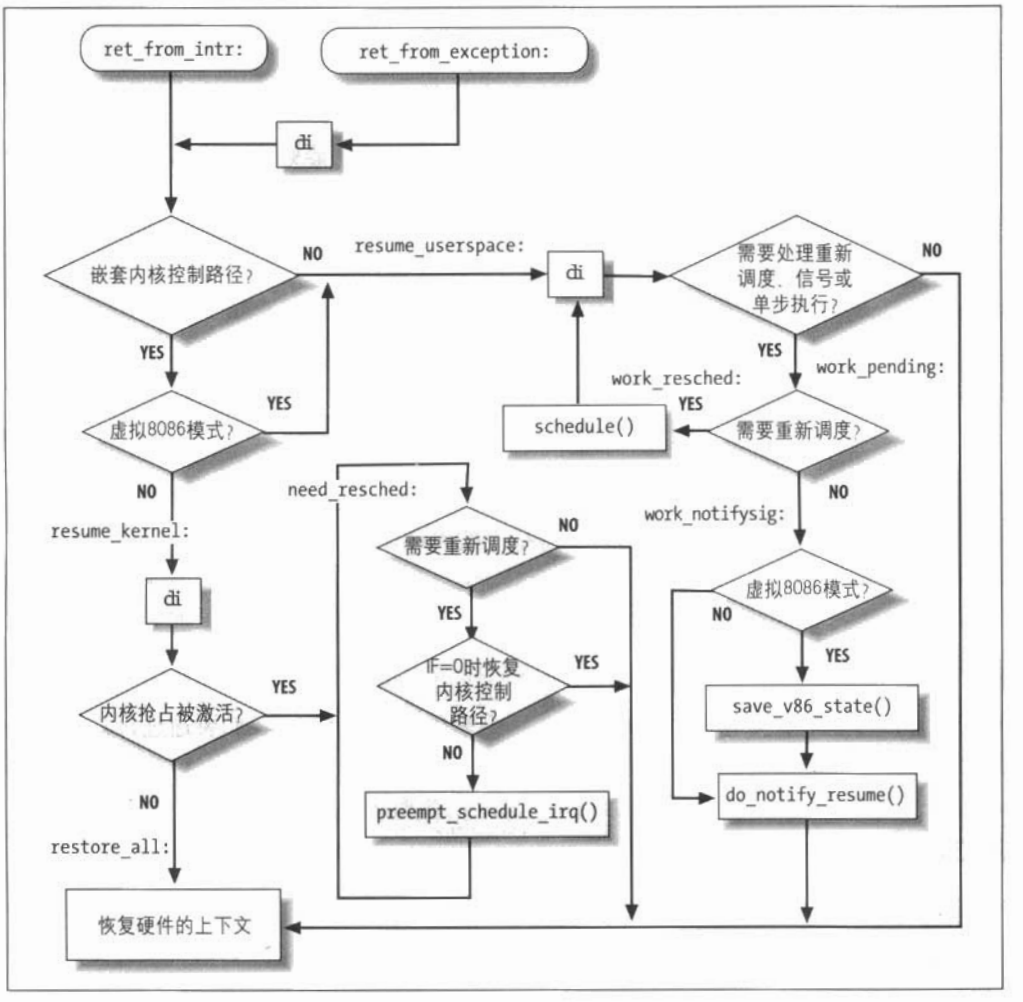


图 4-6：从中断和异常返回

接下来，要根据发生中断或异常时压入栈中的cs和eflags寄存器的值，来确定被中断的程序在中断发生时是否运行在用户态，或确定是否设置了eflags的VM标志(注18)。在任何一种情况下，代码的执行就跳转到resume_userspace处。否则，代码的执行跳转到resume_kernel处。

注18： 这个标志允许程序在虚拟8086模式下执行；更详细的内容参见Pentium手册。

恢复内核控制路径

如果被恢复的程序运行在内核态，就执行resume_kernel处的汇编语言代码：

```
resume_kernel:
    cli                 ; these three instructions are
    cmpl $0, 0x14(%ebp) ; missing if kernel preemption
    jz need_resched     ; is not supported
restore_all:
    popl %ebx
    popl %ecx
    popl %edx
    popl %esi
    popl %edi
    popl %ebp
    popl %eax
    popl %ds
    popl %es
    addl $4, %esp
    iret
```

如果thread_info描述符的preempt_count字段为0（允许内核抢占），则内核跳转到need_resched处，否则，被中断的程序重新开始执行。函数用中断和异常开始时保存的值装载寄存器，然后通过执行iret指令结束其控制。

检查内核抢占

执行检查内核抢占的代码时，所有没执行完的内核控制路径都不是中断处理程序，否则preempt_count字段的值就会是大于0的。但是正如在本章“中断和异常处理程序的嵌套执行”一节所强调的，最多可能有两个与异常（其中一个正在结束）相关的内核控制路径。

```
need_resched:
    movl 0x8(%ebp), %ecx
    testb $(1<<TIF_NEED_RESCHED), %cl
    jz restore_all
    testl $0x00000200,0x30(%esp)
    jz restore_all
    call preempt_schedule_irq
    jmp need_resched
```

如果current->thread_info的flags字段中的TIF_NEED_RESCHED标志为0，说明没有需要切换的进程，因此，程序跳转到restore_all处。如果正在被恢复的内核控制路径是在禁用本地CPU的情况下运行，那么也跳转到restore_all处。在这种情况下，进程切换可能破坏内核数据结构（详情参见第五章“什么时候同步是必需的”一节）。

如果需要进行进程切换，就调用preempt_schedule_irq()函数：它设置preempt_count字段的PREEMPT_ACTIVE标志，把大内核锁计数器暂时置为-1（参见第五章“大内核锁”一节），打开本地中断并调用schedule()以选择另一个进程来运行。当前面的进程要恢复时，preempt_schedule_irq()使大内核计数器的值恢复为以前的值，清除PREEMPT_ACTIVE标志并禁用本地中断。但当前进程的TIF_NEED_RESCHED标志被设置，将继续调用schedule()函数。

恢复用户态程序

如果要恢复的程序原来运行在用户态，就跳转到resume_userspace处：

```
resume_userspace:
    cli
    movl 0x8(%ebp), %ecx
    andl $0x0000ff6e, %ecx
    je restore_all
   jmp work_pending
```

禁用本地中断之后，检测current->thread_info的flags字段的值。如果只设置了TIF_SYSCALL_TRACE、TIF_SYSCALL_AUDIT或TIF_SINGLESTEP标志，就不做任何其他的事情，只是跳转到restore_all，从而恢复用户态程序。

检测重调度标志

thread_info descriptor描述符的flags表示在恢复被中断的程序之前，需要完成额外的工作。

```
work_pending:
    testb $(1<<TIF_NEED_RESCHED), %cl
    jz work_notifysig
work_resched:
    call schedule
    cli
    jmp resume_userspace
```

如果进程切换请求被挂起，就调用schedule()选择另外一个进程投入运行。当前面的进程要恢复时，就跳转回到resume_userspace处。

处理挂起信号、虚拟8086模式和单步执行

除了处理进程切换请求，还有其他的工作需要处理：

```
work_notifysig:
    movl %esp, %eax
    testl $0x00020000, 0x30(%esp)
```

```
        je 1f

    work_notifysig_v86:
        pushl %ecx
        call save_v86_state
        popl %ecx
        movl %eax, %esp
    1:
        xorl %edx, %edx
        call do_notify_resume
        jmp restore_all
```

如果用户态程序eflags寄存器的VM控制标志被设置，就调用save_v86_state()函数在用户态地址空间建立虚拟8086模式的数据结构。然后，调用do_notify_resume()函数处理挂起信号和单步执行。最后，跳转到restore_all标记处，恢复被中断的程序。

第五章



内核同步

你可以把内核看作是不断对请求进行响应的服务器，这些请求可能来自在CPU上执行的进程，也可能来自发出中断请求的外部设备。我们用这个类比来强调内核的各个部分并不是严格按照顺序依次执行的，而是采用交错执行的方式。因此，这些请求可能引起竞争条件，而我们必须采用适当的同步机制对这种情况进行控制。有关这些主题的简要介绍可以参看第一章中的“Unix 内核概述”一节。

本章开始部分我们先回顾一下内核请求是何时以交错（interleave）的方式执行以及交错程度如何。然后我们将介绍内核中所实现的基本同步机制，并说明通常情况下如何应用它们。最后，我们给出了几个实际的例子。

内核如何为不同的请求提供服务

为了更好地理解内核代码是如何执行的，我们把内核看作必须满足两种请求的侍者：一种请求来自顾客，另一种请求来自数量有限的几个不同的老板。对不同的请求，侍者采用如下的策略：

1. 老板提出请求时，如果侍者正空闲，则侍者开始为老板服务。
2. 如果老板提出请求时侍者正在为顾客服务，那么侍者停止为顾客服务，开始为老板服务。
3. 如果一个老板提出请求时侍者正在为另一个老板服务，那么侍者停止为第一个老板提供服务，而开始为第二个老板服务，服务完毕再继续为第一个老板服务。

4. 一个老板可能命令侍者停止正在为顾客提供的服务。侍者在完成对老板最近请求的服务之后，可能会暂时不理会原来的顾客而去为新选中的顾客服务。

侍者提供的服务对应于CPU处于内核态时所执行的代码。如果CPU在用户态执行，则侍者被认为处于空闲状态。

老板的请求相当于中断，而顾客的请求相当于用户态进程发出的系统调用或异常。正如我们将在第十章详细描述的，请求内核服务的用户态进程必须发出适当的指令（在80x86上是`int $0x80`或`sysenter`指令）。这些指令引起一个异常，它迫使CPU从用户态切换到内核态。在本章的其余部分，我们把系统调用和通常的异常都笼统地表示为“异常”。

细心的读者已经把前三条原则和第四章“中断和异常处理程序的嵌套执行”一节所描述的内核控制路径的嵌套联系起来了。第四条原则与Linux 2.6内核中最有趣的新特点相关，即内核抢占（kernel preemption）。

内核抢占

- 要给内核抢占下一个精确的定义简直太困难了。作为第一种尝试，我们说：如果进程正执行内核函数时，即它在内核态运行时，允许发生内核切换（被替换的进程是正执行内核函数的进程），这个内核就是抢占的。遗憾的是，在Linux中（在所有其他的操作系统中也一样），情况要复杂得多。

  - 无论在抢占内核还是非抢占内核中，运行在内核态的进程都可以自动放弃CPU，比如，其原因可能是，进程由于等待资源而不得不转入睡眠状态。我们将把这种进程切换称为计划性进程切换。但是，抢占式内核在响应引起进程切换的异步事件（例如唤醒高优先权进程的中断处理程序）的方式上与非抢占的内核是有差别的，我们将把这种进程切换称做强制性进程切换。
  - 所有的进程切换都由宏`switch_to`来完成。在抢占内核和非抢占内核中，当进程执行完某些具有内核功能的线程，而且调度程序被调用后，就发生进程切换。不过，在非抢占内核中，当前进程是不可能被替换的，除非它打算切换到用户态（参见第三章“执行进程切换“一节）。

因此，抢占内核的主要特点是：一个在内核态运行的进程，可能在执行内核函数期间被另外一个进程取代。

让我们举一对实例来说明抢占内核和非抢占内核的区别。

在进程A执行异常处理程序时（肯定是在内核态），一个具有较高优先级的进程B变为可执行状态。这种情况是有可能出现的，例如，发生了中断请求而且相应的处理程序唤

醒了进程B。如果内核是抢占的，就会发生强制性进程切换，让进程B取代进程A。异常处理程序的执行被暂停，直到调度程序再次选择进程A时才恢复它的执行。相反，如果内核是非抢占的，在进程A完成异常处理程序的执行之前是不会发生进程切换的，除非进程A自动放弃CPU。

再看另外一个例子，考虑一个执行异常处理程序的进程已经用完了它的时间配额（参见第七章“scheduler_tick()”函数一节）的情况。如果内核是抢占的，进程可能会立即被取代，但如果内核是非抢占的，进程继续运行直到它执行完异常处理程序或自动放弃CPU。

使内核可抢占的目的是减少用户态进程的分派延迟（*dispatch latency*），即从进程变为可执行状态到它实际开始运行之间的时间间隔。内核抢占对执行及时被调度的任务（如：硬件控制器、环境监视器、电影播放器等等）的进程确实是有好处的，因为它降低了这种进程被另一个运行在内核态的进程延迟的风险。

使Linux 2.6内核具有可抢占的特性无需对支持非抢占的旧内核在设计上做太大的改变，正如在第四章“从中断和异常返回”一节中所描述的，当被current_thread_info()宏所引用的thread_info描述符的preempt_count字段大于0时，就禁止内核抢占。如第四章中的表4-10所示，该字段的编码对应三个不同的计数器，因此它在如下任何一种情况发生时，取值都大于0：

1. 内核正在执行中断服务例程。
2. 可延迟函数被禁止（当内核正在执行软中断或tasklet时经常如此）。
3. 通过把抢占计数器设置为正数而显式地禁用内核抢占。

上面的原则告诉我们：只有当内核正在执行异常处理程序（尤其是系统调用），而且内核抢占没有被显式地禁用时，才可能抢占内核。此外，正如在第四章“从中断和异常返回”一节中所描述的，本地CPU必须打开本地中断，否则无法完成内核抢占。

表5-1中列出了一些简单的宏，它们处理preempt_count字段的抢占计数器。

表5-1：处理抢占计数器子字段的宏

宏	说明
preempt_count()	在thread_info描述符中选择preempt_count字段
preempt_disable()	使抢占计数器的值加1
preempt_enable_no_resched()	使抢占计数器的值减1

表5-1：处理抢占计数器子字段的宏（续）

宏	说明
preempt_enable()	使抢占计数器的值减1，并在thread_info描述符的TIF_NEED_RESCHED标志被置为1的情况下，调用preempt_schedule()
get_cpu()	与preempt_disable()相似，但要返回本地CPU的数量
put_cpu()	与preempt_enable()相同
put_cpu_no_resched()	与preempt_enable_no_resched()相同

preempt_enable()宏递减抢占计数器，然后检查TIF_NEED_RESCHED标志是否被设置（参见第四章中的表4-15）。在这种情况下，进程切换请求是挂起的，因此宏调用preempt_schedule()函数，它本质上执行下面的代码：

```
if (!current_thread_info->preempt_count && !irqs_disabled()) {
    current_thread_info->preempt_count = PREEMPT_ACTIVE;
    schedule();
    current_thread_info->preempt_count = 0;
}
```

该函数检查是否允许本地中断，以及当前进程的preempt_count字段是否为0，如果两个条件都为真，它就调用schedule()选择另外一个进程来运行。因此，内核抢占可能在结束内核控制路径（通常是一个中断处理程序）时发生，也可能在异常处理程序调用preempt_enable()重新允许内核抢占时发生。正如我们将在本章稍后的“禁止和激活可延迟函数”一节所看到的，内核抢占也可能发生在启用可延迟函数的时候。

在结束本节内容时，我们提醒大家注意：内核抢占会引起不容忽视的开销。因此，Linux 2.6独具特色地允许用户在编译内核时通过设置选项来禁用或启用内核抢占。

什么时候同步是必需的

第一章介绍了竞争条件和进程临界区的概念。这些定义同样适用于内核控制路径。在本章，当计算的结果依赖于两个或两个以上的交叉内核控制路径的嵌套方式时，可能出现竞争条件。临界区是一段代码，在其他的内核控制路径能够进入临界区前，进入临界区的内核控制路径必须全部执行完这段代码。

交叉内核控制路径使内核开发者的工作变得复杂了：他们必须特别小心地识别出异常处理程序、中断处理程序、可延迟函数和内核线程中的临界区。一旦临界区被确定，就必须对其采用适当的保护措施，以确保在任意时刻只有一个内核控制路径处于临界区。

例如，假设两个不同的中断处理程序要访问同一个包含了几个相关变量的数据结构，比如一个缓冲区和一个表示缓冲区大小的整型变量。所有影响该数据结构的语句都必须放入一个单独的临界区。如果是单CPU的系统，可以采取访问共享数据结构时关闭中断的方式来实现临界区，因为只有在开中断的情况下，才可能发生内核控制路径的嵌套。

另外，如果相同的数据结构仅被系统调用服务例程所访问，而且系统中只有一个CPU，就可以非常简单地通过在访问共享数据结构时禁用内核抢占功能来实现临界区。

正如你们所预料的，在多处理器系统中，情况要复杂得多。由于许多CPU可能同时执行内核路径，因此内核开发者不能假设只要禁用内核抢占功能，而且中断、异常和软中断处理程序都没有访问过该数据结构，就能保证这个数据结构能够安全地被访问。

我们将在下一节看到内核提供了各种不同的同步技术。内核设计者通过选择最有效的技术解决了所有的同步难题。

什么时候同步是不必要的

前一章所讨论的一些设计上的选择在某种程度上简化了内核控制路径的同步。让我们简单地回忆一下：

- 所有的中断处理程序响应来自PIC的中断并禁用IRQ线。此外，在中断处理程序结束之前，不允许产生相同的中断事件。
- 中断处理程序、软中断和tasklet既不可以被抢占也不能被阻塞，所以它们不可能长时间处于挂起状态。在最坏的情况下，它们的执行将有轻微的延迟，因为在其执行的过程中可能发生其他的中断（内核控制路径的嵌套执行）。
- 执行中断处理的内核控制路径不能被执行可延迟函数或系统调用服务例程的内核控制路径中断。
- 软中断和 tasklet 不能在一个给定的 CPU 上交错执行。
- 同一个 tasklet 不可能同时在几个 CPU 上执行。

以上的每一种设计选择都可以被看做是一种约束，它能使一些内核函数的编码变得更容易。下面是一些可能简化了的例子：

- 中断处理程序和 tasklet 不必编写成可重入的函数。
- 仅被软中断和 tasklet 访问的每 CPU 变量不需要同步。
- 仅被一种 tasklet 访问的数据结构不需要同步。

本章接下来的部分描述在需要同步的时候应该做些什么，也就是：如何避免由于对共享数据的不安全访问导致的数据崩溃。

同步原语

现在我们考察一下在避免共享数据之间的竞争条件时，内核控制路径是如何交错执行的。表5-2列出了Linux内核使用的同步技术。“适用范围”一栏表示同步技术是适用于系统中的所有CPU还是单个CPU。例如，本地中断的禁止只适用于一个CPU（系统中的其他CPU不受影响）；相反，原子操作影响系统中的所有CPU（当访问同一个数据结构时，几个CPU上的原子操作不能交错）。

表5-2：内核使用的各种同步技术

技术	说明	适用范围
每CPU变量	在CPU之间复制数据结构	所有CPU
原子操作	对一个计数器原子地“读－修改－写”的指令	所有CPU
内存屏障	避免指令重新排序	本地CPU或所有CPU
自旋锁	加锁时忙等	所有CPU
信号量	加锁时阻塞等待（睡眠）	所有CPU
顺序锁	基于访问计数器的锁	所有CPU
本地中断的禁止	禁止单个CPU上的中断处理	本地CPU
本地软中断的禁止	禁止单个CPU上的可延迟函数处理	本地CPU
读－拷贝－更新(RCU)	通过指针而不是锁来访问共享数据结构	所有CPU

现在，让我们简要地描述每种同步技术。在后面“对内核数据结构的同步访问”一节，我们会说明如何把这些同步技术组合在一起来有效地保护内核数据结构。

每CPU变量

最好的同步技术是把设计不需要同步的内核放在首位。正如我们将要看到的，事实上每一种显式的同步原语都有不容忽视的性能开销。

最简单也是最重要的同步技术包括把内核变量声明为每CPU变量（*per-cpu variable*）。每CPU变量主要是数据结构的数组，系统的每个CPU对应数组的一个元素。

一个CPU不应该访问与其他CPU对应的数组元素，另外，它可以随意读或修改它自己

的元素而不用担心出现竞争条件，因为它是唯一有资格这么做的CPU。但是，这也意味着每CPU变量基本上只能在特殊情况下使用，也就是当它确定在系统的CPU上的数据在逻辑上是独立的时候。

每CPU的数组元素在主存中被排列以使每个数据结构存放在硬件高速缓存的不同行(参见第二章“硬件高速缓存”一节)，因此，对每CPU数组的并发访问不会导致高速缓存行的窃用和失效（这种操作会带来昂贵的系统开销)。

虽然每CPU变量为来自不同CPU的并发访问提供保护，但对来自异步函数（中断处理程序和可延迟函数）的访问不提供保护，在这种情况下需要另外的同步原语。

此外，在单处理器和多处理器系统中，内核抢占都可能使每CPU变量产生竞争条件。总的原则是内核控制路径应该在禁用抢占的情况下访问每CPU变量。作为一个例子，简单地考虑一下在下面这种情况下会产生什么后果——一个内核控制路径获得了它的每CPU变量本地副本的地址，然后它因被抢占而转移到另外一个CPU上，但仍然引用原来CPU元素的地址。

表5-3列出了内核提供使用每CPU变量的函数和宏。

表5-3：为每CPU变量提供的函数和宏

宏或函数名	说明
DEFINE_PER_CPU(type, name)	静态分配一个每CPU数组，数组名为name,结构类型为type
per_cpu(name, cpu)	为CPU选择一个每CPU数组元素，CPU由参数cpu指定，数组名称为name
_ _get_cpu_var(name)	选择每CPU数组name的本地CPU元素
get_cpu_var(name)	先禁用内核抢占，然后在每CPU数组name中，为本地CPU选择元素
put_cpu_var(name)	启用内核抢占（不使用name）
alloc_percpu(type)	动态分配type类型数据结构的每CPU数组，并返回它的地址
free_percpu(pointer)	释放被动态分配的每CPU数组，pointer指示其地址
per_cpu_ptr(pointer, cpu)	返回每CPU数组中与参数cpu对应的CPU元素地址，参数pointer给出数组地址

原子操作

若干汇编语言指令具有“读－修改－写”类型——也就是说，它们访问存储器单元两次，第一次读原值，第二次写新值。

假定运行在两个CPU上的两个内核控制路径试图通过执行非原子操作来同时“读－修改－写”同一存储器单元。首先，两个CPU都试图读同一单元，但是存储器仲裁器（对访问RAM芯片的操作进行串行化的硬件电路）插手，只允许其中的一个访问而让另一个延迟。然而，当第一个读操作已经完成后，延迟的CPU从那个存储器单元正好读到同一个（旧）值。然后，两个CPU都试图向那个存储器单元写一新值，总线存储器访问再一次被存储器仲裁器串行化，最终，两个写操作都成功。但是，全局的结果是不对的，因为两个CPU写入同一（新）值。因此，两个交错的“读－修改－写”操作成了一个单独的操作。

避免由于“读－修改－写”指令引起的竞争条件的最容易的办法，就是确保这样的操作在芯片级是原子的。任何一个这样的操作都必须以单个指令执行，中间不能中断，且避免其他的CPU访问同一存储器单元。这些很小的原子操作（*atomic operations*）可以建立在其他更灵活机制的基础之上以创建临界区。

让我们根据那个分类来回顾一下80x86的指令：

- 进行零次或一次对齐内存访问的汇编指令是原子的（注1）。
- 如果在读操作之后、写操作之前没有其他处理器占用内存总线，那么从内存中读取数据、更新数据并把更新后的数据写回内存中的这些“读－修改－写”汇编语言指令（例如`inc`或`dec`）是原子的。当然，在单处理器系统中，永远都不会发生内存总线窃用的情况。
- 操作码前缀是`lock`字节（`0xf0`）的“读－修改－写”汇编语言指令即使在多处理器系统中也是原子的。当控制单元检测到这个前缀时，就“锁定”内存总线，直到这条指令执行完成为止。因此，当加锁的指令执行时，其他处理器不能访问这个内存单元。
- 操作码前缀是一个`rep`字节（`0xf2`，`0xf3`）的汇编语言指令不是原子的，这条指令强行让控制单元多次重复执行相同的指令。控制单元在执行新的循环之前要检查挂起的中断。

注1：当数据项的地址是以字节为单位的整数倍时，数据项在内存中被对齐。例如，一个对齐的短整数的地址必须是2的整数倍，而对齐的整数的地址必须是4的整数倍。一般来说，非对齐的内存访问不是原子的。

在你编写C代码程序时，并不能保证编译器会为a=a+1或甚至像a++这样的操作使用一个原子指令。因此，Linux内核提供了一个专门的atomic_t类型（一个原子访问计数器）和一些专门的函数和宏（参见表5-4），这些函数和宏作用于atomic_t类型的变量，并当作单独的、原子的汇编语言指令来使用。在多处理器系统中，每条这样的指令都有一个lock字节的前缀。

表5-4：Linux中的原子操作

函数	说明
atomic_read(v)	返回 *v
atomic_set(v,i)	把 *v 置成 i
atomic_add(i,v)	给 *v 增加 i
atomic_sub(i,v)	从 *v 中减去 i
atomic_sub_and_test(i, v)	从 *v 中减去 i，如果结果为0，则返回1；否则，返回0
atomic_inc(v)	把1加到 *v
atomic_dec(v)	从 *v 减1
atomic_dec_and_test(v)	从 *v 减1，如果结果为0，则返回1；否则，返回0
atomic_inc_and_test(v)	把1加到 *v，如果结果为0，则返回1；否则，返回0
atomic_add_negative(i, v)	把 i 加到 *v，如果结果为负，则返回1；否则，返回0
atomic_inc_return(v)	把1加到 *v，返回 *v 的新值
atomic_dec_return(v)	从 *v 减1，返回 *v 的新值
atomic_add_return(i, v)	把 i 加到 *v，返回 *v 的新值
atomic_sub_return(i, v)	从 *v 减 i，返回 *v 的新值

另一类原子函数操作作用于位掩码（参见表5-5）。

表5-5：Linux中的原子位处理函数

函数	说明
test_bit(nr, addr)	返回 *addr 的第 nr 位的值
set_bit(nr, addr)	设置 *addr 的第 nr 位
clear_bit(nr, addr)	清 *addr 的第 nr 位
change_bit(nr, addr)	转换 *addr 的第 nr 位
test_and_set_bit(nr, addr)	设置 *addr 的第 nr 位，并返回它的原值
test_and_clear_bit(nr, addr)	清 *addr 的第 nr 位，并返回它的原值
test_and_change_bit(nr, addr)	转换 *addr 的第 nr 位，并返回它的原值

表5-5：Linux中的原子位处理函数（续）

函数	说明
atomic_clear_mask(mask, addr)	清mask指定的*addr的所有位
atomic_set_mask(mask, addr)	设置mask指定的*addr的所有位

优化和内存屏障

当使用优化的编译器时，你千万不要认为指令会严格按它们在源代码中出现的顺序执行。例如，编译器可能重新安排汇编语言指令以使寄存器以最优的方式使用。此外，现代CPU通常并行地执行若干条指令，且可能重新安排内存访问。这种重新排序可以极大地加速程序的执行。

然而，当处理同步时，必须避免指令重新排序。如果放在同步原语之后的一条指令在同步原语本身之前执行，事情很快就会变得失控。事实上，所有的同步原语起优化和内存屏障的作用。

优化屏障（*optimization barrier*）原语保证编译程序不会混淆放在原语操作之前的汇编语言指令和放在原语操作之后的汇编语言指令，这些汇编语言指令在C中都有对应的语句。在Linux中，优化屏障就是barrier()宏，它展开为asm volatile("":::"memory")。指令asm告诉编译程序要插入汇编语言片段（这种情况下为空）。volatile关键字禁止编译器把asm指令与程序中的其他指令重新组合。memory关键字强制编译器假定RAM中的所有内存单元已经被汇编语言指令修改；因此，编译器不能使用存放在CPU寄存器中的内存单元的值来优化asm指令前的代码。注意，优化屏障并不保证不使当前CPU把汇编语言指令混在一起执行——这是内存屏障的工作。

内存屏障（*memory barrier*）原语确保，在原语之后的操作开始执行之前，原语之前的操作已经完成。因此，内存屏障类似于防火墙，让任何汇编语言指令都不能通过。

在80x86处理器中，下列种类的汇编语言指令是“串行的”，因为它们起内存屏障的作用：

- 对I/O端口进行操作的所有指令。
- 有lock前缀的所有指令（参见“原子操作”一节）。
- 写控制寄存器、系统寄存器或调试寄存器的所有指令（例如，cli和sti，用于修改eflags寄存器的IF标志的状态）。
- 在Pentium 4微处理器中引入的汇编语言指令lfence、sfence和mfence，它们分别有效地实现读内存屏障、写内存屏障和读－写内存屏障。

- 少数专门的汇编语言指令，终止中断处理程序或异常处理程序的iret指令就是其中的一个。

Linux使用六个内存屏障原语，如表5-6所示。这些原语也被当作优化屏障，因为我们必须保证编译程序不在屏障前后移动汇编语言指令。“读内存屏障”仅仅作用于从内存读的指令，而“写内存屏障”仅仅作用于写内存的指令。内存屏障既用于多处理器系统，也用于单处理器系统。当内存屏障应该防止仅出现于多处理器系统上的竞争条件时，就使用smp_xxx()原语；在单处理器系统上，它们什么也不做。其他的内存屏障防止出现在单处理器和多处理器系统上的竞争条件。

表5-6：Linux中的内存屏障

宏	说明
mb()	适用于MP和UP的内存屏障
rmb()	适用于MP和UP的读内存屏障
wmb()	适用于MP和UP的写内存屏障
smp_mb()	仅适用于MP的内存屏障
smp_rmb()	仅适用于MP的读内存屏障
smp_wmb()	仅适用于MP的写内存屏障

内存屏障原语的实现依赖于系统的体系结构。在80x86微处理器上，如果CPU支持lfence汇编语言指令，就把rmb()宏展开为asm volatile("lfence")，否则就展开为asm volatile("lock;addl $0,0(%%esp)":::"memory")。asm指令告诉编译器插入一些汇编语言指令并起优化屏障的作用。lock;addl $0,0(%%esp)汇编指令把0加到栈顶的内存单元；这条指令本身没有价值，但是，lock前缀使得这条指令成为CPU的一个内存屏障。

Intel上的wmb()宏实际上更简单，因为它展开为barrier()。这是因为Intel处理器从不对写内存访问重新排序，因此，没有必要在代码中插入一条串行化汇编指令。不过，这个宏禁止编译器重新组合指令。

注意，在多处理器系统上，在前一节“原子操作”中描述的所有原子操作都起内存屏障的作用，因为它们使用了lock字节。

自旋锁

一种广泛应用的同步技术是加锁（*locking*）。当内核控制路径必须访问共享数据结构或进入临界区时，就需要为自己获取一把“锁”。由锁机制保护的资源非常类似于限制于房间内的资源，当某人进入房间时，就把门锁上。如果内核控制路径希望访问资源，就

试图获取钥匙“打开门”。当且仅当资源空闲时，它才能成功。然后，只要它还想使用这个资源，门就依然锁着。当内核控制路径释放了锁时，门就打开，另一个内核控制路径就可以进入房间。

图5-1显示了锁的使用。5个内核控制路径（P_0，P_1，P_2，P_3和P_4）试图访问两个临界区（C_1和C_2）。内核控制路径P_0正在C_1中，而P_2和P_4正等待进入C_1。同时，P_1正在C_2中，而P_3正在等待进入C_2。注意P_0和P_1可以并行运行。临界区C_3的锁现在打开着，因为没有内核控制路径需要进入C_3。

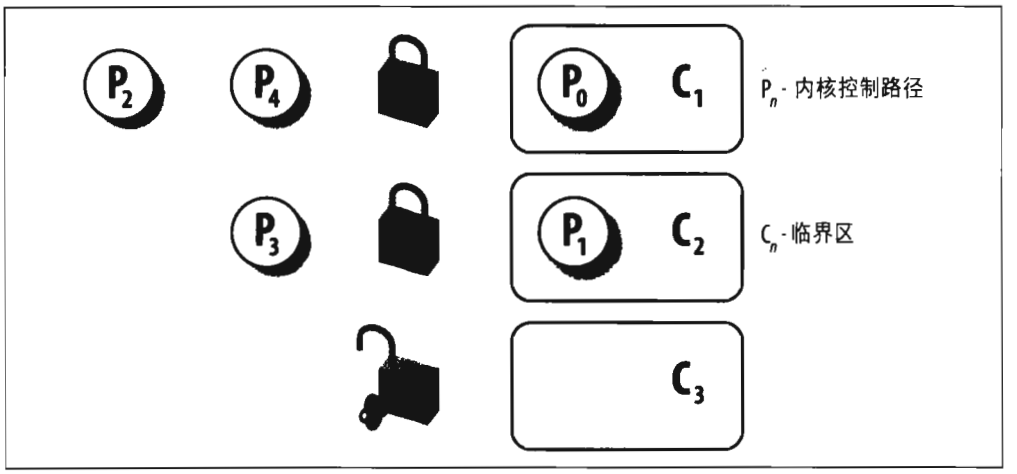


图5-1：用几个锁保护临界区

自旋锁（*spin lock*）是用来在多处理器环境中工作的一种特殊的锁。如果内核控制路径发现自旋锁“开着”，就获取锁并继续自己的执行。相反，如果内核控制路径发现锁由运行在另一个CPU上的内核控制路径“锁着”，就在周围“旋转”，反复执行一条紧凑的循环指令，直到锁被释放。

自旋锁的循环指令表示“忙等”。即使等待的内核控制路径无事可做（除了浪费时间），它也在CPU上保持运行。不过，自旋锁通常非常方便，因为很多内核资源只锁1毫秒的时间片段；所以说，释放CPU和随后又获得CPU都不会消耗多少时间。

一般来说，由自旋锁所保护的每个临界区都是禁止内核抢占的。在单处理器系统上，这种锁本身并不起锁的作用，自旋锁原语仅仅是禁止或启用内核抢占。请注意，在自旋锁忙等期间，内核抢占还是有效的，因此，等待自旋锁释放的进程有可能被更高优先级的进程替代。

在Linux中，每个自旋锁都用`spinlock_t`结构表示，其中包含两个字段：

slock

该字段表示自旋锁的状态：值为1表示“未加锁”状态，而任何负数和0都表示“加锁”状态。

break_lock

表示进程正在忙等自旋锁（只在内核支持SMP和内核抢占的情况下使用该标志）。

表5-7所示的六个宏用于初始化、测试及设置自旋锁。所有这些宏都是基于原子操作的，这样可以保证即使有多个运行在不同CPU上的进程试图同时修改自旋锁，自旋锁也能够被正确地更新（注2）。

表5-7：自旋锁宏

宏	说明
spin_lock_init()	把自旋锁置为1（未锁）
spin_lock()	循环，直到自旋锁变为1（未锁），然后，把自旋锁置为0（锁上）
spin_unlock()	把自旋锁置为1（未锁）
spin_unlock_wait()	等待，直到自旋锁变为1（未锁）
spin_is_locked()	如果自旋锁被置为1（未锁），返回0；否则，返回1
spin_trylock()	把自旋锁置为0（锁上），如果原来锁的值是1，则返回1；否则，返回0

具有内核抢占的spin_lock宏

让我们来详细讨论用于请求自旋琐的spin_lock宏。下面的描述都是针对支持SMP系统的抢占式内核的。该宏获取自旋琐的地址slp作为它的参数，并执行下面的操作：

1. 调用preempt_disable()以禁用内核抢占。
2. 调用函数_raw_spin_trylock()，它对自旋锁的slock字段执行原子性的测试和设置操作。该函数首先执行等价于下列汇编语言片段的一些指令：

```
movb $0, %al
xchgb %al, slp->slock
```

汇编语言指令xchg原子性地交换8位寄存器%al（存0）和slp->slock指示的内存单元的内容。随后，如果存放在自旋锁中的旧值（在xchg指令执行之后存放在%al中）是正数，函数就返回1，否则返回0。

注2： 具有讽刺意味的是，自旋锁是全局的，因此对它本身必须进行保护以防止并发访问。

3. 如果自旋锁中的旧值是正数，宏结束：内核控制路径已经获得自旋锁。

4. 否则，内核控制路径无法获得自旋锁，因此宏必须执行循环一直到在其他CPU上运行的内核控制路径释放自旋锁。调用preempt_enable()递减在第1步递增了的抢占计数器。如果在执行spin_lock宏之前内核抢占被启用，那么其他进程此时可以取代等待自旋锁的进程。

5. 如果break_lock字段等于0，则把它设置为1。通过检测该字段，拥有锁并在其他CPU上运行的进程可以知道是否有其他进程在等待这个锁。如果进程把持某个自旋锁的时间太长，它可以提前释放锁以使等待相同自旋锁的进程能够继续向前运行。

6. 执行等待循环：

```
while (spin_is_locked(slp) && slp->break_lock)
   cpu_relax();
```

   宏cpu_relax()简化为一条pause汇编语言指令。在Pentium 4模型中引入了这条指令以优化自旋锁循环的执行。通过引入一个很短的延迟，加快了紧跟在锁后面的代码的执行并减少了能源消耗。pause与早先的80x86微处理器模型是向后兼容的，因为它对应rep；nop指令，也就是对应空操作。

7. 跳转回到第1步，再次试图获取自旋锁。

非抢占式内核中的spin_lock宏

如果在内核编译时没有选择内核抢占选项，spin_lock宏就与前面描述的spin_lock宏有很大的区别。在这种情况下，宏生成一个汇编语言程序片段，它本质上等价于下面紧凑的忙等待（注3）：

```
1: lock; decb slp->slock
   jns  3f
2: pause
   cmpb $0,slp->slock
   jle 2b
   jmp 1b
3:
```

汇编语言指令decb递减自旋锁的值，该指令是原子的，因为它带有lock字节前缀。随后检测符号标志，如果它被清0，说明自旋锁被设置为1（未锁），因此从标记3处继续正常执行（后缀f表示标签是“向前的”，它在程序的后面出现）。否则，在标签2处（后

注3：忙等待循环的实际实现要稍微复杂些。标号2处的代码（仅在自旋锁忙时被执行）包含在另外的代码段中，以便在大多数情况下（自旋锁已经被释放）不要用不执行的代码填充硬件高速缓存。在我们的讨论中，忽略了这些优化细节。

缀b表示“向后的”标签）执行紧凑循环直到自旋锁出现正值。然后从标签1处开始重新执行，因为不检查其他的处理器是否抢占了锁就继续执行是不安全的。

spin_unlock宏

spin_unlock宏释放以前获得的自旋锁，它本质上执行下列汇编语言指令：

```
movb $1, slp->slock
```

并在随后调用preempt_enable()（如果不支持内核抢占，preempt_enable()什么都不做）。注意，因为现在的80x86微处理器总是原子地执行内存中的只写访问，所以不使用lock字节。

读/写自旋锁

读/写自旋锁的引入是为了增加内核的并发能力。只要没有内核控制路径对数据结构进行修改，读/写自旋锁就允许多个内核控制路径同时读同一数据结构。如果一个内核控制路径想对这个结构进行写操作，那么它必须首先获取读/写锁的写锁，写锁授权独占访问这个资源。当然，允许对数据结构并发读可以提高系统性能。

图5-2显示有两个受读/写锁保护的临界区（C_1和C_2）。内核控制路径R_0和R_1正在同时读取C_1中的数据结构，而W_0正等待获取写锁。内核控制路径W_1正对C_2中的数据结构进行写操作，而R_2和W_2分别等待获取读锁和写锁。

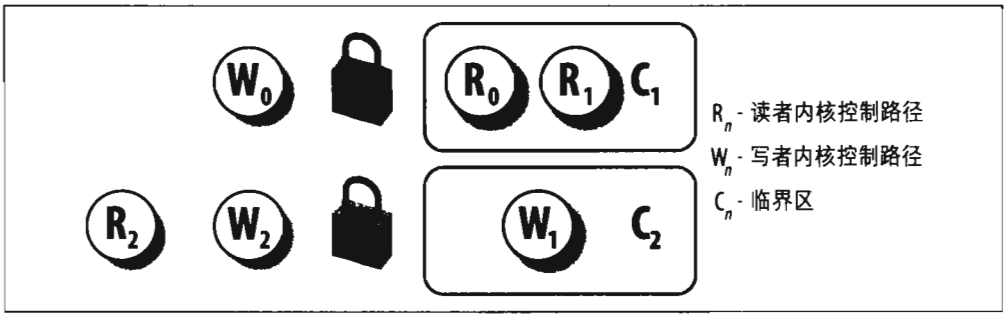


图5-2：读/写自旋锁

每个读/写自旋锁都是一个rwlock_t结构，其lock字段是一个32位的字段，分为两个不同的部分：

- 24位计数器，表示对受保护的数据结构并发地进行读操作的内核控制路径的数目。这个计数器的二进制补码存放在这个字段的0～23位。

- “未锁”标志字段，当没有内核控制路径在读或写时设置该位，否则清0。这个“未锁”标志存放在lock字段的第24位。

注意，如果自旋锁为空（设置了“未锁”标志且无读者），那么lock字段的值为0x01000000；如果写者已经获得自旋锁（“未锁”标志清0且无读者），那么lock字段的值为0x00000000；如果一个、两个或多个进程因为读获取了自旋锁，那么，lock字段的值为0x00ffffff，0x00fffffe等（“未锁”标志清0，读者个数的二进制补码在0～23位上）。与spinlock_t结构一样，rwlock_t结构也包括break_lock字段。

rwlock_init宏把读/写自旋锁的lock字段初始化为0x01000000（“未锁”），把break_lock初始化为0。

为读获取和释放一个锁

read_lock宏，作用于读/写自旋锁的地址rwlp，与前面一节所描述的spin_lock宏非常相似。如果编译内核时选择了内核抢占选项，read_lock宏执行与spin_lock()非常相似的操作，只有一点不同：该宏执行_raw_read_trylock()函数以在第2步有效地获取读/写自旋锁。

```
int _raw_read_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    atomic_dec(count);
    if (atomic_read(count) >= 0)
        return 1;
    atomic_inc(count);
    return 0;
}
```

读/写锁计数器lock字段是通过原子操作来访问的。注意，尽管如此，但整个函数对计数器的操作并不是原子性的。例如，在用if语句完成对计数器值的测试之后并返回1之前，计数器的值可能发生变化。不过，函数能够正常工作：实际上，只有在递减之前计数器的值不为0或负数的情况下，函数才返回1，因为计数器等于0x01000000表示没有任何进程占用锁，等于0x00ffffff表示有一个读者，等于0x00000000表示有一个写者。

如果编译内核时没有选择内核抢占选项，read_lock宏产生下面的汇编语言代码：

```
    movl $rwlp->lock,%eax
    lock; subl $1,(%eax)
    jns 1f
    call _ _read_lock_failed
1:
```

这里，__read_lock_failed()是下列汇编语言函数：

```
__read_lock_failed:
    lock; incl (%eax)
1:  pause
    cmpl $1,(%eax)
    js 1b
    lock; decl (%eax)
    js _ _read_lock_failed
    ret
```

read_lock宏原子地把自旋锁的值减1，由此增加读者的个数。如果递减操作产生一个非负值，就获得自旋锁，否则，调用__read_lock_failed()函数。该函数原子地增加lock字段以取消由read_lock宏执行的递减操作，然后循环，直到lock字段变为正数（大于或等于0）。接下来，__read_lock_failed()又试图获取自旋锁（正好在cmpl指令之后，另一个内核控制路径可能为写获取自旋锁）。

释放读自旋锁是相当简单的，因为read_unlock宏只需使用汇编语言指令简单地增加lock字段的计数器：

```
lock; incl rwlp->lock
```

以减少读者的计数，然后调用preempt_enable()重新启用内核抢占。

为写获取和释放一个锁

write_lock宏实现的方式与spin_lock()和read_lock()相似。例如，如果支持内核抢占，则该函数禁用内核抢占并通过调用_raw_write_trylock()立即获得锁。如果该函数返回0，说明锁已经被占用，因此，该宏像前面章节描述的那样重新启用内核抢占并开始忙等待循环。

_raw_write_trylock()函数描述如下：

```
int _raw_write_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    if (atomic_sub_and_test(0x01000000, count))
        return 1;
    atomic_add(0x01000000, count);
    return 0;
}
```

函数_raw_write_trylock()从读/写自旋锁的值中减去0x01000000，从而清除未上锁标志（第24位）。如果减操作产生0值（没有读者），则获取锁并返回1；否则，函数原子地在自旋锁的值上加0x01000000，以取消减操作。

释放写锁同样非常简单，因为write_unlock宏只需使用汇编语言指令lock; addl

$0x01000000,rwlp把lock字段中的“未锁”标识置位，然后再调用preempt_enable()即可。

顺序锁

当使用读/写自旋锁时，内核控制路径发出的执行read_lock或write_lock操作的请求具有相同的优先权：读者必须等待，直到写操作完成。同样地，写者也必须等待，直到读操作完成。

Linux 2.6中引入了顺序锁（*seqlock*），它与读/写自旋锁非常相似，只是它为写者赋予了较高的优先级：事实上，即使在读者正在读的时候也允许写者继续运行。这种策略的好处是写者永远不会等待（除非另外一个写者正在写），缺点是有些时候读者不得不反复多次读相同的数据直到它获得有效的副本。

每个顺序锁都是包括两个字段的seqlock_t结构：一个类型为spinlock_t的lock字段和一个整型的sequence字段，第二个字段是一个顺序计数器。每个读者都必须在读数据前后两次读顺序计数器，并检查两次读到的值是否相同，如果不相同，说明新的写者已经开始写并增加了顺序计数器，因此暗示读者刚读到的数据是无效的。

通过把SEQLOCK_UNLOCKED赋给变量seqlock_t或执行seqlock_init宏，把seqlock_t变量初始化为“未上锁”。写者通过调用write_seqlock()和write_sequnlock()获取和释放顺序锁。第一个函数获取seqlock_t数据结构中的自旋锁，然后使顺序计数器加1；第二个函数再次增加顺序计数器，然后释放自旋锁。这样可以保证写者在写的过程中，计数器的值是奇数，并且当没有写者在改变数据的时候，计数器的值是偶数。读者执行下面的临界区代码：

```
unsigned int seq;
do {
    seq = read_seqbegin(&seqlock);
    /* ...临界区... */
} while (read_seqretry(&seqlock, seq));
```

read_seqbegin()返回顺序锁的当前顺序号；如果局部变量seq的值是奇数（写者在read_seqbegin()函数被调用后，正更新数据结构），或seq的值与顺序锁的顺序计数器的当前值不匹配（当读者正执行临界区代码时，写者开始工作），read_seqretry()就返回1。

注意，当读者进入临界区时，不必禁用内核抢占；另一方面，由于写者获取自旋锁，所以它进入临界区时自动禁用内核抢占。

并不是每一种资源都可以使用顺序锁来保护。一般来说，必须在满足下述条件时才能使用顺序锁：

- 被保护的数据结构不包括被写者修改和被读者间接引用的指针（否则，写者可能在读者的眼鼻下就修改指针）。
- 读者的临界区代码没有副作用（否则，多个读者的操作会与单独的读操作有不同的结果）。

此外，读者的临界区代码应该简短，而且写者应该不常获取顺序锁，否则，反复的读访问会引起严重的开销。在Linux 2.6中，使用顺序锁的典型例子包括保护一些与系统时间处理相关的数据结构（参见第六章）。

读－拷贝－更新（RCU）

读－拷贝－更新(*RCU*)是为了保护在多数情况下被多个CPU读的数据结构而设计的另一种同步技术。RCU允许多个读者和写者并发执行（相对于只允许一个写者执行的顺序锁有了改进）。而且，RCU是不使用锁的，就是说，它不使用被所有CPU共享的锁或计数器，在这一点上与读/写自旋锁和顺序锁（由于高速缓存行窃用和失效而有很高的开销）相比，RCU具有更大的优势。

RCU是如何不使用共享数据结构而令人惊讶地实现多个CPU同步呢？其关键的思想包括限制RCP的范围，如下所述：

1. RCU只保护被动态分配并通过指针引用的数据结构。
2. 在被RCU保护的临界区中，任何内核控制路径都不能睡眠。

当内核控制路径要读取被RCU保护的数据结构时，执行宏`rcu_read_lock()`，它等同于`preempt_disable()`。接下来，读者间接引用该数据结构指针所对应的内存单元并开始读这个数据结构。正如在前面所强调的，读者在完成对数据结构的读操作之前，是不能睡眠的。用等同于`preempt_enable()`的宏`rcu_read_unlock()`标记临界区的结束。

我们可以想象，由于读者几乎不做任何事情来防止竞争条件的出现，所以写者不得不做得更多一些。事实上，当写者要更新数据结构时，它间接引用指针并生成整个数据结构的副本。接下来，写者修改这个副本。一但修改完毕，写者改变指向数据结构的指针，以使它指向被修改后的副本。由于修改指针值的操作是一个原子操作，所以旧副本和新副本对每个读者或写者都是可见的，在数据结构中不会出现数据崩溃。尽管如此，还需要内存屏障来保证：只有在数据结构被修改之后，已更新的指针对其他CPU才是可见的。如果把自旋锁与RCU结合起来以禁止写者的并发执行，就隐含地引入了这样的内存屏障。

然而，使用RCU技术的真正困难在于：写者修改指针时不能立即释放数据结构的旧副本。实际上，写者开始修改时，正在访问数据结构的读者可能还在读旧副本。只有在CPU上的所有（潜在的）读者都执行完宏rcu_read_unlock()之后，才可以释放旧副本。内核要求每个潜在的读者在下面的操作之前执行rcu_read_unlock()宏：

- CPU执行进程切换（参见前面的约束条件2）
- CPU开始在用户态执行
- CPU执行空循环（参见第三章“内核线程”一节）

对上述每种情况，我们说CPU已经经过了静止状态（*quiescent state*）。

写者调用函数call_rcu()来释放数据结构的旧副本。当所有的CPU都通过静止状态之后，call_rcu()接受rcu_head描述符（通常嵌在要被释放的数据结构中）的地址和将要调用的回调函数的地址作为参数。一旦回调函数被执行，它通常释放数据结构的旧副本。

函数call_rcu()把回调函数和其参数的地址存放在rcu_head描述符中，然后把描述符插入回调函数的每CPU（per-CPU）链表中。内核每经过一个时钟滴答（参见第六章“更新本地CPU统计数”一节）就周期性地检查本地CPU是否经过了一个静止状态。如果所有CPU都经过了静止状态，本地tasklet（它的描述符存放在每CPU变量rcu_tasklet中）就执行链表中的所有回调函数。

RCU是Linux 2.6中新加的功能，用在网络层和虚拟文件系统中。

信号量

我们在第一章“同步和临界区”一节引入了信号量。从本质上说，它们实现了一个加锁原语，即让等待者睡眠，直到等待的资源变为空闲。

实际上，Linux提供两种信号量：

- 内核信号量，由内核控制路径使用
- System V IPC信号量，由用户态进程使用

在本节，我们集中讨论内核信号量，而IPC信号量将在第十九章描述。

内核信号量类似于自旋锁，因为当锁关闭着时，它不允许内核控制路径继续进行。然而，当内核控制路径试图获取内核信号量所保护的忙资源时，相应的进程被挂起。只有在资源被释放时，进程才再次变为可运行的。因此，只有可以睡眠的函数才能获取内核信号量；中断处理程序和可延迟函数都不能使用内核信号量。

内核信号量是struct semaphore类型的对象，包含下面这些字段：

count

存放atomic_t类型的一个值。如果该值大于0，那么资源就是空闲的，也就是说，该资源现在可以使用。相反，如果count等于0，那么信号量是忙的，但没有进程等待这个被保护的资源。最后，如果count为负数，则资源是不可用的，并至少有一个进程等待资源。

wait

存放等待队列链表的地址，当前等待资源的所有睡眠进程都放在这个链表中。当然，如果count大于或等于0，等待队列就为空。

sleepers

存放一个标志，表示是否有一些进程在信号量上睡眠。我们很快就会看到这个字段的作用。

可以用init_MUTEX()和init_MUTEX_LOCKED()函数来初始化互斥访问所需的信号量：这两个宏分别把count字段设置成1（互斥访问的资源空闲）和0（对信号量进行初始化的进程当前互斥访问的资源忙）。宏DECLARE_MUTEX和DECLARE_MUTEX_LOCKED完成同样的功能，但它们也静态分配semaphore结构的变量。注意，也可以把信号量中的count初始化为任意的正数值n，在这种情况下，最多有n个进程可以并发地访问这个资源。

获取和释放信号量

让我们从如何释放一个信号量来开始讨论，这比获取一个信号量要简单得多。当进程希望释放内核信号量锁时，就调用up()函数。这个函数本质上等价于下列汇编语言片段：

```
    movl $sem->count,%ecx
    lock; incl (%ecx)
    jg 1f
    lea %ecx,%eax
    pushl %edx
    pushl %ecx
    call _ _up
    popl %ecx
    popl %edx
1:
```

这里__up()是下列C函数：

```
__attribute__((regparm(3))) void __up(struct semaphore *sem)
{
    wake_up(&sem->wait);
}
```

up()函数增加*sem信号量count字段的值，然后，检查它的值是否大于0。count的增加及其后jump指令所测试的标志的设置都必须原子地执行；否则，另一个内核控制路径有可能同时访问这个字段的值，这会导致灾难性的后果。如果count大于0，说明没有进程在等待队列上睡眠，因此，什么事也不做。否则，调用__up()函数以唤醒一个睡眠的进程。注意，__up()从eax寄存器接受参数（参见第三章“执行进程切换”一节中对函数__switch_to()的说明）。

相反，当进程希望获取内核信号量锁时，就调用down()函数。down()的实现相当棘手，但本质上等价于下列代码：

```
down:
    movl $sem->count,%ecx
    lock; decl (%ecx);
    jns 1f
    lea %ecx, %eax
    pushl %edx
    pushl %ecx
    call __down
    popl %ecx
    popl %edx
1:
```

这里__down()是下列C函数：

```
__attribute__((regparm(3))) void __down(struct semaphore * sem)
{
    DECLARE_WAITQUEUE(wait, current);
    unsigned long flags;
    current->state = TASK_UNINTERRUPTIBLE;
    spin_lock_irqsave(&sem->wait.lock, flags);
    add_wait_queue_exclusive_locked(&sem->wait, &wait);
    sem->sleepers++;
    for (;;) {
        if (!atomic_add_negative(sem->sleepers-1, &sem->count)) {
            sem->sleepers = 0;
            break;
        }
        sem->sleepers = 1;
        spin_unlock_irqrestore(&sem->wait.lock, flags);
        schedule();
        spin_lock_irqsave(&sem->wait.lock, flags);
        current->state = TASK_UNINTERRUPTIBLE;
    }
    remove_wait_queue_locked(&sem->wait, &wait);
    wake_up_locked(&sem->wait);
    spin_unlock_irqrestore(&sem->wait.lock, flags);
    current->state = TASK_RUNNING;
}
```

down()函数减少*sem信号量的count字段的值，然后检查该值是否为负。该值的减少和检查过程都必须是原子的。如果count大于或等于0，当前进程获得资源并继续正常执行。否则，count为负，当前进程必须挂起。把一些寄存器的内容保存在栈中，然后调用__down()。

从本质上说，__down()函数把当前进程的状态从TASK_RUNNING改变为TASK_UNINTERRUPTIBLE，并把进程放在信号量的等待队列。该函数在访问信号量结构的字段之前，要获得用来保护信号量等待队列的sem->wait.lock自旋锁（参见第三章“如何组织进程”），并禁止本地中断。通常当插入和删除元素时，等待队列函数根据需要获取和释放等待队列的自旋锁。函数__down()也用等待队列自旋锁来保护信号量数据结构的其他字段，以使在其他CPU上运行的进程不能读或修改这些字段。最后，__down()使用等待队列函数的“_locked”版本，它假设在调用等待队列函数之前已经获得了自旋锁。

__down()函数的主要任务是挂起当前进程，直到信号量被释放。然而，要实现这种想法是并不容易。为了容易地理解代码，要牢记如果没有进程在信号量等待队列上睡眠，则信号量的sleepers字段通常被置为0，否则被置为1。让我们通过考虑几种典型的情况来解释代码。

MUTEX 信号量打开(count 等于 1, sleepers 等于 0)

down宏仅仅把count字段置为0，并跳到主程序的下一条指令；因此，__down()函数根本就不执行。

MUTEX 信号量关闭，没有睡眠进程 (count 等于 0, sleepers 等于 0)

down宏减count并将count字段置为 -1且sleepers字段置为0来调用__down()函数。在循环体的每次循环中，该函数检查count字段是否为负。（因为当调用atomic_add_negative()函数时，sleepers等于0，因此atomic_add_negative()不改变count字段。）

- 如果count字段为负，__down()就调用schedule()挂起当前进程。count字段仍然置为-1，而sleepers字段置为1。随后，进程在这个循环内恢复自己的运行并又进行测试。
- 如果count字段不为负，则把sleepers置为0，并从循环退出。__down()试图唤醒信号量等待队列中的另一个进程（但在我们的情景中，队列现在为空），并终止保持的信号量。在退出时，count字段和sleepers字段都置为0，这表示信号量关闭且没有进程等待信号量。

MUTEX 信号量关闭，有其他睡眠进程 (count 等于 -1, sleepers 等于 1)

down宏减count并将count字段置为 -2且sleepers字段置为1来调用__down()函数。该函数暂时把sleepers置为2，然后通过把sleepers-1加到count来取

消由down宏执行的减操作。同时，该函数检查count是否依然为负(在__down()进入临界区之前，持有信号量的进程可能正好释放了信号量)。

- 如果count字段为负，__down()函数把sleepers重新设置为1，并调用schedule()挂起当前进程。count字段还是置为－1，而sleepers字段置为1。
- 如果count字段不为负，__down()函数把sleepers置为0，试图唤醒信号量等待队列上的另一个进程，并退出持有的信号量。在退出时，count字段置为0且sleepers字段置为0。这两个字段的值看起来错了，因为还有其他的进程在睡眠。然而，考虑一下在等待队列上的另一个进程已经被唤醒。这个进程进行循环体的另一个次循环；atomic_add_negative()函数从count中减去1，count重新变为-1；此外，唤醒的进程在重新回去睡眠之前，把sleepers重置为1。

可以很容易地验证，代码在所有的情况下都正确地工作。考虑一下，__down()中的wake_up()函数至多唤醒一个进程，因为等待队列中的睡眠进程是互斥的(参见第三章“如何组织进程”一节)。

只有异常处理程序，特别是系统调用服务例程，才可以调用down()函数。中断处理程序或可延迟的函数不必调用down()，因为当信号量忙时，这个函数挂起进程。由于这个原因，Linux提供了down_trylock()函数，前面提及的异步函数可以安全地使用down_trylock()函数。该函数和down()函数除了对资源繁忙情况的处理有所不同外，其他都是相同的。在资源繁忙时，该函数会立即返回，而不是让进程去睡眠。

系统中还定义了一个略有不同的函数，即down_interruptible()。该函数广泛地用在设备驱动程序中，因为如果进程接收了一个信号但在信号量上被阻塞，就允许进程放弃“down”操作。如果睡眠进程在获得需要的资源之前被一个信号唤醒，那么该函数就会增加信号量的count字段的值并返回-EINTR。另一方面，如果down_interruptible()正常结束并得到了需要的资源，就返回0。因此，在返回值是-EINTR时，设备驱动程序可以放弃I/O操作。

最后，因为进程通常发现信号量处于打开状态，因此，就可以优化信号量函数。尤其是，如果信号量等待队列为空，up()函数就不执行跳转指令；同样，如果信号量是打开的，down()函数就不执行跳转指令。信号量实现的复杂性是由于极力在执行流的主分支上避免费时的指令而造成的。

读/写信号量

读/写信号量类似于前面“读/写自旋锁”一节描述的读/写自旋锁，有一点不同：在信号量再次变为打开之前，等待进程挂起而不是自旋。

很多内核控制路径为读可以并发地获取读/写信号量。但是，任何写者内核控制路径必须有对被保护资源的互斥访问。因此，只有在没有内核控制路径为读访问或写访问持有信号量时，才可以为写获取信号量。读/写信号量可以提高内核中的并发度，并改善了整个系统的性能。

内核以严格的FIFO顺序处理等待读/写信号量的所有进程。如果读者或写者进程发现信号量关闭，这些进程就被插入到信号量等待队列链表的末尾。当信号量被释放时，就检查处于等待队列链表第一个位置的进程。第一个进程常被唤醒。如果是一个写者进程，等待队列上其他的进程就继续睡眠。如果是一个读者进程，那么紧跟第一个进程的其他所有读者进程也被唤醒并获得锁。不过，在写者进程之后排队的读者进程继续睡眠。

每个读/写信号量都是由rw_semaphore结构描述的，它包含下列字段：

count
: 存放两个16位的计数器。其中最高16位计数器以二进制补码形式存放非等待写者进程的总数（0或1）和等待的写内核控制路径数。最低16位计数器存放非等待的读者和写者进程的总数。

wait_list
: 指向等待进程的链表。链表中的每个元素都是一个rwsem_waiter结构，该结构包含一个指针和一个标志，指针指向睡眠进程的描述符，标志表示进程是为读需要信号量还是为写需要信号量。

wait_lock
: 一个自旋锁，用于保护等待队列链表和rw_semaphore结构本身。

init_rwsem()函数初始化rw_semaphore结构，即把count字段置为0，wait_lock自旋锁置为未锁，而把wait_list置为空链表。

down_read()和down_write()函数分别为读或写获取读/写信号量。同样，up_read()和up_write()函数为读或写释放以前获取的读/写信号量。down_read_trylock()和down_write_trylock()函数分别类似于down_read()和down_write()函数，但是，在信号量忙的情况下，它们不阻塞进程。最后，函数downgrade_write()自动把写锁转换成读锁。这5个函数的实现代码比较长，但因为它与普通信号量的实现类似，所以容易理解，我们就不再对它们进行说明。

补充原语

Linux 2.6还使用了另一种类似于信号量的原语：补充（*completion*）。引入这种原语是为了解决多处理器系统上发生的一种微妙的竞争条件，当进程A分配了一个临时信号量

变量，把它初始化为关闭的MUTEX，并把其地址传递给进程B，然后在A之上调用down()，进程A打算一但被唤醒就撤消该信号量。随后，运行在不同CPU上的进程B在同一信号量上调用up()。然而，up()和 down()的目前实现还允许这两个函数在同一个信号量上并发执行。因此，进程A可以被唤醒并撤销临时信号量，而进程B还在运行up()函数。结果，up()可能试图访问一个不存在的数据结构。

当然，也可以改变up()和down()的实现以禁止在同一信号量上并发执行。然而，这种改变可能需要另外的指令，这对于频繁使用的函数来说不是什么好事。

补充是专门设计来解决以上问题的同步原语。completion数据结构包含一个等待队列头和一个标志：

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

与up()对应的函数叫做complete()。complete()接收completion数据结构的地址作为参数，在补充等待队列的自旋锁上调用spin_lock_irqsave()，递增done字段，唤醒在wait等待队列上睡眠的互斥进程，最后调用spin_unlock_irqrestore()。

与down()对应的函数叫做wait_for_completion()。wait_for_completion()接收completion数据结构的地址作为参数，并检查done标志的值。如果该标志的值大于0，wait_for_completion()就终止，因为这说明complete()已经在另一个CPU上运行。否则，wait_for_completion()把current作为一个互斥进程加到等待队列的末尾，并把current置为TASK_UNINTERRUPTIBLE状态让其睡眠。一旦current被唤醒，该函数就把current从等待队列中删除，然后，函数检查done标志的值：如果等于0函数就结束，否则，再次挂起当前进程。与complete()函数中的情形一样，wait_for_completion()使用补充等待队列中的自旋锁。

补充原语和信号量之间的真正差别在于如何使用等待队列中包含的自旋锁。在补充原语中，自旋锁用来确保complete()和wait_for_completion()不会并发执行。在信号量中，自旋锁用于避免并发执行的down()函数弄乱信号量的数据结构。

禁止本地中断

确保一组内核语句被当作一个临界区处理的主要机制之一就是中断禁止。即使当硬件设备产生了一个IRQ信号时，中断禁止也让内核控制路径继续执行，因此，这就提供了一种有效的方式，确保中断处理程序访问的数据结构也受到保护。然而，禁止本地中断并不保护运行在另一个CPU上的中断处理程序对数据结构的并发访问，因此，在多处理器

系统上，禁止本地中断经常与自旋锁结合使用（参见后面“对内核数据结构的同步访问”一节）。

宏local_irq_disable()使用cli汇编语言指令关闭本地CPU上的中断，宏local_irq_enable()使用sti汇编语言指令打开被关闭的中断。正如在第四章“IRQ和中断”一节中说明的，汇编语言指令cli和sti分别清除和设置eflags控制寄存器的IF标志。如果eflags寄存器的IF标志被清0，宏irqs_disabled()产生等于1的值；如果IF标志被设置，该宏也产生为1的值。

当内核进入临界区时，通过把eflags寄存器的IF标志清0关闭中断。但是，内核经常不能在临界区的末尾简单地再次设置这个标志。中断可以以嵌套的方式执行，所以内核不必知道当前控制路径被执行之前IF标志的值究竟是什么。在这种情况下，控制路径必须保存先前赋给该标志的值，并在执行结束时恢复它。

保存和恢复eflags的内容是分别通过宏local_irq_save和local_irq_restore来实现的。local_irq_save宏把eflags寄存器的内容拷贝到一个局部变量中，随后用cli汇编语言指令把IF标志清0。在临界区的末尾，宏local_irq_restore恢复eflags原来的内容，因此，只是在这个控制路径发出cli汇编语言指令之前，中断被激活的情况下，中断才处于打开状态。

禁止和激活可延迟函数

在第四章的“软中断”一节，我们说明了可延迟函数可能在不可预知的时间执行（实际上是在硬件中断处理程序结束时）。因此，必须保护可延迟函数访问的数据结构使其避免竞争条件。

禁止可延迟函数在一个CPU上执行的一种简单方式就是禁止在那个CPU上的中断。因为没有中断处理程序被激活，因此，软中断操作就不能异步地开始。

然而，我们在下一节会看到，内核有时需要只禁止可延迟函数而不禁止中断。通过操纵当前thread_info描述符preempt_count字段中存放的软中断计数器，可以在本地CPU上激活或禁止可延迟函数。

回忆一下，如果软中断计数器是正数，do_softirq()函数就不会执行软中断，而且，因为tasklet在软中断之前被执行，把这个计数器设置为大于0的值，由此禁止了在给定CPU上的所有可延迟函数和软中断的执行。

宏local_bh_disable给本地CPU的软中断计数器加1，而函数local_bh_enable()从本地CPU的软中断计数器中减掉1。内核因此能使用几个嵌套的local_bh_disable调

用，只有宏local_bh_enable与第一个local_bh_disable调用相匹配，可延迟函数才再次被激活。

递减软中断计数器之后，local_bh_enable()执行两个重要的操作以有助于保证适时地执行长时间等待的线程：

1. 检查本地CPU的preempt_count字段中硬中断计数器和软中断计数器，如果这两个计数器的值都等于0而且有挂起的软中断要执行，就调用do_softirq()来激活这些软中断（见第四章“软中断”一节）。
2. 检查本地CPU的TIF_NEED_RESCHED标志是否被设置，如果是，说明进程切换请求是挂起的，因此调用preempt_schedule()函数（参见本章前面的“内核抢占”一节）。

对内核数据结构的同步访问

可以使用前面所述的同步原语保护共享数据结构避免竞争条件。当然，系统性能可能随所选择同步原语种类的不同而有很大变化。通常情况下，内核开发者采用下述由经验得到的法则：*把系统中的并发度保持在尽可能高的程度。*

系统中的并发度又取决于两个主要因素：

- 同时运转的I/O设备数
- 进行有效工作的CPU数

为了使I/O吞吐量最大化，应该使中断禁止保持在很短的时间。正如第四章的“IRQ和中断”一节描述的那样，当中断被禁止时，由I/O设备产生的IRQ被PIC暂时忽略，因此，就没有新的活动在这种设备上开始。

为了有效地利用CPU，应该尽可能避免使用基于自旋锁的同步原语。当一个CPU执行紧指令循环等待自旋锁打开时，是在浪费宝贵的机器周期。就像我们前面所描述的，更糟糕的是：由于自旋锁对硬件高速缓存的影响而使其对系统的整体性能产生不利影响。

让我们举例说明在下列两种情况下，既可以维持较高的并发度，也可以达到同步。

- 共享的数据结构是一个单独的整数值，可以把它声明为atomic_t类型并使用原子操作对其更新。原子操作比自旋锁和中断禁止都快，只有在几个内核控制路径同时访问这个数据结构时速度才会慢下来。
- 把一个元素插入到共享链表的操作决不是原子的，因为这至少涉及两个指针赋值。

不过，内核有时并不用锁或禁止中断就可以执行这种插入操作。我们把这种操作的工作机制作为例子来进行说明。考虑一种情况，系统调用服务例程（参见第十章的“系统调用处理程序及服务例程）把新元素插入到一个简单链表中，而中断处理程序或可延迟函数异步地查看该链表。

在C语言中，插入是通过下列指针赋值实现的：

```
new->next = list_element->next;
list_element->next = new;
```

在汇编语言中，插入简化为两个连续的原子指令。第一条指令建立new元素的next指针，但不修改链表。因此，如果中断处理程序在第一条指令和第二条指令执行的中间查看这个链表，看到的就是没有新元素的链表。如果该处理程序在第二条指令执行后查看链表，就会看到有新元素的链表。关键是，在任一种情况下，链表都是一致的且处于未损坏状态。然而，只有在中断处理程序不修改链表的情况下才能确保这种完整性。如果修改了链表，那么在new元素内刚刚设置的next指针就可能变为无效的。

然而，开发者必须确保两个赋值操作的顺序不被编译器或CPU控制单元搅乱；否则，如果中断处理程序在两个赋值之间中断了系统调用服务例程，处理程序就会看到一个损坏的链表。因此，就需要一个写内存屏障原语：

```
new->next = list_element->next;
wmb();
list_element->next = new;
```

在自旋锁、信号量及中断禁止之间选择

遗憾的是，对大多数内核数据结构的访问模式非常复杂，远不像上例所示的那样简单，于是，迫使内核开发者使用信号量、自旋锁、中断禁止和软中断禁止。一般来说，同步原语的选取取决于访问数据结构的内核控制路径的种类，如表5-8所示。记住，只要内核控制路径获得自旋锁（还有读/写锁、顺序锁或RCU“读锁”），就禁用本地中断或本地软中断，自动禁用内核抢占。

表5-8：内核控制路径访问的数据结构所需要的保护

访问数据结构的内核控制路径	单处理器保护	多处理器进一步保护
异常	信号量	无
中断	本地中断禁止	自旋锁
可延迟函数	无	无或自旋锁（参看表5-9）
异常与中断	本地中断禁止	自旋锁

表5-8：内核控制路径访问的数据结构所需要的保护（续）

访问数据结构的内核控制路径	单处理器保护	多处理器进一步保护
异常与可延迟函数	本地软中断禁止	自旋锁
中断与可延迟函数	本地中断禁止	自旋锁
异常、中断与可延迟函数	本地中断禁止	自旋锁

保护异常所访问的数据结构

当一个数据结构仅由异常处理程序访问时，竞争条件通常是易于理解也易于避免的。最常见的产生同步问题的异常就是系统调用服务例程（参看第十章的“系统调用处理程序及服务例程”一节），在这种情况下，CPU运行在内核态而为用户态程序提供服务。因此，仅由异常访问的数据结构通常表示一种资源，可以分配给一个或多个进程。

竞争条件可以通过信号量避免，因为信号量原语允许进程睡眠到资源变为可用。注意，信号量工作方式在单处理器系统和多处理器系统上完全相同。

内核抢占不会引起太大的问题。如果一个拥有信号量的进程是可以被抢占的，运行在同一个CPU上的新进程就可能试图获得这个信号量。在这种情况下，让新进程处于睡眠状态，而且原来拥有信号量的进程最终会释放信号量。只有在访问每CPU变量的情况下，必须显式地禁用内核抢占，就像在本章前面“每CPU变量”一节中所描述的那样。

保护中断所访问的数据结构

假定一个数据结构仅被中断处理程序的“上半部分”访问。我们在第四章的“中断处理”一节了解到每个中断处理程序都相对自己串行地执行——也就是说，中断处理程序本身不能同时多次运行。因此，访问数据结构就无需任何同步原语。

但是，如果多个中断处理程序访问一个数据结构，情况就有所不同了。一个处理程序可以中断另一个处理程序，不同的中断处理程序可以在多处理器系统上同时运行。没有同步，共享的数据结构就很容易被破坏。

在单处理器系统上，必须通过在中断处理程序的所有临界区上禁止中断来避免竞争条件。只能用这种方式进行同步，因为其他的同步原语都不能完成这件事。信号量能够阻塞进程，因此，不能用在中断处理程序上。另一个方面，自旋锁可能使系统冻结：如果访问数据结构的处理程序被中断，它就不能释放锁；因此，新的中断处理程序在自旋锁的紧循环上保持等待。

同样，多处理器系统的要求甚至更加苛刻。不能简单地通过禁止本地中断来避免竞争条

不过，内核有时并不用锁或禁止中断就可以执行这种插入操作。我们把这种操作的工作机制作为例子来进行说明。考虑一种情况，系统调用服务例程（参见第十章的“系统调用处理程序及服务例程）把新元素插入到一个简单链表中，而中断处理程序或可延迟函数异步地查看该链表。

在C语言中，插入是通过下列指针赋值实现的：

```
new->next = list_element->next;
list_element->next = new;
```

在汇编语言中，插入简化为两个连续的原子指令。第一条指令建立new元素的next指针，但不修改链表。因此，如果中断处理程序在第一条指令和第二条指令执行的中间查看这个链表，看到的就是没有新元素的链表。如果该处理程序在第二条指令执行后查看链表，就会看到有新元素的链表。关键是，在任一种情况下，链表都是一致的且处于未损坏状态。然而，只有在中断处理程序不修改链表的情况下才能确保这种完整性。如果修改了链表，那么在new元素内刚刚设置的next指针就可能变为无效的。

然而，开发者必须确保两个赋值操作的顺序不被编译器或CPU控制单元搅乱；否则，如果中断处理程序在两个赋值之间中断了系统调用服务例程，处理程序就会看到一个损坏的链表。因此，就需要一个写内存屏障原语：

```
new->next = list_element->next;
wmb();
list_element->next = new;
```

在自旋锁、信号量及中断禁止之间选择

遗憾的是，对大多数内核数据结构的访问模式非常复杂，远不像上例所示的那样简单，于是，迫使内核开发者使用信号量、自旋锁、中断禁止和软中断禁止。一般来说，同步原语的选取取决于访问数据结构的内核控制路径的种类，如表5-8所示。记住，只要内核控制路径获得自旋锁（还有读/写锁、顺序锁或RCU“读锁”），就禁用本地中断或本地软中断，自动禁用内核抢占。

表5-8：内核控制路径访问的数据结构所需要的保护

访问数据结构的内核控制路径	单处理器保护	多处理器进一步保护
异常	信号量	无
中断	本地中断禁止	自旋锁
可延迟函数	无	无或自旋锁（参看表5-9）
异常与中断	本地中断禁止	自旋锁

表5-8：内核控制路径访问的数据结构所需要的保护（续）

访问数据结构的内核控制路径	单处理器保护	多处理器进一步保护
异常与可延迟函数	本地软中断禁止	自旋锁
中断与可延迟函数	本地中断禁止	自旋锁
异常、中断与可延迟函数	本地中断禁止	自旋锁

保护异常所访问的数据结构

当一个数据结构仅由异常处理程序访问时，竞争条件通常是易于理解也易于避免的。最常见的产生同步问题的异常就是系统调用服务例程（参看第十章的“系统调用处理程序及服务例程”一节），在这种情况下，CPU运行在内核态而为用户态程序提供服务。因此，仅由异常访问的数据结构通常表示一种资源，可以分配给一个或多个进程。

竞争条件可以通过信号量避免，因为信号量原语允许进程睡眠到资源变为可用。注意，信号量工作方式在单处理器系统和多处理器系统上完全相同。

内核抢占不会引起太大的问题。如果一个拥有信号量的进程是可以被抢占的，运行在同一个CPU上的新进程就可能试图获得这个信号量。在这种情况下，让新进程处于睡眠状态，而且原来拥有信号量的进程最终会释放信号量。只有在访问每CPU变量的情况下，必须显式地禁用内核抢占，就像在本章前面“每CPU变量”一节中所描述的那样。

保护中断所访问的数据结构

假定一个数据结构仅被中断处理程序的“上半部分”访问。我们在第四章的“中断处理”一节了解到每个中断处理程序都相对自己串行地执行——也就是说，中断处理程序本身不能同时多次运行。因此，访问数据结构就无需任何同步原语。

但是，如果多个中断处理程序访问一个数据结构，情况就有所不同了。一个处理程序可以中断另一个处理程序，不同的中断处理程序可以在多处理器系统上同时运行。没有同步，共享的数据结构就很容易被破坏。

在单处理器系统上，必须通过在中断处理程序的所有临界区上禁止中断来避免竞争条件。只能用这种方式进行同步，因为其他的同步原语都不能完成这件事。信号量能够阻塞进程，因此，不能用在中断处理程序上。另一个方面，自旋锁可能使系统冻结：如果访问数据结构的处理程序被中断，它就不能释放锁；因此，新的中断处理程序在自旋锁的紧循环上保持等待。

同样，多处理器系统的要求甚至更加苛刻。不能简单地通过禁止本地中断来避免竞争条

件。事实上，即使在一个CPU上禁止了中断，中断处理程序还可以在其他CPU上执行。避免竞争条件最简单的方法是禁止本地中断(以便运行在同一个CPU上的其他中断处理程序不会造成干扰)，并获取保护数据结构的自旋锁或读/写自旋锁。注意，这些附加的自旋锁不能冻结系统，因为即使中断处理程序发现锁被关闭，在另一个CPU上拥有锁的中断处理程序最终也会释放这个锁。

Linux内核使用了几个宏，把本地中断激活/禁止与自旋锁结合起来。表5-9描述了其中的所有宏。在单处理器系统上，这些宏仅激活或禁止本地中断和内核抢占。

表5-9：与中断相关的自旋锁宏

宏	说明
spin_lock_irq(l)	local_irq_disable();spin_lock(l)
spin_unlock_irq(l)	spin_unlock(l);local_irq_enable()
spin_lock_bh(l)	local_bh_disable();spin_lock(l)
spin_unlock_bh(l)	spin_unlock(l);local_bh_enable()
spin_lock_irqsave(l,f)	local_irq_save(f);spin_lock(l)
spin_unlock_irqrestore(l,f)	spin_unlock(l);local_irq_restore(f)
read_lock_irq(l)	local_irq_disable();read_lock(l)
read_unlock_irq(l)	read_unlock(l);local_irq_enable()
read_lock_bh(l)	local_bh_disable();read_lock(l)
read_unlock_bh(l)	read_unlock(l);local_bh_enable()
write_lock_irq(l)	local_irq_disable();write_lock(l)
write_unlock_irq(l)	write_unlock(l);local_irq_enable()
write_lock_bh(l)	local_bh_disable();write_lock(l)
write_unlock_bh(l)	write_unlock(l);local_bh_enable()
read_lock_irqsave(l,f)	local_irq_save(f);read_lock(l)
read_unlock_irqrestore(l,f)	read_unlock(l);local_irq_restore(f)
write_lock_irqsave(l,f)	local_irq_save(f);write_lock(l)
write_unlock_irqrestore(l,f)	write_unlock(l);local_irq_restore(f)
read_seqbegin_irqsave(l,f)	local_irq_save(f);read_seqbegin(l)
read_seqretry_irqrestore(l,v,f)	read_seqretry(l,v);local_irq_restore(f)
write_seqlock_irqsave(l,f)	local_irq_save(f);write_seqlock(l)
write_sequnlock_irqrestore(l,f)	write_sequnlock(l);local_irq_restore(f)
write_seqlock_irq(l)	local_irq_disable();write_seqlock(l)
write_sequnlock_irq(l)	write_sequnlock(l);local_irq_enable()
write_seqlock_bh(l)	local_bh_disable();write_seqlock(l)
write_sequnlock_bh(l)	write_sequnlock(l);local_bh_enable()

保护可延迟函数所访问的数据结构

只被可延迟函数访问的数据结构需要哪种保护呢？这主要取决于可延迟函数的种类。在第四章“软中断及tasklet”一节，我们说明了软中断和ttasklet本质上有不同的并发度。

首先，在单处理器系统上不存在竞争条件。这是因为可延迟函数的执行总是在一个CPU上串行进行——也就是说，一个可延迟函数不会被另一个可延迟函数中断。因此，根本不需要同步原语。

相反，在多处理器系统上，竞争条件的确存在，因为几个可延迟函数可以并发运行。表5-10列出了所有可能的情况。

表5-10：在SMP上可延迟函数访问的数据结构所需的保护

访问数据结构的可延迟函数	保护
软中断	自旋锁
一个tasklet	无
多个tasklet	自旋锁

由软中断访问的数据结构必须受到保护，通常使用自旋锁进行保护，因为同一个软中断可以在两个或多个CPU上并发运行。相反，仅由一种tasklet访问的数据结构不需要保护，因为同种tasklet不能并发运行。但是，如果数据结构被几种tasklet访问，那么，就必须对数据结构进行保护。

保护由异常和中断访问的数据结构

让我们现在考虑一下由异常处理程序（例如系统调用服务例程）和中断处理程序访问的数据结构。

在单处理器系统上，竞争条件的防止是相当简单的，因为中断处理程序不是可重入的且不能被异常中断。只要内核以本地中断禁止访问数据结构，内核在访问数据结构的过程中就不会被中断。不过，如果数据结构正好是被一种中断处理程序访问，那么，中断处理程序不用禁止本地中断就可以自由地访问数据结构。

在多处理器系统上，我们必须关注异常和中断在其他CPU上的并发执行。本地中断禁止还必须外加自旋锁，强制并发的内核控制路径进行等待，直到访问数据结构的处理程序完成自己的工作。

有时，用信号量代替自旋锁可能更好。因为中断处理程序不能被挂起，它们必须用紧循环和down_trylock()函数获得信号量；对这些中断处理程序来说，信号量起的作用本

质上与自旋锁一样。另一方面，系统调用服务例程可以在信号量忙时挂起调用进程。对大部分系统调用而言，这是所期望的行为。在这种情况下，信号量比自旋锁更好，因为信号量使系统具有更高的并发度。

保护由异常和可延迟函数访问的数据结构

异常和可延迟函数都访问的数据结构与异常和中断处理程序访问的数据结构处理方式类似。事实上，可延迟函数本质上是由中断的出现激活的，而可延迟函数执行时不可能产生异常。因此，把本地中断禁止与自旋锁结合起来就足够了。

实际上，这更加充分：异常处理程序可以通过使用local_bh_disable()宏简单地禁止可延迟函数，而不禁止本地中断（参看第四章的“软中断”一节）。仅禁止可延迟函数比禁止中断更可取，因为中断还可以继续在CPU上得到服务。在每个CPU上可延迟函数的执行都被串行化，因此，不存在竞争条件。

同样，在多处理器系统上，要用自旋锁确保任何时候只有一个内核控制路径访问数据结构。

保护由中断和可延迟函数访问的数据结构

这种情况类似于中断和异常处理程序访问的数据结构。当可延迟函数运行时可能产生中断，但是，可延迟函数不能阻止中断处理程序。因此，必须通过在可延迟函数执行期间禁用本地中断来避免竞争条件。不过，中断处理程序可以随意访问被可延迟函数访问的数据结构而不用关中断，前提是没有其他的中断处理程序访问这个数据结构。

在多处理器系统上，还是需要自旋锁禁止对多个CPU上数据结构的并发访问。

保护由异常、中断和可延迟函数访问的数据结构

类似于前面的情况，禁止本地中断和获取自旋锁几乎总是避免竞争条件所必需的。注意，没有必要显式地禁止可延迟函数，因为当中断处理程序终止执行时，可延迟函数才能被实质激活；因此，禁止本地中断就足够了。

避免竞争条件的实例

人们总是期望内核开发者确定和解决由内核控制路径的交错执行所引起的同步问题。但是，避免竞争条件是一项艰巨的任务，因为这需要对内核的各个成分如何相互作用有一个清楚的理解。为了直观地认识内核内部到底是什么样子，需要提及本章所定义同步原语的几种典型用法。

引用计数器

引用计数器广泛地用在内核中以避免由于资源的并发分配和释放而产生的竞争条件。引用计数器（*reference counter*）只不过是一个atomic_t计数器，与特定的资源，如内存页、模块或文件相关。当内核控制路径开始使用资源时就原子地减少计数器的值，当内核控制路径使用完资源时就原子地增加计数器。当引用计数器变为0时，说明该资源未被使用，如果必要，就释放该资源。

大内核锁

在早期的Linux内核版本中，大内核锁（*big kernel block*，也叫全局内核锁或BKL）被广泛使用。在2.0版本中，这个锁是相对粗粒度的自旋锁，确保每次只有一个进程能运行在内核态。2.2和2.4内核具有极大的灵活性，不再依赖一个单独的自旋锁，而是由许多不同的自旋锁保护大量的内核数据结构。在Linux 2.6版本的内核中，用大内核锁来保护旧的代码（绝大多数是与VFS和几个文件系统相关的函数）。

从内核版本2.6.11开始，用一个叫做kernel_sem的信号量来实现大内核锁（在较早的2.6版本中，大内核锁是通过自旋锁来实现的）。但是，大内核锁比简单的信号量要复杂一些。

每个进程描述符都含有lock_depth字段，这个字段允许同一个进程几次获取大内核锁。因此，对大内核锁两次连续的请求不挂起处理器（相对于普通自旋锁）。如果进程未获得过锁，则这个字段的值为-1；否则，这个字段的值加1，表示已经请求了多少次锁。lock_depth字段对中断处理程序、异常处理程序及可延迟函数获取大内核锁都是至关重要的。如果没有这个字段，那么，在当前进程已经拥有大内核锁的情况下，任何试图获得这个锁的异步函数都可能产生死锁。

lock_kernel()和unlock_kernel()内核函数用来获得和释放大内核锁。前一个函数等价于：

```
depth = current->lock_depth + 1;
if (depth == 0)
    down(&kernel_sem);
current->lock_depth = depth;
```

而后者等价于：

```
if (--current->lock_depth < 0)
    up(&kernel_sem);
```

注意，lock_kernel()和unlock_kernel()函数的if语句不需要原子地执行，因为lock_depth不是全局变量——这是每个CPU在自己当前进程描述符中访问的一个字段。在

if语句内的本地中断也不会引起竞争条件。即使新内核控制路径调用了lock_kernel()，它在终止前也必须释放大内核锁。

足以令人吃惊的是，允许一个持有大内核锁的进程调用schedule()，从而放弃CPU！不过，schedule()函数检查被替换进程的lock_depth字段，如果它的值是0或者正数，就自动释放kernel_sem信号量（参见第七章“schedule()函数”一节）。因此，不会有显式调用schedule()的进程在进程切换前后都保持大内核锁。但是，当schedule()函数再次选择这个进程来执行的时候，将为该进程重新获得大内核锁。

然而，如果一个持有大内核锁的进程被另一个进程抢占，情况就有所不同了。一直到内核版本2.6.10还没有出现这种情况，因为获取自旋锁时会自动禁用内核抢占。但是，现在大内核锁的实现是基于信号量的，而且不会由于获得它而自动禁用内核抢占。实际上，在被大内核锁保护的临界区内允许内核抢占是改变大内核锁实现的主要原因。其次，这对于系统的响应时间会产生有益的影响。

当一个持有大内核锁的进程被抢占时，schedule()一定不能释放信号量，因为在临界区内执行代码的进程没有主动触发进程切换。所以，如果释放大内核锁，那么另外一个进程就可能获得它，并破坏由被抢占的进程所访问的数据结构。

为了避免被抢占的进程失去大内核锁，preempt_schedule_irq()临时把进程的lock_depth字段设置为-1（参见第四章“从中断和异常返回”）。观察这个字段的值，schedule()假定被替换的进程不拥有kernel_sem信号量，也就不释放它。结果，被抢占的进程就一直拥有kernel_sem信号量。一旦这个进程再次被调度程序选中，preempt_schedule_irq()函数就恢复lock_depth字段原来的值，并让进程在被大内核锁保护的临界区中继续执行。

内存描述符读/写信号量

mm_struct类型的每个内存描述符在mmap_sem字段中都包含了自己的信号量（参见第九章的“内存描述符”一节）。由于几个轻量级进程之间可以共享一个内存描述符，因此，信号量保护这个描述符以避免可能产生的竞争条件。

例如，让我们假设内核必须为某个进程创建或扩展一个内存区。为了做到这一点，内核调用do_mmap()函数分配一个新的vm_area_struct数据结构。在分配的过程中，如果没有可用的空闲内存，而共享同一内存描述符的另外一个进程可能在运行，那么当前进程可能被挂起。如果没有信号量，那么需要访问内存描述符的第二个进程的任何操作（例如，由于写时复制而产生的缺页）都可能会导致严重的数据崩溃。

这种信号量是作为读/写信号量来实现的，因为一些内核函数，如缺页异常处理程序（参见第九章的“缺页异常处理程序”一节）只需要扫描内存描述符。

slab 高速缓存链表的信号量

slab高速缓存描述符链表(参见第八章的"高速缓存描述符"一节)是通过cache_chain_sem信号量保护的，这个信号量允许互斥地访问和修改该链表。

当 kmem_cache_create()在链表中增加一个新元素，而 kmem_cache_shrink()和 kmem_cache_reap()顺序地扫描整个链表时，可能产生竞争条件。然而，在处理中断时，这些函数从不被调用，在访问链表时它们也从不阻塞。由于内核是支持抢占的，因此这种信号量在多处理器系统和单处理器系统中都会起作用。

索引节点的信号量

正如我们将在第十二章的"索引节点对象"一节中看到的，Linux 把磁盘文件的信息存放在一种叫做索引节点（inode）的内存对象中。相应的数据结构也包括有自己的信号量，存放在 i_sem 字段中。

在文件系统的处理过程中会出现很多竞争条件。实际上，磁盘上的每个文件都是所有用户共有的一种资源，因为所有进程都（可能）会存取文件的内容、修改文件名或文件位置、删除或复制文件等等。例如，让我们假设一个进程在显示某个目录所包含的文件。由于每个磁盘操作都可能会阻塞，因此即使在单处理器系统中，当第一个进程正在执行显示操作的过程中，其他进程也可能存取同一目录并修改它的内容。或者，两个不同的进程可能同时修改同一目录。所有这些竞争条件都可以通过用索引节点信号量保护目录文件来避免。

只要一个程序使用了两个或多个信号量，就存在死锁的可能，因为两个不同的控制路径可能互相死等着释放信号量。一般来说，Linux在信号量请求上很少会发生死锁问题，因为每个内核控制路径通常一次只需要获得一个信号量。然而，在有些情况下，内核必须获得两个或更多的信号量锁。索引节点信号量倾向于这种情况，例如，在 rename()系统调用的服务例程中就会发生这种情况。在这种情况下，操作涉及两个不同的索引节点，因此，必须采用两个信号量。为了避免这样的死锁，信号量的请求按预先确定的地址顺序进行。

第六章



定时测量

很多计算机化的活动都是由定时测量（timing measurement）来驱动的，这常常对用户是不可见的。例如，当你停止使用计算机的控制台以后，屏幕会自动关闭，这得归因于定时器，它允许内核跟踪你按键或移动鼠标后到现在过了多少时间。如果你收到了一个来自系统的警告信息，希望你删除一组不用的文件，这就是由于有一个程序能识别长时间未被访问的所有用户文件。为了进行这些操作，程序必须能从每个文件中检索到文件的最后访问时间，即时间戳(timestamp)，因此，这样的时间标记必须由内核自动地设置。更重要的是，定时机制连同一些更可见的内核活动（如检查超时）来驱使进程切换。

Linux 内核必需完成两种主要的定时测量，我们可以对此加以区别：

- 保存当前的时间和日期，以便能通过 `time()`、`ftime()`和`gettimeofday()`系统调用把它们返回给用户程序（见本章后面的“time()和 gettimeofday()系统调用”一节），也可以由内核本身把当前时间作为文件和网络包的时间戳。
- 维持定时器，这种机制能够告诉内核（参见后面的“软定时器和延迟函数”一节）或用户程序（分别参见后面的“setitimer()和alarm()系统调用”一节和“与POSIX定时器相关的系统调用”一节）某一时间间隔已经过去了。

定时测量是由基于固定频率振荡器和计数器的几个硬件电路完成的。本章由四个不同的部分组成。前两节描述建立定时机制的硬件设备，并给出Linux计时体系结构的总体概貌；接下来描述内核中与时间相关的主要任务：实现CPU分时、更新系统时间和资源使用统计数及维护软定时器。最后一节讨论与定时测量相关的系统调用及相应的服务例程。

时钟和定时器电路

在80x86体系结构上，内核必须显式地与几种时钟和定时器电路打交道。时钟电路同时用于跟踪当前时间和产生精确的时间度量。定时器电路由内核编程，所以它们以固定的、预先定义的频率发出中断。这样的周期性中断对于实现内核和用户程序使用的软定时器都是至关重要的。我们现在将简要描述IBM兼容PC上的时钟和硬件电路。

实时时钟（RTC）

所有的PC都包含一个叫实时时钟（*Renl Time Clock RTC*）的时钟，它是独立于CPU和所有其他芯片的。

即使当PC被切断电源，RTC还继续工作，因为它靠一个小电池或蓄电池供电。CMOS RAM和RTC被集成在一个芯片（Motorola 146818或其他等价的芯片）上。

RTC能在IRQ8上发出周期性的中断，频率在2～8192 Hz之间。也可以对RTC进行编程以使当RTC到达某个特定的值时激活IRQ8线，也就是作为一个闹钟来工作。

Linux只用RTC来获取时间和日期，不过，通过对*/dev/rtc*设备文件进行操作，也允许进程对RTC编程（参见第十三章）。内核通过0x70和0x71 I/O端口访问RTC。系统管理员通过执行Unix系统时钟程序（直接作用于这两个I/O端口）可以设置时钟。

时间戳计数器（TSC）

所有的80x86微处理器都包含一条CLK输入引线，它接收外部振荡器的时钟信号。从Pentium开始，80x86微处理器就都包含一个计数器，它在每个时钟信号到来时加1。该计数器是利用64位的时间戳计数器（*Time Stamp Counter TSC*）寄存器来实现的，可以通过汇编语言指令rdtsc读这个寄存器。当使用这个寄存器时，内核必须考虑到时钟信号的频率：例如，如果时钟节拍的频率是1 GHz，那么，时间戳计数器每纳秒增加一次。

与可编程间隔定时器传递来的时间测量相比，Linux利用这个寄存器可获得更精确的时间测量。为了做到这点，Linux在初始化系统的时候必须确定时钟信号的频率。事实上，因为编译内核时并不声明这个频率，所以同一内核映像可以运行在产生任何时钟频率的CPU上。

算出CPU实际频率的任务是在系统初始化期间完成的。calibrate_tsc()函数通过计

算一个大约在5ms的时间间隔内所产生的时钟信号的个数来算出CPU实际频率。通过适当地设置可编程间隔定时器的一个通道来产生这个时间常量（参见下一节）（注1）。

可编程间隔定时器（PIT）

除了实时时钟和时间戳计数器，IBM兼容PC还包含了第三种类型的时间测量设备，叫做可编程间隔定时器（*Programmable Interval Timer PIT*）。PIT的作用类似于微波炉的闹钟，即让用户意识到烹调的时间间隔已经过了。所不同的是，这个设备不是通过振铃，而是发出一个特殊的中断，叫做时钟中断（*timer interrupt*）来通知内核又一个时间间隔过去了（注2）。与闹钟的另一个区别是，PIT永远以内核确定的固定频率不停地发出中断。每个IBM兼容PC都至少包含一个PIT，PIT通常是使用0x40～0x43 I/O端口的一个8254 CMOS芯片。

在下一节中我们将看到，Linux给PC的第一个PIT进行编程，使它以（大约）1000 Hz的频率向IRQ0发出时钟中断，即每1ms产生一次时钟中断。这个时间间隔叫做一个节拍（*tick*），它的长度以纳秒为单位存放在tick_nsec变量中。在PC上，tick_nsec被初始化为999848ns（产生的时钟信号频率大约为1000.15 Hz），但是如果计算机被外部时钟同步的话，它的值可能被内核自动调整（参见后面的“adjtimex()系统调用”一节）。节拍为系统中的所有活动打拍子，从某种意义上说，它们像音乐家排练节目时节拍器发出的节拍声。

一般而言，短的节拍产生较高分辨度的定时器，当这种定时器执行同步I/O多路复用（poll()和select()系统调用）时，有助于多媒体的平滑播放和较快的响应时间。不过，这是一种折中：短的节拍需要CPU在内核态花费较多的时间，也就是在用户态花费较少的时间。因而，用户程序运行得稍慢一些。

时钟中断的频率取决于硬件体系结构。较慢的机器，其节拍大约为10ms（每秒产生100次时钟中断），而较快的机器的节拍为大约1ms（每秒产生1000或1024次时钟中断）。

在Linux的代码中，有几个宏产生决定时钟中断频率的常量，对此讨论如下：

- HZ产生每秒时钟中断的近似个数，也就是时钟中断的频率。在IBM PC上，这个值设置为1000。

注1：为了避免在整数除法中丢失有意义的位数，calibrate_tsc()的返回值为时钟节拍乘以2^{32}（以μs为单位）。

注2：也用PIT来驱动连接到计算机内部扬声器的音频放大器。

- CLOCK_TICK_RATE产生的值为1 193 182，这个值是8254芯片的内部振荡器频率。
- LATCH产生CLOCK_TICK_RATE和HZ的比值再四舍五入后的整数值。这个值用来对PIT编程。

PIT由setup_pit_timer()进行如下的初始化：

```
spin_lock_irqsave(&i8253_lock, flags);
outb_p(0x34,0x43);
udelay(10);
outb_p(LATCH & 0xff, 0x40);
udelay(10);
outb(LATCH >> 8, 0x40);
spin_unlock_irqrestore(&i8253_lock, flags);
```

outb() C函数等价于outb汇编语言指令：它把第一个操作数拷贝到由第二个操作数指定的I/O端口。outb_p()函数类似于outb()，不过，它会通过一个空操作而产生一个暂停，以避免硬件难以分辨。udelay()宏函数引入了一个更短的延迟（参见后面的“延迟函数”一节）。第一条outb_ p()语句让PIT以新的频率产生中断。接下来的两条outb_ p()和outb()语句为设备提供新的中断频率。把16位LATCH常量作为两个连续的字节发送到设备的8位I/O端口0x40。结果，PIT将以（大约）1000Hz的频率产生时钟中断，也就是说，每1 ms产生一次时钟中断。

CPU本地定时器

在最近80x86微处理器的本地APIC中（参看第四章“中断和异常”一节）还提供了另一种定时测量设备：CPU本地定时器。

CPU本地定时器是一种能够产生单步中断或周期性中断的设备，它类似于方才描述的可编程间隔定时器，不过，还是有几点区别：

- APIC计数器是32位，而PIC计数器是16位；因此，可以对本地定时器编程来产生很低频率的中断（计数器存放中断发生前必须经过的节拍数）。
- 本地APIC定时器把中断只发送给自己的处理器，而PIT产生一个全局性中断，系统中的任一CPU都可以对其处理。
- APIC定时器是基于总线时钟信号的（或在更老式的机器上是基于APIC时钟信号的）。每隔1, 2, 4, 8, 16, 32, 64或128总线时钟信号到来时对该定时器进行递减可以实现对其编程的目的。相反，PIT有其自己的内部时钟振荡器，可以更灵活地编程。

高精度事件定时器（HPET）

高精度事件定时器是由Intel和Microsoft联合开发的一种新型定时器芯片。尽管这种定时器在终端用户机器上还并不普遍，但Linux 2.6已经能够支持它们，所以我们将花一些篇幅来描述它们的特性。

HPET提供了许多可以被内核使用的硬定时器。这种新定时器芯片主要包含8个32位或64位的独立计数器。每个计数器由它自己的时钟信号所驱动，该时钟信号的频率必须至少为10MHz。因此，计数器最少可以每100ns增长一次。任何计数器最多可以与32个定时器相关联，每个定时器由一个比较器和一个匹配寄存器组成。比较器是一组用于检测计数器中的值与匹配寄存器中的值是否匹配的电路，如果找到一组匹配值就产生一个硬件中断。一些定时器可以被激活来产生周期性中断。

可以通过映射到内存空间的寄存器来对HPET芯片编程（与I/O APIC非常相似）。BIOS在自举阶段建立起映射并向操作系统内核报告它的起始内存地址。HPET寄存器允许内核对计数器和匹配寄存器的值进行读和写，允许内核对单步中断进行编程，还允许内核在支持HPET的定时器上激活或禁止周期性中断。

下一代主板将很可能同时包含HPET和8254 PIT。但是在不久的将来，期望HPET将完全取代PIT。

ACPI电源管理定时器

ACPI电源管理定时器（或称作*ACPI PMT*）是另一种时钟设备，包含在几乎所有基于ACPI的主板上。它的时钟信号拥有大约为3.58 MHz的固定频率。该设备实际上是一个简单的计数器，它在每个时钟节拍到来时增加一次。为了读取计数器的当前值，内核需要访问某个I/O端口，该I/O端口的地址由BIOS在初始化阶段确定（参见附录一）。

如果操作系统或者BIOS可以通过动态降低CPU的工作频率或者工作电压来节省电池的电能，那么ACPI电源管理定时器就比TSC更优越。当发生这种情况时，TSC的频率发生改变（这样将造成时间偏差和其他一些不良的效果），而ACPI PMT的频率不会改变。而另一方面，TSC计数器的高频率非常便于测量特别小的时间间隔。

不过，如果系统中存在HPET设备，那么比起其他电路而言它总是成为首选，因为它更为复杂的结构使得功能更强。在本章稍后的表6-2中举例说明了Linux如何使用可利用的定时电路。

现在我们明白了什么是硬定时器，接下来我们将讨论Linux内核如何利用它们来指挥系统中的所有活动。

Linux 计时体系结构

Linux 必定执行与定时相关的操作。例如，内核周期性地：

- 更新自系统启动以来所经过的时间。
- 更新时间和日期。
- 确定当前进程在每个CPU 上已运行了多长时间，如果已经超过了分配给它的时间，则抢占它。时间片（也叫时限）的分配将在第七章讨论。
- 更新资源使用统计数。
- 检查每个软定时器（参见后面“软定时器和延迟函数”一节）的时间间隔是否已到。

Linux 的计时体系结构（*timekeeping architecture*）是一组与时间流相关的内核数据结构和函数。实际上，基于80x86 多处理器机器所具有的计时体系结构与单处理器机器所具有的稍有不同：

- 在单处理器系统上，所有的计时活动都是由全局定时器（可以是可编程间隔定时器也可以是高精度事件定时器）产生的中断触发的。
- 在多处理器系统上，所有普通的活动（像软定时器的处理）都是由全局定时器产生的中断触发的，而具体 CPU 的活动（像监控当前运行进程的执行时间）是由本地 APIC 定时器产生的中断触发的。

可惜，以上两种情况的区别有点模糊。例如，某些早期基于 Intel 80486 处理器的 SMP 系统不拥有本地 APIC。即使到了今天，还有一些 SMP 主板是有瑕疵的，因此本地时钟中断根本不稳定。在这些情况下，SMP 内核必须采用单处理器系统（UP)的计时体系结构。另一方面，新近的单处理器系统拥有本地 APIC，因此 UP 内核通常可以使用 SMP 的计时体系结构。不过，为了简化我们的讨论，我们不打算讨论这些混杂的情况，而集中于两种“纯”的计时体系结构。

Linux 的计时体系结构还依赖于时间戳计数器（TSC）、ACPI 电源管理定时器、高精度事件定时器（HPET）的可用性。内核使用两个基本的计时函数：一个保持当前最新的时间，另一个计算在当前秒内走过的纳秒数。有几种不同的方式获得后一个值：如果 CPU 有 TSC 或 HPET，就可以用一些更精确的方法；在其他情况下，使用精确性差一些的方法（参见后面“time()和 gettimeofday()系统调用”一节）。

计时体系机构的数据结构

Linux 2.6 的计时体系结构使用了大量的数据结构。与以往一样，我们将描述 80x86 体系结构下最重要的变量。

定时器对象

为了使用一种统一的方法来处理可能存在的定时器资源，内核使用了“定时器对象”，它是timer_opts类型的一个描述符，该类型由定时器名称和四个标准的方法组成，如表6-1所示。

表6-1：timer_opts数据结构的各个字段

字段名	说明
name	标识定时器源的一个字符串
mark_offset	记录上一个节拍的准确时间，由时钟中断处理程序调用
get_offset	返回自上一个节拍开始所经过的时间
monotonic_clock	返回自内核初始化开始所经过的纳秒数
delay	等待指定数目的“循环”（参见后面的“延迟函数”一节）

定时器对象中最重要的方法是mark_offset和get_offset。mark_offset方法由时钟中断处理程序调用，并以适当的数据结构记录每个节拍到来时的准确时间。get_offset方法使用已记录的值来计算自上一次时钟中断（节拍）以来经过的时间（以μs为单位）。由于这两种方法，使得Linux计时体系结构能够达到子节拍的分辨度，也就是说，内核能够以比节拍周期更高的精度来测定当前的时间。这种操作被称作“定时插补(*time interpolation*)”。

变量cur_timer存放了某个定时器对象的地址，该定时器是系统可利用的定时器资源中“最好的”。最初，cur_timer指向timer_none，这个timer_none是一个虚拟的定时器资源对象，内核在初始化的时候使用它。在内核初始化期间，select_timer()函数设置cur_timer指向适当定时器对象的地址。表6-2以优先级顺序列出了80x86体系结构中最常用的定时器对象。正如你所看到的，select_timer()将优先选择HPET（如果可以使用）；否则，将选择ACPI电源管理定时器（如果可以使用）；再次之是TSC。作为最后的方案，select_timer()选择总是存在的PIT。“定时插补”一列列出了定时器对象的mark_offset方法和get_offset方法所使用的定时器源，“延迟”一列列出了delay方法使用的定时器源。

表6-2：80x86体系结构下典型的定时器对象，以优先权顺序排列

定时器对象名称	说明	定时插补	延迟
timer_hpet	高精度事件定时器（HPET）	HPET	HPET
timer_pmtmr	ACPI电源管理定时器（ACPI PMT）	ACPI PMT	TSC
timer_tsc	时间戳计数器（TSC）	TSC	TSC
timer_pit	可编程间隔定时器（PIT）	PIT	紧致循环
timer_none	普通虚拟定时器资源（内核初始化时使用）	（无）	紧致循环

注意，本地APIC定时器没有对应的定时器对象。因为本地APIC定时器仅用来产生周期性中断而从不用来获得子节拍的分辨度。

jiffies变量

jiffies变量是一个计数器，用来记录自系统启动以来产生的节拍总数。每次时钟中断发生时（每个节拍）它便加1。在80x86体系结构中，jiffies是一个32位的变量，因此每隔大约50天它的值会回绕（wraparound）到0，这对Linux服务器来说是一个相对较短的时间间隔。不过，由于使用了time_after、time_after_eq、time_before和time_before_eq四个宏（即使发生回绕它们也能产生正确的值），内核干净利索地处理了jiffies变量的溢出。

你可能猜想jiffies在系统启动的时候被初始化为0。实际上，事实并非如此：jiffies被初始化为0xfffb6c20，它是一个32位的有符号值，正好等于-300 000。因此，计数器将会在系统启动后的5分钟内处于溢出状态。这样做是有目的的，使得那些不对jiffies作溢出检测的有缺陷的内核代码在开发阶段被及时地发现，从而不再出现在稳定的内核版本中。

但是在某些情况下，不管jiffies是否溢出，内核需要自系统启动以来产生的系统节拍的真实数目。因此，在80x86系统中，jiffies变量通过连接器被换算成一个64位计数器的低32位，这个64位的计数器被称作jiffies_64。在1ms为一个节拍的情况下，jiffies_64变量将会在数十亿年后才发生回绕，所以我们可以放心地假定它不会溢出。

你可能要问为什么在80x86体系结构中jiffies不直接被声明成64位无符号的长整型数。答案是:在32位的体系结构中不能自动地对64位的变量进行访问。因此，在每次执行对64位数的读操作时，需要一些同步机制来保证当两个32位的计数器（由这两个32位的计数器组成的64位计数器）的值在被读取时这个64位的计数器不会被更新，结果是，每个64位的读操作明显比32位的读操作更慢。

get_jiffies_64()函数用来读取jiffies_64的值并返回该值：

```
unsigned long long get_jiffies_64(void)
{
    unsigned long seq;
    unsigned long long ret;
    do {
       seq = read_seqbegin(&xtime_lock);
       ret = jiffies_64;
    } while (read_seqretry(&xime_lock, seq));
    return ret;
}
```

xtime_lock顺序锁用来保护64位的读操作（参见第五章的“顺序锁”一节）：该函数一

直读 jiffies_64 变量直到确认该变量并没有同时被其他内核控制路径更新时才读取 jiffies_64 变量。

相反，当在临界区增加 jiffies_64 变量的值时必须使用 write_seqlock(&xtime_lock)和 write_sequnlock(&xtime_lock)进行保护。注意++jiffies_64操作同时也会增加32位的 jiffies 变量的值，因为后者对应着 jiffies_64 的低 32 位。

xtime 变量

xtime变量存放当前时间和日期，它是一个timespec类型的数据结构，该结构有两个字段：

tv_sec
: 存放自 1970 年 1 月 1 日（UTC）午夜以来经过的秒数

tv_nsec
: 存放自上一秒开始经过的纳秒数（它的值域范围在 0～999999999 之间）

xtime变量通常是每个节拍更新一次，也就是说，大约每秒更新1000次。正如我们将在后面的“与定时测量相关的系统调用”一节看到的那样，用户程序从xtime变量获得当前时间和日期。内核也经常引用它，例如，在更新节点时间戳时引用（参见第一章的“文件描述符与索引节点”一节）。

xtime_lock顺序锁（seqlock）消除了对xtime变量的同时访问而可能发生的竞争条件。记住xtime_lock同样也保护jiffies_64变量。一般而言，这个顺序锁用来定义计时体系结构中的一些临界区。

单处理器系统上的计时体系结构

在单处理器系统上，所有与定时有关的活动都是由IRQ线0上的可编程间隔定时器产生的中断触发的。同样，在Linux中，某些活动都尽可能在中断产生后立即执行，而其余的活动延迟（参见稍后的“动态定时器”一节）。

初始化阶段

在内核初始化期间，time_init()函数被调用来建立计时体系结构，它通常（注3）执行如下操作：

注3：time_init()函数在mem_init()之前被执行，它初始化内存数据结构。遗憾的是，HPET寄存器是由内存映射的，因此HPET芯片的初始化必须在mem_init()执行之后完成。Linux 2.6采用了一种麻烦的解决办法：如果内核支持HPET芯片，time_init()函数就限定自己去触发hpet_time_init()而使其激活。hpet_time_init()函数在mem_init()之后被执行并执行本节所描述的操作。

1. 初始化xtime变量。利用get_cmos_time()函数从实时时钟上读取自1970年1月1日(UTC)午夜以来经过的秒数。设置xtime的tv_nsec字段，这样使得即将发生的jiffies变量溢出与tv_sec字段的增加保持一致，也就是说，它将落到秒的范围内。

2. 初始化wall_to_monotonic变量。这个变量同xtime一样是timespec类型，只不过它存放将被加到xtime上的秒数和纳秒数，以此来获得单向(只增)的时间流。其实，外部时钟的闰秒和同步都有可能突发地改变xtime的tv_sec和tv_nsec字段，这样使得它们不再是单向递增的。正如我们将在后面的“与POSIX定时器相关的系统调用”一节看到的那样，有时内核需要一个真正单向的时间源。

3. 如果内核支持HPET，它将调用hpet_enable()函数来确认ACPI固件是否探测到了该芯片并将它的寄存器映射到了内存地址空间中。如果结果是肯定的，那么hpet_enable()将对HPET芯片的第一个定时器编程使其以每秒1000次的频率引发IRQ 0处的中断。否则，如果不能获得HPET芯片，内核将使用PIT：该芯片已经被init_IRQ()函数编程，使得它以每秒1000次的频率引发IRQ 0处的中断，正如前面的“可编程间隔定时器(PIT)”一节描述的那样。

4. 调用select_timer()来挑选系统中可利用的最好的定时器资源，并设置cur_timer变量指向该定时器资源对应的定时器对象的地址。

5. 调用setup_irq(0,&irq0)来创建与IRQ0相应的中断门，IRQ0引脚线连接着系统时钟中断源(PIT或HPET)。irq0变量被静态定义如下：

```
struct irqaction irq0 = { timer_interrupt, SA_INTERRUPT, 0,
                          "timer", NULL, NULL };
```

从现在起，timer_interrupt()函数将会在每个节拍到来时被调用，而中断被禁止，因为IRQ0主描述符的状态字段中的SA_INTERRUPT标志被置位。

时钟中断处理程序

timer_interrupt()函数是PIT或HPET的中断服务例程(ISR)，它执行以下步骤：

1. 在xtime_lock顺序锁上产生一个write_seqlock()来保护与定时相关的内核变量(参见第五章“顺序锁”一节)。

2. 执行cur_timer定时器对象的mark_offset方法。正如前面的“计时体系结构的数据结构”一节解释的那样，有四种可能的情况：

   a. cur_timer指向timer_hpet对象：这种情况下，HPET芯片作为时钟中断源。mark_offset方法检查自上一个节拍以来是否丢失时钟中断，在这种不太可能发生的情况下，它会相应地更新jiffies_64。接着，该方法记录下HPET周期计数器的当前值。

b. cur_timer指向timer_pmtmr对象：这种情况下，PIT芯片作为时钟中断源，但是内核使用APIC电源管理定时器以更高的分辨度来测量时间。mark_offset方法检查自上一个节拍以来是否丢失时钟中断，如果丢失则更新jiffies_64。然后，它记录APIC电源管理定时器计数器的当前值。

c. cur_timer指向timer_tsc对象：这种情况下，PIT芯片作为时钟中断源，但是内核使用时间戳计数器以更高的分辨度来测量时间。mark_offset方法执行与上一种情况相同的操作：检查自上一个节拍以来是否丢失时钟中断，如果丢失则更新jiffies_64。然后，它记录TSC计数器的当前值。

d. cur_timer指向timer_pit对象：这种情况下，PIT芯片作为时钟中断源，除此之外没有别的定时器电路。mark_offset方法什么也不做。

3. 调用do_timer_interrupt()函数，do_timer_interrupt()函数执行以下操作：

a. 使jiffies_64的值增1。注意，这样做是安全的，因为内核控制路径仍然为写操作保持着xtime_lock顺序锁。

b. 调用update_times()函数来更新系统日期和时间，并计算当前系统负载。这些活动将在稍后的“更新时间和日期”与“更新系统统计数”两节中讨论。

c. 调用update_process_times()函数为本地CPU执行几个与定时相关的计数操作（参见本章后面的“更新本地CPU统计数”一节）。

d. 调用profile_tick()函数（参见本章后面的“监管内核代码”一节）。

e. 如果使用外部时钟来同步系统时钟（以前已发出过adjtimex()系统调用），则每隔660秒（每隔11分钟）调用一次set_rtc_mmss()函数来调整实时时钟。这个特性用来帮助网络中的系统同步它们的时钟（参见后面的“adjtimex()系统调用”一节）。

4. 调用write_sequnlock()释放xtime_lock顺序锁。

5. 返回值1，报告中断已经被有效地处理了（参见第四章的“I/O中断处理”一节）。

多处理器系统上的计时体系结构

多处理器系统可以依赖两种不同的时钟中断源：可编程间隔定时器或高精度事件定时器产生的中断，以及CPU本地定时器产生的中断。

在Linux 2.6中，PIT或HPET产生的全局时钟中断触发不涉及具体CPU的活动，比如处理软定时器和保持系统时间的更新。相反，一个CPU本地时钟中断触发涉及本地CPU的计时活动，例如监视当前进程的运行时间和更新资源使用统计数。

初始化阶段

全局时钟中断处理程序由time_init()函数初始化，我们已经在前面“单处理器系统上的计时体系结构”一节对该函数作了描述。

Linux内核为本地时钟中断保留第239号（0xep）中断向量（参见第四章表4-2）。在内核初始化阶段，函数apic_intr_init()根据第239号向量和低级中断处理程序apic_timer_interrupt()的地址设置IDT的中断门。此外，每个APIC必须被告知多久产生一次本地时钟中断。函数calibrate_APIC_clock()通过正在启动的CPU的本地APIC来计算在一个节拍内（1 ms）收到了多少个总线时钟信号。然后这个确切的值被用来对本地所有APIC编程，并由此在每个节拍产生一次本地时钟中断。这是由setup_APIC_timer()函数完成的，该函数被系统中的每个CPU执行一次。

所有本地APIC定时器都是同步的，因为它们都基于公共总线时钟信号。这意味着用于引导CPU的calibrate_APIC_clock()函数计算出来的值对系统中的其他CPU同样有效。

全局时钟中断处理程序

SMP版本的timer_interrupt()处理程序与UP版本的该处理程序在几个地方有差异：

- timer_interrupt()调用函数do_timer_interrupt()向I/O APIC芯片的一个端口写入，以应答定时器的中断请求。
- update_process_times()函数不被调用，因为该函数执行与特定CPU相关的操作。
- profile_tick()不被调用，因为该函数同样执行与特定CPU相关的操作。

本地时钟中断处理程序

该处理程序执行系统中与特定CPU相关的计时活动，即监管内核代码并检测当前进程在特定CPU上已经运行了多长时间。

汇编语言函数apic_timer_interrupt()等价于下面的代码：

```
apic_timer_interrupt:
    pushl $(239-256)
    SAVE_ALL
    movl %esp, %eax
    call smp_apic_timer_interrupt
    jmp ret_from_intr
```

正如你所见，该低级处理函数与第四章中描述过的其他低级中断处理函数非常相似。被称作smp_apic_timer_interrupt()的高级中断处理函数执行如下步骤：

1. 获得CPU逻辑号(比如说*n*)。
2. 使irq_stat数组中第*n*项的apic_timer_irqs字段加1（参见本章后面的“检查非屏蔽中断（NMI）监视器”一节）。
3. 应答本地APIC上的中断。
4. 调用irq_enter()函数（参见第四章“do_IRQ()函数”一节）。
5. 调用smp_local_timer_interrupt()函数。
6. 调用irq_exit()函数。

smp_local_timer_interrupt()函数执行每个CPU的计时活动。事实上，它执行下面的主要步骤：

1. 调用profile_tick()函数（参见本章后面的“监管内核代码”一节）。
2. 调用update_process_times()函数检查当前进程运行的时间并更新一些本地CPU统计数（参见本章后面的“更新本地CPU统计数”一节）。

系统管理员通过写入*/proc/profile*文件可以修改内核代码监管器的抽样频率。为实现修改，内核改变本地时钟中断产生的频率。不过，smp_local_timer_interrupt()函数保持每个节拍精确调用update_process_times()函数一次。

更新时间和日期

用户程序从xtime变量中获得当前时间和日期。内核必须周期性地更新该变量，才能使它的值保持相当的精确。

全局时钟中断处理程序调用update_times()函数更新xtime变量的值，代码如下：

```
void update_times(void)
{
    unsigned long ticks;
    ticks = jiffies - wall_jiffies;
    if (ticks) {
        wall_jiffies += ticks;
        update_wall_time(ticks);
    }
    calc_load(ticks);
}
```

我们回忆一下前面对时钟中断处理程序的描述，当执行该函数代码时，已经获得用于写操作的xtime_lock顺序锁。

wall_jiffies变量存放xtime变量最后更新的时间。观察一下，wall_jiffies的值可以小于jiffies-1，因为一些定时器中断会丢失，例如当中断被禁止了很长一段时间；换句话说，内核不必每个时钟节拍更新xtime变量。然而，最后不会有时钟节拍丢失，因此，xtime最终存放正确的系统时间。对丢失的定时器中断的检查在cur_timer的mark_offset中完成，参见前面的“单处理器系统上的计时体系结构”一节。

update_wall_time()函数连续调用update_wall_time_one_tick()函数ticks次，每次调用都给xtime.tv_nsec字段加上1000000。如果xtime.tv_nsec的值大于999999999，那么update_wall_time()函数还会更新xtime的tv_sec字段。如果系统发出adjtimex()系统调用，那么函数可能会稍微调整1000000这个值使时钟稍快或稍慢一点（原因请参见本章后面的“adjtimex()系统调用”一节）。

calc_load()函数的描述请参见本章后面的“记录系统负载”一节。

更新系统统计数

内核在与定时相关的其他任务中必须周期性地收集若干数据用于：

- 检查运行进程的CPU资源限制
- 更新与本地CPU工作负载有关的统计数
- 计算平均系统负载
- 监管内核代码

更新本地CPU统计数

我们曾经提到过，单处理器系统上的全局时钟中断处理程序或多处理器系统上的本地时钟中断处理程序调用update_process_times()函数来更新一些内核统计数。该函数执行以下步骤：

1. 检查当前进程运行了多长时间。当时钟中断发生时，根据当前进程运行在用户态还是内核态，选择调用account_user_time()还是account_system_time()。每个函数基本上执行如下步骤：
   a. 更新当前进程描述符的utime字段（用户态下所经过的节拍数）或stime字段（内核态下所经过的节拍数）。在进程描述符中提供两个被称作cutime和cstime的附加字段，分别用来统计子进程在用户态和内核态下所经过的CPU节拍数。由于效率的原因，update_process_times()并不更新这些字段，而

只是当父进程询问它的其中一个子进程的状态时才对其进行更新(参见第三章的“撤销进程”一节)。

b. 检查是否已达到总的CPU时限，如果是，向current进程发送SIGXCPU和SIGKILL信号。在第三章的“进程资源限制”一节中，讲述了限制是如何被每个进程描述符的signal->rlim[RLIMIT_CPU].rlim_cur字段所控制的。

c. 调用account_it_virt()和account_it_prof()来检查进程定时器（参见本章后面的“setitimer()和alarm()系统调用”一节）。

d. 更新一些内核统计数，这些统计数存放在每CPU变量kstat中。

2. 调用raise_softirq()来激活本地CPU上的TIMER_SOFTIRQ任务队列(参见本章后面的“软定时器和延迟函数”一节)。

3. 如果必须回收一些老版本的、受RCU保护的数据结构，那么检查本地CPU是否经历了静止状态并调用tasklet_schedule()来激活本地CPU的rcu_tasklet任务队列（参见第五章的“读-拷贝-更新(RCU)”一节)。

4. 调用scheduler_tick()函数，该函数使当前进程的时间片计数器减1，并检查计数器是否已减到0。我们将在第七章的“scheduler_tick()函数”一节深入讨论这些操作。

记录系统负载

任何Unix内核都要记录系统进行了多少CPU活动。这些统计数据由各种管理实用程序来使用（如top)。用户输入uptime命令后可以看到一些统计数据：如相对于最后1分钟、5分钟、15分钟的“平均负载”。在单处理器系统上，值0意味着没有活跃的进程(除了*swapper*进程0）在运行，而值1意味着一个单独的进程100%占有CPU，值大于1说明几个运行着的进程共享CPU（注4)。

update_times()在每个节拍都要调用calc_load()函数来计算处于TASK_RUNNING或TASK_UNINTERRUPTIBLE状态的进程数，并用这个数据更新平均系统负载。

监管内核代码

Linux包含一个被称作*readprofiler*的最低要求的代码监管器，Linux开发者用其发现内

注4： Linux在平均负载中包含所有处于TASK_RUNNIG和TASK_UNINTERRUPTIBLE状态的进程。然而，一般情况下，很少有进程处于TASK_UNINTERRUPTIBLE状态，因此，高负载通常指CPU是繁忙的。

核在内核态的什么地方花费时间。监管器确定内核的“热点”（*hot spot*）——执行最频繁的内核代码片段。确定内核“热点”是非常重要的，因为这可以指出应当进一步优化的内核函数。

监管器基于非常简单的蒙特卡洛算法：在每次时钟中断发生时，内核确定该中断是否发生在内核态；如果是，内核从堆栈取回中断发生前的eip寄存器的值，并用这个值揭示中断发生前内核正在做什么。最后，采样数据积聚在“热点”上。

profile_tick()函数为代码监管器采集数据。这个函数在单处理器系统上是由do_timer_interrupt()调用的（即全局时钟中断处理程序调用的），在多处理器系统上是由smp_local_timer_interrupt()函数调用的（即本地时钟中断处理程序调用的）。

为了激活代码监管器，在Linux内核启动时必须传递字符串参数“profile=N”，这里2^N表示要监管的代码段的大小。采集的数据可以从*/proc/profile*文件中读取。可以通过修改这个文件来重置计数器；在多处理器系统上，修改这个文件还可以改变抽样频率（参见前面“多处理器系统上的计时体系结构”一节）。不过，内核开发者并不直接访问*/proc/profile*文件，而是用*readprofile*系统命令。

Linux 2.6内核还包含了另一个监管器，叫做*oprofile*。比起*readprofile*，*oprofile*除了更灵活、更可定制外，还能用于发现内核代码、用户态应用程序以及系统库中的热点。当使用*oprofile*时，profile_tick()调用timer_notify()函数来收集这个新监管器所使用的数据。

检查非屏蔽中断（NMI）监视器

在多处理器系统上，Linux为内核开发者还提供了另外一种功能：看门狗系统（*watchdog system*），这对于探测引起系统冻结的内核bug可能相当有用。为了激活这样的看门狗，必须在内核启动时传递nmi_watchdog参数。

看门狗基于本地和I/O APIC一个巧妙的硬件特性：它们能在每个CPU上产生周期性的NMI中断。因为NMI中断是不能用汇编语言指令cli屏蔽的，所以，即使禁止中断，看门狗也能检测到死锁。

因而，一旦每个时钟节拍到来，所有的CPU，不管其正在做什么，都开始执行NMI中断处理程序；该中断处理程序又调用do_nmi()。这个函数获得CPU的逻辑号*n*，然后检查irq_stat数组第*n*项的apic_timer_irqs字段（参见第四章的表4-8）。如果该CPU字段工作正常，那么，第*n*项的值必定不同于在前一个NMI中断中读出的值。当CPU正常运行时，第*n*项的apic_timer_irq字段就会被本地时钟中断处理程序增加（参见

前面“本地时钟中断处理程序”一节），如果计数器没有被增加，说明本地时钟中断处理程序在整个时钟节拍期间根本就没有被执行。你可以想到，这可不是什么好事。

当NMI中断处理程序检测到一个CPU冻结时，就会敲响所有的钟：它把引起恐慌的信息记录在系统日志文件中，转储该CPU寄存器的内容和内核栈（内核OOP）的内容，最后杀死当前进程。这就为内核开发者提供了发现错误的机会。

软定时器和延迟函数

定时器是一种软件功能，即允许在将来的某个时刻，函数在给定的时间间隔用完时被调用。超时（*time-out*）表示与定时器相关的时间间隔已经用完的那个时刻。

内核和进程广泛使用定时器。大多数设备驱动程序利用定时器检测反常情况，例如，软盘驱动程序使用定时器在软盘暂时不被访问后就关闭设备的发动机，而并行打印机设备利用定时器检测错误的打印机情况。

编程人员也经常利用定时器在将来某一时刻强制执行特定的函数（参见后面的“setitimer()和alarm()系统调用”一节）。

相对来说，实现一个定时器并不难。每个定时器都包含一个字段，表示定时器将需要多长时间才到期。这个字段的初值就是jiffies的当前值加上合适的节拍数。这个字段的值不再改变。每当内核检查定时器时，就把这个到期字段值和当前这一刻jiffies的值相比较，当jiffies大于或等于这个字段存放的值时，定时器到期。

Linux考虑两种类型的定时器，即动态定时器（*dynamic timer*）和间隔定时器（*interval timer*）。第一种类型由内核使用，而间隔定时器可以由进程在用户态创建。

这里是有关Linux定时器的警告：因为对定时器函数的检查总是由可延迟函数进行，而可延迟函数被激活以后很长时间才能被执行，因此，内核不能确保定时器函数正好在定时到期时开始执行，而只能保证在适当的时间执行它们，或者假定延迟到几百毫秒之后执行它们。因此，对于必须严格遵守定时时间的那些实时应用而言，定时器并不适合。

除了软定时器外，内核还使用了延迟函数，它执行一个紧凑的指令循环直到指定的时间间隔用完。我们将在后面的“延迟函数”一节对它们进行讨论。

动态定时器

动态定时器（*dynamic timer*）被动态地创建和撤消，对当前活动动态定时器的个数没有限制。

动态定时器存放在下列timer_list结构中：

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;
    spinlock_t lock;
    unsigned long magic;
    void (*function)(unsigned long);
    unsigned long data;
    tvec_base_t *base;
};
```

function字段包含定时器到期时执行函数的地址。data字段指定传递给定时器函数的参数。正是由于data字段，就可以定义一个单独的通用函数来处理多个设备驱动程序的超时问题，在data字段可以存放设备ID，或其他有意义的数据，定时器函数可以用这些数据区分不同的设备。

expires字段给出定时器到期时间，时间用节拍数表示，其值为系统启动以来所经过的节拍数。当expires的值小于或等于jiffies的值时，就说明计时器到期或终止。

entry字段用于将软定时器插入双向循环链表队列中，该链表根据定时器expires字段的值将它们分组存放。我们将在本章后面描述使用这些链表的算法。

为了创建并激活一个动态定时器，内核必须：

1. 如果需要，创建一个新的timer_list对象，比如说设为t。这可以通过以下几种方式来进行：
   - 在代码中定义一个静态全局变量。
   - 在函数内定义一个局部变量：在这种情况下，这个对象存放在内核堆栈中。
   - 在动态分配的描述符中包含这个对象。
2. 调用init_timer(&t)函数初始化这个对象。实际上是把t.base指针字段置为NULL并把t.lock自旋锁设为“打开”。
3. 把定时器到期时激活函数的地址存入function字段。如果需要，把传递给函数的参数值存入data字段。
4. 如果动态定时器还没有被插入到链表中，给expires字段赋一个合适的值并调用add_timer(&t)函数把t元素插入到合适的链表中
5. 否则，如果动态定时器已经被插入到链表中，则调用mod_timer()函数来更新expires字段，这样也能将对象插入到合适的链表中（下面将讨论）。

一旦定时器到期，内核就自动地把元素t从它的链表中删除。不过，有时进程应该用del_timer()、del_timer_sync()或del_singleshot_timer_sync()函数显式地从定时器链表中删除一个定时器。事实上，在定时器到期前，睡眠的进程可能被唤醒，在这种情况下，唤醒的进程就可以选定撤消某个定时器。虽然从链表中已删除的定时器上调用del_timer()没什么害处，不过，在定时器函数内删除定时器是一种良好的习惯做法。

在Linux 2.6中，动态定时器需要CPU来激活，也就是说，定时器函数总会在第一个执行add_timer()或稍后执行mod_timer()函数的那同一个CPU上运行。不过，del_timer()及与其类似的函数能使所有动态定时器无效，即使该定时器并不依赖于本地CPU激活。

动态定时器与竞争条件

被异步激活的动态定时器有参与竞争条件的倾向。例如，考虑一个动态定时器，它的函数作用于可丢弃的资源（例如，内核模块或文件数据结构）。如果在定时器函数被激活时资源不再存在，那么不停止定时器就释放资源势必导致数据结构的崩溃。因此，一种凭经验的做法就是在释放资源前停止定时器：

```
...
del_timer(&t);
X_Release_Resources();
...
```

然而，在多处理器系统上，这段代码是不安全的，因为当调用del_timer()函数时，定时器函数可能已经在其他CPU上运行了。结果，当定时器函数还作用在资源上时，资源可能被释放。为了避免这种竞争条件，内核提供了del_timer_sync()函数。这个函数从链表中删除定时器，然后检查定时器函数是否还在其他CPU上运行；如果是，del_timer_sync()就等待，直到定时器函数结束。

del_timer_sync()函数相当复杂，而且执行速度慢，因为它必须小心考虑这种情况：定时器函数重新激活它自己。如果内核开发者知道定时器函数从不重新激活定时器，她就能使用更简单更快速的del_singleshot_timer_sync()函数来使定时器无效，并等待直到定时器函数结束。

当然，也存在其他种类的竞争条件。例如，修改已激活定时器expires字段的正确方法是调用mod_timer()，而不是删除定时器随后又创建它。在后一种途径中，要修改同一定时器expires字段的两个内核控制路径可能糟糕地交错在一起。定时器函数在SMP上的安全实现是通过每个timer_list对象包含的lock自旋锁达到的：每当内核必须访问动态定时器的链表时，就禁止中断并获取这个自旋锁。

动态定时器的数据结构

选择合适的数据结构实现动态定时器并不是件容易的事。把所有定时器放在一个单独的链表中会降低系统的性能，因为在每个时钟节拍去扫描一个定时器的长链表太费时。另一方面，维护一个排序的链表效率也不高，因为插入和删除操作也非常费时。

解决的办法基于一种巧妙的数据结构，即把expires值划分成不同的大小，并允许动态定时器从大expires值的链表到小expires值的链表进行有效的过滤。此外，在多处理器系统中活动的动态定时器集合被分配到各个不同的CPU中。

动态定时器的主要数据结构是一个叫做tvec_bases的每CPU变量（参见第五章的“每CPU变量”一节）：它包含NR_CPUS个元素，系统中每个CPU各有一个。每个元素是一个tvec_base_t类型的数据结构，它包含相应CPU中处理动态定时器需要的所有数据。

```
typedef struct tvec_t_base_s {
    spinlock_t lock;
    unsigned long timer_jiffies;
    struct timer_list *running_timer;
    tvec_root_t tv1;
    tvec_t tv2;
    tvec_t tv3;
    tvec_t tv4;
    tvec_t tv5;
} tvec_base_t;
```

字段tv1的数据结构为tvec_root_t类型，它包含一个vec数组，这个数组由256个list_head元素组成（即由256个动态定时器链表组成）。这个结构包含了在紧接着到来的255个节拍内将要到期的所有动态定时器。

字段tv2、tv3和tv4的数据结构都是tvec_t类型，该类型有一个数组vec（包含64个list_head元素）。这些链表包含在紧接着到来的2^{14}-1、2^{20}-1以及2^{26}-1个节拍内将要到期的所有动态定时器。

字段tv5与前面的字段几乎相同，但唯一区别就是vec数组的最后一项是一个大expires字段值的动态定时器链表。tv5从不需要从其他的数组补充。图6-1用图例说明了5个链表组。

timer_jiffies字段的值表示需要检查的动态定时器的最早到期时间：如果这个值与jiffies的值一样，说明可延迟函数没有积压；如果这个值小于jiffies，说明前几个节拍相关的可延迟函数必须处理。该字段在系统启动时被设置成jiffies的值，且只能由run_timer_softirq()函数（将在下一节描述）增加它的值。注意当处理动态定时器的可延迟函数在很长一段时间内都没有被执行时（例如由于这些函数被禁止或者已经执行了大量中断处理程序），timer_jiffies字段可能会落后jiffies许多。

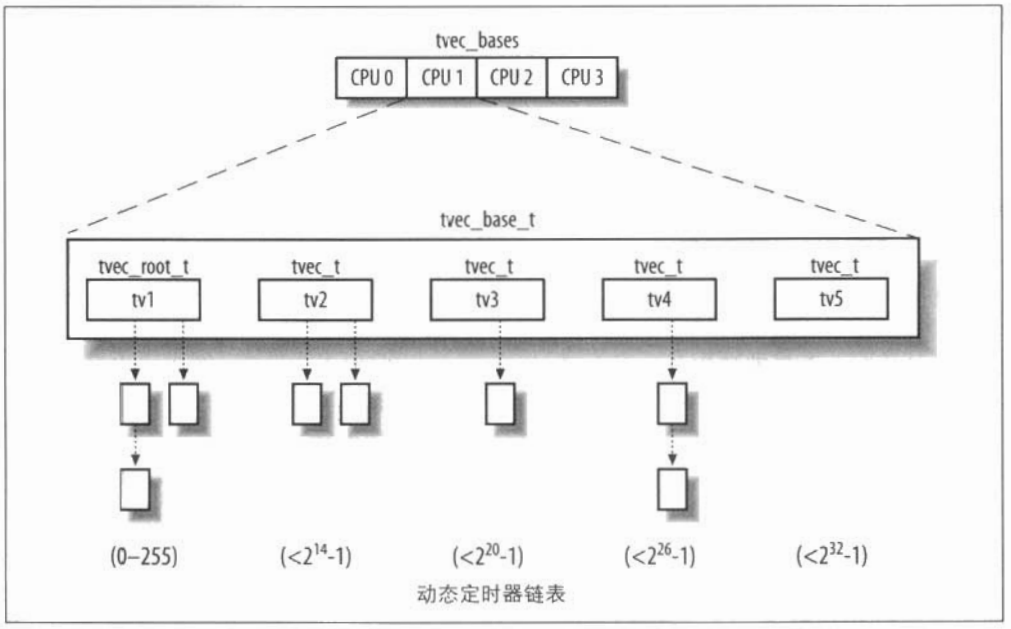


图 6-1：与动态定时器相关的链表组

在多处理器系统中，字段 running_timer 指向由本地 CPU 当前正处理的动态定时器的 timer_list 数据结构。

动态定时器处理

尽管软定时器具有巧妙的数据结构，但是对其处理是一种耗时的活动，所以不应该被时钟中断处理程序执行。在Linux 2.6中该活动由可延迟函数来执行，也就是由TIMER_SOFTIRQ软中断执行。

run_timer_softirq()函数是与 TIMER_SOFTIRQ 软中断请求相关的可延迟函数。它实质上执行如下操作：

1. 把与本地 CPU 相关的 tvec_base_t 数据结构的地址存放到 base 本地变量中。
2. 获得 base->lock 自旋锁并禁止本地中断。
3. 开始执行一个 while 循环，当 base->timer_jiffies 大于 jiffies 的值时终止。在每一次循环过程中，执行下列子步骤：
   a. 计算 base->tv1 中链表的索引，该索引保存着下一次将要处理的定时器：

```
index = base->timer_jiffies & 255;
```

b. 如果索引值为0，说明base->tv1中的所有链表已经被检查过了，所以为空：于是该函数通过调用cascade()来过滤动态定时器：

```
if (!index &&
  (!cascade(base, &base->tv2, (base->timer_jiffies>> 8)&63)) &&
  (!cascade(base, &base->tv3, (base->timer_jiffies>>14)&63)) &&
  (!cascade(base, &base->tv4, (base->timer_jiffies>>20)&63)))
    cascade(base, &base->tv5, (base->timer_jiffies>>26)&63);
```

考虑第一次调用cascade()函数的情况：它接收base的地址、base->tv2的地址、base->tv2（包括在紧接着到来的256个节拍内将要到期的定时器）中链表的索引作为参数。该索引值是通过观察base->timer_jiffies的特殊位上的值来决定的。cascade()函数将base->tv2中链表上的所有动态定时器移到base->tv1的适当链表上。然后，如果所有base->tv2中的链表不为空，它返回一个正值。如base->tv2中的链表为空，cascade()将再次被调用，把base->tv3中的某个链表上包含的定时器填充到base->tv2上，如此等等。

c. 使base->timer_jiffies的值加1。

d. 对于base->tv1.vec[index]链表上的每一个定时器，执行它所对应的定时器函数。特别说明的是，链表上的每个timer_list元素t实质上执行以下步骤：

(1) 将t从base->tv1的链表上删除。

(2) 在多处理器系统中，将base->running_timer设置为&t（t的地址）。

(3) 设置t.base为NULL。

(4) 释放base->lock自旋锁，并允许本地中断。

(5) 传递t.data作为参数，执行定时器函数t.function。

(6) 获得base->lock自旋锁，并禁止本地中断。

(7) 如果链表中还有其他定时器，则继续处理。

e. 链表上的所有定时器已经被处理。继续执行最外层while循环的下一次循环。

4. 最外层的while循环结束，这就意味着所有到期的定时器已经被处理了。在多处理器系统中，设置base->running_timer为NULL。

5. 释放base->lock自旋锁并允许本地中断。

由于jiffies和timer_jiffies的值经常是一样的，所以最外层的while循环常常只执行一次。一般情况下，最外层循环会连续执行jiffies - base->timer_jiffies + 1次。此外，如果在run_timer_softirq()正在执行时发生了时钟中断，那么也得考虑在这个节拍所出现的到期动态定时器，因为jiffies变量的值是由全局时钟中断处理程序异步增加的（参见前面的“时钟中断处理程序”一节）。

请注意，就在进入最外层循环前，run_timer_softirq()要禁止中断并获取base->lock自旋锁；调用每个动态定时器函数前，激活中断并释放自旋锁，直到函数执行结束。这就保证了动态定时器的数据结构不被交错执行的内核控制路径所破坏。

综上所述可知，这种相当复杂的算法确保了极好的性能。让我们来看看为什么，为了简单起见，假定TIMER_SOFTIRQ软中断正好在相应的时钟中断发生后执行。那么，在256次中出现的255次时钟中断（也就是在99.6%的情况中），run_imter_ softirq ()仅仅运行到期定时器的函数。为了周期性地补充base->tv1.vec，在64次补充当中，63次足以把base->tv2指向的链表分成base->tv1指向的256个链表。依次地，base->tv2.vec数组必须在0.006%的情况下得到补充，即每16.4秒一次。类似地，每17分28秒补充一次base->tv3.vec，每18小时38分补充一次base->tv4.vec，而base->tv5.vec不需被补充。

动态定时器应用之一：nanosleep()系统调用

为了说明前面所有活动的结果如何在内核中实际使用，我们给出创建和使用进程延时的例子。

让我们考虑nanosleep()系统调用的服务例程，即sys_nanosleep()，它接收一个指向timespec结构的指针作为参数，并将调用进程挂起直到特定的时间间隔用完。服务例程首先调用copy_from_user()将包含在timespec结构（用户态下）中的值复制到局部变量t中。假设timespec结构定义了一个非空的延迟，接着函数执行如下代码：

```
current->state = TASK_INTERRUPTIBLE;
remaining = schedule_timeout(timespec_to_jiffies(&t)+1);
```

timespec_to_jiffies()函数将存放在timespec结构中的时间间隔转换成节拍数。为保险起见，sys_nanosleep()为timespec_to_jiffies()计算出的值加上一个节拍。

内核使用动态定时器来实现进程的延时。它们出现在schedule_timeout()函数中，该函数执行下列语句：

```
struct timer_list timer;
unsigned long expire = timeout + jiffies;
init_timer(&timer);
timer.expires = expire;
timer.data = (unsigned long) current;
timer.function = process_timeout;
add_timer(&timer);
schedule();      /* 进程挂起直到定时器到时 */
del_singleshot_timer_sync(&timer);
timeout = expire - jiffies;
return (timeout < 0 ? 0 : timeout);
```

当schedule()被调用时，就选择另一个进程执行；当前一个进程恢复执行时，该函数就删除这个动态定时器。在最后一句中，函数返回的值有两种可能，0表示延时到期，timeout表示如果进程因某些其他原因被唤醒，到延时到期时还剩余的节拍数。

当延时到期时，内核执行下列函数：

```
void process_timeout(unsigned long __data)
{
    wake_up_process((task_t *)__data);
}
```

process_timeout()接收进程描述符指针作为它的参数，该指针存放在定时器对象的data字段。结果，挂起的进程被唤醒。

一旦进程被唤醒，它就继续执行sys_nanosleep()系统调用。如果schedule_timeout()返回的值表明进程延时到期（值为0），系统调用就结束。否则，系统调用将自动重新启动，正如第十一章的“系统调用的重新执行”一节中解释的那样。

延迟函数

当内核需要等待一个较短的时间间隔——比方说，不超过几毫秒时，就无需使用软定时器。例如，通常设备驱动器会等待预先定义的数个微秒直到硬件完成某些操作。由于动态定时器通常有很大的设置开销和一个相当大的最小等待时间（1ms），所以设备驱动器使用它会很不方便。

在这些情况下，内核使用udelay()和ndelay()函数：前者接收一个微秒级的时间间隔作为它的参数，并在指定的延迟结束后返回；后者与前者类似，但是指定延迟的参数是纳秒级的。

本质上两个函数定义如下：

```
void udelay(unsigned long usecs)
{
    unsigned long loops;
    loops = (usecs*HZ*current_cpu_data.loops_per_jiffy)/1000000;
    cur_timer->delay(loops);
}

void ndelay(unsigned long nsecs)
{
    unsigned long loops;
    loops = (nsecs*HZ*current_cpu_data.loops_per_jiffy)/1000000000;
    cur_timer->delay(loops);
}
```

两个函数都依赖于cur_timer定时器对象的delay方法（参见前面的“计时体系结构的数据结构”一节），它接收“loops”中的时间间隔作为参数。不过每一次“loop”精确的持续时间取决于cur_timer涉及的定时器对象（参见本章前面的表6-2）。

- 如果cur_timer指向timer_hpet、timer_pmtmr和timer_tsc对象，那么一次“loop”对应于一个CPU循环——也就是两个连续CPU时钟信号间的时间间隔（参见前面的“时间戳计数器（TSC）”一节）。
- 如果cur_timer指向timer_none或timer_pit对象，那么一次“loop”对应于一条紧凑指令循环在一次单独的循环中所花费的时间。

在初始化阶段，select_timer()设置好cur_timer后，内核通过执行calibrate_delay()函数来决定一个节拍里有多少次“loop”。这个值被保存在current_cpu_data.loops_per_jiffy变量中，这样udelay()和ndelay()就能根据它来把微秒和纳秒转换成“loops”。

当然，如果可以利用HPET或TSC硬件电路，那么cur_timer->delay()方法使用它们来获得精确的时间测量。否则，该方法执行一个紧凑指令循环的loops次循环。

与定时测量相关的系统调用

有几个系统调用允许用户态下的进程读取及修改时间和日期，以及创建定时器。让我们对它们进行一些简单的回顾，并讨论一下内核是如何处理它们的。

time()和gettimeofday()系统调用

用户态下的进程通过以下几个系统调用获得当前时间和日期：

time()

返回从1970年1月1日午夜（UTC）开始所走过的秒数。

gettimeofday()

返回从1970年1月1日午夜（UTC）开始所走过的秒数及在前1秒内走过的微秒数，这个值存放在数据结构timeval中（第二个叫做timezone的数据结构目前还没有使用）。

time()系统调用被gettimeofday()取代，但是，为了保持向后兼容，Linux中还包含它们。另一个被广泛使用的函数ftime()不再作为一个系统调用来执行，它返回从1970年1月1日午夜（UTC）开始所走过的秒数与前1秒内所走过的毫秒数。

gettimeofday()系统调用由sys_gettimeofday()函数实现。为了计算一天中的当前时间和日期，这个函数又调用do_ gettimeofday()，它执行下列动作：

1. 为读操作获取xtime_lock顺序锁。
2. 调用cur_timer定时器对象的get_offset方法来确定自上一次时钟中断以来所走过的微秒数。

```
usec = cur_timer->getoffset();
```

正如前面的“计时体系结构的数据结构”一节解释的那样，这里有四种可能的情况：

a. 如果cur_timer指向timer_hpet对象，该方法将HPET计数器的当前值与上一次时钟中断处理程序执行时在同一个计数器中保存的值相比较。

b. 如果cur_timer指向timer_pmtmr对象，该方法将ACPI PMT计数器的当前值与上一次时钟中断处理程序执行时在同一个计数器里保存的值相比较。

c. 如果cur_timer指向timer_tsc对象，该方法将时间戳计数器的当前值与上一次时钟中断处理程序执行时在同一个TSC里保存的值相比较。

d. 如果cur_timer指向timer_pit对象，该方法读取PIT计数器的当前值来计算自上一次PIT时钟中断以来所走过的微秒数。

3. 如果某定时器中断丢失(参见本章前面的“更新时间和日期”一节)，该函数为usec加上相应的延迟：

```
usec += (jiffies - wall_jiffies) * 1000;
```

4. 为usec加上前1秒内走过的微秒数：

```
usec += (xtime.tv_nsec / 1000);
```

5. 将xtime的内容复制到系统调用参数tv指定的用户空间缓冲区中，并给微秒字段的值加上usec：

```
tv->tv_sec = xtime->tv_sec;
tv->tv_usec = usec;
```

6. 在xtime_lock 顺序锁上调用read_seqretry()，并且如果另一条内核控制路径同时为写操作而获得了xtime_lock，则跳回到步骤1。
7. 检查微秒字段是否溢出，如果必要则调整该字段和秒字段：

```
while (tv->tv_usec >= 1000000) {
    tv->tv_usec -= 1000000;
    tv->tv_sec++;
}
```

拥有root权限的用户态下的进程可以用stime()和settimeofday()中任意一种系统调

用来修改当前日期和时间。sys_settimeofday()函数调用do_settimeofday()，该函数执行do_gettimeofday()操作的反操作。

请注意当这两个系统调用修改xtime的值时都没有修改RTC寄存器，因此当系统关机时新的时间会丢失，除非用户执行 *clock* 程序来改变RTC的值。

adjtimex()系统调用

尽管时钟的走动确保了所有的系统最终都会从恰当的时间离开，但是，突然改变时间既是一种管理的失误也是一种危险的行为。例如，设想程序员试图编译一个大规模的程序，并依靠文件时间标记来确保旧的文件对象被重新编译。系统时间大的改动可能搞乱make程序，并导致不正确的编译。当在计算机网络上执行一个分布式文件系统时，保持时钟的调整也是很重要的。在这种情况下，明智的做法是，调整互连PC的时钟以使所存取文件的inode中的时间标记值都保持一致。因此，通常把系统配置成能在常规基准上运行时间同步协议，例如网络定时协议（NTP），以在每个节拍逐渐地调整时间。在Linux中，这个实用程序依赖于adjtimex()系统调用。

尽管这个系统调用不应该用在打算移植的程序中，但它还是出现在几个Unix变种中。adjtimex()接收指向timex结构的指针作为参数，用timex字段中的值更新内核参数，并返回具有当前内核值的同一结构。update_wall_time_one_tick()使用这样的内核值对每一个节拍中加到xtime.tv_usec的微秒数进行细微地调整。

setitimer() 和 alarm()系统调用

Linux允许用户态的进程激活一种叫做间隔定时器的特殊定时器（注5）。这种定时器引起的Unix信号（参见第十一章）被周期性地发送到进程。也可能激活一个间隔定时器以便在指定的延时后它仅发送一个信号。因此，间隔定时器由以下两个方面来刻画：

- 发送信号所必需的频率，或者如果只需要产生一个信号，则频率为空
- 在下一个信号被产生以前所剩余的时间

在本章前面关于精确性的警告同样适用于这些定时器。在要求的时间已过去之后，可以确保这些定时器执行，但是不可能预知恰好在什么时候会执行它们。

通过POSIX setitimer()系统调用可以激活间隔定时器。第一个参数指定应当采取下面的哪一个策略：

注5： 这些软件的构造与本章前面所描述的可编程间隔定时器没有什么共同之处。

ITIMER_REAL

真正过去的时间；进程接收 SIGALRM 信号

ITIMER_VIRTUAL

进程在用户态下花费的时间；进程接收 SIGVTALRM 信号

ITIMER_PROF

进程既在用户态下又在内核态下所花费的时间；进程接收 SIGPROF 信号

间隔定时器既能一次执行也能周期性循环。setitimer()的第二个参数指向一个itimerval类型的结构，它指定了定时器初始的持续时间（以s和ns为单位）以及定时器被自动重新激活后使用的持续时间(对于一次性执行的定时器而言为0)。setitimer()的第三个参数是一个指针，它是可选的，指向一个 itimerval 类型的结构，系统调用将先前定时器的参数填充到该结构中。

为了能分别实现前述每种策略的间隔定时器，进程描述符要包含 3 对字段：

- it_real_incr 和 it_real_value
- it_virt_incr 和 it_virt_value
- it_prof_incr 和 it_prof_value

每对中的第一个字段存放着两个信号之间以节拍为单位的间隔；另一个字段存放着定时器的当前值。

ITIMER_REAL 间隔定时器是利用动态定时器实现的，因为即使进程不在 CPU 上运行时，内核也必须向进程发送信号。因此，每个进程描述符包含一个叫 real_timer 的动态定时器对象。setitimer()系统调用初始化real_timer字段，然后调用add_timer()把动态定时器插入到合适的链表中。当定时器到期时，内核执行 it_real_fn()定时器函数。it_real_fn()函数又向进程发送一个 SIGALRM 信号。如果 it_real_incr 不为空，那么它会再次设置 expires 字段，并重新激活定时器。

ITIMER_VIRTUAL和ITIMER_PROF间隔定时器不需要动态定时器，因为只有当进程运行时，它们才能被更新。account_it_virt()和account_it_prof()由 update_process_times()调用，而update_ process_times()在单处理器上由 PIT 的时钟中断处理程序调用，在多处理器上由本地时钟中断处理程序调用。因此，每个节拍中，这两个间隔定时器都被更新一次，并且如果它们到期，就给当前进程发送一个合适的信号。

alarm()系统调用会在一个指定的时间间隔用完时向调用的进程发送一个SIGALRM信号。当以 ITIMER_REAL 为参数调用时，它非常类似于 setitimer()，因为它利用了包

含在进程描述符中的real_timer动态定时器。因此，具有ITIMER_REAL参数的alarm()和setitimer()不能同时使用。

与POSIX定时器相关的系统调用

POSIX 1003.1b标准为用户态程序引入了一种新型软定时器，尤其是针对多线程和实时应用程序。这些定时器常被称作POSIX定时器。

要执行每个POSIX定时器，必须向用户态程序提供一些*POSIX*时钟，也就是说，虚拟时间源预定义了分辨度和属性。只要应用程序想使用POSIX定时器，它就创建一个新的定时器资源并指定一个现存的POSIX时钟来作为定时基准。表6-3列出了允许用户来处理POSIX时钟和定时器的一些系统调用。

表6-3：与POSIX定时器和时钟相关的系统调用

系统调用	说明
clock_gettime()	获得一个POSIX时钟的当前值
clock_settime()	设置一个POSIX时钟的当前值
clock_getres()	获得一个POSIX时钟的分辨度
timer_create()	在指定POSIX时钟基础上创建一个新的POSIX定时器
timer_gettime()	获得一个POSIX定时器的当前值和增量
timer_settime()	设置一个POSIX定时器的当前值和增量
timer_getoverrun()	获得到期POSIX定时器到期的数目
timer_delete()	销毁一个POSIX定时器
clock_nanosleep()	使进程进入睡眠状态并使用一个POSIX时钟作为时间源

Linux 2.6内核提供两种类型的POSIX时钟：

CLOCK_REALTIME

该虚拟时钟表示系统的实时时钟——本质上是xtime变量的值（参见前面的“更新时间和日期”一节）。clock_getres()系统调用返回的分辨度为999 848ns，对应1s内更新xtime大约1000次。

CLOCK_MONOTONIC

该虚拟时钟表示由于与外部时间源的同步，每次回到初值的系统实时时钟。实际上，该虚拟时钟由xtime和wall_to_monotonic两个变量的和表示（参见前面“单处理器系统上的计时体系结构”一节）。该POSIX时钟的分辨度由clock_getres()返回，返回值为999 848ns。

Linux内核使用动态定时器来实现POSIX定时器。因此，它们与我们在前面一节描述的ITIMER_REAL间隔定时器相似。不过，POSIX定时器比传统间隔定时器更灵活、更可靠。它们之间有两个显著区别：

- 当传统间隔定时器到期时，内核会发送一个SIGALRM信号给进程来激活定时器。而当一个POSIX定时器到期时，内核可以发送各种信号给整个多线程应用程序，也可以发送给单个指定的线程。内核还能在应用程序的某个线程上强制执行一个通告器函数，或者甚至什么也不做（这取决于处理事件的用户态函数库）。
- 如果一个传统间隔定时器到期了很多次但用户态进程不能接收SIGALRM信号（例如由于信号被阻塞或者进程不处于运行态），那么只有第一个信号被接收到，其他所有SIGALRM信号都丢失了。对于POSIX定时器来说会发生同样的情况，但进程可以调用timer_getoverrun()系统调用来得到自第一个信号产生以来定时器到期的次数。

第七章



进程调度

Linux与任何分时系统一样，通过一个进程到另一个进程的快速切换，达到表面上看来多个进程同时执行的神奇效果。进程切换本身已在第三章中讨论过，本章讨论进程调度（*schednling*），主要关心什么时候进行进程切换及选择哪一个进程来运行。

本章由三部分组成。“调度策略”一节从理论上介绍Linux进行进程调度所做的选择。“调度算法”一节讨论实现调度所采用的数据结构和相应的算法。最后，“与调度相关的系统调用”一节描述了影响进程调度的系统调用。

为了叙述起来更简单，我们仍以80x86体系结构为例；尤其是，我们假定系统采用统一内存访问（Uniform Memory Access）模型，而且系统时钟设定为1ms。

调度策略

传统Unix操作系统的调度算法必须实现几个互相冲突的目标：进程响应时间尽可能快，后台作业的吞吐量尽可能高，尽可能避免进程的饥饿现象，低优先级和高优先级进程的需要尽可能调和等等。决定什么时候以怎样的方式选择一个新进程运行的这组规则就是所谓的调度策略（*scheduling policy*）。

Linux的调度基于分时（*time sharing*）技术：多个进程以“时间多路复用”方式运行，因为CPU的时间被分成“片（*slice*）”，给每个可运行进程分配一片（注1）。当然，单处

注1：　调度算法不会选择已被停止和挂起的进程在CPU上运行。

理器在任何给定的时刻只能运行一个进程。如果当前运行进程的时间片或时限（*quantum*）到期时，该进程还没有运行完毕，进程切换就可以发生。分时依赖于定时中断，因此对进程是透明的。不需要在程序中插入额外的代码来保证CPU分时。

调度策略也是根据进程的优先级对它们进行分类。有时用复杂的算法求出进程当前的优先级，但最后的结果是相同的：每个进程都与一个值相关联，这个值表示把进程如何适当地分配给CPU。

在Linux中，进程的优先级是动态的。调度程序跟踪进程正在做什么，并周期性地调整它们的优先级。在这种方式下，在较长的时间间隔内没有使用CPU的进程，通过动态地增加它们的优先级来提升它们。相应地，对于已经在CPU上运行了较长时间的进程，通过减少它们的优先级来处罚它们。

当谈及有关调度的问题时，传统上把进程分类为“I/O受限（*I/O-bound*）”或“CPU受限（*CPU-bound*）”。前者频繁地使用I/O设备，并花费很多时间等待I/O操作的完成；而后者则需要大量CPU时间的数值计算应用程序。

另一种分类法把进程区分为三类：

交互式进程（interactive process）

这些进程经常与用户进行交互，因此，要花很多时间等待键盘和鼠标操作。当接受了输入后，进程必须被很快唤醒，否则用户将发现系统反应迟钝。典型的情况是，平均延迟必须在50～150ms之间。这样的延迟变化也必须进行限制，否则用户将发现系统是不稳定的。典型的交互式程序是命令shell、文本编辑程序及图形应用程序。

批处理进程（batch process）

这些进程不必与用户交互，因此经常在后台运行。因为这样的进程不必被很快地响应，因此常受到调度程序的慢待。典型的批处理进程是程序设计语言的编译程序、数据库搜索引擎及科学计算。

实时进程（real-time process）

这些进程有很强的调度需要。这样的进程决不会被低优先级的进程阻塞，它们应该有一个短的响应时间，更重要的是，响应时间的变化应该很小。典型的实时程序有视频和音频应用程序、机器人控制程序及从物理传感器上收集数据的程序。

我们刚刚提到的两种分类法在一定程度上相互独立。例如，一个批处理进程可能是I/O受限型的（如数据库服务器），或是CPU受限型的（如图像绘制程序）。在Linux中，调度算法可以明确地确认所有实时程序的身份，但没有办法区分交互式程序和批处理程序。Linux 2.6调度程序实现了基于进程过去行为的启发式算法，以确定进程应该被当作交互式进程还是批处理进程。当然，与批处理进程相比，调度程序有偏爱交互式进程的倾向。

程序员可以通过表7-1所列的系统调用改变调度优先级。更详细的内容将在“与调度相关的系统调用”一节中给出。

表 7-1：与调度相关的系统调用

系统调用	说明
nice()	改变一个普通进程的静态优先级
getpriority()	获得一组普通进程的最大静态优先级
setpriority()	设置一组普通进程的静态优先级
sched_getscheduler()	获得一个进程的调度策略
sched_setscheduler()	设置一个进程的调度策略和实时优先级
sched_getparam()	获得一个进程的实时优先级
sched_setparam()	设置一个进程的实时优先级
sched_yield()	自愿放弃处理器而不阻塞
sched_get_ priority_min()	获得一种策略的最小实时优先级
sched_get_ priority_max()	获得一种策略的最大实时优先级
sched_rr_get_interval()	获得时间片轮转策略的时间片值
sched_setaffinity()	设置进程的 CPU 亲和力掩码
sched_getaffinity()	获得进程的 CPU 亲和力掩码

进程的抢占

如第一章所述，Linux的进程是抢占式的。如果进程进入TASK_RUNNING状态，内核检查它的动态优先级是否大于当前正运行进程的优先级。如果是，current的执行被中断，并调用调度程序选择另一个进程运行（通常是刚刚变为可运行的进程）。当然，进程在它的时间片到期时也可以被抢占。此时，当前进程thread_info结构中的TIF_NEED_RESCHED标志被设置，以便时钟中断处理程序终止时调度程序被调用。

例如，让我们考虑一种情况，在这种情况中，只有两个程序——一个文本编辑程序和一个编译程序——正在执行。文本编辑程序是一个交互式程序，因此它的动态优先级高于编译程序。不过，因为编辑程序交替于用户暂停思考与数据输入之间，因此它经常被挂起；此外，两次击键之间的平均延迟相对较长。然而，只要用户一按键，中断就发生，内核唤醒文本编辑进程。内核也确定编辑进程的动态优先级确实高于current的优先级（当前正运行的进程，即编译进程），因此，编辑进程的TIF_NEED_RESCHED标志被设置，如此来强迫内核处理完中断时激活调度程序。调度程序选择编辑进程并执行进程切换；结果，编辑进程很快恢复执行，并把用户键入的字符回显在屏幕上。当处理完字符时，文本编辑进程自己挂起等待下一次击键，编译进程恢复执行。

注意，被抢占的进程并没有被挂起，因为它还处于TASK_RUNNING状态，只不过不再使用CPU。此外，记住Linux 2.6内核是抢占式的，这意味着进程无论是处于内核态还是用户态，都可能被抢占，我们曾在第五章的“内核抢占”一节深入讨论过这个特征。

一个时间片必须持续多长？

时间片的长短对系统性能是很关键的：它既不能太长也不能太短。

如果平均时间片太短，由进程切换引起的系统额外开销就变得非常高。例如，假定进程切换需要5ms，如果时间片也设置为5ms，那么CPU至少把50%的时间花费在进程切换上（注2）。

如果平均时间片太长，进程看起来就不再是并发执行。例如，让我们假定把时间片设置为5s，那么，每个可运行进程运行大约5s，但是暂停的时间更长（一般是5s乘以可运行进程的个数）。

通常认为长的时间片会降低交互式应用程序的响应时间，但这往往是错误的。正如本章前面“进程的抢占”一节中所描述的那样，交互式进程相对有较高的优先级，因此，不管时间片是多长，它们都会很快地抢占批处理进程。

然而在一些情况下，一个太长的时间片会降低系统的响应能力。例如，假定两个用户在各自的shell提示符下并发输入两条命令，其中一条启动一个CPU受限型的进程，而另一条启动一个交互式应用。两个shell都创建一个新进程，并把用户命令的执行委托给新进程。此外，又假定这样的新进程最初有相同的优先级（Linux预先并不知道执行进程是批处理的还是交互式的）。现在，如果调度程序选择CPU受限型的进程执行，则另一个进程开始执行前就可能要等待一个时间片。因此，如果这样的时间片较长，那么看起来系统就可能对用户的请求反应迟钝。

对时间片大小的选择始终是一种折衷。Linux采取单凭经验的方法，即选择尽可能长、同时能保持良好响应时间的一个时间片。

调度算法

早期Linux版本中的调度算法非常简单易懂：在每次进程切换时，内核扫描可运行进程的链表，计算进程的优先级，然后选择“最佳”进程来运行。这个算法的主要缺点是选

注2：实际上，情况可能更糟糕。例如，如果在进程的时间片中还要计算进程切换所需的时间，那么所有的CPU时间都会花费在进程切换上，就没有进程能执行完。

择“最佳”进程所要消耗的时间与可运行的进程数量相关，因此，这个算法的开销太大，在运行数千个进程的高端系统中要消耗太多的时间。

Linux 2.6的调度算法就复杂多了。通过设计，该算法较好地解决了与可运行进程数量的比例关系，因为它在固定的时间内（与可运行的进程数量无关）选中要运行的进程。它也很好地处理了与处理器数量的比例关系，因为每个CPU都拥有自己的可运行进程队列。而且，新算法较好地解决了区分交互式进程和批处理进程的问题。因此，在高负载的系统中，用户感到在Linux2.6中交互应用的响应速度比早期的Linux版本要快。

调度程序总能成功地找到要执行的进程。事实上，总是至少有一个可运行进程，即*swapper*进程，它的PID等于0，而且它只有在CPU不能执行其他进程时才执行。就像在第三章中提到的，每个多处理器系统的CPU都有它自己的*swapper*进程，其PID等于0。

每个Linux进程总是按照下面的调度类型被调度：

SCHED_FIFO
: 先进先出的实时进程。当调度程序把CPU分配给进程的时候，它把该进程描述符保留在运行队列链表的当前位置。如果没有其他可运行的更高优先级实时进程，进程就继续使用CPU，想用多久就用多久，即使还有其他具有相同优先级的实时进程处于可运行状态。

SCHED_RR
: 时间片轮转的实时进程。当调度程序把CPU分配给进程的时候，它把该进程的描述符放在运行队列链表的末尾。这种策略保证对所有具有相同优先级的SCHED_RR实时进程公平地分配CPU时间。

SCHED_NORMAL
: 普通的分时进程。

调度算法根据进程是普通进程还是实时进程而有很大不同。

普通进程的调度

每个普通进程都有它自己的静态优先级，调度程序使用静态优先级来估价系统中这个进程与其他普通进程之间调度的程度。内核用从100（最高优先级）到139（最低优先级）的数表示普通进程的静态优先级。注意，值越大静态优先级越低。

新进程总是继承其父进程的静态优先级。不过，通过把某些“nice值”传递给系统调用nice()和setpriority()（参见本章稍后“与调度相关的系统调用”一节），用户可以改变自己拥有的进程的静态优先级。

基本时间片

静态优先级本质上决定了进程的基本时间片，即进程用完了以前的时间片时，系统分配给进程的时间片长度。静态优先级和基本时间片的关系用下列公式确定：

$$\text{基本时间片（单位为 ms）} = \begin{cases} (140 - \text{静态优先级}) \times 20 & \text{若静态优先级} < 120 \\ (140 - \text{静态优先级}) \times 5 & \text{若静态优先级} \geqslant 120 \end{cases} \quad (1)$$

如你所见，静态优先级越高（其值越小），基本时间片就越长。其结果是，与优先级低的进程相比，通常优先级较高的进程获得更长的CPU时间片。表7-2说明了相对于拥有最高静态优先级、拥有默认静态优先级和拥有最低静态优先级的普通进程，其静态优先级、基本时间片和对应的nice值（表中还列出了交互式的δ值和睡眠时间的极限值，在本章稍后给予说明）。

表 7-2：普通进程优先级的典型值

说明	静态优先级	nice 值	基本时间片	交互式的 δ 值	睡眠时间的极限值
最高静态优先级	100	-20	800ms	-3	299ms
高静态优先级	110	-10	600ms	-1	499ms
缺省静态优先级	120	0	100ms	+2	799ms
低静态优先级	130	+10	50ms	+4	999ms
最低静态优先级	139	+19	5ms	+6	1199ms

动态优先级和平均睡眠时间

普通进程除了静态优先级，还有动态优先级，其值的范围是100（最高优先级）～139（最低优先级）。动态优先级是调度程序在选择新进程来运行的时候使用的数。它与静态优先级的关系用下面的经验公式表示。

$$\text{动态优先级} = \max(100, \min(\text{静态优先级} - \textit{bonus} + 5, 139)) \quad (2)$$

*bonus*是范围从0～10的值，值小于5表示降低动态优先级以示惩罚，值大于5表示增加动态优先级以示奖赏。bonus的值依赖于进程过去的情况，说得更准确一些，是与进程的平均睡眠时间相关。

粗略地讲，平均睡眠时间是进程在睡眠状态所消耗的平均纳秒数。注意，这绝对不是对过去时间的求平均值的操作。例如，在TASK_INTERRUPTIBLE状态与在TASK_UNINTERRUPTIBLE

状态所计算出的平均睡眠时间是不同的。而且，进程在运行的过程中平均睡眠时间递减。最后，平均睡眠时间永远不会大于1s。

表7-3说明了平均睡眠时间和bonus值的关系。(表中还列出了相应的时间片粒度，这将在稍后讨论。)

表 7-3：平均睡眠时间、bonus 值以及时间片粒度

平均睡眠时间	bonus	粒度
大于或等于0小于100ms	0	5120
大于或等于100ms小于200ms	1	2560
大于或等于200ms小于300ms	2	1280
大于或等于300ms小于400ms	3	640
大于或等于400ms小于500ms	4	320
大于或等于500ms小于600ms	5	160
大于或等于600ms小于700ms	6	80
大于或等于700ms小于800ms	7	40
大于或等于800ms小于900ms	8	20
大于或等于900ms小于1000ms	9	10
1 s	10	10

平均睡眠时间也被调度程序用来确定一个给定进程是交互式进程还是批处理进程。更明确地说，如果一个进程满足下面的公式，就被看作是交互式进程：

$$\text{动态优先级} \leqslant 3 \times \text{静态优先级}/4+28 \tag{3}$$

它相当于下面的公式：

$$\text{bonus}-5 \geqslant \text{静态优先级}/4-28$$

表达式：静态优先级/4-28被称为交互式的δ；交互式δ的一些典型值在表7-2中列出。应该注意，高优先级进程比低优先级进程更容易成为交互式进程。例如，具有最高静态优先级（100）的进程，当它的bonus值超过2，即睡眠时间超过200ms时，就被看作是交互式进程。相反，具有最低静态优先级（139）的进程决不会被当作交互式进程，因为bonus值总是小于11，相应地需要交互式δ等于6。一个具有缺省静态优先级（120）的进程，一但其平均睡眠时间超过700ms，就成为交互式进程。

活动和过期进程

即使具有较高静态优先级的普通进程获得了较大的CPU时间片，也不应该使静态优先级较低的进程无法运行。为了避免进程饥饿，当一个进程用完它的时间片时，它应该被还没有用完时间片的低优先级进程取代。为了实现这种机制，调度程序维持两个不相交的可运行进程的集合。

活动进程

这些进程还没有用完它们的时间片，因此允许它们运行。

过期进程

这些可运行进程已经用完了它们的时间片，并因此被禁止运行，直到所有活动进程都过期。

不过，总体的方案要稍微复杂一些，因为调度程序试图提升交互式进程的性能。用完其时间片的活动批处理进程总是变成过期进程。用完其时间片的交互式进程通常仍然是活动进程：调度程序重填其时间片并把它留在活动进程集合中。但是，如果最老的过期进程已经等待了很长时间，或者过期进程比交互式进程的静态优先级高，调度程序就把用完时间片的交互式进程移到过期进程集合中。结果，活动进程集合最终会变为空，过期进程将有机会运行。

实时进程的调度

每个实时进程都与一个实时优先级相关，实时优先级是一个范围从1（最高优先级）～99（最低优先级）的值。调度程序总是让优先级高的进程运行，换句话说，实时进程运行的过程中，禁止低优先级进程的执行。与普通进程相反，实时进程总是被当成活动进程（参见上一节）。用户可以通过系统调用sched_setparam()和sched_setscheduler()改变进程的实时优先级（参见本章稍后“与调度相关的系统调用”一节）。

如果几个可运行的实时进程具有相同的最高优先级，那么调度程序选择第一个出现在与本地CPU的运行队列相应链表中的进程（参见第三章“TASK_RUNNING状态的进程链表”）。

只有在下述事件之一发生时，实时进程才会被另外一个进程取代：

- 进程被另外一个具有更高实时优先级的实时进程抢占。
- 进程执行了阻塞操作并进入睡眠（处于TASK_INTERRUPTIBLE或TASK_UNINTERRUPTIBLE状态）。
- 进程停止（处于TASK_STOPPED或TASK_TRACED状态）或被杀死（处于EXIT_ZOMBIE或EXIT_DEAD状态）。

- 进程通过调用系统调用 sched_yield()(参见本章稍后的“与调度相关的系统调用”一节）自愿放弃CPU。
- 进程是基于时间片轮转的实时进程（SCHED_RR），而且用完了它的时间片。

当系统调用nice()和setpriority()用于基于时间片轮转的实时进程时，不改变实时进程的优先级而会改变其基本时间片的长度。实际上，基于时间片轮转的实时进程的基本时间片的长度与实时进程的优先级无关，而依赖于进程的静态优先级，它们的关系见前面“普通进程的调度“一节中的公式（1）。

调度程序所使用的数据结构

回忆第三章“标识一个进程”一节，进程链表链接所有的进程描述符，而运行队列链表链接所有的可运行进程（也就是处于TASK_RUNNING状态的进程）的进程描述符，*swapper*进程（idle进程）除外。

数据结构runqueue

数据结构runqueue是Linux 2.6调度程序最重要的数据结构。系统中的每个CPU都有它自己的运行队列，所有的runqueue结构存放在runqueues每CPU变量中（参见第五章“每CPU变量”一节）。宏this_rq()产生本地CPU运行队列的地址，而宏cpu_rq(n)产生索引为n 的CPU的运行队列的地址。

表7-4列出了runqueue数据结构所包括的字段，在下面的章节中我们将对其中的大部分字段进行讨论。

表7-4：runquere结构的字段

类型	名称	说明
spinlock_t	lock	保护进程链表的自旋锁
unsigned long	nr_running	运行队列链表中可运行进程的数量
Unsigned long	cpu_load	基于运行队列中进程的平均数量的CPU负载因子
unsigned long	nr_switches	CPU执行进程切换的次数
unsigned long	nr_uninterruptible	先前在运行队列链表中而现在睡眠在TASK_UNINTERRUPTIBLE状态的进程的数量（对所有运行队列来说，只有这些字段的总数才是有意义的）

表 7-4：runquere 结构的字段（续）

类型	名称	说明
unsigned long	expired_timestamp	过期队列中最老的进程被插入队列的时间
unsigned long long	timestamp_last_tick	最近一次定时器中断的时间戳的值
task_t *	curr	当前正在运行进程的进程描述符指针（对本地 CPU，它与 current 相同）
task_t *	idle	当前 CPU（this CPU）上 *swapper* 进程的进程描述符指针
struct mm_struct *	prev_mm	在进程切换期间用来存放被替换进程的内存描述符的地址
prio_array_t *	active	指向活动进程链表的指针
prio_array_t *	expired	指向过期进程链表的指针
prio_array_t [2]	arrays	活动进程和过期进程的两个集合
int	best_expired_prio	过期进程中静态优先级最高的进程（权值最小）
atomic_t	nr_iowait	先前在运行队列的链表中而现在正等待磁盘 I/O 操作结束的进程的数量
struct sched_domain *	sd	指向当前 CPU 的基本调度域（见本章稍后“调度域”一节）
int	active_balance	如果要把一些进程从本地运行队列迁移到另外的运行队列（平衡运行队列），就设置这个标志
int	push_cpu	未使用
task_t *	migration_thread	迁移内核线程的进程描述符指针
struct list_head	migration_queue	从运行队列中被删除的进程的链表

runqueue数据结构中最重要的字段是与可运行进程的链表相关的字段。系统中的每个可运行进程属于且只属于一个运行队列。只要可运行进程保持在同一个运行队列中，它就只可能在拥有该运行队列的CPU上执行。但是，正如我们将要看到的，可运行进程会从一个运行队列迁移到另一个运行队列。

运行队列的arrays字段是一个包含两个prio_array_t结构的数组。每个数据结构都表示一个可运行进程的集合，并包括140个双向链表头（每个链表对应一个可能的进程优先级）、一个优先级位图和一个集合中所包含的进程数量的计数器（参见第三章的表3-2）。

如图7-1所示，runqueue结构的active字段指向arrays中两个prio_array_t数据结构之一：对应于包含活动进程的可运行进程的集合。相反，expired字段指向数组中的另一个prio_array_t数据结构：对应于包含过期进程的可运行进程的集合。

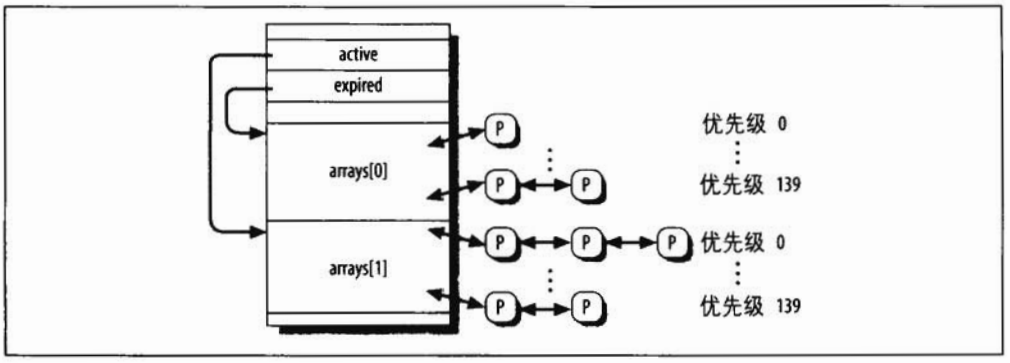


图7-1：runqueue结构和可运行进程的两个集合

ayyays中两个数据结构的作用会发生周期性的变化：活动进程突然变成过期进程，而过期进程变为活动进程，调度程序简单地交换运行队列的active和expired字段的内容以完成这种变化。

进程描述符

每个进程描述符都包括几个与调度相关的字段，如表7-5所示。

表7-5：与调度程序相关的进程描述符字段

类型	名称	说明
unsigned long	thread_info->flags	存放TIF_NEED_RESCHED标志，如果必须调用调度程序，则设置该标志（见第四章“从中断和异常返回”一节）
unsigned int	thread_info->cpu	可运行进程所在运行队列的CPU逻辑号
unsigned long	state	进程的当前状态（见第三章“进程状态”一节）
int	prio	进程的动态优先级
int	static_prio	进程的静态优先级
struct list_head	run_list	指向进程所属的运行队列链表中的下一个和前一个元素

表 7-5：与调度程序相关的进程描述符字段（续）

类型	名称	说明
prio_array_t*	array	指向包含进程的运行队列的集合 prio_array_t
unsigned long	sleep_avg	进程的平均睡眠时间
unsigned long long	timestamp	进程最近插入运行队列的时间，或涉及本进程的最近一次进程切换的时间
unsigned long long	last_ran	最近一次替换本进程的进程切换时间
int	activated	进程被唤醒时所使用的条件代码
unsigned long	policy	进程的调度类型(SCHED_NORMAL、SCHED_RR 或 SCHED_FIFO)
cpumask_t	cpus_allowed	能执行进程的 CPU 的位掩码
unsigned int	time_slice	在进程的时间片中还剩余的时钟节拍数
unsigned int	first_time_slice	如果进程肯定不会用完其时间片，就把该标志设置为 1
unsigned long	rt_priority	进程的实时优先级

当新进程被创建的时候，由 copy_process()调用的函数 sched_fork()用下述方法设置 current 进程（父进程）和 p 进程（子进程）的 time_slice 字段：

```
p->time_slice = (current->time_slice + 1) >> 1;
current->time_slice >>= 1;
```

换句话说，父进程剩余的节拍数被划分成两等份：一份给父进程，另一份给子进程。这样做是为了避免用户通过下述方法获得无限的CPU时间：父进程创建一个运行相同代码的子进程，并随后杀死自己，通过适当地调节创建的速度，子进程就可以总是在父进程过期之前获得新的时间片。因为内核不奖赏创建，所以这种编程技巧不起作用。类似地，用户不能通过在shell中运行几个后台进程，或通过在图形桌面打开许多窗口来不公平地霸占处理器。更通俗地讲就是，一个进程不能通过创建多个后代来霸占资源（除非它有给自己实时策略的特权）。

如果父进程的时间片只剩下一个时钟节拍，则划分操作强行把current->time_slice置为0，从而耗尽父进程的时间片。这种情况下，copy_process()把current->time_slice重新置为 1，然后调用 scheduler_tick()递减该字段（见下一节）。

函数 copy_process()也初始化子进程描述符中与进程调度相关的几个字段：

```
p->first_time_slice = 1;
p->timestamp = sched_clock();
```

因为子进程没有用完它的时间片(如果一个进程在它的第一个时间片内终止或执行新的程序，就把子进程的剩余时间奖励给父进程)，所以first_time_slice标志被置为1。用函数sched_clock()所产生的时间戳的值初始化timestamp字段：实际上，函数sched_clock()返回被转化成纳秒的64位寄存器TSC[见第六章“时间戳计数器（TSC)”一节]的内容。

调度程序所使用的函数

调度程序依靠几个函数来完成调度工作．其中最重要的函数是：

scheduler_tick()
: 维持当前最新的time_slice计数器。

try_to_wake_up()
: 唤醒睡眠进程。

recalc_task_prio()
: 更新进程的动态优先级。

schedule()
: 选择要被执行的新进程。

load_balance()
: 维持多处理器系统中运行队列的平衡。

scheduler_tick()函数

我们已经在第六章“更新本地CPU统计数”一节中说明：每次时钟节拍到来时，scheduler_tick()是如何被调用以执行与调度相关的操作的。它执行的主要步骤如下：

1. 把转换为纳秒的TSC的当前值存入本地运行队列的timestamp_last_tick字段。这个时间戳是从函数sched_clock()(见前一节）获得的。
2. 检查当前进程是否是本地CPU的*swapper*进程，如果是，执行下面的子步骤：
   a. 如果本地运行队列除了*swapper*进程外，还包括另外一个可运行的进程，就设置当前进程的TIF_NEED_RESCHED字段，以强迫进行重新调度。就像我们在本章稍后“schedule()函数一节”将要看到的，如果内核支持超线程技术（见本章稍后“多处理器系统中运行队列的平衡”一节)，那么，只要一个逻辑CPU

运行队列中的所有进程都有比另一个逻辑CPU（两个逻辑CPU对应同一个物理CPU）上已经在执行的进程有低得多的优先级，前一个逻辑CPU就可能空闲，即使它的运行队列中有可运行的进程。

b. 跳转到第7步（没必要更新*swapper*进程的时间片计数器）。

3. 检查current->array是否指向本地运行队列的活动链表。如果不是，说明进程已经过期但还没有被替换：设置TIF_NEED_RESCHED标志，以强制进行重新调度并跳转到第7步。

4. 获得this_rq()->lock自旋锁。

5. 递减当前进程的时间片计数器，并检查是否已经用完时间片。由于进程的调度类型不同，函数所执行的这一步操作也有很大的差别，我们马上会讨论它们。

6. 释放this_rq()->lock自旋锁。

7. 调用rebalance_tick()函数，该函数应该保证不同CPU的运行队列包含数量基本相同的可运行进程。稍后在“多处理器系统中运行队列的平衡”一节我们将讨论运行队列的平衡。

更新实时进程的时间片

如果当前进程是先进先出（FIFO）的实时进程，函数scheduler_tick()什么都不做。实际上在这种情况下，current所表示的进程（当前进程）不可能被比其优先级低或其优先级相等的进程所抢占，因此，维持当前进程的最新时间片计数器是没有意义的。

如果current表示基于时间片轮转的实时进程，scheduler_tick()就递减它的时间片计数器并检查时间片是否被用完：

```
if (current->policy == SCHED_RR && !--current->time_slice) {
    current->time_slice = task_timeslice(current);
    current->first_time_slice = 0;
    set_tsk_need_resched(current);
    list_del(&current->run_list);
    list_add_tail(&current->run_list,
                  this_rq()->active->queue+current->prio);
}
```

如果函数确定时间片确实用完了，就执行一系列操作以达到抢占当前进程的目的，如果必要的话，就尽快抢占。

第一步操作包括调用task_timeslice()来重填进程的时间片计数器。该函数检查进程的静态优先级，并根据在前面“普通进程的调度”一节中列出的公式（1）返回相应的基本时间片。此外，current的first_time_slice字段被清0：该标志被fork()系统调用服务例程中的copy_process()设置，并在进程的第一个时间片刚一用完时立刻清0。

第二步，scheduler_tick()调用函数set_tsk_need_resched()设置进程的TIF_NEED_RESCHED标志。就像第四章“从中断和异常返回”一节中所描述的，该标志强制调用schedule()函数，以便current指向的进程能被另外一个有相同优先级（或更高优先级）的实时进程（如果有这种进程的话）所取代。

scheduler_tick()的最后一步操作包括把进程描述符移到与当前进程优先级相应的运行队列活动链表的尾部。把current指向的进程放到链表的尾部，可以保证在每个优先级与它相同的可运行实时进程获得CPU时间片以前，它不会再次被选择来执行。这是基于时间片轮转的调度策略。进程描述符的移动通过两个步骤完成：先调用list_del()把进程从运行队列的活动链表中删除，然后调用list_add_tail()把进程重新插入到同一个活动链表的尾部。

更新普通进程的时间片

如果当前进程是普通进程，函数scheduler_tick()执行下列操作：

1. 递减时间片计数器(current->time_slice)。
2. 检查时间片计数器。如果时间片用完，函数执行下列操作：
   a. 调用dequeue_task()从可运行进程的this_rq()->active集合中删除current指向的进程。
   b. 调用set_tsk_need_resched()设置TIF_NEED_RESCHED标志。
   c. 更新current指向的进程的动态优先级：

```
current->prio = effective_prio(current);
```

   函数effective_prio()读current的static_prio和sleep_avg字段，并根据在前面“普通进程的调度”一节中的公式（2）计算进程的动态优先级。

   d. 重填进程的时间片：

```
current->time_slice = task_timeslice(current);
current->first_time_slice = 0;
```

   e. 如果本地运行队列数据结构的expired_timestamp字段等于0（即过期进程集合为空），就把当前时钟节拍的值赋给expired_timestamp：

```
if (!this_rq()->expired_timestamp)
   this_rq()->expired_timestamp = jiffies;
```

   f. 把当前进程插入活动进程集合或过期进程集合：

```
if (!TASK_INTERACTIVE(current) || EXPIRED_STARVING(this_rq()) {
    enqueue_task(current, this_rq()->expired);
```

```
        if (current->static_prio < this_rq()->best_expired_prio)
            this_rq()->best_expired_prio = current->static_prio;
    } else
        enqueue_task(current, this_rq()->active);
```

如果用前面“普通进程的调度”一节列出的公式（3）识别出进程是一个交互式进程，TASK_INTERACTIVE宏就产生值1。宏EXPIRED_STARVING检查运行队列中的第一个过期进程的等待时间是否已经超过1000个时钟节拍乘以运行队列中的可运行进程数加1，如果是，宏产生值1。如果当前进程的静态优先级大于一个过期进程的静态优先级，EXPIRED_STARVING宏也产生值1。

3. 否则，如果时间片没有用完（current->time_slice不等于0），检查当前进程的剩余时间片是否太长：

```
if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
        p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
        (p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
        (p->array == rq->active)) {
    list_del(&current->run_list);
    list_add_tail(&current->run_list,
                this_rq()->active->queue+current->prio);
    set_tsk_need_resched(p);
}
```

宏TIMESLICE_GRANULARITY产生两个数的乘积给当前进程的bonus（见本章前面的表7-3），其中一个数为系统中CPU的数量，另一个为成比例的常量。基本上，具有高静态优先级的交互式进程，其时间片被分成大小为TIMESLICE_GRANULARITY的几个片段，以使这些进程不会独占CPU。

try_to_wake_up()函数

try_to_wake_up()函数通过把进程状态设置为TASK_RUNNING，并把该进程插入本地CPU的运行队列来唤醒睡眠或停止的进程。例如，调用该函数唤醒等待队列中的进程（见第三章“如何组织进程”一节）或恢复执行等待信号的进程（见第十一章）。该函数接受的参数有：

- 被唤醒进程的描述符指针（p）
- 可以被唤醒的进程状态掩码（state）
- 一个标志(sync)，用来禁止被唤醒的进程抢占本地CPU上正在运行的进程

该函数执行下列操作：

1. 调用函数task_rq_lock()禁用本地中断，并获得最后执行进程的CPU（它可能不

同于本地CPU)所拥有的运行队列rq的锁。CPU的逻辑号存储在p->thread_info->cpu字段。

2. 检查进程的状态p->state是否属于被当作参数传递给函数的状态掩码state；如果不是，就跳转到第9步终止函数。

3. 如果p->array字段不等于NULL，那么进程已经属于某个运行队列，因此跳转到第8步。

4. 在多处理器系统中，该函数检查要被唤醒的进程是否应该从最近运行的CPU的运行队列迁移到另外一个CPU的运行队列。实际上，函数就是根据一些启发式规则选择一个目标运行队列。例如：

   - 如果系统中某些CPU空闲，就选择空闲CPU的运行队列作为目标。按照优先选择先前正在执行进程的CPU和本地CPU这种顺序来进行。
   - 如果先前执行进程的CPU的工作量远小于本地CPU的工作量，就选择先前的运行队列作为目标。
   - 如果进程最近被执行过，就选择老的运行队列作为目标(可能仍然用这个进程的数据填充硬件高速缓存)。
   - 如果把进程移到本地CPU以缓解CPU之间的不平衡，目标就是本地运行队列(见本章稍后“多处理器系统中运行队列的平衡”一节)。

   执行完这一步，函数已经确定了目标CPU和对应的目标运行队列rq，前者将执行被唤醒的进程，后者就是进程插入的队列。

5. 如果进程处于TASK_UNINTERRUPTIBLE状态，函数递减目标运行队列的nr_uninterruptible字段，并把进程描述符的p->activated字段设置为-1。参见后面的“recalc_task_prio()函数”一节对activated字段的说明。

6. 调用activate_task()函数，它依次执行下面的子步骤：

   a. 调用sched_clock()获取以纳秒为单位的当前时间戳。如果目标CPU不是本地CPU，就要补偿本地时钟中断的偏差，这是通过使用本地CPU和目标CPU上最近一次发生时钟中断的相对时间戳来达到的。

   ```
   now = (sched_clock() - this_rq()->timestamp_last_tick)
         + rq->timestamp_last_tick;
   ```

   b. 调用recalc_task_prio()，把进程描述符的指针和上一步计算出的时间戳传递给它。下一节将详细说明recalc_task_prio()函数。

   c. 根据本章稍后的表7-6设置p->activated字段的值。

   d. 使用在第6a步中计算的时间戳设置p->timestamp字段。

e. 把进程描述符插入活动进程集合：

```
enqueue_task(p, rq->active);
rq->nr_running++;
```

7. 如果目标CPU不是本地CPU，或者没有设置sync标志，就检查可运行的新进程的动态优先级是否比rq运行队列中当前进程的动态优先级高(p->prio < rq->curr->prio)；如果是，就调用resched_task()抢占rq->curr。在单处理器系统中，后面的函数只是执行set_tsk_need_resched()来设置rq->curr进程的TIF_NEED_RESCHED标志。在多处理器系统中，resched_task()也检查TIF_NEED_RESCHED的旧值是否为0、目标CPU与本地CPU是否不同、rq->curr进程的TIF_POLLING_NRFLAG标志是否清0(目标CPU没有轮询进程TIF_NEED_RESCHED标志的值)。如果是，resched_task()调用smp_send_reschedule()产生IPI，并强制目标CPU重新调度（参见第4章“处理器间中断处理一节)。

8. 把进程的p->state字段设置为TASK_RUNNING状态。

9. 调用task_rq_unlock()来打开rq运行队列的锁并打开本地中断。

10. 返回1（如果成功唤醒进程）或0（如果进程没有被唤醒)。

recalc_task_prio()函数

函数recalc_task_prio()更新进程的平均睡眠时间和动态优先级。它接收进程描述符的指针p和由函数sched_clock()计算出的当前时间戳now作为参数。

该函数执行下述操作：

1. 把min(now -p -> timestamp, 10^9)的结果赋给局部变量sleep_time。

   p->timestamp字段包含导致进程进入睡眠状态的进程切换的时间戳，因此，sleep_time中存放的是从进程最后一次执行开始，进程消耗在睡眠状态的纳秒数（如果进程睡眠的时间更长，sleep_time就等于1s)。

2. 如果sleep_time不大于0，就不用更新进程的平均睡眠时间，直接跳转到第8步。

3. 检查进程是否不是内核线程、进程是否从TASK_UNINTERRUPTIBLE状态（p->activated字段等于-1，见前一节的第5步）被唤醒、进程连续睡眠的时间是否超过给定的睡眠时间极限。如果这三个条件都满足，函数把p->sleep_avg字段设置为相当于900个时钟节拍的值(用最大平均睡眠时间减去一个标准进程的基本时间片长度获得的一个经验值)。然后，跳转到第8步。

   睡眠时间极限依赖于进程的静态优先级，表7-2说明了它的一些典型值。简而言之，这个经验规则的目的是保证已经在不可中断模式上（通常是等待磁盘I/O的操作）

睡眠了很长时间的进程获得一个预先确定而且足够长的平均睡眠时间，以使这些进程即能尽快获得服务，又不会因睡眠时间太长而引起其他进程的饥饿。

4. 执行CURRENT_BONUS宏计算进程原来的平均睡眠时间的*bonus*值（见表7-3）。如果(10- *bonus*)大于0，函数用这个值与sleep_time相乘。因为将要把sleep_time加到进程的平均睡眠时间上（见下面的第6步），所以当前平均睡眠时间越短，它增加的就越快。

5. 如果进程处于TASK_UNINTERRUPTIBLE状态而且不是内核线程，执行下述子步骤：

   a. 检查平均睡眠时间p->sleep_avg是否大于或等于进程的睡眠时间极限（见本章前面的表7-2）。如果是，把局部变量sleep_time重新置为0，因此不用调整平均睡眠时间，而直接跳转到第6步。

   b. 如果sleep_time + p->sleep_avg的和大于或等于睡眠时间极限，就把p->sleep_avg字段置为睡眠时间极限并把sleep_time设置为0。

   通过对进程平均睡眠时间的轻微限制，函数不会对睡眠时间很长的批处理进程给予过多的奖赏。

6. 把sleep_time加到进程的平均睡眠时间上(p->sleep_avg)。

7. 检查p->sleep_avg是否超过1000个时钟节拍（以纳秒为单位），如果是，函数就把它减到1000个时钟节拍（以纳秒为单位）。

8. 更新进程的动态优先级：

```
p->prio = effective_prio(p);
```

   函数effective_prio()已经在本章前面“scheduler_tick()函数”一节讨论过。

schedule()函数

函数schedule()实现调度程序。它的任务是从运行队列的链表中找到一个进程，并随后将CPU分配给这个进程。schedule()可以由几个内核控制路径调用，可以采取直接调用或延迟（lazy）调用（可延迟的）的方式。

直接调用

如果current进程因不能获得必需的资源而要立刻被阻塞，就直接调用调度程序。在这种情况下，要阻塞进程的内核路径按下述步骤执行：

1. 把current进程插入适当的等待队列。

2. 把 current 进程的状态改为 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE。
3. 调用 schedule()。
4. 检查资源是否可用，如果不可用就转到第 2 步。
5. 一但资源可用，就从等待队列中删除 current 进程。

内核例程反复检查进程需要的资源是否可用，如果不可用，就调用 schedule()把 CPU 分配给其他进程。稍后，当调度程序再次允许把CPU分配给这个进程时，要重新检查资源的可用性。这些步骤与 wait_event()所执行的步骤很相似，也与第 3 章“如何组织进程”一节中描述的函数很相似。

许多执行长迭代任务的设备驱动程序也直接调用调度程序。每次迭代循环时，驱动程序都检查 TIF_NEED_RESCHED 标志，如果需要就调用 schedule()自动放弃 CPU。

延迟调用

也可以把 current 进程的 TIF_NEED_RESCHED 标志设置为 1，而以延迟方式调用调度程序。由于总是在恢复用户态进程的执行之前检查这个标志的值（见第四章 从“中断和异常返回”一节），所以 schedule()将在不久之后的某个时间被明确地调用。

以下是延迟调用调度程序的典型例子：

- 当current进程用完了它的CPU时间片时，由scheduler_tick()函数完成schedule()的延迟调用。
- 当一个被唤醒进程的优先级比当前进程的优先级高时，由try_to_wake_up()函数完成 schedule()的延迟调用。
- 当发出系统调用 sched_setscheduler()时（见本章稍后“与调度相关的系统调用”一节）。

进程切换之前 schedule()所执行的操作

schedule()函数的任务之一是用另外一个进程来替换当前正在执行的进程。因此，该函数的关键结果是设置一个叫做 next 的变量，使它指向被选中的进程，该进程将取代 current进程。如果系统中没有优先级高于current进程的可运行进程，那么最终 next 与 current 相等，不发生任何进程切换。

schedule()函数在一开始先禁用内核抢占，并初始化一些局部变量：

```
need_resched:
preempt_disable();
```

```
prev = current;
rq = this_rq();
```

正如你所见，把current返回的指针赋给prev，并把与本地CPU相对应的运行队列数据结构的地址赋给rq。

下一步，schedule()要保证prev不占用大内核锁（参见第五章“大内核锁”一节）：

```
if (prev->lock_depth >= 0)
    up(&kernel_sem);
```

注意，schedule()不改变lock_depth字段的值；当prev恢复执行的时候，如果该字段的值不为负数，则prev重新获得kernel_flag自旋锁。因此，通过进程切换会自动释放和重新获取大内核锁。

调用sched_clock()函数以读取TSC，并将它的值转换成纳秒，所获得的时间戳存放在局部变量now中。然后，schedule()计算prev所用的CPU时间片长度：

```
now = sched_clock();
run_time = now - prev->timestamp;
if (run_time > 1000000000)
    run_time = 1000000000;
```

通常使用限制在1s（要转换成纳秒）的时间。run_time的值用来限制进程对CPU的使用。不过，鼓励进程有较长的平均睡眠时间：

```
run_time /= (CURRENT_BONUS(prev) ? : 1);
```

记住，CURRENT_BONUS返回0～10之间的值，它与进程的平均睡眠时间是成比例的。

在开始寻找可运行进程之前，schedule()必须关掉本地中断，并获得所要保护的运行队列的自旋锁：

```
spin_lock_irq(&rq->lock);
```

正如在第三章“进程终止”一节中所描述的，prev可能是一个正在被终止的进程。为了确认这个事实，schedule()检查PF_DEAD标志：

```
if (prev->flags & PF_DEAD)
    prev->state = EXIT_DEAD;
```

接下来，schedule()检查prev的状态。如果不是可运行状态，而且它没有在内核态被抢占（见第四章“从中断和异常返回”一节），就应该从运行队列删除prev进程。不过，如果它是非阻塞挂起信号，而且状态为TASK_INTERRUPTIBLE，函数就把该进程的状态设置为TASK_RUNNING，并将它插入运行队列。这个操作与把处理器分配给prev是不同的，它只是给prev一次被选中执行的机会。

```
if (prev->state != TASK_RUNNING &&
    !(preempt_count() & PREEMPT_ACTIVE)) {
    if (prev->state == TASK_INTERRUPTIBLE && signal_pending(prev))
        prev->state = TASK_RUNNING;
    else {
        if (prev->state == TASK_UNINTERRUPTIBLE)
            rq->nr_uninterruptible++;
        deactivate_task(prev, rq);
    }
}
```

函数 deactivate_task()从运行队列中删除该进程：

```
rq->nr_running--;
dequeue_task(p, p->array);
p->array = NULL;
```

现在，schedule()检查运行队列中剩余的可运行进程数。如果有可运行的进程，schedule()就调用 dependent_sleeper()函数，在绝大多数情况下，该函数立即返回0。但是，如果内核支持超线程技术（见本章稍后“多处理器系统中运行队列的平衡”一节），函数检查要被选中执行的进程，其优先级是否比已经在相同物理CPU的某个逻辑CPU上运行的兄弟进程的优先级低；在这种特殊的情况下，schedule()拒绝选择低优先级的进程，而去执行 *swapper* 进程。

```
if (rq->nr_running) {
    if (dependent_sleeper(smp_processor_id(), rq)) {
        next = rq->idle;
        goto switch_tasks;
    }
}
```

如果运行队列中没有可运行的进程存在，函数就调用 idle_balance()，从另外一个运行队列迁移一些可运行进程到本地运行队列中。idle_balance()与load_balance()类似，在稍后的“load_balance()函数”一节中将对它进行说明。

```
if (!rq->nr_running) {
    idle_balance(smp_processor_id(), rq);
    if (!rq->nr_running) {
        next = rq->idle;
        rq->expired_timestamp = 0;
        wake_sleeping_dependent(smp_processor_id(), rq);
        if (!rq->nr_running)
            goto switch_tasks;
    }
}
```

如果 idle_balance()没有成功地把进程迁移到本地运行队列中，schedule()就调用 wake_sleeping_dependent()重新调度空闲CPU（即每个运行 *swapper* 进程的CPU）中的可运行进程。就像前面讨论 dependent_sleeper()函数时所说明的，通常在内核支

持超线程技术的时候可能会出现这种情况。然而，在单处理器系统中，或者当把进程迁移到本地运行队列的种种努力都失败的情况下，函数就选择*swapper*进程作为next进程并继续进行下一步骤。

我们假设schedule()函数已经肯定运行队列中有一些可运行的进程，现在它必须检查这些可运行进程中是否至少有一个进程是活动的。如果没有，函数就交换运行队列数据结构的active和expired字段的内容。因此，所有的过期进程变为活动进程，而空集合准备接纳将要过期的进程。

```
array = rq->active;
if (!array->nr_active) {
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;
    rq->best_expired_prio = 140;
}
```

现在可以在活动的prio_array_t数据结构中搜索一个可运行进程了（参见第三章“标识一个进程”一节）。首先，schedule()搜索活动进程集合位掩码的第一个非0位。回忆一下，当对应的优先级链表不为空时，就把位掩码的相应位置1。因此，第一个非0位的下标对应包含最佳运行进程的链表。随后，返回该链表的第一个进程描述符：

```
idx = sched_find_first_bit(array->bitmap);
next = list_entry(array->queue[idx].next, task_t, run_list);
```

函数sched_find_first_bit()是基于bsfl汇编语言指令的，它返回32位字中被设置为1的最低位的位下标。

局部变量next现在存放将取代prev的进程描述符指针。schedule()函数检查next->activated字段，该字段的编码值表示进程在被唤醒时的状态，如表7-6所示。

表7-6：进程描述符中activated字段的含义

值	说明
0	进程处于TASK_RUNNING状态
1	进程处于TASK_INTERRUPTIBLE或TASK_STOPPED状态，而且正在被系统调用服务例程或内核线程唤醒
2	进程处于TASK_INTERRUPTIBLE或TASK_STOPPED状态，而且正在被中断处理程序或可延迟函数唤醒
-1	进程处于TASK_UNINTERRUPTIBLE状态而且正在被唤醒

如果next是一个普通进程，而且它正在从TASK_INTERRUPTIBLE或TASK_STOPPED状态被唤醒，调度程序就把自从进程插入运行队列开始所经过的纳秒数加到进程的平均睡眠时间中。换言之，进程的睡眠时间被增加了，以包含进程在运行队列中等待CPU所消耗的时间。

```
if (next->prio >= 100 && next->activated > 0) {
    unsigned long long delta = now - next->timestamp;
    if (next->activated == 1)
        delta = (delta * 38) / 128;
    array = next->array;
    dequeue_task(next, array);
    recalc_task_prio(next, next->timestamp + delta);
    enqueue_task(next, array);
}
next->activated=0;
```

要说明的是，调度程序把被中断处理程序和可延迟函数所唤醒的进程与被系统调用服务例程和内核线程所唤醒的进程区分开来。在前一种情况下，调度程序增加全部运行队列等待时间，而在后一种情况下，它只增加等待时间的部分。这是因为交互式进程更可能被异步事件（考虑用户在键盘上的按键操作）而不是同步事件唤醒。

schedule()完成进程切换时所执行的操作

现在schedule()函数已经要让next进程投入运行。内核将立刻访问next进程的thread_info数据结构，其地址存放在next进程描述符的接近顶部的位置。

```
switch_tasks:

prefetch(next);
```

prefetch宏提示CPU控制单元把next的进程描述符第一部分字段的内容装入硬件高速缓存。正是这一点改善了schedule()的性能，因为对于后续指令的执行（不影响next），数据是并行移动的。

在替代prev之前，调度程序应该完成一些管理的工作：

```
clear_tsk_need_resched(prev);
rcu_qsctr_inc(prev->thread_info->cpu);
```

以防（万一）以延迟方式调用schedule()，clear_tsk_need_resched()函数清除prev的TIF_NEED_RESCHED标志。然后，函数记录CPU正在经历静止状态［参见第五章“读－拷贝－更新（RCU）”一节］。

schedule()函数还必须减少prev的平均睡眠时间，并把它补充给进程所使用的CPU时间片：

```
prev->sleep_avg -= run_time;
if ((long)prev->sleep_avg <= 0)
    prev->sleep_avg = 0;
prev->timestamp = prev->last_ran = now;
```

随后更新进程的时间戳。

prev和next很可能是同一个进程：如果在当前运行队列中没有优先级较高或相等的其他活动进程时，会发生这种情况。在这种情况下，函数不做进程切换：

```
if (prev == next) {
    spin_unlock_irq(&rq->lock);
    goto finish_schedule;
}
```

这里，prev和next是不同的进程，进程切换确实发生了：

```
next->timestamp = now;
rq->nr_switches++;
rq->curr = next;
prev = context_switch(rq, prev, next);
```

context_switch()函数建立next的地址空间。正如我们将在第九章“内核线程的内存描述符”中将要看到的，进程描述符的active_mm字段指向进程所使用的内存描述符，而mm字段指向进程所拥有的内存描述符。对于一般的进程，这两个字段有相同的地址，但是，内核线程没有它自己的地址空间，而且它的mm字段总是被设置为NULL。context_switch()函数确保，如果next是一个内核线程，它使用prev所使用的地址空间：

```
if (!next->mm) {
    next->active_mm = prev->active_mm;
    atomic_inc(&prev->active_mm->mm_count);
    enter_lazy_tlb(prev->active_mm, next);
}
```

一直到Linux 2.2版，内核线程都有自己的地址空间。那种设计选择不是最理想的，因为不管什么时候当调度程序选择一个新进程（即使是一个内核线程）运行时，都必须改变页表。因为内核线程都运行在内核态，它仅使用线性地址空间的第4个GB，其映射对系统的所有进程都是相同的。甚至最坏情况下，写cr3寄存器会使所有的TLB表项无效[参见第二章“转换后援缓冲器（TLB）”一节]，这将导致极大的性能损失。现在的Linux具有更高的效率，因为如果next是内核线程，就根本不触及页表。作为进一步的优化，如果next是内核线程，schedule()函数把进程设置为懒惰TLB模式[参见第二章“转换旁路缓冲器（TLB）”一节]。

相反，如果next是一个普通进程，context-switch函数用next的地址空间替换prev的地址空间：

```
if (next->mm)
    switch_mm(prev->active_mm, next->mm, next);
```

如果prev是内核线程或正在退出的进程，context_switch()函数就把指向prev内存描述符的指针保存到运行队列的prev_mm字段中，然后重新设置prev->active_mm：

```
if (!prev->mm) {
    rq->prev_mm = prev->active_mm;
    prev->active_mm = NULL;
}
```

现在，context_switch()终于可以调用switch_to()执行prev和next之间的进程切换了（见第三章“执行进程切换”一节）：

```
switch_to(prev, next, prev);
return prev;
```

进程切换后schedule()所执行的操作

schedule()函数中在switch_to宏调用之后紧接着的指令并不由next进程立即执行，而是稍后当调度程序又选择prev执行时由prev执行。然而，在那个时刻，prev局部变量并不指向我们开始描述schedule()时所替换出去的原来那个进程，而是指向prev被调度时由prev替换出的原来那个进程（如果你被搞糊涂了，请回到第三章阅读“执行进程切换”一节）。进程切换后的第一部分指令是：

```
barrier();
finish_task_switch(prev);
```

在schedule()中，紧接着context_switch()函数调用之后，宏barrier()产生一个代码优化屏障（见第五章“优化和内存屏障”一节）。然后，执行finish_task_switch()函数：

```
mm = this_rq()->prev_mm;
this_rq()->prev_mm = NULL;
prev_task_flags = prev->flags;
spin_unlock_irq(&this_rq()->lock);
if (mm)
    mmdrop(mm);
if (prev_task_flags & PF_DEAD)
    put_task_struct(prev);
```

如果prev是一个内核线程，那么运行队列的prev_mm字段存放借给prev的内存描述符的地址。正如我们在第九章将要看到的，mmdrop()减少内存描述符的使用计数器；如果该计数器等于0了（可能是因为prew是一个僵死进程），函数还要释放与页表相关的所有描述符和虚拟存储区。

finish_task_switch()函数还要释放运行队列的自旋锁并打开本地中断。然后，检查prev是否是一个正在从系统中被删除的僵死任务（见第三章“进程终止”一节）如果是，就调用put_task_struct()以释放进程描述符引用计数器，并撤消所有其余对该进程的引用（见第三章“进程删除”一节）。

schedule()函数的最后一部分指令是：

```
finish_schedule:

prev = current;
if (prev->lock_depth >= 0)
    __reacquire_kernel_lock();
preempt_enable_no_resched();
if (test_bit(TIF_NEED_RESCHED, &current_thread_info()->flags))
   goto need_resched;
return;
```

如你所见，schedule()在需要的时候重新获得大内核锁，重新启用内核抢占，并检查是否一些其他的进程已经设置了当前进程的TIF_NEED_RESCHED标志。如果是，则整个schedule()函数重新开始执行，否则，函数结束。

多处理器系统中运行队列的平衡

我们在第四章已经看到，Linux一直坚持采用对称多处理模型；这意味着，与其他CPU相比，内核不应该对一个CPU有任何偏向。但是，多处理器机器具有很多不同的风格，而且调度程序的实现随硬件特征的不同而有所不同。我们将特别关注下面3种不同类型的多处理器机器：

标准的多处理器体系结构

直到最近，这是多处理器机器最普通的体系结构。这些机器所共有的RAM芯片集被所有CPU共享。

超线程

超线程芯片是一个立刻执行几个执行线程的微处理器；它包括几个内部寄存器的拷贝，并快速在它们之间切换。这种由Intel发明的技术，使得当前线程在访问内存的间隙，处理器可以使用它的机器周期去执行另外一个线程。一个超线程的物理CPU可以被Linux看作几个不同的逻辑CPU。

NUMA

把CPU和RAM以本地“节点”为单位分组（通常一个节点包括一个CPU和几个RAM芯片）。内存仲裁器（一个使系统中的CPU以串型方式访问RAM的专用电路，见第二章“内存地址”一节）是典型的多处理器系统的性能瓶颈。在NUMA

体系结构中，当CPU访问与它同在一个节点中的“本地”RAM芯片时，几乎没有竞争，因此访问通常是非常快的。另一方面，访问其所属节点外的“远程”RAM芯片就非常慢。我们将在第八章“非一致内存访问（NUMA）”一节讨论Linux内核内存分配器是如何支持NUMA体系结构的。

这些基本的多处理器系统类型经常被组合使用。例如，内核把一个包括两个不同超线程CPU的主板看作四个逻辑CPU。

正如我们在上一节所看到的，schedule()函数从本地CPU的运行队列挑选新进程运行。因此，一个指定的CPU只能执行其相应的运行队列中的可运行进程。另外，一个可运行进程总是存放在某一个运行队列中：任何一个可运行进程都不可能同时出现在两个或多个运行队列中。因此，一个保持可运行状态的进程通常被限制在一个固定的CPU上。

这种设计通常对系统性能是有益的，因为，运行队列中的可运行进程所拥有的数据可能填满每个CPU的硬件高速缓存。但是在有些情况下，把可运行进程限制在一个指定的CPU上可能引起严重的性能损失。例如，考虑频繁使用CPU的大量批处理进程：如果它们绝大多数都在同一个运行队列中，那么系统中的一个CPU将会超负荷，而其他一些CPU几乎处于空闲状态。

因此，内核周期性地检查运行队列的工作量是否平衡，并在需要的时候，把一些进程从一个运行队列迁移到另一个运行队列。但是，为了从多处理器系统获得最佳性能，负载平衡算法应该考虑系统中CPU的拓扑结构。从内核2.6.7版本开始，Linux提出一种基于“调度域”概念的复杂的运行队列平衡算法。正是有了调度域这一概念，使得这种算法能够很容易适应各种已有的多处理器体系结构(甚至诸如那些基于新近出现的“多核”微处理器的体系结构)。

调度域

调度域（scheduling domain）实际上是一个CPU集合，它们的工作量应当由内核保持平衡。一般来说，调度域采取分层的组织形式：最上层的调度域（通常包括系统中的所有CPU）包括多个子调度域，每个子调度域包括一个CPU子集。正是调度域的这种分层结构，使工作量的平衡能以如下有效方式来实现。

每个调度域被依次划分成一个或多个组，每个组代表调度域的一个CPU子集。工作量的平衡总是在调度域的组之间来完成。换言之，只有在某调度域的某个组的总工作量远远低于同一个调度域的另一个组的工作量时，才把进程从一个CPU迁移到另一个CPU。

图7-2给出了3个调度域分层实例，对应3种主要的多处理器机器体系结构。