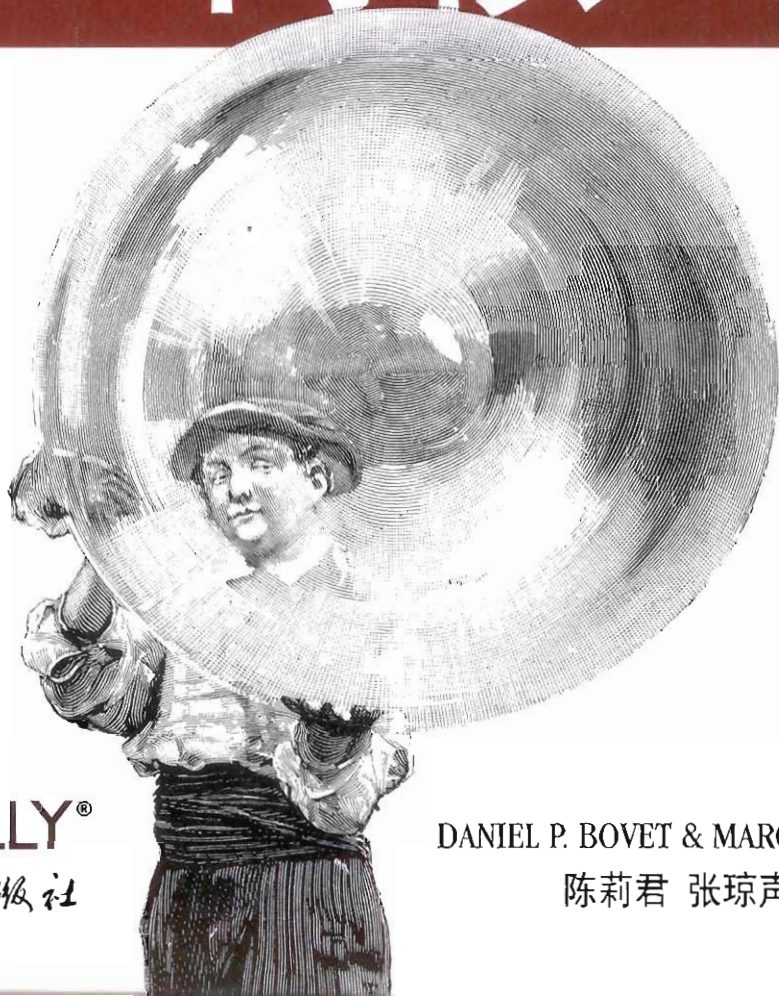


UNDERSTANDING THE LINUX KERNEL

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“覆压三百余里，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、由众人团结协力搭建起的殿堂。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、张琮声、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、十五章由张琮声翻译，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区别，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。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，帮助我们去掉了许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



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 Public 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 Torvalds 注册的商标。

注 2: GNU 项目是由自由软件基金会 (<http://www.gnu.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 interface, 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，Linux 2.6 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 SVR4 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 Linux 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 *Linux* 舍弃，如 STREAMS/I/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 *Linux* 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 Compaq 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/EM64T 64 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 Linux 2.6.11 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妨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。

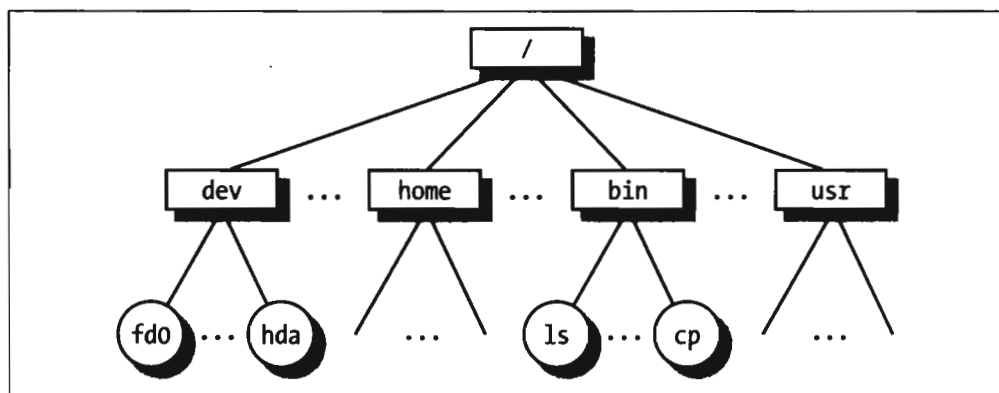


图 1-1: 目录树示例

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（*pathname*），路径名由斜杠及一系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之间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。

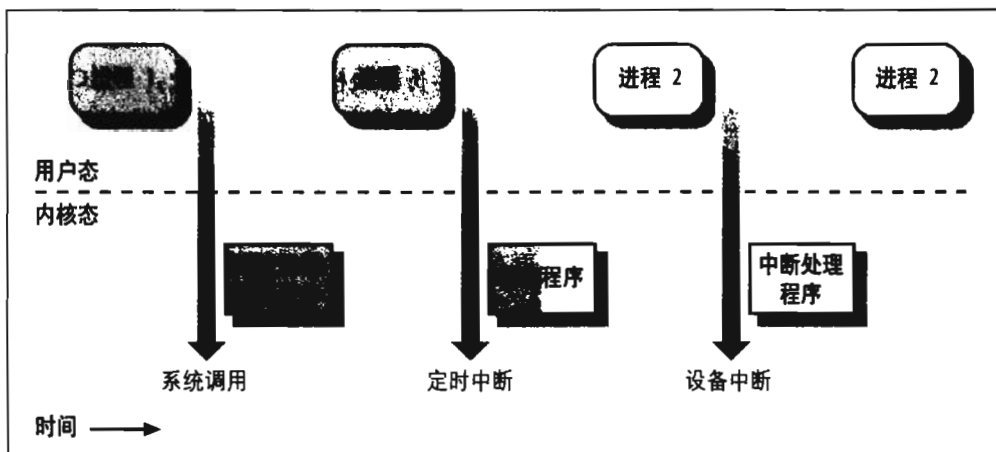


图 1-2：用户态与内核态之间的转换

- 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运行时，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）

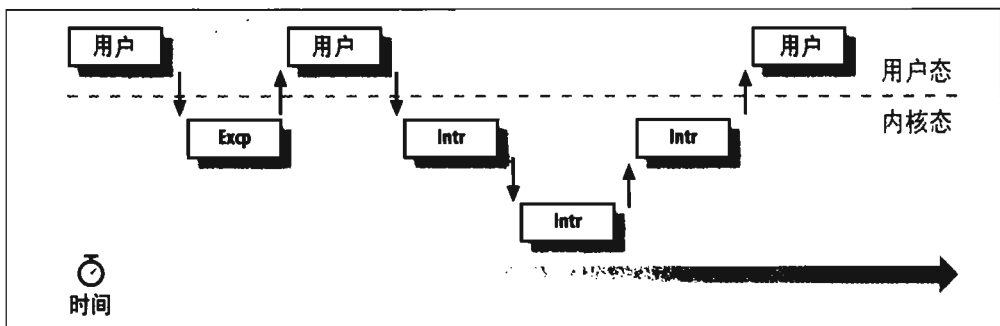


图 1-3：内核控制路径的交错执行

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 `SIGCHLD` 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 SIGTTOU 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毫无疑问地划分为两部分，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。

这一策略的前提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。

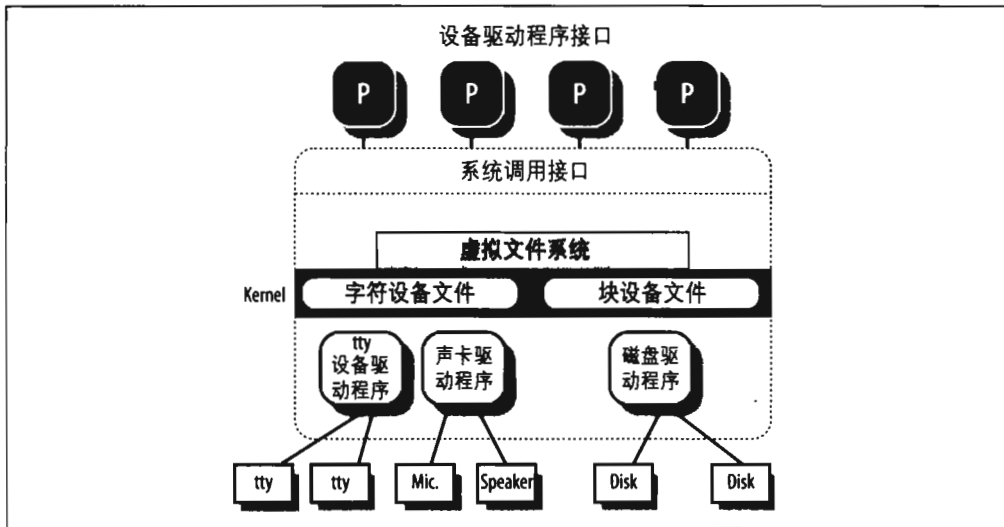


图 1-4：设备驱动程序接口

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)。

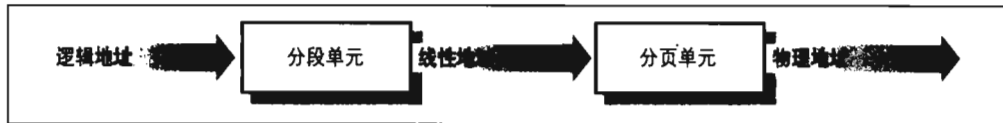


图 2-1: 逻辑地址转换

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。

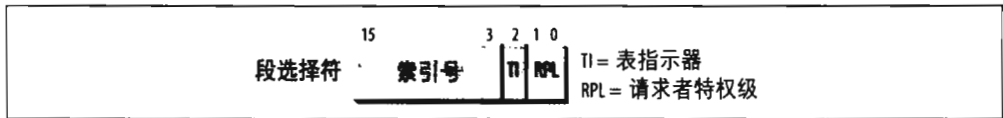


图 2-2：段描述符格式

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 (<i>Descriptor Privilege Level</i>) 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	<i>Segment-Present</i> 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, 取决于它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。

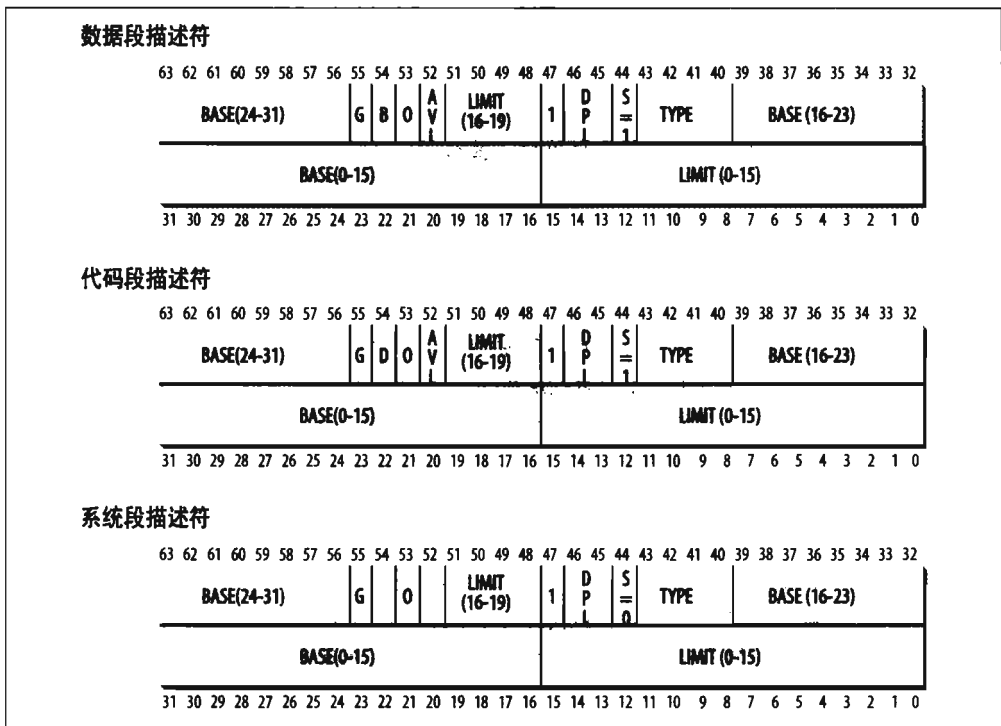


图 2-3：段描述符格式

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）。

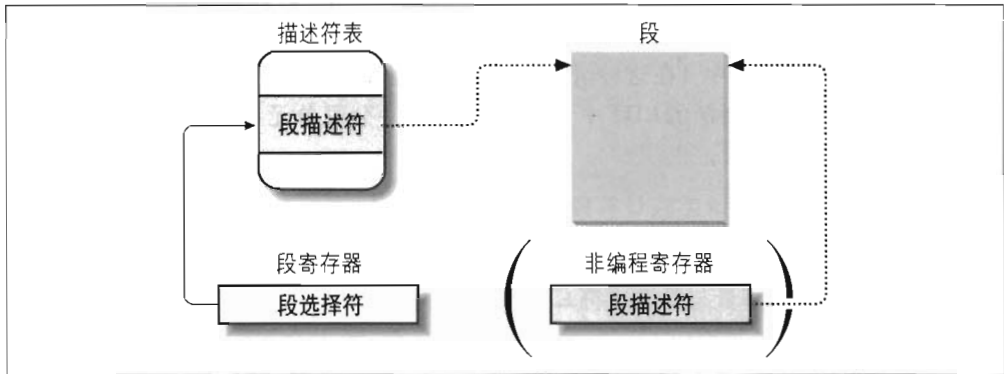


图 2-4：段选择符和段描述符

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 gdt_r 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。

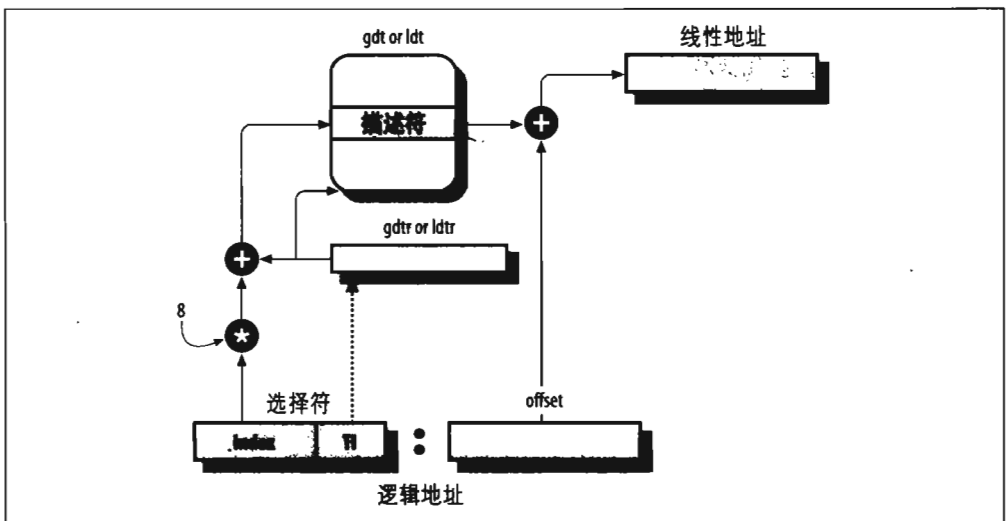


图 2-5: 逻辑地址的转换

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	0xffffffff	1	10	3	1	1
用户数据段	0x00000000	1	0xffffffff	1	2	3	1	1
内核代码段	0x00000000	1	0xffffffff	1	10	0	1	1
内核数据段	0x00000000	1	0xffffffff	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gdt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	Linux 全局描述符表	段选择符
null	0x0	TSS	0x80
reserved		LDT	0x88
reserved		PNPBIOS 32-bit code	0x90
reserved		PNPBIOS 16-bit code	0x98
not used		PNPBIOS 16-bit data	0xa0
not used		PNPBIOS 16-bit data	0xa8
TLS #1	0x33	PNPBIOS 16-bit data	0xb0
TLS #2	0x3b	APMBIOS 32-bit code	0xb8
TLS #3	0x43	APMBIOS 16-bit code	0xc0
reserved		APMBIOS data	0xc8
reserved		not used	
reserved		not used	
kernel code	0x60 (__KERNEL_CS)	not used	
kernel data	0x68 (__KERNEL_DS)	not used	
user code	0x73 (__USER_CS)	not used	
user data	0x7b (__USER_DS)	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 微处理器提供了一种机制，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。

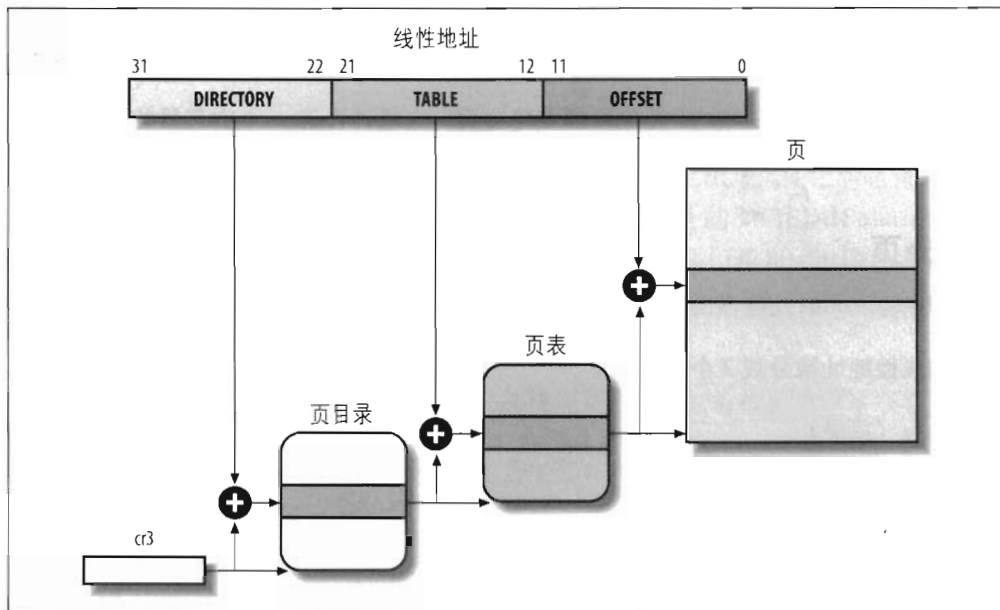


图 2-7：80x86 处理器的分页

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]。

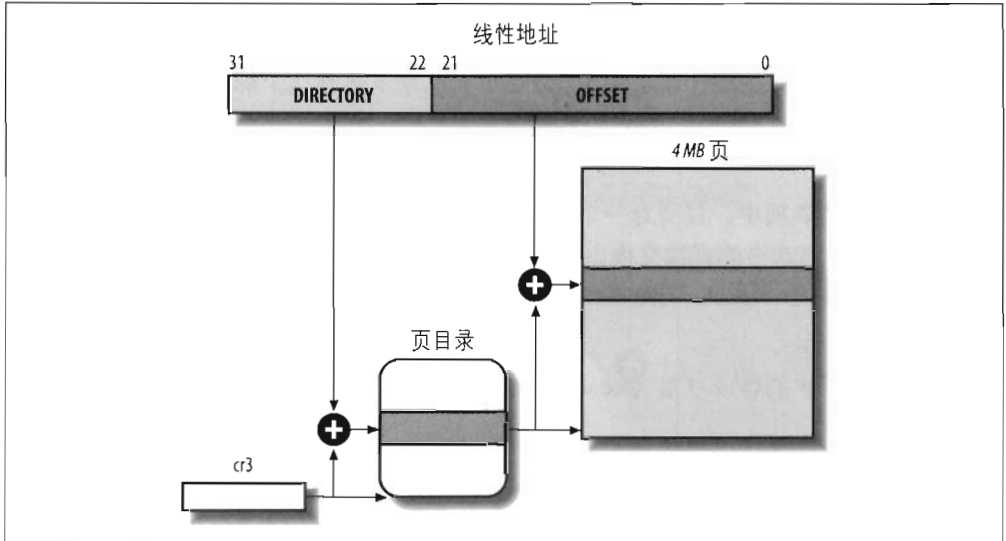


图 2-8：扩展分页

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，Offset 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。

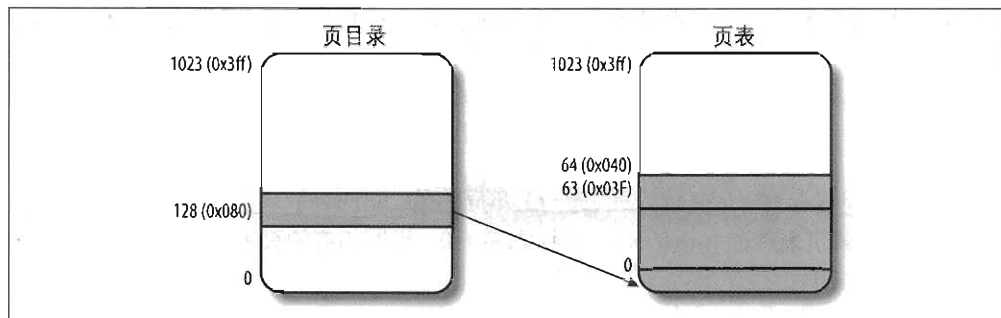


图 2-9: 分页的例子

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 $2^{36} = 64\text{GB}$ 。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。

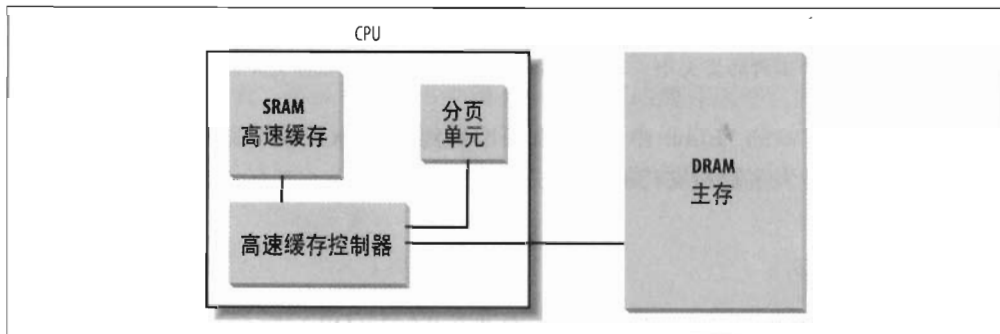


图 2-10: 处理器硬件高速缓存

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。

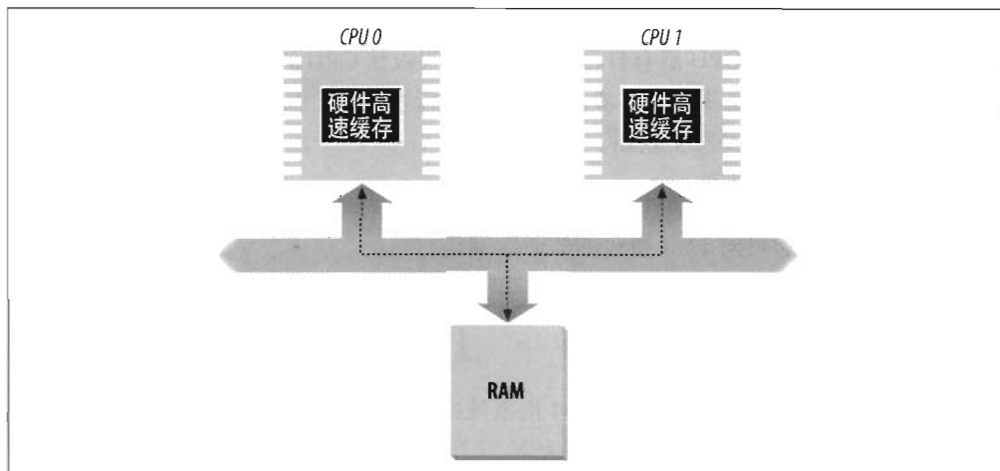


图 2-11：双处理器中的高速缓存

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 Disabl`t) 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）。

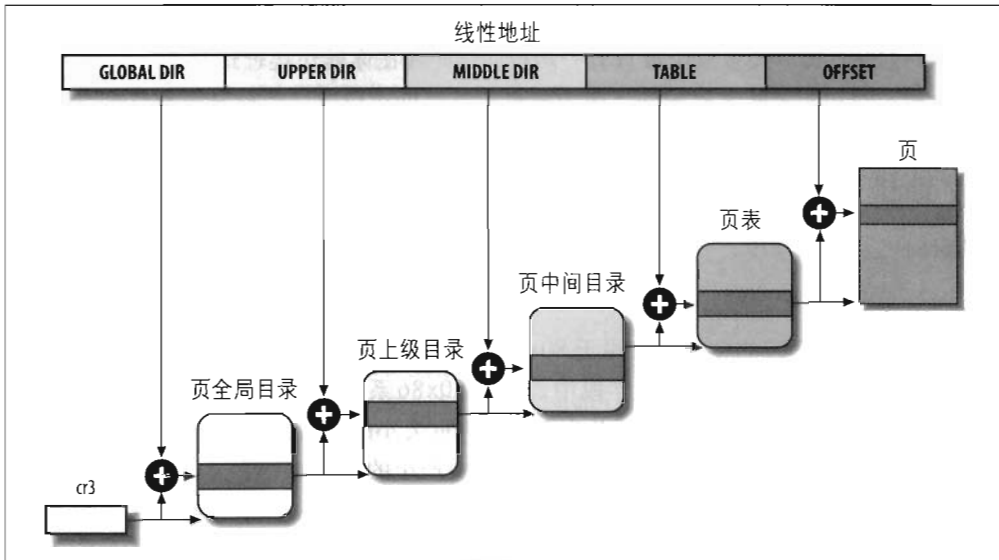


图 2-12: Linux 分页模式

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 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 ($2^{\text{PMD_SHIFT}}$)，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）执行相反的操作，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_pgdn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)。
- Accessed 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
<code>pte_user()</code>	读 User/Supervisor 标志
<code>pte_read()</code>	读 User/Supervisor 标志 (表示 80x86 处理器上的页不受读的保护)
<code>pte_write()</code>	读 Read/Write 标志
<code>pte_exec()</code>	读 User/Supervisor 标志 (80x86 处理器上的页不受代码执行的保护)
<code>pte_dirty()</code>	读 Dirty 标志
<code>pte_young()</code>	读 Accessed 标志
<code>pte_file()</code>	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
<code>mk_pte_huge()</code>	设置页表项中的 Page Size 和 Present 标志
<code>pte_wrprotect()</code>	清除 Read/Write 标志
<code>pte_rdpprotect()</code>	清除 User/Supervisor 标志
<code>pte_exprotect()</code>	清除 User/Supervisor 标志
<code>pte_mkwrite()</code>	设置 Read/Write 标志
<code>pte_mkread()</code>	设置 User/Supervisor 标志
<code>pte_mkexec()</code>	设置 User/Supervisor 标志
<code>pte_mkclean()</code>	清除 Dirty 标志
<code>pte_mkdirty()</code>	设置 Dirty 标志
<code>pte_mkold()</code>	清除 Accessed 标志 (把此页标记为未访问)
<code>pte_mkyoung()</code>	设置 Accessed 标志 (把此页标记为访问过)
<code>pte_modify(p, v)</code>	把页表项 p 的所有访问权限设置为指定的值 v
<code>ptep_set_wrprotect()</code>	与 <code>pte_wrprotect()</code> 类似, 但作用于指向页表项的指针
<code>ptep_set_access_flags()</code>	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 <code>flush_tlb_page()</code> 函数 [参见本章“转换后援缓冲器 (TLB)”一节]
<code>ptep_mkdirty()</code>	与 <code>pte_mkdirty()</code> 类似, 但作用于指向页表项的指针
<code>ptep_test_and_clear_dirty()</code>	与 <code>pte_mkclean()</code> 类似, 但作用于指向页表项的指针并返回 Dirty 标志的旧值
<code>ptep_test_and_clear_young()</code>	与 <code>pte_mkold()</code> 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
<code>pgd_index(addr)</code>	找到线性地址 <code>addr</code> 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
<code>pgd_offset(mm, addr)</code>	接收内存描述符地址 <code>mm</code> (参见第九章) 和线性地址 <code>addr</code> 作为参数。这个宏产生地址 <code>addr</code> 在页全局目录中相应表项的线性地址, 通过内存描述符 <code>mm</code> 内的一个指针可以找到这个页全局目录
<code>pgd_offset_k(addr)</code>	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 <code>addr</code> (参见稍后“内核页表”一节)
<code>pgd_page(pgd)</code>	通过页全局目录项 <code>pgd</code> 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 <code>pud_page()</code> , 后者应用于页上级目录项
<code>pud_offset(pgd, addr)</code>	接收指向页全局目录项的指针 <code>pgd</code> 和线性地址 <code>addr</code> 作为参数。这个宏产生页上级目录中目录项 <code>addr</code> 对应的线性地址。在两级或三级分页系统中, 该宏产生 <code>pgd</code> , 即一个页全局目录项的地址
<code>pud_page(pud)</code>	通过页上级目录项 <code>pud</code> 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 <code>pmd_page()</code> , 后者应用于页中间目录项
<code>pmd_index(addr)</code>	产生线性地址 <code>addr</code> 在页中间目录中所对应目录项的索引 (相对位置)
<code>pmd_offset(pud, addr)</code>	接收指向页上级目录项的指针 <code>pud</code> 和线性地址 <code>addr</code> 作为参数。这个宏产生目录项 <code>addr</code> 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 <code>pud</code> , 即页全局目录项的地址
<code>pmd_page(pmd)</code>	通过页中间目录项 <code>pmd</code> 产生相应页表的页描述符地址。在两级或三级分页系统中, <code>pmd</code> 实际上是页全局目录中的一项
<code>mk_pte(p, prot)</code>	接收页描述符地址 <code>p</code> 和一组存取权限 <code>prot</code> 作为参数, 并创建相应的页表项
<code>pte_index(addr)</code>	产生线性地址 <code>addr</code> 对应的表项在页表中的索引 (相对位置)
<code>pte_offset_kernel(dir, addr)</code>	线性地址 <code>addr</code> 在页中间目录 <code>dir</code> 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
<code>pte_offset_map(dir, addr)</code>	接收指向一个页中间目录项的指针 <code>dir</code> 和线性地址 <code>addr</code> 作为参数, 它产生与线性地址 <code>addr</code> 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 <code>pte_unmap</code> 对它进行释放。 <code>pte_offset_map_nested</code> 宏和 <code>pte_unmap_nested</code> 宏是相同的, 但它们使用不同的临时内核映射
<code>pte_page(x)</code>	返回页表项 <code>x</code> 所引用页的描述符地址
<code>pte_to_pgoff(pte)</code>	从一个页表项的 <code>pte</code> 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)
<code>pgoff_to_pte(offset)</code>	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
<code>pgd_alloc(mm)</code>	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 <code>mm</code> (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
<code>pgd_free(pgd)</code>	释放页全局目录中地址为 <code>pgd</code> 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
<code>pud_alloc(mm, pgd, addr)</code>	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 <code>pgd</code> 的线性地址
<code>pud_free(x)</code>	在两级或三级分页系统下, 这个宏什么也不做
<code>pmd_alloc(mm, pud, addr)</code>	定义这个函数以使普通三级分页系统可以为线性地址 <code>addr</code> 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 <code>pud</code> 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 <code>addr</code> 对应的页中间目录项的线性地址。参数 <code>mm</code> 被忽略
<code>pmd_free(x)</code>	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
<code>pte_alloc_map(mm, pmd, addr)</code>	接收页中间目录项的地址 <code>pmd</code> 和线性地址 <code>addr</code> 作为参数, 并返回与 <code>addr</code> 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 <code>pte_alloc_one()</code> 分配一个新页表。如果分配了一个新页表, <code>addr</code> 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 <code>pte_unmap</code> 对它进行释放
<code>pte_alloc_kernel(mm, pmd, addr)</code>	如果与地址 <code>addr</code> 相关的页中间目录项 <code>pmd</code> 为空, 该函数分配一个新页表。然后返回与 <code>addr</code> 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)
<code>pte_free(pte)</code>	释放与页描述符指针 <code>pte</code> 相关的页表
<code>pte_free_kernel(pte)</code>	等价于 <code>pte_free()</code> , 但由主内核页表使用
<code>clear_page_range(mmu, start, end)</code>	从线性地址 <code>start</code> 到 <code>end</code> 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。

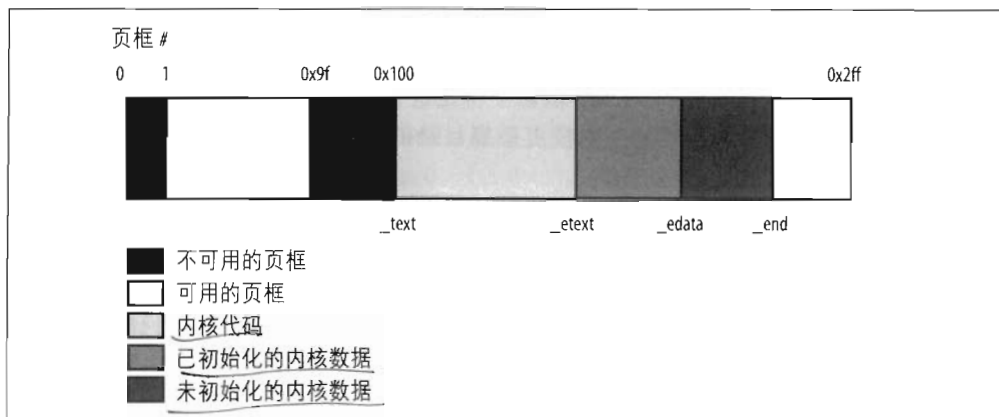


图 2-13: Linux 2.6 的前 768 个页框 (3MB)

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 0xc07ffffff 的线性地址映射到从 0x00000000 到 0x007ffffff 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 0xc07ffffff 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 项的地址字段设置为 `pg0` 的物理地址，而 1 项和 0x301 项的地址字段设置为紧随 `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，正如之前的情况一样。

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 `0xfffffc000` 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 $X - \text{PAGE_OFFSET}$)。

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
<code>flush_tlb_all</code>	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
<code>flush_tlb_kernel_range</code>	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
<code>flush_tlb</code>	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
<code>flush_tlb_mm</code>	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
<code>flush_tlb_range</code>	刷新指定进程的线性地址间隔对应的 TLB 表项	地址间隔时
<code>flush_tlb_pgtables</code>	刷新指定进程中特定的相邻页表集相关的 TLB 表项	释放进程的一些页表时
<code>flush_tlb_page</code>	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
<code>__flush_tlb()</code>	将 cr3 寄存器的当前值重新写回 cr3	<code>flush_tlb</code> , <code>flush_tlb_m</code> , <code>flush_tlb_range</code>
<code>__flush_tlb_global()</code>	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	<code>flush_tlb_all</code> , <code>flush_tlb_</code> <code>kernel_range</code>
<code>__flush_tlb_single(addr)</code>	以 <code>addr</code> 为参数执行 <code>invlpg</code> 汇编语言指令	<code>flush_tlb_page</code>

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，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正常进程之间执行进程切换时（参见第七章的“`schedule()` 函数”一节）。
- 当在一个正常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 上执行过的正常进程的页表集。

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 (*lightweight 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。

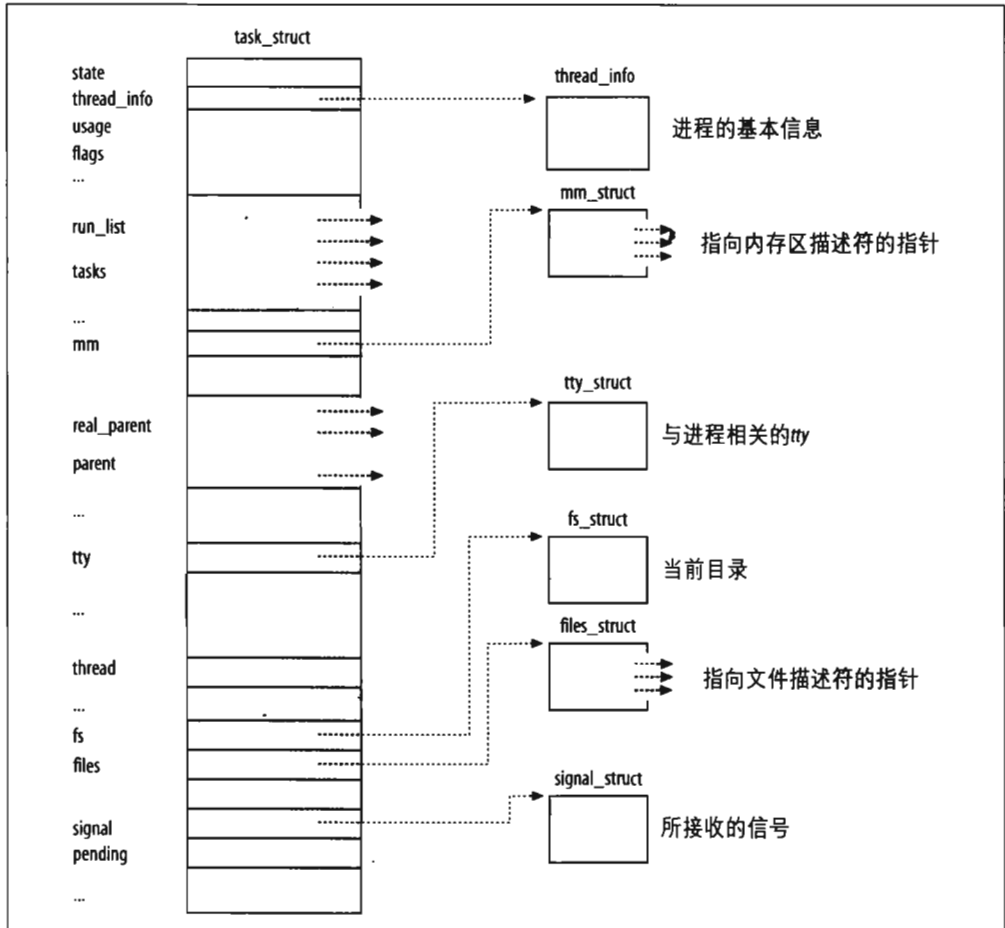


图 3-1: Linux 进程描述符

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。

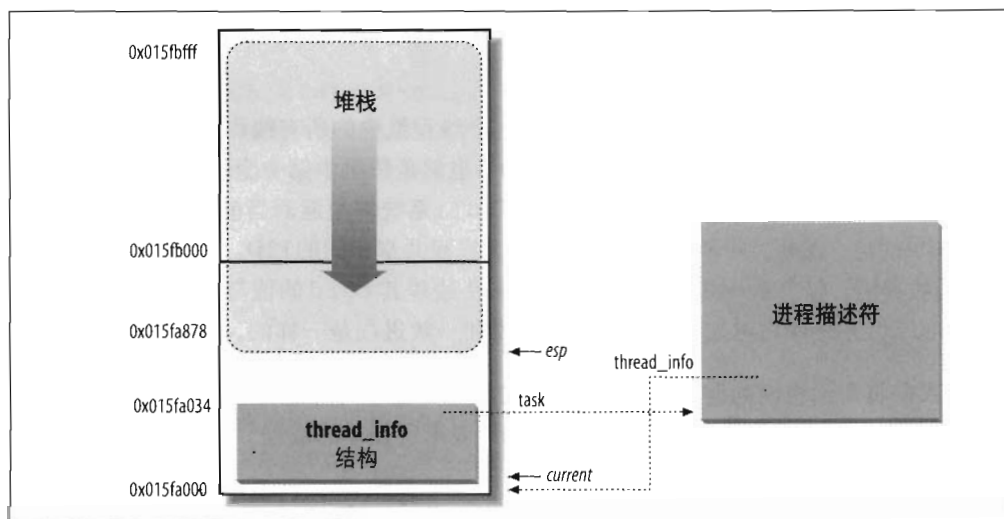


图 3-2: `thread_info` 结构和进程内核栈存放在两个连续的页框中

`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 4R 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_info`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_info`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 $0xffffe000, %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 $0xffffe000, %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)]。

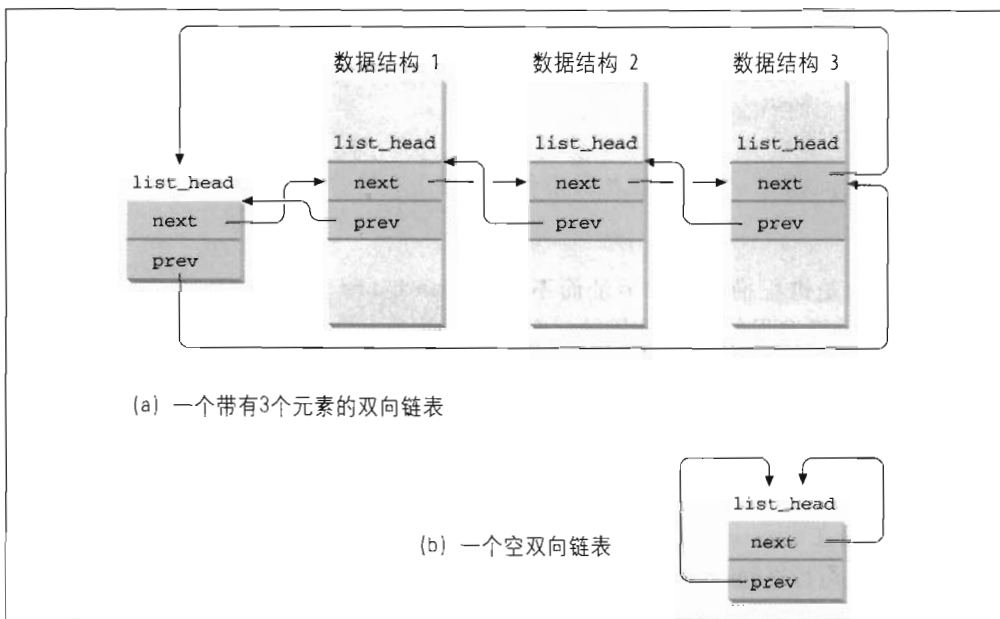


图 3-3: 用 `list_head` 数据结构构造的一个双向链表

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

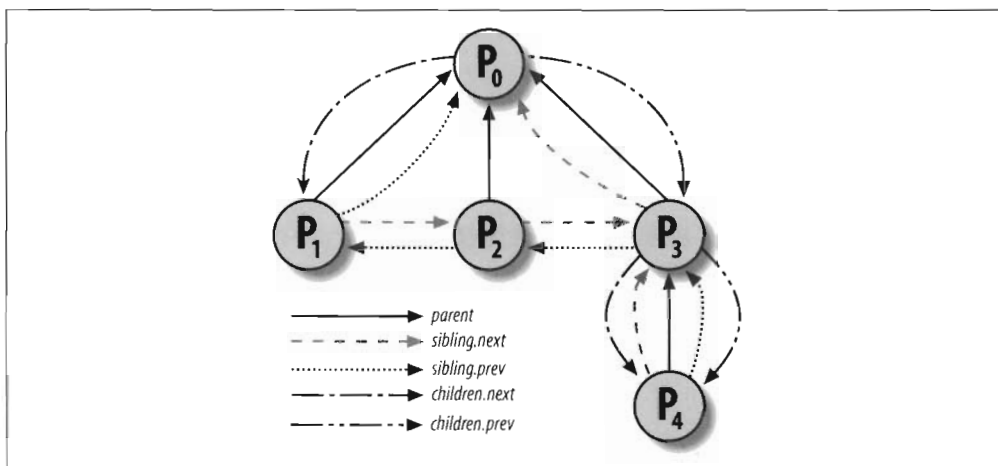


图 3-4: 五个进程间的亲属关系

表 3-4: 建立非亲属关系的进程描述符字段

字段名	说明
<code>group_leader</code>	P 所在进程组的领头进程的描述符指针
<code>signal->pgrp</code>	P 所在进程组的领头进程的 PID
<code>tgid</code>	P 所在线程组的领头进程的 PID
<code>signal->session</code>	P 的登录会话领头进程的 PID
<code>ptrace_children</code>	链表的头, 该链表包含所有被 debugger 程序跟踪的 P 的子进程
<code>ptrace_list</code>	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	<code>pid</code>	进程的 PID
PIDTYPE_TGID	<code>tgid</code>	线程组领头进程的 PID
PIDTYPE_PGID	<code>pgrp</code>	进程组领头进程的 PID
PIDTYPE_SID	<code>session</code>	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2890和29384的两个进程散列到这个表的第200个元素，而进程号（PID）为29385的进程散列到这个表的第1466个元素。

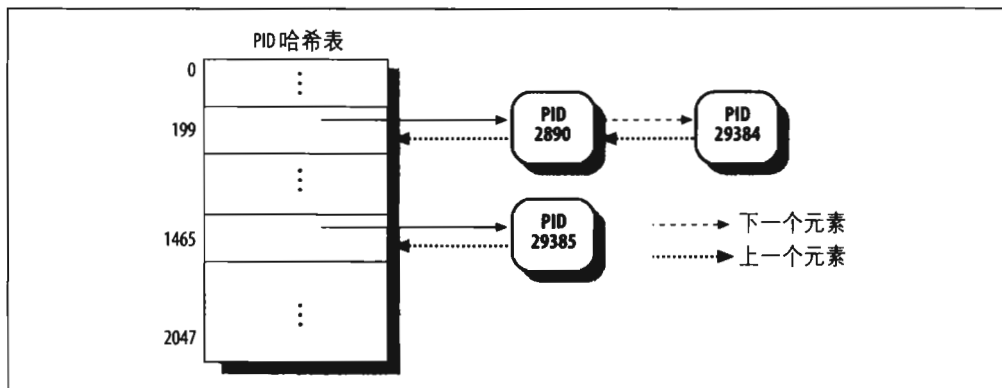


图 3-5: pidhash 表及链表

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32768（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32768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

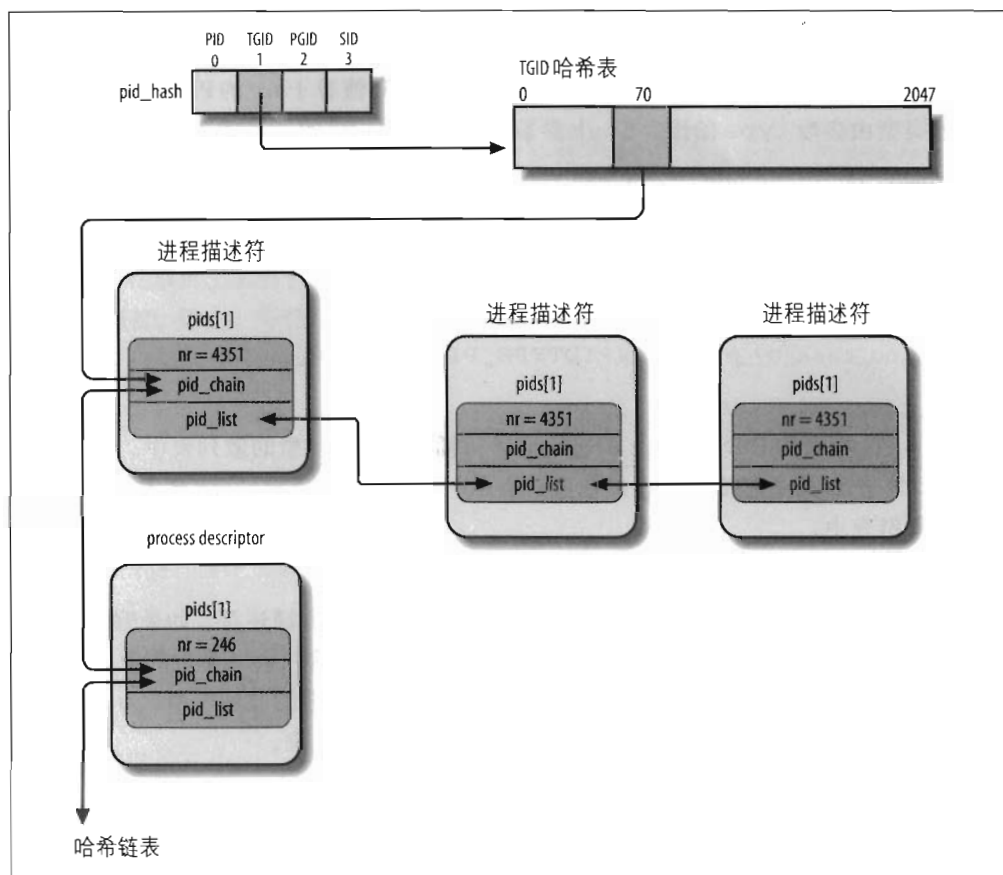


图 3-6: PID 散列表

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（flags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：一旦磁盘传输完成，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。

一旦定义了一个元素，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。

进程一旦被唤醒就执行 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 <code>malloc()</code> 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 <code>SIGXCPU</code> 信号, 然后如果进程还不终止, 再发一个 <code>SIGKILL</code> 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 <code>SIGXFSZ</code> 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 <code>mlock()</code> 或 <code>mlockall()</code> 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“ <code>clone()</code> , <code>fork()</code> 及 <code>vfork()</code> 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, 内核必须有能力和挂起正在 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` 宏一旦使 A 暂停，A 的执行流就冻结。

随后，当内核想再次激活 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。

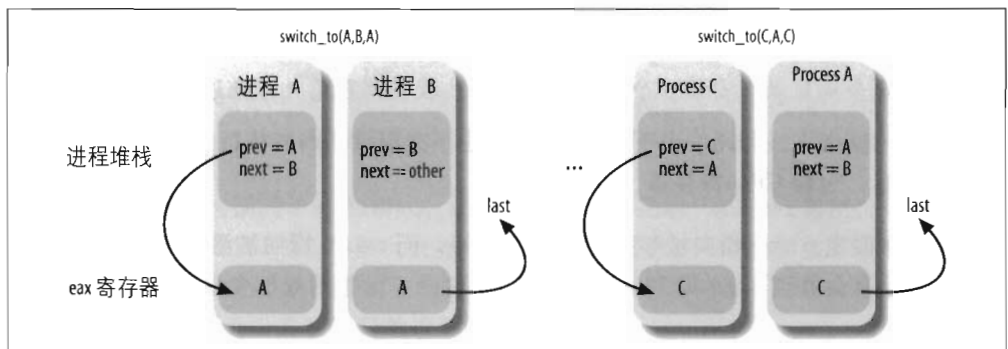


图 3-7：通过一个进程切换保留对进程 C 的引用

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_USED_FPU` 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_USED_FPU` 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_USED_FPU)
    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_USED_FPU标志:

```
prev->thread_info->status &= ~TS_USED_FPU;
```

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_USED_FPU;
}
```

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_USED_FPU` 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_USED_FPU` 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 `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 <code>umask</code>)
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 <code>true</code> , 就必须设置 <code>CLONE_VM</code> 标志
CLONE_PTRACE	如果父进程被跟踪, 那么, 子进程也被跟踪。尤其是, <code>debugger</code> 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 <code>vfork()</code> 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 <code>parent</code> 和 <code>real_parent</code> 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 <code>tgid</code> 字段和 <code>group_leader</code> 字段。如果这个标志位为 <code>true</code> , 就必须设置 <code>CLONE_SIGHAND</code> 标志
CLONE_NEWNS	当 <code>clone</code> 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 <code>CLONE_NEWNS</code> 和 <code>CLONE_FS</code>
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 <code>tls</code> 所指向的结构进行描述
CLONE_PARENT_SETTID	把子进程的 PID 写入由 <code>ptid</code> 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 <code>ctid</code> 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 <code>CLONE_PTRACE</code> 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, 通用寄存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_child_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)。使用计数器变为 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连接。
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。



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：
要根据中断允许标志的设置来判断CPU是否能响应中断请求
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。不受中断允许标志的影响，不能用软件进行屏蔽。

- 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。陷阱的主要用途是用于调试程序。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 Interrupt 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 IRQ_0 。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 Pentium III 开始引入了一种名为 I/O 高级可编程控制器 (I/O Advanced 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。

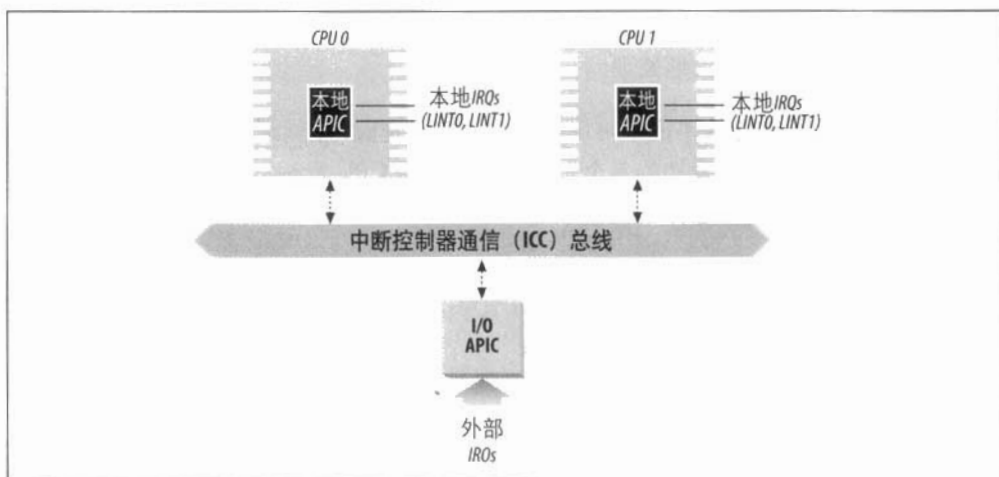


图 4-1: 多 APIC 系统

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 的技术文档中找到。

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 $256 \times 8 = 2048$ 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。

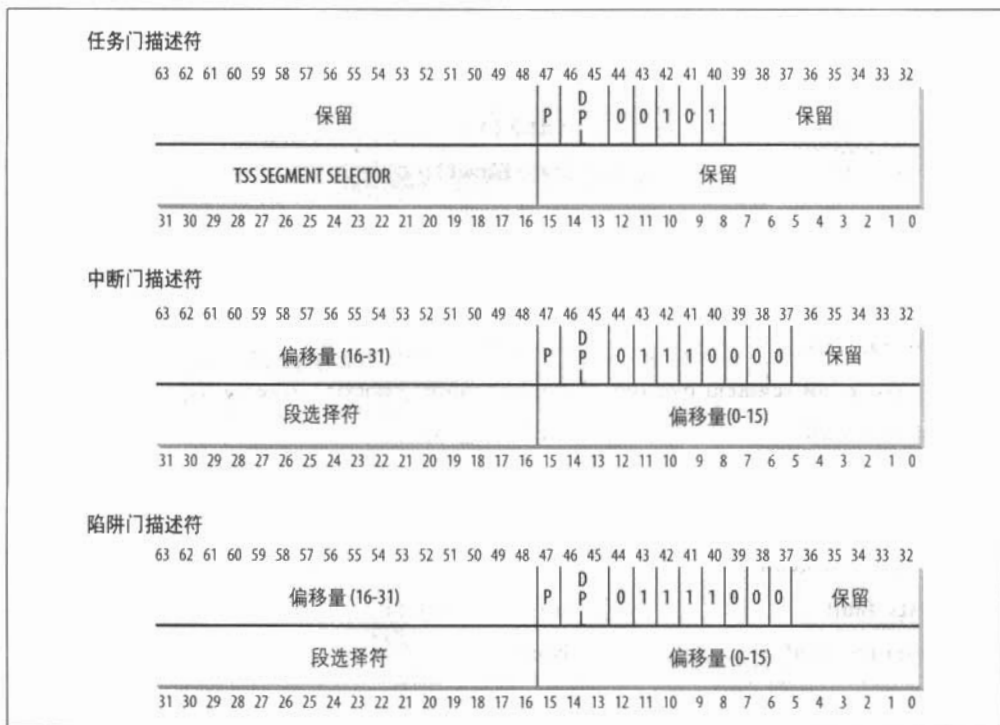


图 4-2: 门描述符的格式

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 ($0 \leq i \leq 255$)。
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。

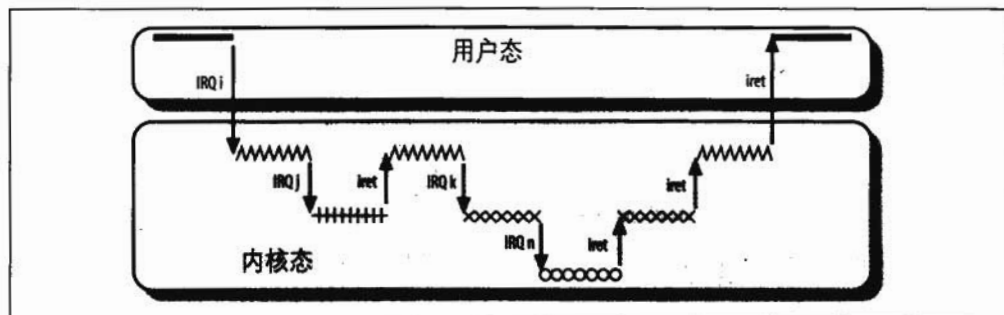


图 4-3：内核控制路径嵌套执行的例子

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，这个栈毫无疑问的属于当前进程。

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 available”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。

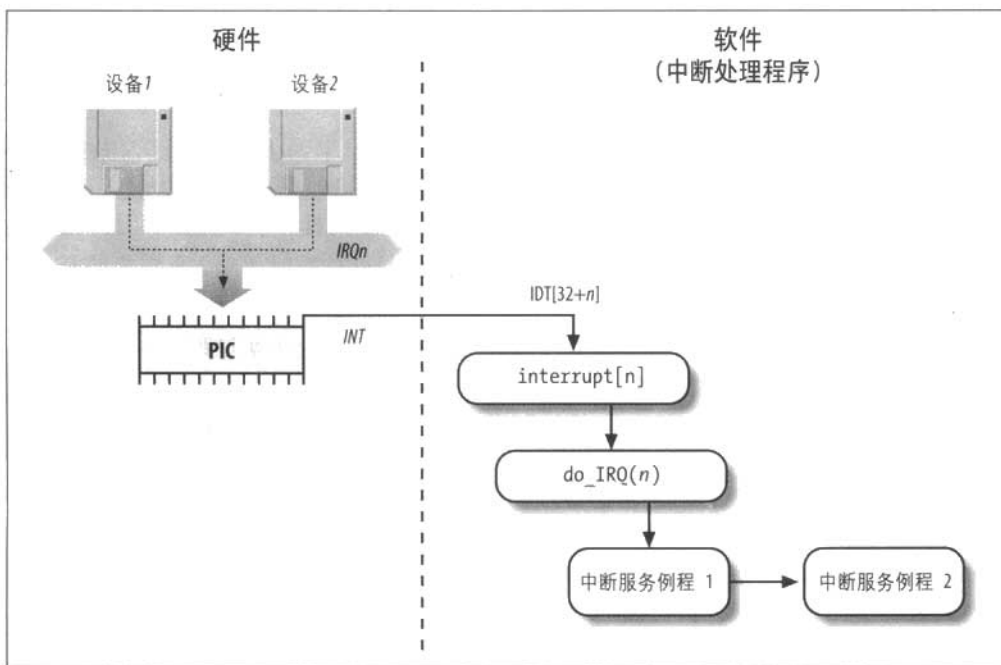


图 4-4: I/O 中断处理

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。

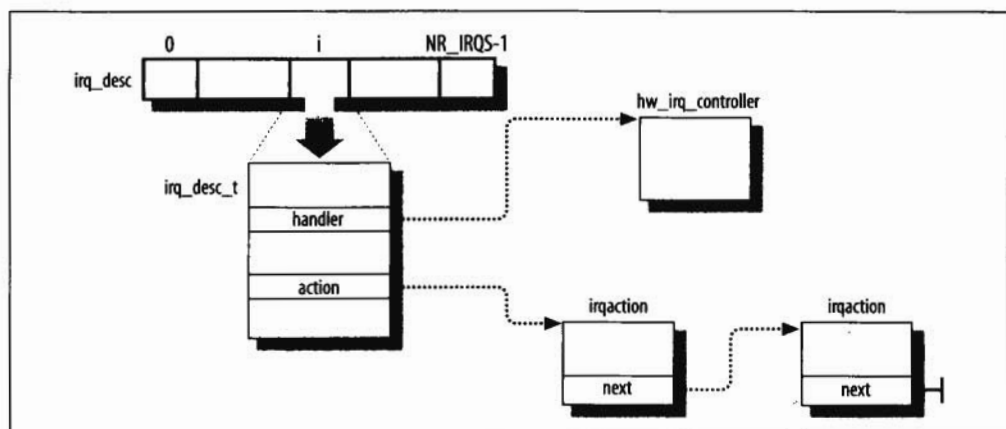


图 4-5: IRQ 描述符

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 irq_s_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 <code>/proc/interrupts</code> 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 <code>/proc/irq/n</code> 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 <code>/dev/random</code> 和 <code>/dev/urandom</code> 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
<code>__softirq_pending</code>	表示挂起的软中断 (见本章后面“软中断”一节), 为一组标志
<code>idle_timestamp</code>	CPU 变为空闲的时间 (只是在 CPU 正空闲的时候才有意义)
<code>__nmi_count</code>	NMI 中断发生的次数
<code>apic_timer_irqs</code>	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，Linux 2.6 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 的亲合力。

`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_irq()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
<code>lock</code>	保护该数据结构的自旋锁
<code>remove_sequence</code>	<code>flush_workqueue()</code> 使用的序列号
<code>insert_sequence</code>	<code>flush_workqueue()</code> 使用的序列号
<code>worklist</code>	挂起链表的头结点
<code>more_work</code>	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
<code>work_done</code>	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
<code>wq</code>	指向 <code>workqueue_struct</code> 结构的指针，其中包含该描述符
<code>thread</code>	指向结构中工作者线程的进程描述符指针
<code>run_depth</code>	<code>run_workqueue()</code> 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 <code>cpu_workqueue_struct</code> 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
<code>schedule_work(w)</code>	<code>queue_work(keventd_wq,w)</code>
<code>schedule_delayed_work(w,d)</code>	<code>queue_delayed_work(keventd_wq,w,d)</code> (在任何 CPU 上)
<code>schedule_delayed_work_on(cpu,w,d)</code>	<code>queue_delayed_work(keventd_wq,w,d)</code> (在某个指定的 CPU 上)
<code>flush_scheduled_work()</code>	<code>flush_workqueue(keventd_wq)</code>

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 Linux 2.6 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 <i>iret</i> 而不是 <i>sysexit</i> 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
    jmp 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)。

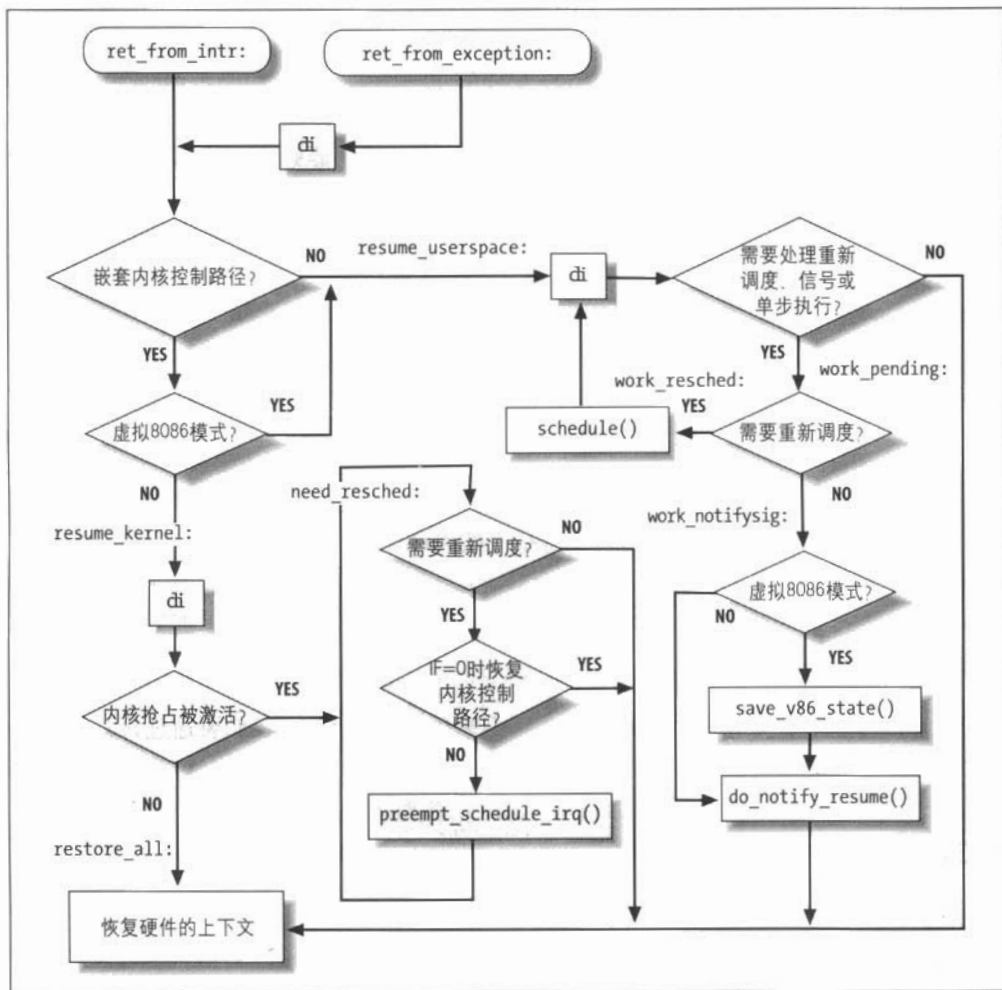


图 4-6: 从中断和异常返回

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` 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。



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，可能会暂时不理睬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，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。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、环境监测器、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: 处理抢占计数器字段的宏 (续)

宏	说明
<code>preempt_enable()</code>	使抢占计数器的值减 1, 并在 <code>thread_info</code> 描述符的 <code>TIF_NEED_RESCHED</code> 标志被置为 1 的情况下, 调用 <code>preempt_schedule()</code>
<code>get_cpu()</code>	与 <code>preempt_disable()</code> 相似, 但要返回本地 CPU 的数量
<code>put_cpu()</code>	与 <code>preempt_enable()</code> 相同
<code>put_cpu_no_resched()</code>	与 <code>preempt_enable_no_resched()</code> 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易。下面是一些可能简化的例子：

- 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的整数配。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
<code>atomic_read(v)</code>	返回 *v
<code>atomic_set(v, i)</code>	把 *v 置成 i
<code>atomic_add(i, v)</code>	给 *v 增加 i
<code>atomic_sub(i, v)</code>	从 *v 中减去 i
<code>atomic_sub_and_test(i, v)</code>	从 *v 中减去 i, 如果结果为 0, 则返回 1; 否则, 返回 0
<code>atomic_inc(v)</code>	把 1 加到 *v
<code>atomic_dec(v)</code>	从 *v 减 1
<code>atomic_dec_and_test(v)</code>	从 *v 减 1, 如果结果为 0, 则返回 1; 否则, 返回 0
<code>atomic_inc_and_test(v)</code>	把 1 加到 *v, 如果结果为 0, 则返回 1; 否则, 返回 0
<code>atomic_add_negative(i, v)</code>	把 i 加到 *v, 如果结果为负, 则返回 1; 否则, 返回 0
<code>atomic_inc_return(v)</code>	把 1 加到 *v, 返回 *v 的新值
<code>atomic_dec_return(v)</code>	从 *v 减 1, 返回 *v 的新值
<code>atomic_add_return(i, v)</code>	把 i 加到 *v, 返回 *v 的新值
<code>atomic_sub_return(i, v)</code>	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
<code>test_bit(nr, addr)</code>	返回 *addr 的第 nr 位的值
<code>set_bit(nr, addr)</code>	设置 *addr 的第 nr 位
<code>clear_bit(nr, addr)</code>	清 *addr 的第 nr 位
<code>change_bit(nr, addr)</code>	转换 *addr 的第 nr 位
<code>test_and_set_bit(nr, addr)</code>	设置 *addr 的第 nr 位, 并返回它的原值
<code>test_and_clear_bit(nr, addr)</code>	清 *addr 的第 nr 位, 并返回它的原值
<code>test_and_change_bit(nr, addr)</code>	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
<code>atomic_clear_mask(mask, addr)</code>	清 <code>mask</code> 指定的 <code>*addr</code> 的所有位
<code>atomic_set_mask(mask, addr)</code>	设置 <code>mask</code> 指定的 <code>*addr</code> 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
<code>mb()</code>	适用于 MP 和 UP 的内存屏障
<code>rmb()</code>	适用于 MP 和 UP 的读内存屏障
<code>wmb()</code>	适用于 MP 和 UP 的写内存屏障
<code>smp_mb()</code>	仅适用于 MP 的内存屏障
<code>smp_rmb()</code>	仅适用于 MP 的读内存屏障
<code>smp_wmb()</code>	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 。

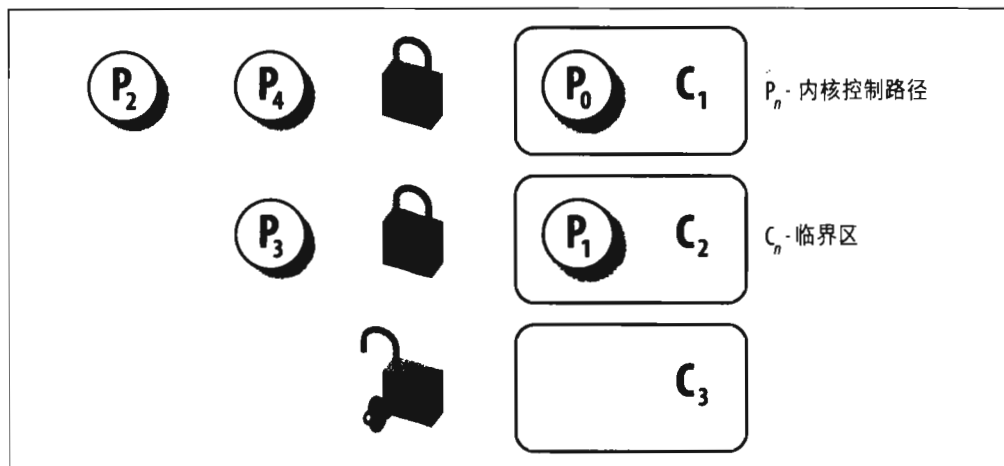


图 5-1: 用几个锁保护临界区

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锁的 spin_lock 宏。下面的描述都是针对支持 SMP 系统的抢占式内核的。该宏获取自旋锁的地址 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。

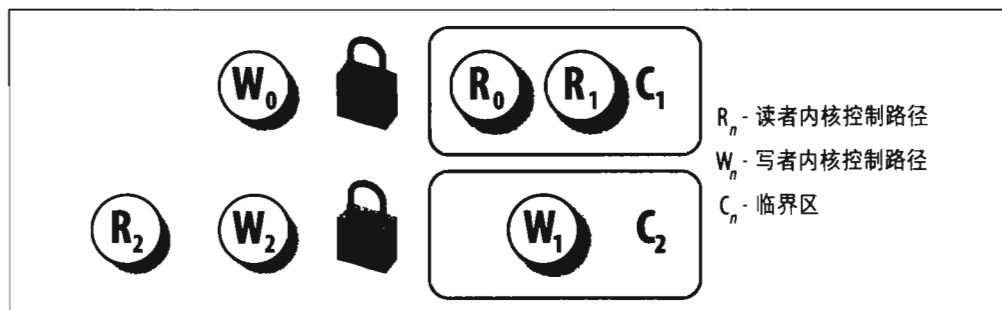


图 5-2: 读/写自旋锁

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 lf
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)
l:  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 rwp->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。一旦修改完毕，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 lf
    lea %ecx, %eax
    pushl %edx
    pushl %ecx
    call __down
    popl %ecx
    popl %edx
l:
```

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 打算一旦被唤醒就撤销该信号量。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表头和一个标志：

```
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
<code>spin_lock_irq(l)</code>	<code>local_irq_disable();spin_lock(l)</code>
<code>spin_unlock_irq(l)</code>	<code>spin_unlock(l);local_irq_enable()</code>
<code>spin_lock_bh(l)</code>	<code>local_bh_disable();spin_lock(l)</code>
<code>spin_unlock_bh(l)</code>	<code>spin_unlock(l);local_bh_enable()</code>
<code>spin_lock_irqsave(l,f)</code>	<code>local_irq_save(f);spin_lock(l)</code>
<code>spin_unlock_irqrestore(l,f)</code>	<code>spin_unlock(l);local_irq_restore(f)</code>
<code>read_lock_irq(l)</code>	<code>local_irq_disable();read_lock(l)</code>
<code>read_unlock_irq(l)</code>	<code>read_unlock(l);local_irq_enable()</code>
<code>read_lock_bh(l)</code>	<code>local_bh_disable();read_lock(l)</code>
<code>read_unlock_bh(l)</code>	<code>read_unlock(l);local_bh_enable()</code>
<code>write_lock_irq(l)</code>	<code>local_irq_disable();write_lock(l)</code>
<code>write_unlock_irq(l)</code>	<code>write_unlock(l);local_irq_enable()</code>
<code>write_lock_bh(l)</code>	<code>local_bh_disable();write_lock(l)</code>
<code>write_unlock_bh(l)</code>	<code>write_unlock(l);local_bh_enable()</code>
<code>read_lock_irqsave(l,f)</code>	<code>local_irq_save(f);read_lock(l)</code>
<code>read_unlock_irqrestore(l,f)</code>	<code>read_unlock(l);local_irq_restore(f)</code>
<code>write_lock_irqsave(l,f)</code>	<code>local_irq_save(f);write_lock(l)</code>
<code>write_unlock_irqrestore(l,f)</code>	<code>write_unlock(l);local_irq_restore(f)</code>
<code>read_seqbegin_irqsave(l,f)</code>	<code>local_irq_save(f);read_seqbegin(l)</code>
<code>read_seqretry_irqrestore(l,v,f)</code>	<code>read_seqretry(l,v);local_irq_restore(f)</code>
<code>write_seqlock_irqsave(l,f)</code>	<code>local_irq_save(f);write_seqlock(l)</code>
<code>write_sequnlock_irqrestore(l,f)</code>	<code>write_sequnlock(l);local_irq_restore(f)</code>
<code>write_seqlock_irq(l)</code>	<code>local_irq_disable();write_seqlock(l)</code>
<code>write_sequnlock_irq(l)</code>	<code>write_sequnlock(l);local_irq_enable()</code>
<code>write_seqlock_bh(l)</code>	<code>local_bh_disable();write_seqlock(l)</code>
<code>write_sequnlock_bh(l)</code>	<code>write_sequnlock(l);local_bh_enable()</code>

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 tasklet 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，竞争条件确实存在，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。



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 (*Real 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1ms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 的定时器的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
<code>name</code>	标识定时器源的一个字符串
<code>mark_offset</code>	记录上一个节拍的准确时间，由时钟中断处理程序调用
<code>get_offset</code>	返回自上一个节拍开始所经过的时间
<code>monotonic_clock</code>	返回自内核初始化开始所经过的纳秒数
<code>delay</code>	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得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率，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
<code>timer_hpet</code>	高精度事件定时器 (HPET)	HPET	HPET
<code>timer_pmtmr</code>	ACPI 电源管理定时器 (ACPI PMT)	ACPI PMT	TSC
<code>timer_tsc</code>	时间戳计数器 (TSC)	TSC	TSC
<code>timer_pit</code>	可编程间隔定时器 (PIT)	PIT	紧致循环
<code>timer_none</code>	普通虚拟定时器资源 (内核初始化时使用) (无)	(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率。

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0xffffb6c20，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率来测量时间。`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率来测量时间。`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 and 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_RUNNING` 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。

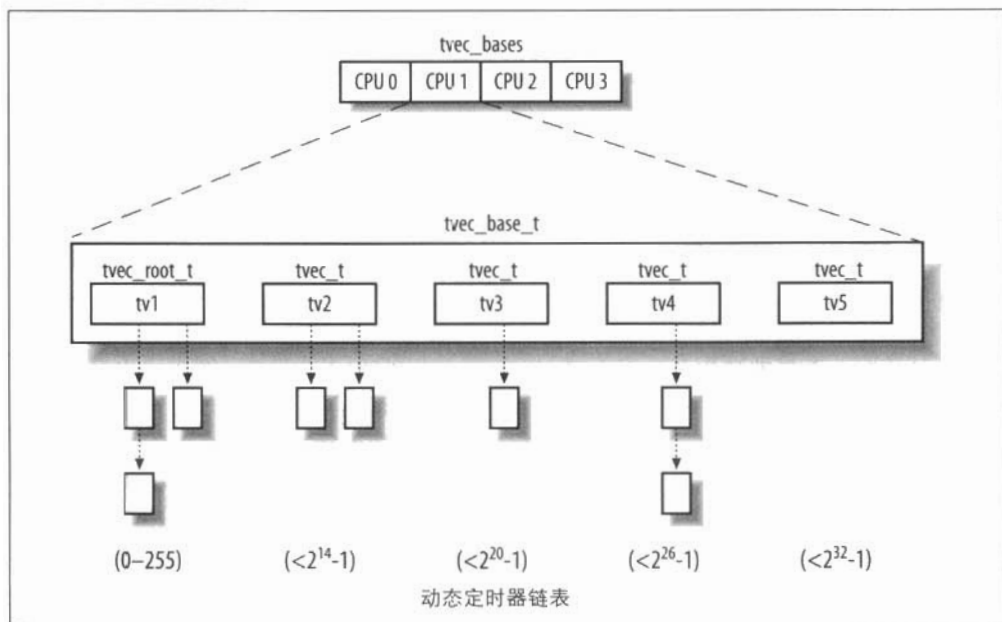


图 6-1：与动态定时器相关的链表组

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_timer_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() 中任何一种系统调

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率和属性。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
<code>clock_gettime()</code>	获得一个 POSIX 时钟的当前值
<code>clock_settime()</code>	设置一个 POSIX 时钟的当前值
<code>clock_getres()</code>	获得一个 POSIX 时钟的分辨率
<code>timer_create()</code>	在指定 POSIX 时钟基础上创建一个新的 POSIX 定时器
<code>timer_gettime()</code>	获得一个 POSIX 定时器的当前值和增量
<code>timer_settime()</code>	设置一个 POSIX 定时器的当前值和增量
<code>timer_getoverrun()</code>	获得到期 POSIX 定时器到期的数目
<code>timer_delete()</code>	销毁一个 POSIX 定时器
<code>clock_nanosleep()</code>	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 and 日期”一节）。`clock_getres()` 系统调用返回的分辨率为 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 时钟的分辨率由 `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
<code>nice()</code>	改变一个普通进程的静态优先级
<code>getpriority()</code>	获得一组普通进程的最大静态优先级
<code>setpriority()</code>	设置一组普通进程的静态优先级
<code>sched_getscheduler()</code>	获得一个进程的调度策略
<code>sched_setscheduler()</code>	设置一个进程的调度策略和实时优先级
<code>sched_getparam()</code>	获得一个进程的实时优先级
<code>sched_setparam()</code>	设置一个进程的实时优先级
<code>sched_yield()</code>	自愿放弃处理器而不阻塞
<code>sched_get_priority_min()</code>	获得一种策略的最小实时优先级
<code>sched_get_priority_max()</code>	获得一种策略的最大实时优先级
<code>sched_rr_get_interval()</code>	获得时间片轮转策略的时间片值
<code>sched_setaffinity()</code>	设置进程的 CPU 亲和力掩码
<code>sched_getaffinity()</code>	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, 在这种情况下, 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：

$$\text{基本时间片 (单位为 ms)} = \begin{cases} (140 - \text{静态优先级}) \times 20 & \text{若静态优先级} < 120 \\ (140 - \text{静态优先级}) \times 5 & \text{若静态优先级} \geq 120 \end{cases} \quad (1)$$

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。

$$\text{动态优先级} = \max(100, \min(\text{静态优先级} - \text{bonus} + 5, 139)) \quad (2)$$

*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：

$$\text{动态优先级} \leq 3 \times \text{静态优先级} / 4 + 28 \quad (3)$$

它相当于下面的公式：

$$\text{bonus} - 5 \geq \text{静态优先级} / 4 - 28$$

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，一旦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: `runqueue` 结构的字段

类型	名称	说明
<code>spinlock_t</code>	<code>lock</code>	保护进程链表的自旋锁
<code>unsigned long</code>	<code>nr_running</code>	运行队列链表中可运行进程的数量
<code>Unsigned long</code>	<code>cpu_load</code>	基于运行队列中进程的平均数量的 CPU 负载因子
<code>unsigned long</code>	<code>nr_switches</code>	CPU 执行进程切换的次数
<code>unsigned long</code>	<code>nr_uninterruptible</code>	先前在运行队列链表中而现在睡眠在 <code>TASK_UNINTERRUPTIBLE</code> 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: runqueue 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。

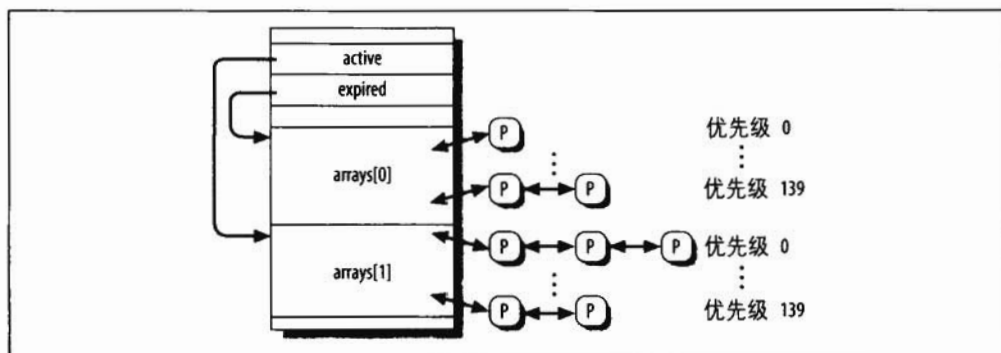


图 7-1: runqueue 结构和可运行进程的两个集合

arrays 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，维持当前进程的更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：
- 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 `smpt_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 $\min(\text{now} - p \rightarrow \text{timestamp}, 10^9)$ 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在于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 $(10 - \textit{bonus})$ 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. 一旦资源可用，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 <code>TASK_RUNNING</code> 状态
1	进程处于 <code>TASK_INTERRUPTIBLE</code> 或 <code>TASK_STOPPED</code> 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 <code>TASK_INTERRUPTIBLE</code> 或 <code>TASK_STOPPED</code> 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 <code>TASK_UNINTERRUPTIBLE</code> 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prev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，负载均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种主要的多处理器体系结构。

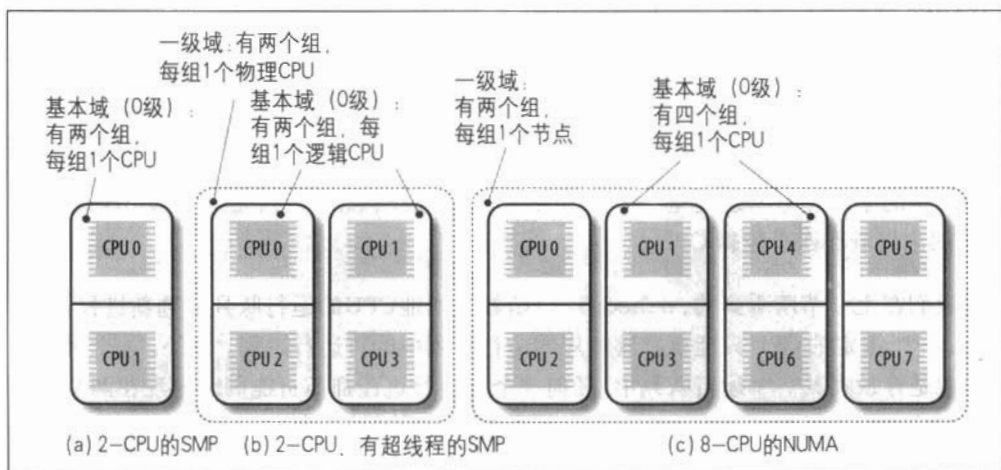


图 7-2: 调度域分层的 3 个实例

图 7-2(a) 表示具有两个 CPU 的标准多处理器体系结构中由单个调度域组成的一个层次结构, 该调度域包括两个组, 每个组有一个 CPU。

图 7-2 (b) 表示一个两层的层次结构, 用于使用超线程技术、有两个 CPU 的多处理器结构中。最上层的调度域包括了系统中所有四个逻辑 CPU, 它由两个组构成。上层域的每个组对应一个子调度域并包括一个物理 CPU。底层的调度域(也被称为基本调度域)包括两个组, 每个组一个逻辑 CPU。

最后, 图 7-2(c) 表示有两个节点, 每个节点有四个 CPU 的 8-CPU NUMA 体系结构上的两层层次结构。最上层的域由两个组构成, 每个组对应一个不同的节点。每个基本调度域包括一个节点内的 CPU, 包括四个组, 每个组包括一个 CPU。

每个调度域由一个 `sched_domain` 描述符表示, 而调度域中的每个组由 `sched_group` 描述符表示。每个 `sched_domain` 描述符包括一个 `groups` 字段, 它指向组描述符链表中的第一个元素。此外, `sched_domain` 结构的 `parent` 字段指向父调度域的描述符 (如果有的话)。

系统中所有物理 CPU 的 `sched_domain` 描述符都存放在每 CPU 变量 `phys_domains` 中。如果内核不支持超线程技术, 这些域就在域层次结构的最底层, 运行队列描述符的 `sd` 字段指向它们, 即它们是基本调度域。相反, 如果内核支持超线程技术, 则底层调度域存放在每 CPU 变量 `cpu_domains` 中。

rebalance_tick()函数

为了保持系统中运行队列的平衡，每次经过一次时钟节拍，`scheduler_tick()`就调用`rebalance_tick()`函数。它接收的参数有：本地CPU的下标`this_cpu`、本地运行队列的地址`this_rq`以及一个标志`idle`，该标志可以取下面的值：

`SCHED_IDLE`

CPU 当前空闲，即 `current` 是 `swapper` 进程。

`NOT_IDLE`

CPU 当前不空闲，即 `current` 不是 `swapper` 进程。

`rebalance_tick()`函数首先确定运行队列中的进程数，并更新运行队列的平均工作量，为了完成这个工作，函数要访问运行队列描述符的`nr_running`和`cpu_load`字段。

随后，`rebalance_tick()`开始在所有调度域上的循环，其路径是从基本域（本地运行队列描述符的`sd`字段所引用的域）到最上层的域。在每次循环中，函数确定是否已到调用函数`load_balance()`的时间，从而在调度域上执行重新平衡的操作。由存放在`sched_domain`描述符中的参数和`idle`值决定调用`load_balance()`的频率。如果`idle`等于`SCHED_IDLE`，那么运行队列为空，`rebalance_tick()`就以很高的频率调用`load_balance()`（大概每一到两个节拍处理一次对应于逻辑和物理CPU的调度域）。相反，如果`idle`等于`NOT_IDLE`，`rebalance_tick()`就以很低的频率调度`load_balance()`（大概每10ms处理一次逻辑CPU对应的调度域，每100ms处理一次物理CPU对应的调度域）。

load_balance()函数

`load_balance()`函数检查是否调度域处于严重的不平衡状态。更确切地说，它检查是否可以通过把最繁忙的组中的一些进程迁移到本地CPU的运行队列来减轻不平衡的状况。如果是，函数尝试实现这个迁移。它接收四个参数：

`this_cpu`

本地CPU的下标。

`this_rq`

本地运行队列的描述符的地址。

`sd`

指向被检查的调度域的描述符。

idle

取值为 SCHED_IDLE (本地 CPU 空闲) 或 NOT_IDLE。

函数执行下面的操作:

1. 获取 `this_rq->lock` 自旋锁。
2. 调用 `find_busiest_group()` 函数分析调度域中各组的工作量。函数返回最繁忙的组的 `sched_group` 描述符的地址, 假设这个组不包括本地 CPU, 在这种情况下, 函数还返回为了恢复平衡而被迁移到本地运行队列中的进程数。另一方面, 如果最繁忙的组包括本地 CPU 或所有的组本来就是平衡的, 函数返回 NULL。这个过程不是微不足道的, 因为函数试图过滤掉统计工作量中的波动。
3. 如果 `find_busiest_group()` 在调度域中没有找到既不包括本地 CPU 又非常繁忙的组, 就释放 `this_rq->lock` 自旋锁, 调整调度域描述符的参数, 以延迟本地 CPU 下一次对 `load_balance()` 的调度, 然后函数终止。
4. 调用 `find_busiest_queue()` 函数以查找在第 2 步中找到的组中最繁忙的 CPU, 函数返回相应运行队列的描述符地址 `busiest`。
5. 获取另一个自旋锁, 也就是 `busiest->lock` 自旋锁。为了避免死锁, 这一操作必须非常小心: 首先释放 `this_rq->lock`, 然后通过增加 CPU 下标获得这两个锁。
6. 调用 `move_tasks()` 函数, 尝试从最繁忙的运行队列中把一些进程迁移到本地运行队列 `this_rq` 中 (见下一节)。
7. 如果函数 `move_task()` 没有成功地把某些进程迁移到本地运行队列, 那么调度域还是不平衡。把 `busiest->active_balance` 标志设置为 1, 并唤醒 `migration` 内核线程, 它的描述符存放在 `busiest->migration_thread` 中。*Migration* 内核线程顺着调度域的链搜索—从最繁忙运行队列的基本域到最上层域, 寻找空闲 CPU。如果找到一个空闲 CPU, 该内核线程就调用 `move_tasks()` 把一个进程迁移到空闲运行队列。
8. 释放 `busiest->lock` 和 `this_rq->lock` 自旋锁。
9. 函数结束。

move_tasks()函数

`move_tasks()` 函数把进程从源运行队列迁移到本地运行队列。它接收 6 个参数: `this_rq` 和 `this_cpu` (本地运行队列描述符和本地 CPU 下标)、`busiest` (源运行队列描述符)、`max_nr_move` (被迁移进程的最大数)、`sd` (在其中执行平衡操作的调度域的描述符地址)

以及 `idle` 标志（除了可以被设置为 `SCHED_IDLE` 和 `NOT_IDLE` 以外，在函数被 `idle_balance()` 间接调用时，该标志还可以被设置为 `NEWLY_IDLE`。见本章前面“`schedule()`函数”一节）。

函数首先分析 `busiest` 运行队列的过期进程，从优先级高的进程开始。当扫描完所有过期进程后，函数扫描 `busiest` 运行队列的活动进程。函数对所有的候选进程调用 `can_migrate_task()`，如果下列条件都满足，则 `can_migrate_task()` 返回 1：

- 进程当前没有在远程 CPU 上执行。
- 本地 CPU 包含在进程描述符的 `cpus_allowed` 位掩码中。
- 至少满足下列条件之一：
 - 本地 CPU 空闲。如果内核支持超线程技术，则所有本地物理芯片中的逻辑 CPU 必须空闲。
 - 内核在平衡调度域时因反复进行进程迁移都不成功而陷入困境。
 - 被迁移的进程不是“高速缓存命中”的（最近不曾在远程 CPU 上执行，因此可以设想远程 CPU 上的硬件高速缓存中没有该进程的数据）。

如果 `can_migrate_task()` 返回 1，`move_tasks()` 就调用 `pull_task()` 函数把候选进程迁移到本地运行队列中。实际上，`pull_task()` 执行 `dequeue_task()` 从远程运行队列删除进程，然后执行 `enqueue_task()` 把进程插入本地运行队列，最后，如果刚被迁移的进程比当前进程拥有更高的动态优先级，就调用 `resched_task()` 抢占本地 CPU 的当前进程。

与调度相关的系统调用

已经介绍的几个系统调用允许进程改变它们的优先级及调度策略。作为一般原则，总是允许用户降低其进程的优先级。然而，如果他们想修改属于其他某一用户进程的优先级，或者如果他们想增加自己进程的优先级，那么，他们必须拥有超级用户的特权。

`nice()` 系统调用

`nice()`（注 3）系统调用允许进程改变它们的基本优先级。包含在 `increment` 参数中的整数值用来修改进程描述符的 `nice` 字段。在 Unix 中的 `nice` 命令（允许用户用修改的调度优先级来运行程序）就是基于这个系统调用的。

注 3： 因为这个系统调用用来降低进程的优先级，因此为了自己的优先级而调用它的用户对他用户来说就是“美好的（`nice`）”。

`sys_nice()` 服务例程处理 `nice()` 系统调用。尽管 `increment` 参数可以有任何值，但是大于 40 的绝对值会被截为 40。从传统上来说，负值相当于请求优先级增加，并请求超级用户特权，而正值相当于请求优先级减少。在负增加的情况下，调用 `capable()` 函数核实进程是否有 `CAP_SYS_NICE` 权能。而且，函数调用 `security_task_setnice()` 安全勾。我们将在第二十章讨论那个函数。如果用户想用请求的权能来改变优先级，`sys_nice()` 就把 `current->static_prio` 转换到 `nice` 值的范围，再加上 `increment` 的值，并调用 `set_user_nice()` 函数。然后，`set_user_nice()` 函数获得本地运行队列锁，更新 `current` 进程的静态优先级，调用 `resched_task()` 函数以允许其他进程抢占 `current` 进程，并释放运行队列锁。

`nice()` 系统调用只维持向后兼容；它已经被下面描述的 `setpriority()` 系统调用取代。

getpriority()和 setpriority()系统调用

`nice()` 系统调用只影响调用它的进程，而另外两个系统调用 `getpriority()` 和 `setpriority()` 则作用于给定组中所有进程的基本优先级。`getpriority()` 返回 20 减去给定组中所有进程之中最低 `nice` 字段的值（即所有进程中的最高优先级）；`setpriority()` 把给定组中所有进程的基本优先级都设置为一个给定的值。

内核对这两个系统调用的实现是通过 `sys_getpriority()` 和 `sys_setpriority()` 服务例程完成的。这两个服务例程本质上作用于同一组相同的参数：

`which`

指定进程组的值。它采用下列值之一：

`PRIO_PROCESS`

根据进程的 ID 选择进程（进程描述符的 `pid` 字段）

`PRIO_PGRP`

根据组 ID 选择进程（进程描述符的 `pgrp` 字段）

`PRIO_USER`

根据用户 ID 选择进程（进程描述符的 `uid` 字段）

`who`

用 `pid`、`pgrp` 或 `uid` 字段的值（取决于 `which` 的值）选择进程。如果 `who` 是 0，则把它的值设置为 `current` 进程相应字段的值。

`niceval`

新的基本优先级值（仅被 `sys_setpriority()` 所需要）。它的取值范围应该在 -20（最高优先级）~ +19（最小优先级）之间。

正如以前提到的，只有具有 `CAP_SYS_NICE` 权能的进程才允许增加它们自己的基本优先级或修改其他进程的优先级。

正如我们将在第十章看到的，只有当出现了某些错误时，系统调用才返回一个负值。由于这个原因，`getpriority()` 不返回 $-20 \sim +19$ 之间正常的 nice 值，而是 $1 \sim 40$ 之间的一个非负值。

`sched_getaffinity()`和 `sched_setaffinity()`系统调用

`sched_getaffinity()`和 `sched_setaffinity()`系统调用分别返回和设置CPU进程亲和力和掩码，也就是允许执行进程的CPU的位掩码。该掩码存放在进程描述符的 `cpus_allowed` 字段中。

`sys_sched_getaffinity()`系统调用服务例程通过调用 `find_task_by_pid()` 搜索进程描述符，返回的值为相应字段 `cpus_allowed` 与可用CPU位图做与运算的结果。

系统调用 `sys_sched_setaffinity()` 有一点复杂。除了寻找目标进程的描述符并更新 `cpus_allowed` 字段以外，该函数还必须检查进程所属的运行队列，其对应的CPU亲和力掩码是否不再是最新值。在这种糟糕的情况下，必须把进程从一个运行队列迁移到另一个运行队列。为了避免死锁和竞争条件的问题，由 `migration` 内核线程（每个CPU有一个这样的线程）完成这个工作。一旦必须把进程从运行队列 `rq1` 迁移到运行队列 `rq2`，`sys_sched_setaffinity()`系统调用就唤醒 `rq1` 的迁移线程 (`rq1->migration_thread`)，该线程从 `rq1` 中删除被迁移的进程，然后把它插入 `rq2`。

与实时进程相关的系统调用

现在我们介绍一组系统调用，它们允许进程改变自己的调度规则，尤其是可以变为实时进程。进程为了修改任何进程（包括自己）的描述符的 `rt_priority` 和 `policy` 字段，同样必须具有 `CAP_SYS_NICE` 权能。

`sched_getscheduler()`和 `sched_setscheduler()`系统调用

`sched_getscheduler()`查询由 `pid` 参数所表示的进程当前所用的调度策略。如果 `pid` 等于 0，将检索调用进程的策略。如果成功，这个系统调用为进程返回策略：`SCHED_FIFO`、`SCHED_RR`或 `SCHED_NORMAL`（后者也称为 `SCHED_OTHER`）。相应的 `sys_sched_getscheduler()`服务例程调用 `find_task_by_pid()`，后一个函数确定给定 `pid` 所对应的进程描述符，并返回其 `policy` 字段的值。

`sched_setscheduler()`系统调用既设置调度策略,也设置由参数`pid`所表示进程的相关参数。如果`pid`等于0,调用进程的调度程序参数将被设置。

相应的`sys_sched_setscheduler()`系统调用服务例程简单地调用`do_sched_setscheduler()`函数。后者检查由参数`policy`指定的调度策略和由参数`param->sched_priority`指定的新优先级是否有效。它还检查进程是否具有`CAP_SYS_NICE`权能,或者进程的拥有者是否有超级用户的权限。如果每个条件都满足,就把进程从它的运行队列(如果进程是可运行的)中删除;更新进程的静态优先级、实时优先级和动态优先级;把进程插回到运行队列;最后,在需要的情况下,调用`resched_task()`函数抢占运行队列的当前进程。

`sched_getparam()`和`sched_setparam()`系统调用

`sched_getparam()`系统调用为`pid`所表示的进程检索调度参数。如果`pid`是0,则`current`进程的参数被检索。正如你所期望的,相应的`sys_sched_getparam()`服务例程找到与`pid`相关的进程描述符指针,把它的`rt_priority`字段存放在类型为`sched_param`的局部变量中,并调用`copy_to_user()`把它拷贝到进程地址空间中由`param`参数指定的地址。

`sched_setparam()`系统调用类似于`sched_setscheduler()`,它与后者的不同在于不让调用者设置`policy`字段的值(注4)。相应的`sys_sched_setparam()`服务例程用几乎与`sys_sched_setscheduler()`相同的参数调用`do_sched_setscheduler()`。

`sched_yield()`系统调用

`sched_yield()`系统调用允许进程在不被挂起的情况下自愿放弃CPU,进程仍然处于`TASK_RUNNING`状态,但调度程序把它放在运行队列的过期进程集合中(如果进程是普通进程),或放在运行队列链表的末尾(如果进程是实时进程)。随后调用`schedule()`函数。在这种方式下,具有相同动态优先级的其他进程将有机会运行。这个调用主要由`SCHED_FIFO`实时进程使用。

`sched_get_priority_min()`和`sched_get_priority_max()`系统调用

`sched_get_priority_min()`和`sched_get_priority_max()`系统调用分别返回最小和最大实时静态优先级的值,这个值由`policy`参数所标识的调度策略来使用。

注4: POSIX标准的一个特殊要求造成了这种异常情况。

如果 `current` 是实时进程，则 `sys_sched_get_priority_min()` 服务例程返回 1，否则返回 0。

如果 `current` 是实时进程，则 `sys_sched_get_priority_max()` 服务例程返回 99（最高优先级），否则返回 0。

sched_rr_get_interval() 系统调用

`sched_rr_get_interval()` 系统调用把参数 `pid` 表示的实时进程的轮转时间片写入用户态地址空间的一个结构中。如果 `pid` 等于 0，系统调用就写当前进程的时间片。

相应的 `sys_sched_rr_get_interval()` 服务例程同样调用 `find_process_by_pid()` 检索与 `pid` 相关的进程描述符。然后，把所选中进程的基本时间片转换为秒数和纳秒数，并把它们拷贝到用户态的结构中。通常，FIFO 实时进程的时间片等于 0。

第八章



内存管理

在第二章中我们已看到，Linux 如何有效地利用 80x86 的分段和分页单元把逻辑地址转换为物理地址。我们还提到，RAM 的某些部分永久地分配给内核，并用来存放内核代码以及静态内核数据结构。

RAM 的其余部分称为动态内存 (dynamic memory)，这不仅是进程所需的宝贵资源，也是内核本身所需的宝贵资源。实际上，整个系统的性能取决于如何有效地管理动态内存。因此，现在所有多任务操作系统都在尽力优化对动态内存的使用，也就是说，尽可能做到当需要时分配，不需要时释放。图 8-1 显示了用作动态内存的页框，详细内容请参见第二章的“物理内存布局”一节。

本章主要通过三部分内容描述内核如何给自己分配动态内存。“页框管理”和“内存区管理”两节分别介绍对连续物理内存区处理的两种不同技术，而“非连续内存区管理”一节介绍了处理非连续内存区的第三种技术。在这几节中我们的主题将涉及诸如内存管理区、内核映射、伙伴系统、slab 高速缓存和内存池。

页框管理

在第二章“硬件中的分页”一节中我们曾介绍过，Intel 的 Pentium 处理器可以采用两种不同的页框大小：4KB 和 4MB（或者如果 PAE 被激活，则为 2MB——参见第二章“物理地址扩展 (PAE) 分页机制”一节）。Linux 采用 4KB 页框大小作为标准的内存分配单元。基于以下两个原因，这会使事情变得简单：

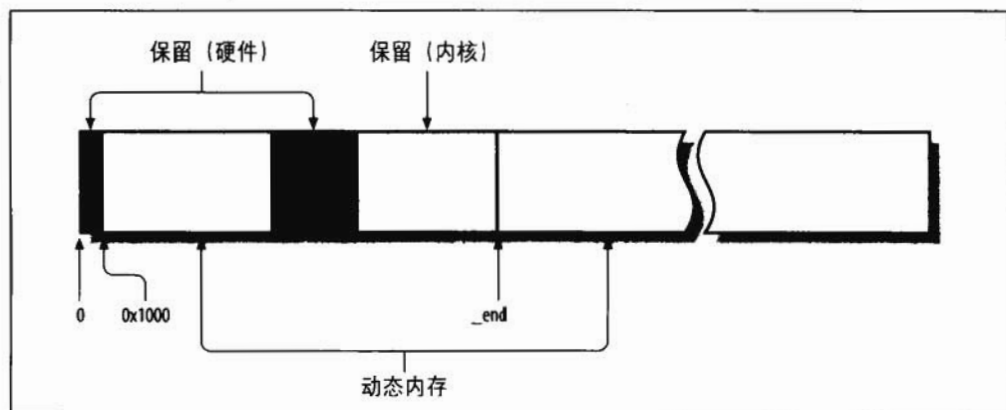


图 8-1：动态内存

- 由分页单元引发的缺页异常很容易得到解释，或者是由于请求的页存在但不允许进程对其访问，或者是由于请求的页不存在。在第二种情况下，内存分配器必须找到一个 4KB 的空闲页框，并将其分配给进程。
- 虽然 4KB 和 4MB 都是磁盘块大小的倍数，但是在绝大多数情况下，当主存和磁盘之间传输小块数据时更高效。

页描述符

内核必须记录每个页框当前的状态。例如，内核必须能区分哪些页框包含的是属于进程的页，而哪些页框包含的是内核代码或内核数据。类似地，内核还必须能够确定动态内存中的页框是否空闲。如果动态内存中的页框不包含有用的数据，那么这个页框就是空闲的。在以下情况下页框是不空闲的：包含用户态进程的数据、某个软件高速缓存的数据、动态分配的内核数据结构、设备驱动程序缓冲的数据、内核模块的代码等等。

页框的状态信息保存在一个类型为 `page` 的页描述符中，其中的字段如表 8-1 所示。所有的页描述符存放在 `mem_map` 数组中。因为 每个描述符长度为 32 字节，所以 `mem_map` 所需要的空间略小于整个 RAM 的 1%。`virt_to_page(addr)` 宏产生线性地址 `addr` 对应的页描述符地址。`pfn_to_page(pfn)` 宏产生与页框号 `Pfn` 对应的页描述符地址。

表 8-1: 页描述符的字段

类型	名字	说明
unsigned long	flags	一组标志 (参见表 8-2)。也对页框所在的管理区进行编号
atomic_t	_count	页框的引用计数器
atomic_t	_mapcount	页框中的页表项数目 (如果没有则为 -1)
unsigned long	private	可用于正在使用页的内核成分 (例如, 在缓冲页的情况下它是一个缓冲器头指针; 参见第十五章的“块缓冲区和缓冲区首部”一节)。如果页是空闲的, 则该字段由伙伴系统使用 (参见本章后面)
struct address_space *	mapping	当页被插入页高速缓存中时使用 (参见第十五章“页高速缓存”一节), 或者当页属于匿名区时使用 (参见第十七章的“匿名页的反向映射”一节)
unsigned long	index	作为不同的含义被几种内核成分使用。例如, 它在页磁盘映像或匿名区中标识存放在页框中的数据的位置 (参见第十五章), 或者它存放一个换出页标识符 (第十七章)
struct list_head	lru	包含页的最近最少使用 (LRU) 双向链表的指针

你不必现在就完全理解页描述符所有字段的作用。在接下来的章节中, 我们常常会回到页描述符的字段。此外, 有几个字段的含义还取决于页框是否空闲及什么样的内核成分在使用页框。

让我们较详细地描述以下两个字段:

`_count`

页的引用计数器。如果该字段为 -1, 则相应页框空闲, 并可被分配给任一进程或内核本身; 如果该字段的值大于或等于 0, 则说明页框被分配给了一个或多个进程, 或用于存放一些内核数据结构。page_count() 函数返回 _count 加 1 后的值, 也就是该页的使用者的数目。

`flags`

包含多达 32 个用来描述页框状态的标志 (参见表 8-2)。对于每个 PG_xyz 标志, 内核都定义了操纵其值的一些宏。通常, PageXYZ 宏返回标志的值, 而 SetPageXYZ 和 ClearPageXYZ 宏分别设置和清除相应的位。

表 8-2: 描述页框状态的标志

标志名	含义
PG_locked	页被锁定。例如, 在磁盘 I/O 操作中涉及的页
PG_error	在传输页时发生 I/O 错误
PG_referenced	刚刚访问过的页
PG_uptodate	在完成读操作后置位, 除非发生磁盘 I/O 错误
PG_dirty	页已经被修改 (参见第十七章的“PFRA 实现”一节)
PG_lru	页在活动或非活动页链表中[参见第十七章的“最近最少使用 (LRU) 链表”一节]
PG_active	页在活动页链表中[参看第十七章的“最近最少使用 (LRU) 链表”一节]
PG_slab	包含在 slab 中的页框 (参见本章后面“内存区管理”一节)
PG_highmem	页框属于 ZONE_HIGHMEM 管理区(参见本章后面“非一致内存访问 (NUMA)”一节)
PG_checked	由一些文件系统 (如 Ext2 和 Ext3) 使用的标志 (参见第十八章)
PG_arch_1	在 80x86 体系结构上没有使用
PG_reserved	页框留给内核代码或没有使用
PG_private	页描述符的 private 字段存放了有意义的数
PG_writeback	正在使用 writepage 方法将页写到磁盘上 (参见第十六章)
PG_nosave	系统挂起 / 唤醒时使用
PG_compound	通过扩展分页机制处理页框 (参见第二章的“扩展分页”一节)
PG_swapcache	页属于对换高速缓存 (参见第十七章的“交换高速缓存”一节)
PG_mappedtodisk	页框中的所有数据对应于磁盘上分配的块
PG_reclaim	为回收内存对页已经做了写入磁盘的标记
PG_nosave_free	系统挂起 / 恢复时使用

非一致内存访问 (NUMA)

我们习惯上认为计算机内存是一种均匀、共享的资源。在忽略硬件高速缓存作用的情况下, 我们期望不管内存单元处于何处, 也不管 CPU 处于何处, CPU 对内存单元的访问都需要相同的时间。可惜, 这种假设在某些体系结构上并不总是成立。例如, 对于某些多处理器 Alpha 或 MIPS 计算机, 这就不成立。

Linux 2.6 支持非一致内存访问 (Non-Uniform Memory Access, NUMA) 模型, 在这

种模型中，给定CPU对不同内存单元的访问时间可能不一样。系统的物理内存被划分为几个节点 (*node*)。在一个单独的节点内，任一给定CPU访问页面所需的时间都是相同的。然而，对不同的CPU，这个时间可能就不同。对每个CPU而言，内核都试图把耗时节点的访问次数减到最少，这就要小心地选择CPU最常引用的内核数据结构的存放位置 (注1)。

每个节点中的物理内存又可以分为几个管理区 (Eone)，这我们将在下一节介绍。每个节点都有一个类型为 `pg_data_t` 的描述符，它的字段如表 8-3 所示。所有节点的描述符存放在一个单向链表中，它的第一个元素由 `pgdat_list` 变量指向。

表 8-3: 节点描述符的字段

类型	名字	说明
<code>struct zone_t[]</code>	<code>node_zones</code>	节点中管理区描述符的数组
<code>struct zonelist_t[]</code>	<code>node_zonelists</code>	页分配器使用的 <code>zonelist</code> 数据结构的数组 (参见后面“内存管理区”一节)
<code>int</code>	<code>nr_zones</code>	节点中管理区的个数
<code>struct page *</code>	<code>node_mem_map</code>	节点中页描述符的数组
<code>struct bootmem_data *</code>	<code>bdata</code>	用在内核初始化阶段
<code>unsigned long</code>	<code>node_start_pfn</code>	节点中第一个页框的下标
<code>unsigned long</code>	<code>node_present_pages</code>	内存节点的大小，不包括洞 (以页框为单位)
<code>unsigned long</code>	<code>node_spanned_pages</code>	节点的大小，包括洞 (以页框为单位)
<code>int</code>	<code>node_id</code>	节点标识符
<code>pg_data_t *</code>	<code>pgdat_next</code>	内存节点链表中的下一项
<code>wait_queue_head_t</code>	<code>kswapd_wait</code>	<code>kswapd</code> 页换出守护进程使用的等待队列 (参见第十七章的“周期回收”一节)
<code>struct task_struct *</code>	<code>kswapd</code>	指针指向 <code>kswapd</code> 内核线程的进程描述符
<code>int</code>	<code>kswapd_max_order</code>	<code>kswapd</code> 将要创建的空闲块大小取对数的值

注1: 另外，Linux 内核甚至在一些特殊的单处理器系统上使用 NUMA，这些系统的物理地址空间中拥有巨大的“洞”。内核通过将有效物理地址的连续附属区域分配给不同的内存节点来处理这些体系结构。

我们同样只关注 80x86 体系结构。IBM 兼容 PC 使用一致访问内存 (UMA) 模型，因此，并不真正需要 NUMA 的支持。然而，即使 NUMA 的支持没有编译进内核，Linux 还是使用节点，不过，这是一个单独的节点，它包含了系统中所有的物理内存。因此，`pgdat_list` 变量指向一个链表，此链表是由一个元素组成的，这个元素就是节点 0 描述符，它被存放在 `contig_page_data` 变量中。

在 80x86 结构中，把物理内存分组在一个单独的节点中可能显得没有用处；但是，这种方式有助于内存代码的处理更具有可移植性，因为内核假定在所有的体系结构中物理内存都被划分为一个或多个节点（注 2）。

内存管理区

在一个理想的计算机体系结构中，一个页框就是一个内存存储单元，可用于任何事情：存放内核数据和用户数据、缓冲磁盘数据等等。任何种类的数据页都可以存放在任何页框中，没有什么限制。

但是，实际的计算机体系结构有硬件的制约，这限制了页框可以使用的方式。尤其是，Linux 内核必须处理 80x86 体系结构的两种硬件约束：

- ISA 总线的直接内存存取 (DMA) 处理器有一个严格的限制：它们只能对 RAM 的前 16MB 寻址。
- 在具有大容量 RAM 的现代 32 位计算机中，CPU 不能直接访问所有的物理内存，因为线性地址空间太小。

为了应对这两种限制，Linux 2.6 把每个内存节点的物理内存划分为 3 个管理区 (zone)。在 80x86 UMA 体系结构中的管理区为：

ZONE_DMA

包含低于 16 MB 的内存页框

ZONE_NORMAL

包含高于 16 MB 且低于 896 MB 的内存页框

ZONE_HIGHMEM

包含从 896MB 开始高于 896 MB 的内存页框

注 2： 我们还有这种设计选择的另外一个例子：即使硬件体系结构仅定义了两级页表，Linux 也使用四级（参见第二章“Linux 中的分页”一节）。

ZONE_DMA 区包含的页框可以由老式基于 ISA 的设备通过 DMA 使用[第十三章“直接内存访问 (DMA)”一节将进一步给出详细的信息]。

ZONE_DMA 和 ZONE_NORMAL 区包含内存的“常规”页框，通过把它们线性地映射到线性地址空间的第 4 个 GB，内核就可以直接进行访问（参见第二章的“内核页表”一节）。相反，ZONE_HIGHMEM 区包含的内存页不能由内核直接访问，尽管它们也线性地映射到了线性地址空间的第 4 个 GB（参见本章后面“高端内存页框的内核映射”一节）。在 64 位体系结构上 ZONE_HIGHMEM 区总是空的。

每个内存管理区都有自己的描述符。它的字段如表 8-4 所示。

表 8-4：管理区描述符的字段

类型	名称	说明
unsigned long	free_pages	管理区中空闲页的数目
unsigned long	pages_min	管理区中保留页的数目（参见本章后面的“保留的页框池”一节）
unsigned long	pages_low	回收页框使用的下界；同时也被管理区分配器作为阈值使用（参见本章后面的“管理区分配器”一节）
unsigned long	pages_high	回收页框使用的上界；同时也被管理区分配器作为阈值使用
Unsigned long[]	lowmem_reserve	指明在处理内存不足的临界情况下每个管理区必须保留的页框数目
struct per_cpu_pageset[]	pageset	数据结构用于实现单一页框的特殊高速缓存（参见本章后面的“每 CPU 页框高速缓存”一节）
spinlock_t	lock	保护该描述符的自旋锁
struct free_area[]	free_area	标识出管理区中的空闲页框块（参见本章后面的“伙伴系统算法”一节）
spinlock_t	lru_lock	活动以及非活动链表使用的自旋锁
struct list head	active_list	管理区中的活动页链表（参见第十七章）
struct list head	inactive_list	管理区中的非活动页链表（参见第十七章）

表 8-4: 管理区描述符的字段 (续)

类型	名称	说明
unsigned long	nr_scan_active	回收内存时需要扫描的活动页数目 (参见第十七章的“内存不足时回收”一节)
unsigned long	nr_scan_inactive	回收内存时需要扫描的非活动页数目
unsigned long	nr_active	管理区的活动链表上的页数目
unsigned long	nr_inactive	管理区的非活动链表上的页数目
unsigned long	pages_scanned	管理区内回收页框时使用的计数器
int	all_unreclaimable	在管理区中填满不可回收页时此标志被置位
int	temp_priority	临时管理区的优先级 (回收页框时使用)
int	prev_priority	管理区优先级, 范围在 12 和 0 之间 (由回收页框算法使用, 参见第十七章的“内存紧缺回收”一节)
wait_queue_head_t*	wait_table	进程等待队列的散列表, 这些进程正在等待管理区中的某页
unsigned long	wait_table_size	等待队列散列表的大小
unsigned long	wait_table_bits	等待队列散列表数组大小, 值为 2^{order}
struct pglist_data*	zone_pgdat	内存节点 [参见前面的“非一致内存访问 (NUMA)”一节]
struct page *	zone_mem_map	指向管理区的第一个页描述符的指针
unsigned long	zone_start_pfn	管理区第一个页框的下标
unsigned long	spanned_pages	以页为单位的管理区的总大小, 包括洞
unsigned long	present_pages	以页为单位的管理区的总大小, 不包括洞
char*	name	指针指向管理区的传统名称: “DMA”, “NORMAL” 或 “HighMem”

管理区结构中的许多字段用于回收页框, 相关内容将在第十七章中描述。

每个页描述符都有到内存节点和到节点内管理区 (包含相应页框) 的链接。为节省空间, 这些链接的存放方式与典型的指针不同, 而是被编码成索引存放在 flags 字段的高位。

实际上，刻画页框的标志的数目是有限的，因此保留 flags 字段的最高位来编码特定内存节点和管理区号总是可能的（注3）。page_zone() 函数接收一个页描述符的地址作为它的参数；它读取页描述符中 flags 字段的最高位，然后通过查看 zone_table 数组来确定相应管理区描述符的地址。在启动时用所有内存节点的所有管理区描述符的地址初始化这个数组。

当内核调用一个内存分配函数时，必须指明请求页框所在的管理区。内核通常指明它愿意使用哪个管理区。例如，如果一个页框必须直接映射在线性地址的第4个GB，但它又不用于ISA DMA的传输，那么，内核不是在ZONE_NORMAL区就是在ZONE_DMA区请求一个页框。当然，如果ZONE_NORMAL没有空闲页框，那么，应该从ZONE_DMA获取页框。为了在内存分配请求中指定首选管理区，内核使用zonelist数据结构，这就是管理区描述符指针数组。

保留的页框池

可以用两种不同的方法来满足内存分配请求。如果有足够的空闲内存可用，请求就会被立刻满足。否则，必须回收一些内存，并且将发出请求的内核控制路径阻塞，直到有内存被释放。

不过，当请求内存时，一些内核控制路径不能被阻塞——例如，这种情况发生在处理中断或在执行临界区内的代码时。在这些情况下，一条内核控制路径应当产生原子内存分配请求（使用GFP_ATOMIC标志；参见稍后的“分区页框分配器”一节）。原子请求从不被阻塞；如果没有足够的空闲页，则仅仅是分配失败而已。

尽管无法保证一个原子内存分配请求决不失败，但是内核会设法尽量减少这种不幸事件发生的可能性。为做到这一点，内核为原子内存分配请求保留了一个页框池，只有在内存不足时才使用。

保留内存的数量（以KB为单位）存放在min_free_kbytes变量中。它的初始值在内核初始化时设置，并取决于直接映射到内核线性地址空间第4个GB的物理内存的数量——也就是说，取决于包含在ZONE_DMA和ZONE_NORMAL内存管理区内的页框数目：

注3： 为索引保留的位的数目取决于内核是否支持NUMA模型以及flags字段的大小。如果不支持NUMA，那么flags字段中管理区索引占两位，节点索引占一位（通常设为0）。在NUMA 32位体系结构上，flags中管理区索引占两位，节点数目占六位。最后，在NUMA 64位体系结构上，64位的flags字段中管理区索引占两位，节点数目占十位。

保留池的大小 = $\lfloor \sqrt{16 \times \text{直接映射内存}} \rfloor$ (KB)

但是，`min_free_kbytes`的初始值不能小于128也不能大于65536（注4）。

ZONE_DMA和ZONE_NORMAL内存管理区将一定数量的页框贡献给保留内存，这个数目与两个管理区的相对大小成比例。例如，如果ZONE_NORMAL管理区比ZONE_DMA大8倍，那么页框的7/8从ZONE_NORMAL获得，而1/8从ZONE_DMA获得。

管理区描述符的`pages_min`字段存储了管理区内保留页框的数目。正如我们将在第十七章看到的，这个字段和`pages_low`、`pages_high`字段一起还在页框回收算法中起作用。`pages_low`字段总是被设为`pages_min`的值的5/4，而`pages_high`总是被设为`pages_min`的值的3/2。

分区页框分配器

被称作分区页框分配器 (*zoned page frame allocator*) 的内核子系统，处理对连续页框组的内存分配请求。它的主要组成如图8-2所示。

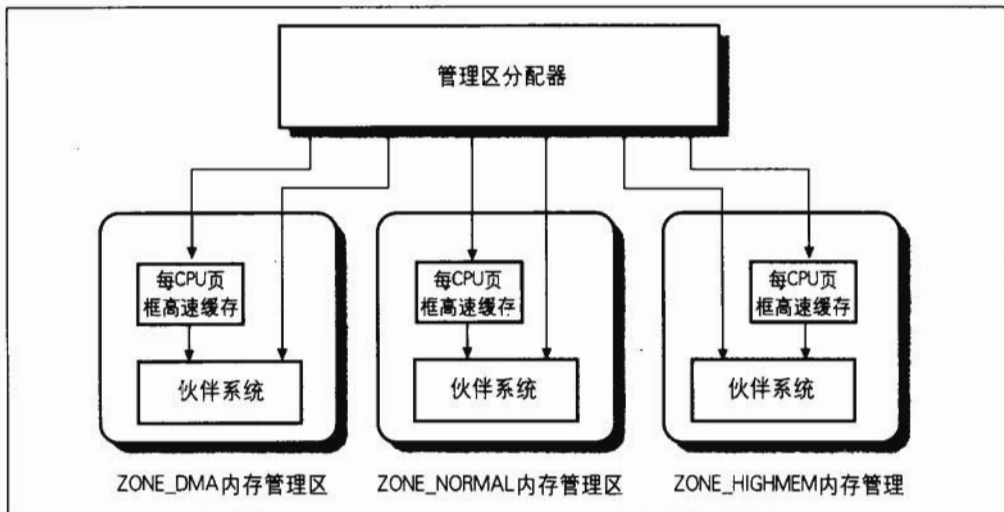


图8-2：分区页框分配器的组成

其中，名为“管理区分配器”部分接受动态内存分配与释放的请求。在请求分配的情况下，该部分搜索一个能满足所请求的一组连续页框内存的管理区（参见后面的“管理区

注4：稍后系统管理员可以通过写入`/proc/sys/vm/min_free_kbytes`文件或通过发出一个适当的`sysctl()`系统调用来更改保留内存的数量。

分配器”一节)。在每个管理区内，页框被名为伙伴系统”(参见后面的“伙伴系统算法”一节)的部分来处理。为达到更好的系统性能，一小部分页框保留在高速缓存中用于快速地满足对单个页框的分配请求(参见后面的“每CPU页框高速缓存”一节)。

请求和释放页框

可以通过6个稍有差别的函数和宏请求页框。除非另作说明，一般情况下，它们都返回第一个所分配页的线性地址，或者如果分配失败，则返回NULL。

`alloc_pages(gfp_mask, order)`

用这个函数请求 2^{order} 个连续的页框。它返回第一个所分配页框描述符的地址，或者如果分配失败，则返回NULL。

`alloc_page(gfp_mask)`

用于获得一个单独页框的宏；它扩展为：

```
alloc_pages(gfp_mask, 0)
```

它返回所分配页框描述符的地址，或者如果分配失败，则返回NULL。

`__get_free_pages(gfp_mask, order)`

该函数类似于`alloc_pages()`，但它返回第一个所分配页的线性地址。

`__get_free_page(gfp_mask)`

用于获得一个单独页框的宏；它扩展为：

```
__get_free_pages(gfp_mask, 0)
```

`get_zeroed_page(gfp_mask)`

函数用来获取填满0的页框；它调用：

```
alloc_pages(gfp_mask | __GFP_ZERO, 0)
```

然后返回所获取页框的线性地址。

`__get_dma_pages(gfp_mask, order)`

用这个宏获得适用于DMA的页框，它扩展为：

```
__get_free_pages(gfp_mask | __GFP_DMA, order)
```

参数`gfp_mask`是一组标志，它指明了如何寻找空闲的页框。能在`gfp_mask`中使用的标志如表8-5所示。

表 8-5: 用于请求页框的标志

标志	说明
<code>__GFP_DMA</code>	所请求的页框必须处于 <code>ZONE_DMA</code> 管理区。等价于 <code>GFP_DMA</code>
<code>__GFP_HIGHMEM</code>	所请求的页框处于 <code>ZONE_HIGHMEM</code> 管理区
<code>__GFP_WAIT</code>	允许内核对等待空闲页框的当前进程进行阻塞
<code>__GFP_HIGH</code>	允许内核访问保留的页框池
<code>__GFP_IO</code>	允许内核在低端内存页上执行 I/O 传输以释放页框
<code>__GFP_FS</code>	如果清 0, 则不允许内核执行依赖于文件系统的操作
<code>__GFP_COLD</code>	所请求的页框可能为“冷的”(参见稍后的“每 CPU 页框高速缓存”一节)
<code>__GFP_NOWARN</code>	一次内存分配失败将不会产生警告信息
<code>__GFP_REPEAT</code>	内核重试内存分配直到成功
<code>__GFP_NOFAIL</code>	与 <code>__GFP_REPEAT</code> 相同
<code>__GFP_NORETRY</code>	一次内存分配失败后不再重试
<code>__GFP_NO_GROW</code>	slab 分配器不允许增大 slab 高速缓存(参见稍后的“slab 分配器”一节)
<code>__GFP_COMP</code>	属于扩展页的页框(参见第二章的“扩展分页”一节)
<code>__GFP_ZERO</code>	任何返回的页框必须被填满 0

实际上, Linux 使用预定义标志值的组合, 如表 8-6 所示, 组名就是你在 6 个页框分配函数中所遇到的参数。

表 8-6: 用于请求页框的一组标志值

组名	相应标志
<code>GFP_ATOMIC</code>	<code>__GFP_HIGH</code>
<code>GFP_NOIO</code>	<code>__GFP_WAIT</code>
<code>GFP_NOFS</code>	<code>__GFP_WAIT</code> <code>__GFP_IO</code>
<code>GFP_KERNEL</code>	<code>__GFP_WAIT</code> <code>__GFP_IO</code> <code>__GFP_FS</code>
<code>GFP_USER</code>	<code>__GFP_WAIT</code> <code>__GFP_IO</code> <code>__GFP_FS</code>
<code>GFP_HIGHUSER</code>	<code>__GFP_WAIT</code> <code>__GFP_IO</code> <code>__GFP_FS</code> <code>__GFP_HIGHMEM</code>

`__GFP_DMA` 和 `__GFP_HIGHMEM` 标志被称作管理区修饰符; 它们标示寻找空闲页框时内核所搜索的管理区。 `contig_page_data` 节点描述符的 `node_zonelists` 字段是一个管

理区描述符链表的数组，它代表后备管理区：对管理区修饰符的每一个设置，相应的链表包含的内存管理区能在原来的管理区缺少页框的情况下被用于满足内存分配请求。在 80x86 UMA 体系结构中，后备管理区如下：

- 如果 `__GFP_DMA` 标志被置位，则只能从 `ZONE_DMA` 内存管理区获取页框。
- 否则，如果 `__GFP_HIGHMEM` 标志没有被置位，则只能按优先次序从 `ZONE_NORMAL` 和 `ZONE_DMA` 内存管理区获取页框。
- 否则（`__GFP_HIGHMEM` 标志被置位），则可以按优先次序从 `ZONE_HIGHMEM`、`ZONE_NORMAL` 和 `ZONE_DMA` 内存管理区获得页框。

下面 4 个函数和宏中的任一个都可以释放页框：

`__free_pages(page, order)`

该函数先检查 `page` 指向的页描述符；如果该页框未被保留（`PG_reserved` 标志为 0），就把描述符的 `count` 字段减 1。如果 `count` 值变为 0，就假定从与 `page` 对应的页框开始的 2^{order} 个连续页框不再被使用。在这种情况下，该函数释放页框，正如后面的“管理区分配器”一节描述的那样。

`free_pages(addr, order)`

这个函数类似于 `__free_pages(page, order)`，但是它接收的参数为要释放的第一个页框的线性地址 `addr`。

`__free_page(page)`

这个宏释放 `page` 所指描述符对应的页框；它扩展为：

```
__free_pages(page, 0)
```

`free_page(addr)`

该宏释放线性地址为 `addr` 的页框。它扩展为：

```
free_pages(addr, 0)
```

高端内存页框的内核映射

与直接映射的物理内存末端、高端内存的始端所对应的线性地址存放在 `high_memory` 变量中，它被设置为 896MB。896MB 边界以上的页框并不映射在内核线性地址空间的第 4 个 GB，因此，内核不能直接访问它们。这就意味着，返回所分配页框线性地址的页分配器函数不适用于高端内存，即不适用于 `ZONE_HIGHMEM` 内存管理区内的页框。

例如，假定内核调用 `__get_free_pages(GFP_HIGHMEM, 0)` 在高端内存分配一个页框。如果分配器在高端内存确实分配了一个页框，那么 `__get_free_pages()` 不能返回它的

线性地址，因为它根本就不存在，因此，函数返回 NULL。依次类推，内核不能使用这个页框；甚至更坏的情况下，也不能释放该页框，因为内核已经丢失了它的踪迹。

在 64 位硬件平台上不存在这个问题，因为可使用的线性地址空间远大于能安装的 RAM 大小，简言之，这些体系结构的 ZONE_HIGHMEM 管理区总是空的。但是在 32 位平台上，如 80x86 体系结构，Linux 设计者不得不找到某种方法来允许内核使用所有可用的 RAM，达到 PAE 所支持的 64GB。采用的方法如下：

- 高端内存页框的分配只能通过 alloc_pages() 函数和它的快捷函数 alloc_page()。这些函数不返回第一个被分配页框的线性地址，因为如果该页框属于高端内存，那么这样的线性地址根本不存在。取而代之，这些函数返回第一个被分配页框的页描述符的线性地址。这些线性地址总是存在的，因为所有页描述符一旦被分配在低端内存中，它们在内核初始化阶段就不会改变。
- 没有线性地址的高端内存中的页框不能被内核访问。因此，内核线性地址空间的最后 128MB 中的一部分专门用于映射高端内存页框。当然，这种映射是暂时的，否则只有 128MB 的高端内存可以被访问。取而代之，通过重复使用线性地址，使得整个高端内存能够在不同的时间被访问。

内核可以采用三种不同的机制将页框映射到高端内存，分别叫做永久内核映射、临时内核映射及非连续内存分配。在本节中，我们集中于前两种技术；第三种技术将在本章后面“非连续内存区管理”一节进行讨论。

建立永久内核映射可能阻塞当前进程；这发生在空闲页表项不存在时，也就是在高端内存上没有页表项可以用作页框的“窗口”时。因此，永久内核映射不能用于中断处理程序和可延迟函数。相反，建立临时内核映射决不会要求阻塞当前进程；不过，它的缺点是只有很少的临时内核映射可以同时建立起来。

使用临时内核映射的内核控制路径必须保证当前没有其他的内核控制路径在使用同样的映射。这意味着内核控制路径永远不能被阻塞，否则其他内核控制路径有可能使用同一个窗口来映射其他的高端内存页。

当然，这些技术中没有一种可以确保对整个 RAM 同时进行寻址。毕竟，只有 128MB 的线性地址留给映射高端内存，尽管 PAE 支持系统高达 64GB RAM。

永久内核映射

永久内核映射允许内核建立高端页框到内核地址空间的长期映射。它们使用主内核页表中一个专门的页表，其地址存放在 pkmap_page_table 变量中。页表中的表项数由 LAST_PKMAP 宏产生。页表照样包含 512 或 1024 项，这取决于 PAE 是否被激活[参见

第二章“物理地址扩展 (PAE) 分页机制”一节]；因此，内核一次最多访问 2MB 或 4MB 的高端内存。

该页表映射的线性地址从 PKMAP_BASE 开始。pkmap_count 数组包含 LAST_PKMAP 个计数器，pkmap_page_table 页表中的每一项都有一个。我们区分以下三种情况：

计数器为 0

对应的页表项没有映射任何高端内存页框，并且是可用的。

计数器为 1

对应的页表项没有映射任何高端内存页框，但是它不能使用，因为自从它最后一次使用以来，其相应的 TLB 表现还未被刷新。

计数器为 n (远大于 1)

相应的页表项映射一个高端内存页框，这意味着正好有 $n-1$ 个内核成分在使用这个页框。

为了记录高端内存页框与永久内核映射包含的线性地址之间的联系，内核使用了 page_address_hhtable 散列表。该表包含一个 page_address_map 数据结构，用于为高端内存中的每一个页框进行当前映射。而该数据结构还包含一个指向页描述符的指针和分配给该页框的线性地址。

page_address() 函数返回页框对应的线性地址，如果页框在高端内存中并且没有被映射，则返回 NULL。这个函数接受一个页描述符指针 page 作为其参数，并区分以下两种情况：

1. 如果页框不在高端内存中 (PG_highmem 标志为 0)，则线性地址总是存在并且是通过计算页框下标，然后将其转换成物理地址，最后根据相应的物理地址得到线性地址。这是由下面的代码完成的：

```
__va((unsigned long)(page - mem_map) << 12)
```

2. 如果页框在高端内存 (PG_highmem 标志为 1) 中，该函数就到 page_address_hhtable 散列表中查找。如果在散列表中找到页框，page_address() 就返回它的线性地址，否则返回 NULL。

kmap() 函数建立永久内核映射。本质上它等价于下列代码：

```
void * kmap(struct page * page)
{
    if (!PageHighMem(page))
        return page_address(page);
    return kmap_high(page);
}
```

如果页框确实属于高端内存，则调用 `kmap_high()` 函数。这个函数本质上等价于下列代码：

```
void * kmap_high(struct page * page)
{
    unsigned long vaddr;
    spin_lock(&kmap_lock);
    vaddr = (unsigned long) page->virtual;
    if (!vaddr)
        vaddr = map_new_virtual(page);
    pkmap_count[(vaddr-PKMAP_BASE) >> PAGE_SHIFT]++;
    spin_unlock(&kmap_lock);
    return (void *) vaddr;
}
```

该函数获取 `kmap_lock` 自旋锁，以保护页表免受多处理器系统上的并发访问。注意，没有必要禁止中断，因为中断处理程序和可延迟函数不能调用 `kmap()`。接下来，`kmap_high()` 函数检查页框是否已经通过调用 `page_address()` 被映射。如果不是，该函数调用 `map_new_virtual()` 函数把页框的物理地址插入到 `pkmap_page_table` 的一个项中并在 `page_address_hhtable` 散列表中加入一个元素。然后，`kmap_high()` 使页框的线性地址所对应的计数器加 1 来将调用该函数的新内核成分考虑在内。最后，`kmap_high()` 释放 `kmap_lock` 自旋锁并返回对该页框进行映射的线性地址。

`map_new_virtual()` 函数本质上执行两个嵌套循环：

```
for (;;) {
    int count;
    DECLARE_WAITQUEUE(wait, current);
    for (count = LAST_PKMAP; count > 0; --count) {
        last_pkmap_nr = (last_pkmap_nr + 1) & (LAST_PKMAP - 1);
        if (!last_pkmap_nr) {
            flush_all_zero_pkmaps();
            count = LAST_PKMAP;
        }
        if (!pkmap_count[last_pkmap_nr]) {
            unsigned long vaddr = PKMAP_BASE +
                (last_pkmap_nr << PAGE_SHIFT);
            set_pte(&(pkmap_page_table[last_pkmap_nr]),
                mk_pte(page, _pgprot(0x63)));
            pkmap_count[last_pkmap_nr] = 1;
            set_page_address(page, (void *) vaddr);
            return vaddr;
        }
    }
    current->state = TASK_UNINTERRUPTIBLE;
    add_wait_queue(&pkmap_map_wait, &wait);
    spin_unlock(&kmap_lock);
    schedule();
    remove_wait_queue(&pkmap_map_wait, &wait);
    spin_lock(&kmap_lock);
    if (page_address(page))
```

```

        return (unsigned long) page_address(page);
    }

```

在内循环中，该函数扫描 `pkmap_count` 中的所有计数器直到找到一个空值。当在 `pkmap_count` 中找到了一个未使用的项时，大的 `if` 代码块运行。这段代码确定该项对应的线性地址，为它在 `pkmap_page_table` 页表中创建一个项，将 `count` 置 1，因为该项现在已经被使用了，调用 `set_page_address()` 函数插入一个新元素到 `page_address_htable` 散列表中，并返回线性地址。

函数从上次停止的地方开始，穿越 `pkmap_count` 数组执行循环。这是函数通过将 `pkmap_page_table` 页表中上次使用过页表项的索引保存在一个名为 `last_pkmap_nr` 的变量中做到的。因此，搜索从上次因调用 `map_new_virtual()` 函数而跳出的地方重新开始。

当在 `pkmap_count` 中搜索到最后一个计数器时，就又从下标为 0 的计数器重新开始搜索。不过，在继续之前，`map_new_virtual()` 调用 `flush_all_zero_pkmaps()` 函数来开始寻找计数器为 1 的另一趟扫描。每个值为 1 的计数器都表示在 `pkmap_page_table` 页表中表项是空闲的，但不能使用，因为相应的 TLB 表项还没有被刷新。`flush_all_zero_pkmaps()` 把它们的计数器重置为 0，删除 `page_address_htable` 散列表中对应的元素，并在 `pkmap_page_table` 的所有项上进行 TLB 刷新。

如果内循环在 `pkmap_count` 中没有找到空的计数器，`map_new_virtual()` 函数就阻塞当前进程，直到某个进程释放了 `pkmap_page_table` 页表中的一个表项。通过把 `current` 插入到 `pkmap_map_wait` 等待队列，把 `current` 状态设置为 `TASK_UNINTERRUPTIBLE` 并调用 `schedule()` 放弃 CPU 来达到此目的。一旦进程被唤醒，该函数就通过调用 `page_address()` 检查是否存在另一个进程已经映射了该页；如果还没有其他进程映射该页，则内循环重新开始。

`kunmap()` 函数撤销先前由 `kmap()` 建立的永久内核映射。如果页确实在高端内存中，则调用 `kunmap_high()` 函数，它本质上等价于下列代码：

```

void kunmap_high(struct page * page)
{
    spin_lock(&kmap_lock);
    if (--pkmap_count[((unsigned long)page_address(page)
                      -PKMAP_BASE)>>PAGE_SHIFT]) == 1)
        if (waitqueue_active(&pkmap_map_wait))
            wake_up(&pkmap_map_wait);
    spin_unlock(&kmap_lock);
}

```

中括号内的表达式从页的线性地址计算出 `pkmap_count` 数组的索引。计数器被减 1 并与 1 相比。匹配成功表明没有进程在使用页。该函数最终能唤醒由 `map_new_virtual()` 添加在等待队列中的进程（如果有的话）。

临时内核映射

临时内核映射比永久内核映射的实现要简单；此外，它们可以用在中断处理程序和可延迟函数的内部，因为它们从不阻塞当前进程。

在高端内存的任一页框都可以通过一个“窗口”（为此而保留的一个页表项）映射到内核地址空间。留给临时内核映射的窗口数是非常少的。

每个CPU都有它自己的包含13个窗口的集合，它们用enum km_type数据结构表示。该数据结构中定义的每个符号，如KM_BOUNCE_READ、KM_USER0或KM_PTE0，标识了窗口的线性地址。

内核必须确保同一窗口永不会被两个不同的控制路径同时使用。因此，km_type结构中的每个符号只能由一种内核成分使用，并以该成分命名。最后一个符号KM_TYPE_NR本身并不表示一个线性地址，但由每个CPU用来产生不同的可用窗口数。

在km_type中的每个符号（除了最后一个）都是固定映射的线性地址的一个下标（参见第二章“固定映射的线性地址”一节）。enum fixed_addresses数据结构包含符号FIX_KMAP_BEGIN和FIX_KMAP_END；把后者赋给下标FIX_KMAP_BEGIN+(KM_TYPE_NR*NR_CPUS)-1。在这种方式下，系统中的每个CPU都有KM_TYPE_NR个固定映射的线性地址。此外，内核用fix_to_virt(FIX_KMAP_BEGIN)线性地址对应的页表项的地址初始化kmap_pte变量。

为了建立临时内核映射，内核调用kmap_atomic()函数，它本质上等价于下列代码：

```
void * kmap_atomic(struct page * page, enum km_type type)
{
    enum fixed_addresses idx;
    unsigned long vaddr;

    current_thread_info()->preempt_count++;
    if (!PageHighMem(page))
        return page_address(page);
    idx = type + KM_TYPE_NR * smp_processor_id();
    vaddr = fix_to_virt(FIX_KMAP_BEGIN + idx);
    set_pte(kmap_pte-idx, mk_pte(page, 0x063));
    __flush_tlb_single(vaddr);
    return (void *) vaddr;
}
```

type参数和CPU标识符（通过smp_processor_id()）指定必须用哪个固定映射的线性地址映射请求页。如果页框不属于高端内存，则该函数返回页框的线性地址；否则，用页的物理地址及Present、Accessed、Read/Write和Dirty位建立该固定映射的线性地址对应的页表项。最后，该函数刷新适当的TLB项并返回线性地址。

为了撤销临时内核映射，内核使用 `kunmap_atomic()` 函数。在 80x86 结构中，这个函数减少当前进程的 `preempt_count`，因此，如果在请求临时内核映像之前能抢占内核控制路径，那么在同一个映射被撤销后可以再次抢占。此外，`kunmap_atomic()` 检查当前进程的 `TIF_NEED_RESCHED` 标志是否被置位，如果是，就调用 `schedule()`。

伙伴系统算法

内核应该为分配一组连续的页框而建立一种健壮、高效的分配策略。为此，必须解决著名的内存管理问题，也就是所谓的外碎片 (*external fragmentation*)。频繁地请求和释放不同大小的一组连续页框，必然导致在已分配页框的块内分散了许多小块的空闲页框。由此带来的问题是，即使有足够的空闲页框可以满足请求，但要分配一个大块的连续页框就可能无法满足。

从本质上说，避免外碎片的方法有两种：

- 利用分页单元把一组非连续的空闲页框映射到连续的线性地址区间。
- 开发一种适当的技术来记录现存的空闲连续页框块的情况，以尽量避免为满足对小块的请求而分割大的空闲块。

基于以下三种原因，内核首选第二种方法：

- 在某些情况下，连续的页框确实是必要的，因为连续的线性地址不足以满足请求。一个典型的例子就是给 DMA 处理器分配缓冲区的内存请求（参见第十三章）。因为当在一次单独的 I/O 操作中传送几个磁盘扇区的数据时，DMA 忽略分页单元而直接访问地址总线，因此，所请求的缓冲区就必须位于连续的页框中。
- 即使连续页框的分配并不是很必要，但它在保持内核页表不变方面所起的作用也是不容忽视的。修改页表会怎样呢？从第二章我们知道，频繁地修改页表势必导致平均访问内存次数的增加，因为这会使 CPU 频繁地刷新转换后援缓冲器 (TLB) 的内容。
- 内核通过 4MB 的页可以访问大块连续的物理内存。这样减少了转换后援缓冲器的失效率，因此提高了访问内存的平均速度 [参见第二章“转换后援缓冲器 (TLB)”一节]。

Linux 采用著名的伙伴系统 (*buddy system*) 算法来解决外碎片问题。把所有的空闲页框分组为 11 个块链表，每个块链表分别包含大小为 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 和 1024 个连续的页框。对 1024 个页框的最大请求对应着 4MB 大小的连续 RAM 块。每个块的第一个页框的物理地址是该块大小的整数倍。例如，大小为 16 个页框的块，其起始地址是 16×2^{12} ($2^{12} = 4096$ ，这是一个常规页的大小) 的倍数。

我们通过一个简单的例子来说明该算法的工作原理。

假设要请求一个256个页框的块（即1MB）。算法先在256个页框的链表中检查是否有一个空闲块。如果没有这样的块，算法会查找下一个更大的页框，也就是，在512个页框的链表中找一个空闲块。如果存在这样的块，内核就把256的页框分成两等份，一半用作满足请求，另一半插入到256个页框的链表中。如果在512个页框的块链表中也没找到空闲块，就继续找更大的块——1024个页框的块。如果这样的块存在，内核把1024个页框块的256个页框用作请求，然后从剩余的768个页框中拿512个插入到512个页框的链表中，再把最后的256个插入到256个页框的链表中。如果1024个页框的链表还是空的，算法就放弃并发出错误信号。

以上过程的逆过程就是页框块的释放过程，也是该算法名字的由来。内核试图把大小为 b 的一对空闲伙伴块合并为一个大小为 $2b$ 的单独块。满足以下条件的两个块称为伙伴：

- 两个块具有相同的大小，记作 b 。
- 它们的物理地址是连续的。
- 第一块的第一个页框的物理地址是 $2 \times b \times 2^{12}$ 的倍数。

该算法是迭代的，如果它成功合并所释放的块，它会试图合并 $2b$ 的块，以再次试图形成更大的块。

数据结构

Linux 2.6为每个管理区使用不同的伙伴系统。因此，在80x86结构中，有三种伙伴系统：第一种处理适合ISA DMA的页框，第二种处理“常规”页框，第三种处理高端内存页框。每个伙伴系统使用的主要数据结构如下：

- 前面介绍过的`mem_map`数组。实际上，每个管理区都关系到`mem_map`元素的子集。子集中的第一个元素和元素的个数分别由管理区描述符的`zone_mem_map`和`size`字段指定。
- 包含有11个元素、元素类型为`free_area`的一个数组，每个元素对应一种块大小。该数组存放在管理区描述符的`free_area`字段中。

我们考虑管理区描述符中`free_area`数组的第 k 个元素，它标识所有大小为 2^k 的空闲块。这个元素的`free_list`字段是双向循环链表的头，这个双向循环链表集中了大小为 2^k 页的空闲块对应的页描述符。更精确地说，该链表包含每个空闲页框块（大小为 2^k ）的起始页框的页描述符，指向链表中相邻元素的指针存放在页描述符的`lru`字段中（注5）。

注5：正如我们稍后将看到的，当页不空闲时页描述符的`lru`字段可被用于其他目的。

除了链表头外，`free_area`数组的第 k 个元素同样包含字段`nr_free`，它指定了大小为 2^k 页的空闲块的个数。当然，如果没有大小为 2^k 的空闲页框块，则`nr_free`等于0且`free_list`为空（`free_list`的两个指针都指向它自己的`free_list`字段）。

最后，一个 2^k 的空闲页块的第一个页的描述符的`private`字段存放了块的`order`，也就是数字 k 。正是由于这个字段，当页块被释放时，内核可以确定这个块的伙伴是否也空闲，如果是的话，它可以把两个块结合成大小为 2^{k+1} 页的单一块。应当注意的是，直到Linux 2.6.10，内核使用了10组标志来对这种信息进行编码。

分配块

`__rmqueue()`函数用来在管理区中找到一个空闲块。该函数需要两个参数：管理区描述符的地址和`order`，`order`表示请求的空闲页块大小的对数值（0表示一个单页块，1表示一个两页块，依次类推）。如果页框被成功分配，`__rmqueue()`函数就返回第一个被分配页框的页描述符。否则，函数返回NULL。

`__rmqueue()`函数假设调用者已经禁止了本地中断并获得了保护伙伴系统数据结构的`zone->lock`自旋锁。从所请求`order`的链表开始，它扫描每个可用块链表（由不指向自己的链表项表示）进行循环搜索，如果需要搜索更大的`order`，就继续搜索：

```
struct free_area *area;
unsigned int current_order;

for (current_order=order; current_order<11; ++current_order) {
    area = zone->free_area + current_order;
    if (!list_empty(&area->free_list))
        goto block_found;
}
return NULL;
```

如果直到循环结束还没有找到合适的空闲块，那么`__rmqueue()`就返回NULL。否则，找到了一个合适的空闲块，在这种情况下，从链表中删除它的第一个页框描述符，并减少管理区描述符中的`free_pages`的值：

```
block_found:
    page = list_entry(area->free_list.next, struct page, lru);
    list_del(&page->lru);
    ClearPagePrivate(page);
    page->private = 0;
    area->nr_free--;
    zone->free_pages -= 1UL << order;
```

如果从`curr_order`链表中找到的块大于请求的`order`，就执行一个while循环。这几行代码蕴含的原理如下：当为了满足 2^h 个页框的请求而有必要使用 2^k 个页框的块时（ $h <$

k)，程序就分配前面的 2^h 个页框，而把后面 $2^k - 2^h$ 个页框循环再分配给 `free_area` 链表中下标在 h 到 k 之间的元素：

```

size = 1 << curr_order;
while (curr_order > order) {
    area--;
    curr_order--;
    size >= 1;
    buddy = page + size;
    /* 插入伙伴作为链表中第一个元素 */
    list_add(&buddy->lru, &area->free_list);
    area->nr_free++;
    buddy->private = curr_order;
    SetPagePrivate(buddy);
}
return page;

```

因为 `__rmqueue()` 函数已经找到了合适的空闲块，所以它返回所分配的第一个页框对应的页描述符的地址 `page`。

释放块

`__free_pages_bulk()` 函数按照伙伴系统的策略释放页框。它使用 3 个基本输入参数 (注 6)：

`page`

被释放块中所包含的第一个页框描述符的地址。

`zone`

管理区描述符的地址。

`order`

块大小的对数。

该函数假设调用者已经禁止本地中断并获得了保护伙伴系统数据结构的 `zone->lock` 自旋锁。`__free_pages_bulk()` 首先声明和初始化一些局部变量：

```

struct page * base = zone->zone_mem_map;
unsigned long buddy_idx, page_idx = page - base;
struct page * buddy, * coalesced;
int order_size = 1 << order;

```

`page_idx` 局部变量包含块中第一个页框的下标，这是相对于管理区中的第一个页框而言的。

注 6： 由于性能的原因，这个内联函数还使用另一个参数；但是它的值可以由正文中说明的 3 个基本参数决定。

`order_size` 局部变量用于增加管理区中空闲页框的计数器：

```
zone->free_pages += order_size;
```

现在函数开始执行循环，最多循环 $(10 - \text{order})$ 次，每次都尽量把一个块和它的伙伴进行合并。函数以最小的块开始，然后向上移动到顶部：

```
while (order < 10) {
    buddy_idx = page_idx ^ (1 << order);
    buddy = base + buddy_idx;
    if (!page_is_buddy(buddy, order))
        break;
    list_del(&buddy->lru);
    zone->free_area[order].nr_free--;
    ClearPagePrivate(buddy);
    buddy->private = 0;
    page_idx &= buddy_idx;
    order++;
}
```

在循环体内，函数寻找块的下标 `buddy_idx`，它是拥有 `page_idx` 页描述符下标的块的伙伴。结果这个下标可以被简单地如下计算：

```
buddy_idx = page_idx ^ (1 << order);
```

实际上，使用 $(1 < \text{order})$ 掩码的异或 (XOR) 转换 `page_idx` 第 `order` 位的值。因此，如果这个位原先是 0，`buddy_idx` 就等于 `page_idx + order_size`；相反，如果这个位原先是 1，`buddy_idx` 就等于 `page_idx - order_size`。

一旦知道了伙伴块下标，就可以通过下式很容易地获得伙伴块的页描述符：

```
buddy = base + buddy_idx;
```

现在函数调用 `page_is_buddy()` 来检查 `buddy` 是否描述了大小为 `order_size` 的空闲页框块的第一个页。

```
int page_is_buddy(struct page *page, int order)
{
    if (PagePrivate(buddy) && page->private == order &&
        !PageReserved(buddy) && page_count(page) == 0)
        return 1;
    return 0;
}
```

正如所见，`buddy` 的第一个页必须为空闲 (`_count` 字段等于 -1)，它必须属于动态内存 (PG_reserved 位清零)，它的 `private` 字段必须有意义 (PG_private 位置位)，最后 `private` 字段必须存放将要被释放的块的 `order`。

如果所有这些条件都符合，伙伴块就被释放，并且函数将它从以 order 排序的空闲块链表上删除，并再执行一次循环以寻找两倍大小的伙伴块。

如果 page_is_buddy() 中至少有一个条件没有被满足，则该函数跳出循环，因为获得的空闲块不能再和其他空闲块合并。函数将它插入适当的链表并以块大小的 order 更新第一个页框的 private 字段。

```
coalesced = base + page_idx;
coalesced->private = order;
SetPagePrivate(coalesced);
list_add(&coalesced->lru, &zone->free_area[order].free_list);
zone->free_area[order].nr_free++;
```

每 CPU 页框高速缓存

正如我们将在本章稍后看到的，内核经常请求和释放单个页框。为了提升系统性能，每个内存管理区定义了一个“每 CPU”页框高速缓存。所有“每 CPU”高速缓存包含一些预先分配的页框，它们被用于满足本地 CPU 发出的单一内存请求。

实际上，这里为每个内存管理区和每个 CPU 提供了两个高速缓存：一个热高速缓存，它存放的页框中所包含的内容很可能就在 CPU 硬件高速缓存中；还有一个冷高速缓存。

如果内核或用户态进程在刚分配到页框后就立即向页框写，那么从热高速缓存中获得页框就对系统性能有利。实际上，每次对页框存储单元的访问将都会导致从另一个页框中给硬件高速缓存“窃取”一行——当然，除非硬件高速缓存包含有一行：它映射刚被访问的“热”页框单元。

反过来，如果页框将要被 DMA 操作填充，那么从冷高速缓存中获得页框是方便的。在这种情况下，不会涉及到 CPU，并且硬件高速缓存的行不会被修改。从冷高速缓存获得页框为其他类型的内存分配保存了热页框储备。

实现每 CPU 页框高速缓存的主要数据结构是存放在内存管理区描述符的 pageset 字段中的一个 per_cpu_pageset 数组数据结构。该数组包含为每个 CPU 提供的一个元素；这个元素依次由两个 per_cpu_pages 描述符组成，一个留给热高速缓存而另一个留给冷高速缓存。per_cpu_pages 描述符的字段在表 8-7 中列出。

表 8-7: per_cpu_pages 描述符的字段

类型	名称	描述
int	count	高速缓存中的页框个数
int	low	下界，表示高速缓存需要补充

表 8-7: per_cpu_pages 描述符的字段 (续)

类型	名称	描述
int	high	上界, 表示高速缓存用尽
int	batch	在高速缓存中将要添加或被删去的页框个数
strut list_head	list	高速缓存中包含的页框描述符链表

内核使用两个位标来监视热高速缓存和冷高速缓存的大小: 如果页框个数低于下界 low, 内核通过从伙伴系统中分配 batch 个单一页框来补充对应的高速缓存; 否则, 如果页框个数高过上界 high, 内核从高速缓存中释放 batch 个页框到伙伴系统中。值 batch、low 和 high 本质上取决于内存管理区中包含的页框个数。

通过每 CPU 页框高速缓存分配页框

buffered_rmqueue() 函数在指定的内存管理区中分配页框。它使用每 CPU 页框高速缓存来处理单一页框请求。

参数为内存管理区描述符的地址, 请求分配的内存大小的对数 order, 以及分配标志 gfp_flags。如果 gfp_flags 中的 __GFP_COLD 标志被置位, 那么页框应当从冷高速缓存中获取, 否则它应从热高速缓存中获取 (此标志只对单一页框请求有意义)。该函数本质上执行如下操作:

1. 如果 order 不等于 0, 每 CPU 页框高速缓存就不能被使用: 函数跳到第 4 步。
2. 检查由 __GFP_COLD 标志所标识的内存管理区本地每 CPU 高速缓存是否需要补充 (per_cpu_pages 描述符的 count 字段小于或等于 low 字段)。在这种情况下, 它执行如下子步骤:
 - a. 通过反复调用 __rmqueue() 函数从伙伴系统中分配 batch 个单一页框。
 - b. 将已分配页框的描述符插入高速缓存链表中。
 - c. 通过给 count 增加实际被分配页框的个数来更新它。
3. 如果 count 值为正, 则函数从高速缓存链表获得一个页框, count 减 1 并跳到第 5 步。(注意, 每 CPU 页框高速缓存有可能为空, 当在第 2a 步调用 __rmqueue() 函数而分配页框失败时就会发生这种情况。)
4. 到这里, 内存请求还没有被满足, 或者是因为请求跨越了几个连续页框, 或者是因为被选中的页框高速缓存为空。调用 __rmqueue() 函数从伙伴系统中分配所请求的页框。

5. 如果内存请求得到满足, 函数就初始化(第一个)页框的页描述符: 清除一些标志, 将 `private` 字段置 0, 并将页框引用计数器置 1。此外, 如果 `gfp_flags` 中的 `__GPF_ZERO` 标志被置位, 则函数将被分配的内存区域填充 0。
6. 返回(第一个)页框的页描述符地址, 如果内存分配请求失败则返回 `NULL`。

释放页框到每 CPU 页框高速缓存

为了释放单个页框到每 CPU 页框高速缓存, 内核使用 `free_hot_page()` 和 `free_cold_page()` 函数。它们都是 `free_hot_cold_page()` 函数的简单封装, 接收的参数为将要释放的页框的描述符地址 `page` 和 `cold` 标志(指定是热高速缓存还是冷高速缓存)。

`free_hot_cold_page()` 函数执行如下操作:

1. 从 `page->flags` 字段获取包含该页框的内存管理区描述符地址 [参见前面的“非一致内存访问 (NUMA)”一节]。
2. 获取由 `cold` 标志选择的管理区高速缓存的 `per_cpu_pages` 描述符的地址。
3. 检查高速缓存是否应该被清空: 如果 `count` 值高于或等于 `high`, 则调用 `free_pages_bulk()` 函数, 将管理区描述符、将被释放的页框个数 (`batch` 字段)、高速缓存链表的地址以及数字 0 (为 0 到 `order` 个页框) 传递给该函数。 `free_pages_bulk()` 函数依次反复调用 `__free_pages_bulk()` 函数来释放指定数量的 (从高速缓存链表获得的) 页框到内存管理区的伙伴系统中。
4. 把释放的页框添加到高速缓存链表上, 并增加 `count` 字段。

应该注意的是, 在当前的 Linux 2.6 内核版本中, 从没有页框被释放到冷高速缓存中: 至于硬件高速缓存, 内核总是假设被释放的页框是热的。当然, 这并不意味着冷高速缓存是空的: 当达到下界时通过 `buffered_rmqueue()` 补充冷高速缓存。

管理区分配器

管理区分配器是内核页框分配器的前端。该成分必须分配一个包含足够多空闲页框的内存管理区, 使它能满足内存请求。这个任务并不像第一眼看上去那么简单, 因为管理区分配器必须满足几个目标:

- 它应当保护保留的页框池 (参见前面的“保留的页框池”一节)。
- 当内存不足且允许阻塞当前进程时, 它应当触发页框回收算法 (参见第十七章), 一旦某些页框被释放, 管理区分配器将再次尝试分配。
- 如果可能, 它应当保存小而珍贵的 `ZONE_DMA` 内存管理区。例如, 如果是对

ZONE_NORMAL或ZONE_HIGHMEM页框的请求,那么管理区分配器会不太愿意分配ZONE_DMA内存管理区中的页框。

我们在前面的“分区页框分配器”一节中已经看到,对一组连续页框的每次请求实质上是通过执行alloc_pages宏来处理的。接着,这个宏又依次调用__alloc_pages()函数,该函数是管理区分配器的核心。它接收以下3个参数:

gfp_mask

在内存分配请求中指定的标志(参见前面的表8-5)。

order

将要分配的一组连续页框数量的对数(即要分配 2^{order} 个连续的页框)。

zonelist

指向zonelist数据结构的指针,该数据结构按优先次序描述了适于内存分配的内存管理区。

__alloc_pages()函数扫描包含在zonelist数据结构中的每个内存管理区。实现代码如下:

```

    for (i = 0; (z=zonelist->zones[i]) != NULL; i++) {
        if (zone_watermark_ok(z, order, ...)) {
            page = buffered_rmqueue(z, order, gfp_mask);
            if (page)
                return page;
        }
    }

```

对于每个内存管理区,该函数将空闲页框的个数与一个阈值作比较,该阈值取决于内存分配标志、当前进程的类型以及管理区被函数检查过的次数。实际上,如果空闲内存不足,那么每个内存管理区一般会被检查几遍,每一遍在所请求的空闲内存最低量的基础上使用更低的阈值扫描。因此前面一段代码在__alloc_pages()函数体内被复制了几次,每次变化很小。在前面的“每CPU页框高速缓存”一节中已经对buffered_rmqueue()函数作了描述:它返回第一个被分配的页框的页描述符;如果内存管理区没有所请求大小的一组连续页框,则返回NULL。

zone_watermark_ok()辅助函数接收几个参数,它们决定内存管理区中空闲页框个数的阈值min。特别是,如果满足下列两个条件则该函数返回值1:

1. 除了被分配的页框外,在内存管理区中至少还有min个空闲页框,不包括为内存不足保留的页框(管理区描述符的lowmem_reserve字段)。
2. 除了被分配的页框外,这里在order至少为k的块中起码还有 $\text{min}/2^k$ 个空闲页框,

其中，对于每个 k ，取值在 1 和分配的 $order$ 之间。因此，如果 $order$ 大于 0，那么在大小至少为 2 的块中起码还有 $min/2$ 个空闲页框；如果 $order$ 大于 1，那么在大小至少为 4 的块中起码还有 $min/4$ 个空闲页框；依此类推。

阈值 min 的值由 `zone_watermark_ok()` 确定，具体如下：

- 作为函数参数被传递的基本值可以是内存管理区界值 `pages_min`、`pages_low` 和 `pages_high` 中的任意一个（参见本章前面的“保留的页框池”一节）。
- 如果作为参数传递的 `gfp_high` 标志被置位，那么 `base` 值被 2 除。通常，如果 `gfp_mask` 中的 `__GFP_WAIT` 标志被置位（也就是说，如果能从高端内存中分配页框），则这个标志等于 1。
- 如果作为参数传递的 `can_try_harder` 标志被置位，则阈值将会再减少四分之一。如果 `gfp_mask` 中的 `__GFP_WAIT` 标志被置位，或者如果当前进程是一个实时进程并且在进程上下文中（在中断处理程序和可延迟函数之外）已经完成了内存分配，则 `can_try_harder` 标志等于 1。

`__alloc_pages()` 函数本质上执行如下步骤：

1. 执行对内存管理区的第一次扫描（参见前面列出的代码）。在第一次扫描中，阈值 min 被设为 `z->pages_low`，其中的 z 指向正在被分析的管理区描述符（参数 `can_try_harder` 和 `gfp_high` 被设为 0）。
2. 如果函数在上一步没有终止，那么没有剩下多少空闲内存：函数唤醒 `kswapd` 内核线程来异步地开始回收页框（参见第十七章）。
3. 执行对内存管理区的第二次扫描，将值 `z->pages_min` 作为阈值 `base` 传递。正如前面解释的，实际阈值由 `can_try_harder` 和 `gfp_high` 标志决定。这一步与第 1 步相似，但该函数使用了较低的阈值。
4. 如果函数在上一步没有终止，那么系统内存肯定不足。如果产生内存分配请求的内核控制路径不是一个中断处理程序或一个可延迟函数，并且它试图回收页框（或者是 `current` 的 `PF_MEMALLOC` 标志被置位，或者是它的 `PF_MEMDIE` 标志被置位），那么函数随即执行对内存管理区的第三次扫描，试图分配页框并忽略内存不足的阈值，也就是说，不调用 `zone_watermark_ok()`。唯有在这种情况下才允许内核控制路径耗用为内存不足预留的页（由管理区描述符的 `lowmem_reserve` 字段指定）。其实，在这种情况下产生内存请求的内核控制路径最终将试图释放页框，因此只要有可能它就应当得到它所请求的。如果没有任何内存管理区包含足够的页框，函数就返回 `NULL` 来提示调用者发生了错误。
5. 在这里，正在调用的内核控制路径并没有试图回收内存。如果 `gfp_mask` 的

`__GFP_WAIT`标志没有被置位，函数就返回 `NULL` 来提示该内核控制路径内存分配失败：在这种情况下，如果不阻塞当前进程就没有办法满足请求。

6. 在这里当前进程能够被阻塞：调用 `cond_resched()` 检查是否有其它的进程需要 CPU。
7. 设置 `current` 的 `PF_MEMALLOC` 标志来表示进程已经准备好执行内存回收。
8. 将一个指向 `reclaim_state` 数据结构的指针存入 `current->reclaim_state`。这个数据结构只包含一个字段 `reclaimed_slab`，被初始化为 0（我们将在本章后面的“slab 分配器与分区页框分配器的接口”一节看到如何使用这个字段）。
9. 调用 `try_to_free_pages()` 寻找一些页框来回收（参见第十七章的“内存紧缺回收”一节）。后一个函数可能阻塞当前进程。一旦函数返回，`__alloc_pages()` 就重设 `current` 的 `PF_MEMALLOC` 标志并再次调用 `cond_resched()`。
10. 如果上一步已经释放了一些页框，那么该函数还要执行一次与第 3 步相同的内存管理区扫描。如果内存分配请求不能被满足，那么函数决定是否应当继续扫描内存管理区：如果 `__GFP_NORETRY` 标志被清除，并且内存分配请求跨越了多达 8 个页框或 `__GFP_REPEAT` 和 `__GFP_NOFAIL` 标志其中之一被置位，那么函数就调用 `blk_congestion_wait()` 使进程休眠一会儿（参见第十四章），并且跳回到第 6 步。否则，函数返回 `NULL` 来提示调用者内存分配失败了。
11. 如果在第 9 步中没有释放任何页框，就意味着内核遇到很大的麻烦，因为空闲页框已经少到了危险的地步，并且不可能回收任何页框。也许到了该作出重要决定的时候了。如果允许内核控制路径执行依赖于文件系统的操作来杀死一个进程（`gfp_mask` 中的 `__GFP_FS` 标志被置位）并且 `__GFP_NORETRY` 标志为 0，那么执行如下子步骤：
 - a. 使用等于 `z->pages_high` 的阈值再一次扫描内存管理区。
 - b. 调用 `out_of_memory()` 通过杀死一个进程开始释放一些内存（参见第十七章的“内存不足删除程序”一节）。
 - c. 跳回第 1 步。

因为第 11a 步使用的阈值远比前面扫描时使用的阈值要高，所以这个步骤很容易失败。实际上，只有当另一个内核控制路径已经杀死一个进程来回收它的内存后，第 11a 步才会成功执行。因此，第 11a 步避免了两个无辜的进程（而不是一个）被杀死。

释放一组页框

管理区分配器同样负责释放页框；幸运的是，释放内存比分配它要简单得多。

在前面的“分区页框分配器”一节描述的用来释放页框的所有内核宏和函数都依赖于 `__free_pages()` 函数。它接收的参数为将要释放的第一个页框的页描述符的地址 (page) 和将要释放的一组连续页框的数量的对数 (order)。该函数执行如下步骤：

1. 检查第一个页框是否真正属于动态内存 (它的 `PG_reserved` 标志被清 0)；如果不是，则终止。
2. 减少 `page->_count` 使用计数器的值；如果它仍然大于或等于 0，则终止。
3. 如果 `order` 等于 0，那么该函数调用 `free_hot_page()` 来释放页框给适当内存管理区的每 CPU 热高速缓存 (参见前面的“每 CPU 页框高速缓存”一节)。
4. 如果 `order` 大于 0，那么它将页框加入到本地链表中，并调用 `free_pages_bulk()` 函数把它们释放到适当内存管理区的伙伴系统中 (参见前面的“每 CPU 页框高速缓存”一节中描述的 `free_hot_cold_page()` 的第 3 步)。

内存区管理

本节关注内存区 (*memory area*)，也就是说，关注具有连续的物理地址和任意长度的内存单元序列。

伙伴系统算法采用页框作为基本内存区，这适合于对大块内存的请求，但我们如何处理对小内存区的请求呢，比如说几十或几百个字节？

显然，如果为了存放很少的字节而给它分配一个整页框，这显然是一种浪费。取而代之的正确方法就是引入一种新的数据结构来描述在同一页框中如何分配小内存区。但这样也引出了一个新的问题，即所谓的内碎片 (internal fragmentation)。内碎片的产生主要是由于请求内存的大小与分配给它的大小不匹配而造成的。

一种典型的解决方法 (早期 Linux 版本采用) 就是提供按几何分布的内存区大小，换句话说，内存区大小取决于 2 的幂而不取决于所存放的数据大小。这样，不管请求内存的大小是多少，我们都可保证内碎片小于 50%。为此，内核建立了 13 个按几何分布的空闲内存区链表，它们的大小从 32 到 131072 字节。伙伴系统的调用既为了获得存放新内存区所需的额外页框，也为了释放不再包含内存区的页框。用一个动态链表来记录每个页框所包含的空闲内存区。

slab 分配器

在伙伴算法之上运行内存区分配算法没有显著的效率。一种更好的算法源自 *slab* 分配器模式，该模式最早用于 Sun 公司的 Solaris 2.4 操作系统中。新算法基于下列前提：

- 所存放数据的类型可以影响内存区的分配方式。例如，当给用户态进程分配一个页框时，内核调用 `get_zeroed_page()` 函数用 0 填充这个页。

slab 分配器概念扩充了这种思想，并把内存区看作对象 (*object*)，这些对象由一组数据结构和几个叫做构造 (*constructor*) 或析构 (*destructor*) 的函数 (或方法) 组成。前者初始化内存区，而后者回收内存区。

为了避免重复初始化对象，slab 分配器并不丢弃已分配的对象，而是释放但把它们保存在内存中。当以后又要请求新的对象时，就可以从内存获取而不用重新初始化。

- 内核函数倾向于反复请求同一类型的内存区。例如，只要内核创建一个新进程，它就要为一些固定大小的表 [如进程描述符、打开文件对象等等 (参见第三章)] 分配内存区。当进程结束时，包含这些表的内存区还可以被重新使用。因为进程的创建和撤消非常频繁，在没有 slab 分配器时，内核把时间浪费在反复分配和回收那些包含同一内存区的页框上，slab 分配器把那些页框保存在高速缓存中并很快地重新使用它们。
- 对内存区的请求可以根据它们发生的频率来分类。对于预期频繁请求一个特定大小的内存区而言，可以通过创建一组具有适当大小的专用对象来高效地处理，由此以避免内碎片的产生。另一种情况，对于很少遇到的内存区大小，可以通过基于一系列几何分布大小 (如早期 Linux 版本所使用的 2 的幂次方大小) 的对象的分配模式来处理，即使这种方法会导致内碎片的产生。
- 在引入的对象大小不是几何分布的情况下，也就是说，数据结构的起始地址不是物理地址值的 2 的幂次方，事情反倒好办。这可以借助处理器硬件高速缓存而导致较好的性能。
- 硬件高速缓存的高性能又是尽可能地限制对伙伴系统分配器调用的另一个理由，因为对伙伴系统函数的每次调用都“弄脏”硬件高速缓存，所以增加了对内存的平均访问时间。内核函数对硬件高速缓存的影响就是所谓的函数“足迹 (*footprint*)”，其定义为函数结束时重写高速缓存的百分比。显而易见，大的“足迹”导致内核函数刚执行之后较慢的代码执行，因为硬件高速缓存此时充满了无用的信息。

slab 分配器把对象分组放进高速缓存。每个高速缓存都是同种类型对象的一种“储备”。例如，当一个文件被打开时，存放相应“打开文件”对象所需的内存区是从一个叫做 *filp* (“文件指针”) 的 slab 分配器的高速缓存中得到的。

包含高速缓存的主内存区被划分为多个 slab，每个 slab 由一个或多个连续的页框组成，这些页框中既包含已分配的对象，也包含空闲的对象 (如图 8-3 所示)。

图8-3
Cache —— 高速缓存
Object ----- 对象

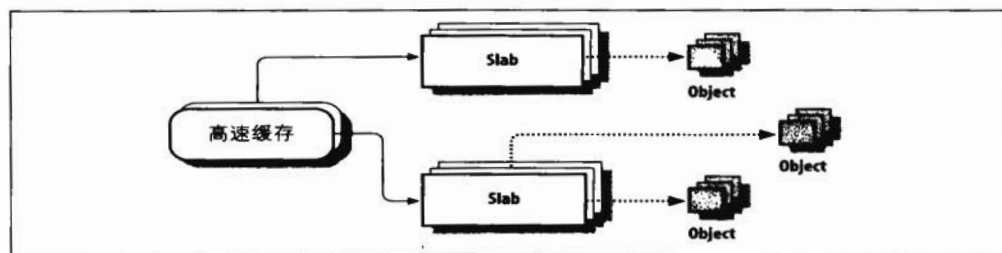


图 8-3: slab 分配器的组成

我们将在第十七章看到，内核周期性地扫描高速缓存并释放空 slab 对应的页框。

高速缓存描述符

每个高速缓存都是由 `kmem_cache_t`（等价于 `struct kmem_cache_s` 类型）类型的数据结构来描述的，表 8-8 列出了它的字段。我们在表中省略了用于收集统计信息 and 调试的几个字段。

表 8-8: `kmem_cache_t` 描述符的字段

类型	名称	说明
<code>struct array_cache * []</code>	<code>array</code>	每 CPU 指针数组指向包含空闲对象的本地高速缓存（参见本章后面的“空闲 slab 对象的本地高速缓存”一节）
<code>unsigned int</code>	<code>batchcount</code>	要转移进本地高速缓存或从本地高速缓存中转移出的大批对象的数量
<code>unsigned int</code>	<code>limit</code>	本地高速缓存中空闲对象的最大数目。这是可调的
<code>struct kmem_list3</code>	<code>lists</code>	参见下一个表
<code>unsigned int</code>	<code>objsize</code>	高速缓存中包含的对象的大小
<code>unsigned int</code>	<code>flags</code>	描述高速缓存永久属性的一组标志
<code>unsigned int</code>	<code>num</code>	封装在一个单独 slab 中的对象个数（高速缓存中的所有 slab 具有相同的大小）
<code>unsigned int</code>	<code>free_limit</code>	整个 slab 高速缓存中空闲对象的上限
<code>spinlock_t</code>	<code>spinlock</code>	高速缓存自旋锁
<code>unsigned int</code>	<code>gfpporder</code>	一个单独 slab 中包含的连续页框数目的对数

表 8-8: kmem_cache_t 描述符的字段 (续)

类型	名称	说明
unsigned int	gfpflags	分配页框时传递给伙伴系统函数的一组标志
size_t	colour	slab 使用的颜色个数 (参见本章后面的“slab 着色”一节)
unsigned int	colour_off	slab 中的基本对齐偏移
unsigned int	colour_next	下一个被分配的 slab 使用的颜色
kmem_cache_t *	slabp_cache	指针指向包含 slab 描述符的普通 slab 高速缓存 (如果使用了内部 slab 描述符, 则这个字段为 NULL; 参见下一节)
unsigned int	slab_size	单个 slab 的大小
unsigned int	dflags	描述高速缓存动态属性的一组标志
void *	ctor	指向与高速缓存相关的构造方法的指针
void *	dtor	指向与高速缓存相关的析构方法的指针
const char *	name	存放高速缓存名字的字符数组
struct list_head	next	高速缓存描述符双向链表使用的指针

kmem_cache_t 描述符的 lists 字段又是一个结构体, 它的字段在表 8-9 中列出。

表 8-9: kmem_list3 结构的字段

类型	名称	说明
struct list_head	slabs_partial	包含空闲和非空闲对象的 slab 描述符双向循环链表
struct list_head	slabs_full	不包含空闲对象的 slab 描述符双向循环链表
struct list_head	slabs_free	只包含空闲对象的 slab 描述符双向循环链表
unsigned long	free_objects	高速缓存中空闲对象的个数
int	free_touched	由 slab 分配器的页回收算法使用 (参见第十七章)
unsigned long	next_reap	由 slab 分配器的页回收算法使用 (参见第十七章)
struct array_cache *	shared	指向所有 CPU 共享的一个本地高速缓存的指针 (参见后面的“空闲 slab 对象的本地高速缓存”一节)

slab 描述符

高速缓存中的每个 slab 都有自己的类型为 slab 的描述符, 如表 8-10 所示。

表 8-10: slab 描述符的字段

类型	名称	说明
struct list_head	list	slab 描述符的三个双向循环链表中的一个 (在高速缓存描述符的 kmem_list3 结构中的 slabs_full、slabs_partial 或 slabs_free 链表) 使用的指针
unsigned long	colouroff	slab 中第一个对象的偏移 (参见本章后面的“slab 着色”一节)
void *	s_mem	slab 中第一个对象 (或者已被分配, 或者空闲) 的地址
unsigned int	inuse	当前正在使用的 (非空闲) slab 中的对象个数
unsigned int	free	slab 中下一个空闲对象的下标, 如果没有剩下空闲对象则为 BUFCTL_END (参见本章后面的“对象描述符”一节)

slab 描述符可以存放在两个可能的地方:

外部 slab 描述符

存放在 slab 外部, 位于 cache_sizes 指向的一个不适合 ISA DMA 的普通高速缓存中 (参见下一节)。

内部 slab 描述符

存放在 slab 内部, 位于分配给 slab 的第一个页框的起始位置。

当对象小于 512MB 时, 或者当内碎片在 slab 内部为 slab 描述符将在后面介绍 (及对象描述符) 留下足够的空间时, slab 分配器选择第二种方案。如果 slab 描述符存放在 slab 外部, 那么高速缓存描述符的 flags 字段中的 CFLGS_OFF_SLAB 标志被置 1, 否则它被置 0。

图 8-4 显示了高速缓存描述符和 slab 描述符之间的主要关系。全满的 slab、部分满的 slab 及空闲的 slab 链接在不同的链表中。

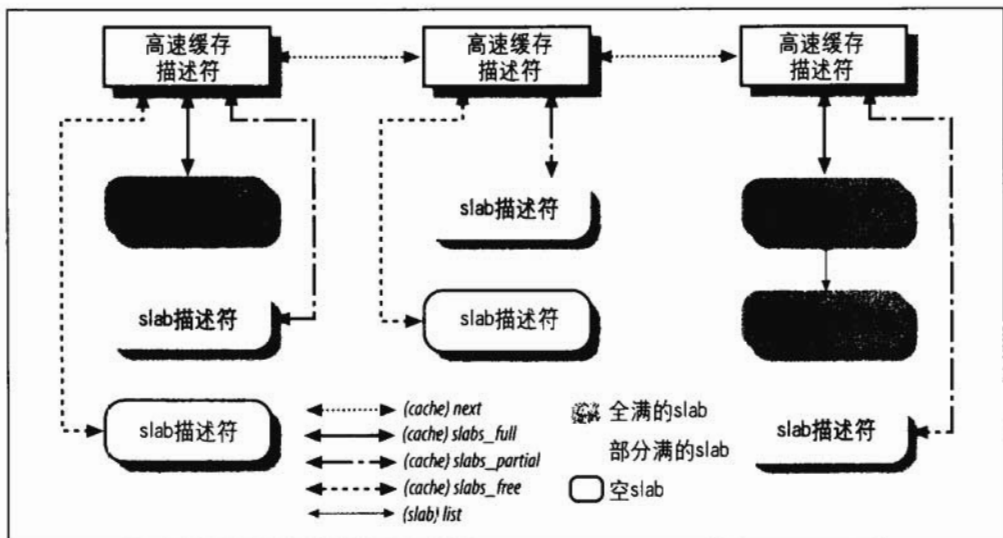


图 8-4：高速缓存描述符与 slab 描述符之间的关系

普通和专用高速缓存

高速缓存被分为两种类型：普通和专用。普通高速缓存只由slab分配器用于自己的目的，而专用高速缓存由内核的其余部分使用。

普通高速缓存是：

- 第一个高速缓存叫做 `kmem_cache`，包含由内核使用的其余高速缓存的高速缓存描述符。 `cache_cache` 变量包含第一个高速缓存的描述符。
- 另外一些高速缓存包含用作普通用途的内存区。内存区大小的范围一般包括 13 个几何分布的内存区。一个叫做 `malloc_sizes` 的表（其元素类型为 `cache_sizes`）分别指向 26 个高速缓存描述符。与其相关的内存区大小为 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536 和 131072 字节。对于每种大小，都有两个高速缓存：一个适用于 ISA DMA 分配，另一个适用于常规分配。

在系统初始化期间调用 `kmem_cache_init()` 和 `kmem_cache_sizes_init()` 来建立普通高速缓存。

专用高速缓存是由 `kmem_cache_create()` 函数创建的。这个函数首先根据参数确定处理新高速缓存的最佳方法（例如，是在 slab 的内部还是外部包含 slab 描述符）。然后它从 `cache_cache` 普通高速缓存中为新的高速缓存分配一个高速缓存描述符，并把这个描

述符插入到高速缓存描述符的 `cache_chain` 链表中(当获得了用于保护链表避免被同时访问的 `cache_chain_sem` 信号量后, 插入操作完成)。

还可以调用 `kmem_cache_destroy()` 撤销一个高速缓存并将它从 `cache_chain` 链表上删除。这个函数主要用于模块中, 即模块装入时创建自己的高速缓存, 卸载时撤销高速缓存。为了避免浪费内存空间, 内核必须在撤销高速缓存本身之前就撤销其所有的 slab。`kmem_cache_shrink()` 函数通过反复调用 `slab_destroy()` 撤销高速缓存中所有的 slab (参见后面“从高速缓存中释放 slab”一节)。

所有普通和专用高速缓存的名字都可以在运行期间通过读取 `/proc/slabinfo` 文件得到。这个文件也指明每个高速缓存中空闲对象的个数和已分配对象的个数。

slab 分配器与分区页框分配器的接口

当 slab 分配器创建新的 slab 时, 它依靠分区页框分配器来获得一组连续的空闲页框。为了达到此目的, 它调用 `kmem_getpages()` 函数, 在 UMA 系统上该函数本质上等价于如下代码片段:

```
void * kmem_getpages(kmem_cache_t *cachep, int flags)
{
    struct page *page;
    int i;

    flags |= cachep->gfpflags;
    page = alloc_pages(flags, cachep->gfporder);
    if (!page)
        return NULL;
    i = (1 << cachep->gfporder);
    if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
        atomic_add(i, &slab_reclaim_pages);
    while (i--)
        SetPageSlab(page++);
    return page_address(page);
}
```

两个参数的含义如下:

`cachep`

指向需要额外页框的高速缓存的高速缓存描述符 (请求页框的个数由存放在 `cachep->gfporder` 字段中的 `order` 决定)。

`flags`

说明如何请求页框 (参见本章前面“分区页框分配器”一节)。这组标志与存放在高速缓存描述符的 `gfpflags` 字段中的专用高速缓存分配标志相结合。

内存分配请求的大小由高速缓存描述符的`gfporder`字段指定,该字段将高速缓存中slab的大小编码(注7)。如果已经创建了slab高速缓存并且`SLAB_RECLAIM_ACCOUNT`标志已经置位,那么当内核检查是否有足够的内存来满足一些用户态请求时,分配给slab的页框将被记录为可回收的页。函数还将所分配页框的页描述符中的`PG_slab`标志置位。

在相反的操作中,通过调用`kmem_freepages()`函数可以释放分配给slab的页框(参见本章后面“从高速缓存中释放slab”一节):

```
void kmem_freepages(kmem_cache_t *cachep, void *addr)
{
    unsigned long i = (1<<cachep->gfporder);
    struct page *page = virt_to_page(addr);

    if (current->reclaim_state)
        current->reclaim_state->reclaimed_slab += i;
    while (i-->0)
        ClearPageSlab(page++);
    free_pages((unsigned long) addr, cachep->gfporder);
    if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
        atomic_sub(1<<cachep->gfporder, &slab_reclaim_pages);
}
```

这个函数从线性地址`addr`开始释放页框,这些页框曾分配给由`cachep`标识的高速缓存中的slab。如果当前进程正在执行内存回收(`current->reclaim_state`字段非`NULL`),`reclaim_state`结构的`reclaimed_slab`字段就被适当地增加,于是刚被释放的页就能通过页框回收算法(参见第十七章的“内存紧缺回收”一节)被记录下来。此外,如果`SLAB_RECLAIM_ACCOUNT`标志置位(参见上面),`slab_reclaim_pages`变量则被适当地减少。

给高速缓存分配 slab

一个新创建的高速缓存没有包含任何slab,因此也没有空闲的对象。只有当以下两个条件都为真时,才给高速缓存分配slab:

- 已发出一个分配新对象的请求。
- 高速缓存不包含任何空闲对象。

注7: 注意不可能从`ZONE_HIGHMEM`内存管理区分配页框,因为`kmem_getpages()`函数返回由`page_address()`函数产生的线性地址;正如在本章前面的“高端内存页框的内核映射”一节解释的那样,该函数为未映射的高端内存页框返回`NULL`。

当这些情况发生时，slab 分配器通过调用 `cache_grow()` 函数给高速缓存分配一个新的 slab。而这个函数调用 `kmem_getpages()` 从分区页框分配器获得一组页框来存放一个单独的 slab，然后又调用 `alloc_slabmgmt()` 获得一个新的 slab 描述符。如果高速缓存描述符的 `CFLGS_OFF_SLAB` 标志置位，则从高速缓存描述符的 `slabp_cache` 字段指向的普通高速缓存中分配这个新的 slab 描述符；否则，从 slab 的第一个页框中分配这个 slab 描述符。

给定一个页框，内核必须确定它是否被 slab 分配器使用，如果是，就迅速得到相应高速缓存和 slab 描述符的地址。因此，`cache_grow()` 扫描分配给新 slab 的页框的所有页描述符，并将高速缓存描述符和 slab 描述符的地址分别赋给页描述符中 `lru` 字段的 `next` 和 `prev` 子字段。这项工作不会出错，因为只有当页框空闲时伙伴系统的函数才会使用 `lru` 字段，而只要涉及伙伴系统，slab 分配器函数所处理的页框就不空闲并将 `PG_slab` 标志置位（注 8）。相反的问题——在高速缓存中给定一个 slab，用哪些页框来实现它？这个问题可以通过使用 slab 描述符的 `s_mem` 字段和高速缓存描述符的 `gfporder` 字段（slab 的大小）来回答。

接着，`cache_grow()` 调用 `cache_init_objs()`，它将构造方法（如果定义了的话）应用到新 slab 包含的所有对象上。

最后，`cache_grow()` 调用 `list_add_tail()` 来将新得到的 slab 描述符 `*slabp`，添加到高速缓存描述符 `*cachep` 的全空 slab 链表的末端，并更新高速缓存中的空闲对象计数器：

```
list_add_tail(&slabp->list, &cachep->lists->slabs_free);
cachep->lists->free_objects += cachep->num;
```

从高速缓存中释放 slab

在两种条件下才能撤销 slab：

- slab 高速缓存中有太多的空闲对象。
- 被周期性调用的定时器函数确定是否有完全未使用的 slab 能被释放（参见第十七章）。

在两种情况下，调用 `slab_destroy()` 函数撤销一个 slab，并释放相应的页框到分区页框分配器：

```
void slab_destroy(kmem_cache_t *cachep, slab_t *slabp)
{
    if (cachep->dtor) {
```

注 8：我们将在第十七章看到，`lru` 字段还被页框回收算法使用。

```

    int i;
    for (i = 0; i < cachep->num; i++) {
        void* objp = slabp->s_mem+cachep->objsize*i;
        (cachep->dtor)(objp, cachep, 0);
    }
    kmem_freepages(cachep, slabp->s_mem - slabp->colouroff);
    if (cachep->flags & CFLGS_OFF_SLAB)
        kmem_cache_free(cachep->slabp_cache, slabp);
}

```

这个函数检查高速缓存是否为它的对象提供了析构方法（dtor 字段不为 NULL），如果是，就使用析构方法释放 slab 中的所有对象。objp 局部变量记录当前已检查的对象。接下来，又调用 kmem_freepages()，该函数把 slab 使用的所有连续页框返回给伙伴系统。最后，如果 slab 描述符存放在 slab 的外面，那么，就从 slab 描述符的高速缓存释放这个 slab 描述符。

实际上，该函数稍微更复杂一些。例如，可以使用 SLAB_DESTROY_BY_RCU 标志来创建 slab 高速缓存，这就意味着应使用 call_rcu() 函数注册一个回调来延期释放 slab[参见第五章的“读—拷贝—更新(RCU)”一节]。正如前面描述的主要情形，回调函数接着调用 kmem_freepages()，也可能调用 kmem_cache_free()。

对象描述符

每个对象都有类型为 kmem_bufctl_t 的一个描述符。对象描述符存放在一个数组中，位于相应的 slab 描述符之后。因此，与 slab 描述符本身类似，slab 的对象描述符也可以用两种可能的方式来存放，如图 8-5 所示。

外部对象描述符

存放在 slab 的外面，位于高速缓存描述符的 slabp_cache 字段指向的一个普通高速缓存中。内存区的大小（尤其是存放对象描述符的普通高速缓存的大小）取决于在 slab 中所存放的对象个数（高速缓存描述符的 num 字段）。

内部对象描述符

存放在 slab 内部，正好位于描述符所描述的对象之前。

数组中的第一个对象描述符描述 slab 中的第一个对象，依次类推。对象描述符只不过是一个无符号整数，只有在对象空闲时才有意义。它包含的是下一个空闲对象在 slab 中的下标，因此实现了 slab 内部空闲对象的一个简单链表。空闲对象链表中的最后一个元素的对象描述符用常规值 BUFCTL_END (0xffff) 标记。

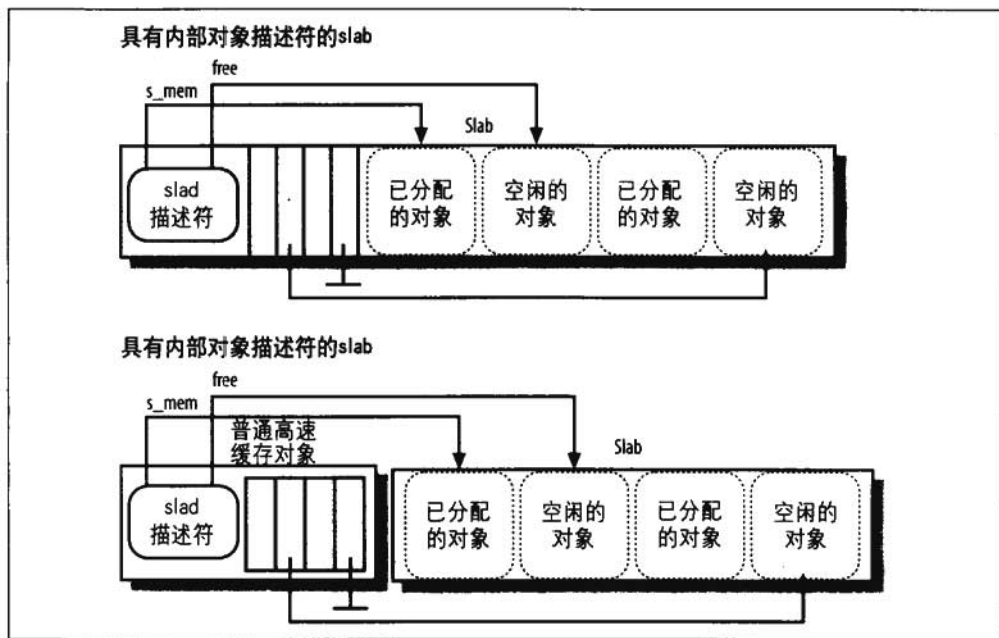


图 8-5: slab 描述符与对象描述符之间的关系

对齐内存中的对象

slab 分配器所管理的对象可以在内存中进行对齐，也就是说，存放它们的内存单元的起始物理地址是一个给定常量的倍数，通常是2的倍数。这个常量就叫对齐因子 (*alignment factor*)。

slab 分配器所允许的最大对齐因子是 4096，即页框大小。这就意味着通过访问对象的物理地址或线性地址就可以对齐对象。在这两种情况下，只有最低的 12 位可以通过对齐来改变。

通常情况下，如果内存单元的物理地址是字大小（即计算机的内部内存总线的宽度）对齐的，那么，微机对内存单元的存取会非常快。因此，缺省情况下，`kmem_cache_create()` 函数根据 `BYTES_PER_WORD` 宏所指定的字大小来对齐对象。对于 80x86 处理器，这个宏产生的值为 4，因为字长是 32 位。

当创建一个新的 slab 高速缓存时，就可以让它所包含的对象在第一级硬件高速缓存中对齐。为了做到这点，设置 `SLAB_HWCACHE_ALIGN` 高速缓存描述符标志。`kmem_cache_create()` 函数按如下方式处理请求：

- 如果对象的大小大于高速缓存行(cache line)的一半,就在RAM中根据L1_CACHE_BYTES的倍数(也就是行的开始)对齐对象。
- 否则,对象的大小就是L1_CACHE_BYTES的因子取整。这可以保证一个小对象不会横跨两个高速缓存行。

显然,slab分配器在这里所做的事情就是以内存空间换取访问时间,即通过人为地增加对象的大小来获得较好的高速缓存性能,由此也引起额外的内碎片。

slab 着色

从第二章我们知道,同一硬件高速缓存行可以映射RAM中很多不同的块。在本章我们也已看到,相同大小的对象倾向于存放在高速缓存内相同的偏移量处。在不同的slab内具有相同偏移量的对象最终很可能映射在同一高速缓存行中。高速缓存的硬件可能因此而花费内存周期在同一高速缓存行与RAM内存单元之间来来往往传送两个对象,而其他的高速缓存行并未充分使用。slab分配器通过一种叫做slab着色(*slab coloring*)的策略,尽量降低高速缓存的这种不愉快行为:把叫做颜色(*color*)的不同随机数分配给slab。

在讨论slab着色之前,我们先看一下高速缓存内对象的布局。让我们考虑某个高速缓存,它的对象在RAM中被对齐。这就意味着对象的地址肯定是某个给定正数值(比如说 aln)的倍数。连对齐的约束也考虑在内,在slab内放置对象就有很多种可能的方式。方式的选择取决于对下列变量所做的决定:

num

可以在slab中存放的对象个数(其值在高速缓存描述符的*num*字段中)。

osize

对象的大小,包括对齐的字节。

dsiz

slab描述符的大小加上所有对象描述符的大小,就等于硬件高速缓存行大小的最小倍数。如果slab描述符和对象描述符都存放在slab的外部,那么这个值等于0。

free

在slab内未用字节(没有分配给任一对象的字节)的个数。

一个slab中的总字节长度可以表示为如下表达式:

$$\text{slab 的长度} = (\text{num} \times \text{osize}) + \text{dsiz} + \text{free}$$

*free*总是小于*osize*,因为否则的话,就有可能把另外的对象放在slab内。不过,*free*可以大于 aln 。

slab分配器利用空闲未用的字节 $free$ 来对slab着色。术语“着色”只是用来再细分slab，并允许内存分配器把对象展开在不同的线性地址之中。这样的话，内核从微处理器的硬件高速缓存中可能获得最好性能。

具有不同颜色的slab把slab的第一个对象存放在不同的内存单元，同时满足对齐约束。可用颜色的个数是 $free/aln$ （这个值存放在高速缓存描述符的 $colour$ 字段）。因此，第一个颜色表示为0，最后一个颜色表示为 $(free/aln)-1$ 。（一种特殊情况是，如果 $free$ 比 aln 小，那么 $colour$ 被设为0，不过所有slab都使用颜色0，因此颜色真正的个数为1。）

如果用颜色 col 对一个slab着色，那么，第一个对象的偏移量（相对于slab的起始地址）就等于 $col \times aln + dsize$ 字节。图8-6显示了slab内对象的布局对slab颜色的依赖情况。着色本质上导致把slab中的一些空闲区域从末尾移到开始。

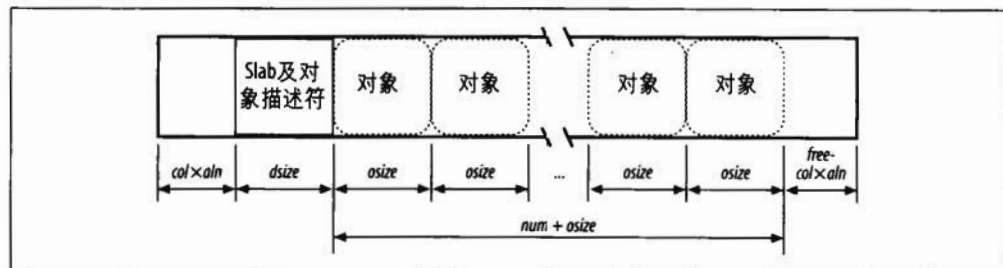


图8-6：具有颜色 col 与对齐 aln 的slab

只有当 $free$ 足够大时，着色才起作用。显然，如果对象没有请求对齐，或者如果slab内的未用字节数小于所请求的对齐（ $free \leq aln$ ），那么，唯一可能着色的slab就是具有颜色0的slab，也就是说，把这个slab的第一个对象的偏移量赋为0。

通过把当前颜色存放在高速缓存描述符的 $colour_next$ 字段，就可以在一个给定对象类型的slab之间平等地发布各种颜色。 $cache_grow()$ 函数把 $colour_next$ 所表示的颜色赋给一个新的slab，并递增这个字段的值。当 $colour_next$ 的值变为 $colour$ 后，又从0开始。这样，每个新创建的slab都与前一个slab具有不同的颜色，直到最大可用颜色。此外， $cache_grow()$ 函数从高速缓存描述符的 $colour_off$ 字段获得值 aln ，根据slab内对象的个数计算 $dsize$ ，最后把 $col \times aln + dsize$ 的值存放到slab描述符的 $colouroff$ 字段中。

空闲 Slab 对象的本地高速缓存

Linux 2.6 对多处理器系统上 slab 分配器的实现不同于 Solaris 2.4 上最初的实现。为了减少处理器之间对自旋锁的竞争并更好地利用硬件高速缓存，slab 分配器的每个高速缓

存包含一个被称作slab本地高速缓存的每CPU数据结构,该结构由一个指向被释放对象的小指针数组组成。slab对象的大多数分配和释放只影响本地数组,只有在本地数组下溢或上溢时才涉及slab数据结构。该技术非常类似于本章前面的“每CPU页框高速缓存”一节中的技术。

高速缓存描述符的array字段是一组指向array_cache数据结构的指针,系统中的每个CPU对应于一个元素。每个array_cache数据结构是空闲对象的本地高速缓存的一个描述符,它的字段显示在表8-11中。

表 8-11: array_cache 结构的字段

类型	名称	说明
unsigned int	avail	指向本地高速缓存中可使用对象的指针的个数。它同时也作为高速缓存中第一个空槽的下标
unsigned int	limit	本地高速缓存的大小,也就是本地高速缓存中指针的最大个数
unsigned int	batchcount	本地高速缓存重新填充或腾空时使用的块大小
unsigned int	touched	如果本地高速缓存最近已经被使用过,则该标志设为1

注意,本地高速缓存描述符并不包含本地高速缓存本身的地址;事实上,它正好位于描述符之后。当然,本地高速缓存存放的是指向已释放对象的指针,而不是对象本身,对象本身总是位于高速缓存的slab中。

当创建一个新的slab高速缓存时,kmem_cache_create()函数决定本地高速缓存的大小(将这个值存放在高速缓存描述符的limit字段中)、分配本地高速缓存,并将它们的指针存放在高速缓存描述符的array字段。这个大小取决于存放在slab高速缓存中对象的大小,范围从1(相对于非常大的对象)到120(相对于小对象)。此外,batchcount字段的初始值,也就是从一个本地高速缓存的块里添加或删除的对象的个数,被初始化为本地高速缓存大小的一半(注9)。

在多处理器系统中,小对象使用的slab高速缓存同样包含一个附加的本地高速缓存,它的地址被存放在高速缓存描述符的lists.shared字段中。共享的本地高速缓存正如它的名字暗示的那样,被所有CPU共享,它使得将空闲对象从一个本地高速缓存移动到另一个高速缓存的任务更容易(参见下一节)。它的初始大小等于batchcount字段的值的8倍。

注9: 系统管理员通过写入/proc/slabinfo文件可以为每个高速缓存调整本地高速缓存的大小以及batchcount字段的值。

分配 slab 对象

通过调用 `kmem_cache_alloc()` 函数可以获得新对象。参数 `cachep` 指向高速缓存描述符，新空闲对象必须从该高速缓存描述符获得，而参数 `flag` 表示传递给分区页框分配器函数的标志，该高速缓存的所有 slab 应当是满的。

该函数本质上等价于下列代码：

```
void * kmem_cache_alloc(kmem_cache_t *cachep, int flags)
{
    unsigned long save_flags;
    void *objp;
    struct array_cache *ac;

    local_irq_save(save_flags);
    ac = cache_p->array[smp_processor_id()];
    if (ac->avail) {
        ac->touched = 1;
        objp = ((void **) (ac+1))[--ac->avail];
    } else
        objp = cache_alloc_refill(cachep, flags);
    local_irq_restore(save_flags);
    return objp;
}
```

函数首先试图从本地高速缓存获得一个空闲对象。如果有空闲对象，`avail` 字段就包含指向最后被释放的对象的项在本地高速缓存中的下标。因为本地高速缓存数组正好存放在 `ac` 描述符后面，所以 `((void**) (ac+1))[--ac->avail]` 获得那个空闲对象的地址并递减 `ac->avail` 的值。当本地高速缓存中没有空闲对象时，调用 `cache_alloc_refill()` 函数重新填充本地高速缓存并获得一个空闲对象。

`cache_alloc_refill()` 函数本质上执行如下步骤：

1. 将本地高速缓存描述符的地址存放在 `ac` 局部变量中：

```
ac = cachep->array[smp_processor_id()];
```

2. 获得 `cachep->spinlock`。
3. 如果 slab 高速缓存包含共享本地高速缓存，并且该共享本地高速缓存包含一些空闲对象，函数就通过从共享本地高速缓存中上移 `ac->batchcount` 个指针来重新填充 CPU 的本地高速缓存。然后，函数跳到第 6 步。
4. 函数试图填充本地高速缓存，填充值为高速缓存的 slab 中包含的多达 `ac->batchcount` 个空闲对象的指针：
 - a. 查看高速缓存描述符的 `slabs_partial` 和 `slabs_free` 链表，并获得 slab 描述

符的地址 `slabp`，该 `slab` 描述符的相应 `slab` 或者部分被填充，或者为空。如果不存在这样的描述符，则函数转到第 5 步。

- b. 对于 `slab` 中的每个空闲对象，函数增加 `slab` 描述符的 `inuse` 字段，将对象的地址插入本地高速缓存，并更新 `free` 字段使得它存放了 `slab` 中下一个空闲对象的下标：

```
slabp->inuse++;
((void**) (ac+1)) [ac->avail++] =
    slabp->s_mem + slabp->free * cachep->obj_size;
slabp->free = ((kmem_bufctl_t*) (slabp+1)) [slabp->free];
```

- c. 如果必要，将清空的 `slab` 插入到适当的链表上，可以是 `slab_full` 链表，也可以是 `slab_partial` 链表。

5. 在这一步，被加到本地高速缓存上的指针个数被存放在 `ac->avail` 字段：函数递减同样数量的 `kmem_list3` 结构的 `free_objects` 字段来说明这些对象不再空闲。
6. 释放 `cachep->spinlock`。
7. 如果现在 `ac->avail` 字段大于 0（一些高速缓存再填充的情况发生了），函数将 `ac->touched` 字段设为 1，并返回最后插入到本地高速缓存的空闲对象指针：

```
return ((void**) (ac+1)) [--ac->avail];
```

8. 否则，没有发生任何高速缓存再填充情况：调用 `cache_grow()` 获得一个新 `slab`，从而获得了新的空闲对象。
9. 如果 `cache_grow()` 失败了，则函数返回 `NULL`；否则它返回到第 1 步重复该过程。

释放 Slab 对象

`kmem_cache_free()` 函数释放一个曾经由 `slab` 分配器分配给某个内核函数的对象。它的参数为 `cachep` 和 `objp`，前者是高速缓存描述符的地址，而后者是将被释放对象的地址：

```
void kmem_cache_free(kmem_cache_t *cachep, void *objp)
{
    unsigned long flags;
    struct array_cache *ac;

    local_irq_save(flags);
    ac = cachep->array[smp_processor_id()];
    if (ac->avail == ac->limit)
        cache_flusharray(cachep, ac);
    ((void**) (ac+1)) [ac->avail++] = objp;
    local_irq_restore(flags);
}
```

函数首先检查本地高速缓存是否有空间给指向一个空闲对象的额外指针。如果有, 该指针就被加到本地高速缓存然后函数返回。否则, 它首先调用 `cache_flusharray()` 来清空本地高速缓存, 然后将指针加到本地高速缓存。

`cache_flusharray()` 函数执行如下操作:

1. 获得 `cachep->spinlock` 自旋锁。
2. 如果 slab 高速缓存包含一个共享本地高速缓存, 并且如果该共享本地高速缓存还没有满, 函数就通过从 CPU 的本地高速缓存中上移 `ac->batchcount` 个指针来重新填充共享本地高速缓存。
3. 调用 `free_block()` 函数将当前包含在本地高速缓存中的 `ac->batchcount` 个对象归还给 slab 分配器。对于在地址 `objp` 处的每个对象, 函数执行如下步骤:

a. 增加高速缓存描述符的 `lists.free_objects` 字段。

b. 确定包含对象的 slab 描述符的地址:

```
slabp = (struct slab *) (virt_to_page(objp)->lru.prev);
```

(请记住, slab 页的描述符的 `lru.prev` 字段指向相应的 slab 描述符。)

c. 从它的 slab 高速缓存链表 (`cachep->lists.slabs_partial` 或是 `cachep->lists.slabs_full`) 上删除 slab 描述符。

d. 计算 slab 内对象的下标:

```
objnr = (objp - slabp->s_mem) / cachep->objsize;
```

e. 将 `slabp->free` 的当前值存放在对象描述符中, 并将对象的下标放入 `slabp->free` (最后被释放的对象将再次成为首先被分配的对象):

```
((kmem_bufctl_t *) (slabp+1))[objnr] = slabp->free;  
slabp->free = objnr;
```

f. 递减 `slabp->inuse` 字段。

g. 如果 `slabp->inuse` 等于 0 (也就是 slab 中所有对象空闲), 并且整个 slab 高速缓存中空闲对象的个数 (`cachep->lists.free_objects`) 大于 `cachep->free_limit` 字段中存放的限制, 那么函数将 slab 的页框释放到分区页框分配器:

```
cachep->lists.free_objects -= cachep->num;  
slab_destroy(cachep, slabp);
```

存放在 `cachep->free_limit` 字段中的值通常等于 `cachep->num + (1+N) * cachep->batchcount`, 其中 N 代表系统中 CPU 的个数。

h. 否则, 如果 `slab->inuse` 等于 0, 但整个 slab 高速缓存中空闲对象的个数小于

cache->free_limit, 函数就将slab描述符插入到cache->lists.slabs_free链表中。

- i. 最后, 如果slab->inuse大于0, slab被部分填充, 则函数将slab描述符插入到cache->lists.slabs_partial链表中。
4. 释放cache->spinlock自旋锁。
5. 通过减去被移到共享本地高速缓存或被释放到slab分配器的对象的个数来更新本地高速缓存描述符的avail字段。
6. 移动本地高速缓存数组起始处的那个本地高速缓存中的所有指针。这一步是必需的, 因为已经把第一个对象指针从本地高速缓存上删除, 因此剩下的指针必须上移。

通用对象

正如“伙伴系统算法”一节中所描述的那样, 如果对存储区的请求不频繁, 就用一组普通高速缓存来处理, 普通高速缓存中的对象具有几何分布的大小, 范围为32~131072字节。

调用kmalloc()函数就可以得到这种类型的对象, 函数等价于下列代码片段:

```
void * kmalloc(size_t size, int flags)
{
    struct cache_sizes *csizes = malloc_sizes;
    kmem_cache_t * cache;
    for (; csizes->cs_size; csizes++) {
        if (size > csizes->cs_size)
            continue;
        if (flags & __GFP_DMA)
            cache = csizes->cs_dmacache;
        else
            cache = csizes->cs_cache;
        return kmem_cache_alloc(cache, flags);
    }
    return NULL;
}
```

该函数使用malloc_sizes表为所请求的大小分配最近的2的幂次方大小的内存。然后, 调用kmem_cache_alloc()分配对象, 传递的参数或者为适用于ISA DMA页框的高速缓存描述符, 或者为适用于“常规”页框的高速缓存描述符, 这取决于调用者是否指定了__GFP_DMA标志。

调用kmalloc()所获得的对象可以通过调用kfree()来释放:

```
void kfree(const void *objp)
{
    kmem_cache_t * c;
```

```

unsigned long flags;
if (!objp)
    return;
local_irq_save(flags);
c = (kmem_cache_t *) (virt_to_page(objp) -> lru.next);
kmem_cache_free(c, (void *)objp);
local_irq_restore(flags);
}

```

通过读取内存区所在的第一个页框描述符的 `lru.next` 子字段，就可确定出合适的高速缓存描述符。通过调用 `kmem_cache_free()` 来释放相应的内存区。

内存池

内存池 (*memory pool*) 是 Linux 2.6 的一个新特性。基本上讲，一个内存池允许一个内核成分，如块设备子系统，仅在内存不足的紧急情况下分配一些动态内存来使用。

不应该将内存池与前面“保留的页框池”一节中描述的保留页框混淆。实际上这些页框只能用于满足中断处理程序或内部临界区发出的原子内存分配请求。而内存池是动态内存的储备，只能被特定的内核成分（即池的“拥有者”）使用。拥有者通常不使用储备，但是，如果动态内存变得极其稀有以至于所有普通内存分配请求都将失败的话，那么作为最后的解决手段，内核成分就能调用特定的内存池函数提取储备得到所需的内存。因此，创建一个内存池就像手头存放一些罐装食物作为储备，当没有新鲜食物时就使用开罐器。

一个内存池常常叠加在 slab 分配器之上——也就是说，它被用来保存 slab 对象的储备。但是一般而言，内存池能被用来分配任何一种类型的动态内存，从整个页框到使用 `kmalloc()` 分配的小内存区。因此，我们一般将内存池处理的内存单元看作“内存元素”。

内存池由 `mempool_t` 对象描述，它的字段如表 8-12 所示。

表 8-12: `mempool_t` 对象的字段

类型	名称	说明
<code>spinlock_t</code>	<code>lock</code>	用来保护对象字段的自旋锁
<code>int</code>	<code>min_nr</code>	内存池中元素的最大个数
<code>int</code>	<code>curr_nr</code>	当前内存池中元素的个数
<code>void **</code>	<code>elements</code>	指向一个数组的指针，该数组由指向保留元素的指针组成
<code>void *</code>	<code>pool_data</code>	池的拥有者可获得的私有数据
<code>mempool_alloc_t *</code>	<code>alloc</code>	分配一个元素的方法

表 8-12: mempool_t 对象的字段 (续)

类型	名称	说明
mempool_free_t *	free	释放一个元素的方法
wait_queue_head_t	wait	当内存池为空时使用的等待队列

min_nr 字段存放了内存池中元素的初始个数。换句话说, 存放在该字段中的值代表了内存元素的个数, 内存池的拥有者确信能从内存分配器得到这个数目。curr_nr 字段总是低于或等于 min_nr, 它存放了内存池中当前包含的内存元素个数。内存元素自身被一个指针数组引用, 指针数组的地址存放在 elements 字段中。

alloc 和 free 方法与基本的内存分配器进行交互, 分别用于获得和释放一个内存元素。两个方法可以是拥有内存池的内核成分提供的定制函数。

当内存元素是 slab 对象时, alloc 和 free 方法一般由 mempool_alloc_slab() 和 mempool_free_slab() 函数实现, 它们只是分别调用 kmem_cache_alloc() 和 kmem_cache_free() 函数。在这种情况下, mempool_t 对象的 pool_data 字段存放了 slab 高速缓存描述符的地址。

mempool_create() 函数创建一个新的内存池, 它接收的参数为内存元素的个数 min_nr、实现 alloc 和 free 方法的函数的地址和赋给 pool_data 字段的任意值。该函数分别为 mempool_t 对象和指向内存元素的指针数组分配内存, 然后反复调用 alloc 方法来得到 min_nr 个内存元素。相反地, mempool_destroy() 函数释放池中所有内存元素, 然后释放元素数组和 mempool_t 对象自己。

为了从内存池分配一个元素, 内核调用 mempool_alloc() 函数, 将 mempool_t 对象的地址和内存分配标志传递给它 (参见本章前面的表 8-5 和表 8-6)。函数本质上依据参数所指定的内存分配标志, 试图通过调用 alloc 方法从基本内存分配器分配一个内存元素。如果分配成功, 函数返回获得的内存元素而不触及内存池。否则, 如果分配失败, 就从内存池获得内存元素。当然, 在内存不足的情况下过多的分配会用尽内存池: 在这种情况下, 如果 __GFP_WAIT 标志没有置位, 则 mempool_alloc() 阻塞当前进程直到有一个内存元素被释放到内存池中。

相反地, 为了释放一个元素到内存池, 内核调用 mempool_free() 函数。如果内存池未滿 (curr_min 小于 min_nr), 则函数将元素加到内存池中。否则, mempool_free() 调用 free 方法来释放元素到基本内存分配器。

非连续内存区管理

从前面的讨论中我们已经知道，把内存区映射到一组连续的页框是最好的选择，这样会充分利用高速缓存并获得较低的平均访问时间。不过，如果对内存区的请求不是很频繁，那么，通过连续的线性地址来访问非连续的页框这样一种分配模式就会很有意义。这种模式的主要优点是避免了外碎片，而缺点是必须打乱内核页表。显然，非连续内存区的大小必须是4096的倍数。Linux在几个方面使用非连续内存区，例如，为活动的交换区分配数据结构（参见第十七章中的“激活和禁用交换区”一节），为模块分配空间（参见附录二），或者给某些I/O驱动程序分配缓冲区。此外，非连续内存区还提供了另一种使用高端内存页框的方法（参见后面的“分配非连续内存区”一节）。

非连续内存区的线性地址

要查找线性地址的一个空闲区，我们可以从PAGE_OFFSET开始查找（通常为0xc0000000，即第4个GB的起始地址）。图8-7显示了如何使用第4个GB的线性地址：

- 内存区的开始部分包含的是对前896MB RAM进行映射的线性地址（参见第二章“进程页表”一节）；直接映射的物理内存末尾所对应的线性地址保存在high_memory变量中。
- 内存区的结尾部分包含的是固定映射的线性地址（参见第二章“固定映射的线性地址”一节）。
- 从PKMAP_BASE开始，我们查找用于高端内存页框的永久内核映射的线性地址（参见本章前面“高端内存页框的内核映射”一节）。
- 其余的线性地址可以用于非连续内存区。在物理内存映射的末尾与第一个内存区之间插入一个大小为8MB（宏VMALLOC_OFFSET）的安全区，目的是为了“捕获”对内存的越界访问。出于同样的理由，插入其他4KB大小的安全区来隔离非连续的内存区。

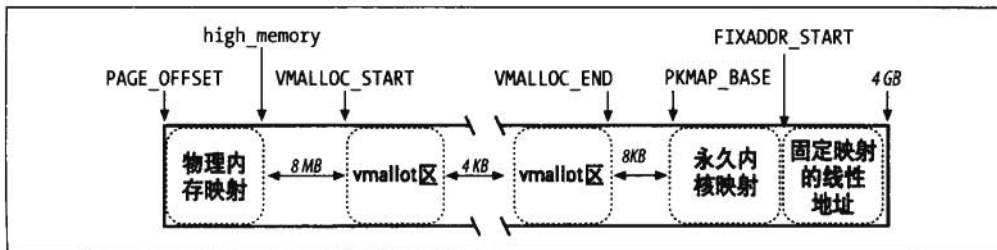


图 8-7：从 PAGE_OFFSET 开始的线性地址区间

为非连续内存区保留的线性地址空间的起始地址由 `VMALLOC_START` 宏定义, 而末尾地址由 `VMALLOC_END` 宏定义。

非连续内存区的描述符

每个非连续内存区都对应着一个类型为 `vm_struct` 的描述符, 表 8-13 列出了它的字段。

表 8-13: `vm_struct` 描述符的字段

类型	名称	说明
<code>void *</code>	<code>addr</code>	内存区内第一个内存单元的线性地址
<code>unsigned long</code>	<code>size</code>	内存区的大小加 4096 (内存区之间的安全区间的大小)
<code>unsigned long</code>	<code>flags</code>	非连续内存区映射的内存的类型
<code>struct page **</code>	<code>pages</code>	指向 <code>nr_pages</code> 数组的指针, 该数组由指向页描述符的指针组成
<code>unsigned int</code>	<code>nr_pages</code>	内存区填充的页的个数
<code>unsigned long</code>	<code>phys_addr</code>	该字段设为 0, 除非内存已被创建来映射一个硬件设备的 I/O 共享内存
<code>struct vm_struct *</code>	<code>next</code>	指向下一个 <code>vm_struct</code> 结构的指针

通过 `next` 字段, 这些描述符被插入到一个简单的链表中, 链表第一个元素的地址存放在 `vmlist` 变量中。对这个链表的访问依靠 `vmlist_lock` 读/写自旋锁来保护。`flags` 字段标识了非连续区映射的内存的类型: `VM_ALLOC` 表示使用 `vmalloc()` 得到的页, `VM_MAP` 表示使用 `vmap()` 映射的已经被分配的页 (参见下一节), 而 `VM_IOREMAP` 表示使用 `ioremap()` 映射的硬件设备的板上内存 (参见第十三章)。

`get_vm_area()` 函数在线性地址 `VMALLOC_START` 和 `VMALLOC_END` 之间查找一个空闲区域。该函数使用两个参数: 将被创建的内存区的字节大小 (`size`) 和指定空闲区类型 (参见上面) 的标志 (`flag`)。步骤执行如下:

1. 调用 `kmalloc()` 为 `vm_struct` 类型的新描述符获得一个内存区。
2. 为写得到 `vmlist_lock` 锁, 并扫描类型为 `vm_struct` 的描述符链表来查找线性地址一个空闲区域, 至少覆盖 `size + 4096` 个地址 (4096 是内存区之间的安全区间大小)。
3. 如果存在这样一个区间, 函数就初始化描述符的字段, 释放 `vmlist_lock` 锁, 并以 返回这个非连续内存区的起始地址而结束。

4. 否则, `get_vm_area()` 释放先前得到的描述符, 释放 `vmlist_lock`, 然后返回 `NULL`。

分配非连续内存区

`vmalloc()` 函数给内核分配一个非连续内存区。参数 `size` 表示所请求内存区的大小。如果这个函数能够满足请求, 就返回新内存区的起始地址; 否则, 返回一个 `NULL` 指针:

```
void * vmalloc(unsigned long size)
{
    struct vm_struct *area;
    struct page **pages;
    unsigned int array_size, i;
    size = (size + PAGE_SIZE - 1) & PAGE_MASK;
    area = get_vm_area(size, VM_ALLOC);
    if (!area)
        return NULL;
    area->nr_pages = size >> PAGE_SHIFT;
    array_size = (area->nr_pages * sizeof(struct page *));
    area->pages = pages = kmalloc(array_size, GFP_KERNEL);
    if (!area->pages) {
        remove_vm_area(area->addr);
        kfree(area);
        return NULL;
    }
    memset(area->pages, 0, array_size);
    for (i=0; i<area->nr_pages; i++) {
        area->pages[i] = alloc_page(GFP_KERNEL|_ _GFP_HIGHMEM);
        if (!area->pages[i]) {
            area->nr_pages = i;
fail:    vfree(area->addr);
            return NULL;
        }
    }
    if (map_vm_area(area, _ _pgprot(0x63), &pages))
        goto fail;
    return area->addr;
}
```

函数首先将参数 `size` 设为 4096 (页框大小) 的整数倍。然后, `vmalloc()` 调用 `get_vm_area()` 来创建一个新的描述符, 并返回分配给这个内存区的线性地址。描述符的 `flags` 字段被初始化为 `VM_ALLOC` 标志, 该标志意味着通过使用 `vmalloc()` 函数, 非连续页框将被映射到一个线性地址区间。然后 `vmalloc()` 函数调用 `kmalloc()` 来请求一组连续页框, 这组连续页框足够包含一个页描述符指针数组。调用 `memset()` 函数来将所有这些指针设为 `NULL`。接着重复调用 `alloc_page()` 函数, 每一次为区间中 `nr_pages` 个页的每一个分配一个页框, 并把对应页描述符的地址存放在 `area->pages` 数组中。注意, 必须使用 `area->pages` 数组是因为页框可能属于 `ZONE_HIGHMEM` 内存管理区, 所以此时它们不必被映射到一个线性地址上。

现在到了棘手的部分。直到这里，已经得到了一个新的连续线性地址区间，并且已经分配了一组非连续页框来映射这些线性地址。最后至关重要的步骤是修改内核使用的页表项，以此表明分配给非连续内存区的每个页框现在对应着一个线性地址，这个线性地址被包含在 `vmalloc()` 产生的非连续线性地址区间中。这就是 `map_vm_area()` 所要做的。

`map_vm_area()` 函数使用以下 3 个参数：

`area`

指向内存区的 `vm_struct` 描述符的指针。

`prot`

已分配页框的保护位。它总是被置为 `0x63`，对应着 Present、Accessed、Read/Write 及 Dirty。

`pages`

指向一个指针数组的变量的地址，该指针数组的指针指向页描述符（因此，`struct page ***` 被当作数据类型使用！）。

函数首先把内存区的开始和末尾的线性地址分别分配给局部变量 `address` 和 `end`：

```
address = area->addr;
end = address + (area->size - PAGE_SIZE);
```

请记住，`area->size` 存放的是内存区的实际地址加上 4KB 内存之间的安全区间。然后函数使用 `pgd_offset_k` 宏来得到在主内核页全局目录中的目录项，该项对应于内存区起始线性地址，然后获得内核页表自旋锁：

```
pgd = pgd_offset_k(address);
spin_lock(&init_mm.page_table_lock);
```

然后，函数执行下列循环：

```
int ret = 0;
for (i = pgd_index(address); i < pgd_index(end-1); i++) {
    pud_t *pud = pud_alloc(&init_mm, pgd, address);
    ret = -ENOMEM;
    if (!pud)
        break;
    next = (address + PGDIR_SIZE) & PGDIR_MASK;
    if (next < address || next > end)
        next = end;
    if (map_area_pud(pud, address, next, prot, pages))
        break;
    address = next;
    pud++;
    ret = 0;
}
```

```
spin_unlock(&init_mm.page_table_lock);
flush_cache_vmap((unsigned long)area->addr, end);
return ret;
```

每次循环都首先调用 `pud_alloc()` 来为新内存区创建一个页上级目录, 并把它的物理地址写入内核页全局目录的合适表项。然后调用 `alloc_area_pud()` 为新的页上级目录分配所有相关的页表。接下来, 把常量 2^{30} (在 PAE 被激活的情况下, 否则为 2^{22}) 与 `address` 的当前值相加 (2^{30} 就是一个页上级目录所跨越的线性地址范围的大小), 最后增加指向页全局目录的指针 `pgd`。

循环结束的条件是: 指向非连续内存区的所有页表项全被建立。

`map_area_pud()` 函数为页上级目录所指向的所有页表执行一个类似的循环:

```
do {
    pmd_t * pmd = pmd_alloc(&init_mm, pud, address);
    if (!pmd)
        return -ENOMEM;
    if (map_area_pmd(pmd, address, end-address, prot, pages))
        return -ENOMEM;
    address = (address + PUD_SIZE) & PUD_MASK;
    pud++;
} while (address < end);
```

`map_area_pmd()` 函数为页中间目录所指向的所有页表执行一个类似的循环:

```
do {
    pte_t * pte = pte_alloc_kernel(&init_mm, pmd, address);
    if (!pte)
        return -ENOMEM;
    if (map_area_pte(pte, address, end-address, prot, pages))
        return -ENOMEM;
    address = (address + PMD_SIZE) & PMD_MASK;
    pmd++;
} while (address < end);
```

`pte_alloc_kernel()` 函数 (参见第二章的“页表处理”一节) 分配一个新的页表, 并更新页中间目录中相应的目录项。接下来, `map_area_pte()` 为页表中相应的表项分配所有的页框。`address` 值增加 2^{22} (2^{22} 就是一个页表所跨越的线性地址区间的大小), 并且循环反复执行。

`map_area_pte()` 的主循环为:

```
do {
    struct page * page = **pages;
    set_pte(pte, mk_pte(page, prot));
    address += PAGE_SIZE;
    pte++;
}
```

```
    (*pages)++;  
} while (address < end);
```

将被映射的页框的页描述符地址 `page` 是从地址 `pages` 处的变量指向的数组项读得的。通过 `set_pte` 和 `mk_pte` 宏，把新页框的物理地址写进页表。把常量 4096（即一个页框的长度）加到 `address` 上之后，循环又重复执行。

注意，`map_vm_area()` 并不触及当前进程的页表。因此，当内核态的进程访问非连续内存区时，缺页发生，因为该内存区所对应的进程页表中的表项为空。然而，缺页处理程序要检查这个缺页线性地址是否在主内核页表中（也就是 `init_mm.pgd` 页全局目录和它的子页表，参见第二章“内核页表”一节）。一旦处理程序发现一个主内核页表含有这个线性地址的非空项，就把它值拷贝到相应的进程页表项中，并恢复进程的正常执行。这种机制将在第九章“缺页异常处理程序”一节描述。

除了 `vmalloc()` 函数之外，非连续内存区还能由 `vmalloc_32()` 函数分配，该函数与 `vmalloc()` 很相似，但是它只从 `ZONE_NORMAL` 和 `ZONE_DMA` 内存管理区中分配页框。

Linux 2.6 还特别提供了一个 `vmap()` 函数，它将映射非连续内存区中已经分配的页框：本质上，该函数接收一组指向页描述符的指针作为参数，调用 `get_vm_area()` 得到一个新 `vm_struct` 描述符，然后调用 `map_vm_area()` 来映射页框。因此该函数与 `vmalloc()` 相似，但是它不分配页框。

释放非连续内存区

`vfree()` 函数释放 `vmalloc()` 或 `vmalloc_32()` 创建的非连续内存区，而 `vunmap()` 函数释放 `vmap()` 创建的内存区。两个函数都使用同一个参数——将要释放的内存区的起始线性地址 `address`；它们都依赖于 `__vunmap()` 函数来做实质性的工作。

`__vunmap()` 函数接收两个参数：将要释放的内存区的起始地址的地址 `addr`，以及标志 `deallocate_pages`，如果被映射到内存区内的页框应当被释放到分区页框分配器（调用 `vfree()`）中，那么这个标志被置位，否则被清除（`vunmap()` 被调用）。该函数执行以下操作：

1. 调用 `remove_vm_area()` 函数得到 `vm_struct` 描述符的地址 `area`，并清除非连续内存区中的线性地址对应的内核的页表项。
2. 如果 `deallocate_pages` 被置位，函数扫描指向页描述符的 `area->pages` 指针数组；对于数组的每一个元素，调用 `__free_page()` 函数释放页框到分区页框分配器。此外，执行 `kfree(area->pages)` 来释放数组本身。

3. 调用 `kfree(area)` 来释放 `vm_struct` 描述符。

`remove_vm_area()` 函数执行如下循环：

```
write_lock(&vmlist_lock);
for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
    if (tmp->addr == addr) {
        unmap_vm_area(tmp);
        *p = tmp->next;
        break;
    }
}
write_unlock(&vmlist_lock);
return tmp;
```

内存区本身通过调用 `unmap_vm_area()` 来释放。这个函数接收单个参数，即指向内存区的 `vm_struct` 描述符的指针 `area`。它执行下列循环以进行 `map_vm_area()` 的反向操作：

```
address = area->addr;
end = address + area->size;
pgd = pgd_offset_k(address);
for (i = pgd_index(address); i <= pgd_index(end-1); i++) {
    next = (address + PGDIR_SIZE) & PGDIR_MASK;
    if (next <= address || next > end)
        next = end;
    unmap_area_pud(pgd, address, next - address);
    address = next;
    pgd++;
}
```

`unmap_area_pud()` 依次在循环中执行 `map_area_pud()` 的反操作：

```
do {
    unmap_area_pmd(pud, address, end-address);
    address = (address + PUD_SIZE) & PUD_MASK;
    pud++;
} while (address && (address < end));
```

`unmap_area_pmd()` 函数在循环中执行 `map_area_pmd()` 的反操作：

```
do {
    unmap_area_pte(pmd, address, end-address);
    address = (address + PMD_SIZE) & PMD_MASK;
    pmd++;
} while (address < end);
```

最后，`unmap_area_pte()` 在循环中执行 `map_area_pte()` 的反操作：

```
do {
    pte_t page = ptep_get_and_clear(pte);
    address += PAGE_SIZE;
```

```
pte++;
if (!pte_none(page) && !pte_present(page))
    printk("Whee... Swapped out page in kernel page table\n");
} while (address < end);
```

在每次循环过程中，`ptep_get_and_clear` 宏将 `pte` 指向的页表项设为 0。

与 `vmalloc()` 一样，内核修改主内核页全局目录和它的子页表中的相应项（参见第二章“内核页表”一节），但是映射第 4 个 GB 的进程页表的项保持不变。这是在情理之中的，因为内核永远也不会收回扎根于主内核页全局目录中的页上级目录、页中间目录和页表。

例如，假定内核态的进程访问一个随后要释放的非连续内存区。进程的页全局目录项等于主内核页全局目录中的相应项，由于在第九章“缺页异常处理程序”一节中所描述的机制，这些目录项指向相同的页上级目录、页中间目录和页表。`unmap_area_pte()` 函数只清除页表中的项（不回收页表本身）。进程对已释放非连续内存区的进一步访问必将由于空的页表项而触发缺页异常。但是，缺页处理程序会认为这样的访问是一个错误，因为主内核页表不包含有效的表项。



进程地址空间

在前一章我们已经看到，内核中的函数以相当直接了当的方式获得动态内存，这是通过调用以下几种函数中的一个达到的：① `__get_free_pages()` 或 `alloc_pages()` 从分区页框分配器中获得页框，② `kmem_cache_alloc()` 或 `kmalloc()` 使用 slab 分配器为专用或通用对象分配块③ 而 `vmalloc()` 或 `vmalloc_32()` 获得一块非连续的内存区。如果所请求的内存区得以满足，这些函数都返回一个页描述符地址或线性地址（即所分配动态内存区的起始地址）。

使用这些简单方法是基于以下两个原因：

- 内核是操作系统中优先级最高的成分。如果某个内核函数请求动态内存，那么，必定有正当的理由发出那个请求，因此，没有道理试图推迟这个请求。
- 内核信任自己。所有的内核函数都被假定是没有错误的，因此内核函数不必插入针对编程错误的任何保护措施。

当给用户态进程分配内存时，情况完全不同：

- 进程对动态内存的请求被认为是不紧迫的。例如，当进程的可执行文件被装入时，进程并不一定立即对所有的代码页进行访问。类似地，当进程调用 `malloc()` 以获得请求的动态内存时，也并不意味着进程很快就会访问所有所获得的内存。因此，一般来说，内核总是尽量推迟给用户态进程分配动态内存。
- 由于用户进程是不可信任的，因此，内核必须能随时准备捕获用户态进程引起的所有寻址错误。

本章我们会看到，内核使用一种新的资源成功实现了对进程动态内存的推迟分配。当用户态进程请求动态内存时，并没有获得请求的页框，而仅仅获得对一个新的线性地址区间的使用权，而这一线性地址区间就成为进程地址空间的一部分。这一区间叫做“线性区”（memory region）（译注1）。

在下一节，我们讨论进程是怎样看待动态内存的。然后，在“线性区”一节中描述进程地址空间的基本组成。接下来，我们仔细分析缺页异常处理程序在推迟给进程分配页框中所起的作用。然后，我们阐述内核怎样创建和删除进程的整个地址空间。最后，我们讨论与进程的地址空间管理有关的API和系统调用。

进程的地址空间

进程的地址空间（*address space*）由允许进程使用的全部线性地址组成。每个进程所看到的线性地址集合是不同的，一个进程所使用的地址与另外一个进程所使用的地址之间没有什么关系。后面我们会看到，内核可以通过增加或删除某些线性地址区间来动态地修改进程的地址空间。

内核通过所谓线性区的资源来表示线性地址区间，线性区是由起始线性地址、长度和一些访问权限来描述的。为了效率起见，起始地址和线性区的长度都必须是4096的倍数，以便每个线性区所识别的数据完全填满分配给它的页框。下面是进程获得新线性区的一些典型情况：

- 当用户在控制台输入一条命令时，shell进程创建一个新的进程去执行这个命令。结果是，一个全新的地址空间（也就是一组线性区）分配给了新进程（参见本章后面的“创建和删除进程的地址空间”一节和第二十章）。
- 正在运行的进程有可能决定装入一个完全不同的程序。在这种情况下，进程标识符仍然保持不变，可是在装入这个程序以前所使用的线性区却被释放，并有一组新的线性区被分配给这个进程（参见第二十章中的“exec函数”一节）。
- 正在运行的进程可能对一个文件（或它的一部分）执行“内存映射”。在这种情况下，内核给这个进程分配一个新的线性区来映射这个文件（参见第十六章中的“内存映射”一节）。
- 进程可能持续向它的用户态堆栈增加数据，直到映射这个堆栈的线性区用完为止。

译注1：“memory region”字面含义为内存区，但实际含义为线性地址空间中的一个区字段，在此把它译为线性区更贴切一些。其实际含义就是通常所指的虚拟内存中的一个区间，可以称为“虚存区”（Virtual Memory Area, VMA）。

在这种情况下，内核也许会决定扩展这个线性区的大小（参见本章后面的“缺页异常处理程序”一节）。

- 进程可能创建一个IPC共享线性区来与其他合作进程共享数据。在这种情况下，内核给这个进程分配一个新的线性区以实现这个方案（参见第十九章中的“IPC共享内存”一节）。
- 进程可能通过调用类似`malloc()`这样的函数扩展自己的动态区（堆）。结果是，内核可能决定扩展给这个堆所分配的线性区（参见本章后面的“堆的管理”一节）。

表9-1显示了与前面提到的任务相关的一些系统调用。除`brk()`在本章的最后进行讨论外，其余的系统调用在其他章节阐述。

表9-1：与创建、删除线性区相关的系统调用

系统调用	说明
<code>brk()</code>	改变进程堆的大小
<code>execve()</code>	装入一个新的可执行文件，从而改变进程的地址空间
<code>_exit()</code>	结束当前进程并撤销它的地址空间
<code>fork()</code>	创建一个新进程，并为它创建新的地址空间
<code>mmap()</code> , <code>mmap2()</code>	为文件创建一个内存映射，从而扩大进程的地址空间
<code>mremap()</code>	扩大或缩小线性区
<code>remap_file_pages()</code>	为文件创建非线性映射（参见第十六章）
<code>munmap()</code>	撤销对文件的内存映射，从而缩小进程的地址空间
<code>shmat()</code>	创建一个共享线性区
<code>shmdt()</code>	撤消一个共享线性区

我们会在“缺页异常处理程序”一节中看到，确定一个进程当前所拥有的线性区（即进程的地址空间）是内核的基本任务，因为这可以让缺页异常处理程序有效地区分引发这个异常处理程序的两种不同类型的无效线性地址：

- 由编程错误引发的无效线性地址。
- 由缺页引发的无效线性地址，即使这个线性地址属于进程的地址空间，但是对应于这个地址的页框仍然有待分配。

从进程的观点来看，后一种地址不是无效的，内核要利用这种缺页以实现请求调页：内核通过提供页框来处理这种缺页，并让进程继续执行。

内存描述符

与进程地址空间有关的全部信息都包含在一个叫做内存描述符 (*memory descriptor*) 的数据结构中 (译注 2), 这个结构的类型为 `mm_struct`, 进程描述符的 `mm` 字段就指向这个结构。内存描述符的字段如表 9-2 所示:

表 9-2: 内存描述符中的字段

类型	字段	说明
<code>struct</code>	<code>mmap</code>	指向线性区对象的链表头
<code>vm_area_struct *</code>		
<code>struct rb_root</code>	<code>mm_rb</code>	指向线性区对象的红-黑树的根
<code>struct</code>	<code>mmap_cache</code>	指向最后一个引用的线性区对象
<code>vm_area_struct *</code>		
<code>unsigned long (*)(*)()</code>	<code>get_unmapped_area</code>	在进程地址空间中搜索有效线性地址区间的方法
<code>void (*)(*)()</code>	<code>unmap_area</code>	释放线性地址区间时调用的方法
<code>unsigned long</code>	<code>mmap_base</code>	标识第一个分配的匿名线性区或文件内存映射的线性地址 (参见第二十章“程序段和进程的线性区”一节)
<code>unsigned long</code>	<code>free_area_cache</code>	内核从这个地址开始搜索进程地址空间中线性地址的空闲区间
<code>pgd_t *</code>	<code>pgd</code>	指向页全局目录
<code>atomic_t</code>	<code>mm_users</code>	次使用计数器
<code>atomic_t</code>	<code>mm_count</code>	主使用计数器
<code>int</code>	<code>map_count</code>	线性区的个数
<code>struct rw_semaphore</code>	<code>mmap_sem</code>	线性区的读/写信号量
<code>spinlock_t</code>	<code>page_table_lock</code>	线性区的自旋锁和页表的自旋锁
<code>struct list_head</code>	<code>mmlist</code>	指向内存描述符链表中的相邻元素
<code>unsigned long</code>	<code>start_code</code>	可执行代码的起始地址
<code>unsigned long</code>	<code>end_code</code>	可执行代码的最后地址
<code>unsigned long</code>	<code>start_data</code>	已初始化数据的起始地址
<code>unsigned long</code>	<code>end_data</code>	已初始化数据的最后地址
<code>unsigned long</code>	<code>start_brk</code>	堆的起始地址

译注 2: 实际上就是描述进程虚拟内存的数据结构。

表 9-2: 内存描述符中的字段 (续)

类型	字段	说明
unsigned long	brk	堆的当前最后地址
unsigned long	start_stack	用户态堆栈的起始地址
unsigned long	arg_start	命令行参数的起始地址
unsigned long	arg_end	命令行参数的最后地址
unsigned long	env_start	环境变量的起始地址
unsigned long	env_end	环境变量的最后地址
unsigned long	rss	分配给进程的页框数
unsigned long	anon_rss	分配给匿名内存映射的页框数
unsigned long	total_vm	进程地址空间的大小 (页数)
unsigned long	locked_vm	“锁住”而不能换出的页的个数 (参见第十七章)
unsigned long	shared_vm	共享文件内存映射中的页数
unsigned long	exec_vm	可执行内存映射中的页数
unsigned long	stack_vm	用户态堆栈中的页数
unsigned long	reserved_vm	在保留区中的页数或在特殊线性区中的页数
unsigned long	def_flags	线性区默认的访问标志
unsigned long	nr_ptes	this 进程的页表数
unsigned long []	saved_auxv	开始执行 ELF 程序时使用 (参见第二十章)
unsigned int	dumpable	表示是否可以产生内存信息转储的标志
cpumask_t	cpu_vm_mask	用于懒惰 TLB 交换的位掩码 (参见第二章)
mm_context_t	context	指向有关特定体系结构信息的表 (例如, 在 80x86 平台上的 LDT 地址)
unsigned long	swap_token_time	进程在这个时间将有资格获得交换标记 (参见第十七章“交换标记”一节)
char	recent_pagein	如果最近发生了主缺页, 则设置该标志
int	core_waiters	正在把进程地址空间的内容卸载到转储文件中的轻量级进程的数量 (参见本章后面“删除进程的地址空间”一节)
struct completion *	core_startup_done	指向创建内存转储文件时的补充原语 (参见第五章“补充原语”一节)

表 9-2: 内存描述符中的字段 (续)

类型	字段	说明
struct completion	core_done	创建内存转储文件时使用的补充原语
rwlock_t	ioctx_list_lock	用于保护异步 I/O 上下文链表的锁 (参见第十六章)
struct kiocx *	ioctx_list	异步 I/O 上下文链表 (参见第十六章)
struct kiocx	default_kiocx	默认的异步 I/O 上下文 (参见第十六章)
unsigned long	hiwater_rss	进程所拥有的最大页框数
unsigned long	hiwater_vm	进程线性区中的最大页数

所有的内存描述符存放在一个双向链表中。每个描述符在 `mmlist` 字段存放链表相邻元素的地址。链表的第一个元素是 `init_mm` 的 `mmlist` 字段, `init_mm` 是初始化阶段进程 0 所使用的内存描述符。`mmlist_lock` 自旋锁保护多处理器系统对链表的同时访问。

`mm_users` 字段存放共享 `mm_struct` 数据结构的轻量级进程的个数 (参见第三章“`clone()`、`fork()`及`vfork()`系统调用”一节)。`mm_count` 字段是内存描述符的主使用计数器,在 `mm_users` 次使用计数器中的所有用户在 `mm_count` 中只作为一个单位。每当 `mm_count` 递减时,内核都要检查它是否变为 0,如果是,就要解除这个内存描述符,因为不再有用户使用它。

我们用一个例子来解释 `mm_users` 和 `mm_count` 之间的不同。考虑一个内存描述符由两个轻量级进程共享。它的 `mm_users` 字段通常存放的值为 2,而 `mm_count` 字段存放的值为 1 (两个所有者进程算作一个)。

如果把内存描述符暂时借给一个内核线程 (参见下一节),那么,内核就增加 `mm_count`。这样,即使两个轻量级进程都死亡,且 `mm_users` 字段变为 0,这个内存描述符也不被释放,直到内核线程使用完为止,因为 `mm_count` 字段仍然大于 0。

如果内核想确保内存描述符在一个长操作的中间不被释放,那么,就应该增加 `mm_users` 字段而不是 `mm_count` 字段的值 (这正是函数 `try_to_unuse()` 所做的事,参见第十七章“激活和禁用交换区”一节)。最终的结果是相同的,因为 `mm_users` 的增加确保了 `mm_count` 不变为 0,即使拥有这个内存描述符的所有轻量级进程全部死亡。

`mm_alloc()` 函数用来获得一个新的内存描述符。由于这些描述符被保存在 slab 分配器高速缓存中,因此, `mm_alloc()` 调用 `kmem_cache_alloc()` 来初始化新的内存描述符,并把 `mm_count` 和 `mm_users` 字段都置为 1。

相反, `mmapput()` 函数递减内存描述符的 `mm_users` 字段。如果该字段变为 0, 这个函数就释放局部描述符表、线性区描述符 (参见本章后面的部分) 及由内存描述符所引用的页表, 并调用 `mmdrop()`。后一个函数把 `mm_count` 字段减 1, 如果该字段变为 0, 就释放 `mm_struct` 数据结构。

`mmap`、`mm_rb`、`mmlist` 和 `mmap_cache` 字段将在下一节进行讨论。

内核线程的内存描述符

内核线程仅运行在内核态, 因此, 它们永远不会访问低于 `TASK_SIZE` (等于 `PAGE_OFFSET`, 通常为 `0xc0000000`) 的地址。与普通进程相反, 内核线程不用线性区, 因此, 内存描述符的很多字段对内核线程是没有意义的。

因为大于 `TASK_SIZE` 线性地址的相应页表项都应该总是相同的, 因此, 一个内核线程到底使用什么样的页表集根本就没有什么关系。为了避免无用的 TLB 和高速缓存刷新, 内核线程使用一组最近运行的普通进程的页表。结果, 在每个进程描述符中包含了两种内存描述符指针: `mm` 和 `active_mm`。

进程描述符中的 `mm` 字段指向进程所拥有的内存描述符, 而 `active_mm` 字段指向进程运行时所使用的内存描述符。对于普通进程而言, 这两个字段存放相同的指针。但是, 内核线程不拥有任何内存描述符, 因此, 它们的 `mm` 字段总是为 `NULL`。当内核线程得以运行时, 它的 `active_mm` 字段被初始化为前一个运行进程的 `active_mm` 值 (参看第七章 “`schedule()` 函数” 一节)。

然而, 事情有点复杂。只要处于内核态的一个进程为 “高端” 线性地址 (高于 `TASK_SIZE`) 修改了页表项, 那么, 它也就应当更新系统中所有进程页表集中的相应表项。事实上, 一旦内核态的一个进程进行了设置, 那么, 映射应该对内核态的其他所有进程都有效。触及所有进程的页表集合是相当费时的操作, 因此, Linux 采用一种延迟方式。

我们在第八章 “非连续内存区管理” 一节已经提到这种延迟方式: 每当一个高端地址必须被重新映射时 (一般是通过 `vmalloc()` 或 `vfree()`), 内核就更新根目录在 `swapper_pg_dir` 主内核页全局目录 (参见第二章 “内核页表” 一节) 中的常规页表集合。这个页全局目录由主内存描述符 (*master memory descriptor*) 的 `pgd` 字段所指向, 而主内存描述符存放于 `init_mm` 变量 (注 1)。

注 1: 我们在第三章 “内核线程” 一节中提到, `swapper` 内核线程在初始化阶段使用 `init_mm`。但是, 一旦初始化完成, `swapper` 再不使用这个内存描述符。

在随后的“处理非连续内存区访问”一节，我们将描述缺页处理程序如何在必要时维护存放在常规页表中的扩展信息。

线性区

Linux 通过类型为 `vm_area_struct` 的对象实现线性区，它的字段如表 9-3 所示（注 2）。

表 9-3：线性区对象的字段

类型	字段	说明
<code>struct mm_struct *</code>	<code>vm_mm</code>	指向线性区所在的内存描述符
<code>unsigned long</code>	<code>vm_start</code>	线性区内的第一个线性地址
<code>unsigned long</code>	<code>vm_end</code>	线性区之后的第一个线性地址
<code>struct vm_area_struct *</code>	<code>vm_next</code>	进程链表中的下一个线性区
<code>pgprot_t</code>	<code>vm_page_prot</code>	线性区中页框的访问许可权
<code>unsigned long</code>	<code>vm_flags</code>	线性区的标志
<code>struct rb_node</code>	<code>vm_rb</code>	用于红-黑树的数据（参见本章后面）
<code>union</code>	<code>shared</code>	链接到反映射所使用的数据结构（参见第十七章“对映射页的反映射”一节）
<code>struct list_head</code>	<code>anon_vma_node</code>	指向匿名线性区链表的指针（参见第十七章“映射页的反向映射”一节）
<code>struct anon_vma *</code>	<code>anon_vma</code>	指向 <code>anon_vma</code> 数据结构的指针（参见第十七章“映射页的反向映射”一节）
<code>struct vm_operations_struct *</code>	<code>vm_ops</code>	指向线性区的方法
<code>unsigned long</code>	<code>vm_pgoff</code>	在映射文件中的偏移量（参见第十六章）。对匿名页，它等于 0 或 <code>vm_start/PAGE_SIZE</code> （参见第十七章）
<code>struct file *</code>	<code>vm_file</code>	指向映射文件的文件对象（如果有的话）
<code>void *</code>	<code>vm_private_data</code>	指向内存区的私有数据
<code>unsigned long</code>	<code>vm_truncate_count</code>	释放非线性文件内存映射中的一个线性地址区间时使用

注 2：我们对 NUMA 系统中使用的一些附加字段不予说明。

每个线性区描述符表示一个线性地址区间。vm_start字段包含区间的第一个线性地址，而vm_end字段包含区间之外的第一个线性地址。vm_end - vm_start表示线性区的长度。vm_mm字段指向拥有这个区间的进程的mm_struct内存描述符。我们稍后将描述vm_area_struct的其他字段。

进程所拥有的线性区从来不重叠，并且内核尽力把新分配的线性区与紧邻的现有线性区进行合并。如果两个相邻区的访问权限相匹配，就能把它们合并在一起。

如图9-1所示，当一个新的线性地址区间加入到进程的地址空间时，内核检查一个已经存在的线性区是否可以扩大（情况a）。如果不能，就创建一个新的线性区（情况b）。类似地，如果从进程的地址空间删除一个线性地址区间，内核就要调整受影响的线性区大小（情况c）。有些情况下，调整大小迫使一个线性区被分成两个更小的部分（情况d）（注3）。

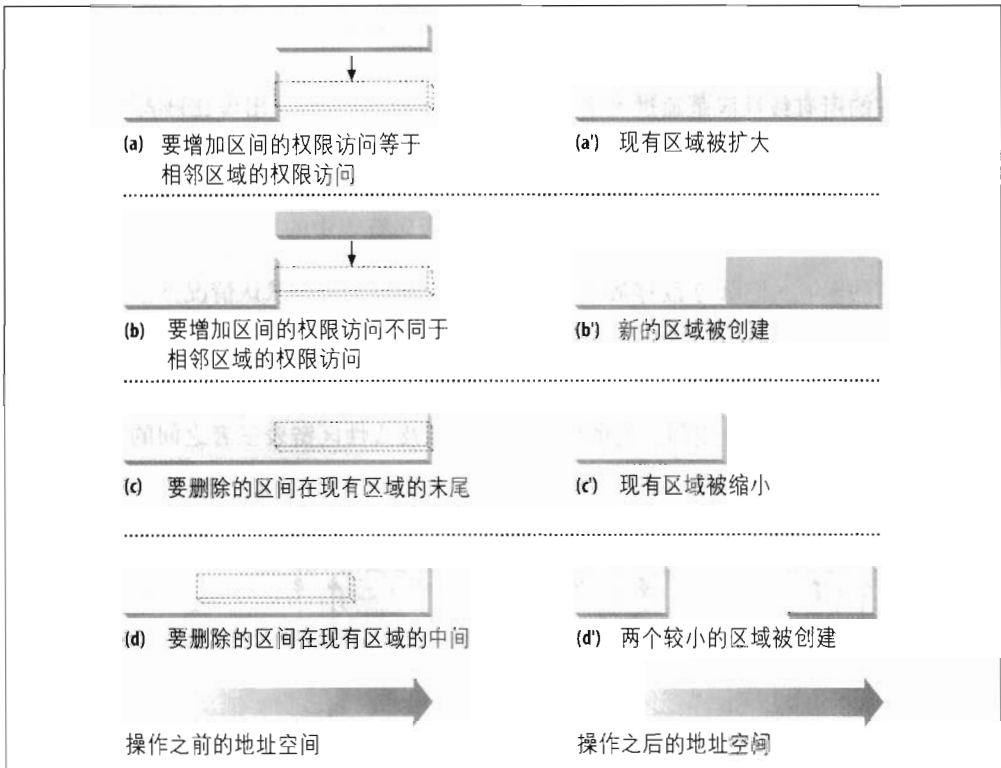


图9-1：增加或删除一个线性地址区间

注3：从理论上说，删除一个线性地址区间可能会失败，因为没有空闲的内存给新的内存描述符使用。

`vm_ops` 字段指向 `vm_operations_struct` 数据结构，该结构中存放的是线性区的方法。只有如表 9-4 所示的 4 种方法可应用于 UMA 系统。

表 9-4：作用于线性区的方法

方法	说明
<code>open</code>	当把线性区增加到进程所拥有的线性区集合时调用
<code>close</code>	当从进程所拥有的线性区集合删除线性区时调用
<code>nopage</code>	当进程试图访问 RAM 中不存在的一个页，但该页的线性地址属于线性区时，由缺页异常处理程序调用（参见后面“缺页异常处理程序”一节）
<code>populate</code>	设置线性区的线性地址（预缺页）所对应的页表项时调用。主要用于非线性文件内存映射

线性区数据结构

进程所拥有的所有线性区是通过一个简单的链表链接在一起的。出现在链表中的线性区是按内存地址的升序排列的；不过，每两个线性区可以由未用的内存地址区隔开。每个 `vm_area_struct` 元素的 `vm_next` 字段指向链表的下一个元素。内核通过进程的内存描述符的 `mmap` 字段来查找线性区，其中 `mmap` 字段指向链表中的第一个线性区描述符。

内存描述符的 `map_count` 字段存放进程所拥有的线性区数目。默认情况下，一个进程可以最多拥有 65536 个不同的线性区，系统管理员可以通过写 `/proc/sys/vm/max_map_count` 文件来修改这个限定值。

图 9-2 显示了进程的地址空间、它的内存描述符以及线性区链表三者之间的关系。

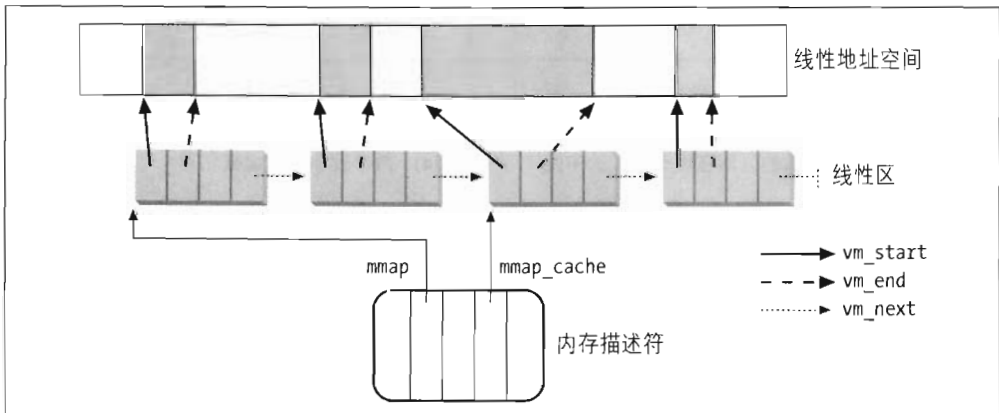


图 9-2：与进程地址空间相关的描述符

内核频繁执行的一个操作就是查找包含指定线性地址的线性区。由于链表是经过排序的，因此，只要在指定线性地址之后找到一个线性区，搜索就可以结束。

然而，仅当进程的线性区非常少时使用这种链表才是很方便的，比如说只有一二十个线性区。在链表中查找元素、插入元素、删除元素涉及许多操作，这些操作所花费的时间与链表的长度成线性比例。

尽管多数的Linux进程使用的线性区非常少，但是诸如面向对象的数据库，或malloc()的专用调试器那样过于庞大的大型应用程序可能会有成百上千的线性区。在这种情况下，线性区链表的管理变得非常低效，因此，与内存相关的系统调用的性能就降低到令人无法忍受的程度。

因此，Linux 2.6把内存描述符存放在叫做红-黑树 (red-black tree) 的数据结构中。在红-黑树中，每个元素(或节点)通常有两个孩子：左孩子和右孩子。树中的元素被排序。对每个节点N，N的左子树上的所有元素都排在N之前，相反，N的右子树上的所有元素都排在N之后 [如图9-3 (a) 所示]；节点的关键字被写入节点内部。此外，红-黑树必须满足下列4条规则：

1. 每个节点必须或为黑或为红。
2. 树的根必须为黑。
3. 红节点的孩子必须为黑。
4. 从一个节点到后代叶子节点的每个路径都包含相同数量的黑节点。当统计黑节点个数时，空指针也算作黑节点。

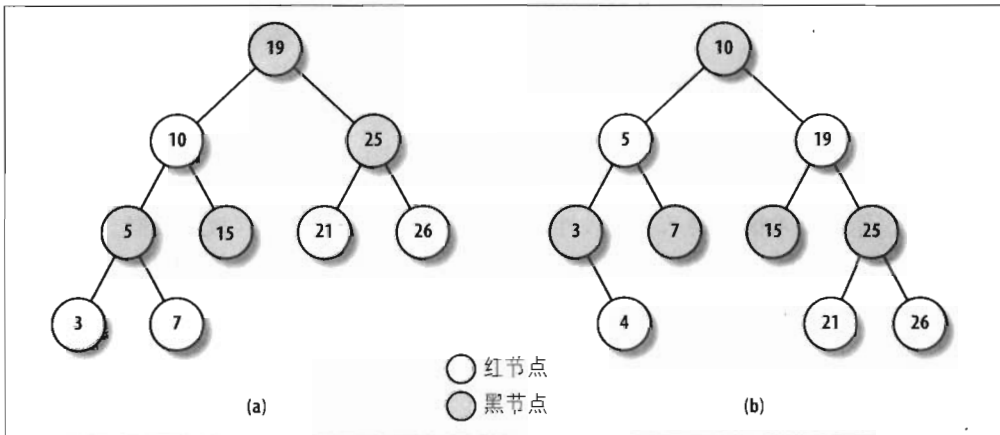


图 9-3：红-黑树实例

这4条规则确保具有 n 个内部节点的任何红-黑树其高度最多为 $2 \times \log(n+1)$ 。

在红-黑树中搜索一个元素因此而变得非常高效,因为其操作的执行时间与树大小的对数成线性比例。换句话说,双倍的线性区个数只多增加一次循环。

在红-黑树中插入和删除一个元素也是高效的,因为算法能很快地遍历树以确定插入元素的位置或删除元素的位置。任何新节点必须作为一个叶子插入并着成红色。如果操作违背了上述规则,就必须移动或重新着色树的几个节点。

例如,假如值为4的一个元素必须插入到图9-3(a)所示的红-黑树中。它的正确位置是关键值为3的节点的右孩子,但是,一旦把它插入,值为3的红节点就具有红孩子,因此而违背了规则3。为了满足这条规则,值为3、4、7的节点颜色就得改变。但是,这种操作又会违背规则4,因此,算法在以关键值为19的节点为根节点的子树上执行“旋转”操作,产生如图9-3(b)所示的新红-黑树。这看起来较复杂,但是,在红-黑树上插入或删除一个元素只需要少量的操作——这个数与树大小的对数成线性比例。

因此,为了存放进程的线性区, Linux 既使用了链表,也使用了红-黑树。这两种数据结构包含指向同一线性区描述符的指针,当插入或删除一个线性区描述符时,内核通过红-黑树搜索前后元素,并用搜索结果快速更新链表而不用扫描链表。

链表的头由内存描述符的 `mmmap` 字段所指向。任何线性区对象都在 `vm_next` 字段存放指向链表下一个元素的指针。红-黑树的首部由内存描述符的 `mm_rb` 字段所指向。任何线性区对象都在类型为 `rb_node` 的 `vm_rb` 字段中存放节点颜色以及指向双亲、左孩子和右孩子的指针。

一般来说,红-黑树用来确定含有指定地址的线性区,而链表通常在扫描整个线性区集合时来使用。

线性区访问权限

在讲述下一部分以前,我们先阐明页与线性区之间的关系。正如第二章中所提到的,我们使用“页”这个术语既表示一组线性地址又表示这组地址中所存放的数据。尤其是,我们把介于0~4095之间的线性地址区间称为第0页,介于4096~8191之间的线性地址区间称为第1页,依此类推。因此每个线性区都由一组号码连续的页所构成。

在前几章我们已经讨论了与页相关的两种标志:

- 在每个页表项中存放的几个标志,如: `Read/Write`、`Present` 或 `User/Supervisor` (参见第二章中的“常规分页”一节)。

- 存放在每个页描述符 flags 字段中的一组标志（参见第八章中的“页框管理”一节）。

第一种标志由 80x86 硬件用来检查能否执行所请求的寻址类型；第二种标志由 Linux 用于许多不同的目的（见表 8-2）。

现在介绍第三种标志，即与线性区的页相关的那些标志。它们存放在 `vm_area_struct` 描述符的 `vm_flags` 字段中（见表 9-5）。一些标志给内核提供有关这个线性区全部页的信息，例如它们包含有什么内容，进程访问每个页的权限是什么。另外的标志描述线性区自身，例如它应该如何增长。

表 9-5：线性区标志

标志名	说明
<code>VM_READ</code>	页是可读的
<code>VM_WRITE</code>	页是可写的
<code>VM_EXEC</code>	页是可执行的
<code>VM_SHARED</code>	页可以由几个进程共享
<code>VM_MAYREAD</code>	可以设置 <code>VM_READ</code> 标志
<code>VM_MAYWRITE</code>	可以设置 <code>VM_WRITE</code> 标志
<code>VM_MAYEXEC</code>	可以设置 <code>VM_EXEC</code> 标志
<code>VM_MAYSHARE</code>	可以设置 <code>VM_SHARE</code> 标志
<code>VM_GROWSDOWN</code>	线性区可以向低地址扩展
<code>VM_GROWSUP</code>	线性区可以向高地址扩展
<code>VM_SHM</code>	线性区用于 IPC 的共享内存
<code>VM_DENYWRITE</code>	线性区映射一个不能打开用于写的文件
<code>VM_EXECUTABLE</code>	线性区映射一个可执行文件
<code>VM_LOCKED</code>	线性区中的页被锁住，且不能换出
<code>VM_IO</code>	线性区映射设备的 I/O 地址空间
<code>VM_SEQ_READ</code>	应用程序顺序地访问页
<code>VM_RAND_READ</code>	应用程序以真正的随机顺序访问页
<code>VM_DONTCOPY</code>	当创建一个新进程时不拷贝线性区
<code>VM_DONTEXPAND</code>	通过 <code>mremap()</code> 系统调用禁止线性区扩展
<code>VM_RESERVED</code>	线性区是特殊的（如：它映射某个设备的 I/O 地址空间），因此它的页不能被交换出去
<code>VM_ACCOUNT</code>	创建 IPC 共享线性区时检查是否有足够的空闲内存用于映射（参见第十九章）

表 9-5: 线性区标志 (续)

标志名	说明
VM_HUGETLB	通过扩展分页机制处理线性区中的页 (参见第二章“扩展分页”一节)
VM_NONLINEAR	线性区实现非线性文件映射

线性区描述符所包含的页访问权限可以任意组合。例如,存在这样一种可能性,允许一个线性区中的页可以执行但是不可以读取。为了有效地实现这种保护方案,与线性区的页相关的访问权限(读、写及执行)必须被复制到相应的所有表项中,以便由分页单元直接执行检查。换句话说,页访问权限表示何种类型的访问应该产生一个缺页异常。稍后我们会看到, Linux 委派缺页处理程序查找导致缺页的原因,因为缺页处理程序实现了许多页处理策略。

页表标志的初值(正如我们看到的,同一线性区所有页标志的初值必须一样)存放在 `vm_area_struct` 描述符的 `vm_page_prot` 字段中。当增加一个页时,内核根据 `vm_page_prot` 字段的值设置相应页表项中的标志。

然而,并不能把线性区的访问权限直接转换成页保护位,这是因为:

- 在某些情况下,即使由相应线性区描述符的 `vm_flags` 字段所指定的某个页的访问权限允许对该页进行访问,但是,对该页的访问还是应当产生一个缺页异常。例如,我们在本章后面的“写时复制”一节会看到,内核可能决定把属于两个不同进程的两个完全一样的可写私有页(它的 `VM_SHARE` 标志被清 0) 存入同一个页框中;在这种情况下,无论哪一个进程试图改动这个页都应当产生一个异常。
- 正如在第二章中提到的,80x86 处理器的页表仅有两个保护位,即 `Read/Write` 和 `User/Supervisor` 标志。此外,一个线性区所包含的任何一个页的 `User/Supervisor` 标志必须总置为 1,因为用户态进程必须总能够访问其中的页。
- 启用 PAE 的新近 Intel Pentium 4 微处理器,在所有 64 位页表项中支持 `NX(No eXecute)` 标志。

如果内核没有被编译成支持 PAE,那么 Linux 采取以下规则以克服 80x86 微处理器的硬件限制:

- 读访问权限总是隐含着执行访问权限,反之亦然。
- 写访问权限总是隐含着读访问权限。

反之,如果内核被编译成支持 PAE,而且 CPU 有 `NX` 标志, Linux 就采取不同的规则:

- 执行访问权限总是隐含着读访问权限。
- 写访问权限总是隐含着读访问权限。

此外，为了做到在“写时复制”技术中适当地推迟页框的分配（参见本章后面的内容），只要相应的页不是由多个进程所共享，那么，这种页框都是写保护的。

因此，要根据以下规则精简由读、写、执行和共享访问权限的16种可能组合：

- 如果页具有写和共享两种访问权限，那么，Read/Write位被设置为1。
- 如果页具有读或执行访问权限，但是既没有写也没有共享访问权限，那么，Read/Write位被清0。
- 如果支持NX位，而且页没有执行访问权限，那么，把NX位设置为1。
- 如果页没有任何访问权限，那么，Present位被清0，以便每次访问都产生一个缺页异常。然而，为了把这种情况与真正的页框不存在的情况相区分，Linux还把Page size位置为1（注4）。

访问权限的每种组合所对应的精简后的保护位存放在`protection_map`数组的16个元素中。

线性区的处理

对控制内存处理所用的数据结构和状态信息有了基本理解以后，我们来看一组对线性区描述符进行操作的低层函数。这些函数应当被看作简化了`do_map()`和`do_unmap()`实现的辅助函数。这两个函数将在本章后面的“分配线性地址区间”和“释放线性地址区间”两节中进行描述，它们分别扩大或者缩小进程的地址空间。这两个函数所处的层次比我们在这里所考虑函数的层次要高一些，它们并不接受线性区描述符作为参数，而是使用一个线性地址区间的起始地址、长度和访问限权作为参数。

查找给定地址的最邻近区：`find_vma()`

`find_vma()`函数有两个参数：进程内存描述符的地址`mm`和线性地址`addr`。它查找线性区的`vm_end`字段大于`addr`的第一个线性区的位置，并返回这个线性区描述符的地址；如果没有这样的线性区存在，就返回一个NULL指针。注意由`find_vma()`函数所选择的线性区并不一定要包含`addr`，因为`addr`可能位于任何线性区之外。

注4： 你可能认为Page size位的这种用法并不正当，因为这个位本来是表示实际页的大小。但是，Linux可以侥幸逃脱这种骗局，因为80×86芯片在页目录项中检查Page size位，而不是在页表的表项中检查该位。

每个内存描述符包含一个 `mmap_cache` 字段，这个字段保存进程最后一次引用线性区的描述符地址。引进这个附加的字段是为了减少查找一个给定线性地址所在线性区而花费的时间。程序中引用地址的局部性使下面这种情况出现的可能性很大：如果检查的最后一个线性地址属于某一给定的线性区，那么，下一个要检查的线性地址也属于这一个线性区。

因此，该函数一开始就检查由 `mmap_cache` 所指定的线性区是否包含 `addr`。如果是，就返回这个线性区描述符的指针：

```
vma = mm->mmap_cache;
if (vma && vma->vm_end > addr && vma->vm_start <= addr)
    return vma;
```

否则，必须扫描进程的线性区，并在红-黑树中查找线性区：

```
rb_node = mm->mm_rb.rb_node;
vma = NULL;
while (rb_node) {
    vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
    if (vma_tmp->vm_end > addr) {
        vma = vma_tmp;
        if (vma_tmp->vm_start <= addr)
            break;
        rb_node = rb_node->rb_left;
    } else
        rb_node = rb_node->rb_right;
}
if (vma)
    mm->mmap_cache = vma;
return vma;
```

函数使用的宏 `rb_entry`，从指向红-黑树中一个节点的指针导出相应线性区描述符的地址。

函数 `find_vma_prev()` 与 `find_vma()` 类似，不同的是它把函数选中的前一个线性区描述符的指针赋给附加参数 `ppre`。

最后，函数 `find_vma_prepare()` 确定新叶子节点在与给定线性地址对应的红-黑树中的位置，并返回前一个线性区的地址和要插入的叶子节点的父节点的地址。

查找一个与给定的地址区间相重叠的线性区：`find_vma_intersection()`

`find_vma_intersection()` 函数查找与给定的线性地址区间相重叠的第一个线性区。`mm` 参数指向进程的内存描述符，而线性地址 `start_addr` 和 `end_addr` 指定这个区间。

```
vma = find_vma(mm, start_addr);
if (vma && end_addr <= vma->vm_start)
    vma = NULL;
return vma;
```

如果没有这样的线性区存在，函数就返回一个NULL指针。准确地说，如果find_vma()函数返回一个有效的地址，但是所找到的线性区是从这个线性地址区间的末尾开始的，vma就被置为NULL。

查找一个空闲的地址区间：get_unmapped_area()

函数get_unmapped_area()搜查进程的地址空间以找到一个可以使用的线性地址区间。len参数指定区间的长度，而非空的addr参数指定必须从哪个地址开始进行查找。如果查找成功，函数返回这个新区间的起始地址；否则返回错误码-ENOMEM。

如果参数addr不等于NULL，函数就检查所指定的地址是否在用户态空间并与页边界对齐。接下来，函数根据线性地址区间是否应该用于文件内存映射或匿名内存映射，调用两个方法(get_unmapped_area文件操作和内存描述符的get_unmapped_area方法)中的一个。在前一种情况下，函数执行get_unmapped_area文件操作，在第十六章将对此进行讨论。

在第二种情况下，函数执行内存描述符的get_unmapped_area方法。根据进程的线性区类型，由函数arch_get_unmapped_area()或arch_get_unmapped_area_topdown()实现get_unmapped_area方法。在第二十章“程序段和进程的线性区”一节，我们将会看到通过系统调用mmap()，每个进程都可能获得两种不同形式的线性区：一种从线性地址0x40000000开始并向高端地址增长，另一种正好从用户态堆栈开始并向低端地址增长。

现在我们讨论函数arch_get_unmapped_area()，在分配从低端地址向高端地址移动的线性区时使用这个函数。它本质上等价于下面的代码片段：

```
if (len > TASK_SIZE)
    return -ENOMEM;
addr = (addr + 0xfff) & 0xfffff000;
if (addr && addr + len <= TASK_SIZE) {
    vma = find_vma(current->mm, addr);
    if (!vma || addr + len <= vma->vm_start)
        return addr;
}
start_addr = addr = mm->free_area_cache;
for (vma = find_vma(current->mm, addr); ; vma = vma->vm_next) {
    if (addr + len > TASK_SIZE) {
        if (start_addr == (TASK_SIZE/3+0xfff)&0xfffff000)
            return -ENOMEM;
        start_addr = addr = (TASK_SIZE/3+0xfff)&0xfffff000;
        vma = find_vma(current->mm, addr);
    }
    if (!vma || addr + len <= vma->vm_start) {
        mm->free_area_cache = addr + len;
        return addr;
    }
}
```

```

    }
    addr = vma->vm_end;
}

```

函数首先检查区间的长度是否在用户态下线性地址区间的限长 `TASK-SIZE` (通常为 3GB) 之内。如果 `addr` 不为 0, 函数就试图从 `addr` 开始分配区间。为了安全起见, 函数把 `addr` 的值调整为 4KB 的倍数。

如果 `addr` 等于 0 或前面的搜索失败, 函数 `arch_get_unmapped_area()` 就扫描用户态线性地址空间以查找一个可以包含新区的足够大的线性地址范围, 但任何已有的线性区都不包括这个地址范围。为了提高搜索的速度, 让搜索从最近被分配的线性区后面的线性地址开始。把内存描述符的字段 `mm->free_area_cache` 初始化为用户态线性地址空间的三分之一 (通常是 1GB), 并在以后创建新线性区时对它进行更新。如果函数找不到一个合适的线性地址范围, 就从用户态线性地址空间的三分之一的开始处重新开始搜索: 其实, 用户态线性地址空间的三分之一是为有预定义起始线性地址的线性区 (典型的是可执行文件的正文段、数据段和 `bss` 段, 参见第二十章) 而保留的。

函数调用 `find_vma()` 以确定搜索起点之后第一个线性区终点的位置。可能出现三种情况:

- 如果所请求的区间大于正待扫描的线性地址空间部分 (`addr + len > TASK-SIZE`), 函数就从用户态地址空间的三分之一处重新开始搜索, 如果已经完成第二次搜索, 就返回 `-ENOMEM` (没有足够的线性地址空间来满足这个请求)。
- 刚刚扫描过的线性区后面的空闲区没有足够的大小 (`vma != NULL && vma->vm_start < addr + len`)。此时, 继续考虑下一个线性区。
- 如果以上两种情况都没有发生, 则找到一个足够大的空闲区, 此时, 函数返回 `addr`。

向内存描述符链表中插入一个线性区: `insert_vm_struct()`

`insert_vm_struct()` 函数在线性区对象链表和内存描述符的红-黑树中插入一个 `vm_area_struct` 结构。这个函数使用两个参数: `mm` 指定进程内存描述符的地址, `vmp` 指定要插入的 `vm_area_struct` 对象的地址。线性区对象的 `vm_start` 和 `vm_end` 字段必定已经初始化过。该函数调用 `find_vma_prepare()` 在红-黑树 `mm->mm_rb` 中查找 `vma` 应该位于何处。然后, `insert_vm_struct()` 又调用 `vma_link()` 函数, 后者依次执行以下操作:

1. 在 `mm->mmap` 所指向的链表中插入线性区。
2. 在红-黑树 `mm->mm_rb` 中插入线性区。

3. 如果线性区是匿名的,就把它插入以相应的 `anon_vma` 数据结构作为头节点的链表中(参见第十七章“匿名页的反向映射一节”)。
4. 递增 `mm->map_count` 计数器。

如果线性区包含一个内存映射文件,则 `vma_link()` 函数执行在第十七章描述的其他任务。

`__vma_unlink()` 函数接收的参数为一个内存描述符地址 `mm` 和两个线性区对象地址 `vma` 和 `prev`。两个线性区都应当属于 `mm`, `prev` 应当在线性区的排序中位于 `vma` 之前。该函数从内存描述符链表和红-黑树中删除 `vma`, 如果 `mm->mmap_cache` (存放刚被引用的线性区) 字段指向刚被删除的线性区, 则还要对 `mm->mmap_cache` 进行更新。

分配线性地址区间

现在让我们讨论怎样分配一个新的线性地址区间。为了做到这点, `do_mmap()` 函数为当前进程创建并初始化一个新的线性区。不过, 分配成功之后, 可以把这个新的线性区与进程已有的其他线性区进行合并。

`do_mmap()` 函数使用下面的参数:

`file` 和 `offset`

如果新的线性区将把一个文件映射到内存, 则使用文件描述符指针 `file` 和文件偏移量 `offset`。这个主题将在第十六章进行讨论。在这一节中, 我们假定不需要内存映射, 并且 `file` 和 `offset` 都为空。

`addr`

这个线性地址指定从何处开始查找一个空闲的区间。

`len`

线性地址区间的长度。

`prot`

这个参数指定这个线性区所包含页的访问权限。可能的标志有 `PROT_READ`、`PROT_WRITE`、`PROT_EXEC` 和 `PROT_NONE`。前三个标志与标志 `VM_READ`、`VM_WRITE` 及 `VM_EXEC` 的意义一样。`PROT_NONE` 表示进程没有以上三个访问权限中的任意一个。

`flag`

这个参数指定线性区的其他标志:

`MAP_GROWSDOWN`、`MAP_LOCKED`、`MAP_DENYWRITE` 和 `MAP_EXECUTABLE`

它们的含义与表 9-5 中所列出标志的含义相同。

MAP_SHARED 和 MAP_PRIVATE

前一个标志指定线性区中的页可以被几个进程共享；后一个标志作用相反。这两个标志都指向 `vm_area_struct` 描述符中的 `VM_SHARED` 标志。

MAP_FIXED

区间的起始地址必须是由参数 `addr` 所指定的。

MAP_ANONYMOUS

没有文件与这个线性区相关联（参见第十六章）。

MAP_NORESERVE

函数不必预先检查空闲页框的数目。

MAP_POPULATE

函数应该为线性区建立的映射提前分配需要的页框。该标志仅对映射文件的线性区（参见第十六章）和 IPC 共享的线性区（参见第十九章）有意义。

MAP_NONBLOCK

只有在 `MAP_POPULATE` 标志置位时才有意义：提前分配页框时，函数肯定不阻塞。

`do_mmap()` 函数对 `offset` 的值进行一些初步检查，然后执行 `do_mmap_pgoff()` 函数。本章假设新的线性地址区间映射的不是磁盘文件（在第十六章将详细讨论文件内存映射）。这里仅对实现匿名线性区的 `do_mmap_pgoff()` 函数进行说明。

1. 检查参数的值是否正确，所提的请求是否能被满足。尤其是要检查以下不能满足请求的条件：
 - 线性地址区间的长度为 0 或者包含的地址大于 `TASK_SIZE`。
 - 进程已经映射了过多的线性区，因此 `mm` 内存描述符的 `map_count` 字段的值超过了允许的最大值。
 - `flag` 参数指定新线性地址区间的页必须被锁在 RAM 中，但不允许进程创建上锁的线性区，或者进程加锁页的总数超过了保存在进程描述符 `signal->rlim[RLIMIT_MEMLOCK].rlim_cur` 字段中的阈值。

如果以上情况中的任何一个成立，则 `do_mmap_pgoff()` 函数终止并返回一个负值。如果线性地址区间的长度为 0，则函数不执行任何操作就返回。

2. 调用 `get_unmapped_area()` 获得新线性区的线性地址区间（参见上一节“线性区的处理”）
3. 通过把存放在 `prot` 和 `flags` 参数中的值进行组合来计算新线性区描述符的标志：

```

vm_flags = calc_vm_prot_bits(prot, flags) |
           calc_vm_flag_bits(prot, flags) |
           mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
if (flags & MAP_SHARED)
    vm_flags |= VM_SHARED | VM_MAYSHARE;

```

只有在 `prot` 中设置了相应的 `PROT_READ`、`PROT_WRITE` 和 `PROT_EXEC` 标志，`calc_vm_prot_bits()` 函数才在 `vm_flags` 中设置 `VM_READ`、`VM_WRITE` 和 `VM_EXEC` 标志；只有在 `flags` 设置了相应的 `MAP_GROWSDOWN`、`MAP_DENYWRITE`、`MAP_EXECUTABLE` 和 `MAP_LOCKED` 标志，`calc_vm_flag_bits()` 也才在 `vm_flags` 中设置 `VM_GROWSDOWN`、`VM_DENYWRITE`、`VM_EXECUTABLE` 和 `VM_LOCKED` 标志。在 `vm_flags` 中还有几个标志被置为 1：`VM_MAYREAD`、`VM_MAYWRITE`、`VM_MAYEXEC`，在 `mm->def_flags`（注 5）中所有线性区的默认标志，以及如果线性区的页与其他进程共享时的 `VM_SHARED` 和 `VM_MAYSHARE`。

4. 调用 `find_vma_prepare()` 确定处于新区间之前的线性区对象的位置，以及在红-黑树中新线性区的位置：

```

for (;;) {
    vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
    if (!vma || vma->vm_start >= addr + len)
        break;
    if (do_munmap(mm, addr, len))
        return -ENOMEM;
}

```

`find_vma_prepare()` 函数也检查是否还存在与新区间重叠的线性区。这种情况发生在函数返回一个非空的地址，这个地址指向一个线性区，而该区的起始位置位于新区间结束地址之前的时候。在这种情况下，`do_mmap_pgoff()` 调用 `do_munmap()` 删除新的区间，然后重复整个步骤（参见下一节“释放线性地址区间”）。

5. 检查插入新的线性区是否引起进程地址空间的大小 $(mm->total_vm << PAGE_SHIFT) + len$ 超过存放在进程描述符 `signal->rlim[RLIMIT_AS].rlim_cur` 字段中的阈值。如果是，就返回出错码 `-ENOMEM`。注意，这个检查只在这里进行，而不在第 1 步与其他检查一起进行，因为一些线性区可能在第 4 步就被删除。
6. 如果在 `flags` 参数中没有设置 `MAP_NORESERVE` 标志，新的线性区包含私有可写页，并且没有足够的空闲页框，则返回出错码 `-ENOMEM`；这最后一个检查是由 `security_vm_enough_memory()` 函数实现的。
7. 如果新区间是私有的（没有设置 `VM_SHARED`），且映射的不是磁盘上的一个文件，那么，调用 `vma_merge()` 检查前一个线性区是否可以以这样的方式进行扩展来包

注 5：实际上，内存描述符的 `def_flags` 字段只能由 `mlockall()` 系统调用修改，这个系统调用可以设置 `VM_LOCKED` 标志，由此而锁住 RAM 中调用进程的未来所有页。

含新的区间。当然，前一个线性区必须与在 `vm_flags` 局部变量中存放标志的那些线性区具有完全相同的标志。如果前一个线性区可以扩展，那么，`vma_merge()` 也试图把它与随后的线性区进行合并（这发生在新区间填充两个线性区之间的空洞，且三个线性区全部具有相同的标志的时候）。万一在扩展前一个线性区时获得成功，则跳到第 12 步。

8. 调用 slab 分配函数 `kmem_cache_alloc()` 为新的线性区分配一个 `vm_area_struct` 数据结构。
9. 初始化新的线性区对象（由 `vma` 指向）：

```
vma->vm_mm = mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = vm_flags;
vma->vm_page_prot = protection_map[vm_flags & 0x0f];
vma->vm_ops = NULL;
vma->vm_pgoff = pgoff;
vma->vm_file = NULL;
vma->vm_private_data = NULL;
vma->vm_next = NULL;
INIT_LIST_HEAD(&vma->shared);
```

10. 如果 `MAP_SHARED` 标志被设置（以及新的线性区不映射磁盘上的文件），则该线性区是一个共享匿名区；调用 `shmem_zero_setup()` 对它进行初始化。共享匿名区主要用于进程间通信（参见第十九章“IPC 共享内存”一节）。
11. 调用 `vma_link()` 把新线性区插入到线性区链表和红-黑树中（参见前面“线性区的处理”一节）。
12. 增加存放在内存描述符 `total_vm` 字段中的进程地址空间的大小。
13. 如果设置了 `VM_LOCKED` 标志，就调用 `make_pages_present()` 连续分配线性区的所有页，并把它们锁在 RAM 中：

```
if (vm_flags & VM_LOCKED) {
    mm->locked_vm += len >> PAGE_SHIFT;
    make_pages_present(addr, addr + len);
}
```

`make-pages-present()` 函数按如下方式调用 `get_user_pages()`：

```
write = (vma->vm_flags & VM_WRITE) != 0;
get_user_pages(current, current->mm, addr, len, write, 0, NULL,
NULL);
```

`get_user_pages()` 函数在 `addr` 和 `addr+len` 之间页的所有起始线性地址上循环，对于其中的每个页，该函数调用 `follow_page()` 检查在当前页表中是否有到物理页的映射。如果没有这样的物理页存在，则 `get_user_pages()` 调用 `handle_mm_fault()`，

我们将在“处理地址空间内的错误地址”一节看到，后一个函数分配一个页框并根据内存描述符的 `vm_flags` 字段设置它的页表项。

14. 最后，函数通过返回新线性区的线性地址而终止。

释放线性地址区间

内核使用 `do_munmap()` 函数从当前进程的地址空间中删除一个线性地址区间。参数为：进程内存描述符的地址 `mm`，地址区间的起始地址 `start` 和它的长度 `len`。要删除的区间并不总是对应一个线性区，它或许是一个线性区的一部分，或许跨越两个或多个线性区。

`do_munmap()` 函数

该函数经过两个主要的阶段。第一阶段（第 1~6 步），扫描进程所拥有的线性区链表，并把包含在进程地址空间的线性地址区间中的所有线性区从链表中解除链接。第二阶段（第 7~12 步），更新进程的页表，并把第一阶段找到并标识出的线性区删除。函数利用稍后要说明的 `split_vma()` 和 `unmap_region()` 函数。`do_munmap()` 执行下面的步骤：

1. 对参数值进行一些初步检查：如果线性地址区间所含的地址大于 `TASK_SIZE`，如果 `start` 不是 4096 的倍数，或者如果线性地址区间的长度为 0，则函数返回一个错误代码 `-EINVAL`。
2. 确定要删除的线性地址区间之后第一个线性区 `mpnt` 的位置 (`mpnt->end > start`)，如果有这样的线性区：

```
mpnt = find_vma_prev(mm, start, &prev);
```

3. 如果没有这样的线性区，也没有与线性地址区间重叠的线性区，就什么都不做，因为在该区间上没有线性区：

```
end = start + len;
if (!mpnt || mpnt->vm_start >= end)
    return 0;
```

4. 如果线性区的起始地址在线性区 `mpnt` 内，就调用 `split_vma()`（在下面说明）把线性区 `mpnt` 划分成两个较小的区：一个区在线性地址区间外部，而另一个在区间内部。

```
if (start > mpnt->vm_start) {
    if (split_vma(mm, mpnt, start, 0))
        return -ENOMEM;
    prev = mpnt;}

```

更新局部变量 `prev`，以前它存储的是指向线性区 `mpnt` 前面一个线性区的指针，现在

要让它指向 `mpnt`，即指向线性地址区间外部的一个新线性区。这样，`prev` 仍然指向要删除的第一个线性区前面的那个线性区。

5. 如果线性地址区间的结束地址在一个线性区内部，就再次调用 `split_vma()` 把最后重叠的那个线性区划分成两个较小的区：一个区在线性地址区间内部，而另一个在区间外部（注6）：

```
last = find_vma(mm, end);
if (last && end > last->vm_start){
    if (split_vma(mm, last, start, end, 1))
        return -ENOMEM;
}
```

6. 更新 `mpnt` 的值，使它指向线性地址区间的第一个线性区。如果 `prev` 为 `NULL`，即没有上述线性区，就从 `mm->mmap` 获得第一个线性区的地址：

```
mpnt = prev ? prev->vm_next : mm->mmap;
```

7. 调用 `detach_vmas_to_be_unmapped()` 从进程的线性地址空间中删除位于线性地址区间中的线性区。该函数本质上执行下面的代码：

```
vma = mpnt;
insertion_point = (prev ? &prev->vm_next : &mm->mmap);
do {
    rb_erase(&vma->vm_rb, &mm->mm_rb);
    mm->map_count--;
    tail_vma = vma;
    vma = vma->next;
} while (vma && vma->start < end);
*insertion_point = vma;
tail_vma->vm_next = NULL;
mm->map_cache = NULL;
```

要删除的线性区的描述符放在一个排好序的链表中，局部变量 `mpnt` 指向该链表的头（实际上，这个链表只是进程初始线性区链表的一部分）。

8. 获得 `mm->page_table_lock` 自旋锁。
9. 调用 `unmap_region()` 清除与线性地址区间对应的页表项并释放相应的页框（稍后讨论）：

```
unmap_region(mm, mpnt, prev, start, end);
```

10. 释放 `mm->page_table_lock` 自旋锁。

注6： 如果线性地址区间正好包含在某个线性区内部，就必须用个较小的新线性区取代该线性区。当发生这种情况时，在第4步和第5步把该线性区分成三个较小的线性区：删除中间的那个线性区，而保留第一个和最后一个线性区。

11. 释放在第7步建立链表时收集的线性区描述符：

```
do {
    struct vm_area_struct * next = mpnt->vm_next;
    unmap_vma(mm, mpnt);
    mpnt = next;
} while (mpnt != NULL);
```

对在链表中的所有线性区调用 `unmap_vma()` 函数，它本质上执行下述步骤：

- a. 更新 `mm->total_vm` 和 `mm->locked_vm` 字段。
- b. 执行内存描述符的 `mm->unmap_area` 方法。根据进程线性区的不同类型（参见前面“线性区的处理”一节）可以选择 `arch_unmap_area()` 或 `arch_unmap_area_topdown()` 中的一个来实现 `mm->unmap_area` 方法。如果必要，在两种情况下都要更新 `mm->free_area_cache` 字段。
- c. 调用线性区的 `close` 方法（如果定义了的话）。
- d. 如果线性区是匿名的，则函数把它从 `mm->anon_vma` 所指向的匿名线性区链表中删除。
- e. 调用 `kmem_cache_free()` 释放线性区描述符。

12. 返回 0（成功）。

split_vma()函数

`split_vma()` 函数的功能是把与线性地址区间交叉的线性区划分成两个较小的区，一个在线性地址区间外部，另一个在区间的内部。该函数接收 4 个参数：内存描述符指针 `mm`，线性区描述符指针 `vma`（标识要被划分的线性区），表示区间与线性区之间交叉点的地址 `addr`，以及表示区间与线性区之间交叉点在区间起始处还是结束处的标志 `new_below`。函数执行下述基本步骤：

1. 调用 `kmem_cache_alloc()` 获得线性区描述符 `vm_area_struct`，并把它地址存在新的局部变量中，如果没有可用的空闲空间，就返回 `-ENOMEM`。
2. 用 `vma` 描述符的字段值初始化新描述符的字段。
3. 如果标志 `new_below` 为 0，说明线性地址区间的起始地址在 `vma` 线性区的内部，因此必须把新线性区放在 `vma` 线性区之后，所以函数把 `new->vm_start` 和 `vma->vm_end` 字段都赋值为 `addr`。
4. 相反，如果 `new_below` 标志等于 1，说明线性地址区间的结束地址在 `vma` 线性区的内部，因此必须把新线性区放在 `vma` 线性区的前面，所以函数把字段 `new->vm_end` 和 `vma->vm_start` 都赋值为 `addr`。
5. 如果定义了新线性区的 `open` 方法，函数就执行它。

6. 把新线性区描述符链接到线性区链表 `mm->mmap` 和红-黑树 `mm->mm_rb`。此外，函数还要根据线性区 `vma` 的最新大小对红-黑树进行调整。
7. 返回 0（成功）。

`unmap_region()` 函数

`unmap_region()` 函数遍历线性区链表并释放它们的页框。该函数作用于 5 个参数：内存描述符指针 `mm`，指向第一个被删除线性区描述符的指针 `vma`，指向进程链表中 `vma` 前面的线性区的指针 `prev`（参见 `do_munmap()` 执行步骤中心的第 2 步和第 4 步），以及两个地址 `start` 和 `end`，它们界定被删除线性地址区间的范围。函数本质上执行下述步骤：

1. 调用 `lru_add_drain()`（参见第十七章）。
2. 调用 `tlb_gather_mmu()` 函数初始化每 CPU 变量 `mmu_gathers`。`mmu_gathers` 的值依赖于体系结构：通常该变量应该存放成功更新进程页表项所需要的所有信息。在 80x86 体系结构中，`tlb_gather_mmu()` 函数只是简单地把内存描述符指针 `mm` 的值赋给本地 CPU 的 `mmu_gathers` 变量。
3. 把 `mmu_gathers` 变量的地址保存在局部变量 `tlb` 中。
4. 调用 `unmap_vmas()` 扫描线性地址空间的所有页表项：如果只有一个有效 CPU，函数就调用 `free_swap_and_cache()` 反复释放相应的页（参见第十七章）；否则，函数就把相应页描述符的指针保存在局部变量 `mmu_gathers` 中。
5. 调用 `free_pgtables(tlb, prev, start, end)` 回收在上一步已经清空的进程页表。
6. 调用 `tlb_finish_mmu(tlb, start, end)` 结束 `unmap_region()` 函数的工作，`tlb_finish_mmu(tlb, start, end)` 依次执行下面的操作：
 - a. 调用 `flush_tlb_mm()` 刷新 TLB（参见第二章“处理硬件高速缓存和 TLB”一节）。
 - b. 在多处理器系统中，调用 `free_pages_and_swap_cache()` 释放页框，这些页框的指针已经集中存放在 `mmu_gather` 数据结构中了。该函数的说明见第十七章。

缺页异常处理程序

如前所述，Linux 的缺页（Page Fault）异常处理程序必须区分以下两种情况：由编程错误所引起的异常，及由引用属于进程地址空间但还尚未分配物理页框的页所引起的异常。

线性区描述符可以让缺页异常处理程序非常有效地完成它的工作。`do_page_fault()` 函数是 80x86 上的缺页中断服务程序，它把引起缺页的线性地址和当前进程的线性区相比较，从而能够根据和图 9-4 所示的方案选择适当的方法处理这个异常。

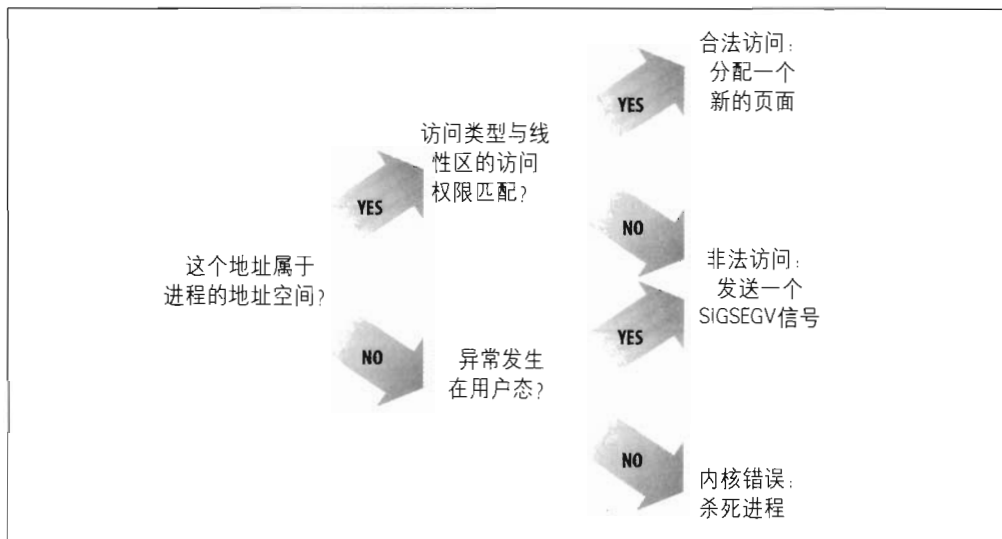


图 9-4: 缺页异常处理程序的总体方案

在实际中, 情况更复杂一些, 因为缺页处理程序必须处理多种分得更细的特殊情况, 它们不宜在总体方案中列出来, 还必须区分许多合理的访问。处理程序的详细流程图如图 9-5 所示。

标识符 `vmalloc_fault`、`good_area`、`bad_area` 和 `no_context` 是出现在 `do_page_fault()` 中的标记, 它们有助于你理清流程图中的块与代码中特定行之间的关系。

`do_page_fault()` 函数接收以下输入参数:

- `pt_regs` 结构的地址 `regs`, 该结构包含当异常发生时的微处理器寄存器的值。
- 3 位的 `error_code`, 当异常发生时由控制单元压入栈中 (参见第四章中的“中断和异常的硬件处理”一节)。这些位有以下含义:
 - 如果第 0 位被清 0, 则异常由访问一个不存在的页所引起 (页表项中的 `Present` 标志被清 0); 否则, 如果第 0 位被设置, 则异常由无效的访问权限所引起。
 - 如果第 1 位被清 0, 则异常由读访问或者执行访问所引起; 如果该位被设置, 则异常由写访问所引起。
 - 如果第 2 位被清 0, 则异常发生在处理器处于内核态时; 否则, 异常发生在处理器处于用户态时。

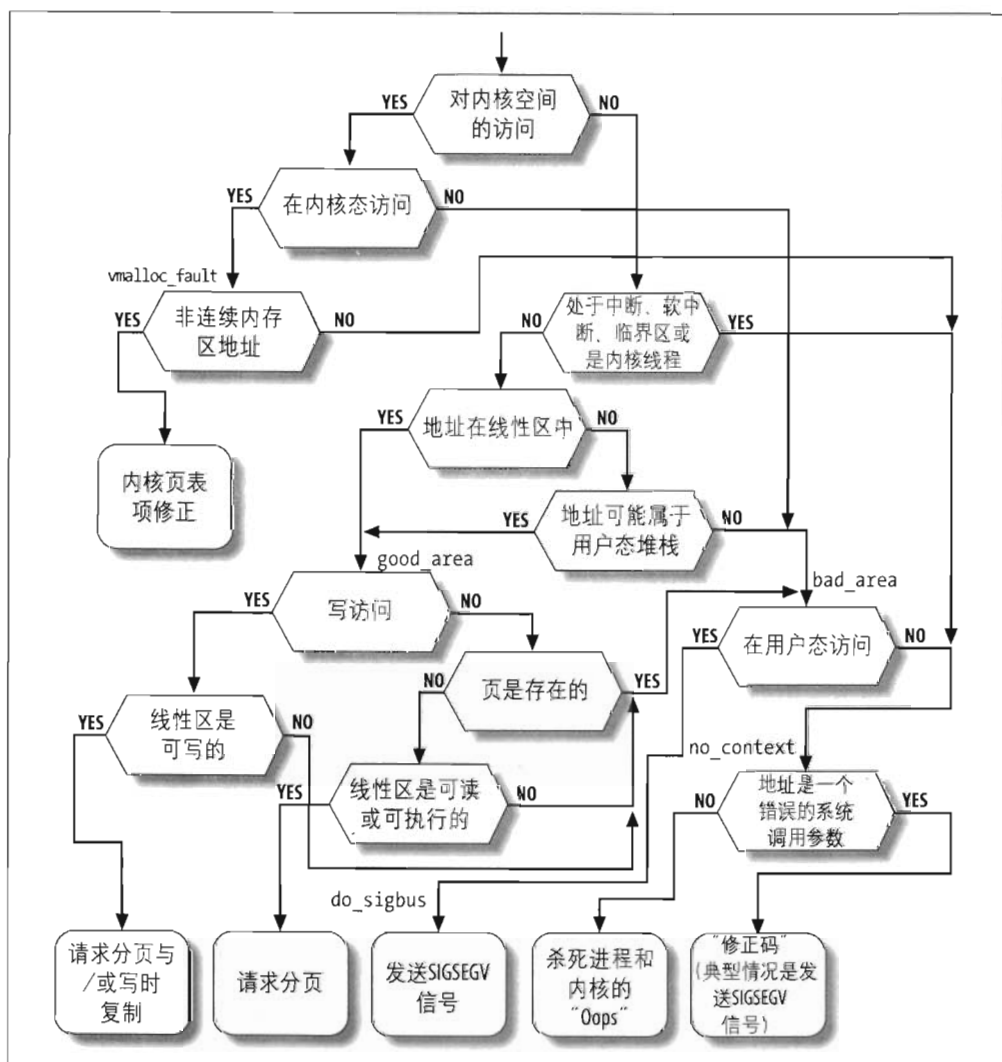


图 9-5: 缺页处理程序流程图

`do_page_fault()` 的第一步操作是读取引起缺页的线性地址。当异常发生时，CPU 控制单元把这个值存放在 `cr2` 控制寄存器中：

```

asm! "movl %%cr2, %0" : "=r" (address);
if (regs->eflags & 0x00020200)
    local_irq_enable();
tsk = current;

```

这个线性地址保存在 `address` 局部变量中。如果缺页发生之前或 CPU 运行在虚拟 8086

模式时就打开了本地中断，那么该函数还要确保本地中断打开，并把指向 `current` 进程描述符的指针保存在 `tsk` 局部变量中。

正如图 9-5 中的顶部所示，`do_page_fault()` 首先检查引起缺页的线性地址是否属于第 4 个 GB：

```
info.si_code = SEGV_MAPERR;
if (address >= TASK_SIZE ) {
    if (!(error_code & 0x101))
        goto vmalloc_fault;
    goto bad_area_nosemaphore;
}
```

如果发生了由于内核试图访问不存在的页框引起的异常，就跳转去执行 `vmalloc_fault` 标记处的代码，该部分代码处理可能由于在内核态访问非连续内存区而引起的缺页，我们在稍后“处理非连续内存区访问”一节中对此进行说明。否则，就跳转去执行 `bad_area_nosemaphore` 标记处的代码，在稍后“处理地址空间以外的错误地址”一节中将对此进行说明。

接下来，缺页处理程序检查异常发生时是否内核正在执行一些关键例程或正在运行内核线程（回想一下，对内核线程而言，进程描述符的 `mm` 字段总为 `NULL`）：

```
if (in_atomic( ) || !tsk->mm)
    goto bad_area_nosemaphore;
```

如果缺页发生在下面任何一种情况下，则 `in_atomic()` 宏产生等于 1 的值：

- 内核正在执行中断处理程序或可延迟函数。
- 内核正在禁用内核抢占的情况下执行临界区代码（参见第五章“内核抢占”一节）。

如果缺页的确发生在中断处理程序、可延迟函数、临界区或内核线程中，`do_page_fault()` 就不会试图把这个线性地址与 `current` 的线性区做比较。内核线程从来不使用小于 `TASK_SIZE` 的地址。同样，中断处理程序、可延迟函数和临界区代码也不应该使用小于 `TASK_SIZE` 的地址，因为这可能导致当前进程的阻塞。（参见本章稍后“处理地址空间以外的错误地址”一节中，对 `info` 局部变量的信息说明和对 `bad_area_nosemaphor` 标记处代码的说明。）

让我们假定缺页没有发生在中断处理程序、可延迟函数、临界区或者内核线程中。于是函数必须检查进程所拥有的线性区以决定引起缺页的线性地址是否包含在进程的地址空间中，为此必须获得进程的 `mmap_sem` 读/写信号量：

```
if (!down_read_trylock(&tsk->mm>mmap_sem)) {
    if ((error_code & 4) == 0 &&
```

```

    !search_exception_table(regs->eip)
    goto bad_area_nosemaphore;
    down_read(&tsk->mm->mmap_sem);
}

```

如果内核 bug 和硬件故障有可能被排除，那么当缺页发生时，当前进程就还没有为写而获得信号量 `mmap_sem`。尽管如此，`do_page_fault()` 还是想确定的确没有获得这个信号量，因为如果不是这样就会发生死锁。所以，函数使用 `down_read_trylock()` 而不是 `down_read()`（参见第五章“读/写信号量”一节）。如果这个信号量被关闭而且缺页发生在内核态，`do_page_fault()` 就要确定异常发生的时候，是否正在使用作为系统调用参数被传递给内核的线性地址（参见下一节“处理地址空间以外的错误地址”一节）。此时，因为每个系统调用服务例程都小心地避免在访问用户态地址空间以前为写而获得 `mmap_sem` 信号量，所以 `do_page_fault()` 确信 `mmap_sem` 信号量由另外一个进程占有了，从而 `do_page_fault()` 一直等待直到该信号量被释放。否则，如果缺页是由于内核 bug 或严重的硬件故障引起的，函数就跳转到 `bad_area_nosemaphore` 标记处。

我们假设已经为读而获得了 `mmap_sem` 信号量。现在，`do_page_fault()` 开始搜索错误线性地址所在的线性区：

```

vma = find_vma(tsk->mm, address);
if (!vma)
    goto bad_area;
if (vma->vm_start <= address)
    goto good_area;

```

如果 `vma` 为 `NULL`，说明 `address` 之后没有线性区，因此这个错误的地址肯定是无效的。另一方面，如果在 `address` 之后结束处的第一个线性区包含 `address`，则函数跳到标记为 `good_area` 的代码处。

如果两个“if”条件都不满足，则函数已确定 `address` 没有包含在任何线性区中。可是，它还必须执行进一步的检查，由于这个错误地址可能是由 `push` 或 `pusha` 指令在进程的用户态堆栈上的操作所引起的。

我们稍微离题一点，解释一下栈是如何映射到线性区上的。每个向低地址扩展的栈所在的区，它的 `VM_GROWSDOWN` 标志被设置，这样，当 `vm_start` 字段的值可能被减小的时候，而 `vm_end` 字段的值保持不变。这种线性区的边界包括、但不严格限定用户态堆栈当前的大小。这种细微的差别主要基于以下原因：

- 线性区的大小是 4KB 的倍数（必须包含完整的页），而栈的大小却是任意的。
- 分配给一个线性区的页框在这个线性区被删除前永远不被释放。尤其是，一个栈所在线性区的 `vm_start` 字段的值只能减小，永远也不能增加。甚至进程执行一系列的 `pop` 指令时，这个线性区的大小仍然保持不变。

现在这一点就很清楚了，当进程填满分配给它的堆栈的最后一个页框后，进程如何引起一个“缺页”异常——`push`引用了这个线性区以外的一个地址（即引用一个不存在的页框）。注意，这种异常不是由程序错误引起的，因此它必须由缺页处理程序单独处理。

我们现在回到对 `do_page_fault()` 的描述，它检查上面所描述的情况：

```
if (!(vma->vm_flags & VM_GROWSDOWN))
    goto bad_area;
if (error_code & 4 /* User Mode */
    && address + 32 < regs->esp)
    goto bad_area;
if (expand_stack(vma, address))
    goto bad_area;
goto good_area;
```

如果线性区的 `VM_GROWSDOWN` 标志被设置，并且异常发生在用户态，函数就要检查 `address` 是否小于 `regs->esp` 栈指针（它应该只小于一点点）。因为几个与栈相关的汇编语言指令（如 `pusha`）只有在访问内存之后才执行减 `esp` 寄存器的操作，所以允许进程有 32 字节的后备区间。如果这个地址足够高（在允许的范围内），则代码调用 `expand_stack()` 函数检查是否允许进程既扩展它的栈也扩展它的地址空间。如果一切都可以，就把 `vma` 的 `vm_start` 字段设为 `address`，并返回 0；否则，返回 `-ENOMEM`。

注意：只要线性区的 `VM_GROWSDOWN` 标志被设置，但异常不是发生在用户态，上述代码就跳过容错检查。这些条件意味着内核正在访问用户态的栈，也意味着这段代码总是应当运行 `expand_stack()`。

处理地址空间以外的错误地址

如果 `address` 不属于进程的地址空间，那么 `do_page_fault()` 函数继续执行 `bad_area` 标记处的语句。如果错误发生在用户态，则发送一个 `SIGSEGV` 信号给 `current` 进程（参见第十一章的“产生信号”一节）并结束函数：

```
bad_area:
up_read(&tsk->mm->mmap_sem);
bad_area_nosemaphore:
if (error_code & 4) { /* User Mode */
    tsk->thread.cr2 = address;
    tsk->thread.error_code = error_code | (address >= TASK_SIZE);
    tsk->thread.trap_no = 14;
    info.si_signo = SIGSEGV;
    info.si_errno = 0;
    info.si_addr = (void *) address;
    force_sig_info(SIGSEGV, &info, tsk);
    return;
}
```

`force_sig_info()` 确信进程不忽略或阻塞 `SIGSEGV` 信号，并通过 `info` 局部变量传递附加信息的同时把该信号发送给用户态进程（参见第十一章“产生信号”一节）。`info.si_code` 字段已被置为 `SEGV_MAPERR`（如果异常是由于一个不存在的页框引起），或置为 `SEGV_ACCERR`（如果异常是由于对现有页框的无效访问引起）。

如果异常发生在内核态（`error_code` 的第 2 位被清 0），仍然有两种可选的情况：

- 异常的引起是由于把某个线性地址作为系统调用的参数传递给内核。
- 异常是因一个真正的内核缺陷所引起的。

函数这样区分这两种可选的情况：

```
no_context:
if ((fixup = search_exception_table(regs->eip)) != 0) {
    regs->eip = fixup;
    return;
}
```

在第一种情况中，代码跳到一段“修正代码”处，这段代码的典型操作就是向当前进程发送 `SIGSEGV` 信号，或用一个适当的出错码终止系统调用处理程序（参见第十章中的“动态地址检查：修正代码”一节）。

在第二种情况中，函数把 CPU 寄存器和内核态堆栈的全部转储打印到控制台，并输出到一个系统消息缓冲区，然后调用函数 `do_exit()` 杀死当前进程（参见第二十章）。这就是所谓按所显示的消息命名的“内核漏洞 (*Kernel oops*)”错误。这些输出值可由内核编程高手用于推测引发此错误的条件，进而发现并纠正错误。

处理地址空间内的错误地址

如果 `addr` 地址属于进程的地址空间，则 `do_page_fault()` 转到 `good_area` 标记处的语句执行：

```
good_area:
info.si_code = SEGV_ACCERR;
write = 0;
if (error_code & 2) { /* write access */
    if (!(vma->vm_flags & VM_WRITE))
        goto bad_area;
    write++;
} else /* read access */
    if ((error_code & 1) || !(vma->vm_flags & (VM_READ | VM_EXEC)))
        goto bad_area;
```

如果异常由写访问引起，函数检查这个线性区是否可写。如果不可写，跳到 `bad_area` 代码处；如果可写，把 `write` 局部变量置为 1。

如果异常由读或执行访问引起，函数检查这一页是否已经存在于RAM中。在存在的情况下，异常发生是由于进程试图访问用户态下的一个有特权的页框（页框的User/Supervisor标志被清除），因此函数跳到bad_area代码处（注7）。在不存在的情况下，函数还将检查这个线性区是否可读或可执行。

如果这个线性区的访问权限与引起异常的访问类型相匹配，则调用handle_mm_fault()函数分配一个新的页框：

```
survive:
ret = handle_mm_fault(tsk->mm, vma, address, write);
if (ret == VM_FAULT_MINOR || ret == VM_FAULT_MAJOR) {
    if (ret == VM_FAULT_MINOR) tsk->min_flt++; else tsk->maj_flt++;
    up_read(&tsk->mm->mmap_sem);
    return;
}
```

如果handle_mm_fault()函数成功地给进程分配一个页框，则返回VM_FAULT_MINOR或VM_FAULT_MAJOR。值VM_FAULT_MINOR表示在没有阻塞当前进程的情况下处理了缺页，这种缺页叫做次缺页 (*minor fault*)。值VM_FAULT_MAJOR表示缺页迫使当前进程睡眠（很可能是由于当用磁盘上的数据填充所分配的页框时花费时间），阻塞当前进程的缺页就叫做主缺页 (*major fault*)。函数也返回VM_FAULT_OOM（没有足够的内存）或VM_FAULT_STGBOS（其他任何错误）。

如果handle_mm_fault()返回值VM_FAULT_SIGBUS，则向进程发送SIGBUS信号：

```
if (ret == VM_FAULT_SIGBUS) {
do_sigbus:
    up_read(&tsk->mm->mmap_sem);
    if (!(error_code & 4)) /*内核态*/
        goto no_context;
    tsk->thread.cr2 = address;
    tsk->thread.error_code = error_code;
    tsk->thread.trap_no = 14;
    info.si_signo = SIGBUS;
    info.si_errno = 0;
    info.si_code = BUS_ADRERR;
    info.si_addr = (void *) address;
    force_sig_info(SIGBUS, &info, tsk);
}
```

如果handle_mm_fault()不分配新的页框，就返回值VM_FAULT_OOM，此时内核通常杀死当前进程。不过，如果当前进程是init进程，则只是把它放在运行队列的末尾并调用调度程序；一旦init恢复执行，则handle_mm_fault()又执行：

注7： 然而，这种情况从不会发生，因为内核不会把具有特权的页框赋给进程。

```

if (ret == VM_FAULT_OOM) {
    out_of_memory:
    up_read(&tsk->mm->mmap_sem);
    if (tsk->pid != 1) {
        if (error_code & 4) /* User Mode */
            do_exit(SIGKILL);
        goto no_context;
    }
    yield();
    down_read(&tsk->mm->mmap_sem);
    goto survive;
}

```

handle_mm_fault()函数作用于4个参数:

mm

指向异常发生时正在CPU上运行的进程的内存描述符。

vma

指向引起异常的线性地址所在线性区的描述符。

address

引起异常的线性地址。

write_access

如果tsk试图向address写,则置为1;如果tsk试图在address读或执行,则置为0。

这个函数首先检查用来映射address的页中间目录和页表是否存在。即使address属于进程的地址空间,相应的页表也可能还没有被分配,因此,在做别的事情之前首先执行分配页目录和页表的任务:

```

pgd = pgd_offset(mm, address);
spin_lock(&mm->page_table_lock);
pud = pud_alloc(mm, pgd, address);
if (pud) {
    pmd = pmd_alloc(mm, pud, address);
    if (pmd) {
        pte = pte_alloc_map(mm, pmd, address);
        if (pte)
            return handle_pte_fault(mm, vma, address,
                                    write_access, pte, pmd);
    }
}
spin_unlock(&mm->page_table_lock);
return VM_FAULT_OOM;

```

pgd局部变量包含引用address的页全局目录项。如果需要的话,调用pud_alloc()和

`pmd_alloc()`函数分别分配一个新的页上级目录和页中间目录(注8)。然后,如果需要的话,调用`pte_alloc_map()`函数分配一个新的页表。如果这两步都成功,`pte`局部变量所指向的页表项就是引用`address`的表项。然后调用`handle_pte_fault()`函数检查`address`地址所对应的页表项,并决定如何为进程分配一个新页框:

- 如果被访问的页不存在,也就是说,这个页还没有被存放在任何一个页框中,那么,内核分配一个新的页框并适当地初始化。这种技术称为请求调页(demand paging)。
- 如果被访问的页存在但是标记为只读,也就是说,它已经被存放在一个页框中,那么,内核分配一个新的页框,并把旧页框的数据拷贝到新页框来初始化它的内容。这种技术称为写时复制(Copy On Write, COW)。

请求调页

术语“请求调页”指的是一种动态内存分配技术,它把页框的分配推迟到不能再推迟为止,也就是说,一直推迟到进程要访问的页不在RAM中时为止,由此引起一个缺页异常。

请求调页技术背后的动机是:进程开始运行的时候并不访问其地址空间中的全部地址,事实上,有一部分地址也许永远不被进程使用。此外,程序的局部性原理(参见第二章中的“硬件高速缓存”一节)保证了在程序执行的每个阶段,真正引用的进程页只有一小部分,因此临时用不着的页所在的页框可以由其他进程来使用。因此,对于全局分配(一开始就给进程分配所需要的全部页框,直到程序结束才释放这些页框)来说,请求调页是首选的,因为它增加了系统中的空闲页框的平均数,从而更好地利用空闲内存。从另一个观点来看,在RAM总数保持不变的情况下,请求调页从总体上能使系统有更大的吞吐量。

为这一切优点付出的代价是系统额外的开销:由请求调页所引发的每个“缺页”异常必须由内核处理,这将浪费CPU的时钟周期。幸运的是,局部性原理保证了一旦进程开始在一组页上运行,在接下来相当长的一段时间内它会一直停留在这些页上而不去访问其他的页,这样我们就可以认为“缺页”异常是一种稀有事件。

被访问的页可能不在主存中,其原因或者是进程从没访问过该页,或者是内核已经回收了相应的页框(参见第十七章)。

注8: 在80x86微处理器中,这种分配永远不会发生,因为页上级目录总是包含在页全局目录中,并且页中间目录或者包含在页上级目录中(PAE未激活),或者与页上级目录一块被分配(PAE被激活)。

在这两种情况下，缺页处理程序必须为进程分配新的页框。不过，如何初始化这个页框取决于是哪一种页以及页以前是否被进程访问过。特殊情况下：

1. 或者这个页从未被进程访问到且没有映射磁盘文件，或者页映射了磁盘文件。内核能够识别这些情况，这是因为页表相应的表项被填充为0，也就是说，`pte_none`宏返回1。
2. 页属于非线性磁盘文件的映射（参见第十六章“非线性内存映射”一节）。内核能够识别这种情况，因为 `Present` 标志被清0而且 `Dirty` 标志被置1，也就是说，`pte_file`宏返回1。
3. 进程已经访问过这个页，但是其内容被临时保存在磁盘上。内核能够识别这种情况，这是因为相应表项没被填充为0，但是 `Present` 和 `Dirty` 标志被清0。

因此，`handle_pte_fault()`函数通过检查`address`对应的页表项能够区分这三种情况：

```
entry = *pte;
if (!pte_present(entry)) {
    if (pte_none(entry))
        return do_no_page(mm, vma, address, write_access, pte, pmd);
    if (pte_file(entry))
        return do_file_page(mm, vma, address, write_access, pte, pmd);
    return do_swap_page(mm, vma, address, pte, pmd, entry, write_access);
}
```

我们将在第十六章和第十七章分别讨论第2和第3种情况。

在第1种情况下，当页从未被访问或页线性地映射磁盘文件时则调用`do_no_page()`函数。有两种方法装入所缺的页，这取决于这个页是否被映射到一个磁盘文件。该函数通过检查`vma`线性区描述符的`nopage`字段来确定这一点，如果页被映射到一个文件，`nopage`字段就指向一个函数，该函数把所缺的页从磁盘装入到RAM。因此，可能的情况是：

- `vma->vm_ops->nopage`字段不为NULL。在这种情况下，线性区映射了一个磁盘文件，`nopage`字段指向装入页的函数。这种情况将在第十六章的“内存映射的请求调页”一节和第十九章的“IPC 共享内存”一节进行阐述。
- 或者`vma->vm_ops`字段为NULL，或者`vma->vm_ops->nopage`字段为NULL。在这种情况下，线性区没有映射磁盘文件，也就是说，它是一个匿名映射 (*anonymous mapping*)。因此，`do_no_page()`调用`do_anonymous_page()`函数获得一个新的页框：

```
if (!vma->vm_ops || !vma->vm_ops->nopage)
    return do_anonymous_page(mm, vma, page_table, pmd,
                             write_access, address);
```

do_anonymous_page()函数（注9）分别处理写请求和读请求：

```
if (write_access) {
    pte_unmap(page_table);
    spin_unlock(&mm->page_table_lock);
    page = alloc_page(GFP_HIGHUSER | __GFP_ZERO);
    spin_lock(&mm->page_table_lock);
    page_table = pte_offset_map(pmd, addr);
    mm->rss++;
    entry = maybe_mkwrite(pte_mkdirty(mk_pte(page,
                                                vma->vm_page_prot)), vma);

    lru_cache_add_active(page);
    SetPageReferenced(page);
    set_pte(page_table, entry);
    pte_unmap(page_table);
    spin_unlock(&mm->page_table_lock);
    return VM_FAULT_MINOR;
}
```

pte_unmap宏的第一次执行释放一种临时内核映射，它映射了在调用handle_pte_fault()函数之前由pte_offset_map所建立页表项的高端内存物理地址（参见第二章“页表处理”一节中的表2-7）。pte_offset_map和pte_unmap这对宏获取和释放同一个临时内核映射。临时内核映射必须在调用alloc_page()之前释放，因为这个函数可能阻塞当前进程。

函数递增内存描述符的rss字段以记录分配给进程的页框总数。相应的页表项设置为页框的物理地址，页表框被标记为既脏又可写的（注10）。lru_cache_add_active()函数把新页框插入与交换相关的数据结构中，我们在第十七章对它进行说明。

相反，当处理读访问时，页的内容是无关紧要的，因为进程第一次对它访问。给进程一个填充为0的页要比给它一个由其他进程填充了信息的旧页更为安全。Linux在请求调页方面做得更深入一些。没有必要立即给进程分配一个填充为0的新页框，由于我们也可以给它一个现有的称为零页（zero page）的页，这样可以进一步推迟页框的分配。零页在内核初始化期间被静态分配，并存放在empty_zero_page变量中（长为4096字节的数组，并用0填充）。

因此用零页的物理地址设置页表项：

注9： 为了简化对这个函数的说明，我们略过处理反映射的语句，有关反映射的内容见第十七章“反向映射”一节。

注10： 如果调试程序试图往被跟踪进程只读线性区中的页中写数据，内核就不设置Read/Write标志。函数maybe_mkwrite()处理这种特殊情况。

```
entry = pte_wrprotect(mk_pte(virt_to_page(empty_zero_page),
                             vma->vm_page_prot));
set_pte(page_table, entry);
spin_unlock(&mm->page_table_lock);
return VM_FAULT_MINOR;
```

由于这个页被标记为不可写的，因此如果进程试图写这个页，则写时复制机制被激活。当且仅当在这个时候，进程才获得一个属于自己的页并对它进行写操作。这种机制将在下一节中进行描述。

写时复制

第一代 Unix 系统实现了一种傻瓜式的进程创建：当发出 `fork()` 系统调用时，内核原样复制父进程的整个地址空间并把复制的那一份分配给子进程。这种行为是非常耗时的，因为它需要：

- 为子进程的页表分配页框
- 为子进程的页分配页框
- 初始化子进程的页表
- 把父进程的页复制到子进程相应的页中

这种创建地址空间的方法涉及许多内存访问，消耗许多 CPU 周期，并且完全破坏了高速缓存中的内容。在大多数情况下，这样做常常是毫无意义的，因为许多子进程通过装入一个新的程序开始它们的执行，这样就完全丢弃了所继承的地址空间（参见第二十章）。

现在的 Unix 内核（包括 Linux）采用一种更为有效的方法，称之为写时复制（*Copy On Write*, COW）。这种思想相当简单：父进程和子进程共享页框而不是复制页框。然而，只要页框被共享，它们就不能被修改。无论父进程还是子进程何时试图写一个共享的页框，就产生一个异常，这时内核就把这个页复制到一个新的页框中并标记为可写。原来的页框仍然是写保护的：当其他进程试图写入时，内核检查写进程是否是这个页框的唯一属主，如果是，就把这个页框标记为对这个进程是可写的。

页描述符的 `_count` 字段用于跟踪共享相应页框的进程数目。只要进程释放一个页框或者在它上面执行写时复制，它的 `_count` 字段就减小；只有当 `_count` 变为 -1 时，这个页框才被释放（参见第八章“页描述符”一节）

现在我们讲述 Linux 怎样实现写时复制。当 `handle_pte_fault()` 确定缺页异常是由访问内存中现有的一个页而引起时，它执行以下指令：

```
if (pte_present(entry)) {
```

```

    if (write_access) {
        if (!pte_write(entry))
            return do_wp_page(mmm, vma, address, pte, pmd, entry);
        entry = pte_mkdirty(entry);
    }
    entry = pte_mkyoung(entry);
    set_pte(pte, entry);
    flush_tlb_page(vma, address);
    pte_unmap(pte);
    spin_unlock(&mmm->page_table_lock);
    return VM_FAULT_MINOR;
}

```

handle_pte_fault()函数是与体系结构无关的：它考虑任何违背页访问权限的可能。然而，在80x86体系结构上，如果页是存在的，那么，访问权限是写允许的而页框是写保护的（参见前面“处理地址空间内的错误地址”一节）。因此，总是要调用do_wp_page()函数。

do_wp_page()函数（注11）首先获取与缺页异常相关的页框描述符（缺页表项对应的页框）。接下来，函数确定页的复制是否真正必要。如果仅有一个进程拥有这个页，那么，写时复制就不必应用，且该进程应当自由地写该页。具体来说，函数读取页描述符的_count字段：如果它等于0（只有一个所有者），写时复制就不必进行。实际上，检查要稍微复杂些，因为当页插入到交换高速缓存（参见第十七章“交换高速缓存”一节）并且当设置了页描述符的PG_private标志时，_count字段也增加。不过，当写时复制不进行时，就把该页框标记为可写的，以免试图写时引起进一步的缺页异常：

```

set_pte(page_table, maybe_mkwrite(pte_mkyoung(pte_mkdirty(pte)), vma));
flush_tlb_page(vma, address);
pte_unmap(page_table);
spin_unlock(&mmm->page_table_lock);
return VM_FAULT_MINOR;

```

如果两个或多个进程通过写时复制共享页框，那么函数就把旧页框(old_page)的内容复制到新分配的页框(new_page)中。为了避免竞争条件，在开始复制操作前调用get_page()把old_page的使用计数加1：

```

old_page = pte_page(pte);
pte_unmap(page_table);
get_page(old_page);
spin_unlock(&mmm->page_table_lock);
if (old_page == virt_to_page(empty_zero_page))
    new_page = alloc_page(GFP_HIGHUSER | __GFP_ZERO);

```

注11：为了简化对这个函数的说明，我们略过处理反映射的语句，有关反映射的内容见第十七章“反向映射”一节。

```

} else {
    new_page = alloc_page(GFP_HIGHUSER);
    vfrom = kmap_atomic(old_page, KM_USER0);
    vto = kmap_atomic(new_page, KM_USER1);
    copy_page(vto, vfrom);
    kunmap_atomic(vfrom, KM_USER0);
    kunmap_atomic(vto, KM_USER0);
}

```

如果旧页框是零页，就在分配新的页框时（`__GFP_ZERO`标志）把它填充为0。否则，使用 `copy_page()` 宏复制页框的内容。不要求一定要对零页做特殊的处理，但是特殊处理确实能够提高系统的性能，因为它减少地址引用而保护了微处理器的硬件高速缓存。

因为页框的分配可能阻塞进程，因此，函数检查自从函数开始执行以来是否已经修改了页表项（`pte` 和 `*page_table` 具有不同的值）。如果是，新的页框被释放，`old_page` 的使用计数器被减少（取消以前的增加），函数结束。

如果所有的事情看起来进展顺利，那么，新页框的物理地址最终被写进页表项，且使相应的 TLB 寄存器无效：

```

spin_lock(&mm->page_table_lock);
entry = maybe_mkwrite(pte_mkdirty(mk_pte(new_page,
                                     vma->vm_page_prot)), vma);
set_pte(page_table, entry);
flush_tlb_page(vma, address);
lru_cache_add_active(new_page);
pte_unmap(page_table);
spin_unlock(&mm->page_table_lock);

```

`lru_cache_add_active()` 函数把新页框插入到与交换相关的数据结构中；参见第十七章对其的描述。

最后，`do_wp_page()` 把 `old_page` 的使用计数器减少两次。第一次的减少是取消复制页框内容之前进行的安全性增加；第二次的减少是反映当前进程不再拥有该页框这一事实。

处理非连续内存区访问

我们已经在第八章“非连续内存区管理”一节中看到，内核在更新非连续内存区对应的页表项时是非常懒惰的。事实上，`vmalloc()` 和 `vfree()` 函数只把自己限制在更新主内核页表（即页全局目录 `init_mm.pgd` 和它的子页表）。

然而，一旦内核初始化阶段结束，任何进程或内核线程便都不直接使用主内核页表。因此，我们来考虑内核态进程对非连续内存区的第一次访问。当把线性地址转换为物理地址时，CPU 的内存管理单元遇到空的页表项并产生一个缺页。但是，缺页异常处理程序

认识这种特殊情况，因为异常发生在内核态且产生缺页的线性地址大于 `TASK_SIZE`。因此，`do_page_fault()` 检查相应的主内核页表项：

```
vmalloc_fault:
asm("movl %%cr3,%0":"=r" (pgd_paddr));
pgd = pgd_index(address) + (pgd_t *) _va(pgd_paddr);
pgd_k = init_mm.pgd + pgd_index(address);
if (!pgd_present(*pgd_k))
    goto no_context;
pud = pud_offset(pgd, address);
pud_k = pud_offset(pgd_k, address);
if (!pud_present(*pud_k))
    goto no_context;
pmd = pmd_offset(pud, address);
pmd_k = pmd_offset(pud_k, address);
if (!pmd_present(*pmd_k))
    goto no_context;
set_pmd(pmd, *pmd_k);
pte_k = pte_offset_kernel(pmd_k, address);
if (!pte_present(*pte_k))
    goto no_context;
return;
```

把存放在 `cr3` 寄存器中的当前进程页全局目录的物理地址赋给局部变量 `pgd_paddr`（注 12），把与 `pgd_paddr` 相应的线性地址赋给局部变量 `pgd`，并且把主内核页全局目录的线性地址赋给 `pgd_k` 局部变量。

如果产生缺页的线性地址所对应的主内核页全局目录项为空，则函数跳到标号为 `no_context` 的代码处（参见前面“处理地址空间以外的错误地址”一节）。否则，函数检查与错误线性地址相对应的主内核页上级目录项和主内核页中间目录项。如果它们中有一个为空，就再次跳转到 `no_context` 标记处。否则，就把主目录项复制到进程页中间目录的相应项中（注 13）。随后对主页表项重复上述整个操作。

创建和删除进程的地址空间

除了“进程的地址空间”这一节所提到的进程获得一个新的线性区的六种典型的情况之

注 12： 内核不使用 `current->mm->pgd` 导出当前进程的页全局目录地址，因为这种缺页可能在任何时刻都发生，甚至在进程切换期间发生。

注 13： 你也许还记得在第二章“Linux 中的分页”一节中的内容：如果 PAE 被激活，页上级目录项就不可能为空；如果没有激活 PAE，设置页中间目录项的同时也就隐含设置了页上级目录项。

外，首先要指出的是 `fork()` 系统调用要求为子进程创建一个完整的新地址空间。相反，当进程结束时，内核撤消它的地址空间。这一节我们讨论 Linux 如何执行这两种操作。

创建进程的地址空间

我们在第三章的“`clone()`、`fork()`及`vfork()`系统调用”一节中已经提到，当创建一个新的进程时内核调用 `copy_mm()` 函数。这个函数通过建立新进程的所有页表和内存描述符来创建进程的地址空间。

通常，每个进程都有自己的地址空间，但是轻量级进程可以通过调用 `clone()` 函数（设置了 `CLONE_VM` 标志）来创建。这些轻量级进程共享同一地址空间，也就是说，允许它们对同一组页进行寻址。

按照前面讲述的写时复制方法，传统的进程继承父进程的地址空间，只要页是只读的，就依然共享它们。当其中的一个进程试图对某个页进行写时，这个页就被复制一份。一段时间之后，所创建的进程通常获得与父进程不一样的完全属于自己的地址空间。另一方面，轻量级的进程使用父进程的地址空间。Linux 实现轻量级进程很简单，即不复制父进程地址空间。创建轻量级的进程比创建普通进程相应要快得多，而且只要父进程和子进程谨慎地协调它们的访问，就可以认为页的共享是有益的。

如果通过 `clone()` 系统调用已经创建了新进程，并且 `flag` 参数的 `CLONE_VM` 标志被设置，则 `copy_mm()` 函数把父进程 (`current`) 地址空间给予进程 (`tsk`):

```
if (clone_flags & CLONE_VM) {
    atomic_inc(&current->mm->mm_users);
    spin_unlock_wait(&current->mm->page_table_lock);
    tsk->mm = current->mm;
    tsk->active_mm = current->mm;
    return 0;
}
```

如果其他 CPU 持有进程页表自旋锁，就调用 `spin_unlock_wait()` 函数保证在释放锁之前，缺页处理程序不会结束。实际上，这个自旋锁除了保护页表以外，还必须禁止创建新的轻量级进程，因为它共享 `current->mm` 描述符。

如果没有设置 `CLONE_VM` 标志，`copy_mm()` 函数就必须创建一个新的地址空间（在进程请求一个地址之前，即使在地址空间内没有分配内存）。这个函数分配一个新的内存描述符，把它的地址存放在新进程描述符 `tsk` 的 `mm` 字段中，并把 `current->mm` 的内容复制到 `tsk->mm` 中。然后改变新进程描述符的一些字段：

```
tsk->mm = kmem_cache_alloc(mm_cachep, SLAB_KERNEL);
memcpy(tsk->mm, current->mm, sizeof(*tsk->mm));
```

```
atomic_set(&tsk->mm->mm_users, 1);
atomic_set(&tsk->mm->mm_count, 1);
init_rwsem(&tsk->mm->mmmap_sem);
tsk->mm->core_waiters = 0;
tsk->mm->page_table_lock = SPIN_LOCK_UNLOCKED;
tsk->mm->ioctx_list_lock = RW_LOCK_UNLOCKED;
tsk->mm->ioctx_list = NULL;
tsk->mm->default_kioctx = INIT_KIOCTX(tsk->mm->default_kioctx,
                                     *tsk->mm);
tsk->mm->free_area_cache = (TASK_SIZE/3+0xfff)&0xfffff000;
tsk->mm->pgd = pgd_alloc(tsk->mm);
tsk->mm->def_flags = 0;
```

回想一下，`pgd_alloc()`宏为新进程分配页全局目录。

然后调用依赖于体系结构的 `init_new_context()` 函数：对于 80x86 处理器，该函数检查当前进程是否拥有定制的局部描述符表，如果是，`init_new_context()` 复制一份 `current` 的局部描述符表并把它插入 `tsk` 的地址空间。

最后，调用 `dup_mmap()` 函数既复制父进程的线性区，也复制父进程的页表。`dup_mmap()` 函数把新内存描述符 `tsk->mm` 插入到内存描述符的全局链表中，然后，从 `current->mm->mmap` 所指向的线性区开始扫描父进程的线性区链表。它复制遇到的每个 `vm_area_struct` 线性区描述符，并把复制品插入到子进程的线性区链表和红-黑树中。

在插入一个新的线性区描述符之后，如果需要的话，`dup_mmap()` 立即调用 `copy_page_range()` 创建必要的页表来映射这个线性区所包含的一组页，并且初始化新页表的表项。尤其是，与私有的、可写的页（`VM_SHARED` 标志关闭，`VM_MAYWRITE` 标志打开）所对应的任一页框都标记为对父子进程是只读的，以便这种页框能用写时复制机制进行处理。

删除进程的地址空间

当进程结束时，内核调用 `exit_mm()` 函数释放进程的地址空间：

```
mm_release(tsk, tsk->mm);
if (!(mm = tsk->mm)) /* kernel thread ? */
    return;
down_read(&mm->mmmap_sem);
```

`mm_release()` 函数唤醒在 `tsk->vfork_done` 补充原语上睡眠的任一进程（参见第五章“补充原语”一节）。典型地，只有当现有进程通过 `vfork()` 系统调用被创建时，相应的等待队列才会为非空（参见第三章“`clone()`、`fork()`及 `vfork()`系统调用”一节）。

如果正在被终止的进程不是内核线程，`exit_mm()` 函数就必须释放内存描述符和所有相关的数据结构。首先，它检查 `mm->core_waiters` 标志是否被置位：如果是，进程就把

内存的所有内容卸载到一个转储文件中。为了避免转储文件的混乱，函数利用 `mm->core_done` 和 `mm->core_startup_done` 补充原语使共享同一个内存描述符 `mm` 的轻量级进程的执行串行化。

接下来，函数递增内存描述符的主使用计数器，重新设置进程描述符的 `mm` 字段，并使处理器处于懒惰 TLB 模式（参见第二章的“处理硬件高速缓存和 TLB”一节）：

```
atomic_inc(&mm->mm_count);
spin_lock(tsk->alloc_lock);
tsk->mm = NULL;
up_read(&mm->map_sem);
enter_lazy_tlb(mm, current);
spin_unlock(tsk->alloc_lock);
mmapput(mm);
```

最后，调用 `mmapput()` 函数释放局部描述符表、线性区描述符和页表。不过，因为 `exit_mm()` 已经递增了主使用计数器，所以并不释放内存描述符本身。当要把正在被终止的进程从本地 CPU 撤消时，将由 `finish_task_switch()` 函数释放内存描述符（参见第七章“`schedule()`函数”一节）。

堆的管理

每个 Unix 进程都拥有一个特殊的线性区，这个线性区就是所谓的堆 (*heap*)，堆用于满足进程的动态内存请求。内存描述符的 `start_brk` 与 `brk` 字段分别限定了这个区的开始地址和结束地址。

进程可以使用下面的 API 来请求和释放动态内存：

`malloc(size)`

请求 `size` 个字节的动态内存。如果分配成功，就返回所分配内存单元第一个字节的线性地址。

`calloc(n, size)`

请求含有 `n` 个大小为 `size` 的元素的一个数组。如果分配成功，就把数组元素初始化为 0，并返回第一个元素的线性地址。

`realloc(ptr, size)`

改变由前面的 `malloc()` 或 `calloc()` 分配的内存区字段的大小。

`free(addr)`

释放由 `malloc()` 或 `calloc()` 分配的起始地址为 `addr` 的线性区。

`brk(addr)`

直接修改堆的大小。`addr` 参数指定 `current->mm->brk` 的新值，返回值是线性区新的结束地址（进程必须检查这个地址和所请求的地址值 `addr` 是否一致）。

`sbrk(incr)`

类似于 `brk()`，不过其中的 `incr` 参数指定是增加还是减少以字节为单位的堆大小。

`brk()` 函数和以上列出的函数有所不同，因为它是唯一以系统调用的方式实现的函数，而其他所有的函数都是使用 `brk()` 和 `mmap()` 系统调用实现的 C 语言库函数（注 14）。

当用户态的进程调用 `brk()` 系统调用时，内核执行 `sys_brk(addr)` 函数。该函数首先验证 `addr` 参数是否位于进程代码所在的线性区。如果是，则立即返回，因为堆不能与进程代码所在的线性区重叠：

```
mm = current->mm;
down_write(&mm->mmap_sem);
if (addr < mm->end_code) {
out:
    up_write(&mm->mmap_sem);
    return mm->brk;
}
```

由于 `brk()` 系统调用作用于某一个线性区，它分配和释放完整的页。因此，该函数把 `addr` 的值调整为 `PAGE_SIZE` 的倍数，然后把调整的结果与内存描述符的 `brk` 字段的值进行比较：

```
newbrk = (addr + 0xfff) & 0xfffff000;
oldbrk = (mm->brk + 0xfff) & 0xfffff000;
if (oldbrk == newbrk) {
    mm->brk = addr;
    goto out;
}
```

如果进程请求缩小堆，则 `sys_brk()` 调用 `do_munmap()` 函数完成这项任务，然后返回：

```
if (addr <= mm->brk) {
    if (!do_munmap(mm, newbrk, oldbrk-newbrk))
        mm->brk = addr;
    goto out;
}
```

如果进程请求扩大堆，则 `sys_brk()` 首先检查是否允许进程这样做。如果进程企图分配在其限制范围之外的内存，函数并不多分配内存，只简单地返回 `mm->brk` 的原有值：

```
rlim = current->signal->rlim[RLIMIT_DATA].rlim_cur;
```

注 14: `realloc()` 库函数也可以使用 `mremap()` 系统调用。

```
if (rlim < RLIM_INFINITY && addr - mm->start_data > rlim)
    goto out;
```

然后，函数检查扩大后的堆是否和进程的其他线性区相重叠，如果是，不做任何事情就返回：

```
if (find_vma_intersection(mm, oldbrk, newbrk+PAGE_SIZE))
    goto out;
```

如果一切都顺利，则调用 `do_brk()` 函数。如果它返回 `oldbrk`，则分配成功且 `sys_brk()` 函数返回 `addr` 的值，否则，返回旧的 `mm->brk` 值：

```
if (do_brk(oldbrk, newbrk-oldbrk) == oldbrk)
    mm->brk = addr;
goto out;
```

`do_brk()` 函数实际上是仅处理匿名线性区的 `do_mmap()` 的简化版。可以认为它的调用等价于：

```
do_mmap(NULL, oldbrk, newbrk-oldbrk, PROT_READ|PROT_WRITE|PROT_EXEC,
        MAP_FIXED|MAP_PRIVATE, 0)
```

当然，`do_brk()` 比 `do_mmap()` 稍快，因为前者假定线性区不映射磁盘上的文件，从而避免了检查线性区对象的几个字段。



操作系统为在用户态运行的进程与硬件设备（如CPU、磁盘、打印机等等）进行交互提供了一组接口。在应用程序和硬件之间设置一个额外层具有很多优点。首先，这使得编程更加容易，把用户从学习硬件设备的低级编程特性中解放出来。其次，这极大地提高了系统的安全性，因为内核在试图满足某个请求之前在接口级就可以检查这种请求的正确性。最后，更重要的是这些接口使得程序更具有可移植性，因为只要内核所提供的一组接口相同，那么在任一内核之上就可以正确地编译和执行程序。

Unix系统通过向内核发出系统调用（*system call*）实现了用户态进程和硬件设备之间的大部分接口。本章将详细讨论Linux内核是如何实现这些由用户态进程向内核发出的系统调用的。

POSIX API 和系统调用

让我们先强调一下应用编程接口（API）与系统调用之不同。前者只是一个函数定义，说明了如何获得一个给定的服务；而后者是通过软中断向内核态发出一个明确的请求。

Unix系统给程序员提供了很多API的库函数。*libc*的标准C库所定义的一些API引用了封装例程（*wrapper routine*）（其唯一目的就是发布系统调用）。通常情况下，每个系统调用对应一个封装例程，而封装例程定义了应用程序使用的API。

反之则不然，顺便说一句，一个API没必要对应一个特定的系统调用。首先，API可能直接提供用户态的服务（例如一些抽象的数学函数，根本没必要使用系统调用）。其次，一个单独的API函数可能调用几个系统调用。此外，几个API函数可能调用封装了不同

功能的同一系统调用。例如，Linux的*libc*库函数实现了`malloc()`、`calloc()`和`free()`等POSIX API，这几个函数分配和释放所请求的内存，并都利用`brk()`系统调用来扩大或缩小进程的堆（heap）（参见第九章中的“堆的管理”一节）。

POSIX标准针对API而不针对系统调用。判断一个系统是否与POSIX兼容要看它是否提供了一组合适的应用程序接口，而不管对应的函数是如何实现的。事实上，一些非Unix系统被认为是与POSIX兼容的，因为它们为用户态的库函数中提供了传统Unix能提供的所有服务。

从编程者的观点看，API和系统调用之间的差别是没有关系的：唯一相关的事情就是函数名、参数类型及返回代码的含义。然而，从内核设计者的观点看，这种差别确实有关系，因为系统调用属于内核，而用户态的库函数不属于内核。

大部分封装例程返回一个整数，其值的含义依赖于相应的系统调用。返回值-1通常表示内核不能满足进程的请求。系统调用处理程序的失败可能是由无效参数引起的，也可能是因为缺乏可用资源，或硬件出了问题等等。在*libc*库中定义的`errno`变量包含特定的出错码。

每个出错码都定义为一个常量宏（产生一个相应的正整数值）。POSIX标准指定了很多出错码的宏名。在基于80x86系统的Linux中，在一个名为`include/asm-i386/errno.h`的头文件中定义了这些宏。为了使各种Unix系统上的C程序具有可移植性，在标准的C库头文件`/usr/include/errno.h`中也包含了`include/asm-i386/errno.h`头文件。其他的系统有它们自己专门的头文件子目录。

系统调用处理程序及服务例程

当用户态的进程调用一个系统调用时，CPU切换到内核态并开始执行一个内核函数。正如我们在下一节将要看到的那样，在80x86体系结构中，可以用两种不同的方式调用Linux的系统调用。两种方式的最终结果都是跳转到所谓系统调用处理程序（*system call handler*）的汇编语言函数。

因为内核实现了很多不同的系统调用，因此进程必须传递一个名为系统调用号（*system call number*）的参数来识别所需的系统调用，`eax`寄存器就用作此目的。正如我们将在本章的“参数传递”一节所看到的，当调用一个系统调用时通常还要传递另外的参数。

所有的系统调用都返回一个整数值。这些返回值与封装例程返回值的约定是不同的。在内核中，正数或0表示系统调用成功结束，而负数表示一个出错条件。在后一种情况下，这个值就是存放在`errno`变量中必须返回给应用程序的负出错码。内核没有设置或使用`errno`变量，而封装例程从系统调用返回之后设置这个变量。

系统调用处理程序与其他异常处理程序的结构类似，执行下列操作：

- 在内核态栈保存大多数寄存器的内容（这个操作对所有的系统调用都是通用的，并用汇编语言编写）。
- 调用名为系统调用服务例程（*system call service routine*）的相应的C函数来处理系统调用。
- 退出系统调用处理程序：用保存在内核栈中的值加载寄存器，CPU从内核态切换回到用户态（所有的系统调用都要执行这一相同的操作，该操作用汇编语言代码实现）。

xyz()系统调用对应的服务例程的名字通常是 sys_xyz()。不过也有一些例外。

图 10-1 显示了调用系统调用的应用程序、相应的封装例程、系统调用处理程序及系统调用服务例程之间的关系。箭头表示函数之间的执行流。占位符 "SYSCALL" 和 "SYSEXIT" 是真正的汇编语言指令，它们分别把 CPU 从用户态切换到内核态和从内核态切换到用户态。

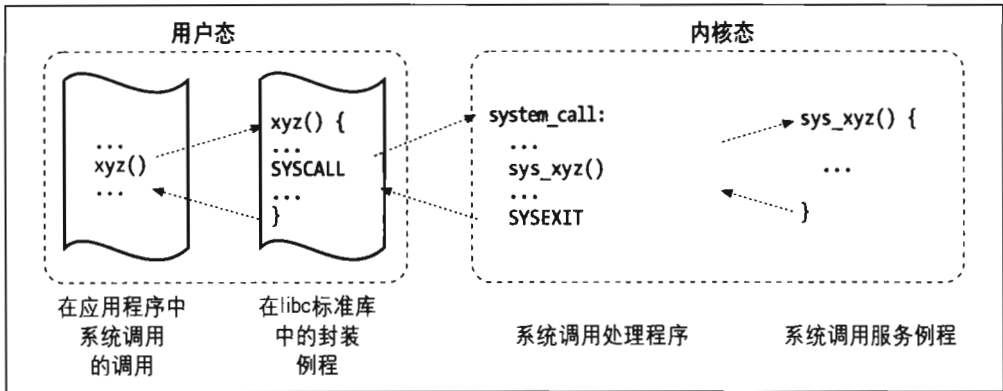


图 10-1：调用一个系统调用

为了把系统调用号与相应的服务例程关联起来，内核利用了一个系统调用分派表（*dispatch table*）。这个表存放在 sys_call_table 数组中，有 NR_syscalls 个表项（在 Linux 2.6.11 内核中是 289）：第 n 个表项包含系统调用号为 n 的服务例程的地址。

NR_syscalls 宏只是对可实现的系统调用最大个数的静态限制，并不表示实际已实现的系统调用个数。实际上，分派表中的任意一个表项也可以包含 sys_ni_syscall() 函数的地址，这个函数是“未实现”系统调用的服务例程，它仅仅返回出错码 -ENOSYS。

进入和退出系统调用

本地应用（注1）可以通过两种不同的方式调用系统调用：

- 执行 `int $0x80` 汇编语言指令。在 Linux 内核的老版本中，这是从用户态切换到内核态的唯一方式。
- 执行 `sysenter` 汇编语言指令。在 Intel Pentium II 微处理器芯片中引入了这条指令，现在 Linux 2.6 内核支持这条指令。

同样，内核可以通过两种方式从系统调用退出，从而使 CPU 切换回到用户态：

- 执行 `iret` 汇编语言指令。
- 执行 `sysexit` 汇编语言指令，它和 `sysenter` 指令同时在 Intel Pentium II 微处理器中引入。

但是，支持进入内核的两种不同方式并不像看起来那么简单，因为：

- 内核必须既支持只使用 `int $0x80` 指令的旧函数库，同时支持也可以使用 `sysenter` 指令的新函数库。
- 使用 `sysenter` 指令的标准库必须能处理仅支持 `int $0x80` 指令的旧内核。
- 内核和标准库必须既能运行在不包含 `sysenter` 指令的旧处理器上，也能运行在包含它的新处理器上。

在本章稍后“通过 `sysenter` 指令发出系统调用”一节，我们将看到 Linux 内核是如何解决这些兼容性问题的。

通过 `int $0x80` 指令发出系统调用

调用系统调用的传统方法是使用汇编语言指令 `int`，在第四章“中断和异常的硬件处理”一节曾讨论过这条指令。

向量 128（十六进制 `0x80`）对应于内核入口点。在内核初始化期间调用的函数 `trap_init()`，用下面的方式建立对应于向量 128 的中断描述符表项：

注1：就像我们在第二十章“执行域一节”中将要看到的，Linux 可以执行其他操作系统所编译的程序。因此内核提供了一种进入系统调用的兼容模式：执行 iBCS 和 Solaris/x86 程序的用户态进程可以通过跳转到默认局部描述符表中的适当调用门进入内核。

```
set_system_gate(0x80, &system_call);
```

该调用把下列值存入这个门描述符的相应字段（参见第四章中的“中断门、陷阱门及系统门”一节）：

Segment Selector

内核代码段 `__KERNEL_CS` 的段选择符。

Offset

指向 `system_call()` 系统调用处理程序的指针。

Type

置为 15。表示这个异常是一个陷阱，相应的处理程序不禁止可屏蔽中断。

DPL (描述符特权级)

置为 3。这就允许用户态进程调用这个异常处理程序（参见第四章中的“中断和异常的硬件处理”一节）。

因此，当用户态进程发出 `int $0x80` 指令时，CPU 切换到内核态并开始从地址 `system_call` 处开始执行指令。

system_call() 函数

`system_call()` 函数首先把系统调用号和这个异常处理程序可以用到的所有 CPU 寄存器保存到相应的栈中，不包括由控制单元已自动保存的 `eflags`、`cs`、`eip`、`ss` 和 `esp` 寄存器（参见第四章中的“中断和异常的硬件处理”一节）。在第四章的“*I/O 中断处理*”一节中已经讨论的 `SAVE_ALL` 宏，也在 `ds` 和 `es` 中装入内核数据段的段选择符：

```
system_call:
    pushl %eax
    SAVE_ALL
    movl $0xffff000, %ebx /* or 0xffff000 for 4-KB stacks */
    andl %esp, %ebx
```

随后，这个函数在 `ebx` 中存放当前进程的 `thread_info` 数据结构的地址，这是通过获得内核栈指针的值并把它取整到 4KB 或 8KB 的倍数而完成的（参见第三章中的“标识一个进程”一节）。

接下来，`system_call()` 函数检查 `thread_info` 结构 `flag` 字段的 `TIF_SYSCALL_TRACE` 和 `TIF_SYSCALL_AUDIT` 标志之一是否被设置为 1，也就是检查是否有某一调试程序正在跟踪执行程序对系统调用的调用。如果是这种情况，那么 `system_call()` 函数两次调用 `do_syscall_trace()` 函数：一次正好在这个系统调用服务例程执行之前，一次在其之后（稍后对其进行说明）。这个函数停止 `current`，并因此允许调试进程收集关于 `current` 的信息。

然后，对用户态进程传递来的系统调用号进行有效性检查。如果这个号大于或等于系统调用分派表中的表项数，系统调用处理程序就终止：

```
    cmpl $NR_syscalls, %eax
    jb  nobadsys
    movl $(-ENOSYS), 24(%esp)
    jmp resume_userspace
nobadsys:
```

如果系统调用号无效，该函数就把 `-ENOSYS` 值存放在栈中曾保存 `eax` 寄存器的单元中（从当前栈顶开始偏移量为 24 的单元）。然后跳到 `resume_userspace`（见下面）。这样，当进程恢复它在用户态的执行时，会在 `eax` 中发现一个负的返回码。

最后，调用与 `eax` 中所包含的系统调用号对应的特定服务例程：

```
    call *sys_call_table(0, %eax, 4)
```

因为分派表中的每个表项占 4 个字节，因此首先把系统调用号乘以 4，再加上 `sys_call_table` 分派表的起始地址，然后从这个地址单元获取指向服务例程的指针，内核就找到了要调用的服务例程。

从系统调用退出

当系统调用服务例程结束时，`system_call()` 函数从 `eax` 获得它的返回值，并把这个返回值存放在曾保存用户态 `eax` 寄存器值的那个栈单元的位置上：

```
    movl %eax, 24(%esp)
```

因此，用户态进程将在 `eax` 中找到系统调用的返回码。

然后，`system_call()` 函数关闭本地中断并检查当前进程的 `thread_info` 结构中的标志：

```
    cli
    movl 8(%ebp), %ecx
    testw $0xffff, %cx
    je  restore_all
```

`flags` 字段在 `thread_info` 结构中的偏移量为 8，掩码 `0xffff` 选择与表 4-15 中列出的所有标志（不包括 `TIF_POLLING_NRFLAG`）相对应的位。如果所有的标志都没有被设置，函数就跳转到 `restore_all` 标记处，就像在第四章“从中断和异常返回”一节中所描述的：`restore_all` 标记处的代码恢复保存在内核栈中的寄存器的值，并执行 `iret` 汇编语言指令以重新开始执行用户态进程（你可以参考如图 4-6 所示的流程图）。

只要有任何一种标志被设置，那么就要在返回用户态之前完成一些工作。如果 `TIF_SYSCALL_TRACE` 标志被设置，`system_call()` 函数就第二次调用 `do_syscall_trace()`

函数，然后跳转到 `resume_userspace` 标记处。否则，如果 `TIF_SYSCALL_TRACE` 标志没有被设置，函数就跳转到 `work_pending` 标记处。

就像在第四章“从中断和异常返回”一节中所描述的，在 `resume_userspace` 和 `work_pending` 标记处的代码检查重新调度请求、虚拟8086模式、挂起信号和单步执行，最终跳转到 `restore_all` 标记处以恢复用户态进程的执行。

通过 `sysenter` 指令发出系统调用

汇编语言指令 `int` 由于要执行几个一致性和安全性检查，所以速度较慢（这条指令的详细说明见第四章“中断和异常的硬件处理”一节）。

在 Intel 文档中被称为“快速系统调用”的 `sysenter` 指令，提供了一种从用户态到内核态的快速切换方法。

`sysenter` 指令

汇编语言指令 `sysenter` 使用三种特殊的寄存器，它们必须装入下述信息（注2）：

`SYSENTER_CS_MSR`

内核代码段的段选择符

`SYSENTER_EIP_MSR`

内核入口点的线性地址

`SYSENTER_ESP_MSR`

内核堆栈指针

执行 `sysenter` 指令时，CPU 控制单元：

1. 把 `SYSENTER_CS_MSR` 的内容拷贝到 `cs`。
2. 把 `SYSENTER_EIP_MSR` 的内容拷贝到 `eip`。
3. 把 `SYSENTER_ESP_MSR` 的内容拷贝到 `esp`。
4. 把 `SYSENTER_CS_MSR` 加 8 的值装入 `ss`。

因此，CPU 切换到内核态并开始执行内核入口点的第一条指令。就像我们在第二章

注2：“MSR”是“Model-Specific Register”的缩写，表示仅在当前一些 80×86 微处理器中存在的某个寄存器。

“Linux GDT”一节中所见到的，内核堆栈段与内核数据段是一致的，而且在全局描述符表中，其描述符紧跟在内核代码段的描述符之后，所以，第4步把正确的段选择符装入了 `ss` 寄存器。

在内核初始化期间，一旦系统中的每个CPU执行函数 `enable_sep_cpu()`，三个特定于模型的寄存器就由该函数初始化了。`enable_sep_cpu()`函数执行以下步骤：

1. 把内核代码 (`__KERNEL_CS`) 的段选择符写入 `SYSENTER_CS_MSR` 寄存器。
2. 把下面要说明的函数 `sysenter_entry()` 的线性地址写入 `SYSENTER_CS_EIP` 寄存器。
3. 计算本地TSS末端的线性地址，并把这个值写入 `SYSENTER_CS_ESP` 寄存器（注3）。

对 `SYSENTER_CS_ESP` 寄存器的设置有必要进行一些说明。系统调用开始的时候，内核堆栈是空的，因此 `esp` 寄存器应该指向4KB或8KB内存区域的末端，该内存区域包括内核堆栈和当前进程的描述符（见图3-2）。因为用户态的封装例程不知道这个内存区域的地址，所以它不能正确设置这个寄存器。另一方面，由于必须在切换到内核态之前设置该寄存器的值，因此，内核初始化这个寄存器以便为本地CPU的任务状态段编址。就像我们在 `__switch_to()` 函数的第3步所描述的（见第三章“执行进程切换”一节），在每次进程切换时，内核把当前进程的内核栈指针保存到本地TSS的 `esp0` 字段。这样，系统调用处理程序读 `esp` 寄存器，计算本地TSS的 `esp0` 字段的地址，然后把正确的内核堆栈指针装入 `esp` 寄存器。

vsyscall 页

只要CPU和Linux内核都支持 `sysenter` 指令，标准库 `libc` 中的封装函数就可以使用它。

这个兼容性问题需要非常复杂的解决方案。本质上，在初始化阶段，`sysenter_setup()` 函数建立一个称为 `vsyscall` 页的页框，其中包括一个小的EFL共享对象（也就是）一个很小的EFL动态链接库）。当进程发出 `execve()` 系统调用而开始执行一个EFL程序时，`vsyscall` 页中的代码就会自动地被链接到进程的地址空间（参见第二十章“`exec` 函数”一节）。`vsyscall` 页中的代码使用最有用的指令发出系统调用。

函数 `sysenter_setup()` 为 `vsyscall` 页分配一个新页框，并把它的物理地址与 `FIX_VSYSCALL` 固定映射的线性地址相关联（参见第二章“固定映射的线性地址”一节）。然后，函数 `sysenter_setup()` 把预先定义好的一个或两个EFL共享对象拷贝到该页中：

注3：把本地TSS地址编码写入 `SYSENTER_ESP_MSR` 寄存器，是因为该寄存器应该指向一个实际的栈，该栈向低地址方向增长。实际上可以用任何值初始化 `SYSENTER_ESP_MSR` 寄存器，只要能够从这个值获得本地TSS的地址即可。

- 如果 CPU 不支持 `sysenter`, `sysenter_setup()` 函数建立一个包括下列代码的 `vsyscall` 页:

```
__kernel_vsyscall:
    int $0x80
    ret
```

- 否则, 如果 CPU 的确支持 `sysenter`, `sysenter_setup()` 函数建立一个包括下列代码的 `vsyscall` 页:

```
__kernel_vsyscall:
    pushl %ecx
    pushl %edx
    pushl %ebp
    movl %esp, %ebp
    sysenter
```

当标准库中的封装例程必须调用系统调用时, 都调用 `__kernel_vsyscall()` 函数, 不管它的实现代码是什么。

最后一个兼容性问题是由于老版本的 Linux 内核不支持 `sysenter` 指令, 在这种情况下, 内核当然不建立 `vsyscall` 页, 而且函数 `__kernel_vsyscall()` 不会被链接到用户态进程的地址空间。当新近的标准库识别出这种状况后, 就简单地执行 `int $0x80` 指令来调用系统调用。

进入系统调用

当用 `sysenter` 指令发出系统调用时, 依次执行下述步骤:

1. 标准库中的封装例程把系统调用号装入 `eax` 寄存器, 并调用 `__kernel_vsyscall()` 函数。
2. 函数 `__kernel_vsyscall()` 把 `ebp`、`edx` 和 `ecx` 的内容保存到用户态堆栈中 (系统调用处理程序将使用这些寄存器), 把用户栈指针拷贝到 `ebp` 中, 然后执行 `sysenter` 指令。
3. CPU 从用户态切换到内核态, 内核开始执行 `sysenter_entry()` 函数 (由 `SYSENTER_EIP_MSR` 寄存器指向)。
4. `sysenter_entry()` 汇编语言函数执行下述步骤:
 - a. 建立内核堆栈指针:

```
movl -508(%esp), %esp
```

开始时, `esp` 寄存器指向本地 TSS 的第一个位置, 本地 TSS 的大小为 512 字节。因此, `sysenter` 指令把本地 TSS 中偏移量为 4 处的字段的内容 (就是 `esp0`

字段的内容) 装入 `esp`。就像前面我们已经说明的, `esp0` 字段总是存放当前进程的内核堆栈指针。

- b. 打开本地中断:

```
sti
```

- c. 把用户数据段的段选择符、当前用户栈指针、`eflags` 寄存器、用户代码段的段选择符以及从系统调用退出时要执行的指令的地址保存到内核堆栈中:

```
pushl $(__USER_DS)
pushl %ebp
pushfl
pushl $(__USER_CS)
pushl $SYSENTER_RETURN
```

这些指令仿效汇编语言指令 `int` 所执行的一些操作(参见第四章“中断和异常的硬件处理”一节中对 `int` 描述的第 5c 步和第 7 步操作)。

- d. 把原来由封装例程传递的寄存器的值恢复到 `ebp` 中:

```
movl (%ebp), %ebp
```

上面这条指令完成恢复的工作, 因为 `__kernel_vsyscall()` 函数把 `ebp` 的原始值存入用户态堆栈中, 并在随后把用户堆栈指针的当前值装入 `ebp` 中。

- e. 通过执行一系列指令调用系统调用处理程序, 这些指令与前面“通过 `int $0x80` 指令发出系统调用”一节所描述的在 `system_call` 标记处开始的指令是一样的。

退出系统调用

当系统调用服务例程结束时, `sysenter_entry()` 函数本质上执行与 `system_call()` 函数相同的操作(见上一节)。首先, 它从 `eax` 获得系统调用服务例程的返回码, 并将返回码存入内核栈中保存用户态 `eax` 寄存器值的位置。然后, 函数禁止本地中断, 并检查 `current` 的 `thread_info` 结构中的标志。

如果有任何标志被设置, 那么在返回到用户态之前还需要完成一些工作。为了避免代码复制(像在 `system_call()` 函数中所做的那样正确处理这种情况), 函数跳转到 `resume_userspace` 或 `work_pending` 标记处(参见第四章图 4-6 所示的流程图)。最后, 汇编语言指令 `iret` 从内核堆栈中去取 5 个参数(这些参数是在 `sysenter_entry()` 函数的第 4c 步被保存到内核堆栈中的, 这样, CPU 切换到用户态并开始执行 `SYSENTER_RETURN` 标记处的代码(见下面)。

如果 `sysenter_entry()` 函数确定标志都被清 0, 它就快速返回到用户态:

```
movl 40(%esp), %edx
movl 52(%esp), %ecx
xorl %ebp, %ebp
sti
sysexit
```

把在上一节由 `sysenter_entry()` 函数在第 4c 步保存的一对堆栈值加载到 `edx` 和 `ecx` 寄存器中：`edx` 获得 `SYSENTER_RETURN` 标记处的地址，而 `ecx` 获得当前用户数据栈的指针。

sysexit 指令

`sysexit` 是与 `sysenter` 配对的汇编语言指令：它允许从内核态快速切换到用户态。执行这条指令时，CPU 控制单元执行下述步骤：

1. 把 `SYSENTER_CS_MSR` 寄存器中的值加 16 所得到的结果加载到 `cs` 寄存器。
2. 把 `edx` 寄存器的内容拷贝到 `eip` 寄存器。
3. 把 `SYSENTER_CS_MSR` 寄存器中的值加 24 所得到的结果加载到 `ss` 寄存器。
4. 把 `ecx` 寄存器的内容拷贝到 `esp` 寄存器。

因为 `SYSENTER_CS_MSR` 寄存器加载的是内核代码的段选择符，`cs` 寄存器加载的是用户代码的段选择符，而 `ss` 寄存器加载的是用户数据段的段选择符（参见第二章“LinuxLDT”一节）。

结果，CPU 从内核态切换到用户态，并开始执行其地址存放在 `edx` 中的那条指令。

SYSENTER_RETURN 的代码

`SYSENTER_RETURN` 标记处的代码存放在 `vsyscall` 页中，当通过 `sysenter` 进入的系统调用被 `iret` 或 `sysexit` 指令终止时，该页框中的代码被执行。

该代码片段恢复保存在用户态堆栈中的 `ebp`、`edx` 和 `ecx` 寄存器的原始内容，并把控制权返回给标准库中的封装例程：

```
SYSENTER_RETURN:
    popl %ebp
    popl %edx
    popl %ecx
    ret
```

参数传递

与普通函数类似，系统调用通常也需要输入/输出参数，这些参数可能是实际的值（例如数值），也可能是用户态进程地址空间的变量，甚至是指向用户态函数的指针的数据结构地址（参见第十一章“与信号处理相关的系统调用”一节）。

因为`system_call()`和`sysenter_entry()`函数是Linux中所有系统调用的公共入口点，因此每个系统调用至少有一个参数，即通过`eax`寄存器传递来的系统调用号。例如，如果一个应用程序调用`fork()`封装例程，那么在执行`int $0x80`或`sysenter`汇编指令之前就把`eax`寄存器置为2（即`__NR_fork`）。因为这个寄存器的设置是由`libc`库中的封装例程进行的，因此程序员通常并不关心系统调用号。

`fork()`系统调用并不需要其他的参数。不过，很多系统调用确实需要由应用程序明确地传递另外的参数。例如，`mmap()`系统调用可能需要多达6个额外参数（除了系统调用号以外）。

普通C函数的参数传递是通过把参数值写入活动的程序栈（用户态栈或者内核态栈）实现的。因为系统调用是一种横跨用户和内核两大块地的特殊函数，所以既不能使用用户态栈也不能使用内核态栈。更确切地说，在发出系统调用之前，系统调用的参数被写入CPU寄存器，然后在调用系统调用服务例程之前，内核再把存放在CPU中的参数拷贝到内核态堆栈中，这是因为系统调用服务例程是普通的C函数。

为什么内核不直接把参数从用户态的栈拷贝到内核态的栈呢？首先，同时操作两个栈是比较复杂的。其次，寄存器的使用使得系统调用处理程序的结构与其他异常处理程序的结构类似。

然而，为了用寄存器传递参数，必须满足两个条件：

- 每个参数的长度不能超过寄存器的长度，即32位（注4）。
- 参数的个数不能超过6个（除了`eax`中传递的系统调用号），因为80x86处理器的寄存器的数量是有限的。

第一个条件总能成立，因为根据POSIX标准，不能存放在32位寄存器中的长参数必须通过指定它们的地址来传递。一个典型的例子就是`settimeofday()`系统调用，它必须读一个64位的结构。

注4：与往常一样，这里指的是80x86处理器的32位体系结构。本节的讨论并不适用于64位体系结构。

然而，确实存在多于6个参数的系统调用。在这样的情况下，用一个单独的寄存器指向进程地址空间中这些参数值所在的一个内存区。当然，编程者不用关心这个工作区。正如任何C调用一样，当调用封装例程时，参数被自动地保存在栈中。封装例程将找到合适的方式把参数传递给内核。

用于存放系统调用号和系统调用参数的寄存器是（以字母递增的顺序）：`eax`（存放系统调用号）、`ebx`、`ecx`、`edx`、`esi`、`edi`以及`ebp`。正如以前看到的一样，`system_call()`和`sysenter_entry()`使用`SAVE_ALL`宏把这些寄存器的值保存在内核态堆栈中。因此，当系统调用服务例程转到内核态堆栈时，就会找到`system_call()`或`sysenter_entry()`的返回地址，紧接着是存放在`ebx`中的参数（即系统调用的第一个参数），存放在`ecx`中的参数等等（参见第四章中的“为中断处理程序保存寄存器的值”一节）。这种栈结构与普通函数调用的栈结构完全相同，因此，系统调用服务例程很容易通过使用C语言结构来引用它的参数。

让我们来看一个例子。处理`write()`系统调用的`sys_write()`服务例程的声明如下：

```
int sys_write (unsigned int fd, const char * buf, unsigned int count)
```

C编译器产生一个汇编语言函数，该函数期望在栈的顶部找到`fd`、`buf`和`count`参数，而这些参数位于返回地址（就是用来分别存放`ebx`、`ecx`和`edx`寄存器的那些位置）的下面。

在少数情况下，即使系统调用不使用任何参数，相应的服务例程也需要知道在发出系统调用之前CPU寄存器的内容。例如，实现了`fork()`的`do_fork()`函数需要知道有关寄存器的值，以便在子进程的`thread`字段中复制它们（参见第三章的“`thread`字段”一节）。在这些情况下，类型为`pt_regs`的一个单独参数允许服务例程访问由`SAVE_ALL`宏保存在内核态堆栈中的值（参见第四章中的“`do_IRQ()`函数”一节）：

```
int sys_fork (struct pt_regs regs)
```

服务例程的返回值必须写入`eax`寄存器中。这是在执行“`return n;`”指令时由C编译程序自动完成的。

验证参数

在内核打算满足用户的请求之前，必须仔细地检查所有的系统调用参数。检查的类型既依赖于系统调用，也依赖于特定的参数。让我们再回到前面引入的`write()`系统调用：`fd`参数应该是描述一个特定文件的文件描述符，因此，`sys_write()`必须检查`fd`是否确实是以前已打开文件的一个文件描述符，是否允许进程向这个文件中写数据（参见第一章中的“文件操作的系统调用”一节）。如果这些条件中有一个不成立，那么这个处理程序必须返回一个负数，在这种情况下的出错码为`-EBADF`。

然而，有一种检查对所有的系统调用都是通用的。只要一个参数指定的是地址，那么内核必须检查它是否在这个进程的地址空间之内。有两种可能的方式来执行这种检查：

- 验证这个线性地址是否属于进程的地址空间，如果是，这个线性地址所在的线性区就具有正确的访问权限。
- 仅仅验证这个线性地址是否小于PAGE_OFFSET(即没有落在留给内核的线性地址区间内)。

早期的Linux内核执行第一种检查。但是这是非常费时的，因为它必须对系统调用中包含的每个地址参数都进行检查；此外，这通常没有什么意义，因为有错误的程序并不是很普遍。

因此，从Linux 2.2内核开始执行第二种检查。这是一种更高效的检查，因为不需要对进程的线性区描述符进行任何扫描。很显然，这是一种非常粗略的检查，验证线性地址小于PAGE_OFFSET是判断它的有效性的必要条件而不是充分条件。但是，因为其他的错误可以在随后捕获到，所以内核使用这种有限的检查没有任何风险。

因此，接着采用的方法是将真正的检查尽可能向后推迟，也就是说，推迟到分页单元将线性地址转换为物理地址时。我们将在本章稍后的“动态地址检查：修正代码”一节中讨论，缺页异常处理程序如何成功地检测到由用户态进程以参数传递的这些地址在内核态是无效的。

在这里，你可能想知道究竟为什么要进行这种粗略检查。事实上，这种粗略的检查是至关重要的，它确保了进程地址空间和内核地址空间都不被非法访问。我们在第二章中已经看到，RAM的映射是从PAGE_OFFSET开始的。这就意味着内核例程能对内存中现有的所有页进行寻址。因此，如果不进行这种粗略检查，用户态进程就可能把属于内核地址空间的一个地址作为参数来传递，然后还能对内存中现有的任何页进行读写而不引起缺页异常。

对系统调用所传递地址的检查是通过access_ok()宏实现的，它有两个分别为addr和size的参数。该宏检查addr到addr+size-1之间的地址区间，本质上等价于下面的C函数：

```
int access_ok(const void * addr, unsigned long size)
{
    unsigned long a = (unsigned long) addr;
    if (a + size < a ||
        a + size > current_thread_info()->addr_limit.seg)
        return 0;
    return 1;
}
```

该函数首先验证 $addr+size$ (要检查的最高地址) 是否大于 $2^{32}-1$ ；这是因为 GNU C 编译器 (gcc) 用 32 位数表示无符号长整型数和指针，这就等价于对溢出条件进行检查。该函数还检查 $addr+size$ 是否超过 `current` 的 `thread_info` 结构的 `addr_limit.seg` 字段中存放的值。通常情况下，普通进程这个字段的值是 `PAGE_OFFSET`，内核线程是 `0xffffffff`。可以通过 `get_fs` 和 `set_fs` 宏动态地改变 `addr_limit.seg` 字段的值；这就允许内核绕过由 `access_ok()` 执行的安全性检查，调用系统调用的服务例程，并直接把内核数据段的地址传递给它们。

函数 `verify_area()` 执行与 `access_ok()` 宏类似的检查，虽然它被认为是陈旧过时的，但在源代码中仍然被广泛使用。

访问进程地址空间

系统调用服务例程需要非常频繁地读写进程地址空间的数据。Linux 包含的一组宏使这种访问更加容易。我们将描述其中的两个名为 `get_user()` 和 `put_user()` 的宏。第一个宏用来从一个地址读取 1、2 或 4 个连续字节，而第二个宏用来把这几种大小的内容写入一个地址中。

这两个函数都接收两个参数，一个要传送的值 `x` 和一个变量 `ptr`。第二个变量还决定有多少个字节要传送。因此，在 `get_user(x, ptr)` 中，由 `ptr` 指向的变量大小使该函数展开为 `__get_user_1()`、`__get_user_2()` 或 `__get_user_4()` 汇编语言函数。让我们看一下其中一个，比如，`__get_user_2()`：

```
__get_user_2:
    addl $1, %eax
    jc bad_get_user
    movl $0xffffe000, %edx /* or 0xfffff000 for 4-KB stacks */
    andl %esp, %edx
    cmpl 24(%edx), %eax
    jae bad_get_user
2:  movzwl -1(%eax), %edx
    xorl %eax, %eax
    ret
bad_get_user:
    xorl %edx, %edx
    movl $-EFAULT, %eax
    ret
```

`eax` 寄存器包含要读取的第一个字节的地址 `ptr`。前 6 个指令所执行的检查事实上与 `access_ok()` 宏相同，即确保要读取的两个字节的地址小于 4GB 并小于 `current` 进程的 `addr_limit.seg` 字段 (这个字段位于 `current` 的 `thread_info` 结构中偏移量为 24 处，出现在 `cmpl` 指令的第一个操作数中)。

如果这个地址有效，函数就执行 `movzwl` 指令，把要读的数据存到 `edx` 寄存器的两个低字节，而把两个高字节置为 0，然后在 `eax` 中设置返回码 0 并终止。如果这个地址无效，函数清 `edx`，将 `eax` 置为 `-EFAULT` 并终止。

`put_user(x, ptr)` 宏类似于前边讨论的 `get_user`，但它把值 `x` 写入以地址 `ptr` 为起始地址的进程地址空间。根据 `x` 的大小，它使用 `__put_user_asm()` 宏（大小为 1、2 或 4 字节），或 `__put_user_u64()` 宏（大小为 8 字节）。这两个宏如果成功地写入了值，那它们在 `eax` 寄存器中返回 0，否则返回 `-EFAULT`。

在表 10-1 中列出了内核态下用来访问进程地址空间的另外几个函数或宏。注意，许多函数或宏的名字前缀有两个下划线 (`__`)。首部没有下划线的函数或宏要用额外的时间对所请求的线性地址区间进行有效性检查，而有下划线的则会跳过检查。当内核必须重复访问进程地址空间的同一块线性区时，比较高效的办法是开始时只对该地址检查一次，以后就不用再对该进程区进行检查了。

表 10-1：访问进程地址空间的函数和宏

函数	操作
<code>get_user __get_user</code>	从用户空间读一个整数（1、2 或 4 字节）
<code>put_user __put_user</code>	给用户空间写一个整数（1、2 或 4 字节）
<code>copy_from_user __copy_from_user</code>	从用户空间复制任意大小的块
<code>copy_to_user __copy_to_user</code>	把任意大小的块复制到用户空间
<code>strncpy_from_user __strncpy_from_user</code>	从用户空间复制一个以 null 结束的字符串
<code>strlen_user strlen_user</code>	返回用户空间以 null 结束的字符串的长度
<code>clear_user __clear_user</code>	用 0 填充用户空间的一个内存区域

动态地址检查:修正代码

正如前面所看到的，`access_ok()` 宏对系统调用以参数传递来的线性地址的有效性只进行粗略检查。该检查只保证用户态进程不会试图侵扰内核地址空间。但是，由参数传递的线性地址依然可能不属于进程地址空间。在这种情况下，当内核试图使用任何这种错误地址时，将会发生缺页异常。

在描述内核如何检测这种错误之前，我们先说明一下在内核态引起缺页异常的四种情况。这些情况必须由缺页异常处理程序来区分，因为不同情况采取的操作很不相同：

1. 内核试图访问属于进程地址空间的页，但是，或者是相应的页框不存在，或者是内核试图去写一个只读页。在这些情况下，处理程序必须分配和初始化一个新的页框（参见第九章“请求调页”和“写时复制”两节）。

2. 内核寻址到属于其地址空间的页,但是相应的页表项还没有被初始化(参见第九章“处理非连续内存区访问”一节)。在这种情况下,内核必须在当前进程页表中适当地建立一些表项。
3. 某一内核函数包含编程错误,当这个函数运行时就引起异常;或者,可能由于瞬时的硬件错误引起异常。当这种情况发生时,处理程序必须执行一个内核漏洞(参见第九章的“处理地址空间以外的错误地址”一节)。
4. 本章所讨论的一种情况:系统调用服务例程试图读写一个内存区,而该内存区的地址是通过系统调用参数传递来的,但却不属于进程的地址空间。

通过确定错误的线性地址是否属于进程所拥有的线性地址区间,缺页处理程序可以很容易地识别第一种情况。通过检查相应的主内核页表是否包含一个映射该地址的非空项可以检测第二种情况。下面让我们解释一下缺页处理程序如何区分另外两种情况。

异常表

决定缺页的来源关键在于内核使用有限的范围访问进程的地址空间。只有前一节描述的少数函数和宏用来访问进程的地址空间;因此,如果异常是由一个无效的参数引起的,那么,引起异常的指令一定包含在其中的一个函数中或由展开的宏产生。这些对用户空间寻址的指令数量是非常少的。

因此,把访问进程地址空间的每条内核指令的地址放到一个叫异常表(*exception table*)的结构中并不用费太多功夫。如果我们成功地做到这点,其他的事情就很容易了。当在内核态发生缺页异常时,do_page_fault()处理程序检查异常表:如果表中包含产生异常的指令地址,那么这个错误就是由非法的系统调用参数引起的,否则,就是由某一更严重的bug引起的。

Linux定义了几个异常表。主要的异常表在建立内核程序映像时由C编译器自动生成。它存放在内核代码段的__ex_table节,其起始与终止地址由C编译器产生的两个符号__start__ex_table和__stop__ex_table来标识。

此外,每个动态装载的内核模块(参看附录二)都包含有自己的局部异常表。这个表是在建立模块映像时由C编译器自动产生的,当把模块插入到运行中的内核时把这个表装入内存。

每一个异常表的表项是一个exception_table_entry结构,它有两个字段:

insn

访问进程地址空间的指令的线性地址。

fixup

当存放在 `insn` 单元中的指令所触发的缺页异常发生时，`fixup` 就是要调用的汇编语言代码的地址。

修正代码由几条汇编指令组成，用以解决由缺页异常所引起的问题。在后面我们将会看到，修正通常由插入的一个指令序列组成，这个指令序列强制服务例程返回一个出错码给用户态进程。这些指令通常在访问进程地址空间的同一函数或宏中定义；由 C 编译器把它们放置在内核代码段的一个叫作 `.fixup` 的独立部分。

`search_exception_tables()` 函数用来在所有异常表中查找一个指定地址：若这个地址在某一个表中，则返回指向相应 `exception_table_entry` 结构的指针；否则，返回 `NULL`。因此，缺页处理程序 `do_page_fault()` 执行下列语句：

```
if ((fixup = search_exception_tables(regs->eip)) {
    regs->eip = fixup->fixup;
    return 1;
}
```

`regs->eip` 字段包含异常发生时保存到内核态栈 `eip` 寄存器中的值。如果 `eip` 寄存器（指令指针）中的这个值在某个异常表中，`do_page_fault()` 就把所保存的值替换为 `search_exception_tables()` 的返回地址。然后缺页处理程序终止，被中断的程序以修正代码的执行而恢复运行。

生成异常表和修正代码

GNU 汇编程序（Assembler）伪指令 `.section` 允许程序员指定可执行文件的哪部分包含紧接着要执行的代码。我们将在第二十章中看到，可执行文件包括一个代码段，这个代码段可能又依次被划分为节。因此，下面的汇编指令在异常表中加入一个表项；“a”属性指定必须把这一节与内核映像的剩余部分一块加载到内存中。

```
.section __ex_table, "a"
    .long faulty_instruction_address, fixup_code_address
.previous
```

`.previous` 伪指令强制汇编程序把紧接着的代码插入到遇到上一个 `.section` 伪指令时激活的节。

让我们再看一下前面论及过的 `__get_user_1()`、`__get_user_2()` 和 `__get_user_4()` 函数。访问进程地址空间的指令用 1、2 和 3 标记。

```
__get_user_1:
    [...]
```

```

1: movzbl (%eax), %edx
   [...]
__get_user_2:
   [...]
2: movzwl -1(%eax), %edx
   [...]
__get_user_4:
   [...]
3: movl -3(%eax), %edx
   [...]
bad_get_user:
   xorl %edx, %edx
   movl $-EFAULT, %eax
   ret
.section __ex_table,"a"
   .long 1b, bad_get_user
   .long 2b, bad_get_user
   .long 3b, bad_get_user
.previous

```

每个异常表项由两个标号组成。第一个是一个数字标号，其前缀b表示标号是“向后的”，换句话说，标号出现在程序的前一行。修正代码对这三个函数是公用的，且被标记为bad_get_user。如果缺页异常是由标号1、2或3处的指令产生的，那么修正代码就执行。在bad_get_user处的修正代码给发出系统调用的进程只简单地返回一个出错码-EFAULT。

其他作用于用户态地址空间的内核函数也使用修正代码技术。再看下一个例子，strlen_user(string)宏。该宏或者返回系统调用中作为参数传递的以null结束的字符串string的长度，或在出错时返回0。这个宏本质上产生以下的汇编指令：

```

   movl $0, %eax
   movl $0x7fffffff, %ecx
   movl %ecx, %ebx
   movl string, %edi
0: repne; scasb
   subl %ecx, %ebx
   movl %ebx, %eax
1:
.section .fixup,"ax"
2: xorl %eax, %eax
   jmp 1b
.previous
.section _ __ex_table,"a"
   .long 0b, 2b
.previous

```

ecx和ebx寄存器的初始值设为0x7fffffff，表示用户态地址空间字符串的最大长度。repne; scasb汇编指令循环扫描由edi指向的字符串，在eax中查找值为0的字符（字

字符串的结束标志\0字符)。因为每一次循环 `scasb` 都将 `ecx` 减 1，所以 `eax` 中最后存放的是在字符串中所扫描过的字节总数（也就是字符串的长度）。

这个宏的修正代码被插入 `.fixup` 节。“`ax`”属性指定这一节必须加载到内存并包含可执行代码。如果缺页异常是由标号为 0 的指令引起，就执行修正代码，它只简单地把 `eax` 置为 0——因此强制该宏返回一个出错码 0 而不是字符串长度——然后跳转到标号 1，即宏之后的相应指令。

第二个 `.section` 指令在 `__ex_table` 中增加一个表项，内容包括 `repne, scasb` 指令的地址和相应的修正代码的地址。

内核封装例程

尽管系统调用主要由用户态进程使用，但也可以被内核线程调用，内核线程不能使用库函数。为了简化相应封装例程的声明，Linux 定义了 7 个从 `_syscall0` 到 `_syscall6` 的一组宏。

每个宏名字中的数字 0~6 对应着系统调用所用的参数个数(系统调用号除外)。也可以用这些宏来声明没有包含在 `libc` 标准库中的封装例程（例如，因为 Linux 系统调用还未受到库的支持）。然而，不能用这些宏来为超过 6 个参数(系统调用号除外)的系统调用或产生非标准返回值的系统调用定义封装例程。

每个宏严格地需要 $2+2 \times n$ 个参数， n 是系统调用的参数个数。前两个参数指明系统调用的返回值类型和名字，每一对附加参数指明相应的系统调用参数的类型和名字。因此，以 `fork()` 系统调用为例，其封装例程可以通过如下语句产生：

```
__syscall0(int, fork)
```

而 `write()` 系统调用的封装例程可以通过如下语句产生：

```
__syscall3(int, write, int, fd, const char *, buf, unsigned int, count)
```

在后一种情况下，可以把这个宏展开成如下的代码：

```
int write(int fd, const char * buf, unsigned int count)
{
    long __res;
    asm("int $0x80
        : "=a" (__res)
        : "0" (__NR_write), "b" ((long)fd),
          "c" ((long)buf), "d" ((long)count));
    if ((unsigned long) __res >= (unsigned long) -129) {
        errno = -__res;
        __res = -1;
    }
}
```

```
    return (int) __res;
}
```

`__NR_write` 宏来自 `_syscall3` 的第二个参数；它可以展开成 `write()` 的系统调用号。当编译前面的函数时，生成下面的汇编代码：

```
write:
    pushl %ebx           ; 将 ebx 推入堆栈
    movl 8(%esp), %ebx   ; 将第一个参数放入 ebx
    movl 12(%esp), %ecx  ; 将第二个参数放入 ecx
    movl 16(%esp), %edx  ; 将第三个参数放入 edx
    movl $4, %eax        ; 将 __NR_write 放入 eax
    int $0x80           ; 调用系统调用
    cmpl $-125, %eax     ; 检测返回码
    jbe .L1             ; 如无错则跳转
    negl %eax           ; 求 eax 的补码
    movl %eax, errno    ; 将结果放入 errno
    movl $-1, %eax      ; 将 eax 置为 -1
.L1: popl %ebx          ; 从堆栈弹出 ebx
    ret                 ; 返回调用程序
```

注意 `write()` 函数的参数是如何在执行 `int $0x80` 指令前被装入到 CPU 寄存器中的。如果 `eax` 中的返回值在 `-1 ~ -129` 之间，则必须被解释为出错码（内核假定在 `include/generic/errno.h` 中定义的最大出错码为 129）。如果是这种情况，封装例程就在 `errno` 中存放 `-eax` 的值并返回值 `-1`；否则，返回 `eax` 中的值。

第十一章



信号

信号在最早的Unix系统中即被引入，用于在用户态进程间通信。内核也用信号通知进程系统所发生的事件。信号已有30多年的历史，但只有很小的变化。

本章的第一部分详细考察Linux内核如何处理信号，然后，我们讨论几个允许进程交换信号的系统调用。

信号的作用

信号(signal)是很短的消息，可以被发送到一个进程或一组进程。发送给进程的唯一信息通常是一个数，以此来标识信号。在标准信号中，对参数、消息或者其他相随的信息没有给予关注。

名字前缀为SIG的一组宏用来标识信号。在前几章中，我们已经涉及到几个信号。例如，在第三章的“clone()、fork()及vfork()系统调用”一节中已提及到的SIGCHLD宏。在Linux中，这个宏扩展为值17，当某一子进程停止或终止时，SIGCHLD宏产生发送给父进程的信号标识符。SIGSEGV宏扩展为值11，在第九章的“缺页异常处理程序”一节中已提及到：当一个进程引用无效的内存时，SIGSEGV宏产生发送给进程的信号标识符。

使用信号的两个主要目的是：

- 让进程知道已经发生了一个特定的事件。
- 强迫进程执行它自己代码中的信号处理程序。

当然，这两个目的不是互斥的，因为进程经常通过执行一个特定的例程来对某一事件做出反应。

表 11-1 列出了基于 80x86 的 Linux 2.6 所处理的前 31 个信号（像 SIGCHLD 或 SIGSTOP 这样的一些信号是与体系结构相关的；此外，像 SIGSTKFLT 这样的一些信号只为特定的体系结构而定义）。缺省操作的含义将在下一节描述。

表 11-1: Linux/i386 中的前 31 个信号

编号	信号名称	缺省操作	解释	POSIX
1	SIGHUP	Terminate	挂起控制终端或进程	是
2	SIGINT	Terminate	来自键盘的中断	是
3	SIGQUIT	Dump	从键盘退出	是
4	SIGILL	Dump	非法指令	是
5	SIGTRAP	Dump	跟踪的断点	否
6	SIGABRT	Dump	异常结束	是
6	SIGIOT	Dump	等价于 SIGABRT	否
7	SIGBUS	Dump	总线错误	否
8	SIGFPE	Dump	浮点异常	是
9	SIGKILL	Terminate	强迫进程终止	是
10	SIGUSR1	Terminate	对进程可用	是
11	SIGSEGV	Dump	无效的内存引用	是
12	SIGUSR2	Terminate	对进程可用	是
13	SIGPIPE	Terminate	向无读者的管道写	是
14	SIGALRM	Terminate	实时定时器时钟	是
15	SIGTERM	Terminate	进程终止	是
16	SIGSTKFLT	Terminate	协处理器栈错误	否
17	SIGCHLD	Ignore	子进程停止、结束或在被跟踪时获得信号	是
18	SIGCONT	Continue	如果已停止则恢复执行	是
19	SIGSTOP	Stop	停止进程执行	是
20	SIGTSTP	Stop	从 tty 发出停止进程	是
21	SIGTTIN	Stop	后台进程请求输入	是
22	SIGTTOU	Stop	后台进程请求输出	是
23	SIGURG	Ignore	套接字上的紧急条件	否

表 11-1: Linux/i386 中的前 31 个信号 (续)

编号	信号名称	缺省操作	解释	POSIX
24	SIGXCPU	Dump	超过 CPU 时限	否
25	SIGXFSZ	Dump	超过文件大小的限制	否
26	SIGVTALRM	Terminate	虚拟定时器时钟	否
27	SIGPROF	Terminate	概况定时器时钟	否
28	SIGWINCH	Ignore	窗口调整大小	否
29	SIGIO	Terminate	I/O 现在可能发生	否
29	SIGPOLL	Terminate	等价于 SIGIO	否
30	SIGPWR	Terminate	电源供给失效	否
31	SIGSYS	Dump	坏的系统调用	否
31	SIGUNUSED	Dump	等价于 SIGSYS	否

除了在这张表中描述的常规信号 (*regular signal*) 外, POSIX 标准还引入了一类新的信号, 叫做实时信号 (*real-time signal*); 在 Linux 中它们的编码范围为 32~64。它们与常规信号有很大的不同, 因为它们必须排队以便发送的多个信号能被接收到。另一方面, 同种类型的常规信号并不排队: 如果一个常规信号被连续发送多次, 那么, 只有其中的一个发送到接收进程。尽管 Linux 内核并不使用实时信号, 它还是通过几个特定的系统调用完全实现了 POSIX 标准。

许多系统调用允许程序员发送信号, 并决定他们的进程如何响应所接收的信号。表 11-2 简洁地描述了这些系统调用, 更详细的内容将在后面“与信号处理相关的系统调用”一节中描述。

表 11-2: 与信号相关的最重要的系统调用

系统调用	说明
kill()	向线程组发送一个信号
tkill()	向进程发送一个信号
tgkill()	向一个特定线程组中的进程发送信号
sigaction()	改变与信号相关的操作
signal()	类似于 sigaction()
sigpending()	检查是否有挂起的信号
sigprocmask()	修改阻塞信号的集合
sigsuspend()	等待一个信号

表 11-2: 与信号相关的最重要的系统调用 (续)

系统调用	说明
<code>rt_sigaction()</code>	改变与实时信号相关的操作
<code>rt_sigpending()</code>	检查是否挂起实时信号
<code>rt_sigprocmask()</code>	修改阻塞的实时信号的集合
<code>rt_sigqueueinfo()</code>	向线程组发送一个实时信号
<code>rt_sigsuspend()</code>	等待一个实时信号
<code>rt_sigtimedwait()</code>	类似于 <code>rt_sigsuspend()</code>

信号的一个重要特点是它们可以随时被发送给状态经常不可预知的进程。发送给非运行进程的信号必须由内核保存，直到进程恢复执行。阻塞一个信号（后面描述）要求信号的传递拖延，直到随后解除阻塞，这使得信号产生一段时间之后才能对其传递这一问题变得更加严重。

因此，内核区分信号传递的两个不同阶段：

信号产生

内核更新目标进程的数据结构以表示一个新信号已被发送。

信号传递

内核强迫目标进程通过以下方式对信号做出反应：或改变目标进程的执行状态，或开始执行一个特定的信号处理程序，或两者都是。

每个所产生的信号至多被传递一次。信号是可消费资源：一旦它们已传递出去，进程描述符中有关这个信号的所有信息都被取消。

已经产生但还没有传递的信号称为挂起信号 (*pending signal*)。任何时候，一个进程仅存在给定类型的一个挂起信号，同一进程同种类型的其他信号不被排队，只被简单地丢弃。但是，实时信号是不同的：同种类型的挂起信号可以有好几个。

一般来说，信号可以保留不可预知的挂起时间。必须考虑下列因素：

- 信号通常只被当前正运行的进程传递（即由 `current` 进程传递）。
- 给定类型的信号可以由进程选择性地阻塞 (*blocked*)（参见“修改阻塞信号的集合”一节）。这种情况下，在取消阻塞前进程将不接收这个信号。
- 当进程执行一个信号处理程序的函数时，通常“屏蔽”相应的信号，即自动阻塞这个信号直到处理程序结束。因此，所处理的信号的另一次出现不能中断信号处理程序，所以，信号处理函数不必是可重入的。

尽管信号表示比较直观，但内核的实现相当复杂。内核必须：

- 记住每个进程阻塞哪些信号。
- 当从内核态切换到用户态时，对任何一个进程都要检查是否有一个信号已到达。这几乎在每个定时中断时都发生（大约每毫秒发生一次）。
- 确定是否可以忽略信号。这发生在下列所有的条件都满足时：
 - 目标进程没有被另一个进程跟踪（进程描述符中 `ptrace` 字段的 `PT_PTRACED` 标志等于 0）（注 1）。
 - 信号没有被目标进程阻塞。
 - 信号被目标进程忽略（或者因为进程已显式地忽略了信号，或者因为进程没有改变信号的缺省操作且这个缺省操作就是“忽略”）。
- 处理这样的信号，即信号可能在进程运行期间的任一时刻请求把进程切换到一个信号处理函数，并在这个函数返回以后恢复原来执行的上下文。

此外，Linux 必须考虑 BSD 和 System V 所采用的不同的信号语义，而且，还必须与相当麻烦的 POSIX 标准相兼容。

传递信号之前所执行的操作

进程以三种方式对一个信号做出应答：

1. 显式地忽略信号。
2. 执行与信号相关的缺省操作（参见表 11-1）。由内核预定义的缺省操作取决于信号的类型，可以是下列类型之一：

Terminate

进程被终止（杀死）。

Dump

进程被终止（杀死），并且，如果可能，创建包含进程执行上下文的核心转储文件；这个文件可以用于调试。

Ignore

信号被忽略。

注 1： 如果一个进程正在被跟踪时接收到一个信号，内核就停止这个进程，并向跟踪进程发送一个 `SIGCHLD` 信号以通知它一下。跟踪进程又可以使用 `SIGCOUNT` 信号重新恢复被跟踪进程的执行。

Stop

进程被停止，即把进程置为 `TASK_STOPPED` 状态（参见第三章的“进程状态”一节）。

Continue

如果进程被停止（`TASK_STOPPED`），就把它置为 `TASK_RUNNING` 状态。

3. 通过调用相应的信号处理函数捕获信号。

注意，被对一个信号的阻塞和忽略是不同的：只要信号被阻塞，它就不被传递；只有在信号解除阻塞后才传递它。而一个被忽略的信号总是被传递，只是没有进一步的操作。

`SIGKILL` 和 `SIGSTOP` 信号不可以被显式地忽略、捕获或阻塞，因此，通常必须执行它们的缺省操作。因此，`SIGKILL` 和 `SIGSTOP` 允许具有适当特权的用户分别终止并停止任何进程（注2），不管程序执行时采取怎样的防御措施。

如果信号的传递会引起内核杀死一个进程，那么这个信号对该进程就是致命的。`SIGKILL` 信号总是致命的；而且，缺省操作为 `Terminate` 的每个信号，以及不被进程捕获的信号对该进程也是致命的。注意，如果一个被进程所捕获的信号，其对应的信号处理函数终止了这个进程，那么这个信号就不是致命的，因为进程自己选择了终止，而不是被内核杀死。

POSIX 信号和多线程应用

POSIX 1003.1 标准对多线程应用的信号处理有一些严格的要求：

- 信号处理程序必须在多线程应用的所有线程之间共享；不过，每个线程必须有自己的挂起信号掩码和阻塞信号掩码。
- POSIX 库函数 `kill()` 和 `sigqueue()`（见稍后“与信号处理相关的系统调用”一节）必须向所有的多线程应用而不是某个特殊的线程发送信号。所有由内核产生的信号同样如此（如：`SIGCHLD`、`SIGINT` 或 `SIGQUIT`）。
- 每个发送给多线程应用的信号仅传送给一个线程，这个线程是由内核在从不会阻塞该信号的线程中随意选择出来的。
- 如果向多线程应用发送了一个致命的信号，那么内核将杀死该应用的所有线程，而不仅仅是杀死接收信号的那个线程。

注2： 有两个例外：不可能给进程0（`swapper`）发送信号，而发送给进程1（`init`）的信号在捕获到它们之前也总被丢弃。因此，进程0永不死亡，而进程1只有当 `init` 程序终止时才死亡。

为了遵循 POSIX 标准，Linux 内核 2.6 把多线程应用实现为一组属于同一个线程组的轻量级进程（参见第三章“进程、轻量级进程和线程”一节）。

在本章中，术语“线程组”指任意一种线程组，包括仅由单一（普通）进程构成的线程组。例如，当我们规定 `kill()` 能够向线程组发送信号时，我们的意思是这个系统调用也能够向普通进程发送信号。我们将使用术语“进程”表示普通进程或轻量级进程，即属于某个线程组的特定成员。

此外，如果一个挂起信号被发送给了某个特定进程，那么这个信号是私有的；如果被发送给了整个线程组，它就是共享的。

与信号相关的数据结构

对系统中的每个进程来说，内核必须跟踪什么信号当前正在挂起或被屏蔽，以及每个线程组是如何处理所有信号的。为了完成这些操作，内核使用几个从处理器描述符可存取的数据结构。最重要的数据结构如图 11-1 所示。

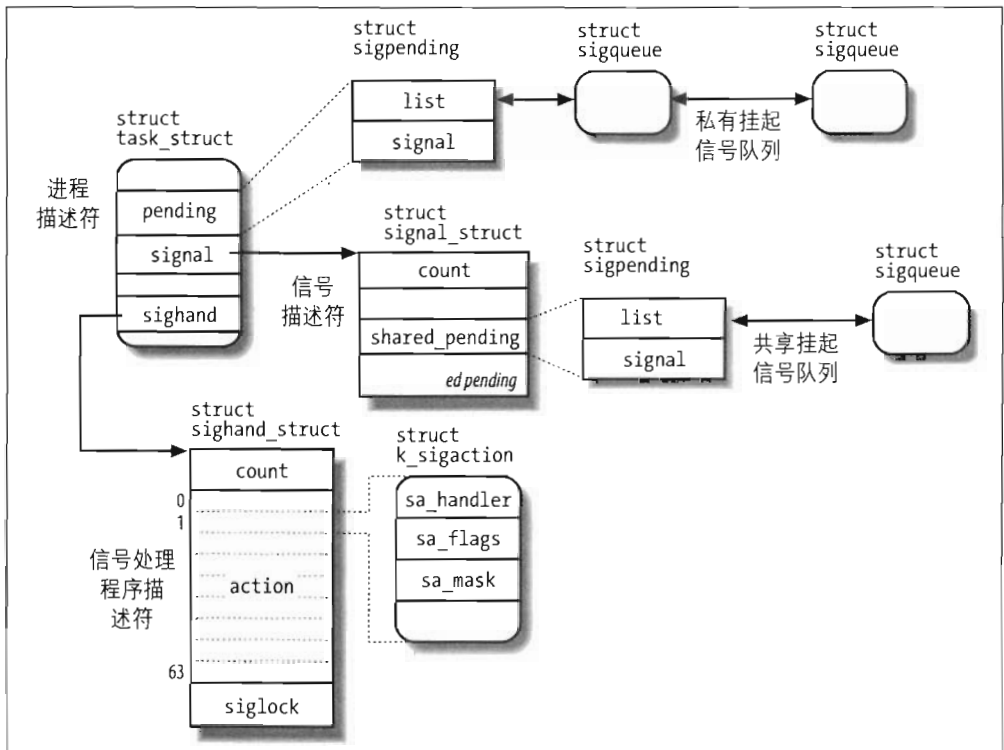


图 11-1：与信号处理相关的最重要的数据结构

与信号处理相关的进程描述符中的字段如表 11-3 所示：

表 11-3：与信号处理相关的进程描述符中的字段

类型	名称	说明
struct signal_struct *	signal	指向进程的信号描述符的指针
struct sighand_struct *	sighand	指向进程的信号处理程序描述符的指针
sigset_t	blocked	被阻塞信号的掩码
sigset_t	real_blocked	被阻塞信号的临时掩码（由 rt_sigtimedwait() 系统调用使用）
struct sigpending	pending	存放私有挂起信号的数据结构
unsigned long	sas_ss_sp	信号处理程序备用堆栈的地址
size_t	sas_ss_size	信号处理程序备用堆栈的大小
int (*) (void *)	notifier	指向一个函数的指针，设备驱动程序用这个函数阻塞进程的某些信号
void *	notifier_data	指向 notifier 函数（表中的前一个字段）可能使用的数据
sigset_t *	notifier_mask	设备驱动程序通过 notifier 函数所阻塞的信号的位掩码

blocked 字段存放进程当前所屏蔽的信号。它是一个 sigset_t 位数组，每种信号类型对应一个元素：

```
typedef struct {
    unsigned long sig[2];
} sigset_t;
```

因为每个无符号长整数由 32 位组成，所以在 Linux 中可以声明的信号最大数是 64（_NSIG 宏表示这个值）。没有值为 0 的信号，因此，信号的编号对应于 sigset_t 类型变量中的相应位下标加 1。1~31 之间的编号对应于表 11-1 所列出的信号，而 32~64 之间的编号对应于实时信号。

信号描述符和信号处理程序描述符

进程描述符的 signal 字段指向信号描述符（*signal descriptor*）—— 一个 signal_struct 类型的结构，用来跟踪共享挂起信号。实际上，信号描述符还包括与信号处理关系并不密切的一些字段，如：每进程的资源限制数组 rlim（见第三章“进程资源限制”一节）、分别用于存放进程的组领头进程和会话领头进程 PID 的字段 pgrp 和 session

(见第三章“进程间的关系”一节)。实际上，就像在第三章“clone()、fork()及vfork()系统调用”一节所提到的，信号描述符被属于同一线程组的所有进程共享，也就是被调用clone()系统调用(CLONE_THREAD标志置位)创建的所有进程共享，因此，对属于同一线程组的每个进程而言，信号描述符中的字段必须都是相同的。

信号描述符中与信号处理有关的字段如表 11-4 所示：

表 11-4：信号描述符中与信号处理有关的字段

类型	名称	说明
atomic_t	count	信号描述符的使用计数器
atomic_t	live	线程组中活动进程的数量
wait_queue_head_t	wait_chldexit	在系统调用wait4()中睡眠的进程的等待队列
struct task_struct *	curr_target	接收信号的线程组中最后一个进程的描述符
struct sigpending	shared_pending	存放共享挂起信号的数据结构
int	group_exit_code	线程组的进程终止代码
struct task_struct *	group_exit_task	在杀死整个线程组的时候使用
int	notify_count	在杀死整个线程组的时候使用
int	group_stop_count	在停止整个线程组的时候使用
unsigned int	flags	在传递修改进程状态的信号时使用的标志

除了信号描述符以外，每个进程还引用一个信号处理程序描述符 (*signal handler descriptor*)，它是一个 `sigand_struct` 类型的结构，用来描述每个信号必须怎样被线程组处理。它的字段在表 11-5 中说明。

表 11-5：信号处理程序描述符的字段

类型	名称	说明
atomic_t	count	信号处理程序描述符的使用计数器
struct k_sigaction [64]	action	说明在所传递信号上执行操作的结构数组
spinlock_t	siglock	保护信号描述符和信号处理程序描述符的自旋锁

正如在第三章的“clone()、fork()及vfork()系统调用”一节中所提到的，在调用clone()系统调用时设置CLONE_SIGHAND标志，信号处理程序描述符就可以由几个进程共享。

描述符的 `count` 字段表示共享该结构的进程个数。在一个 POSIX 的多线程应用中，线程组中的所有轻量级进程都引用相同的信号描述符和信号处理程序描述符。

sigaction 数据结构

一些体系结构把特性赋给仅对内核可见的信号。因此，信号的特性存放在 `k_sigaction` 结构中，`k_sigaction` 结构既包含对用户态进程所隐藏的特性，也包含大家熟悉的 `sigaction` 结构，该结构保存了用户态进程能看见的所有特性。实际上，在 80x86 平台上，信号的所有特性对用户态的进程都是可见的。因此，`k_sigaction` 结构只不过简化为类型为 `sigaction` 的单个 `sa` 结构，该结构包含下列字段（注 3）。

`sa_handler`

这个字段指定要执行操作的类型。它的值可以是指向信号处理程序的一个指针，`SIG_DFL`（即值 0，指定执行缺省操作），或者 `SIG_IGN`（即值 1，指定忽略信号）。

`sa_flags`

这是一个标志集，指定必须怎样处理信号。其中的一些标志在表 11-6 中列出（注 4）。

`sa_mask`

这是类型为 `sigset_t` 的变量，指定当运行信号处理程序时要屏蔽的信号。

表 11-6：指定如何处理信号的一组标志

标志的名称	说明
<code>SA_NOCLDSTOP</code>	仅应用于 <code>SIGCHLD</code> ，当进程被停止时不向父进程发送 <code>SIGCHLD</code> 信号
<code>SA_NOCLDWAIT</code>	仅应用于 <code>SIGCHLD</code> ，当进程终止时不创建僵死状态
<code>SA_SIGINFO</code>	为信号处理程序提供附加信息（参见后面“改变信号的操作”一节）
<code>SA_ONSTACK</code>	为信号处理程序的执行使用一个备用栈（参见后面“捕获信号”一节）
<code>SA_RESTART</code>	自动地重新开始执行被中断的系统调用（参见后面“系统调用的重新执行”一节）

注 3： 用户态应用程序所使用的 `sigaction` 结构向 `signal()` 和 `sigaction()` 系统调用传递参数，它与内核所使用的结构稍有不同，虽然本质上它们存放相同的信息。

注 4： 由于历史原因，这些标志与 `irqaction` 描述符的标志一样都有相同的前缀“`SA_`”（参见第四章的表 4-7），不过这两种标志集合之间没有关系。

表 11-6: 指定如何处理信号的一组标志 (续)

标志的名称	说明
SA_NODEFER, SA_NOMASK	当执行信号处理程序时不屏蔽信号
SA_RESETHAND,	执行信号处理程序后重新设置缺省操作
SA_ONESHOT	

挂起信号队列

如我们在本章前面的表 11-2 中所见到的, 有几个系统调用能产生发送给整个线程组的信号, 如 `kill()` 和 `rt_sigqueueinfo()`, 而其他的一些则产生发送给特定进程的信号, 如 `tkill()` 和 `tgkill()`。

因此, 为了跟踪当前的挂起信号是什么, 内核把两个挂起信号队列与每个进程相关联:

- 共享挂起信号队列, 它位于信号描述符的 `shared_pending` 字段, 存放整个线程组的挂起信号。
- 私有挂起信号队列, 它位于进程描述符的 `pending` 字段, 存放特定进程 (轻量级进程) 的挂起信号。

挂起信号队列由 `sigpending` 数据结构组成, 它的定义如下:

```
struct sigpending {
    struct list_head list;
    sigset_t signal;
}
```

`signal` 字段是指定挂起信号的位掩码, 而 `list` 字段是包含 `sigqueue` 数据结构的双向链表的头, `sigqueue` 的字段如表 11-7 所示。

表 11-7: `sigqueue` 数据结构的字段

类型	名称	说明
<code>struct list_head</code>	<code>list</code>	链接挂起信号队列的链表
<code>spinlock_t *</code>	<code>lock</code>	指向与挂起信号相应的信号处理程序描述符中 <code>siglock</code> 字段的指针
<code>int</code>	<code>flags</code>	<code>Sigqueue</code> 数据结构的标志
<code>siginfo_t</code>	<code>info</code>	描述产生信号的事件
<code>struct user_struct *</code>	<code>user</code>	指向进程拥有者的每用户数据结构的指针 (见第三章 “ <code>clone()</code> 、 <code>fork()</code> 及 <code>vfork()</code> 系统调用” 一节)

`siginfo_t` 是一个 128 字节的数据结构，其中存放有关出现特定信号的信息。它包含下列字段：

`si_signo`

信号编号。

`si_errno`

引起信号产生的指令的出错码，或者如果没有错误则为 0。

`si_code`

发送信号者的代码（参见表 11-8）。

表 11-8：最重要的信号发送者代码

代码名	发送者
SI_USER	<code>kill()</code> 和 <code>raise()</code> (参见后面“与信号处理相关的系统调用”一节)
SI_KERNEL	一般内核函数
SI_QUEUE	<code>sigqueue()</code> (参见后面“与信号处理相关的系统调用”一节)
SI_TIMER	定时器到期
SI_ASYNCIO	异步 I/O 完成
SI_TKILL	<code>tkill()</code> 和 <code>tgkill()</code> (参见后面“与信号处理相关的系统调用”一节)

`_sifields`

存放依赖于信号类型的信息的联合体。例如，相对于 SIGKILL 信号的出现，`siginfo_t` 数据结构在这里记录发送者进程的 PID 和 UID；相反，相对于 SIGSEGV 信号的出现，该数据结构存放某个内存地址，对该地址的访问导致信号产生。

在信号数据结构上的操作

内核使用几个函数和宏来处理信号。在下面的描述中，`set` 是指向 `sigset_t` 类型变量的一个指针，`nsig` 是信号的编号，`mask` 是无符号长整数的位掩码。

`sigemptyset(set)` 和 `sigfillset(set)`

把 `sigset_t` 类型的变量中的位分别置为 0 或 1。

`sigaddset(set, nsig)` 和 `sigdelset(set, nsig)`

把 `nsig` 信号在 `sigset_t` 类型变量中对应的位分别置为 1 或 0。实际上，`sigaddset()` 简化为：

```
set->sig[(nsig - 1) / 32] |= 1UL << ((nsig - 1) % 32);
```

并且把 `sigdelset()` 简化为:

```
set->sig[(nsig - 1) / 32] &= ~(1UL << ((nsig - 1) % 32));
```

`sigaddsetmask(set,mask)` 和 `sigdelsetmask(set,mask)`

把 `mask` 中的位在 `sigset_t` 类型变量中对应的所有位分别设置为 1 或 0。它们仅用于编号为 1~32 之间的信号。对应的函数简化为:

```
set->sig[0] |= mask;
```

和

```
set->sig[0] &= ~mask;
```

`sigismember(set,nsig)`

返回 `nsig` 信号在 `sigset_t` 类型变量中对应位的值。实际上, 这个函数简化为:

```
return 1 & (set->sig[(nsig - 1) / 32] >> ((nsig - 1) % 32));
```

`sigmask(nsig)`

产生 `nsig` 信号的位索引。换句话说, 如果内核需要设置、清除或测试一个特定信号在 `sigset_t` 类型变量中对应的位, 那么就能通过这个宏得到合适的位。

`sigandsets(d,s1,s2)`、`sigorsets(d,s1,s2)` 和 `sigandsets(d,s1,s2)`

在 `sigset_t` 类型的 `s1` 和 `s2` 变量之间分别执行逻辑“与”、逻辑“或”及逻辑“与非”。其结果保存在 `d` 指向的 `sigset_t` 类型的变量中。

`sigtestsetmask(set,mask)`

如果 `mask` 在 `sigset_t` 类型的变量中对应的任何一位被设置, 就返回值 1; 否则返回 0。这只用于编号为 1~32 的信号。

`siginitset(set,mask)`

用 `mask` 中的位初始化 1~32 之间的信号在 `sigset_t` 类型的变量中对应的低位, 并把 33~63 之间信号的对应位清 0。

`siginitsetinv(set,mask)`

用 `mask` 中位的补码初始化 1~32 之间的信号在 `sigset_t` 类型的变量中对应的低位, 并把 33~63 之间信号的对应位置位。

`signal_pending(p)`

如果 `*p` 进程描述符所表示的进程有非阻塞的挂起信号, 就返回值 1 (真), 否则返回 0 (假)。该函数只是通过检查进程的 `TIF_SIGPENDING` 标志就可实现。

`recalc_sigpending_tsk(t)` 和 `recalc_sigpending()`

第一个函数检查是 `*t` 进程描述符所表示的进程有挂起的信号 (通过检查 `t->pending->signa` (字段)), 还是进程所属的线程组有挂起的信号 (通过检查 `t-`

>signal->shared_pending->signal字段)。然后函数把t->thread_info->flags 的 TIF_SIGPENDING 标志置位。函数 recalc_sigpending() 等价于 recalc_sigpending_tsk(current)。

rm_from_queue(mask,q)

从挂起信号队列 q 中删除与 mask 位掩码相对应的挂起信号。

flush_sigqueue(q)

从挂起信号队列 q 中删除所有的挂起信号。

flush_signals(t)

删除发送给 * t 进程描述符所表示的进程的所有信号。这是通过清除 t -> thread_info->flags 中的 TIF_SIGPENDING 标志，并在 t->pending 和 t-> signal->shared_pending 队列上两次调用 flush_sigqueue() 函数来实现的。

产生信号

很多内核函数都会产生信号：它们完成信号处理第一步的工作（在前面“信号的作用”一节中已经描述），即根据需要更新一个或多个进程的描述符。它们不直接执行第二步的信号传递操作，而是可能根据信号的类型和目标进程的状态唤醒一些进程，并促使这些进程接收信号。

当发送给进程一个信号时，这个信号可能来自内核，也可能来自另一个进程。内核通过对如表 11-9 所示的某个函数进行调用而产生信号。

表 11-9：为进程产生信号的内核函数

函数名	说明
send_sig()	向单一进程发送信号
send_sig_info()	与 send_sig() 类似，只是还使用 siginfo_t 结构中的扩展信息
force_sig()	发送既不能被进程显式忽略，也不能被进程阻塞的信号
force_sig_info()	与 force_sig() 类似，只是还使用 siginfo_t 结构中的扩展信息
force_sig_specific()	与 force_sig() 类似，但优化了对 SIGSTOP 和 SIGKILL 信号的处理
sys_tkill()	tkill() 的系统调用处理函数（参见后面“与信号处理相关的系统调用”一节）
sys_tgkill()	tgkill() 的系统调用处理函数

表11-9中的所有函数在结束时都调用`specific_send_sig_info()`函数,下一节将对其进行描述。

当一个信号被发往整个线程组时,这个信号可能来自内核,也可能来自另外一个进程。内核对如表11-10所示的某个函数进行调用产生这类信号。

表 11-10: 为线程组产生信号的内核函数

函数名	说明
<code>send_group_sig_info()</code>	向某一个线程组发送信号,该线程组由它的一个成员进程的描述符来标识
<code>kill_pg()</code>	向一个进程组中的所有线程组发送信号(参见第一章“进程管理”一节)
<code>kill_pg_info()</code>	与 <code>kill_pg()</code> 类似,只是还使用 <code>siginfo_t</code> 结构中的扩展信息
<code>kill_proc()</code>	向某一个线程组发送信号,该线程组由它的一个成员进程的PID来标识
<code>kill_proc_info()</code>	与 <code>kill_proc()</code> 类似,只是还使用 <code>siginfo_t</code> 结构中的扩展信息
<code>sys_kill()</code>	<code>kill()</code> 的系统调用处理函数(参见后面“与信号处理相关的系统调用”一节)
<code>sys_rt_sigqueueinfo()</code>	<code>rt_sigqueueinfo()</code> 的系统调用处理函数

表11-10中的所有函数在结束时都调用`group_send_sig_info()`函数,将在后面的“`group-send-sig-info()`函数”一节中对其进行描述。

specific_send_sig_info()函数

`specific_send_sig_info()`函数向指定进程发送信号,它作用于三个参数:

`sig`

信号编号。

`info`

或者是`siginfo_t`表的地址,或者是三个特殊值中的一个:0意味着信号是由用户态进程发送的,1意味着是由内核发送的,2意味着是由内核发送的SIGSTOP或SIGKILL信号。

`t`

指向目标进程描述符的指针。

必须在关本地中断和已经获得 `t->sigband->siglock` 自旋锁的情况下调用 `specific_send_sig_info()` 函数。函数执行下面的步骤：

1. 检查进程是否忽略信号，如果是就返回0（不产生信号）。当下面的三个忽略信号的条件全部满足时，信号就被忽略：
 - 进程没有被跟踪（`t->ptrace` 中的 `PT_PTRACED` 标志被清0）
 - 信号没有被阻塞（`sigismember(&t->blocked, sig)` 返回0）
 - 或者显式地忽略信号（`t->sigband->action[sig-1]` 的 `sa_handler` 字段等于 `SIG_IGN`），或者隐舍地忽略信号（`sa_handler` 字段等于 `SIG_DFL` 而且信号是 `SIGCONT`、`SIGCHLD`、`SIGWINCH` 或 `SIGURG`）
2. 检查信号是否是实时的（`sig<32`），而且是否在进程的私有挂起信号队列上已经有另外一个相同的挂起信号（`sigismember(&t->pending.signal, sig)` 返回1），如果是，就什么都不需要做，因此返回0。
3. 调用 `send_signal(sig, info, t, &t->pending)` 把信号添加到进程的挂起信号集合中，在下一节将详细描述这个函数。
4. 如果 `send_signal()` 成功地结束，而且信号不被阻塞（`sigismember(&t->blocked, sig)` 返回0），就调用 `signal_wake_up()` 函数通知进程有新的挂起信号。随后，该函数执行下述步骤：
 - a. 把 `t->thread_info->flags` 中的 `TIF_SIGPENDING` 标志置位。
 - b. 如果进程处于 `TASK_INTERRUPTIBLE` 或 `TASK_STOPPED` 状态，而且信号是 `SIGKILL`，就调用 `try_to_wake_up()`（参见第七章“`try_to_wake_up()` 函数”一节）唤醒进程。
 - c. 如果 `try_to_wake_up()` 返回0，那么说明进程已经是可运行的：这种情况下，它检查进程是否已经在另外一个 CPU 上运行，如果是就向那个 CPU 发送一个处理器间中断，以强制当前进程的重新调度（参见第四章“处理器间中断的处理”一节）。因为在从调度函数返回时，每个进程都检查是否存在挂起信号，因此，处理器间中断保证了目标进程能很快注意到新的挂起信号。
5. 返回1（已经成功地产生信号）。

send_signal() 函数

`send_signal()` 函数在挂起信号队列中插入一个新元素，它接收信号编号 `sig`、`siginfo_t` 数据结构的地址 `info`（或一个特殊编码，见上一节对 `specific_send_sig_info()` 的描述）、目标进程描述符的地址 `t` 以及挂起信号队列的地址 `signals` 作为它的参数。

函数执行下面的步骤：

1. 如果 info 的值是 2，这个信号就是 SIGKILL 或 SIGSTOP，而且已经由内核通过 force_sig_specific() 函数产生：在这种情况下，函数跳转到第 9 步，内核立即强制执行与这些信号相关的操作，因此函数不用把信号添加到挂起信号队列中。
2. 如果进程拥有者的挂起信号的数量 (t->user->sigpending) 小于当前进程的资源限制 (t->signal->rlim[RLIMIT_SIGPENDING].rlim_cur)，函数就为新出现的信号分配 sigqueue 数据结构。

```
q = kmem_cache_alloc(sigqueue_cachep, GFP_ATOMIC);
```

3. 如果进程拥有者的挂起信号的数量太多，或者上一步的内存分配失败，就跳转到第 9 步。
4. 递增拥有者挂起信号的数量 (t->user->sigpending) 和 t->user 所指向的每用户数据结构的引用计数器。
5. 在挂起信号队列 signals 中增加 sigqueue 数据结构。

```
list_add_tail(&q->list, &signals->list);
```

6. 在 sigqueue 数据结构中填充表 siginfo_t。

```
if ((unsigned long)info == 0) {
    q->info.si_signo = sig;
    q->info.si_errno = 0;
    q->info.si_code = SI_USER;
    q->info._sifields._kill._pid = current->pid;
    q->info._sifields._kill._uid = current->uid;
} else if ((unsigned long)info == 1) {
    q->info.si_signo = sig;
    q->info.si_errno = 0;
    q->info.si_code = SI_KERNEL;
    q->info._sifields._kill._pid = 0;
    q->info._sifields._kill._uid = 0;
} else
    copy_siginfo(&q->info, info);
```

copy_siginfo() 函数复制由调用者传递的 siginfo_t 表。

7. 把队列位掩码中与信号相应的位置 1：

```
sigaddset(&signals->signal, sig);
```

8. 返回 0：说明信号已被成功地追加到挂起信号队列中。
9. 此时，不再向信号挂起队列中增加元素，因为已经有太多的挂起信号，或已经没有可以分给 sigqueue 数据结构的空闲空间，或者信号已经由内核强制立即发送。如果信号是实时的，并已经通过内核函数发送给队列排队，则 send_signal() 函数返回错误代码 -EAGAIN：

```
if (sig>=32 && info && (unsigned long) info != 1 &&
    info->si_code != SI_USER)
    return -EAGAIN;
```

10. 设置队列的位掩码中与信号相关的位:

```
sigaddset(&signals->signal, sig);
```

11. 返回0: 即使信号没有被追加到队列中, 挂起信号掩码中相应的位也被设置。

即使在挂起队列中没有空间存放相应的挂起信号, 让目标进程能接收信号也是至关重要的。假设一个进程正在消耗过多内存的情形。内核必须保证即使没有空闲内存, `kill()` 系统调用也能够成功执行, 否则, 系统管理员就没有机会通过终止有害进程来恢复系统。

group_send_sig_info()函数

`group_send_sig_info()` 函数向整个线程组发送信号。它作用于三个参数: 信号编号 `sig`, `siginfo_t` 表的地址 `info` (可选的值为0、1或2, 如前面 “`specific_send_sig_info()`” 函数“一节中所描述的) 以及进程描述符的地址 `p`。

该函数主要执行下面的步骤:

1. 检查参数 `sig` 是否正确:

```
if (sig < 0 || sig > 64)
    return -EINVAL;
```

2. 如果信号是由用户态进程发送的, 则该函数确定是否允许这个操作。下列条件中至少有一个成立时信号才能被传递:

- 发送进程的拥有者具有适当的权能 (这通常意味着通过系统管理员发布信号, 参见第二十章)。
- 信号为 `SIGCONT` 且目标进程与发送进程处于同一个注册会话中。
- 两个进程属于同一个用户。

如果不允许用户态进程发送信号, 函数就返回值 `-EPERM`。

3. 如果参数 `sig` 的值为0, 则函数不产生任何信号, 立即返回:

```
if (!sig || !p->sigband)
    return 0;
```

因为0是无效的信号编码, 用于让发送进程检查它是否有向目标线程组发送信号所必需的特权。如果目标进程正在被杀死 (通过检查它的信号处理程序描述符是否已经被释放来获知), 那么函数也返回。

4. 获取 `p->sigband->siglock` 自旋锁并关闭本地中断。
5. 调用 `handle_stop_signal()` 函数, 该函数检查信号的某些类型, 这些类型可能使目标线程组的其他挂起信号无效。 `handle_stop_signal()` 函数执行下面的步骤:
 - a. 如果线程组正在被杀死 (信号描述符的 `flags` 字段的 `SIGNAL_GROUP_EXIT` 标志被设置), 则函数返回。
 - b. 如果 `sig` 是 `SIGSTOP`、`SIGTSTP`、`SIGTTIN` 或 `SIGTTOU` 信号, 就调用 `rm_from_queue()` 函数从共享挂起信号队列 `p->signal->shared_pending` 和线程组所有成员的私有信号队列中删除 `SIGCONT` 信号。
 - c. 如果 `sig` 是 `SIGCONT` 信号, 就调用 `rm_from_queue()` 函数从共享挂起信号队列 `p->signal->shared_pending` 中删除所有的 `SIGSTOP`、`SIGTSTP`、`SIGTTIN` 和 `SIGTTOU` 信号, 然后从属于线程组的进程的私有挂起信号队列中删除上述信号, 并唤醒进程:

```

rm_from_queue(0x003c0000, &p->signal->shared_pending);
t = p;
do {
    rm_from_queue(0x003c0000, &t->pending);
    try_to_wake_up(t, TASK_STOPPED, 0);
    t = next_thread(t);
} while (t != p);

```

掩码 `0x003c0000` 选择以上四种停止信号。宏 `next_thread` 每次循环都返回线程组中不同轻量级进程的描述符地址 (见第三章“进程间的关系”一节) (注 5)。

6. 检查线程组是否忽略信号, 如果是就返回 0 值 (成功)。如果在前面“信号的作用”一节中所提到的忽略信号的三个条件都满足 (也可参见前面“`specific-send-sig.info()` 函数”一节中的第 1 步), 就忽略信号。
7. 检查信号是否是非实时的, 并且在线程组的共享挂起信号队列中已经有另外一个相同的信号, 如果是, 就什么都不需要做, 因此返回 0 值 (成功)。


```

if (sig < 32 && sigismember(&p->signal->shared_pending.signal, sig))
    return 0;

```
8. 调用 `send_signal()` 函数把信号添加到共享挂起信号队列中 (参见前面“`send_signal()` 函数”一节)。如果 `send_signal()` 返回非 0 的错误代码, 则函数终止并返回相同的值。

注 5: 实际代码比这里所给出的代码片段要复杂得多, 因为 `handle_stop_signal()` 还要考虑 `SIGCONT` 信号被捕获的特殊情况, 以及当线程组的所有进程都正在被停止时由 `SIGCONT` 信号引起竞争条件的特殊情况。

9. 调用 `__group_complete_signal()` 函数唤醒线程组中的一个轻量级进程（见下面）。
10. 释放 `p->sigband->siglock` 自旋锁并打开本地中断。
11. 返回 0（成功）。

函数 `__group_complete_signal()` 扫描线程组中的进程，查找能接收新信号的进程。满足下述所有条件的进程可能被选中：

- 进程不阻塞信号。
- 进程的状态不是 `EXIT_ZOMBIE`、`EXIT_DEAD`、`TASK_TRACED` 或 `TASK_STOPPED`（作为一种异常情况，如果信号是 `SIGKILL`，那么进程可能处于 `TASK_TRACED` 或者 `TASK_STOPPED` 状态）。
- 进程没有正在被杀死，即它的 `PF_EXITING` 标志没有置位。
- 进程或者当前正在 CPU 上运行，或者它的 `TIF_SIGPENDING` 标志还没有设置。（实际上，唤醒一个有挂起信号的进程是毫无意义的：通常，唤醒操作已经由设置了 `TIF_SIGPENDING` 标志的内核控制路径执行；另一方面，如果进程正在执行，则应该向它通报有新的挂起信号。）

一个线程组可能有很多满足上述条件的进程，函数按照下面的规则选择其中的一个进程：

- 如果 `p` 标识的进程（由 `group_send_sig_info()` 的参数传递的描述符地址）满足所有的优先准则，并因此而能接收信号，函数就选择该进程。
- 否则，函数通过扫描线程组的成员搜索一个适当的进程，搜索从接收线程组最后一个信号的进程（`p->signal->curr_target`）开始。

如果函数 `__group_complete_signal()` 成功地找到一个适当的进程，就开始向被选中的进程传递信号。首先，函数检查信号是否是致命的，如果是，通过向线程组中的所有轻量级进程发送 `SIGKILL` 信号杀死整个线程组。否则，函数调用 `signal_wake_up()` 函数通知被选中的进程：有新的挂起信号到来（见前面“`specific_send_sig_info()` 函数”一节的第 4 步）。

传递信号

我们假定内核已注意到一个信号的到来，并调用前面所介绍的函数为接收此信号的进程准备描述符。但万一这个进程在那一刻并不在 CPU 上运行，内核就延迟传递信号的任务。我们现在转向另一个主题，即为确保进程的挂起信号得到处理内核所执行的操作。

我们在第四章“从中断和异常返回”一节中提到，内核在允许进程恢复用户态下的执行之前，检查进程 `TIF_SIGPENDING` 标志的值。每当内核处理完一个中断或异常时，就检查是否存在挂起信号。

为了处理非阻塞的挂起信号，内核调用 `do_signal()` 函数，它接收两个参数：

`regs`

栈区的地址，当前进程在用户态下寄存器的内容存放在这个栈中。

`oldset`

变量的地址，假设函数把阻塞信号的位掩码数组存放在这个变量中。如果没有必要保存位掩码数组，则它为 `NULL`。

对于 `do_signal()` 函数，我们将重点说明信号传递的一般机制。它的实现代码很累赘，这是因为要对竞争条件和其他特殊情况（如：冻结系统、产生内存信息转储、停止和杀死整个线程组等等）进行详细处理，因此我们将悄然地略过这些细节。

就像已经提到过的，通常只是在 CPU 要返回到用户态时才调用 `do_signal()` 函数。因此，如果中断处理程序调用 `do_signal()`，则该函数立刻返回：

```
if ((regs->xcs & 3) != 3)
    return 1;
```

如果 `oldset` 参数为 `NULL`，函数就用 `current->blocked` 字段的地址对它初始化：

```
if (!oldset)
    oldset = &current->blocked;
```

`do_signal()` 函数的核心由重复调用 `dequeue_signal()` 函数的循环组成，直到在私有挂起信号队列和共享挂起信号队列中都没有非阻塞的挂起信号时，循环才结束。`dequeue_signal()` 的返回码存放在 `signr` 局部变量中。如果值为 0，意味着所有挂起的信号已全部被处理，并且 `do_signal()` 可以结束。只要返回一个非 0 值，就意味着挂起的信号正等待被处理，并且 `do_signal()` 处理了当前信号后又调用了 `dequeue_signal()`。

`dequeue_signal()` 函数首先考虑私有挂起信号队列中的所有信号，并从最低编号的挂起信号开始。然后考虑共享队列中的信号。它更新数据结构以表示信号不再是挂起的，并返回它的编号。这就涉及清 `current->pending.signal` 或 `current->signal->shared_pending.signal` 中对应的位，并调用 `recalc_sigpending()` 更新 `TIF_SIGPENDING` 标志的值。

让我们来看 `do_signal()` 函数如何处理每一个挂起的信号，其编号由 `dequeue_signal()` 返回。首先，它检查 `current` 接收进程是否正受到其他一些进程的监控，在肯定的情况

下, `do_signal()`调用 `do_notify_parent_cldstop()`和 `schedule()`让监控进程知道进程的信号处理。

然后, `do_signal()`把要处理信号的 `k_sigaction` 数据结构的地址赋给局部变量 `ka`:

```
ka = &current->sig->action[signr-1];
```

根据 `ka` 的内容可以执行三种操作: 忽略信号、执行缺省操作或执行信号处理程序。

如果显式忽略被传递的信号, 那么 `do_signal()` 函数仅仅继续执行循环, 并由此考虑另一个挂起信号:

```
if (ka->sa.sa_handler == SIG_IGN)
    continue;
```

在下面两节, 我们将说明如何执行缺省操作和信号处理程序。

执行信号的缺省操作

如果 `ka->sa.sa_handler` 等于 `SIG_DFL`, `do_signal()` 就必须执行信号的缺省操作。唯一的例外是当接收进程是 `init` 时, 在这种情况下, 正如前面“传递信号之前所执行的操作”一节中所描述的那样, 这个信号被丢弃:

```
if (current->pid == 1)
    continue;
```

如果接收进程是其他进程, 对缺省操作是 `Ignore` 的信号进行处理也很简单:

```
if (signr==SIGCONT || signr==SIGCHLD ||
    signr==SIGWINCH || signr==SIGURG)
    continue;
```

缺省操作是 `Stop` 的信号可能停止线程组中的所有进程。为此, `do_signal()` 把进程的状态都置为 `TASK_STOPPED`, 并在随后调用 `schedule()` 函数 (参见第七章“`schedule()` 函数”一节)。

```
if (signr==SIGSTOP || signr==SIGTSTP ||
    signr==SIGTTIN || signr==SIGTTOU) {
    if (signr != SIGSTOP &&
        is_orphaned_pgrp(current->signal->pgrp))
        continue;
    do_signal_stop(signr);
}
```

`SIGSTOP` 与其他信号的差异比较微妙: `SIGSTOP` 总是停止线程组, 而其他信号只停止不在“孤儿进程组”中的线程组。POSIX 标准规定, 只要进程组中有一个进程有父进程,

尽管父进程处于不同的进程组中但在同一个会话中,那么这个进程组就不是孤儿。因此,如果父进程死亡,但启动该进程的用户仍登录在线,那么该进程组就不是一个孤儿。

`do_signal_stop()`函数检查 `current` 是否是线程组中第一个被停止的进程,如果是,它激活“组停止”:本质上,该函数把一个正数值赋给信号描述符中的 `group_stop_count` 字段,并唤醒线程组中的所有进程。所有这样的进程都检查该字段以确认正在进行“组停止”操作,然后把进程的状态置为 `TASK_STOPPED`,并调用 `schedule()`。如果线程组领头进程的父进程没有设置 `SIGCHLD` 的 `SA_NOCLDSTOP` 标志,那么 `do_signal_stop()` 函数还要向它发送 `SIGCHLD` 信号。

缺省操作为 `Dump` 的信号可以在进程的工作目录中创建一个“转储”文件,这个文件列出进程地址空间和CPU寄存器的全部内容。`do_signal()` 创建了转储文件后,就杀死这个线程组。剩余 18 个信号的缺省操作是 `Terminate`,它仅仅是杀死线程组。为了杀死整个线程组,函数调用 `do_group_exit()` 执行彻底的“组退出”过程(参见第三章的“进程终止”一节)。

捕获信号

如果信号有一个专门的处理程序,`do_signal()`就函数必须强迫该处理程序执行。这是通过调用 `handle_signal()` 进行的:

```
handle_signal(signr, &info, &ka, oldset, regs);
if (ka->sa.sa_flags & SA_ONESHOT)
    ka->sa.sa_handler = SIG_DFL;
return 1;
```

如果所接收信号的 `SA_ONESHOT` 标志被置位,就必须重新设置它的缺省操作,以便同一信号的再次出现不会再次触发这一信号处理程序的执行。注意 `do_signal()` 在处理了一个单独的信号后怎样返回。直到下一次调用 `do_signal()` 时才考虑其他挂起的信号。这种方式确保了实时信号将以适当的顺序得到处理。

执行一个信号处理程序是件相当复杂的任务,因为在用户态和内核态之间切换时需要谨慎地处理栈中的内容。我们将正确地解释这里所承担的任务。

信号处理程序是用户态进程所定义的函数,并包含在用户态的代码段中。`handle_signal()` 函数运行在内核态,而信号处理程序运行在用户态,这就意味着在当前进程恢复“正常”执行之前,它必须首先执行用户态的信号处理程序。此外,当内核打算恢复进程的正常执行时,内核态堆栈不再包含被中断程序的硬件上下文,因为每当从内核态向用户态转换时,内核态堆栈都被清空。

而另外一个复杂性是因为信号处理程序可以调用系统调用,在这种情况下,执行了系统调用的服务例程以后,控制权必须返回到信号处理程序而不是到被中断程序的正常代码流。

Linux所采用的解决方法是把保存在内核态堆栈中的硬件上下文拷贝到当前进程的用户态堆栈中。用户态堆栈也以这样的方式被修改,即当信号处理程序终止时,自动调用 `sigreturn()` 系统调用把这个硬件上下文拷贝回到内核态堆栈中,并恢复用户态堆栈中原来的内容。

图 11-2 说明了有关捕获一个信号的函数的执行流。一个非阻塞的信号发送给一个进程。当中断或异常发生时,进程切换到内核态。正要返回到用户态前,内核执行 `do_signal()` 函数,这个函数又依次处理信号(通过调用 `handle_signal()`)和建立用户态堆栈(通过调用 `setup_frame()` 或 `setup_rt_frame()`)。当进程又切换到用户态时,因为信号处理程序的起始地址被强制放进程序计数器中,因此开始执行信号处理程序。当处理程序终止时, `setup_frame()` 或 `setup_rt_frame()` 函数放在用户态堆栈中的返回代码就被执行。这个代码调用 `sigreturn()` 或 `rt_sigreturn()` 系统调用,相应的服务例程把正常程序的用户态堆栈硬件上下文拷贝到内核态堆栈,并把用户态堆栈恢复到它原来的状态(通过调用 `restore_sigcontext()`)。当这个系统调用结束时,普通进程就因此能恢复自己的执行。

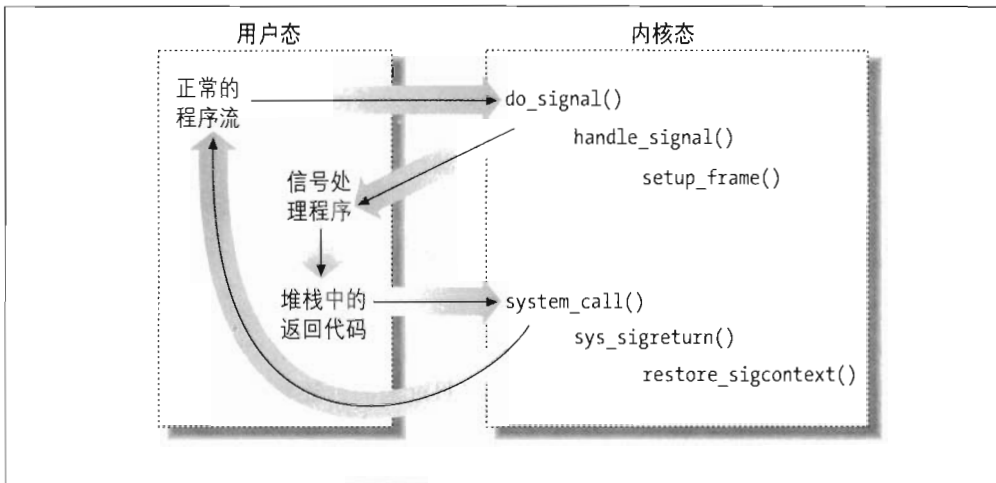


图 11-2: 捕获一个信号

现在我们详细考察如何实施这种方案。

建立帧

为了适当地建立进程的用户态堆栈，`handle_signal()`函数或者调用`setup_frame()`（对不需要`siginfo_t`表的信号；参见本章后面“与信号处理相关的系统调用”一节），或者调用`setup_rt_frame()`（对需要`siginfo_t`表的信号）。为了在这两个函数之间进行选择，内核检查与信号相关的`sigaction`表`sa_flags`字段的`SA_SIGINFO`标志值。

`setup_frame()`函数接收四个参数，它们具有下列含义：

`sig`

信号编号

`ka`

与信号相关的`k_sigaction`表的地址

`oldset`

阻塞信号的位掩码数组的地址

`regs`

用户态寄存器的内容保存在内核态堆栈区的地址

`setup_frame()`函数把一个叫做帧（*frame*）的数据结构推进用户态堆栈中，这个帧含有处理信号所需要的信息，并确保正确返回到`handle_signal()`函数。一个帧就是包含下列字段的`sigframe`表（见图 11-3）：

`precode`

信号处理函数的返回地址，它指向`__kernel_sigreturn`标记处的代码（稍后列出）。

`sig`

信号编号，这是信号处理程序所需的参数。

`sc`

类型为`sigcontext`的结构，它包含正好切换到内核态前用户态进程的硬件上下文（这种信息是从`current`的内核态堆栈中拷贝过来的），还包含进程被阻塞的常规信号的位数组。

`fpstate`

类型为`_fpstate`的结构，可以用来存放用户态进程的浮点寄存器内容（参见第三章的“保存和加载 FPU、MMX 及 XMM 寄存器”一节）。

`extramask`

被阻塞的实时信号的位数组。

retcode

发出 `sigreturn()` 系统调用的 8 字节代码。在 Linux 的早期版本中，这段程序有效地执行从信号处理程序返回的功能，但在 Linux 2.6 中，它仅被当做一个标记 (signature) 来使用，以便调式程序能识别出信号栈帧。

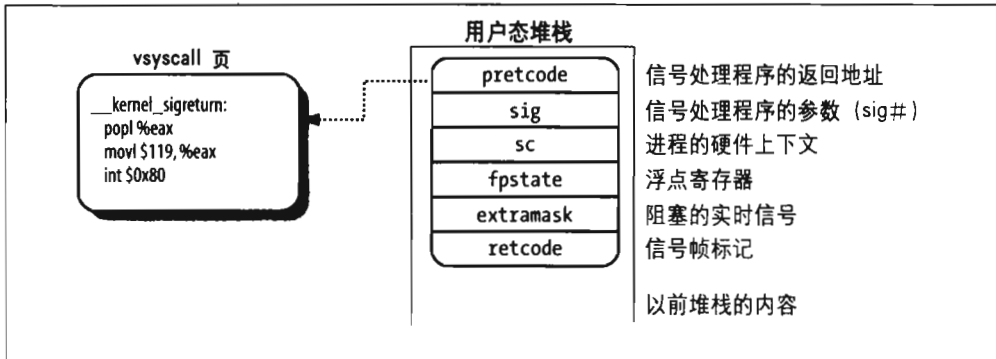


图 11-3: 在用户态堆栈中的帧

`setup_frame()` 函数首先调用 `get_sigframe()` 计算帧的第一个内存单元，这个内存单元通常是在用户态堆栈中 (注 6)，因此函数返回值：

```
(regs->esp - sizeof(struct sigframe)) & 0xfffffff8
```

因为栈朝低地址方向延伸，所以通过把当前栈顶的地址减去它的大小，使其结果与 8 的倍数对齐，就获得了帧的起始地址。

然后用 `access_ok` 宏对返回地址进行验证。如果地址有效，`setup_frame()` 就反复调用 `__put_user()` 填充帧的所有字段。帧的 `pretcode` 字段初始化为 `&__kernel_sigreturn`，一些粘合代码的地址放在 `vsyscall` 页中 (参见第十章“通过 `sysenter` 指令发出系统调用”一节)。

一旦完成了这个操作，就修改内核态堆栈的 `regs` 区，这就保证了当 `current` 恢复它在用户态的执行时，控制权将传递给信号处理程序。

注 6: Linux 允许进程通过调用 `signalstack()` 系统调用来为它们的信号处理程序指定一个预备的栈，这种特点也是 X/Open 标准所要求的。当一个预备的栈存在时，`get_sigframe()` 函数就返回这个栈中的一个地址。我们在此不进一步讨论这种情况，因为它从概念上非常类似于常规信号的处理。

```

regs->esp = (unsigned long) frame;
regs->eip = (unsigned long) ka->sa.sa_handler;
regs->eax = (unsigned long) sig;
regs->edx = regs->ecx = 0;
regs->xds = regs->xes = regs->xss = __USER_DS;
regs->xcs = __USER_CS;

```

setup_frame() 函数把保存在内核态堆栈的段寄存器内容重新设置成它们的缺省值以后才结束。现在，信号处理程序所需的信息就在用户态堆栈的顶部。

setup_rt_frame() 函数与 setup_frame() 非常相似，但它把用户态堆栈存放在一个扩展的帧中（保存在 rt_sigframe 数据结构中），这个帧也包含了与信号相关的 siginfo_t 表的内容。此外，该函数设置 precode 字段以使它指向 vsyscall 页中的 __kernel_rt_sigreturn 代码。

检查信号标志

建立了用户态堆栈以后，handle_signal() 函数检查与信号相关的标志值。如果信号没有设置 SA_NODEFER 标志，在 sigaction 表中 sa_mask 字段对应的信号就必须在信号处理程序执行期间被阻塞：

```

if (!(ka->sa.sa_flags & SA_NODEFER)) {
    spin_lock_irq(&current->sigband->siglock);
    sigorsets(&current->blocked, &current->blocked, &ka->sa.sa_mask);
    sigaddset(&current->blocked, sig);
    recalc_sigpending(current);
    spin_unlock_irq(&current->sigband->siglock);
}

```

如前所述，recalc_sigpending() 函数检查进程是否有非阻塞的挂起信号，并因此而设置它的 TIF_SIGPENDING 标志。

然后，handle_signal() 返回到 do_signal()，do_signal() 也立即返回。

开始执行信号处理程序

do_signal() 返回时，当前进程恢复它在用户态的执行。由于如前所述 setup_frame() 的准备，eip 寄存器指向信号处理程序的第一条指令，而 esp 指向已推进用户态堆栈顶的帧的第一个内存单元。因此，信号处理程序被执行。

终止信号处理程序

信号处理程序结束时，返回栈顶地址，该地址指向帧的 precode 字段所引用的 vsyscall 页中的代码：

```
__kernel_sigreturn:
    popl %eax
    movl $__NR_sigreturn, %eax
    int $0x80
```

因此，信号编号（即帧的 sig 字段）被从栈中丢弃，然后调用 sigreturn() 系统调用。

sys_sigreturn() 函数计算类型为 pt_regs 的数据结构 regs 的地址，其中 pt_regs 包含用户态进程的硬件上下文（参见第十章“参数传递”一节）。从存放在 esp 字段中的值，由此而导出并检查帧在用户态堆栈内的地址：

```
frame = (struct sigframe *) (regs.esp - 8);
if (verify_area(VERIFY_READ, frame, sizeof(*frame)) {
    force_sig(SIGSEGV, current);
    return 0;
}
```

然后，函数把调用信号处理程序前所阻塞的信号的位数组从帧的 sc 字段拷贝到 current 的 blocked 字段。结果，为信号处理函数的执行而屏蔽的所有信号解除阻塞。然后调用 recalc_sigpending() 函数。

此时，sys_sigreturn() 函数必须把来自帧的 sc 字段的进程硬件上下文拷贝到内核态堆栈中，并从用户态堆栈中删除帧，这两个任务是通过调用 restore_sigcontext() 函数完成的。

像 rt_sigqueueinfo() 这样的系统调用需要与信号相关的 siginfo_t 表，如果信号是这种系统调用发送的，则其实现机制非常相似。扩展帧的 pretcode 字段指向 vsyscall 页面中的 __kernel_rt_sigreturn 代码，它依次调用 rt_sigreturn() 系统调用，其相应的 sys_rt_sigreturn() 服务例程把来自扩展帧的进程硬件上下文拷贝到内核态堆栈，并通过从用户态堆栈删除扩展帧以恢复用户态堆栈原来的内容。

系统调用的重新执行

内核并不总是能立即满足系统调用发出的请求，在这种情况下发生时，把发出系统调用的进程置为 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE 状态。

如果进程处于 TASK_INTERRUPTIBLE 状态，并且某个进程向它发送了一个信号，那么，内核不完成系统调用就把进程置成 TASK_RUNNING 状态（参看第四章的“从中断和异常返回”一节）。当切换回用户态时信号被传递给进程。当这种情况发生时，系统调用服务例程没有完成它的工作，但返回 EINTR、ERESTARTNOHAND、ERESTART_RESTARTBLOCK、ERESTARTSYS 或 ERESTARTNOINTR 错误码。

实际上, 这种情况下用户态进程获得的唯一错误码是 `EINTR`, 这个错误码表示系统调用还没有执行完(应用程序的编写者可以测试这个错误码并决定是否重新发出系统调用)。内核内部使用剩余的错误码来指定信号处理程序结束后是否自动重新执行系统调用。

表 11-11 列出了与未完成的系统调用相关的出错码及这些出错码对信号三种可能的操作产生的影响。在表项中出现的几个术语的含义如下:

Terminate

不会自动重新执行系统调用; 进程在 `int $0x80` 或 `sysenter` 指令紧接着的那条指令处将恢复它在用户态的执行, 这时 `eax` 寄存器包含的值为 `-EINTR`。

Reexecute

内核强迫用户态进程把系统调用号重新装入 `eax` 寄存器, 并重新执行 `int $0x80` 指令或 `sysenter` 指令。进程意识不到这种重新执行, 因此出错码也不传递给进程。

Depends

只有被传递信号的 `SA_RESTART` 标志被设置, 才重新执行系统调用; 否则, 系统调用以 `-EINTR` 出错码结束。

表 11-11: 系统调用的重新执行

信号 操作	错误码及其对系统调用执行的影响			
	<code>EINTR</code>	<code>ERESTARTSYS</code>	<code>ERESTART_NOHAND</code> ^a	<code>ERESTARTNOINTR</code>
Default	Terminate	Reexecute	Reexecute	Reexecute
Ignore	Terminate	Reexecute	Reexecute	Reexecute
Catch	Terminate	Depends	Terminate	Reexecute

a. 在允许重新开始执行系统调用的机制中, 错误代码 `ERESTARTNOHAND` 和 `ERESTART_RESTARTBLOCK` 是有所不同的 (见下面)。

当传递信号时, 内核在试图重新执行一个系统调用前必须确定进程确实发出过这个系统调用。这就是 `regs` 硬件上下文的 `orig_eax` 字段起重要作用之处。让我们回顾一下中断或异常处理程序开始时是如何初始化这个字段的:

中断

这个字段包含的值为与中断相关的 `IRQ` 号减去 256 (参看第四章的“为中断处理程序保存寄存器的值”一节)。

0x80 异常 (或者 sysenter)

这个字段包含系统调用号 (参看第十章的“进入和退出系统调用”一节)。

其他异常

这个字段包含的值为-1(参看第四章的“为异常处理程序保存寄存器的值”一节)。

因此, `orig_eax` 字段中的非负数意味着信号已经唤醒了在系统调用上睡眠的 `TASK_INTERRUPTIBLE` 进程。服务例程认识到系统调用曾被中断, 并返回前面提到的某个错误码。

重新执行被未捕获信号中断的系统调用

如果信号被显式地忽略, 或者如果它的缺省操作已被强制执行, `do_signal()` 就分析系统调用的出错码, 并如表 11-11 中所说明的那样决定是否重新自动执行未完成的系统调用。如果必须重新开始执行系统调用, 那么 `do_signal()` 就修改 `regs` 硬件上下文, 以便在进程返回到用户态时, `eip` 指向 `int $0x80` 指令或 `sysenter` 指令, 且 `eax` 包含系统调用号:

```
if (regs->orig_eax >= 0) {
    if (regs->eax == -ERESTARTNOHAND || regs->eax == -ERESTARTSYS ||
        regs->eax == -ERESTARTNOINTR) {
        regs->eax = regs->orig_eax;
        regs->eip -= 2;
    }
    if (regs->eax == -ERESTART_RESTARTBLOCK) {
        regs->eax = __NR_restart_syscall;
        regs->eip -= 2;
    }
}
```

把系统调用服务例程的返回代码赋给 `regs->eax` 字段(参见第十章“进入和退出系统调用”一节)。注意, `int $0x80` 和 `sysreturn` 的长度都是两个字节, 因此该函数从 `eip` 中减去 2, 使 `eip` 指向引起系统调用的指令。

`ERESTART_RESTARTBLOCK` 错误代码是特殊的, 因为 `eax` 寄存器中存放了 `restart_syscall()` 的系统调用号, 因此, 用户态进程不会重新执行被信号中断的同一个系统调用。这个错误代码仅用于与时间相关的系统调用, 当重新执行这些系统调用时, 应该调整它们的用户态参数。一个典型的例子是 `nanosleep()` 系统调用(参见第六章“动态定时器应用之一: `nanosleep()` 系统调用”一节): 假设进程为了暂停执行 20ms 而调用了 `nanosleep()`, 而在 10ms 后出现了一个信号。如果像通常那样重新执行该系统调用(不调整其用户态参数), 那么总的延迟会超过 30ms。

可以采用另一种方式, `nanosleep()` 系统调用的服务例程把重新执行时所使用的特定服务例程的地址赋给 `current` 的 `thread_info` 结构中的 `restart_block` 字段, 并在被中断时返回 `-ERESTART_RESTARTBLOCK`。 `sys_restart_syscall()` 服务例程只执行特

定的nanosleep()的服务例程,考虑到原始系统调用的调用到重新执行之间有时间间隔,该服务例程调整这种延迟。

为所捕获的信号重新执行系统调用

如果信号被捕获,那么handle_signal()分析出错码,也可能分析sigaction表的SA_RESTART标志来决定是否必须重新执行未完成的系统调用:

```

if (regs->orig_eax >= 0) {
    switch (regs->eax) {
        case -ERESTART_RESTARTBLOCK:
        case -ERESTARTNOHAND:
            regs->eax = -EINTR;
            break;
        case -ERESTARTSYS:
            if (!(ka->sa.sa_flags & SA_RESTART)) {
                regs->eax = -EINTR;
                break;
            }
            /* fallthrough */
        case -ERESTARTNOINTR:
            regs->eax = regs->orig_eax;
            regs->eip -= 2;
    }
}

```

如果系统调用必须被重新开始执行,handle_signal()就与do_signal()完全一样地继续执行;否则,它向用户态进程返回一个出错码-EINTR。

与信号处理相关的系统调用

正如本章已提到的,在用户态运行的进程可以发送和接收信号。这意味着必须定义一组系统调用来完成这些操作。遗憾的是,由于历史的原因,已经存在几个具有相同功能的系统调用,因此,其中一些系统调用从未被调用。例如:系统调用sys_sigaction()和sys_rt_sigaction()几乎是相同的,因此C库中封装函数sigaction()调用sys_rt_sigaction()而不是sys_sigaction()。下面几节我们将描述其中一些最重要的系统调用。

kill()系统调用

一般用kill(pid,sig)系统调用向普通进程或多线程应用发送信号,其相应的服务例程是sys_kill()函数。整数参数pid的几个含义取决于它的值:

pid > 0

把sig信号发送到其PID等于pid的进程所属的线程组。

pid = 0

把 `sig` 信号发送到与调用进程同组的进程的所有线程组。

pid = -1

把信号发送到所有进程，除了 `swapper` (PID 0)、`init` (PID 1) 和 `current` 以外。

pid < -1

把信号发送到进程组 `-pid` 中进程的所有线程组。

`sys_kill()` 函数为信号建立最小的 `siginfo_t` 表，然后调用 `kill_something_info()` 函数：

```
info.si_signo = sig;
info.si_errno = 0;
info.si_code = SI_USER;
info._sifields._kill._pid = current->tgid;
info._sifields._kill._uid = current->uid;
return kill_something_info(sig, &info, pid);
```

`kill_something_info` 还依次调用 `kill_proc_info()` (通过 `group_send_sig_info()` 向一个单独的线程组发送信号)，或者调用 `kill_pg_info()` (扫描目标进程组的所有进程，并为目标进程组中的每个进程调用 `send_sig_info()`)，或者为系统中的所有进程反复调用 `group_send_sig_info()` (如果 `pid` 等于 `-1`)。

`kill()` 系统调用能发送任何信号，即使编号在 `32~64` 之间的实时信号。然而，我们在前面“产生信号”一节已看到，`kill()` 系统调用不能确保把一个新的元素加入到目标进程的挂起信号队列，因此，挂起信号的多个实例可能被丢失。实时信号应当通过 `rt_sigqueueinfo()` 系统调用进行发送（参见后面“实时信号的系统调用”一节）。

System V 和 BSD Unix 各种版本还有一个 `killpg()` 系统调用，它能显式地向一组进程发送信号。在 Linux 中，这个函数是作为一个库函数来实现的，其实现利用了 `kill()` 系统调用。另外一个变体是 `raise()`，向当前进程（即正在执行该函数的进程）发送信号，该函数在 Linux 中是作为库函数来实现的。

tkill()和tgkill()系统调用

`tkill()` 和 `tgkill()` 系统调用向线程组中的指定进程发送信号。所有遵循 POSIX 标准的 pthread 库的 `pthread_kill()` 函数，都是调用这两个函数中的任意一个向指定的轻量级进程发送信号。

`tkill()` 系统调用需要两个参数：信号接收进程的 `pid` PID 和信号编号 `sig`。`sys_tkill()` 服务例程为 `siginfo` 表赋值、获取进程描述符地址、进行许可性检查

(如前面“group_send_sig_info()函数”一节第2步所执行的操作),并调用specific_send_sig_info()发送信号。

tgkill()系统调用和tkill()有所不同, tgkill()还需要第三个参数:信号接收进程所在线程组的线程组ID(tgid)。sys_tgkill()服务例程执行的操作与sys_tkill()完全一样,不过还要检查信号接收进程是否确实属于线程组tgid。这个附加的检查解决了在向一个正在被杀死的进程发送消息时出现的竞争条件的问题:如果另外一个多线程应用正以足够快的速度创建轻量级进程,信号就可能被传递给一个错误的进程。因为线程组ID在多线程应用的整个生存期中是不会改变的,所以系统调用tgkill()解决了这个问题。

改变信号的操作

sigaction(sig,act,oact)系统调用允许用户为信号指定一个操作。当然,如果没有自定义的信号操作,那么内核执行与传递的信号相关的缺省操作。

相应的sys_sigaction()服务例程作用于两个参数:sig信号编号和类型为old_sigaction的act表(表示新的操作)。第三个可选的输出参数oact可以用来获得与信号相关的以前的操作。(old_sigaction数据结构包括与sigaction结构相同的字段,只是字段的顺序不同,在前面“与信号相关的数据结构”一节对sigaction结构进行过说明)。

这个函数首先检查act地址的有效性。然后用*act相应的字段填充类型为k_sigaction的new_ka局部变量的sa_handler、sa_flags和sa_mask字段:

```
__get_user(new_ka.sa.sa_handler, &act->sa_handler);
__get_user(new_ka.sa.sa_flags, &act->sa_flags);
__get_user(mask, &act->sa_mask);
siginitset(&new_ka.sa.sa_mask, mask);
```

函数还调用do_sigaction()把新的new_ka表拷贝到current->sig->action的在sig-1位置的表项中(信号的编号大于在数组中的位置,因为没有0信号):

```
k = &current->sig->action[sig-1];
if (act) {
    *k = *act;
    sigdelsetmask(&k->sa.sa_mask, sigmask(SIGKILL) | sigmask(SIGSTOP));
    if (k->sa.sa_handler == SIG_IGN || (k->sa.sa_handler == SIG_DFL &&
        (sig==SIGCONT || sig==SIGCHLD || sig==SIGWINCH || sig==SIGURG))) {
        rm_from_queue(sigmask(sig), &current->signal->shared_pending);
        t = current;
        do {
            rm_from_queue(sigmask(sig), &current->pending);
            recalc_sigpending_tsk(t);
            t = next_thread(t);
```

```

        } while (t != current);
    }
}

```

POSIX标准规定,当缺省操作是“Ignore”时,把信号操作设置成SIG_IGN或SIG_DFL将引起同类型的的任一挂起信号被丢弃。此外还要注意,对信号处理程序来说,不论请求屏蔽的信号是什么, SIGKILL 和 SIGSTOP 从不被屏蔽。

sigaction()系统调用还允许用户初始化表 sigaction 的 sa_flags 字段。在表 11-6 (本章前面)中,我们列出了这个字段的可能取值及其相关的含义。

原来的 System V Unix 变体提供了 signal()系统调用,它仍由编程人员广泛使用。新近的C库调用 rt_sigaction()实现了 signal()。不过, Linux 依然支持原来的C库,并提供了 sys_signal()服务例程:

```

new_sa.sa.sa_handler = handler;
new_sa.sa.sa_flags = SA_ONESHOT | SA_NOMASK;
ret = do_sigaction(sig, &new_sa, &old_sa);
return ret ? ret : (unsigned long)old_sa.sa.sa_handler;

```

检查挂起的阻塞信号

sigpending()系统调用允许进程检查挂起的阻塞信号的集合,也就是说,检查信号被阻塞时已产生的那些信号。相应的服务例程 sys_sigpending()只作用于一个参数 set,即用户变量的地址,必须将位数组拷贝到这个变量中:

```

sigorsets(&pending, &current->pending.signal,
          &current->signal->shared_pending.signal);
sigandsets(&pending, &current->blocked, &pending);
copy_to_user(set, &pending, 4);

```

修改阻塞信号的集合

sigprocmask()系统调用允许进程修改阻塞信号的集合。这个系统调用只应用于常规信号(非实时信号)。相应的 sys_sigprocmask()服务例程作用于三个参数:

o set

进程地址空间的一个指针,指向存放以前位掩码的一个位数组。

set

进程地址空间的一个指针,指向包含新位掩码的位数组。

how

一个标志,可以有下列的一个值:

SIG_BLOCK

*set 位掩码数组，指定必须加到阻塞信号的位掩码数组中的信号。

SIG_UNBLOCK

*set 位掩码数组，指定必须从阻塞信号的位掩码数组中删除的信号。

SIG_SETMASK

*set 位掩码数组，指定阻塞信号新的位掩码数组。

sys_sigprocmask()调用 copy_from_user()把 set 参数所指向的值拷贝到局部变量 new_set 中，并把 current 标准阻塞信号的位掩码数组拷贝到 old_set 局部变量中。然后根据 how 标志来指定这两个变量的值：

```

if (copy_from_user(&new_set, set, sizeof(*set)))
    return -EFAULT;
new_set &= ~(sigmask(SIGKILL) | sigmask(SIGSTOP));
old_set = current->blocked.sig[0];
if (how == SIG_BLOCK)
    sigaddsetmask(&current->blocked, new_set);
else if (how == SIG_UNBLOCK)
    sigdelsetmask(&current->blocked, new_set);
else if (how == SIG_SETMASK)
    current->blocked.sig[0] = new_set;
else
    return -EINVAL;
recalc_sigpending(current);
if (oset && copy_to_user(oset, &old_set, sizeof(*oset)))
    return -EFAULT;
return 0;

```

挂起进程

sigsuspend()系统调用把进程置为 TASK_INTERRUPTIBLE 状态，当然这是把 mask 参数指向的位掩码数组所指定的标准信号阻塞以后设置的。只有当一个非忽略、非阻塞的信号发送到进程以后，进程才被唤醒。

相应的 sys_sigsuspend()服务例程执行下列这些语句：

```

mask &= ~(sigmask(SIGKILL) | sigmask(SIGSTOP));
saveset = current->blocked;
siginitsigset(&current->blocked, mask);
recalc_sigpending(current);
regs->eax = -EINTR;
while (1) {
    current->state = TASK_INTERRUPTIBLE;
    schedule( );
    if (do_signal(regs, &saveset))

```

```
        return -EINTR;
    }
```

`schedule()`函数选择另一个进程运行。当发出`sigsuspend()`系统调用的进程又开始执行时，`sys_sigsuspend()`调用`do_signal()`函数来传递唤醒了该进程的信号。如果`do_signal()`的返回值为1，则不忽略这个信号。因此，这个系统调用返回`-EINTR`出错码后终止。

`sigsuspend()`系统调用可能看似多余，因为`sigprocmask()`和`sleep()`的组合执行显然能产生同样的效果。但这并不正确：这是因为进程可能在任何时候交错执行，你必须意识到调用一个系统调用执行操作A，紧接着又调用另一个系统调用执行操作B，并不等于调用一个单独的系统调用执行操作A，然后执行操作B。

在这种特殊情况中，`sigprocmask()`可以在调用`sleep()`之前解除对所传递信号的阻塞。如果这种情况发生，进程就可以一直停留在`TASK_INTERRUPTIBLE`状态，等待已被传递的信号。另一方面，在解除阻塞之后、`schedule()`调用之前，因为其他进程在这个时间间隔内无法获得CPU，因此，`sigsuspend()`系统调用不允许信号被发送。

实时信号的系统调用

因为前面所提到的系统调用只应用到标准信号，因此，必须引入另外的系统调用来允许用户态进程处理实时信号。

实时信号的几个系统调用（`rt_sigaction()`、`rt_sigpending()`、`rt_sigprocmask()`及`rt_sigsuspend()`）与前面描述的类似，因此不再进一步讨论。出于同样的理由，我们也不进一步讨论处理实时信号队列的两个系统调用：

`rt_sigqueueinfo()`

发送一个实时信号以便把它加入到目标进程的共享挂起信号队列中。一般通过标准库函数`sigqueue()`调用`rt_sigqueueinfo()`。

`rt_sigtimedwait()`

把阻塞的挂起信号从队列中删除而不传递它，并向调用者返回信号编号；如果没有阻塞的信号在挂起，就把当前进程挂起一个固定的时间间隔。一般通过标准库函数`sigwaitinfo()`和`sigtimedwait()`调用`rt_sigtimedwait()`。

第十二章



虚拟文件系统

Linux 成功的关键因素之一是它具有与其他操作系统和谐共存的能力。你能够透明地安装具有其他操作系统文件格式的磁盘或分区，这些操作系统如 Windows、其他版本的 Unix，甚至像 Amiga 那样的市场占有率很低的系统。通过所谓的虚拟文件系统概念，Linux 使用与其他 Unix 变体相同的方式设法支持多种文件系统类型。

虚拟文件系统所隐含的思想是把表示很多不同种类文件系统的共同信息放入内核；其中有一个字段或函数来支持 Linux 所支持的所有实际文件系统所提供的任何操作。对所调用的每个读、写或其他函数，内核都能把它们替换成支持本地 Linux 文件系统、NTFS 文件系统，或者文件所在的其他任何文件系统的实际函数。

本章讨论 Linux 虚拟文件系统的设计目标、结构及其实现。集中讨论五个 Unix 标准文件类型中的三个文件类型，即普通文件、目录文件和符号链接文件。设备文件将在第十三章中进行介绍，而管道文件会在第十九章中进行讨论。为了进一步说明实际文件系统如何工作，将在第十八章中对第二扩展文件系统（Second Extended Filesystem）进行讨论（几乎所有的 Linux 系统都使用了 Ext2）。

虚拟文件系统（VFS）的作用

虚拟文件系统 (*Virtual Filesystem*) 也可以称之为虚拟文件系统转换 (*Virtual Filesystem Switch*, VFS)，是一个内核软件层，用来处理与 Unix 标准文件系统相关的所有系统调用。其健壮性表现在能为各种文件系统提供一个通用的接口。

例如，假设一个用户输入以下 shell 命令：

```
$ cp /floppy/TEST /tmp/test
```

其中 */floppy* 是 MS-DOS 磁盘的一个安装点，而 */tmp* 是一个标准的第二扩展文件系统 (second Extended Filesystem, Ext2) 的目录。正如图 12-1 (a) 所示，VFS 是用户的应用程序与文件系统实现之间的抽象层。因此，*cp* 程序并不需要知道 */floppy/TEST* 和 */tmp/test* 是什么文件系统类型。相反，*cp* 程序直接与 VFS 交互，这是通过 Unix 程序设计人员都熟悉的普通系统调用来进行的 (参见第一章中的“文件操作的系统调用”一节)。 *cp* 的执行代码如图 12-1 (b) 所示。

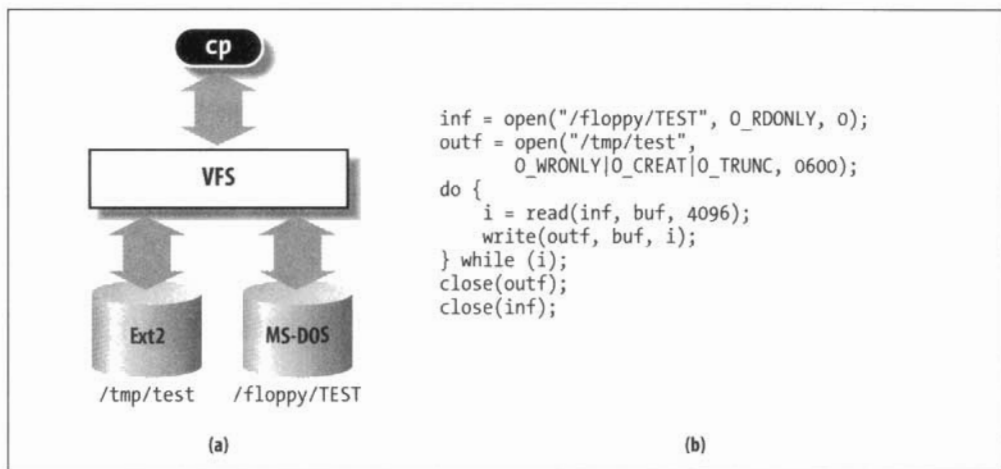


图 12-1: VFS 在一个简单的文件复制操作中的作用

VFS 支持的文件系统可以划分为三种主要类型:

磁盘文件系统

这些文件系统管理在本地磁盘分区中可用的存储空间或者其他可以起到磁盘作用的设备 (比如说一个 USB 闪存)。VFS 支持的基于磁盘的某些著名文件系统还有:

- Linux 使用的文件系统，如广泛使用的第二扩展文件系统 (Ext2)，新近的第三扩展文件系统 (Third Extended Filesystem, Ext3) 及 Reiser 文件系统 (ReiserFS) (注 1)。
- Unix 家族的文件系统，如 sysv 文件系统 (System V、Coherent、Xenix)、UFS (BSD、Solaris、NEXTSTEP)，MINIX 文件系统及 VERITAS VxFS (SCO UnixWare)。

注 1: 尽管这些文件系统诞生于 Linux，但已经移植到其他几个操作系统中。

- 微软公司的文件系统，如 MS-DOS、VFAT（Windows 95 及随后的版本）及 NTFS（Windows NT 以及随后的版本）。
- ISO9660 CD-ROM 文件系统（以前的 High Sierra 文件系统）和通用磁盘格式 (UDF) 的 DVD 文件系统。
- 其他有专利权的文件系统，如 HPFS（IBM 公司的 OS/2）、HFS（苹果公司的 Macintosh）、AFFS（Amiga 公司的快速文件系统）以及 ADFS（Acorn 公司的磁盘文件归档系统）。
- 起源于非 Linux 系统的其他日志文件系统，如 IBM 的 JFS 和 SGI 的 XFS。

网络文件系统

这些文件系统允许轻易地访问属于其他网络计算机的文件系统所包含的文件。虚拟文件系统所支持的一些著名的网络文件系统有：NFS、Coda、AFS（Andrew 文件系统）、CIFS（用于 Microsoft Windows 的通用网络文件系统）以及 NCP（Novell 公司的 NetWare Core Protocol）。

特殊文件系统

这些文件系统不管理本地或者远程磁盘空间。`/proc` 文件系统是特殊文件系统的—个典型范例（参见稍后“特殊文件系统”—节）。

由于篇幅所限，本书只详细描述 Ex2 和 Ext3 文件系统（参见第十八章），对其他文件系统将不做介绍。

在第一章“Unix 文件系统概述”—节中曾提到，Unix 的目录建立了一棵根目录为“/”的树。根目录包含在根文件系统（*root filesystem*）中，在 Linux 中这个根文件系统通常就是 Ext2 或 Ext3 类型。其他所有的文件系统都可以被“安装”在根文件的子目录中（注 2）。

基于磁盘的文件系统通常存放在硬件块设备中，如硬盘、软盘或者 CD-ROM。Linux VFS 的一个有用特点是能够处理如 `/dev/loop0` 这样的虚拟块设备，这种设备可以用来安装普通文件所在的文件系统。作为一种可能的应用，用户可以保护自己的私有文件系统，这可以通过把自己文件系统的加密版本存放在一个普通文件中来实现。

注 2： 当一个文件系统被安装在某一个目录上时，在父文件系统中的目录内容不再是可访问的，因为任何路径（包括安装点），都将引用已安装的文件系统。但是，当被安装文件系统卸载时，原目录的内容又可再现。这种令人惊异的 Unix 文件系统特点可以由系统管理员用来隐藏文件，他们只需把一个文件系统安装在要隐藏文件的目录中即可。

第一个虚拟文件系统包含在 1986 年由 Sun 公司发布的 SunOS 操作系统中。从那时起，多数 Unix 文件系统都包含 VFS。然而，Linux 的 VFS 支持最广泛的文件系统。

通用文件模型

VFS 所隐含的主要思想在于引入了一个通用的文件模型 (*common file model*)，这个模型能够表示所有支持的文件系统。该模型严格反映传统 Unix 文件系统提供的文件模型。这并不奇怪，因为 Linux 希望以最小的额外开销运行它的本地文件系统。不过，要实现每个具体的文件系统，必须将其物理组织结构转换为虚拟文件系统的通用文件模型。

例如，在通用文件模型中，每个目录被看作一个文件，可以包含若干文件和其他的子目录。但是，存在几个非 Unix 的基于磁盘的文件系统，它们利用文件分配表 (File Allocation Table, FAT) 存放每个文件在目录树中的位置，在这些文件系统中，存放的是目录而不是文件。为了符合 VFS 的通用文件模型，对上述基于 FAT 的文件系统的实现，Linux 必须在必要时能够快速建立对应于目录的文件。这样的文件只作为内核内存的对象而存在。

从本质上说，Linux 内核不能对一个特定的函数进行硬编码来执行诸如 `read()` 或 `ioctl()` 这样的操作，而是对每个操作都必须使用一个指针，指向要访问的具体文件系统的适当函数。

为了进一步说明这一概念，参见图 12-1，其中显示了内核如何把 `read()` 转换为专对 MS-DOS 文件系统的的一个调用。应用程序对 `read()` 的调用引起内核调用相应的 `sys_read()` 服务例程，这与其他系统调用完全类似。我们在本章后面会看到，文件在内核内存中是由一个 `file` 数据结构来表示的。这种数据结构中包含一个称为 `f_op` 的字段，该字段中包含一个指向专对 MS-DOS 文件的函数指针，当然还包括读文件的函数。`sys_read()` 查找指向该函数的指针，并调用它。这样一来，应用程序的 `read()` 就被转化为相对间接的调用：

```
file->f_op->read(...);
```

与之类似，`write()` 操作也会引发一个与输出文件相关的 Ext2 写函数的执行。简而言之，内核负责把一组合适的指针分配给与每个打开文件相关的 `file` 变量，然后负责调用针对每个具体文件系统的函数（由 `f_op` 字段指向）。

你可以把通用文件模型看作是面向对象的，在这里，对象是一个软件结构，其中既定义了数据结构也定义了其上的操作方法。出于效率的考虑，Linux 的编码并未采用面向对象的程序设计语言（比如 C++）。因此对象作为普通的 C 数据结构来实现，数据结构中指向函数的字段就对应于对象的方法。

通用文件模型由下列对象类型组成：

超级块对象 (superblock object)

存放已安装文件系统的有关信息。对基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的文件系统控制块 (filesystem control block)。

索引节点对象 (inode object)

存放关于具体文件的一般信息。对基于磁盘的文件系统，这类对象通常对应于存放在磁盘上的文件控制块 (file control block)。每个索引节点对象都有一个索引节点号，这个节点号唯一地标识文件系统中的文件。

文件对象(file object)

存放打开文件与进程之间进行交互的有关信息。这类信息仅当进程访问文件期间存在于内核内存中。

目录项对象(dentry object)

存放目录项 (也就是文件的特定名称) 与对应文件进行链接的有关信息。每个磁盘文件系统都以自己特有的方式将该类信息存在磁盘上。

如图12-2所示是一个简单的示例，说明进程怎样与文件进行交互。三个不同进程已经打开同一个文件，其中两个进程使用同一个硬链接。在这种情况下，其中的每个进程都使用自己的文件对象，但只需要两个目录项对象，每个硬链接对应一个目录项对象。这两个目录项对象指向同一个索引节点对象，该索引节点对象标识超级块对象，以及随后的普通磁盘文件。

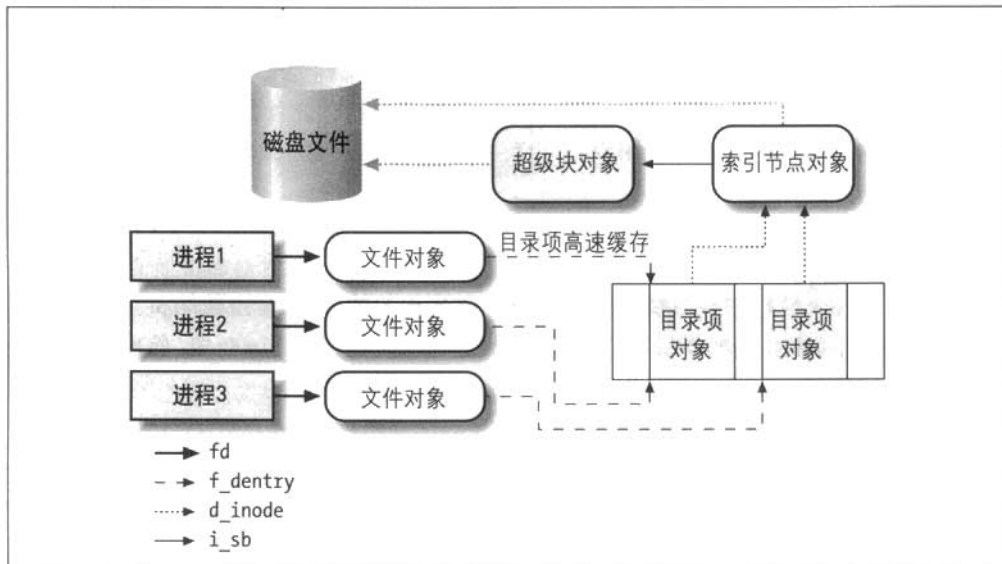


图 12-2: 进程与 VFS 对象之间的交互

VFS除了能为所有文件系统的实现提供一个通用接口外,还具有另一个与系统性能相关的重要作用。最近最常使用的目录项对象被放在所谓目录项高速缓存 (*dentry cache*) 的磁盘高速缓存中,以加速从文件路径名到最后一个路径分量的索引节点的转换过程。

一般说来,磁盘高速缓存 (*disk cache*) 属于软件机制,它允许内核将原本存在磁盘上的某些信息保存在RAM中,以便对这些数据的进一步访问能快速进行,而不必慢速访问磁盘本身。

注意,磁盘高速缓存不同于硬件高速缓存或内存高速缓存,后两者都与磁盘或其他设备无关。硬件高速缓存是一个快速静态RAM,它加快了直接对慢速动态RAM的请求(参见第二章中的“硬件高速缓存”一节)。内存高速缓存是一种软件机制,引入它是为了绕过内核内存分配器(参见第八章中的“slab分配器”一节)。

除了目录项高速缓存和索引结点高速缓存之外, Linux 还使用其他磁盘高速缓存。其中最重要的一种就是所谓的页高速缓存,我们将在第十五章中进行详细介绍。

VFS 所处理的系统调用

表 12-1 列出了 VFS 的系统调用,这些系统调用涉及文件系统、普通文件、目录文件以及符号链接文件。另外还有少数几个由 VFS 处理的其他系统调用,诸如 `ioperm()`、`ioctl()`、`pipe()` 和 `mknod()`,涉及设备文件和管道文件,这些将在后续章节中讨论。最后一组由 VFS 处理的系统调用,诸如 `socket()`、`connect()` 和 `bind()` 属于套接字系统调用,并用于实现网络功能。与表 12-1 列出的系统调用对应的一些内核服务例程,我们会在本章或第十八章中陆续进行讨论。

表 12-1: 由 VFS 处理的一些系统调用

系统调用名	说明
<code>mount()</code> <code>umount()</code> <code>umount2()</code>	安装/卸载文件系统
<code>sysfs()</code>	获取文件系统信息
<code>statfs()</code> <code>fstatfs()</code> <code>statfs64()</code> <code>fstatfs64()</code>	获取文件系统统计信息
<code>ustat()</code>	
<code>chroot()</code> <code>pivot_root()</code>	更改根目录
<code>chdir()</code> <code>fchdir()</code> <code>getcwd()</code>	对当前目录进行操作
<code>mkdir()</code> <code>rmdir()</code>	创建/删除目录
<code>getdents()</code> <code>getdents64()</code> <code>readdir()</code> <code>link()</code>	对目录项进行操作
<code>unlink()</code> <code>rename()</code> <code>lookup_dcookie()</code>	
<code>readlink()</code> <code>symlink()</code>	对软链接进行操作

表 12-1: 由 VFS 处理的一些系统调用 (续)

系统调用名	说明
chown() fchown() lchown() chown16() fchown16() lchown16()	更改文件所有者性
chmod() fchmod() utime()	更改文件属性
stat() fstat() lstat() access() oldstat() oldfstat() oldlstat() stat64() lstat64() fstat64()	读取文件状态
open() close() creat() umask()	打开 / 关闭 / 创建文件
dup() dup2() fcntl() fcntl64()	对文件描述符进行操作
select() poll()	等待一组文件描述符上发生的事件
truncate() ftruncate() truncate64() ftruncate64()	更改文件长度
lseek() _llseek()	更改文件指针
read() write() readv() writev() sendfile() sendfile64() readahead()	进行文件 I/O 操作
io_setup() io_submit() io_getevents() io_cancel() io_destroy()	异步 I/O (允许多个读和写请求)
pread64() pwrite64()	搜索并访问文件
mmap() mmap2() munmap() madvise() mincore() remap_file_pages()	处理文件内存映射
fdatasync() fsync() sync() msync() flock()	同步文件数据 处理文件锁
setxattr() lsetxattr() fsetxattr() getxattr() lgetxattr() fgetxattr() listxattr() llistxattr() flistxattr() removexattr() lremovexattr() fremovexattr()	处理文件扩展属性

前面我们已经提到, VFS 是应用程序和具体文件系统之间的一层。不过, 在某些情况下, 一个文件操作可能由 VFS 本身去执行, 无需调用低层函数。例如, 当某个进程关闭一个打开的文件时, 并不需要涉及磁盘上的相应文件, 因此 VFS 只需释放对应的文件对象。类似地, 当系统调用 `lseek()` 修改一个文件指针, 而这个文件指针是打开文件与进程交互所涉及的一个属性时, VFS 就只需修改对应的文件对象, 而不必访问磁盘上的文件, 因此, 无需调用具体文件系统的函数。从某种意义上说, 可以把 VFS 看成“通用”文件系统, 它在必要时依赖某种具体文件系统。

VFS 的数据结构

每个VFS对象都存放在一个适当的数据结构中,其中包括对象的属性和指向对象方法表的指针。内核可以动态地修改对象的方法,因此可以为对象建立专用的行为。下面几节详细介绍VFS的对象及其内在关系。

超级块对象

超级块对象由 `super_block` 结构组成,表 12-2 列举了其中的字段。

表 12-2: 超级块对象的字段

类型	字段	说明
<code>struct list_head</code>	<code>s_list</code>	指向超级块链表的指针
<code>dev_t</code>	<code>s_dev</code>	设备标识符
<code>unsigned long</code>	<code>s_blocksize</code>	以字节为单位的块大小
<code>unsigned long</code>	<code>s_old_blocksize</code>	基本块设备驱动程序中提到的以字节为单位的块大小
<code>unsigned char</code>	<code>s_blocksize_bits</code>	以位为单位的块大小
<code>unsigned char</code>	<code>s_dirt</code>	修改(脏)标志
<code>unsigned long long</code>	<code>s_maxbytes</code>	文件的最长长度
<code>struct file_system_type *</code>	<code>s_type</code>	文件系统类型
<code>struct super_operations *</code>	<code>s_op</code>	超级块方法
<code>struct dqquot_operations *</code>	<code>dq_op</code>	磁盘限额处理方法
<code>struct quotactl_ops *</code>	<code>s_qcop</code>	磁盘限额管理方法
<code>struct export_operations</code>	<code>s_export_op</code>	网络文件系统使用的输出操作
<code>unsigned long</code>	<code>s_flags</code>	安装标志
<code>unsigned long</code>	<code>s_magic</code>	文件系统的魔数
<code>struct dentry *</code>	<code>s_root</code>	文件系统根目录的目录项对象
<code>struct rw_semaphore</code>	<code>s_umount</code>	卸载所用的信号量
<code>struct semaphore</code>	<code>s_lock</code>	超级块信号量
<code>int</code>	<code>s_count</code>	引用计数器
<code>int</code>	<code>s_syncing</code>	表示对超级块的索引节点进行同步的标志
<code>int</code>	<code>s_need_sync_fs</code>	对超级块的已安装文件系统进行同步的标志

表 12-2: 超级块对象的字段 (续)

类型	字段	说明
atomic_t	s_active	次级引用计数器
void *	s_security	指向超级块安全数据结构的指针
struct xattr_handler **	s_xattr	指向超级块扩展属性结构的指针
struct list_head	s_inodes	所有索引节点的链表
struct list_head	s_dirty	改进型索引节点的链表
struct list_head	s_io	等待被写入磁盘的索引节点的链表
struct hlist_head	s_anon	用于处理远程网络文件系统的匿名目录项的链表
struct list_head	s_files	文件对象的链表
struct block_device *	s_bdev	指向块设备驱动程序描述符的指针
struct list_head	s_instances	用于给定文件系统类型的超级块对象链表的指针 (参见后面的“文件系统类型注册”一节)
struct quota_info	s_dquot	磁盘限额的描述符
int	s_frozen	冻结文件系统时使用的标志 (强制置于一致状态)
wait_queue_head_t	s_wait_unfrozen	进程挂起的等待队列, 直到文件系统被解冻
char[]	s_id	包含超级块的块设备名称
void *	s_fs_info	指向特定文件系统的超级块信息的指针
struct semaphore	s_vfs_rename_sem	当VFS通过目录重命名文件时使用的信号量
u32	s_time_gran	时间戳的粒度 (纳秒级)

所有超级块对象都以双向循环链表的形式链接在一起。链表中第一个元素用 `super_blocks` 变量来表示, 而超级块对象的 `s_list` 字段存放指向链表相邻元素的指针。 `sb_lock` 自旋锁保护链表免受多处理器系统上的同时访问。

`s_fs_info` 字段指向属于具体文件系统的超级块信息, 例如, 我们在第十八章将会看到, 假如超级块对象指的是 Ext2 文件系统, 该字段就指向 `ext2_sb_info` 数据结构, 该结构包括磁盘分配位掩码和其他与 VFS 的通用文件模型无关的数据。

通常，为了效率起见，由 `s_fs_info` 字段所指向的数据被复制到内存。任何基于磁盘的文件系统都需要访问和更改自己的磁盘分配位图，以便分配或释放磁盘块。VFS 允许这些文件系统直接对内存超级块的 `s_fs_info` 字段进行操作，而无需访问磁盘。

但是，这种方法带来一个新问题：有可能 VFS 超级块最终不再与磁盘上相应的超级块同步。因此，有必要引入一个 `s_dirt` 标志来表示该超级块是否是脏的——那磁盘上的数据是否必须要更新。缺乏同步还会导致产生我们熟悉的一个问题：当一台机器的电源突然断开而用户来不及正常关闭系统时，就会出现文件系统崩溃。我们将会在第 15 章的“把脏页写入磁盘”一节中看到，Linux 是通过周期性地将所有“脏”的超级块写回磁盘来减少该问题带来的危害。

与超级块关联的方法就是所谓的超级块操作。这些操作是由数据结构 `super_operations` 来描述的，该结构的起始地址存放在超级块的 `s_op` 字段中。

每个具体的文件系统都可以定义自己的超级块操作。当 VFS 需要调用其中一个操作时，比如说 `read_inode()`，它执行下列操作：

```
sb->s_op->read_inode(inode);
```

这里 `sb` 存放所涉及超级块对象的地址。`super_operations` 表的 `read_inode` 字段存放这一函数的地址，因此，这一函数被直接调用。

让我们简要描述一下超级块操作，其中实现了一些高级操作，比如删除文件或安装磁盘。下面这些操作按照它们在 `super_operation` 表中出现的顺序来排列：

`alloc_inode(sb)`

为索引节点对象分配空间，包括具体文件系统的数据库所需要的空间。

`destroy_inode(inode)`

撤消索引节点对象，包括具体文件系统的数据库。

`read_inode(inode)`

用磁盘上的数据填充以参数传递过来的索引节点对象的字段；索引节点对象的 `i_ino` 字段标识从磁盘上要读取的具体文件系统的索引节点。

`dirty_inode(inode)`

当索引节点标记为修改（脏）时调用。像 ReiserFS 和 Ext3 这样的文件系统用它来更新磁盘上的文件系统日志。

`write_inode(inode, flag)`

用通过传递参数指定的索引节点对象的内容更新一个文件系统的索引节点。索引节点对象的 `i_ino` 字段标识所涉及磁盘上文件系统的索引节点。`flag` 参数表示 I/O 操作是否应当同步。

`put_inode(inode)`

释放索引节点时调用（减少该节点引用计数器值）以执行具体文件系统操作。

`drop_inode(inode)`

在即将撤消索引节点时调用——也就是说，当最后一个用户释放该索引节点时，实现该方法的文件系统通常使用 `generic_drop_inode()` 函数。该函数从 VFS 数据结构中移走对索引节点的每一个引用，如果索引节点不再出现在任何目录中，则调用超级块方法 `delete_inode` 将它从文件系统中删除。

`delete_inode(inode)`

在必须撤消索引节点时调用。删除内存中的 VFS 索引节点和磁盘上的文件数据及元数据。

`put_super(super)`

释放通过传递的参数指定的超级块对象（因为相应的文件系统被卸载）。

`write_super(super)`

用指定对象的内容更新文件系统的超级块。

`sync_fs(sb, wait)`

在清除文件系统来更新磁盘上的具体文件系统数据结构时调用（由日志文件系统使用）。

`write_super_lockfs(super)`

阻塞对文件系统的修改并用指定对象的内容更新超级块。当文件系统被冻结时调用该方法，例如，由逻辑卷管理器驱动程序（LVM）调用。

`unlockfs(super)`

取消由 `write_super_lockfs()` 超级块方法实现的 对文件系统更新的阻塞。

`statfs(super, buf)`

将文件系统的统计信息返回，填写在 `buf` 缓冲区中。

`remount_fs(super, flags, data)`

用新的选项重新安装文件系统（当某个安装选项必须被修改时被调用）。

`clear_inode(inode)`

当撤消磁盘索引节点执行具体文件系统操作时调用。

`umount_begin(super)`

中断一个安装操作，因为相应的卸载操作已经开始（只在网络文件系统中使用）。

`show_options(seq_file, vfsmount)`

用来显示特定文件系统的选项。

```
quota_read(super, type, data, size, offset)
```

限额系统使用该方法从文件中读取数据，该文件详细说明了所在文件系统的限制（注3）。

```
quota_write(super, type, data, size, offset)
```

限额系统使用该方法将数据写入文件中，该文件详细说明了所在文件系统的限制。

前述的方法对所有可能的文件系统类型均是可用的。但是，只有其中的一个子集应用到每个具体的文件系统；未实现的方法对应的字段置为 `NULL`。注意，系统没有定义 `get_super` 方法来读超级块，那么，内核如何能够调用一个对象的方法而从磁盘读出该对象？我们将在描述文件系统类型的另一个对象中找到等价的 `get_sb` 方法（参见后面的“文件系统类型注册”一节）。

索引节点对象

文件系统处理文件所需要的所有信息都放在一个名为索引节点的数据结构中。文件名可以随时更改，但是索引节点对文件是唯一的，并且随文件的存在而存在。内存中的索引节点对象由一个 `inode` 数据结构组成，其字段如表 12-3 所示。

表 12-3：索引节点对象的字段

类型	字段	说明
Struct <code>hlist_node</code>	<code>i_hash</code>	用于散列链表的指针
<code>struct list_head</code>	<code>i_list</code>	用于描述索引节点当前状态的链表的指针
<code>struct list_head</code>	<code>i_sb_list</code>	用于超级块的索引节点链表的指针
<code>struct list_head</code>	<code>i_dentry</code>	引用索引节点的目录项对象链表的头
<code>unsigned long</code>	<code>i_ino</code>	索引节点号
<code>atomic_t</code>	<code>i_count</code>	引用计数器
<code>umode_t</code>	<code>i_mode</code>	文件类型与访问权限
<code>unsigned int</code>	<code>i_nlink</code>	硬链接数目
<code>uid_t</code>	<code>i_uid</code>	所有者标识符

注3： 限额系统 (*quota system*) 为每个用户和 (或) 组限制了它们在给定文件系统上所能使用的空间大小 (参见 `quotactl()` 系统调用)。

表 12-3: 索引节点对象的字段 (续)

类型	字段	说明
gid_t	i_gid	组标识符
dev_t	i_rdev	实设备标识符
loff_t	i_size	文件的字节数
struct timespec	i_atime	上次访问文件的时间
struct timespec	i_mtime	上次写文件的时间
struct timespec	i_ctime	上次修改索引节点的时间
unsigned int	i_blkbits	块的位数
unsigned long	i_blksize	块的字节数
unsigned long	i_version	版本号 (每次使用后自动递增)
unsigned long	i_blocks	文件的块数
unsigned short	i_bytes	文件中最后一个块的字节数
unsigned char	i_sock	如果文件是一个套接字则为非零
spinlock_t	i_lock	保护索引节点一些字段的自旋锁
struct semaphore	i_sem	索引节点信号量
struct rw_semaphore	i_alloc_sem	在直接 I/O 文件操作中避免出现竞争条件的读 / 写信号量
struct inode_operations *	i_op	索引节点的操作
struct file_operations *	i_fop	缺省文件操作
struct super_block *	i_sb	指向超级块对象的指针
struct file_lock *	i_flock	指向文件锁链表的指针
struct address_space *	i_mapping	指向 address_space 对象的指针 (参见第十五章)
struct address_space	i_data	文件的 address_space 对象
struct dquot * []	i_dquot	索引节点磁盘限额
struct list_head	i_devices	用于具体的字符或块设备索引节点链表的指针 (参见第十三章)
struct pipe_inode_info *	i_pipe	如果文件是一个管道则使用它 (参见第十九章)
struct block_device *	i_bdev	指向块设备驱动程序的指针
struct cdev * i_cdev		指向字符设备驱动程序的指针
int	i_cindex	拥有一组次设备号的设备文件的索引

表 12-3: 索引节点对象的字段 (续)

类型	字段	说明
<code>_u32</code>	<code>i_generation</code>	索引节点版本号 (由某些文件系统使用)
<code>unsigned long</code>	<code>i_dnotify_mask</code>	目录通知事件的位掩码
<code>struct dnotify_struct *</code>	<code>i_dnotify</code>	用于目录通知
<code>unsigned long</code>	<code>i_state</code>	索引节点的状态标志
<code>unsigned long</code>	<code>dirtied_when</code>	索引节点的弄脏时间 (以节拍为单位)
<code>unsigned int</code>	<code>i_flags</code>	文件系统的安装标志
<code>atomic_t</code>	<code>i_writecount</code>	用于写进程的引用计数器
<code>void *</code>	<code>i_security</code>	指向索引节点安全结构的指针
<code>void *</code>	<code>u.generic_ip</code>	指向私有数据的指针
<code>seqcount_t</code>	<code>i_size_seqcount</code>	SMP 系统为 <code>i_size</code> 字段获取一致值时使用的顺序计数器

每个索引节点对象都会复制磁盘索引节点包含的一些数据, 比如分配给文件的磁盘块数。如果 `i_state` 字段的值等于 `I_DIRTY_SYNC`、`I_DIRTY_DATASYNC` 或 `I_DIRTY_PAGES`, 该索引节点就是“脏”的, 也就是说, 对应的磁盘索引节点必须被更新。`I_DIRTY` 宏可以用来立即检查这三个标志的值 (详细内容参见后面)。`i_state` 字段的其他值有 `I_LOCK` (涉及的索引节点对象处于 I/O 传送中)、`I_FREEING` (索引节点对象正在被释放)、`I_CLEAR` (索引节点对象的内容不再有意义) 以及 `I_NEW` (索引节点对象已经分配但还没有用从磁盘索引节点读取来的数据填充)。

每个索引节点对象总是出现在下列双向循环链表的某个链表中 (所有情况下, 指向相邻元素的指针存放在 `i_list` 字段中):

- 有效未使用的索引节点链表, 典型的如那些镜像有效的磁盘索引节点, 且当前未被任何进程使用。这些索引节点不为脏, 且它们的 `i_count` 字段置为 0。链表中的首元素和尾元素是由变量 `inode_unused` 的 `next` 字段和 `prev` 字段分别指向的。这个链表用作磁盘高速缓存。
- 正在使用的索引节点链表, 也就是那些镜像有效的磁盘索引节点, 且当前被某些进程使用。这些索引节点不为脏, 但它们的 `i_count` 字段为正数。链表中的首元素和尾元素是由变量 `inode_in_use` 引用的。
- 脏索引节点的链表。链表中的首元素和尾元素是由相应超级块对象的 `s_dirty` 字段引用的。

这些链表都是通过适当的索引节点对象的 `i_list` 字段链接在一起的。

此外，每个索引节点对象也包含在每文件系统 (*per-filesystem*) 的双向循环链表中，链表的头存放在超级块对象的 `s_inodes` 字段中；索引节点对象的 `i_sb_list` 字段存放了指向链表相邻元素的指针。

最后，索引节点对象也存放在一个称为 `inode_hashtable` 的散列表中。散列表加快了对索引节点对象的搜索，前提是系统内核要知道索引节点号及文件所在文件系统对应的超级块对象的地址。由于散列技术可能引发冲突，所以索引节点对象包含一个 `i_hash` 字段，该字段中包含向前和向后的两个指针，分别指向散列到同一地址的前一个索引节点和后一个索引节点；该字段因此创建了由这些索引节点组成的一个双向链表。

与索引节点对象关联的方法也叫索引节点操作。它们由 `inode_operations` 结构来描述，该结构的地址存放在 `i_op` 字段中。以下是索引节点的操作，以它们在 `inode_operations` 表中出现的次序来排列：

```
create(dir, dentry, mode, nameidata)
```

在某一目录下，为与目录项对象相关的普通文件创建一个新的磁盘索引节点。

```
lookup(dir, dentry, nameidata)
```

为包含在一个目录项对象中的文件名对应的索引节点查找目录。

```
link(old_dentry, dir, new_dentry)
```

创建一个新的名为 `new_dentry` 的硬链接，它指向 `dir` 目录下名为 `old_dentry` 的文件。

```
unlink(dir, dentry)
```

从一个目录中删除目录项对象所指定文件的硬链接。

```
symlink(dir, dentry, symname)
```

在某个目录下，为与目录项对象相关的符号链接创建一个新的索引节点。

```
mkdir(dir, dentry, mode)
```

在某个目录下，为与目录项对象相关的目录创建一个新的索引节点。

```
rmdir(dir, dentry)
```

从一个目录删除子目录，子目录的名称包含在目录项对象中。

```
mknod(dir, dentry, mode, rdev)
```

在某个目录中，为与目录项对象相关的特定文件创建一个新的磁盘索引节点。其中参数 `mode` 和 `rdev` 分别表示文件的类型和设备的主次设备号。

`rename(old_dir, old_dentry, new_dir, new_dentry)`

将 `old_dir` 目录下由 `old_dentry` 标识的文件移到 `new_dir` 目录下。新文件名包含在 `new_dentry` 指向的目录项对象中。

`readlink(dentry, buffer, buflen)`

将目录项所指定的符号链接中对应的文件路径名拷贝到 `buffer` 所指定的用户态内存区。

`follow_link(inode, nameidata)`

解析索引节点对象所指定的符号链接；如果该符号链接是一个相对路径名，则从第二个参数所指定的目录开始进行查找。

`put_link(dentry, nameidata)`

释放由 `follow_link` 方法分配的用于解析符号链接的所有临时数据结构。

`truncate(inode)`

修改与索引节点相关的文件长度。在调用该方法之前，必须将 `inode` 对象的 `i_size` 字段设置为需要的新长度值。

`permission(inode, mask, nameidata)`

检查是否允许对与索引节点所指的文件进行指定模式的访问。

`setattr(dentry, iattr)`

在触及索引节点属性后通知一个“修改事件”。

`getattr(mnt, dentry, kstat)`

由一些文件系统用于读取索引节点属性。

`setxattr(dentry, name, value, size, flags)`

为索引节点设置“扩展属性”（扩展属性存放在任何索引节点之外的磁盘块中）。

`getxattr(dentry, name, buffer, size)`

获取索引节点的扩展属性。

`listxattr(dentry, buffer, size)`

获取扩展属性名称的整个链表。

`removexattr(dentry, name)`

删除索引节点的扩展属性。

上述列举的方法对所有可能的索引节点和文件系统类型都是可用的。不过，只有其中的一个子集应用到某一特定的索引节点和文件系统；未实现的方法对应的字段被置为 `NULL`。

文件对象

文件对象描述进程怎样与一个打开的文件进行交互。文件对象是在文件被打开时创建的，由一个 `file` 结构组成，其中包含的字段如表 12-4 所示。注意，文件对象在磁盘上没有对应的映像，因此 `file` 结构中没有设置“脏”字段来表示文件对象是否已被修改。

表 12-4：文件对象的字段

类型	字段	说明
<code>struct list_head</code>	<code>f_list</code>	用于通用文件对象链表的指针
<code>struct dentry *</code>	<code>f_dentry</code>	与文件相关的目录项对象
<code>struct vfsmount *</code>	<code>f_vfsmnt</code>	含有该文件的已安装文件系统
<code>struct file_operations *</code>	<code>f_op</code>	指向文件操作表的指针
<code>atomic_t</code>	<code>f_count</code>	文件对象的引用计数器
<code>unsigned int</code>	<code>f_flags</code>	当打开文件时所指定的标志
<code>mode_t</code>	<code>f_mode</code>	进程的访问模式
<code>int</code>	<code>f_error</code>	网络写操作的错误码
<code>loff_t</code>	<code>f_pos</code>	当前的文件位移量（文件指针）
<code>struct fown_struct</code>	<code>f_owner</code>	通过信号进行 I/O 事件通知的数据
<code>unsigned int</code>	<code>f_uid</code>	用户的 UID
<code>unsigned int</code>	<code>f_gid</code>	用户的 GID
<code>struct file_ra_state</code>	<code>f_ra</code>	文件预读状态（参见第十六章）
<code>size_t</code>	<code>f_maxcount</code>	一次单一的操作所能读或写的最大字节数（当前设置为 $2^{31}-1$ ）
<code>unsigned long</code>	<code>f_version</code>	版本号，每次使用后自动递增
<code>void *</code>	<code>f_security</code>	指向文件对象的安全结构的指针
<code>void *</code>	<code>private_data</code>	指向特定文件系统或设备驱动程序所需的数据的指针
<code>struct list_head</code>	<code>f_ep_links</code>	文件的事件轮询等待者链表的头
<code>spinlock_t</code>	<code>f_ep_lock</code>	保护 <code>f_ep_links</code> 链表的自旋锁
<code>struct address_space *</code>	<code>f_mapping</code>	指向文件地址空间对象的指针（参见第十五章）

存放在文件对象中的主要信息是文件指针，即文件中当前的位置，下一个操作将在该位置发生。由于几个进程可能同时访问同一文件，因此文件指针必须存放在文件对象而不是索引节点对象中。

文件对象通过一个名为 `filp` 的 slab 高速缓存分配, `filp` 描述符地址存放在 `filp_cachep` 变量中。由于分配的文件对象数目是有限的, 因此 `files_stat` 变量在其 `max_files` 字段中指定了可分配文件对象的最大数目, 也就是系统可同时访问的最大文件数 (注 4)。

“在使用“文件对象包含在由具体文件系统的超级块所确立的几个链表中。每个超级块对象把文件对象链表的头存放在 `s_files` 字段中; 因此, 属于不同文件系统的文件对象就包含在不同的链表中。链表中分别指向前一个元素和后一个元素的指针都存放在文件对象的 `f_list` 字段中。`files_lock` 自旋锁保护超级块的 `s_files` 链表免受多处理器系统上的同时访问。

文件对象的 `f_count` 字段是一个引用计数器: 它记录使用文件对象的进程数 (记住, 以 `CLONE_FILES` 标志创建的轻量级进程共享打开文件表, 因此它们可以使用相同的文件对象)。当内核本身使用该文件对象时也要增加计数器的值——例如, 把对象插入链表中或发出 `dup()` 系统调用时。

当 VFS 代表进程必须打开一个文件时, 它调用 `get_empty_filp()` 函数来分配一个新的文件对象。该函数调用 `kmem_cache_alloc()` 从 `filp` 高速缓存中获得一个空闲的文件对象, 然后初始化这个对象的字段, 如下所示:

```
memset(f, 0, sizeof(*f));
INIT_LIST_HEAD(&f->f_ep_links);
spin_lock_init(&f->f_ep_lock);
atomic_set(&f->f_count, 1);
f->f_uid = current->fsuid;
f->f_gid = current->fsgid;
f->f_owner.lock = RW_LOCK_UNLOCKED;
INIT_LIST_HEAD(&f->f_list);
f->f_maxcount = INT_MAX;
```

正如在“通用文件模型”一节中讨论过的那样, 每个文件系统都有其自己的文件操作集合, 执行诸如读写文件这样的操作。当内核将一个索引节点从磁盘装入内存时, 就会把指向这些文件操作的指针存放在 `file_operations` 结构中, 而该结构的地址存放在该索引节点对象的 `i_fop` 字段中。当进程打开这个文件时, VFS 就用存放在索引节点中的这个地址初始化新文件对象的 `f_op` 字段, 使得对文件操作的后续调用能够使用这些函数。如果需要, VFS 随后也可以通过在 `f_op` 字段存放一个新值而修改文件操作的集合。

注 4: 内核初始化期间, `files_init()` 函数把 `max_files` 字段设置为可用 RAM 大小的 1/10 (以千字节为单位)。不过, 系统管理员可以通过写 `/proc/sys/fs/file-max` 文件来修改这个值。而且, 即使 `max_files` 文件对象已经被分配, 超级用户也总是可以获得一个文件对象。

下面的列表描述了文件的操作，以它们在 `file_operations` 表中出现的次序来排列：

`lseek(file, offset, origin)`

更新文件指针。

`read(file, buf, count, offset)`

从文件的 `*offset` 处开始读出 `count` 个字节，然后增加 `*offset` 的值（一般与文件指针对应）。

`aio_read(req, buf, len, pos)`

启动一个异步 I/O 操作，从文件的 `pos` 处开始读出 `len` 个字节的数据并将它们放入 `buf` 中（引入它是为了支持 `io_submit()` 系统调用）。

`write(file, buf, count, offset)`

从文件的 `*offset` 处开始写入 `count` 个字节，然后增加 `*offset` 的值（一般与文件指针对应）。

`aio_write(req, buf, len, pos)`

启动一个异步 I/O 操作，从 `buf` 中取 `len` 个字节写入文件 `pos` 处。

`readdir(dir, dirent, filldir)`

返回一个目录的下一个目录项，返回值存入参数 `dirent`；参数 `filldir` 存放一个辅助函数的地址，该函数可以提取目录项的各个字段。

`poll(file, poll_table)`

检查是否在一个文件上有操作发生，如果没有则睡眠，直到该文件上有操作发生。

`ioctl(inode, file, cmd, arg)`

向一个基本硬件设备发送命令。该方法只适用于设备文件。

`unlocked_ioctl(file, cmd, arg)`

与 `ioctl` 方法类似，但是它不用获得大内核锁（参见第五章的“大内核锁”一节）。我们认为所有的设备驱动程序和文件系统都将使用这个新方法而不是 `ioctl` 方法。

`compat_ioctl(file, cmd, arg)`

64 位的内核使用该方法执行 32 位的系统调用 `ioctl()`。

`mmap(file, vma)`

执行文件的内存映射，并将映射放入进程的地址空间（参见第十六章的“内存映射”一节）。

`open(inode, file)`

通过创建一个新的文件对象而打开一个文件，并把它链接到相应的索引节点对象（参见本章后面的“`open()` 系统调用”一节）。

`flush(file)`

当打开文件的引用被关闭时调用该方法。该方法的实际用途取决于文件系统。

`release(inode, file)`

释放文件对象。当打开文件的最后一个引用被关闭时（即文件对象 `f_count` 字段的值变为 0 时）调用该方法。

`fsync(file, dentry, flag)`

将文件所缓存的全部数据写入磁盘。

`aio_fsync(req, flag)`

启动一次异步 I/O 刷新操作。

`fasync(fd, file, on)`

通过信号来启用或禁止 I/O 事件通告。

`lock(file, cmd, file_lock)`

对 `file` 文件申请一个锁（参见本章后面的“文件加锁”一节）。

`readv(file, vector, count, offset)`

从文件中读字节，并把结果放入 `vector` 描述的缓冲区中；缓冲区的个数由 `count` 指定。

`writev(file, vector, count, offset)`

把 `vector` 描述的缓冲区中的字节写入文件；缓冲区的个数由 `count` 指定。

`sendfile(in_file, offset, count, file_send_actor, out_file)`

把数据从 `in_file` 传送到 `out_file`（引入它是为了支持 `sendfile()` 系统调用）。

`sendpage(file, page, offset, size, pointer, fill)`

把数据从文件传送到页高速缓存的页；这个低层方法由 `sendfile()` 和用于套接字的网络代码使用。

`get_unmapped_area(file, addr, len, offset, flags)`

获得一个未用的地址范围来映射文件。

`check_flags(flags)`

当设置文件的状态标志（`F_SETFL` 命令）时，`fcntl()` 系统调用的服务例程调用该方法执行附加的检查。当前只适用于 NFS 网络文件系统。

`dir_notify(file, arg)`

当建立一个目录更改通告（`F_NOTIFY` 命令）时，由 `fcntl()` 系统调用的服务例程调用该方法。当前只适用于 CIFS（Common Internet File system，公用互联网文件系统）网络文件系统。


```
flock(file, flag, lock)
```

用于定制 `flock()` 系统调用的行为。官方 Linux 文件系统不使用该方法。

以上描述的方法对所有可能的文件类型都是可用的。不过，对于一个具体的文件类型，只使用其中的一个子集；那些未实现的方法对应的字段被置为 `NULL`。

目录项对象

在“通用文件模型”一节中我们曾提到，VFS 把每个目录看作由若干子目录和文件组成的一个普通文件。在第十八章我们将会讨论如何在具体的文件系统上实现目录。然而，一旦目录项被读入内存，VFS 就把它转换成基于 `dentry` 结构的一个目录项对象，该结构的字段如表 12-5 所示。对于进程查找的路径名中的每个分量，内核都为其创建一个目录项对象；目录项对象将每个分量与其对应的索引节点相联系。例如，在查找路径名 `/tmp/test` 时，内核为根目录 `/` 创建一个目录项对象，为根目录下的 `tmp` 项创建一个第二级目录项对象，为 `/tmp` 目录下的 `test` 项创建一个第三级目录项对象。

请注意，目录项对象在磁盘上并没有对应的映像，因此在 `dentry` 结构中不包含指出该对象已被修改的字段。目录项对象存放在名为 `dentry_cache` 的 slab 分配器高速缓存中。因此，目录项对象的创建和删除是通过调用 `kmem_cache_alloc()` 和 `kmem_cache_free()` 实现的。

表 12-5: 目录项对象的字段

类型	字段	说明
<code>atomic_t</code>	<code>d_count</code>	目录项对象引用计数器
<code>unsigned int</code>	<code>d_flag</code>	目录项高速缓存标志
<code>spinlock_t</code>	<code>d_lock</code>	保护目录项对象的自旋锁
<code>struct inode *</code>	<code>d_inode</code>	与文件名关联的索引节点
<code>struct dentry *</code>	<code>d_parent</code>	父目录的目录项对象
<code>struct qstr</code>	<code>d_name</code>	文件名
<code>struct list_head</code>	<code>d_lru</code>	用于未使用目录项链表的指针
<code>struct list_head</code>	<code>d_child</code>	对目录而言，用于同一父目录中的目录项链表的指针
<code>struct list_head</code>	<code>d_subdirs</code>	对目录而言，子目录项链表的头
<code>struct list_head</code>	<code>d_alias</code>	用于与同一索引节点（别名）相关的目录项链表的指针

表12-5: 目录项对象的字段

类型	字段	说明
unsigned long	d_time	由 d_revalidate 方法使用
struct dentry_operations*	d_op	目录项方法
struct super_block *	d_sb	文件的超级块对象
void *	d_fsdata	依赖于文件系统的数据库
struct rcu_head	d_rcu	回收目录项对象时, 由 RCU 描述符使用 (参见第五章的“读-拷贝-更新 (RCU)”一节)
struct dcookie_struct *	d_cookie	指向内核配置文件使用的数据结构指针
struct hlist_node	d_hash	指向散列表表项链表的指针
int	d_mounted	对目录而言, 用于记录安装该目录项的文件系统数的计数器
unsigned char[]	d_iname	存放短文件名的空间

每个目录项对象可以处于以下四种状态之一:

空闲状态 (*free*)

处于该状态的目录项对象不包括有效的信息, 且还没有被 VFS 使用。对应的内存区由 slab 分配器进行处理。

未使用状态 (*unused*)

处于该状态的目录项对象当前还没有被内核使用。该对象的引用计数器 d_count 的值为 0, 但其 d_inode 字段仍然指向关联的索引节点。该目录项对象包含有效的信息, 但为了在必要时回收内存, 它的内容可能被丢弃。

正在使用状态 (*in use*)

处于该状态的目录项对象当前正在被内核使用。该对象的引用计数器 d_count 的值为正数, 其 d_inode 字段指向关联的索引节点对象。该目录项对象包含有效的信息, 并且不能被丢弃。

负状态 (*negative*)

与目录项关联的索引节点不复存在, 那是因为相应的磁盘索引节点已被删除, 或者因为目录项对象是通过解析一个不存在文件的路径名创建的。目录项对象的 d_inode 字段被置为 NULL, 但该对象仍然被保存在目录项高速缓存中, 以便后续对同一文件目录名的查找操作能够快速完成。术语“负状态”容易使人误解, 因为根本不涉及任何负值。

与目录项对象关联的方法称为目录项操作。这些方法由 `dentry_operations` 结构加以描述，该结构的地址存放在目录项对象的 `d_op` 字段中。尽管一些文件系统定义了它们自己的目录项方法，但是这些字段通常为 `NULL`，而 VFS 使用缺省函数代替这些方法。以下按照其在 `dentry_operations` 表中出现的顺序来列举一些方法。

`d_revalidate(dentry, nameidata)`

在把目录项对象转换为一个文件路径名之前，判定该目录项对象是否仍然有效。缺省的 VFS 函数什么也不做，而网络文件系统可以指定自己的函数。

`d_hash(dentry, name)`

生成一个散列值，这是用于目录项散列表的、特定于具体文件系统的散列函数。参数 `dentry` 标识包含路径分量的目录。参数 `name` 指向一个结构，该结构包含要查找的路径名分量以及由散列函数生成的散列值。

`d_compare(dir, name1, name2)`

比较两个文件名。 `name1` 应该属于 `dir` 所指的目录。缺省的 VFS 函数是常用的字符串匹配函数。不过，每个文件系统可用自己的方式实现这一方法。例如，MS-DOS 文件系统不区分大写和小写字母。

`d_delete(dentry)`

当对目录项对象的最后一个引用被删除（`d_count` 变为“0”）时，调用该方法。缺省的 VFS 函数什么也不做。

`d_release(dentry)`

当要释放一个目录项对象时（放入 slab 分配器），调用该方法。缺省的 VFS 函数什么也不做。

`d_iput(dentry, ino)`

当一个目录项对象变为“负”状态（即丢弃它的索引节点）时，调用该方法。缺省的 VFS 函数调用 `iput()` 释放索引节点对象。

目录项高速缓存

由于从磁盘读入一个目录项并构造相应的目录项对象需要花费大量的时间，所以，在完成对目录项对象的操作后，可能后面还要使用它，因此仍在内存中保留它有重要的意义。例如，我们经常需要编辑文件，随后编译它，或者编辑并打印它，或者复制它并编辑这个拷贝，在诸如此类的情况中，同一个文件需要被反复访问。

为了最大限度地提高处理这些目录项对象的效率，Linux 使用目录项高速缓存，它由两种类型的数据结构组成：

- 一个处于正在使用、未使用或负状态的目录项对象的集合。
- 一个散列表，从中能够快速获取与给定的文件名和目录名对应的目录项对象。同样，如果访问的对象不在目录项高速缓存中，则散列函数返回一个空值。

目录项高速缓存的作用还相当于索引节点高速缓存 (*inode cache*) 的控制器。在内核内存中，并不丢弃与未用目录项相关的索引节点，这是由于目录项高速缓存仍在使用它们。因此，这些索引节点对象保存在 RAM 中，并能够借助相应的目录项快速引用它们。

所有“未使用”目录项对象都存放在一个“最近最少使用 (Least Recently used, LRU)”的双向链表中，该链表按照插入的时间排序。换句话说，最后释放的目录项对象放在链表的首部，所以最近最少使用的目录项对象总是靠近链表的尾部。一旦目录项高速缓存的空间开始变小，内核就从链表的尾部删除元素，使得最近最常使用的对象得以保留。LRU 链表的首元素和尾元素的地址存放在 `list_head` 类型的 `dentry_unused` 变量的 `next` 字段和 `prev` 字段中。目录项对象的 `d_lru` 字段包含指向链表中相邻目录项的指针。

每个“正在使用”的目录项对象都被插入一个双向链表中，该链表由相应索引节点对象的 `i_dentry` 字段所指向（由于每个索引节点可能与若干硬链接关联，所以需要—个链表）。目录项对象的 `d_alias` 字段存放链表中相邻元素的地址。这两个字段的类型都是 `struct list_head`。

当指向相应文件的最后一个硬链接被删除后，一个“正在使用”的目录项对象可能变成“负”状态。在这种情况下，该目录项对象被移到“未使用”目录项对象组成的 LRU 链表中。每当内核缩减目录项高速缓存时，“负”状态目录项对象就朝着 LRU 链表的尾部移动，这样一来，这些对象就逐渐被释放（参见第十七章中的“回收可压缩磁盘高速缓存的页”一节）。

散列表是由 `dentry_hashtable` 数组实现的。数组中的每个元素是一个指向链表的指针，这种链表就是把具有相同散列表值的目录项进行散列而形成的。该数组的长度取决于系统已安装 RAM 的数量，缺省值是每兆字节 RAM 包含 256 个元素。目录项对象的 `d_hash` 字段包含指向具有相同散列值的链表中的相邻元素。散列函数产生的值是由目录的目录项对象及文件名计算出来的。

`dcache_lock` 自旋锁保护目录项高速缓存数据结构免受多处理器系统上的同时访问。`d_lookup()` 函数在散列表中查找给定的父目录项对象和文件名，为了避免发生竞争，使用顺序锁 (`seqlock`)（参见第五章中的“顺序锁”一节）。`__d_lookup()` 函数与之类似，不过它假定不会发生竞争，因此不使用顺序锁。

与进程相关的文件

在第一章的“Unix 文件系统概述”一节中我们曾提到，每个进程都有它自己当前的工作目录和它自己的根目录。这仅仅是内核用来表示进程与文件系统相互作用所必须维护的数据中的两个例子。类型为 `fs_struct` 的整个数据结构就用于此目的（参见表 12-6），且每个进程描述符的 `fs` 字段就指向进程的 `fs_struct` 结构。

表 12-6: `fs_struct` 结构中的字段

类型	字段	说明
<code>atomic_t</code>	<code>count</code>	共享这个表的进程个数
<code>rwlock_t</code>	<code>lock</code>	用于表中字段的读 / 写自旋锁
<code>Int</code>	<code>umask</code>	当打开文件设置文件权限时所使用的位掩码
<code>struct dentry *</code>	<code>root</code>	根目录的目录项
<code>struct dentry *</code>	<code>pwd</code>	当前工作目录的目录项
<code>struct dentry *</code>	<code>altroot</code>	模拟根目录的目录项（在 80x86 结构上始终为 NULL）
<code>struct vfsmount *</code>	<code>rootmnt</code>	根目录所安装的文件系统对象
<code>struct vfsmount *</code>	<code>pwdmnt</code>	当前工作目录所安装的文件系统对象
<code>struct vfsmount *</code>	<code>altrootmnt</code>	模拟根目录所安装的文件系统对象（在 80x86 结构上始终为 NULL）

第二个表表示进程当前打开的文件，表的地址存放于进程描述符的 `files` 字段。该表的类型为 `files_struct` 结构，它的各个字段如表 12-7 所示。

表 12-7: `files_struct` 结构中的字段

类型	字段	说明
<code>atomic_t</code>	<code>count</code>	共享该表的进程数目
<code>rwlock_t</code>	<code>file_lock</code>	用于表中字段的读 / 写自旋锁
<code>Int</code>	<code>max_fds</code>	文件对象的当前最大数目
<code>Int</code>	<code>max_fdset</code>	文件描述符的当前最大数目
<code>Int</code>	<code>next_fd</code>	所分配的最大文件描述符加 1
<code>struct file **</code>	<code>fd</code>	指向文件对象指针数组的指针
<code>fd_set *</code>	<code>close_on_exec</code>	指向执行 <code>exec()</code> 时需要关闭的文件描述符的指针

表 12-7: files_struct 结构中的字段 (续)

类型	字段	说明
fd_set *	open_fds	指向打开文件描述符的指针
fd_set	close_on_exec_init	执行 exec() 时需要关闭的文件描述符的初始集合
fd_set	open_fds_init	文件描述符的初始集合
struct file *[]	fd_array	文件对象指针的初始化数组

fd 字段指向文件对象的指针数组。该数组的长度存放在 max_fds 字段中。通常, fd 字段指向 files_struct 结构的 fd_array 字段, 该字段包括 32 个文件对象指针。如果进程打开的文件数目多于 32, 内核就分配一个新的、更大的文件指针数组, 并将其地址存放在 fd 字段中, 内核同时也更新 max_fds 字段的值。

对于在 fd 数组中有元素的每个文件来说, 数组的索引就是文件描述符 (*file descriptor*)。通常, 数组的第一个元素 (索引为 0) 是进程的标准输入文件, 数组的第二个元素 (索引为 1) 是进程的标准输出文件, 数组的第三个元素 (索引为 2) 是进程的标准错误文件 (参见图 12-3)。Unix 进程将文件描述符作为主文件标识符。请注意, 借助于 dup(), dup2() 和 fcntl() 系统调用, 两个文件描述符可以指向同一个打开的文件, 也就是说, 数组的两个元素可能指向同一个文件对象。当用户使用 shell 结构 (如 2>&1) 将标准错误文件重定向到标准输出文件上时, 用户总能看到这一点。

进程不能使用多于 NR_OPEN (通常为 1 048 576) 个文件描述符。内核也在进程描述符的 signal->rlim[RLIMIT_NOFILE] 结构上强制动态限制文件描述符的最大数; 这个值通常为 1024, 但是如果进程具有超级用户特权, 就可以增大这个值。

open_fds 字段最初包含 open_fds_init 字段的地址, open_fds_init 字段表示当前已打开文件的文件描述符的位图。max_fdset 字段存放位图中的位数。由于 fd_set 数据结构有 1024 位, 所以通常不需要扩大位图的大小。不过, 如果确有必要的话, 内核仍能动态增加位图的大小, 这非常类似于文件对象的数组的情形。

当内核开始使用一个文件对象时, 内核提供 fget() 函数以供调用。这个函数接收文件描述符 fd 作为参数, 返回在 current->files->fd[fd] 中的地址, 即对应文件对象的地址, 如果没有任何文件与 fd 对应, 则返回 NULL。在第一种情况下, fget() 使文件对象引用计数器 f_count 的值增 1。

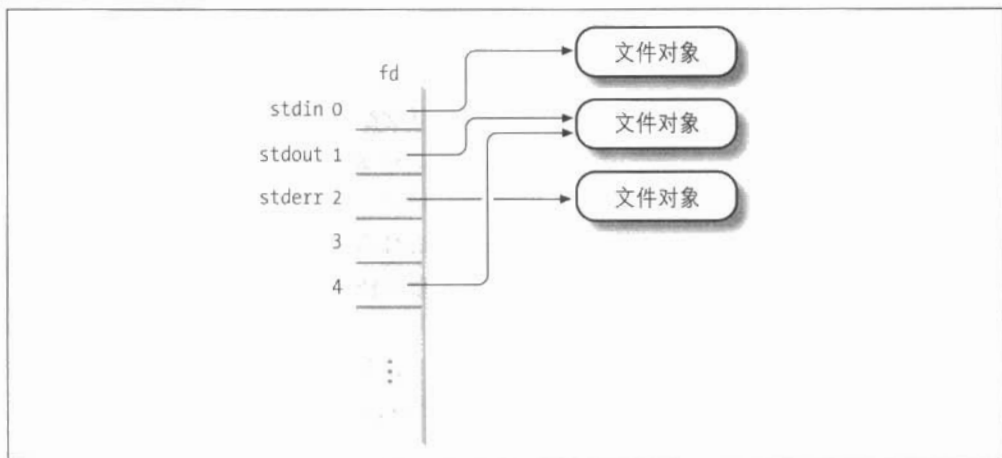


图 12-3: fd 数组

当内核控制路径完成对文件对象的使用时，调用内核提供的 `fput()` 函数。该函数将文件对象的地址作为参数，并减少文件对象引用计数器 `f_count` 的值。另外，如果这个字段变为 0，该函数就调用文件操作的 `release` 方法（如果已定义），减少索引节点对象的 `i_write count` 字段的值（如果该文件是可写的），将文件对象从超级块链表中移走，释放文件对象给 slab 分配器，最后减少相关的文件系统描述符的目录项对象的引用计数器的值（参见“文件系统安装”一节）。

`fget_light()` 和 `fget_light()` 函数是 `fget()` 和 `fput()` 的快速版本：内核要使用它们，前提是能够安全地假设当前进程已经拥有文件对象，即进程先前已经增加了文件对象引用计数器的值。例如，它们由接收一个文件描述符作为参数的系统调用服务例程使用，这是由于先前的 `open()` 系统调用已经增加了文件对象引用计数器的值。

文件系统类型

Linux 内核支持很多不同的文件系统类型。在下面的内容中，我们介绍一些特殊的文件系统类型，它们在 Linux 内核的内部设计中具有非常重要的作用。

接下来，我们将讨论文件系统注册——也就是通常在系统初始化期间并且在使用文件系统类型之前必须执行的基本操作。一旦文件系统被注册，其特定的函数对内核就是可用的，因此文件系统类型可以安装在系统的目录树上。

特殊文件系统

当网络和磁盘文件系统能够使用户处理存放在内核之外的信息时,特殊文件系统可以为系统程序员和管理员提供一种容易的方式来操作内核的数据结构并实现操作系统的特殊特征。表12-8列出了Linux中所用的最常用的特殊文件系统,对于其中的每个文件系统,表中给出了它的安装点和简短描述。

注意,有几个文件系统没有固定的安装点(表中的关键词“任意”)。这些文件系统可以由用户自由地安装和使用。此外,一些特殊文件系统根本没有安装点(表中的关键词“无”)。它们不是用于与用户交互,但是内核可以用它们来很容易地重新使用VFS层的某些代码;例如,我们在第十九章会看到,有了pipefs特殊文件系统,就可以把管道和FIFO文件以相同的方式对待。

表12-8: 最常用的特殊文件系统

名字	安装点	说明
<i>bdev</i>	无	块设备(参见第十三章)
<i>binfmt_misc</i>	任意	其他可执行格式(参见第二十章)
<i>devpts</i>	<i>/dev/pts</i>	伪终端支持(开放组织的Unix98标准)
<i>eventpollfs</i>	无	由有效事件轮询机制使用
<i>futextfs</i>	无	由futext(快速用户空间加锁)机制使用
<i>pipefs</i>	无	管道(参见第十九章)
<i>proc</i>	<i>/proc</i>	对内核数据结构的常规访问点
<i>rootfs</i>	无	为启动阶段提供一个空的根目录
<i>shm</i>	无	IPC共享线性区(参见第十九章)
<i>mqueue</i>	任意	实现POSIX消息队列时使用(参见第十九章)
<i>sockfs</i>	无	套接字
<i>sysfs</i>	<i>/sys</i>	对系统数据的常规访问点(参见第十三章)
<i>tmpfs</i>	任意	临时文件(如果不被交换出去就保持在RAM中)
<i>usbfs</i>	<i>/proc/bus/usb</i>	USB设备

特殊文件系统不限于物理块设备。然而,内核给每个安装的特殊文件系统分配一个虚拟的块设备,让其主设备号为0而次设备号具有任意值(每个特殊文件系统有不同的值)。`set_anon_super()`函数用于初始化特殊文件系统的超级块;该函数本质上获得一个未使用的次设备号dev,然后用主设备号0和次设备号dev设置新超级块的s_dev字段。而另一个`kill_anon_super()`函数移走特殊文件系统的超级块。`unnamed_dev_idr`变

量包含指向一个辅助结构（记录当前在用的次设备号）的指针。尽管有些内核设计者不喜欢虚拟块设备标识符，但是这些标识符有助于内核以统一的方式处理特殊文件系统和普通文件系统。

我们在后面“安装普通文件系统”一节会看到内核如何定义和初始化一个特殊文件系统的实际例子。

文件系统类型注册

通常，用户在为自己的系统编译内核时可以把Linux配置为能够识别所有需要的文件系统。但是，文件系统的源代码实际上要么包含在内核映像中，要么作为一个模块被动态装入（参见附录二）。VFS必须对代码目前已在内核中的所有文件系统的类型进行跟踪。这就是通过进行文件系统类型注册来实现的。

每个注册的文件系统都用一个类型为`file_system_type`的对象来表示，该对象的所有字段在表12-9中列出。

表12-9: `file_system_type`对象的字段

类型	字段	说明
<code>const char *</code>	<code>name</code>	文件系统名
<code>int</code>	<code>fs_flags</code>	文件系统类型标志
<code>struct super_block * (*)()</code>	<code>get_sb</code>	读超级块的方法
<code>void (*)()</code>	<code>kill_sb</code>	删除超级块的方法
<code>struct module *</code>	<code>owner</code>	指向实现文件系统的模块的指针(参见附录二)
<code>struct file_system_type *</code>	<code>next</code>	指向文件系统类型链表中下一个元素的指针
<code>struct list_head</code>	<code>fs_supers</code>	具有相同文件系统类型的超级块对象链表的头

所有文件系统类型的对象都插入到一个单向链表中。由变量`file_systems`指向链表的第一个元素，而结构中的`next`字段指向链表的下一个元素。`file_systems_lock`读/写自旋锁保护整个链表免受同时访问。

`fs_supers`字段表示给定类型的已安装文件系统所对应的超级块链表的头（第一个伪元素）。链表元素的向后和向前链接存放在超级块对象的`s_instances`字段中。

`get_sb`字段指向依赖于文件系统类型的函数，该函数分配一个新的超级块对象并初始化它（如果需要，可读磁盘）。而`kill_sb`字段指向删除超级块的函数。

`fs_flags`字段存放几个标志，如表12-10所示。

表12-10: 文件系统类型的标志

名字	说明
FS_REQUIRES_DEV	这种类型的任何文件系统必须位于物理磁盘设备上
FS_BINARY_MOUNTDATA	文件系统使用的二进制安装数据
FS_REVAL_DOT	始终在目录项高速缓存中使“.”和“..”路径重新生效（针对网络文件系统）
FS_ODD_RENAME	“重命名”操作就是“移动”操作（针对网络文件系统）

在系统初始化期间，调用 `register_filesystem()` 函数来注册编译时指定的每个文件系统；该函数把相应的 `file_system_type` 对象插入到文件系统类型的链表中。

当实现了文件系统的模块被装入时，也要调用 `register_filesystem()` 函数。在这种情况下，当该模块被卸载时，对应的文件系统也可以被注销（调用 `unregister_filesystem()` 函数）。

`get_fs_type()` 函数（接收文件系统名作为它的参数）扫描已注册的文件系统链表以查找文件系统类型的 `name` 字段，并返回指向相应的 `file_system_type` 对象（如果存在）的指针。

文件系统处理

就像每个传统的Unix系统一样，Linux也使用系统的根文件系统（*system's root filesystem*）：它由内核在引导阶段直接安装，并拥有系统初始化脚本以及最基本的系统程序。

其他文件系统要么由初始化脚本安装，要么由用户直接安装在已安装文件系统的目录上。作为一个目录树，每个文件系统都拥有自己的根目录（*root directory*）。安装文件系统的这个目录称之为安装点（*mount point*）。已安装文件系统属于安装点目录的一个子文件系统。例如，`/proc` 虚拟文件系统是系统的根文件系统的孩子（且系统的根文件系统是 `/proc` 的父亲）。已安装文件系统的根目录隐藏了父文件系统的安装点目录原来的内容，而且父文件系统的整个子树位于安装点之下（注5）。

注5： 文件系统的根目录有可能不同于进程的根目录：正如我们在前面“与文件相关的进程”一节所见，进程的根目录是与“/”路径对应的目录。缺省情况下，进程的根目录与系统的根文件系统的根目录一致（更准确地说是与进程的命名空间中的根文件系统的根目录一致，这一点将在下一节描述），但是可以通过调用 `chroot()` 系统调用改变进程的根目录。

命名空间

在传统的 Unix 系统中，只有一个已安装文件系统树：从系统的根文件系统开始，每个进程通过指定合适的路径名可以访问已安装文件系统中的任何文件。从这个方面考虑，Linux 2.6 更加的精确：每个进程可拥有自己的已安装文件系统树——叫做进程的命名空间 (*namespace*)。

通常大多数进程共享同一个命名空间，即位于系统的根文件系统且被 *init* 进程使用的已安装文件系统树。不过，如果 `clone()` 系统调用以 `CLONE_NEWNS` 标志创建一个新进程，那么进程将获取一个新的命名空间（参见第三章的“`clone()`、`fork()`及`vfork()`系统调用”一节）。这个新的命名空间随后由子进程继承（如果父进程没有以 `CLONE_NEWNS` 标志创建这些子进程）。

当进程安装或卸载一个文件系统时，仅修改它的命名空间。因此，所做的修改对共享同一命名空间的所有进程都是可见的，并且也只对它们可见。进程甚至可通过使用 Linux 特有的 `pivot_root()` 系统调用来改变它的命名空间的根文件系统。

进程的命名空间由进程描述符的 `namespace` 字段指向的 `namespace` 结构描述。该结构的字段如表 12-11 所示。

表 12-11: namespace 结构中的字段

类型	字段	说明
<code>atomic_t</code>	<code>count</code>	引用计数器（共享命名空间的进程数）
<code>struct vfsmount *</code>	<code>root</code>	命名空间根目录的已安装文件系统描述符
<code>struct list_head</code>	<code>list</code>	所有已安装文件系统描述符链表的头
<code>struct rw_semaphore</code>	<code>sem</code>	保护这个结构的读/写信号量

`list` 字段是双向循环链表的头，该表聚集了属于命名空间的所有已安装文件系统。`root` 字段表示已安装文件系统，它是这个命名空间的已安装文件系统树的根。正如我们在下一节将看到的，已安装文件系统由 `vfsmount` 结构描述。

文件系统安装

在大多数传统的类 Unix 内核中，每个文件系统只能安装一次。假定存放在 `/dev/fd0` 软磁盘上的 Ext2 文件系统通过如下命令安装在 `/flp`：

```
mount -t ext2 /dev/fd0 /flp
```

在用 `umount` 命令卸载该文件系统前，所有其他作用于 `/dev/fd0` 的安装命令都会失败。

然而，Linux 有所不同：同一个文件系统被安装多次是可能的。当然，如果一个文件系统被安装了 n 次，那么它的根目录就可通过 n 个安装点来访问。尽管同一文件系统可以通过不同的安装点来访问，但是文件系统的的确是唯一的。因此，不管一个文件系统被安装了多少次，都仅有一个超级块对象。

安装的文件系统形成一个层次：一个文件系统的安装点可能成为第二个文件系统的目录，而第二个文件系统又安装在第三个文件系统之上，等等（注 6）。

把多个安装堆叠在一个单独的安装点上也是可能的。尽管已经使用先前安装下的文件和目录的进程可以继续使用，但在同一安装点上的新安装隐藏前一个安装的文件系统。当最顶层的安装被删除时，下一层的安装再一次变为可见的。

你可以想像，跟踪已安装的文件系统很快会变为一场恶梦。对于每个安装操作，内核必须在内存中保存安装点和安装标志，以及要安装文件系统与其他已安装文件系统之间的关系。这样的信息保存在已安装文件系统描述符中；每个描述符是一个具有 `vfsmount` 类型的数据结构，其字段如表 12-12 所示。

表 12-12: `vfsmount` 数据结构中的字段

类型	字段	说明
<code>struct list_head</code>	<code>mnt_hash</code>	用于散列表链表的指针
<code>struct vfsmount *</code>	<code>mnt_parent</code>	指向父文件系统，这个文件系统安装在其上
<code>struct dentry *</code>	<code>mnt_mountpoint</code>	指向这个文件系统安装点目录的 <code>dentry</code>
<code>struct dentry *</code>	<code>mnt_root</code>	指向这个文件系统根目录的 <code>dentry</code>
<code>struct super_block *</code>	<code>mnt_sb</code>	指向这个文件系统的超级块对象
<code>struct list_head</code>	<code>mnt_mounts</code>	包含所有文件系统描述符链表的头（相对于这个文件系统）
<code>struct list_head</code>	<code>mnt_child</code>	用于已安装文件系统链表 <code>mnt_mounts</code> 的指针
<code>atomic_t</code>	<code>mnt_count</code>	引用计数器（增加该值以禁止文件系统被卸载）
<code>Int</code>	<code>mnt_flags</code>	标志

注 6：令人非常惊讶的是，一个文件系统的安装点可能就是这同一文件系统中的—个目录，假定这个文件系统以前已经安装。例如：

```
mount -t ext2 /dev/fd0/flp; touch /flp/foo
mkdir /flp/mnt; mount -t ext2 /dev/fd0/flp/mnt
```

现在，软盘文件系统上的空 `foo` 文件就可以通过 `/flp/foo` 和 `/flp/mnt/foo` 来访问。

表 12-12: vfstmount 数据结构中的字段 (续)

类型	字段	说明
Int	<code>mnt_expiry_mark</code>	如果文件系统标记为到期, 那么就设置该标志为 true (如果设置了该标志, 并且没有任何人使用它, 那么就可以自动卸载这个文件系统)
char *	<code>mnt_devname</code>	设备文件名
struct list_head	<code>mnt_list</code>	已安装文件系统描述符的 namespace 链表的指针
struct list_head	<code>mnt_fslink</code>	具体文件系统到期链表的指针
struct namespace *	<code>mnt_namespace</code>	指向安装了文件系统的进程命名空间的指针

vfstmount 数据结构保存在几个双向循环链表中:

- 由父文件系统 vfstmount 描述符的地址和安装点目录的目录项对象的地址索引的散列表。散列表存放在 `mount_hashtable` 数组中, 其大小取决于系统中 RAM 的容量。表中每一项是具有同一散列值的所有描述符形成的双向循环链表的头。描述符的 `mnt_hash` 字段包含指向链表中相邻元素的指针。
- 对于每一个命名空间, 所有属于此命名空间的已安装的文件系统描述符形成了一个双向循环链表。namespace 结构的 `list` 字段存放链表的头, vfstmount 描述符的 `mnt_list` 字段包含链表中指向相邻元素的指针。
- 对于每一个已安装的文件系统, 所有已安装子文件系统形成了一个双向循环链表。每个链表的头存放在已安装的文件系统描述符的 `mnt_mounts` 字段; 此外, 描述符的 `mnt_child` 字段存放指向链表中相邻元素的指针。

`vfstmount_lock` 自旋锁保护已安装文件系统对象的链表免受同时访问。

描述符的 `mnt_flags` 字段存放几个标志的值, 用以指定如何处理已安装文件系统中的某些种类的文件。这些标志可通过 `mount` 命令的选项进行设置, 其标志如表 12-13 所示。

表 12-13: 已安装文件系统中的标志

名字	说明
<code>MNT_NOSUID</code>	在已安装文件系统中禁止 <code>setuid</code> 和 <code>setgid</code> 标志
<code>MNT_NODEV</code>	在已安装文件系统中禁止访问设备文件
<code>MNT_NOEXEC</code>	在已安装文件系统中不允许程序执行

下列函数处理已安装文件系统描述符:

`alloc_vfsmnt (name)`

分配和初始化一个已安装文件系统描述符。

`free_vfsmnt (mnt)`

释放由 `mnt` 指向的已安装文件系统描述符。

`lookup_mnt (mnt, dentry)`

在散列表中查找一个描述符并返回它的地址。

安装普通文件系统

我们现在描述安装一个文件系统时内核所要执行的操作。我们首先考虑一个文件系统将被安装在一个已安装文件系统之上的情形（在这里我们把这种新文件系统看作“普通的”）。

`mount()` 系统调用被用来安装一个普通文件系统；它的服务例程 `sys_mount()` 作用于以下参数：

- 文件系统所在的设备文件的路径名，或者如果不需要的话就为 `NULL`（例如，当要安装的文件系统是基于网络时）
- 文件系统被安装其上的某个目录的目录路径名（安装点）
- 文件系统的类型，必须是已注册文件系统的名字
- 安装标志（所允许的值如表 12-14 所示）
- 指向一个与文件系统相关的数据结构的指针（也许为 `NULL`）

表 12-14: `mount()` 系统调用使用的安装标志

宏	说明
<code>MS_RDONLY</code>	文件只能被读
<code>MS_NOSUID</code>	禁止 <code>setuid</code> 和 <code>setgid</code> 标志
<code>MS_NODEV</code>	禁止访问设备文件
<code>MS_NOEXEC</code>	不允许程序执行
<code>MS_SYNCHRONOUS</code>	文件和目录上的写操作是即时的
<code>MS_REMOUNT</code>	重新安装改变了安装标志的文件系统
<code>MS_MANDLOCK</code>	允许强制加锁
<code>MS_DIRSYNC</code>	目录上的写操作是即时的
<code>MS_NOATIME</code>	不更新文件访问时间

表 12-14: mount()系统调用使用的安装标志 (续)

宏	说明
MS_NODIRATIME	不更新目录访问时间
MS_BIND	创建一个“绑定安装”，这就使得一个文件或目录在系统目录树的另外一个点上可以看得见 (mount 命令的 <code>__bind</code> 选项)
MS_MOVE	自动把一个已安装文件系统移动到另一个安装点 (mount 命令的 <code>__move</code> 选项)
MS_REC	为目录子树递归地创建“绑定安装”
MS_VERBOSE	在安装出错时产生内核消息

`sys_mount()` 函数把参数的值拷贝到临时内核缓冲区，获取大内核锁，并调用 `do_mount()` 函数。一旦 `do_mount()` 返回，则这个服务例程释放大内核锁并释放临时内核缓冲区。

`do_mount()` 函数通过执行下列操作处理真正的安装操作：

1. 如果安装标志 `MS_NOSUID`、`MS_NODEV` 或 `MS_NOEXEC` 中任一被设置，则清除它们，并在已安装文件系统对象中设置相应的标志 (`MNT_NOSUID`、`MNT_NODEV`、`MNT_NOEXEC`)。
2. 调用 `path_lookup()` 查找安装点的路径名；该函数把路径名查找的结果存放在 `nameidata` 类型的局部变量 `nd` 中（参见后面的“路径名查找”一节）。
3. 检查安装标志以决定必须做什么。尤其是：
 - a. 如果 `MS_REMOUNT` 标志被指定，其目的通常是改变超级块对象 `s_flags` 字段的安装标志，以及已安装文件系统对象 `mnt_flags` 字段的安装文件系统标志。`do_remount()` 函数执行这些改变。
 - b. 否则，检查 `MS_BIND` 标志。如果它被指定，则用户要求在在系统目录树的另一个安装点上的文件或目录能够可见。
 - c. 否则，检查 `MS_MOVE` 标志。如果它被指定，则用户要求改变已安装文件系统的安装点。`do_move_mount()` 函数原子地完成这一任务。
 - d. 否则，调用 `do_new_mount()`。这是最普通的情况。当用户要求安装一个特殊文件系统或存放在磁盘分区中的普通文件系统时，触发该函数。它调用 `do_kern_mount()` 函数，给它传递的参数为文件系统类型、安装标志以及块设备名。`do_kern_mount()` 处理实际的安装操作并返回一个新安装文件系统描述符的地址（如下描述）。然后，`do_new_mount()` 调用 `do_add_mount()`，后者本质上执行下列操作：

- (1) 获得当前进程的写信号量 `namespace->sem`，因为函数要更改 `namespace` 结构。
 - (2) `do_kern_mount()` 函数可能让当前进程睡眠；同时，另一个进程可能在完全相同的安装点上安装文件系统或者甚至更改根文件系统(`current->namespace->root`)。验证在该安装点上最近安装的文件系统是否仍指向当前的 `namespace`；如果不是，则释放读 / 写信号量并返回一个错误码。
 - (3) 如果要安装的文件系统已经被安装在由系统调用的参数所指定的安装点上，或该安装点是一个符号链接，则释放读 / 写信号量并返回一个错误码。
 - (4) 初始化由 `do_kern_mount()` 分配的新安装文件系统对象的 `mnt_flags` 字段的标志。
 - (5) 调用 `graft_tree()` 把新安装的文件系统对象插入到 `namespace` 链表、散列表及父文件系统的子链表中。
 - (6) 释放 `namespace->sem` 读 / 写信号量并返回。
4. 调用 `path_release()` 终止安装点的路径名查找（参见后面的“路径名查找”一节）并返回 0。

`do_kern_mount()` 函数

安装操作的核心是 `do_kern_mount()` 函数，它检查文件系统类型标志以决定安装操作是如何完成的。该函数接收下列参数：

`fstype`

要安装的文件系统的类型名。

`flags`

安装标志（参见表 12-14）。

`name`

存放文件系统的块设备的路径名（或特殊文件系统的类型名）。

`data`

指向传递给文件系统的 `read_super` 方法的附加数据的指针。

本质上，该函数通过执行下列操作实现实际的安装操作：

1. 调用 `get_fs_type()` 在文件系统类型链表中搜索并确定存放在 `fstype` 参数中的名字的位置；返回局部变量 `type` 中对应 `file_system_type` 描述符的地址。

2. 调用 `alloc_vfsmnt()` 分配一个新的已安装文件系统的描述符, 并将它的地址存放在 `mnt` 局部变量中。
3. 调用依赖于文件系统的 `type->get_sb()` 函数分配, 并初始化一个新的超级块 (参见下面)。
4. 用新超级块对象的地址初始化 `mnt->mnt_sb` 字段。
5. 将 `mnt->mnt_root` 字段初始化为与文件系统根目录对应的目录项对象的地址, 并增加该目录项对象的引用计数器值。
6. 用 `mnt` 中的值初始化 `mnt->mnt_parent` 字段 (对于普通文件系统, 当 `graft_tree()` 把已安装文件系统的描述符插入到合适的链表中时, 要把 `mnt_parent` 字段置为合适的值; 参见 `do_mount()` 的第 3d5 步)。
7. 用 `current->namespace` 中的值初始化 `mnt->mnt_namespace` 字段。
8. 释放超级块对象的读/写信号量 `s_umount` (在第 3 步中分配对象时获得)。
9. 返回已安装文件系统对象的地址 `mnt`。

分配超级块对象

文件系统对象的 `get_sb` 方法通常是由单行函数实现的。例如, 在 Ext2 文件系统中该方法的实现如下:

```
struct super_block * ext2_get_sb(struct file_system_type *type,
                                int flags, const char *dev_name, void *data)
{
    return get_sb_bdev(type, flags, dev_name, data, ext2_fill_super);
}
```

`get_sb_bdev()` VFS 函数分配并初始化一个新的适合于磁盘文件系统的超级块; 它接收 `ext2_fill_super()` 函数的地址, 该函数从 Ext2 磁盘分区读取磁盘超级块。

为了分配适合于特殊文件系统的超级块, VFS 也提供 `get_sb_pseudo()` 函数 (对于没有安装点的特殊文件系统, 例如 `pipefs`)、`get_sb_single()` 函数 (对于具有唯一安装点的特殊文件系统, 例如 `sysfs`) 以及 `get_sb_nODEV()` 函数 (对于可以安装多次的特殊文件系统, 例如 `tmpfs`; 参见下面)。

`get_sb_bdev()` 执行的最重要的操作如下:

1. 调用 `open_bdev_excl()` 打开设备文件名为 `dev_name` 的块设备 (参见第十三章的“字符设备驱动程序”一节)。
2. 调用 `sget()` 搜索文件系统的超级块对象链表 (`type->fs_supers`, 参见前面的“文

件系统类型注册”一节)。如果找到一个与块设备相关的超级块,则返回它的地址。否则,分配并初始化一个新的超级块对象,把它插入到文件系统链表和超级块全局链表中,并返回其地址。

3. 如果不是新的超级块(它不是上一步分配的,因为文件系统已经被安装),则跳到第6步。
4. 把参数 `flags` 中的值拷贝到超级块的 `s_flags` 字段,并将 `s_id`、`s_old_blocksize` 以及 `s_blocksize` 字段设置为块设备的合适值。
5. 调用依赖文件系统的函数(该函数作为传递给 `get_sb_bdev()` 的最后一个参数)访问磁盘上的超级块信息,并填充新超级块对象的其他字段。
6. 返回新超级块对象的地址。

安装根文件系统

安装根文件系统是系统初始化的关键部分。这是一个相当复杂的过程,因为Linux内核允许根文件系统存放在很多不同的地方,比如硬盘分区、软盘、通过NFS共享的远程文件系统,甚至保存在 *ramdisk* 中(RAM中的虚拟块设备)。

为了使叙述变得简单,让我们假定根文件系统存放在硬盘分区(毕竟这是最常见的情况)。当系统启动时,内核就要在变量 `ROOT_DEV` 中寻找包含根文件系统的磁盘主设备号(参见附录一)。当编译内核时,或者向最初的启动装入程序传递一个合适的“*root*”选项时,根文件系统可以被指定为 `/dev` 目录下的一个设备文件。类似地,根文件的安装标志存放在 `root_mountflags` 变量中。用户可以指定这些标志,或者通过对已编译的内核映像使用 *rdev* 外部程序,或者向最初的启动装入程序传递一个合适的 *rootflags* 选项来达到(参见附录一)。

安装根文件系统分两个阶段,如下所示:

1. 内核安装特殊 *rootfs* 文件系统,该文件系统仅提供一个作为初始安装点的空目录。
2. 内核在空目录上安装实际根文件系统。

为什么内核不怕麻烦,要在安装实际根文件系统之前安装 *rootfs* 文件系统呢?我们知道, *rootfs* 文件系统允许内核容易地改变实际根文件系统。事实上,在某些情况下,内核逐个地安装和卸载几个根文件系统。例如,一个发布版的初始启动光盘可能把具有一组最小驱动程序的内核装入RAM中,内核把存放在 *ramdisk* 中的一个最小的文件系统作为根安装。接下来,在这个初始根文件系统中的程序探测系统的硬件(例如,它们判断硬盘是否是EIDE、SCSI等等),装入所有必需的内核模块,并从物理块设备重新安装根文件系统。

阶段 1：安装 rootfs 文件系统

第一阶段是由 `init_rootfs()` 和 `init_mount_tree()` 函数完成的，它们在系统初始化过程中执行。

`init_rootfs()` 函数注册特殊文件系统类型 *rootfs*：

```
struct file_system_type rootfs_fs_type = {
    .name = "rootfs ";
    .get_sb = rootfs_get_sb;
    .kill_sb = kill_litter_super;
};
register_filesystem(&rootfs_fs_type);
```

`init_mount_tree()` 函数执行如下操作：

1. 调用 `do_kern_mount()` 函数，把字符串“rootfs”作为文件系统类型参数传递给它，并把该函数返回的新安装文件系统描述符的地址保存在 `mnt` 局部变量中。正如前一节所介绍的，`do_kern_mount()` 最终调用 *rootfs* 文件系统的 `get_sb` 方法，也即 `rootfs_get_sb()` 函数：

```
struct superblock *rootfs_get_sb(struct file_system_type *fs_type,
                                int flags, const char *dev_name, void *data)
{
    return get_sb_nodev(fs_type, flags|MS_NOUSER, data,
                       ramfs_fill_super);
}
```

`get_sb_nodev()` 函数执行如下步骤：

- a. 调用 `sget()` 函数分配新的超级块，传递 `set_anon_super()` 函数的地址作为参数（参见前面的“特殊文件系统”一节）。接下来，用合适的方式设置超级块的 `s_dev` 字段：主设备号为 0，次设备号不同于其他已安装的特殊文件系统的次设备号。
 - b. 将 `flags` 参数的值拷贝到超级块的 `s_flags` 字段中。
 - c. 调用 `ramfs_fill_super()` 函数分配索引节点对象和对应的目录项对象，并填充超级块字段值。由于 *rootfs* 是一种特殊文件系统，没有磁盘超级块，因此只需执行两个超级块操作。
 - d. 返回新超级块的地址。
2. 为进程 0 的命名空间分配一个 `namespace` 对象，并将它插入到由 `do_kern_mount()` 函数返回的已安装文件系统描述符中：

```
namespace = kmalloc(sizeof(*namespace), GFP_KERNEL);
list_add(&mnt->mnt_list, &namespace->list);
```

```
namespace->root = mnt;
mnt->mnt_namespace = init_task.namespace = namespace;
```

3. 将系统中其他每个进程的 namespace 字段设置为 namespace 对象的地址；同时初始化引用计数器 namespace->count（缺省情况下，所有的进程共享同一个初始 namespace）。
4. 将进程 0 的根目录和当前工作目录设置为根文件系统。

阶段 2：安装实际根文件系统

根文件系统安装操作的第二阶段是由内核在系统初始化即将结束时进行的。根据内核被编译时所选择的选项，和内核装入程序所传递的启动选项，可以有几种方法安装实际根文件系统。为了简单起见，我们只考虑磁盘文件系统的情况，它的设备文件名已通过“root”启动参数传递给内核。同时我们假定除了 *rootfs* 文件系统外，没有使用其他初始特殊文件系统。

prepare_namespace() 函数执行如下操作：

1. 把 root_device_name 变量置为从启动参数“root”中获取的设备文件名。同样，把 ROOT_DEV 变量置为同一设备文件的主设备号和次设备号。
2. 调用 mount_root() 函数，依次执行如下操作：
 - a. 调用 sys_mknod() (mknod() 系统调用的服务例程) 在 *rootfs* 初始根文件系统中创建设备文件 /dev/root，其主、次设备号与存放在 ROOT_DEV 中的一样。
 - b. 分配一个缓冲区并用文件系统类型名链表填充它。该链表要么通过启动参数“rootstype”传送给内核，要么通过扫描文件系统类型单向链表中的元素建立。
 - c. 扫描上一步建立的文件系统类型名链表。对每个名字，调用 sys_mount() 试图在根设备上安装给定的文件系统类型。由于每个特定于文件系统的方法使用不同的魔数，因此，对 get_sb() 的调用大都会失败，但有一个例外，那就是用根设备上实际使用过的文件系统的函数来填充超级块的那个调用，该文件系统被安装在 *rootfs* 文件系统的 /root 目录上。
 - d. 调用 sys_chdir (“/root”) 改变进程的当前目录。
3. 移动 *rootfs* 文件系统根目录上的已安装文件系统的安装点。

```
sys_mount(".", "/", NULL, MS_MOVE, NULL);
sys_chroot(".");
```

注意，*rootfs* 特殊文件系统没有被卸载：它只是隐藏在基于磁盘的根文件系统下了。

卸载文件系统

`umount()` 系统调用用来卸载一个文件系统。相应的 `sys_umount()` 服务例程作用于两个参数：文件名（多是安装点目录或是块设备文件名）和一组标志。该函数执行下列操作：

1. 调用 `path_lookup()` 查找安装点路径名；该函数把返回的查找操作结果存放在 `nameidata` 类型的局部变量 `nd` 中（参见下一节）。
2. 如果查找的最终目录不是文件系统的安装点，则设置 `retval` 返回码为 `-EINVAL` 并跳到第 6 步。这种检查是通过验证 `nd->mnt->mnt_root`（它包含由 `nd.dentry` 指向的目录项对象地址）进行的。
3. 如果要卸载的文件系统还没有安装在命名空间中，则设置 `retval` 返回码为 `-EINVAL` 并跳到第 6 步（回想一下，某些特殊文件系统没有安装点）。这种检查是通过在 `nd->mnt` 上调用 `check_mnt()` 函数进行的。
4. 如果用户不具有卸载文件系统的特权，则设置 `retval` 返回码为 `-EPERM` 并跳到第 6 步。
5. 调用 `do_umount()`，传递给它的参数为 `nd.mnt`（已安装文件系统对象）和 `flags`（一组标志）。该函数执行下列操作：
 - a. 从已安装文件系统对象的 `mnt_sb` 字段检索超级块对象 `sb` 的地址。
 - b. 如果用户要求强制卸载操作，则调用 `umount_begin` 超级块操作中中断任何正在进行的安装操作。
 - c. 如果要卸载的文件系统是根文件系统，且用户并不要求真正地把它卸载下来，则调用 `do_remount_sb()` 重新安装根文件系统为只读并终止。
 - d. 为进行写操作而获取当前进程的 `namespace->sem` 读/写信号量和 `vfsmount_lock` 自旋锁。
 - e. 如果已安装文件系统不包含任何子安装文件系统的安装点，或者用户要求强制卸载文件系统，则调用 `umount_tree()` 卸载文件系统（及其所有子文件系统）。
 - f. 释放 `vfsmount_lock` 自旋锁和当前进程的 `namespace->sem` 读/写信号量。
6. 减少相应文件系统根目录的目录项对象和已安装文件系统描述符的引用计数器值；这些计数器值由 `path_lookup()` 增加。
7. 返回 `retval` 的值。

路径名查找

当进程必须识别一个文件时，就把它的文件路径名传递给某个 VFS 系统调用，如 `open()`、`mkdir()`、`rename()` 或 `stat()`。本节我们要说明 VFS 如何实现路径名查找，也就是说如何从文件路径名导出相应的索引节点。

执行这一任务的标准过程就是分析路径名并把它拆分成一个文件名序列。除了最后一个文件名以外，所有的文件名都必定是目录。

如果路径名的第一个字符是“/”，那么这个路径名是绝对路径，因此从 `current->fs->root`（进程的根目录）所标识的目录开始搜索。否则，路径名是相对路径，因此从 `current->fs->pwd`（进程的当前目录）所标识的目录开始搜索。

在对初始目录的索引节点进行处理的过程中，代码要检查与第一个名字匹配的目录项，以获得相应的索引节点。然后，从磁盘读出包含那个索引节点的目录文件，并检查与第二个名字匹配的目录项，以获得相应的索引节点。对于包含在路径中的每个名字，这个过程反复执行。

目录项高速缓存极大地加速了这一过程，因为它把最近最常使用的目录项对象保留在内存中。正如我们以前看到的，每个这样的对象使特定目录中的一个文件名与它相应的索引节点相联系。因此在很多情况下，路径名的分析可以避免从磁盘读取中间目录。

但是，事情并不像看起来那么简单，因为必须考虑如下的 Unix 和 VFS 文件系统的特点：

- 对每个目录的访问权必须进行检查，以验证是否允许进程读取这一目录的内容。
- 文件名可能是与任意一个路径名对应的符号链接；在这种情况下，分析必须扩展到那个路径名的所有分量。
- 符号链接可能导致循环引用；内核必须考虑这个可能性，并能在出现这种情况时将循环终止。
- 文件名可能是一个已安装文件系统的安装点。这种情况必须检测到，这样，查找操作必须延伸到新的文件系统。
- 路径名查找应该在发出系统调用的进程的命名空间中完成。由具有不同命名空间的两个进程使用的相同路径名，可能指定了不同的文件。

路径名查找是由 `path_lookup()` 函数执行的，它接收三个参数：

`name`

指向要解析的文件路径名的指针。

flags

标志的值，表示将会怎样访问查找的文件。在后面的表 12-16 中列出了所允许的标志。

nd

nameidata 数据结构的地址，这个结构存放了查找操作的结果，其字段如表 12-15 所示。

当 path_lookup() 返回时，nd 指向的 nameidata 结构用与路径名查找操作有关的数据来填充。

表 12-15: nameidata 数据结构的字段

类型	字段	说明
struct dentry *	dentry	目录项对象的地址
struct vfs_mount *	mnt	已安装文件系统对象的地址
struct qstr	last	路径名的最后一个分量（当 LOOKUP_PARENT 标志被设置时使用）
unsigned int	flags	查找标志
int	last_type	路径名最后一个分量的类型（当 LOOKUP_PARENT 标志被设置时使用）
unsigned int	depth	符号链接嵌套的当前级别（参见下面）；它必须小于 6
char[] *	saved_names	与嵌套的符号链接关联的路径名数组
union	intent	单个成员联合体，指定如何访问文件

dentry 和 mnt 字段分别指向所解析的最后一个路径分量的目录项对象和已安装文件系统对象。这两个字段“描述”由给定路径名表示的文件。

由于 path_lookup() 函数返回的 nameidata 结构中的目录项对象和已安装文件系统对象代表了查找操作的结果，因此在 path_lookup() 的调用者完成使用查找结果之前，这两个对象都不能被释放。因此，path_lookup() 增加两个对象引用计数器的值。如果调用者想释放这些对象，则调用 path_release() 函数，传递给它的参数为 nameidata 结构的地址。

flags 字段存放查找操作中使用的某些标志的值；它们在表 12-16 中列出。这些标志中的大部分可由调用者在 path_lookup() 的 flags 参数中进行设置。

表12-16: 查找操作的标志

宏	说明
LOOKUP_FOLLOW	如果最后一个分量是符号链接, 则解释 (追踪) 它
LOOKUP_DIRECTORY	最后一个分量必须是目录
LOOKUP_CONTINUE	在路径名中还有文件名要检查
LOOKUP_PARENT	查找最后一个分量名所在的目录
LOOKUP_NOALT	不考虑模拟根目录 (在 80x86 体系结构中没有用)
LOOKUP_OPEN	试图打开一个文件
LOOKUP_CREATE	试图创建一个文件 (如果不存在)
LOOKUP_ACCESS	试图为一个文件检查用户的权限

path_lookup() 函数执行下列步骤:

1. 如下初始化 nd 参数的某些字段:
 - a. 把 last_type 字段置为 LAST_ROOT (如果路径名是一个 “/” 或 “/” 序列, 那么这是必需的, 参见后面的 “父路径名查找” 一节)。
 - b. 把 flags 字段置为参数 flags 的值。
 - c. 把 depth 字段置为 0。
2. 为进行读操作而获取当前进程的 current->fs->lock 读 / 写信号量。
3. 如果路径名的第一个字符是 “/”, 那么查找操作必须从当前根目录开始: 获取相应已安装文件对象(current->fs->rootmnt)和目录项对象(current->fs->root)的地址, 增加引用计数器的值, 并把它们的地址分别存放在 nd->mnt 和 nd->dentry 中。
4. 否则, 如果路径名的第一个字符不是 “/”, 则查找操作必须从当前工作目录开始: 获得相应已安装文件系统对象(current->fs->pwdmnt)和目录项对象(current->fs->pwd)的地址, 增加引用计数器的值, 并把它们的地址分别存放在 nd->mnt 和 nd->dentry 中。
5. 释放当前进程的 current->fs->lock 读 / 写信号量。
6. 把当前进程描述符中的 total_link_count 字段置为 0 (参见后面的 “符号链接的查找” 一节)。
7. 调用 link_path_walk() 函数处理正在进行的查找操作:

```
return link_path_walk(name, nd);
```

我们现在准备描述路径名查找操作的核心, 也就是 link_path_walk() 函数。它接收的参数为要解析的路径名指针 name 和 nameidata 数据结构的地址 nd。

为了简单起见，我们首先描述当 `LOOKUP_PARENT` 未被设置且路径名不包含符号链接时，`link_path_walk()` 做些什么（标准路径名查找）。接下来，我们讨论 `LOOKUP_PARENT` 被设置的情况：这种类型的查找在创建、删除或更名一个目录项时是需要的，也就是在父目录名查找过程中是需要的。最后，我们阐明该函数如何解析符号链接。

标准路径名查找

当 `LOOKUP_PARENT` 标志被清零时，`link_path_walk()` 执行下列步骤：

1. 用 `nd->flags` 初始化 `lookup_flags` 局部变量。
2. 跳过路径名第一个分量前的任何斜杠（/）。
3. 如果剩余的路径名为空，则返回 0。在 `nameidata` 数据结构中，`dentry` 和 `mnt` 字段指向原路径名最后一个所解析分量对应的对象。
4. 如果 `nd` 描述符中的 `depth` 字段的值为正，则把 `lookup_flags` 局部变量置为 `LOOKUP_FOLLOW` 标志（参见“符号链接的查找”一节）。
5. 执行一个循环，把 `name` 参数中传递的路径名分解为分量（中间的“/”被当作文件名分隔符对待）；对于每个找到的分量，该函数：
 - a. 从 `nd->dentry->d_inode` 检索最近一个所解析分量的索引节点对象的地址（在第一次循环中，索引节点指向开始路径名查找的目录）。
 - b. 检查存放在索引节点中的最近那个所解析分量的许可权是否允许执行（在 Unix 中，只有目录是可执行的，它才可以被遍历）。如果索引节点有自定义的 `permission` 方法，则执行它；否则，执行 `exec_permission_lite()` 函数，该函数检查存放在索引节点 `i_mode` 字段的访问模式和运行进程的特权。在两种情况中，如果最近所解析分量不允许执行，那么 `link_path_walk()` 跳出循环并返回一个错误码。
 - c. 考虑要解析的下一个分量。从它的名字，函数为目录项高速缓存散列表计算一个 32 位的散列值。
 - d. 如果“/”终止了要解析的分量名，则跳过“/”之后的任何尾部“/”。
 - e. 如果要解析的分量是原路径名中的最后一个分量，则跳到第 6 步。
 - f. 如果分量名是一个“.”（单个圆点），则继续下一个分量（“.”指的是当前目录，因此，这个点在目录内没有什么效果）。
 - g. 如果分量名是“..”（两个圆点），则尝试回到父目录：

- (1) 如果最近解析的目录是进程的根目录 (`nd->dentry` 等于 `current->fs->root`, 而 `nd->mnt` 等于 `current->fs->rootmnt`), 那么再向上追踪是不允许的: 在最近解析的分量上调用 `follow_mount()` (见下面), 继续下一个分量。
- (2) 如果最近解析的目录是 `nd->mnt` 文件系统的根目录 (`nd->dentry` 等于 `nd->mnt->mnt_root`), 并且这个文件系统也没有被安装在其他文件系统之上 (`nd->mnt` 等于 `nd->mnt->mnt_parent`), 那么 `nd->mnt` 文件系统通常 (注7) 就是命名空间的根文件系统: 在这种情况下, 再向上追踪是不可能的, 因此在最近解析的分量上调用 `follow_mount()` (参见下面), 继续下一个分量。
- (3) 如果最近解析的目录是 `nd->mnt` 文件系统的根目录, 而这个文件系统被安装在其他文件系统之上, 那么就需要文件系统交换。因此, 把 `nd->dentry` 置为 `nd->mnt->mnt_mountpoint`, 且把 `nd->mnt` 置为 `nd->mnt->mnt_parent`, 然后重新开始第5g步 (回想一下, 几个文件系统可以安装在同一个安装点上)。
- (4) 如果最近解析的目录不是已安装文件系统的根目录, 那么必须回到父目录: 把 `nd->dentry` 置为 `nd->dentry->d_parent`, 在父目录上调用 `follow_mount()`, 继续下一个分量。

`follow_mount()` 函数检查 `nd->dentry` 是否是某文件系统的安装点 (`nd->dentry->d_mounted` 的值大于0); 如果是, 则调用 `lookup_mnt()` 搜索目录项高速缓存中已安装文件系统的根目录, 并把 `nd->dentry` 和 `nd->mnt` 更新为相应已安装文件系统的对象地址; 然后重复整个操作 (几个文件系统可以安装在同一个安装点上)。从本质上说, 由于进程可能从某个文件系统的目录开始路径名的查找, 而该目录被另一个安装在其父目录上的文件系统所隐藏, 那么当需要回到父目录时, 则调用 `follow_mount()` 函数。

- h. 分量名既不是“.”, 也不是“..”, 因此函数必须在目录项高速缓存中查找它。如果低级文件系统有一个自定义的 `d_hash` 目录项方法, 则调用它来修改已在第5c步计算出的散列值。
- i. 把 `nd->flags` 字段中 `LOOKUP_CONTINUE` 标志对应的位置位, 这表示还有一个分量要分析。
- j. 调用 `do_lookup()`, 得到与给定的父目录 (`nd->dentry`) 和文件名 (要解析的路径名分量) 相关的目录项对象。该函数本质上首先调用 `__d_lookup()` 在目

注7: 这种情况还可能发生在解除网络文件系统与命名空间的目录树的连接时。

录项高速缓存中搜索分量的目录项对象。如果没有找到这样的目录项对象，则调用 `real_lookup()`。而 `real_lookup()` 执行索引节点的 `lookup` 方法从磁盘读取目录，创建一个新的目录项对象并把它插入到目录项高速缓存中，然后创建一个新的索引节点对象并把它插入到索引节点高速缓存中（注8）。在这一步结束时，`next` 局部变量中的 `dentry` 和 `mnt` 字段将分别指向这次循环要解析的分量名的目录项对象和已安装文件系统对象。

- k. 调用 `follow_mount()` 函数检查刚解析的分量 (`next.dentry`) 是否指向某个文件系统安装点的一个目录 (`next.dentry->d_mounted` 值大于0)。`follow_mount()` 更新 `next.dentry` 和 `next.mnt` 的值，以使它们指向由这个路径名分量所表示的目录上安装的最上层文件系统的目录项对象和已安装文件系统对象（参见第5g步）。
 - l. 检查刚解析的分量是否指向一个符号链接 (`next.dentry->d_inode` 具有一个自定义的 `follow_link` 方法)。我们将在后面的“符号链接的查找”一节中描述。
 - m. 检查刚解析的分量是否指向一个目录 (`next.dentry->d_inode` 具有一个自定义的 `lookup` 方法)。如果没有，返回一个错误码 `-ENOTDIR`，因为这个分量位于原路径名的中间。
 - n. 把 `nd->dentry` 和 `nd->mnt` 分别置为 `next.dentry` 和 `next.mnt`，然后继续路径名的下一个分量。
6. 现在，除了最后一个分量，原路径名的所有分量都被解析。清除 `nd->flags` 中的 `LOOKUP_CONTINUE` 标志。
 7. 如果路径名尾部有一个“/”，则把 `lookup_flags` 局部变量中 `LOOKUP_FOLLOW` 和 `LOOKUP_DIRECTORY` 标志对应的位置位，以强制由后面的函数来解释最后一个作为目录名的分量。
 8. 检查 `lookup_flags` 变量中 `LOOKUP_PARENT` 标志的值。下面假定这个标志被置为0，并把相反的情况推迟到下一节介绍。
 9. 如果最后一个分量名是“.”（单个圆点），则终止执行并返回值0（无错误）。在 `nd` 指向的 `nameidata` 数据结构中，`dentry` 和 `mnt` 字段指向路径名中倒数第二个分量对应的对象（任何分量“.”在路径名中没有效果）。

注8： 在少数情况下，函数 `real_lookup()` 可能发现所请求的索引节点已经在索引节点高速缓存中。路径名分量是最后一个路径名而且不是指向一个目录，与路径名相应的文件有几个硬链接，并且最近通过与这个路径名中被使用过的硬链接不同的硬链接访问过相应的文件。

10. 如果最后一个分量名是“..”（两个圆点），则尝试回到父目录：
 - a. 如果最后解析的目录是进程的根目录(`nd->dentry`等于`current->fs->root`, `nd->mnt`等于`current->fs->rootmnt`)，则在倒数第二个分量上调用`follow_mount()`，终止执行并返回值0（无错误）。`nd->dentry`和`nd->mnt`指向路径名的倒数第二个分量对应的对象，也就是进程的根目录。
 - b. 如果最后解析的目录是`nd->mnt`文件系统的根目录（`nd->dentry`等于`nd->mnt->mnt_root`），并且该文件系统没有被安装在另一个文件系统之上（`nd->mnt`等于`nd->mnt->mnt_parent`），那么再向上搜索是不可能的，因此在倒数第二个分量上调用`follow_mount()`，终止执行并返回值0（无错误）。
 - c. 如果最后解析的目录是`nd->mnt`文件系统的根目录，并且该文件系统被安装在其他文件系统之上，那么把`nd->dentry`和`nd->mnt`分别置为`nd->mnt->mnt_mountpoint`和`nd->mnt->mnt_parent`，然后重新执行第10步。
 - d. 如果最后解析的目录不是已安装文件系统的根目录，则把`nd->dentry`置为`nd->dentry->d_parent`，在父目录上调用`follow_mount()`，终止执行并返回值0（无错误）。`nd->dentry`和`nd->mnt`指向前一个分量（即路径名倒数第二个分量）对应的对象。
11. 路径名的最后分量名既不是“.”也不是“..”，因此，必须在高速缓存中查找它。如果低级文件系统有自定义的`d_hash`目录项方法，则该函数调用它来修改在第5c步已经计算出的散列值。
12. 调用`do_lookup()`，得到与父目录和文件名相关的目录项对象（参见第5j步）。在这一步结束时，`next`局部变量存放的是指向最后分量名对应的目录项和已安装文件系统描述符的指针。
13. 调用`follow_mount()`检查最后一个分量名是否是某个文件系统的安装点，如果是，则把`next`局部变量更新为最上层已安装文件系统根目录对应的目录项对象和已安装文件系统对象的地址。
14. 检查在`lookup_flags`中是否设置了`LOOKUP_FOLLOW`标志，且索引节点对象`next.dentry->d_inode`是否有一个自定义的`follow_link`方法。如果是，分量就是一个必须进行解释的符号链接，这将在后面的“符号链接的查找”一节描述。
15. 要解析的分量不是一个符号链接或符号链接不该被解释。把`nd->mnt`和`nd->dentry`字段分别置为`next.mnt`和`next.dentry`的值。最后的目录项对象就是整个查找操作的结果。
16. 检查`nd->dentry->d_inode`是否为`NULL`。这发生在没有索引节点与目录项对象关联时，通常是因为路径名指向一个不存在的文件。在这种情况下，返回一个错误码`-ENOENT`。

17. 路径名的最后一个分量有一个关联的索引节点。如果在 `lookup_flags` 中设置了 `LOOKUP_DIRECTORY` 标志, 则检查索引节点是否有一个自定义的 `lookup` 方法, 也就是说它是一个目录。如果没有, 则返回一个错误码 `-ENOTDIR`。
18. 返回值 0 (无错误)。`nd->dentry` 和 `nd->mnt` 指向路径名的最后分量。

父路径名查找

在很多情况下, 查找操作的真正目的并不是路径名的最后一个分量, 而是最后一个分量的前一个分量。例如, 当文件被创建时, 最后一个分量表示还不存在的文件的文件名, 而路径名中的其余路径指定新链接必须插入的目录。因此, 查找操作应当取回最后分量的前一个分量的目录项对象。另举一个例子, 把路径名 `/foo/bar` 表示的文件 `bar` 拆分出来就包含从目录 `foo` 中移去 `bar`。因此, 内核真正的兴趣在于访问文件目录 `foo` 而不是 `bar`。

当查找操作必须解析的是包含路径名最后一个分量的目录而不是最后一个分量本身时, 使用 `LOOKUP_PARENT` 标志。

当 `LOOKUP_PARENT` 标志被设置时, `link_path_walk()` 函数也在 `nameidata` 数据结构中建立 `last` 和 `last_type` 字段。`last` 字段存放路径名中的最后一个分量名。`last_type` 字段标识最后一个分量的类型; 可以把它置为如表 12-17 所示的值之一。

表 12-17: 在 `nameidata` 数据结构中 `last_type` 字段的值

值	说明
<code>LAST_NORM</code>	最后一个分量是普通文件名
<code>LAST_ROOT</code>	最后一个分量是 “/” (也就是整个路径名为 “/”)
<code>LAST_DOT</code>	最后一个分量是 “.”
<code>LAST_DOTDOT</code>	最后一个分量是 “..”
<code>LAST_BIND</code>	最后一个分量是链接到特殊文件系统的符号链接

当整个路径名的查找操作开始时, `LAST_ROOT` 标志是由 `path_lookup()` 设置的缺省值 (参见 “路径名查找” 一节开始部分的描述)。如果路径名正好是 “/”, 则内核不改变 `last_type` 字段的初始值。

`last_type` 字段的其他值在 `LOOKUP_PARENT` 标志置位时由 `link_path_walk()` 设置; 在这种情况下, 函数执行前一节描述的步骤, 直到第 8 步。不过, 从第 8 步往后, 路径名中最后一个分量的查找操作是不同的:

1. 把 `nd->last` 置为最后一个分量名。

2. 把 `nd->last_type` 初始化为 `LAST_NORM`。
3. 如果最后一个分量为“.”（一个圆点），则把 `nd->last_type` 置为 `LAST_DOT`。
4. 如果最后一个分量为“..”（两个圆点），则把 `nd->last_type` 置为 `LAST_DOTDOT`。
5. 通过返回值 0（无错误）终止。

你可以看到，最后一个分量根本就没有被解释。因此，当函数终止时，`nameidata` 数据结构的 `dentry` 和 `mnt` 字段指向最后一个分量所在目录对应的对象。

符号链接的查找

回想一下，符号链接是一个普通文件，其中存放的是另一个文件的路径名。路径名可以包含符号链接，且必须由内核来解析。

例如，如果 `/foo/bar` 是指向（包含路径名）`../dir` 的一个符号链接，那么，`/foo/bar/file` 路径名必须由内核解析为对 `/dir/file` 文件的引用。在这个例子中，内核必须执行两个不同的查找操作。第一个操作解析 `/foo/bar`；当内核发现 `bar` 是一个符号链接名时，就必须提取它的内容并把它解释为另一个路径名。第二个路径名操作从第一个操作所达到的目录开始，继续到符号链接路径名的最后一个分量被解析。接下来，原来的查找操作从第二个操作所达到的目录项恢复，且有了原目录名中紧随符号链接的分量。

对于更复杂的情景，含有符号链接的路径名可能包含其他的符号链接。你可能认为解析这类符号链接的内核代码是相当难理解的，但并非如此，代码实际上是相当简单的，因为它是递归的。

然而，难以驾驭的递归本质上是危险的。例如，假定一个符号链接指向自己。当然，解析含有这样符号链接的路径名可能导致无休止的递归调用流，这又依次引发内核栈的溢出。当前进程的描述符中的 `link_count` 字段用来避免这种问题：每次递归执行前增加这个字段的值，执行之后减少其值。如果该字段的值达到 6，整个循环操作就以错误码结束。因此，符号链接嵌套的层数不超过 5。

此外，当前进程的描述符中的 `total_link_count` 字段记录在原查找操作中有多少符号链接（甚至非嵌套的）被跟踪。如果这个计数器的值到 40，则查找操作中止。没有这个计数器，怀有恶意的用户就可能创建一个病态的路径名，让其中包含很多连续的符号链接，使内核在无休止的查找操作中冻结。

这就是代码基本工作的方式：一旦 `link_path_walk()` 函数检索到与路径名分量相关的目录项对象，就检查相应的索引节点对象是否有自定义的 `follow_link` 方法（参见“标准路径名查找”一节中的第 51 步和第 14 步）。如果是，索引节点就是一个符号链接，在原路径名的查找操作进行之前就必须先对这个符号链接进行解释。

在这种情况下，`link_path_walk()`函数调用`do_follow_link()`，前者传递给后者的参数为符号链接目录项对象的地址`dentry`和`nameidata`数据结构的地址`nd`。`do_follow_link()`依次执行下列步骤：

1. 检查`current->link_count`小于5；否则，返回错误码`-ELOOP`。
2. 检查`current->total_link_count`小于40；否则，返回错误码`-ELOOP`。
3. 如果当前进程需要，则调用`cond_resched()`进行进程交换（设置当前进程描述符`thread_info`中的`TIF_NEED_RESCHED`标志）。
4. 递增`current->link_count`、`current->total_link_count`和`nd->depth`的值。
5. 更新与要解析的符号链接关联的索引节点的访问时间。
6. 调用与具体文件系统相关的函数来实现`follow_link`方法，给它传递的参数为`dentry`和`nd`。它读取存放在符号链接索引节点中的路径名，并把这个路径名保存在`nd->saved_names`数组的合适项中。
7. 调用`__vfs_follow_link()`函数，给它传递的参数为地址`nd`和`nd->saved_names`数组中（参见下面）路径名的地址。
8. 如果定义了索引节点对象的`put_link`方法，就执行它，释放由`follow_link`方法分配的临时数据结构。
9. 减少`current->link_count`和`nd->depth`字段的值。
10. 返回由`__vfs_follow_link()`函数返回的错误码（0表示无错误）。

`__vfs_follow_link()`函数本质上依次执行下列操作：

1. 检查符号链接路径名的第一个字符是否是“/”：在这种情况下，已经找到一个绝对路径名，因此没有必要在内存中保留前一个路径的任何信息。如果是，对`nameidata`数据结构调用`path_release()`，因此释放由前一个查找步骤产生的对象；然后，设置`nameidata`数据结构的`dentry`和`mnt`字段，以使它们指向当前进程的根目录。
2. 调用`link_path_walk()`解析符号链的路径名，传递给它的参数为路径名和`nd`。
3. 返回从`link_path_walk()`取回的值。

当`do_follow_link()`最后终止时，它把局部变量`next`的`dentry`字段设置为目录项对象的地址，而这个地址由符号链接传递给原先就执行的`link_path_walk()`。`link_path_walk()`函数然后进行下一步。

VFS 系统调用的实现

为了简短起见，我们打算对表 12-1 中列出的所有 VFS 系统调用的实现进行讨论。不过，概略叙述几个系统调用的实现还是有用的，这里仅仅说明 VFS 的数据结构怎样互相作用。

让我们重新考虑一下在本章开始所提到的例子，用户发出了一条 shell 命令：把 */floppy/TEST* 中的 MS-DOS 文件拷贝到 */tmp/test* 中的 Ext2 文件中。命令 shell 调用一个外部程序（如 *cp*），我们假定 *cp* 执行下列代码片段：

```
inf = open("/floppy/TEST ", O_RDONLY, 0);
outf = open("/tmp/test ", O_WRONLY | O_CREAT | O_TRUNC, 0600);
do {
    len = read(inf, buf, 4096);
    write(outf, buf, len);
} while (len);
close(outf);
close(inf);
```

实际上，真正的 *cp* 程序的代码要更复杂些，因为它还必须检查由每个系统调用返回的可能的出错码。在我们的例子中，我们只把注意力集中在拷贝操作的“正常”行为上。

open() 系统调用

`open()` 系统调用的服务例程为 `sys_open()` 函数，该函数接收的参数为：要打开文件的路径名 `filename`、访问模式的一些标志 `flags`，以及如果该文件被创建所需要的许可位掩码 `mode`。如果该系统调用成功，就返回一个文件描述符，也就是指向文件对象的指针数组 `current->files->fd` 中分配给新文件的索引；否则，返回 `-1`。

在我们的例子中，`open()` 被调用两次；第一次是为读（`O_RDONLY` 标志）而打开 */floppy/TEST*，第二次是为写（`O_WRONLY` 标志）而打开 */tmp/test*。如果 */tmp/test* 不存在，则该文件被创建（`O_CREAT` 标志），文件主对该文件具有独占的读写访问权限（在第三个参数中的八进制数 `0600`）。

相反，如果该文件已经存在，则从头开始重写它（`O_TRUNC` 标志）。表 12-18 列出了 `open()` 系统调用的所有标志。

表 12-18: `open()` 系统调用的标志

标志名	说明
<code>O_RDONLY</code>	为读而打开
<code>O_WRONLY</code>	为写而打开

表 12-18: `open()`系统调用的标志 (续)

标志名	说明
<code>O_RDWR</code>	为读和写而打开
<code>O_CREAT</code>	如果文件不存在, 就创建它
<code>O_EXCL</code>	对于 <code>O_CREAT</code> 标志, 如果文件已经存在, 则失败
<code>O_NOCTTY</code>	从不把文件看作控制终端
<code>O_TRUNC</code>	截断文件 (删除所有现有的内容)
<code>O_APPEND</code>	总是在文件末尾写
<code>O_NONBLOCK</code>	没有系统调用在文件上阻塞
<code>O_NDELAY</code>	与 <code>O_NONBLOCK</code> 相同
<code>O_SYNC</code>	同步写 (阻塞, 直到物理写终止)
<code>FASYNC</code>	通过信号发出 I/O 事件通告
<code>O_DIRECT</code>	直接 I/O 传送 (无内核缓冲)
<code>O_LARGEFILE</code>	大型文件 (长度大于 2GB)
<code>O_DIRECTORY</code>	如果文件不是一个目录, 则失败
<code>O_NOFOLLOW</code>	不解释路径名中尾部的符号链接
<code>O_NOATIME</code>	不更新索引节点的上次访问时间

下面来描述一下 `sys_open()` 函数的操作。它执行如下操作:

1. 调用 `getname()` 从进程地址空间读取该文件的路径名。
2. 调用 `get_unused_fd()` 在 `current->files->fd` 中查找一个空的位置。相应的索引 (新文件描述符) 存放在 `fd` 局部变量中。
3. 调用 `filp_open()` 函数, 传递给它的参数为路径名、访问模式标志以及许可权位掩码。这个函数依次执行下列步骤:
 - a. 把访问模式标志拷贝到 `namei_flags` 标志中, 但是, 用特殊的格式对访问模式标志 `O_RDONLY`、`O_WRONLY` 和 `O_RDWR` 进行编码: 如果文件访问需要读特权, 那么只设置 `namei_flags` 标志的下标为 0 的位 (最低位); 类似地, 如果文件访问需要写特权, 就只设置下标为 1 的位。注意, 不可能在 `open()` 系统调用中不指定文件访问的读或写特权; 不过, 这种情况在涉及符号链接的路径名查找中则是有意义的。
 - b. 调用 `open_namei()`, 传递给它的参数为路径名、修改的访问模式标志以及局部 `nameidata` 数据结构的地址。该函数以下列方式执行查找操作:

- 如果访问模式标志中没有设置 `O_CREAT`，则不设置 `LOOKUP_PARENT` 标志而设置 `LOOKUP_OPEN` 标志后开始查找操作。此外，只有 `O_NOFOLLOW` 被清零，才设置 `LOOKUP_FOLLOW` 标志，而只有设置了 `O_DIRECTORY` 标志，才设置 `LOOKUP_DIRECTORY` 标志。
- 如果在访问模式标志中设置了 `O_CREAT`，则以 `LOOKUP_PARENT`、`LOOKUP_OPEN` 和 `LOOKUP_CREATE` 标志的设置开始查找操作。一旦 `path_lookup()` 函数成功返回，则检查请求的文件是否已存在。如果不存在，则调用父索引节点的 `create` 方法分配一个新的磁盘索引节点。

`open_namei()` 函数也在查找操作确定的文件上执行几个安全检查。例如，该函数检查与已找到的目录项对象关联的索引节点是否存在、它是否是一个普通文件，以及是否允许当前进程根据访问模式标志访问它。如果文件也是为写打开的，则该函数检查文件是否被其他进程加锁。

- c. 调用 `dentry_open()` 函数，传递给它的参数为访问模式标志、目录项对象的地址以及由查找操作确定的已安装文件系统对象。该函数依次执行下列操作：

- (1) 分配一个新的文件对象。
- (2) 根据传递给 `open()` 系统调用的访问模式标志初始化文件对象的 `f_flags` 和 `f_mode` 字段。
- (3) 根据作为参数传递来的目录项对象的地址和已安装文件系统对象的地址初始化文件对象的 `f_fentry` 和 `f_vfsmnt` 字段。
- (4) 把 `f_op` 字段设置为相应索引节点对象 `i_fop` 字段的内容。这就为进一步的文件操作建立起所有的方法。
- (5) 把文件对象插入到文件系统超级块的 `s_files` 字段所指向的打开文件的链表。
- (6) 如果文件操作的 `open` 方法被定义，则调用它。
- (7) 调用 `file_ra_state_init()` 初始化预读的数据结构（参见第十六章）。
- (8) 如果 `O_DIRECT` 标志被设置，则检查直接 I/O 操作是否可以作用于文件（参见第十六章）。
- (9) 返回文件对象的地址。

- d. 返回文件对象的地址。

4. 把 `current->files->fd[fd]` 置为由 `dentry_open()` 返回的文件对象的地址。

5. 返回 `fd`。

read()和 write()系统调用

让我们再回到 *cp* 例子的代码。open()系统调用返回两个文件描述符，分别存放在 *inf* 和 *outf* 变量中。然后，程序开始循环。在每次循环中，*ifloppy/TEST* 文件的一部分被拷贝到本地缓冲区 (read()系统调用) 中，然后，这个本地缓冲区中的数据又被拷贝到 *tmp/test* 文件 (write()系统调用)。

read()和 write()系统调用非常相似。它们都需要三个参数：一个文件描述符 *fd*、一个内存区的地址 *buf* (该缓冲区包含要传送的数据)，以及一个数 *count* (指定应该传送多少字节)。当然，read()把数据从文件传送到缓冲区，而 write()执行相反的操作。两个系统调用都返回所成功传送的字节数，或者发送一个错误条件的信号并返回 -1。

返回值小于 *count* 并不意味着发生了错误。即使请求的字节没有都被传送，也总是允许内核终止系统调用，因此用户应用程序必须检查返回值并重新发出系统调用 (如果必要)。在以下几种典型情况下返回小的值：当从管道或终端设备读取时，当读到文件的末尾时，或者当系统调用被信号中断时。文件结束条件 (EOF) 很容易从 read()的空返回值中判断出来。这个条件不会与因信号引起的异常终止混淆在一起，因为如果读取数据之前 read()被一个信号中断，则发生一个错误。

读或写操作总是发生在由当前文件指针所指定的文件偏移处 (文件对象的 *f_pos* 字段)。两个系统调用都通过把所传送的字节数加到文件指针上而更新文件指针。

简而言之，*sys_read()* (*read()* 的服务例程) 和 *sys_write()* (*write()* 的服务例程) 几乎都执行相同的步骤：

1. 调用 *fget_light()* 从 *fd* 获取相应文件对象的地址 *file* (参见前面的“与进程相关的文件”一节)。
2. 如果 *file->f_mode* 中的标志不允许所请求的访问 (读或写操作)，则返回一个错误码 -EBADF。
3. 如果文件对象没有 *read()* 或 *aio_read()* (*write()* 或 *aio_write()*) 文件操作，则返回一个错误码 -EINVAL。
4. 调用 *access_ok()* 粗略地检查 *buf* 和 *count* 参数 (参见第十章的“验证参数”一节)。
5. 调用 *rw_verify_area()* 对要访问的文件部分检查是否有冲突的强制锁。如果有，则返回一个错误码，如果该锁已经被 *F_SETLKW* 命令请求，那么就挂起当前进程 (参见本章后面的“文件加锁”一节)。
6. 调用 *file->f_op->read* 或 *file->f_op->write* 方法 (如果已定义) 来传送数据，否则，调用 *file->f_op->aio_read* 或 *file->f_op->aio_write* 方法。所有这些

方法（在第十六章讨论）都返回实际传送的字节数。另一方面的作用是，文件指针被适当地更新。

7. 调用 `fput_light()` 释放文件对象。
8. 返回实际传送的字节数。

close()系统调用

在我们例子的代码中，循环结束发生在 `read()` 系统调用返回 0 时，也就是说，发生在 `/floppy/TEST` 中的所有字节被拷贝到 `/tmp/test` 中时。然后，程序关闭打开的文件，这是因为拷贝操作已经完成。

`close()` 系统调用接收的参数为要关闭文件的文件描述符 `fd`。`sys_close()` 服务例程执行下列操作：

1. 获得存放在 `current->files->fd[fd]` 中的文件对象的地址；如果它为 `NULL`，则返回一个出错码。
2. 把 `current->files->fd[fd]` 置为 `NULL`。释放文件描述符 `fd`，这是通过清除 `current->files` 中的 `open_fds` 和 `close_on_exec` 字段的相应位来进行的（参见第二十章中有关关闭执行标志的内容）。
3. 调用 `filp_close()`，该函数执行下列操作：
 - a. 调用文件操作的 `flush` 方法（如果已定义）。
 - b. 释放文件上的任何强制锁（参见下一节）。
 - c. 调用 `fput()` 释放文件对象。
4. 返回 0 或一个出错码。出错码可由 `flush` 方法或文件中的前一个写操作错误产生。

文件加锁

当一个文件可以被多个进程访问时，就会出现同步问题。如果两个进程试图对文件的同一位置进行写会出现什么情况？或者，如果一个进程从文件的某个位置进行读而另一个进程正在对同一位置进行写会出现什么情况？

在传统的 Unix 系统中，对文件同一位置的同时访问会产生不可预料的结果。但是，Unix 系统提供了一种允许进程对一个文件区进行加锁的机制，以使同时访问可以很容易地被避免。

POSIX标准规定了基于 `fcntl()` 系统调用的文件加锁机制。这样就有可能对文件的任意一部分（甚至一个字节）加锁或对整个文件（包含以后要追加的数据）加锁。因为进程可以选择仅仅对文件的一部分加锁，因此，它也可以在文件的不同部分保持多个锁。

这种锁并不把不知道加锁的其他进程关在外面。与用于保护代码中临界区的信号量类似，可以认为这种锁起“劝告”的作用，因为只有访问文件之前其他进程合作检查锁的存在时，锁才起作用。因此，POSIX的锁被称为劝告锁（*advisory lock*）。

传统的BSD变体通过 `flock()` 系统调用来实现劝告锁。这个调用不允许进程对文件的一个区字段进行加锁，而只能对整个文件进行加锁。传统的System V变体提供了 `lockf()` 库函数，它仅仅是 `fcntl()` 的一个接口。

更重要的是，System V Release 3 引入了强制加锁（*mandatory locking*）：内核检查 `open()`、`read()` 和 `write()` 系统调用的每次调用都不违背在所访问文件上的强制锁。因此，强制锁甚至在非合作的进程之间也被强制加上（注9）。

不管进程是使用劝告锁还是强制锁，它们都可以使用共享读锁和独占写锁。在文件的某个区字段上，可以有任意多个进程进行读，但在同一时刻只能有一个进程进行写。此外，当其他进程对同一个文件都拥有自己的读锁时，就不可能获得一个写锁，反之亦然。

Linux 文件加锁

Linux 支持所有的文件加锁方式：劝告锁和强制锁，以及 `fcntl()`、`flock()` 和 `lockf()` 系统调用。不过，`lockf()` 系统调用仅仅是一个标准的库函数。

`flock()` 系统调用不管 `MS_MANDLOCK` 安装标志如何设置，只产生劝告锁。这是任何类 Unix 操作系统所期望的系统调用行为。在 Linux 中，增加了一种特殊的 `flock()` 强制锁，以允许对专有的网络文件系统的实现提供适当的支持。这就是所谓的共享模式强制锁；当这个锁被设置时，其他任何进程都不能打开与锁访问模式冲突的文件。不鼓励本地 Unix 应用程序中使用这个特征，因为这样加锁的源代码是不可移植的。

在 Linux 中还引入了另一种基于 `fcntl()` 的强制锁，叫做租借锁（*lease*）。当一个进程试图打开由租借锁保护的文件时，它照样被阻塞。然而，拥有锁的进程接收到一个信号。

注9：很奇怪，即使一个进程拥有某个文件上的强制锁，其他某个进程仍然会解除链接（或删除）这个文件。这种令人困惑的情况是可能发生的，因为当一个进程删除文件硬链接时，它不修改其内容，而只是修改它的父目录的内容。

一旦该进程得到通知，它应当首先更新文件，以使文件的内容保持一致，然后释放锁。如果拥有者不在预定的时间间隔（可以通过在 `/proc/sys/fs/lease-break-time` 文件中写入秒数来进行调整，通常为 45s）内这么做，则租借锁由内核自动删除，且允许阻塞的进程继续执行。

进程可以采用以下两种方式获得或释放一个文件劝告锁：

- 发出 `flock()` 系统调用。传递给它的两个参数为文件描述符 `fd` 和指定锁操作的命令。该锁应用于整个文件。
- 使用 `fcntl()` 系统调用。传递给它的三个参数为文件描述符 `fd`、指定锁操作的命令以及指向 `flock` 结构的指针（参见表 12-20）。`flock` 结构中的几个字段允许进程指定要加锁的文件部分。因此进程可以在同一文件的不同部分保持几个锁。

`fcntl()` 和 `flock()` 系统调用可以在同一文件上同时使用，但是通过 `fcntl()` 加锁的文件看起来与通过 `flock()` 加锁的文件不一样，反之亦然。这样当应用程序使用一种依赖于某个库的锁，而该库同时使用另一种类型的锁时，可以避免发生死锁。

处理强制文件锁要更复杂些。步骤如下：

1. 安装文件系统时强制锁是必需的，可使用 `mount` 命令的 `-o mand` 选项在 `mount()` 系统调用中设置 `MS_MANDLOCK` 标志。缺省操作是不使用强制锁。
2. 通过设置文件的 `set-group` 位（`SGID`）和清除 `group-execute` 许可权位将它们标记为强制锁的候选者。因为当 `group-execute` 位为 0 时，`set-group` 位也没有任何意义，因此内核将这种合并解释成使用强制锁而不是劝告锁。
3. 使用 `fcntl()` 系统调用（参见下面）获得或释放一个文件锁。

处理租借锁比处理强制锁要容易得多：调用具有 `F_SETLEASE` 或 `F_GETLEASE` 命令的系统调用 `fcntl()` 就足够了。使用另一个带有 `F_SETSIG` 命令的 `fcntl()` 系统调用可以改变传送给租借锁进程拥有者的信号类型。

当维护所有可以修改文件内容的系统调用时，除了 `read()` 和 `write()` 系统调用中的检查以外，内核还需要考虑强制锁的存在性。例如，如果文件中存在任何强制锁，那么带有 `O_TRUNC` 标志的 `open()` 系统调用就会失效。

下一节描述内核使用的主要数据结构，它们用于处理由 `flock()`（`FL_FLOCK` 锁）和 `fcntl()` 系统调用（`FL_POSIX` 锁）实现的文件锁。

文件锁的数据结构

Linux 中所有类型的锁都是由相同的 `file_lock` 数据结构描述的，它的字段如表 12-19 所示。

表 12-19: `file_lock` 数据结构的字段

类型	字段	说明
<code>struct file_lock *</code>	<code>fl_next</code>	与索引节点关联的锁链表中的下一个元素
<code>struct list_head</code>	<code>fl_link</code>	用于活动或阻塞链表的指针
<code>struct list_head</code>	<code>fl_block</code>	用于锁的等待者链表的指针
<code>struct files_struct *</code>	<code>fl_owner</code>	文件所有者的 <code>files_struct</code>
<code>unsigned int</code>	<code>fl_pid</code>	进程拥有者的 PID
<code>wait_queue_head_t</code>	<code>fl_wait</code>	阻塞进程的等待队列
<code>struct file *</code>	<code>fl_file</code>	指向文件对象的指针
<code>unsigned char</code>	<code>fl_flags</code>	锁标志
<code>unsigned char</code>	<code>fl_type</code>	锁类型
<code>loff_t</code>	<code>fl_start</code>	被锁区字段的起始偏移量
<code>loff_t</code>	<code>fl_end</code>	被锁区字段的末尾偏移量
<code>struct fasync_struct *</code>	<code>fl_fasync</code>	用于租借锁中断通知
<code>unsigned long</code>	<code>fl_break_time</code>	租借结束前的剩余时间
<code>struct file_lock_operations *</code>	<code>fl_ops</code>	指向文件锁操作的指针
<code>struct lock_manager_operations *</code>	<code>fl_mops</code>	指向锁管理操作的指针
Union	<code>fl_u</code>	具体文件系统的信息

指向磁盘上同一文件的所有 `lock_file` 结构都被收集在一个单向链表中，其第一个元素由索引节点对象的 `i_flock` 字段所指向。`file_lock` 结构的 `fl_next` 字段指向链表中的下一个元素。

当发出阻塞系统调用的进程请求一个独占锁而同一文件也存在共享锁时，该请求不能立即得到满足，并且进程必须被挂起。因此该进程被插入到由阻塞锁 `file_lock` 结构的 `fl_wait` 字段指向的等待队列中。使用两个链表区分已满足的锁请求（活动锁）和那些不能立刻得到满足的锁请求（阻塞锁）。

所有的活动锁被链接在“全局文件锁链表”中，该表的首元素被存放在 `file_lock_list` 变量中。类似地，所有的阻塞锁被链接在“阻塞链表”中，该表的首元素被存放在

blocked_list变量中。使用fl_link字段可把lock_file结构插入到上述任何一个链表中。

最后的一项要点是，内核必须跟踪所有与给定活动锁（“blocker”）关联的阻塞锁（“waiters”）：这就是为什么要使用链表根据给定的blocker把所有的waiter链接在一起的原因。blocker的fl_block字段是链表的伪首部，而waiter的fl_block字段存放了指向链表中相邻元素的指针。

FL_FLOCK 锁

FL_LOCK锁总是与一个文件对象相关联，因此由一个打开该文件的进程（或共享同一打开文件的子进程）来维护。当一个锁被请求或允许时，内核就把进程保持在同一文件对象上的任何其他锁都替换掉。这发生在进程想把一个已经拥有的读锁改变为一个写锁，或把一个写锁改变为一个读锁时。此外，当fput()函数正在释放一个文件对象时，对这个文件对象加的所有FL_LOCK锁都被撤销。不过，也有可能由其他进程对这同一文件（索引节点）设置了其他FL_LOCK读锁，它们依然是有效的。

flock()系统调用允许进程在打开文件上申请或删除劝告锁。它作用于两个参数：要加锁文件的文件描述符fd和指定锁操作的参数cmd。如果cmd参数为LOCK_SH，则请求一个共享的读锁；为LOCK_EX，则请求一个互斥的写锁；为LOCK_UN，则释放一个锁（注10）。

如果请求不能立即得到满足，系统调用通常阻塞当前进程，例如，如果进程请求一个独占锁而其他某个进程已获得了该锁。不过，如果LOCK_NB标记与LOCK_SH或LOCK_EX操作进行“或”，则这个系统调用不阻塞；换句话说，如果不能立即获得该锁，则该系统调用就返回一个错误码。

当sys_flock()服务例程被调用时，则执行下列步骤：

1. 检查fd是否是一个有效的文件描述符；如果不是，就返回一个错误码。否则，获得相应文件对象filp的地址。
2. 检查进程在打开文件上是否有读和/或写权限；如果没有，就返回一个错误码。
3. 获得一个新的file_lock对象锁并用适当的锁操作初始化它：根据参数cmd的值设置fl_type字段，把fl_file字段设为文件对象filp的地址，fl_flags字段设为FL_FLOCK，fl_pid字段设为current->tgid，并把fl_end字段设为-1，这表示对整个文件（而不是文件的一部分）加锁的事实。

注10：实际上，flock()系统调用还可以通过指定LOCK_MAND命令建立共享模式强制锁。不过，对此我们不做更多的讨论。

4. 如果参数 `cmd` 不包含 `LOCK_NB` 位, 则把 `FL_SLEEP` 标志加入 `fl_flags` 字段。
5. 如果文件具有一个 `flock` 文件操作, 则调用它, 传递给它的参数为文件对象指针 `filp`、一个标志 (`F_SETLKW` 或 `F_SETLK`, 取决于 `LOCK_NB` 位的值) 以及新的 `file_lock` 对象锁的地址。
6. 否则, 如果没有定义 `flock` 文件操作 (通常情况下), 则调用 `flock_lock_file_wait()` 试图执行请求的锁操作。传递给它的两个参数为: 文件对象指针 `filp` 和在第 3 步创建的新的 `file_lock` 对象的地址 `lock`。
7. 如果上一步中还没有把 `file_lock` 描述符插入活动或阻塞链表中, 则释放它。
8. 返回 0 (成功)。

`flock_lock_file_wait()` 函数执行下列循环操作:

1. 调用 `flock_lock_file()`, 传递给它的参数为文件对象指针 `filp` 和新的 `file_lock` 对象锁的地址 `lock`。这个函数依次执行下列操作:
 - a. 搜索 `filp->f_dentry->d_inode->i_flock` 指向的链表。如果在同一文件对象中找到 `FL_FLOCK` 锁, 则检查它的类型 (`LOCK_SH` 或 `LOCK_EX`): 如果该锁的类型与新锁相同, 则返回 0 (什么也没有做)。否则, 从索引节点锁链表和全局文件锁链表中删除这个 `file_lock` 元素, 唤醒 `fl-block` 链表中在该锁的等待队列上睡眠的所有进程, 并释放 `file_lock` 结构。
 - b. 如果进程正在执行开锁 (`LOCK_UN`), 则什么事情都不需要做: 该锁已不存在或已被释放, 因此返回 0。
 - c. 如果已经找到同一个文件对象的 `FL_FLOCK` 锁——表明进程想把一个已经拥有的读锁改变为一个写锁 (反之亦然), 那么调用 `cond_resched()` 给予其他更高优先级进程 (特别是先前在原文件锁上阻塞的任何进程) 一个运行的机会。
 - d. 再次搜索索引节点锁链表以验证现有的 `FL_FLOCK` 锁并不与所请求的锁冲突。在索引节点链表中, 肯定没有 `FL_FLOCK` 写锁, 此外, 如果进程正在请求一个写锁, 那么根本就没有 `FL_FLOCK` 锁。
 - e. 如果不存在冲突锁, 则把新的 `file_lock` 结构插入索引节点锁链表和全局文件锁链表中, 然后返回 0 (成功)。
 - f. 发现一个冲突锁: 如果 `fl_flags` 字段中 `FL_SLEEP` 对应的标志位置位, 则把新锁 (waiter 锁) 插入到 `blocker` 锁循环链表和全局阻塞链表中。
 - g. 返回一个错误码 `-EAGAIN`。
2. 检查 `flock_lock_file()` 的返回码:

- a. 如果返回码为 0 (没有冲突迹象), 则返回 0 (成功)。
- b. 不相容的情况。如果 `fl_flags` 字段中的 `FL_SLEEP` 标志被清除, 就释放 `file_lock` 锁描述符, 并返回一个错误码 `-EAGAIN`。
- c. 否则, 不相容但进程能够睡眠的情况: 调用 `wait_event_interruptible()` 把当前进程插入到 `lock->fl_wait` 等待队列中并挂起它。当进程被唤醒时 (正好在释放 `blocker` 锁后), 跳转到第 1 步再次执行这个操作。

FL_POSIX 锁

`FL_POSIX` 锁总是与一个进程和一个索引节点相关联。当进程死亡或一个文件描述符被关闭时 (即使该进程对同一文件打开了两次或复制了一个文件描述符), 这种锁会被自动地释放。此外, `FL_POSIX` 锁绝不会被子进程通过 `fork()` 继承。

当使用 `fcntl()` 系统调用对文件加锁时, 该系统调用作用于三个参数: 要加锁文件的文件描述符 `fd`、指向锁操作的参数 `cmd`, 以及指向存放在用户态进程地址空间中的 `flock` 数据结构 (注 11) 的指针 `fl`。 `flock` 结构中的字段如表 12-20 所示。

表 12-20: `flock` 数据结构中的字段

类型	字段	说明
short	<code>l_type</code>	<code>F_RDLCK</code> (请求一个共享锁), <code>F_WRLCK</code> (请求一个独占锁), <code>F_UNLCK</code> (释放锁)
short	<code>l_whence</code>	<code>SEEK_SET</code> (从文件的开始处), <code>SEEK_CURRENT</code> (从当前文件指针处), <code>SEEK_END</code> (从文件末尾处)
off_t	<code>l_start</code>	与 <code>l_whence</code> 的值相关的加锁区域的初始偏移量
off_t	<code>l_len</code>	加锁区域的长度 (0 表示该区域包含所有可能写过当前文件末尾的区域)
pid_t	<code>l_pid</code>	拥有者的 PID

`sys_fcntl()` 服务例程执行的操作取决于在 `cmd` 参数中所设置的标志值:

`F_GETLK`

确定由 `flock` 结构描述的锁是否与另一个进程已获得的某个 `FL_POSIX` 锁互相冲突。在冲突的情况下, 用现有锁的有关信息重写 `flock` 结构。

注 11: Linux 还定义了 `flock64` 结构, 它的 `offset` 和 `length` 字段使用 64 位长整数。下面我们主要讨论 `flock` 数据结构, 这些内容对 `flock64` 同样有效。

F_SETLK

设置由 flock 结构描述的锁。如果不能获得该锁,则这个系统调用返回一个错误码。

F_SETLKW

设置由 flock 结构描述的锁。如果不能获得该锁,则这个系统调用阻塞,也就是说,调用进程进入睡眠状态直到该锁可用时为止。

F_GETLK64, F_SETLK64, F_SETLKW64

与前面描述的几个标志相同,但是使用的是 flock64 结构而不是 flock 结构。

sys_fcntl() 服务例程首先获取与参数 fd 对应的文件对象,然后调用 fcntl_getlk() 或 fcntl_setlk() 函数(这取决于传递的参数: F_GETLK 表示前一个函数, F_SETLK 或 F_SETLKW 表示后一个函数)。我们仅仅考虑第二种情况。

fcntl_setlk() 函数作用于三个参数:指向文件对象的指针 filp、cmd 命令 (F_SETLK 或 F_SETLKW),以及指向 flock 数据结构的指针。该函数执行下列操作:

1. 读取局部变量中的参数 fl 所指向的 flock 结构。
2. 检查这个锁是否应该是一个强制锁,且文件是否有一个共享内存映射(参见第十六章的“内存映射”一节)。在肯定的情况下,该函数拒绝创建锁并返回 -EAGAIN 出错码,说明文件正在被另一个进程访问。
3. 根据用户 flock 结构的内容和存放在文件索引节点中的文件大小,初始化一个新的 file_lock 结构。
4. 如果命令 cmd 为 F_SETLKW,则该函数把 file_lock 结构的 fl_flags 字段设为 FL_SLEEP 标志对应的位置位。
5. 如果 flock 结构中的 l_type 字段为 F_RDLCK,则检查是否允许进程从文件读取,类似地,如果 l_type 为 F_WRLCK,则检查是否允许进程写入文件。如果都不是,则返回一个出错码。
6. 调用文件操作的 lock 方法(如果已定义)。对于磁盘文件系统,通常不定义该方法。
7. 调用 __posix_lock_file() 函数,传递给它的参数为文件的索引节点对象地址以及 file_lock 对象地址。该函数依次执行下列操作:
 - a. 对于索引节点的锁链表中的每个 FL_POSIX 锁,调用 posix_locks_conflict()。该函数检查这个锁是否与所请求的锁互相冲突。从本质上说,在索引节点的链表中,必定没有用于同一区的 FL_POSIX 写锁,并且,如果进程正在请求一个写锁,那么同一个区字段也可能根本没有 FL_POSIX 锁。但是,同一个进程所拥有的锁从不会冲突;这就允许进程改变它已经拥有的锁的特性。

- b. 如果找到一个冲突锁，则检查是否以 `F_SETLKW` 标志调用 `fcntl()`。如果是，当前进程应当被挂起：在这种情况下，调用 `posix_locks_deadlock()` 来检查在等待 `FL_POSIX` 锁的进程之间没有产生死锁条件，然后把新锁（waiter 锁）插入到冲突锁（blocker 锁）blocker 链表和阻塞链表中，最后返回一个出错码。否则，如果以 `F_SETLK` 标志调用 `fcntl()`，则返回一个出错码。
 - c. 只要索引节点的锁链表中不包含冲突的锁，就检查把文件区重叠起来的当前进程的所有 `FL_POSIX` 锁，当前进程想按需要对文件区中相邻的区字段进行锁定、组合及拆分。例如，如果进程为某个文件区请求一个写锁，而这个文件区落在一个较宽的读锁区字段内，那么，以前的读锁就会被拆分为两部分，这两部分覆盖非重叠区域，而中间区域由新的写锁进行保护。在重叠的情况下，新锁总是代替旧锁。
 - d. 把新的 `file_lock` 结构插入到全局锁链表和索引节点链表中。
 - e. 返回值 0（成功）。
8. 检查 `__posix_lock_file()` 的返回码：
- a. 如果返回码为 0（没有冲突迹象），则返回 0（成功）。
 - b. 不相容的情况。如果 `fl_flags` 字段的 `FL_SLEEP` 标志被清除，就释放新的 `file_lock` 描述符，并返回一个错误码 `-EAGAIN`。
 - c. 否则，如果不相容但进程能够睡眠时，调用 `wait_event_interruptible()` 把当前进程插入到 `lock->fl_wait` 等待队列中并挂起它。当进程被唤醒时（正好在释放 blocker 锁后），跳转到第 7 步再次执行这个操作。

第十三章



I/O 体系结构和设备驱动程序

上一章的虚拟文件系统在某种意义上依靠低层函数以进行适合于每个设备的读、写或其他操作。前一章还包括对不同文件系统如何处理这些操作的简单讨论。本章我们将看一下内核如何在实际的设备上调用这些操作。

在“I/O 体系结构”一节，我们简单考察一下 80x86 的 I/O 体系结构。在“设备驱动程序模型”一节，我们介绍 Linux 设备驱动程序模型。接下来，在“设备文件”一节，我们说明 VFS 如何把叫做“设备文件”的特殊文件与每个不同的硬件设备相对应，从而使应用程序可以用相同的方式使用所有的设备。在“设备驱动程序”一节，我们介绍一些常用的设备驱动程序特性。最后，在“字符设备驱动程序”一节，我们说明 Linux 字符设备驱动程序的整体组织结构。我们将在下一章中讨论块设备驱动程序。

有兴趣自己开发设备驱动程序的读者最好参考由 O'Reilly 出版，Jonathan Corbet Alessandro Rubini 和 Greg Kroah-Hartman 编写的《Linux Device Drivers》（第三版）一书。

I/O 体系结构

为了确保计算机能够正常工作，必须提供数据通路，让信息在连接到个人计算机的 CPU、RAM 和 I/O 设备之间流动。这些数据通路总称为总线，担当计算机内部主通信通道的作用。

所有计算机都拥有一条系统总线，它连接大部分内部硬件设备。一种典型的系统总线是 PCI (*Peripheral Component Interconnect*) 总线。目前使用其他类型的总线也很多，例

如 ISA、EISA、MCA、SCSI 和 USB。典型的情况是，一台计算机包括几种不同类型的总线，它们通过被称作“桥”的硬件设备连接在一起。两条高速总线用于在内存芯片上来回传送数据：前端总线将 CPU 连接到 RAM 控制器上，而后端总线将 CPU 直接连接到外部硬件的高速缓存上。主机上的桥将系统总线和前端总线连接在一起。

任何 I/O 设备有且仅能连接一条总线。总线的类型影响 I/O 设备的内部设计，也影响着内核如何处理设备。本节我们将讨论所有 PC 体系结构共有的功能性特点，而不具体介绍特定总线类型的技术细节。

CPU 和 I/O 设备之间的数据通路通常称为 I/O 总线。80x86 微处理器使用 16 位的地址总线对 I/O 设备进行寻址，使用 8 位、16 位或 32 位的数据总线传输数据。每个 I/O 设备依次连接到 I/O 总线上，这种连接使用了包含 3 个元素的硬件组织层次：I/O 端口、接口和设备控制器。图 13-1 显示了 I/O 体系结构的这些成分。

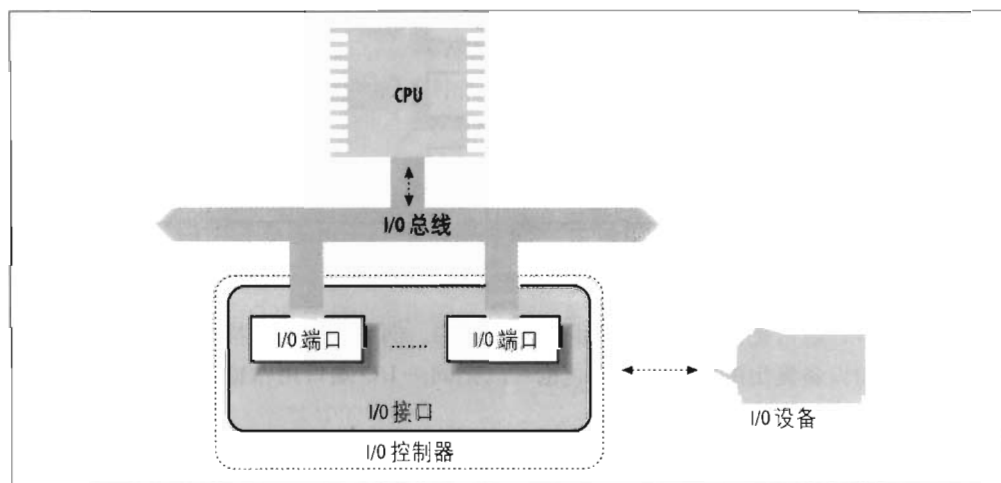


图 13-1：PC 的 I/O 体系结构

I/O 端口

每个连接到 I/O 总线上的设备都有自己的 I/O 地址集，通常称为 I/O 端口 (I/O port)。在 IBM PC 体系结构中，I/O 地址空间一共提供了 65536 个 8 位的 I/O 端口。可以把两个连续的 8 位端口看成一个 16 位端口，但是这必须从偶数地址开始。同理，也可以把两个连续的 16 位端口看成一个 32 位端口，但是这必须是从 4 的整数倍地址开始。有四条专用的汇编语言指令可以允许 CPU 对 I/O 端口进行读写，它们是 `in`、`ins`、`out` 和 `outs`。在执行其中的一条指令时，CPU 使用地址总线选择所请求的 I/O 端口，使用数据总线在 CPU 寄存器和端口之间传送数据。

I/O 端口还可以被映射到物理地址空间。因此，处理器和 I/O 设备之间的通信就可以使用对内存直接进行操作的汇编语言指令（例如，`mov`、`and`、`or` 等等）。现代的硬件设备更倾向于映射的 I/O，因为这样处理的速度较快，并可以和 DMA 结合起来。

系统设计者的主要目的是对 I/O 编程提供统一的方法，但又不牺牲性能。为了达到这个目的，每个设备的 I/O 端口都被组织成如图 13-2 所示的一组专用寄存器。CPU 把要发送给设备的命令写入设备控制寄存器（*device control register*），并从设备状态寄存器（*device status register*）中读出表示设备内部状态的值。CPU 还可以通过读取设备输入寄存器（*device input register*）的内容从设备取得数据，也可以通过向设备输出寄存器（*device output register*）中写入字节而把数据输出到设备。

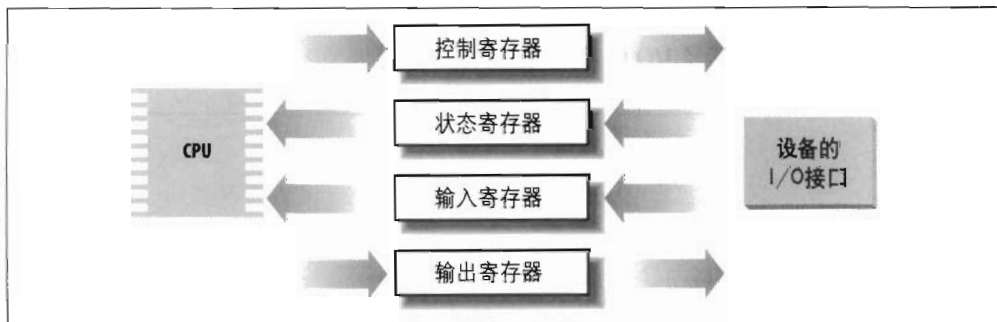


图 13-2：专用 I/O 端口

为了降低成本，通常把同一 I/O 端口用于不同目的。例如，某些位描述设备的状态，而其他位指定向设备发出的命令。同理，也可以把同一 I/O 端口用作输入寄存器或输出寄存器。

访问 I/O 端口

`in`、`out`、`ins` 和 `outs` 汇编语言指令都可以访问 I/O 端口。内核中包含了以下辅助函数来简化这种访问：

```
inb(), inw(), inl()
```

分别从 I/O 端口读取 1、2 或 4 个连续字节。后缀“b”、“w”、“l”分别代表一个字节（8 位）、一个字（16 位）以及一个长整型（32 位）。

```
inb_p(), inw_p(), inl_p()
```

分别从 I/O 端口读取 1、2 或 4 个连续字节，然后执行一条“哑元（dummy，即空指令）”指令使 CPU 暂停。

`outb()`, `outw()`, `outl()`

分别向一个 I/O 端口写入 1、2 或 4 个连续字节。

`outb_p()`, `outw_p()`, `outl_p()`

分别向一个 I/O 端口写入 1、2 或 4 个连续字节，然后执行一条“哑元”指令使 CPU 暂停。

`insb()`, `insw()`, `insl()`

分别从 I/O 端口读取以 1、2 或 4 个字节为一组的连续字节序列。字节序列的长度由该函数的参数给出。

`outsb()`, `outsw()`, `outsl()`

分别向 I/O 端口写入以 1、2 或 4 个字节为一组的连续字节序列。

虽然访问 I/O 端口非常简单，但是检测哪些 I/O 端口已经分配给 I/O 设备可能就不这么简单了，对基于 ISA 总线的系统来说更是如此。通常，I/O 设备驱动程序为了探测硬件设备，需要盲目地向某一 I/O 端口写入数据；但是，如果其他硬件设备已经使用了这个端口，那么系统就会崩溃。为了防止这种情况的发生，内核必须使用“资源”来记录分配给每个硬件设备的 I/O 端口。

资源 (*resource*) 表示某个实体的一部分，这部分被互斥地分配给设备驱动程序。在我们的情况中，一个资源表示 I/O 端口地址的一个范围。每个资源对应的信息存放在 `resource` 数据结构中，其字段如表 13-1 所示。所有的同种资源都插入到一个树型数据结构中；例如，表示 I/O 端口地址范围的所有资源都包含在一个根节点为 `ioport_resource` 的树中。

表 13-1: `resource` 数据结构中的字段

类型	字段	说明
<code>const char *</code>	<code>name</code>	资源拥有者的描述
<code>unsigned long</code>	<code>start</code>	资源范围的开始
<code>unsigned long</code>	<code>end</code>	资源范围的结束
<code>unsigned long</code>	<code>flags</code>	各种标志
<code>struct resource *</code>	<code>parent</code>	指向资源树中父亲的指针
<code>struct resource *</code>	<code>sibling</code>	指向资源树中兄弟的指针
<code>struct resource *</code>	<code>child</code>	指向资源树中第一个孩子的指针

节点的孩子被收集在一个链表中，其第一个元素由 `child` 指向。`sibling` 字段指向链表中的下一个节点。

为什么使用树？例如，考虑一下 IDE 硬盘接口所使用的 I/O 端口地址——比如说从 0xf000 到 0xf00f。然后，start 字段为 0xf000 且 end 字段为 0xf00f 的这样一个资源包含在树中，控制器的常规名字存放在 name 字段中。但是，IDE 设备驱动程序需要记住另外的信息，也就是 IDE 链 (IDE chain) 的主盘 (master disk) 使用 0xf000~0xf007 的子范围，从盘 (slave disk) 使用 0xf008~0xf00f 的子范围。为了做到这点，设备驱动程序把两个子范围对应的孩子插入到 0xf000~0xf00f 的整个范围对应的资源下。一般来说，树中的每个节点肯定相当于父节点对应范围的一个子范围。I/O 端口资源树 (ioport_resource) 的根节点跨越了整个 I/O 地址空间 (从端口 0~65535)。

任何设备驱动程序都可以使用下面三个函数，传递给它们的参数为资源树的根节点和要插入的新资源数据结构的地址：

```
request_resource()
```

把一个给定范围分配给一个 I/O 设备。

```
allocate_resource()
```

在资源树中寻找一个给定大小和排列方式的可用范围；若存在，就将这个范围分配给一个 I/O 设备（主要由 PCI 设备驱动程序使用，这种驱动程序可以配置成使用任意的端口号和主板上的内存地址对其进行配置）。

```
release_resource()
```

释放以前分配给 I/O 设备的给定范围。

内核也为以上应用于 I/O 端口的函数定义了一些快捷函数：`request_region()` 分配 I/O 端口的给定范围，`release_region()` 释放以前分配给 I/O 端口的范围。当前分配给 I/O 设备的所有 I/O 地址的树都可以从 `/proc/ioports` 文件中获得。

I/O 接口

I/O 接口 (*I/O interface*) 是处于一组 I/O 端口和对应的设备控制器之间的一种硬件电路。它起翻译器的作用，即把 I/O 端口中的值转换成设备所需要的命令和数据。在相反的方向上，它检测设备状态的变化，并对起状态寄存器作用的 I/O 端口进行相应的更新。还可以通过一条 IRQ 线把这种电路连接到可编程中断控制器上，以使它代表相应的设备发出中断请求。

有两种类型的接口：

专用 I/O 接口

专门用于一个特定的硬件设备。在一些情况下，设备控制器与这种 I/O 接口处于同

一块卡中（注1）。连接到专用 I/O 接口上的设备可以是内部设备（位于 PC 机箱内部的设备），也可以是外部设备（位于 PC 机箱外部的设备）。

通用 I/O 接口

用来连接多个不同的硬件设备。连接到通用 I/O 接口上的设备通常都是外部设备。

专用 I/O 接口

专用 I/O 接口的种类很多，因此目前已装在 PC 上设备的种类也很多，我们无法一一列出，在此只列出一些最通用的接口：

键盘接口

连接到一个键盘控制器上，这个控制器包含一个专用微处理器。这个微处理器对按下的组合键进行译码，产生一个中断并把相应的键盘扫描码写入输入寄存器。

图形接口

和图形卡中对应的控制器封装在一起，图形卡有自己的帧缓冲区，还有一个专用处理器以及存放在只读存储器（ROM）芯片中的一些代码。帧缓冲区是显卡上固化的存储器，其中存放的是当前屏幕内容的图形描述。

磁盘接口

由一条电缆连接到磁盘控制器，通常磁盘控制器与磁盘放在一起。例如，IDE 接口由一条 40 线的带形电缆连接到智能磁盘控制器上，在磁盘本身就可以找到这个控制器。

总线鼠标接口

由一条电缆把接口和控制器连接在一起，控制器就包含在鼠标中。

网络接口

与网卡中的相应控制器封装在一起，用以接收或发送网络报文。虽然广泛采用的网络标准很多，但还是以太网(IEEE 802.3)最为通用。

通用 I/O 接口

现代 PC 都包含连接很多外部设备的几个通用 I/O 接口。最常用的接口有：

并口

传统上用于连接打印机，它还可以用来连接可移动磁盘、扫描仪、备份设备、其他计算机等等。数据的传送以每次 1 字节（8 位）为单位进行。

注 1： 每块卡都要插入 PC 的一个可用空闲总线插槽中。如果一块卡通过一条外部电缆连接到一个外部设备上，那么在 PC 后面的面板中就有一个对应的连接器。

串口

与并口类似,但数据的传送是逐位进行的。串口包括一个通用异步收发器(UART)芯片,它可以把要发送的字节信息拆分成位序列,也可以把接收到的位流重新组装成字节信息。由于串口本质上速度低于并口,因此主要用于连接那些不需要高速操作的外部设备,如调制解调器、鼠标以及打印机。

PCMCIA 接口

大多数便携式计算机都包含这种接口。在不重新启动系统的情况下,这种形状类似于信用卡的外部设备可以被插入插槽或从插槽中拔走。最常用的PCMCIA设备是硬盘、调制解调器、网卡和扩展RAM。

SCSI (小型计算机系统接口) 接口

是把PC主总线连接到次总线(称为SCSI总线)的电路。SCSI-2总线允许一共8个PC和外部设备(硬盘、扫描仪、CR-ROM刻录机等等)连接在一起。如果有附加接口,宽带SCSI-2和新的SCSI-3接口可以允许你连接多达16个以上的设备。SCSI标准是通过SCSI总线连接设备的通信协议。

通用串行总线(USB)

高速运转的通用I/O接口,可用于连接外部设备,代替传统的并口、串口以及SCSI接口。

设备控制器

复杂的设备可能需要一个设备控制器(device controller)来驱动。从本质上说,控制器起两个重要作用:

- 对从I/O接口接收到的高级命令进行解释,并通过向设备发送适当的电信号序列强制设备执行特定的操作。
- 对从设备接收到的电信号进行转换和适当地解释,并修改(通过I/O接口)状态寄存器的值。

典型的设备控制器是磁盘控制器,它从微处理器(通过I/O接口)接收诸如“写这个数据块”之类的高级命令,并将其转换成诸如“把磁头定位在正确的磁道上”和“把数据写入这个磁道”之类的低级磁盘操作。现代的磁盘控制器相当复杂,因为它们可以把磁盘数据快速保存到内存的高速缓存中,还可以根据实际磁盘的几何结构重新安排CPU的高级请求,使其最优化。

比较简单的设备没有设备控制器,可编程中断控制器(参见第四章中的“中断和异常”一节)和可编程间隔定时器(参见第六章中的“可编程间隔定时器(PIT)一节”)就是这样的设备。

很多硬件设备都有自己的存储器，通常称之为 I/O 共享存储器。例如，所有比较新的图形卡在帧缓冲区中都有几 MB 的 RAM，用它来存放要在屏幕上显示的屏幕映像。我们将在本章的“访问 I/O 共享存储器”一节中讨论 I/O 共享存储器。

设备驱动程序模型

Linux 内核的早期版本为设备驱动程序的开发者提供微不足道的基本功能：分配动态内存，保留 I/O 地址范围或中断请求（IRQ），激活一个中断服务例程来响应设备的中断。事实上，在更老的硬件设备上编程棘手而困难重重，还有即使两种不同的硬件设备连在同一条总线上，但二者也很少有共同点。因此，试图为这种硬件设备的驱动程序开发者提供一种统一的模型是难以做到的。

现在的情形大不一样。诸如 PCI 这样的总线类型对硬件设备的内部设计提出了强烈的要求，因此，新的硬件设备即使类型不同但也有相似的功能。对这种设备的驱动程序应当特别关注：

- 电源管理（控制设备电源线上不同的电压级别）
- 即插即用（配置设备时透明的资源分配）
- 热插拔（系统运行时支持设备的插入和移走）

系统中所有硬件设备由内核全权负责电源管理。例如，在以电池供电的计算机进入“待机”状态时，内核应立刻强制每个硬件设备（硬盘、显卡、声卡、网卡、总线控制器等等）处于低功率状态。因此，每个能够响应“待机”状态的设备驱动程序必须包含一个回调函数，它能够使得硬件设备处于低功率状态。而且，硬件设备必须按准确的顺序进入“待机”状态，否则一些设备可能会处于错误的电源状态。例如，内核必须首先将硬盘置于“待机”状态，然后才是它们的磁盘控制器，因为若按照相反的顺序执行，磁盘控制器就不能向硬盘发送命令。

为了实现这些操作，Linux 2.6 提供了一些数据结构和辅助函数，它们为系统中所有的总线、设备以及设备驱动程序提供了一个统一的视图；这个框架被称为设备驱动程序模型。

sysfs 文件系统

sysfs 文件系统是一种特殊的文件系统，被安装于 `/sys` 目录下的 `/proc` 文件系统相似。`/proc` 文件系统是首次被设计成允许用户态应用程序访问内核内部数据结构的一种文件系统。`/sysfs` 文件系统本质上与 `/proc` 有相同的目的，但是它还提供关于内核数据结构的附加信

息，此外，*/sysfs*的组织结构比*/proc*更有条理。或许，在不远的将来，*/proc*和*/sysfs*将会继续共存。

*sysfs*文件系统的目标是要展现设备驱动程序模型组件间的层次关系。该文件系统的相应高层目录是：

block

块设备，它们独立于所连接的总线。

devices

所有被内核识别的硬件设备，依照连接它们的总线对其进行组织。

bus

系统中用于连接设备的总线。

drivers

在内核中注册的设备驱动程序。

class

系统中设备的类型（声卡、网卡、显卡等等），同一类可能包含由不同总线连接的设备，于是由不同的驱动程序驱动。

power

处理一些硬件设备电源状态的文件。

firmware

处理一些硬件设备的固件的文件。

*sysfs*文件系统中所表示的设备驱动程序模型组件之间的关系就像目录和文件之间符号链接的关系一样。例如，文件 */sys/block/sda/device* 可以是一个符号链接，指向在 */sys/devices/pci0000:00*（表示连接到 PCI 总线的 SCSI 控制器）中嵌入的一个子目录。此外，文件 */sys/block/sda/device/block* 是到目录 */sys/block/sda* 的一个符号链接，这表明这个 PCI 设备是 SCSI 磁盘的控制器。

sysfs 文件系统中普通文件的主要作用是表示驱动程序和设备的属性。例如，位于目录 */sys/block/hda* 下的 *dev* 文件含有第一个 IDE 链主磁盘的主设备号和次设备号。

kobject

设备驱动程序模型的核心数据结构是一个普通的数据结构，叫做 *kobject*，它与 *sysfs* 文件系统自然地绑定在一起：每个 *kobject* 对应于 *sysfs* 文件系统中的—个目录。

kobject 被嵌入一个叫做“容器”的更大对象中，容器描述设备驱动程序模型中的组件（注 2）。容器的典型例子有总线、设备以及驱动程序的描述符，例如，第一个 IDE 磁盘的第一个分区描述符对应于 `/sys/block/hda/hda1` 目录。

将一个 kobject 嵌入容器中允许内核：

- 为容器保持一个引用计数器。
- 维持容器的层次列表或组（例如，与块设备相关的 `sysfs` 目录为每个磁盘分区包含一个不同的子目录）。
- 为容器的属性提供一种用户态查看的视图。

kobject、kset 和 subsystem

每个 kobject 由 kobject 数据结构描述，其各字段如表 13-2 所示。

表 13-2: kobject 数据结构中的字段

类型	字段	说明
char *	k_name	指向含有容器名称的字符串
char []	name	含有容器名称的字符串，如果它不超过 20 个字节
struct k_ref	kref	容器的引用计数器
struct list_head	entry	用于 kobject 所插入的链表的指针
struct kobject *	parent	指向父 kobject（如果存在的话）
struct kset *	kset	指向包含的 kset
struct kobj_type *	ktype	指向 kobject 的类型描述符
struct dentry *	dentry	指向与 kobject 相对应的 <code>sysfs</code> 文件的 <code>dentry</code> 数据结构

ktype 字段指向 `kobj_type` 对象，该对象描述了 kobject 的“类型”——本质上，它描述的是包括 kobject 的容器的类型。`kobj_type` 数据结构包括三个字段：`release` 方法（当 kobject 被释放时执行），指向 `sysfs` 操作表的 `sysfs_ops` 指针以及 `sysfs` 文件系统的缺省属性链表。

kref 字段是一个 `k_ref` 类型的结构，它仅包括一个 `refcount` 字段。顾名思义，这个字段就是 kobject 的引用计数器，但它也可以作为 kobject 容器的引用计数器。`kobject_get()`

注 2: kobject 主要用于实现设备驱动程序模型，但是为了使用 kobject，还要继续努力改变其他的一些内核部件，如模块子系统。

和 `kobject_put()` 函数分别用于增加和减少引用计数器的值，如果该计数器的值等于 0，就会释放 `kobject` 使用的资源，并且执行 `kobject` 的类型描述符 `kobj_type` 对象的 `release` 方法。该方法用于释放容器本身，通常只有在动态地分配 `kobject` 容器时才定义该方法。

通过 `kset` 数据结构可将 `kobjects` 组织成一棵层次树。`kset` 是同类型 `kobject` 结构的一个集合体——也就是说，相关的 `kobject` 包含在同类型的容器中。`kset` 数据结构的字段如表 13-3 所示。

表 13-3: `kset` 数据结构中的字段

类型	字段	说明
<code>struct subsystem *</code>	<code>subsys</code>	指向 <code>subsystem</code> 描述符
<code>struct kobj_type *</code>	<code>ktype</code>	指向 <code>kset</code> 的 <code>kobject</code> 类型描述符
<code>struct list_head</code>	<code>list</code>	包含在 <code>kset</code> 中的 <code>kobject</code> 链表的首部
<code>struct kobject</code>	<code>kobj</code>	嵌入的 <code>kobject</code> 结构（见下文）
<code>struct kset_hotplug_ops *</code>	<code>hotplug_ops</code>	指向用于处理 <code>kobject</code> 结构的过滤和热插拔操作的回调函数表

`list` 字段表示包含在 `kset` 中的 `kobject` 结构的双向循环链表的首部。`ktype` 字段是指向 `kset` 中的 `kobj_type` 描述符的指针，该描述符被 `kset` 中所有的 `kobject` 结构共享。

`kobj` 字段是嵌入在 `kset` 数据结构中的 `kobject`，而位于 `kset` 中的 `kobject`，其 `parent` 字段指向这个内嵌的 `kobject` 结构。因此，一个 `kset` 就是 `kobject` 集合体，但是它依赖于层次树中用于引用计数和连接的更高层 `kobject`。这种设计编码效率很高，并可获得最大的灵活性。例如，分别用于增加和减少 `kset` 引用计数器值的 `kset_get()` 函数和 `kset_put()` 函数，只需简单地调用内嵌的 `kobject` 结构中的 `kobject_get()` 函数和 `kobject_put()` 函数，因为 `kset` 的引用计数器只不过是内嵌在 `kset` 中的类型为 `kobject` 的 `kobj` 的引用计数器。而且，由于有了内嵌的 `kobject` 结构，`kset` 数据结构可以嵌入到“容器”对象中，非常类似于嵌入的 `kobject` 数据结构。最后，`kset` 可以作为其他 `kset` 的一个成员：它足以将内嵌的 `kobject` 插入到更高层次的 `kset` 中。

还存在所谓 `subsystem` 的 `kset` 集合。一个 `subsystem` 可以包括不同类型的 `kset`，用包含两个字段的 `subsystem` 数据结构来描述：

`kset`

内嵌的 `kset` 结构，用于存放 `subsystem` 中的 `kset`。

`rwsem`

读写信号量，保护递归地包含于 `subsystem` 中的所有 `kset` 和 `kobject`。

subsystem 数据结构甚至也可以嵌入到一个更大的“容器”对象中；因此，容器的引用计数器也是内嵌 subsystem 的引用计数器——也就是嵌入在 subsystem 中的 kset 所做的 kobject 的引用计数器。subsys_get() 和 subsys_put() 函数分别用于增加和减少这个引用计数器的值。

图 13-3 显示了设备驱动程序模型层次的一个例子。bus 子系统包括一个 pci 子系统，pci 子系统又依次包含驱动程序的一个 kset。这个 kset 包含一个串口 kobject（具有唯一 new-id 属性的串口对应的设备驱动程序程序）。

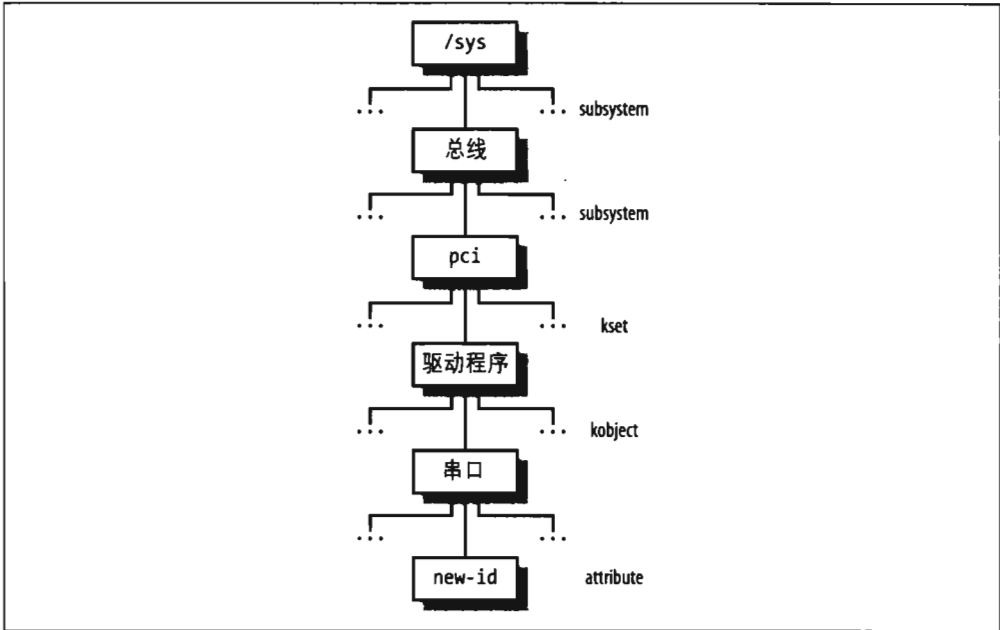


图 13-3：设备驱动程序层次模型的一个实例

注册 kobject、kset 和 subsystem

一般来说，如果想让 kobject、kset 或 subsystem 出现在 sysfs 子树中，就必须首先注册它们。与 kobject 对应的目录总是出现在其父 kobject 的目录中。例如，位于同一个 kset 中的 kobject 的目录出现在 kset 本身的目录中。因此，sysfs 子树的结构就描述了各种已注册的 kobject 之间以及各种容器对象之间的层次关系。

通常，sysfs 文件系统的上层目录肯定是已注册的 subsystem。

kobject_register() 函数用于初始化 kobject，并且将其相应的目录增加到 sysfs 文件系统中。在调用此函数之前，调用程序应该先设置 kobject 结构中的 kset 字段，使它指向其

父 `kset` (如果有的话)。`kobject_unregister()` 函数则将 `kobject` 的目录从 `sysfs` 文件系统中移走。为了更易于内核开发者进行开发, `Linux` 也提供了 `kset_register()` 和 `kset_unregister()` 函数, 以及 `subsystem_register()` `subsystem_unregister()` 函数, 但本质上它们是围绕 `kobject_register()` 和 `kobject_unregister()` 的封装函数。

如前所述, 许多 `kobject` 目录都包括称作属性 (*attribute*) 的普通文件。`sysfs_create_file()` 函数接收 `kobject` 的地址和属性描述符作为它的参数, 并在合适的目录中创建特殊文件。`sysfs` 文件系统中所描述的对象间的其他关系可以通过符号链接的方式来建立: `sysfs_create_link()` 函数为目录中与其他 `kobject` 相关联的特定 `kobject` 创建一个符号链接。

设备驱动程序模型的组件

设备驱动程序模型建立在几个基本数据结构之上, 这些结构描述了总线、设备、设备驱动器等等。让我们来考察一下它们。

设备

设备驱动程序模型中的每个设备是由一个 `device` 对象来描述的, 其字段如表 13-4 所示。

表 13-4: `device` 对象中的字段

类型	字段	说明
<code>struct list_head</code>	<code>node</code>	指向兄弟设备链表的指针
<code>struct list_head</code>	<code>bus_list</code>	指向连于同一类型总线上的设备链表的指针
<code>struct list_head</code>	<code>driver_list</code>	指向设备驱动程序链表的指针
<code>struct list_head</code>	<code>children</code>	子设备链表的首部
<code>struct device *</code>	<code>parent</code>	指向父设备的指针
<code>struct kobject</code>	<code>kobj</code>	内嵌的 <code>kobject</code> 结构
<code>char []</code>	<code>bus_id</code>	连接到总线上设备的位置
<code>struct bus_type *</code>	<code>bus</code>	指向所连接总线的指针
<code>struct device_driver *</code>	<code>driver</code>	指向控制设备驱动程序的指针
<code>void *</code>	<code>driver_data</code>	指向驱动程序私有数据的指针
<code>void *</code>	<code>platform_data</code>	指向遗留设备驱动程序的私有数据的指针
<code>struct dev_pm_info</code>	<code>power</code>	电源管理信息

表 13-4: device 对象中的字段 (续)

类型	字段	说明
unsigned long	detach_state	卸载设备驱动程序时电源进入的状态
unsigned long long *	dma_mask	指向设备的 DMA 屏蔽字的指针 [参见稍后的“直接内存方向 (DMA)”一节]
unsigned long long	coherent_dma_mask	设备的一致性 DMA 的屏蔽字
struct list_head	dma_pools	聚集的 DMA 缓冲池链表的首部
struct dma_coherent_mem *	dma_mem	指向设备所使用的一致性 DMA 存储器描述符的指针 [参见稍后的“直接内存访问 (DMA)”一节]
void (*)(struct device *)	release	释放设备描述符的回调函数

device 对象全部收集在 `devices_subsys` 子系统中, 该子系统对应的目录为 `/sys/devices` (参见前面的“kobject”一节)。设备是按照层次关系组织的: 一个设备是某个“孩子”的“父亲”, 其条件为子设备离开父设备无法正常工作。例如, 在基于 PCI 总线的计算机上, 位于 PCI 总线和 USB 总线之间的桥就是连接在 USB 总线上的所有设备的父设备。device 对象的 `parent` 字段是指向其父设备描述符的指针, `children` 字段是子设备链表的首部, 而 `node` 字段存放指向 `children` 链表中相邻元素的指针。device 对象中内嵌的 kobject 间的亲子关系也反映了设备的层次关系, 因此, `/sys/devices` 下的目录结构与硬件设备的物理组织是匹配的。

每个设备驱动程序都保持一个 device 对象链表, 其中链接了所有可被管理的设备, device 对象的 `driver_list` 字段存放指向相邻对象的指针, 而 `driver` 字段指向设备驱动程序描述符。此外, 对于任何总线类型来说, 都有一个链表存放连接到该类型总线上的所有设备, device 对象的 `bus_list` 字段存放指向相邻对象的指针, 而 `bus` 字段指向总线类型描述符。

引用计数器记录 device 对象的使用情况, 它包含在 kobject 类型的 `kobj` 结构中, 通过调用 `get_device()` 和 `put_device()` 函数分别增加和减少该计数器的值。

`device_register()` 函数的功能是往设备驱动程序模型中插入一个新的 device 对象, 并自动地在 `/sys/devices` 目录下为其创建一个新的目录。相反地, `device_unregister()` 函数的功能是从设备驱动程序模型中移走一个设备。

通常, device 对象被静态地嵌入到一个更大的描述符中。例如, PCI 设备是由数据结构 `pci_dev` 描述, 该数据结构的 `dev` 字段就是一个 device 对象, 而其他字段则是 PCI 总

线所特有的。在PCI内核层上,当注册或注销设备时就会分别执行`device_register()`函数和`device_unregister()`函数。

驱动程序

设备驱动程序模型中的每个驱动程序都可由`device_driver`对象描述,其各字段如表13-5所示。

表 13-5: `device_driver` 对象中的字段

类型	字段	说明
<code>char *</code>	<code>name</code>	设备驱动程序的名称
<code>struct bus_type *</code>	<code>bus</code>	指向总线描述符的指针,总线连接所支持的设备
<code>struct semaphore</code>	<code>unload_sem</code>	禁止卸载设备驱动程序的信号量;当引用计数器的值为0时释放该信号量
<code>struct kobject</code>	<code>kobj</code>	内嵌的 <code>kobject</code> 结构
<code>struct list_head</code>	<code>devices</code>	驱动程序支持的所有设备组成的链表的首部
<code>struct module *</code>	<code>owner</code>	标识实现设备驱动程序的模块,如果有的话(参见附录二)
<code>int (*)(struct device *)</code>	<code>probe</code>	探测设备的方法(检验设备驱动程序是否可以控制该设备)
<code>int (*)(struct device *)</code>	<code>remove</code>	移走设备时所调用的方法
<code>void (*)(struct device *)</code>	<code>shutdown</code>	设备断电(关闭)时所调用的方法
<code>int (*)(struct device *, unsigned long, unsigned long)</code>	<code>suspend</code>	设备置于低功率状态时所调用的方法
<code>int (*)(struct device *, unsigned long)</code>	<code>resume</code>	设备恢复正常状态时所调用的方法

`device_driver`对象包括四个方法,它们用于处理热插拔、即插即用和电源管理。当总线设备驱动程序发现一个可能由它处理的设备时就会调用`probe`方法;相应的函数将会探测该硬件,从而对该设备进行更进一步的检查。当移走一个可热插拔的设备时驱动程序会调用`remove`方法;而驱动程序本身被卸载时,它所处理的每个设备也会调用`remove`方法。当内核必须改变设备的供电状态时,设备会调用`shutdown`、`suspend`和`resume`三个方法。

内嵌在描述符中的`kobject`类型的`kobj`所包含的引用计数器用于记录`device_driver`对

象的使用情况。通过调用 `get_driver()` 函数和 `put_driver()` 函数可分别增加和减少该计数器的值。

`driver_register()` 函数的功能是往设备驱动程序模型中插入一个新的 `device_driver` 对象, 并自动地在 `sysfs` 文件系统下为其创建一个新的目录。相反, `driver_unregister()` 函数的功能则是从设备驱动程序模型中移走一个设备驱动对象。

通常, `device_driver` 对象静态地被嵌入到一个更大的描述符中。例如, PCI 设备驱动程序是由数据结构 `pci_driver` 描述的, 该数据结构的 `driver` 字段是一个 `device_driver` 对象, 而其他字段则是 PCI 总线所特有的。

总线

内核所支持的每一种总线类型都由一个 `bus_type` 对象描述, 其各字段如表 13-6 所示。

表 13-6: `bus_type` 对象中的字段

类型	字段	说明
<code>char *</code>	<code>name</code>	总线类型的名称
<code>struct subsystem</code>	<code>subsys</code>	与总线类型相关的 <code>kobject</code> 子系统
<code>struct kset</code>	<code>drivers</code>	驱动程序的 <code>kobject</code> 集合
<code>struct kset</code>	<code>devices</code>	设备的 <code>kobject</code> 集合
<code>struct bus_attribute *</code>	<code>bus_attrs</code>	指向对象的指针, 该对象包含总线属性和用于导出此属性到 <code>sysfs</code> 文件系统的方法
<code>struct device_attribute *</code>	<code>dev_attrs</code>	指向对象的指针, 该对象包含设备属性和用于导出此属性到 <code>sysfs</code> 文件系统的方法
<code>struct driver_attribute *</code>	<code>drv_attrs</code>	指向对象的指针, 该对象包含设备驱动程序属性和用于导出此属性到 <code>sysfs</code> 文件系统的方法
<code>int (*)(struct device *, struct device_driver *)</code>	<code>match</code>	检验给定的设备驱动程序是否支持特定设备的方法
<code>int (*)(struct device *, char **, int, char *, int)</code>	<code>hotplug</code>	注册设备时调用的方法
<code>int (*)(struct device *, unsigned long)</code>	<code>suspend</code>	保存硬件设备的上下文状态并改变设备供电状态的方法
<code>int (*)(struct device *)</code>	<code>resume</code>	改变供电状态和恢复硬件设备上下文的方法

每个 `bus_type` 类型的对象都包含一个内嵌的子系统，存放于 `bus_subsys` 变量中的子系统把嵌入在 `bus_type` 对象中的所有子系统都集合在一起。`bus_subsys` 子系统与目录 `/sys/bus` 是对应的；因此，例如，有一个 `/sys/bus/pci` 目录，它与 PCI 总线类型相对应。每种总线的子系统通常包括两个 `kset`，它们是 `drivers` 和 `devices`（分别对应于 `bus_type` 对象中的 `drivers` 和 `devices` 字段）。

名为 `drivers` 的 `kset` 包含描述符 `device_driver`，它描述与该总线类型相关的所有设备驱动程序，而名为 `devices` 的 `kset` 包含描述符 `device`，它描述给定总线类型上连接的所有设备。因为设备的 `kobject` 目录已经出现在 `/sys/devices` 下的 `sysfs` 文件系统中，所以每种总线子系统的 `devices` 目录存放了指向 `/sys/devices` 下目录的符号链接。`bus_for_each_drv()` 和 `bus_for_each_dev()` 函数分别用于循环扫描 `drivers` 和 `devices` 链表中的所有元素。

当内核检查一个给定的设备否可以由给定的驱动程序处理时，就会执行 `match` 方法。对于连接设备的总线而言，即使其上每个设备的标识符都拥有一个特定的格式，实现 `match` 方法的函数通常也很简单，因为它只需要在所支持标识符的驱动程序表中搜索设备的描述符。在设备驱动程序模型中注册某个设备时会执行 `hotplug` 方法；实现函数应该通过环境变量把总线的具体信息传递给用户态程序，以通告一个新的可用设备（参见后面的“注册设备驱动程序”一节）。最后，当特定类型总线上的设备必须改变其供电状态时，就会执行 `suspend` 和 `resume` 方法。

类

每个类是由一个 `class` 对象描述的。所有的类对象都属于与 `/sys/class` 目录相对应的 `class_subsys` 子系统。此外，每个类对象还包括一个内嵌的子系统；因此，例如有一个 `/sys/class/input` 目录，它就与设备驱动程序模型的 `input` 类相对应。

每个类对象包括一个 `class_device` 描述符链表，其中每个描述符描述了一个属于该类的单独逻辑设备。`class_device` 结构中包含一个 `dev` 字段，它指向一个设备描述符，因此一个逻辑设备总是对应于设备驱动程序模型中的一个给定的设备。然而，可以存在多个 `class_device` 描述符对应同一个设备。事实上，一个硬件设备可能包括几个不同的子设备，每个子设备都需要一个不同的用户态接口。例如，声卡就是一个硬件设备，它通常包括一个 DSP（digital signal processor，数字信号处理器）、一个混音器、一个游戏端口接口等等；每个子设备需要一个属于自己的用户态接口，因此 `sysfs` 文件系统中都有与它们相对应的目录。

同一类中的设备驱动程序可以对用户态应用程序提供相同的功能；例如，声卡上的所有设备驱动程序都提供一个可以向 DSP 中写入声音样本的方法。

设备驱动程序模型中的类本质上是要提供一个标准的方法,从而为向用户态应用程序导出逻辑设备的接口。每个 `class_device` 描述符中内嵌一个 `kobject`, 这是一个名为 `dev` 的属性 (特殊文件)。该属性存放设备文件的主设备号和次设备号, 通过它们可以访问相应的逻辑设备 (参见下一节)。

设备文件

正如在第一章中所提到的那样, 类 Unix 操作系统都是基于文件概念的, 文件是由字节序列而构成的信息载体。根据这一点, 可以把 I/O 设备当作设备文件 (*device file*) 这种所谓的特殊文件来处理; 因此, 与磁盘上的普通文件进行交互所用的同一系统调用可直接用于 I/O 设备。例如, 用同一 `write()` 系统调用既可以向普通文件中写入数据, 也可以通过向 `/dev/lp0` 设备文件中写入数据从而把数据发往打印机。

根据设备驱动程序的基本特性, 设备文件可以分为两种: 块和字符。这两种硬件设备之间的差异并不容易划分, 但我们至少可以假定以下的差异:

- 块设备的数据可以被随机访问, 而且从人类用户的观点看, 传送任何数据块所需的时间都是较少且大致相同的。块设备的典型例子是硬盘、软盘、CD-ROM 驱动器及 DVD 播放器。
- 字符设备的数据或者不可以被随机访问 (考虑声卡这样的例子), 或者可以被随机访问, 但是访问随机数据所需的时间很大程度上依赖于数据在设备内的位置 (考虑磁带驱动器这样的例子)。

网卡是这种模式的一种明显的例外, 因为网卡是不直接与设备文件相对应的硬件设备。

自从 Unix 操作系统早期版本以来, 设备文件就一直在使用。设备文件是存放在文件系统实际文件中的。然而, 它的索引节点并不包含指向磁盘上数据块 (文件的数据) 的指针, 因为它们是空的。相反, 索引节点必须包含硬件设备的一个标识符, 它对应字符或块设备文件。

传统上, 设备标识符由设备文件的类型 (字符或块) 和一对参数组成。第一个参数称为主设备号 (*major number*), 它标识了设备的类型。通常, 具有相同主设备号和类型的所有设备文件共享相同的文件操作集合, 因为它们是由同一个设备驱动程序处理的。第二个参数称为次设备号 (*minor number*), 它标识了主设备号相同的设备组中的一个特定设备。例如, 由相同的磁盘控制器管理的一组磁盘具有相同的主设备号和不同的次设备号。

`mknod()` 系统调用用来创建设备文件。其参数有设备文件名、设备类型、主设备号及次

设备号。设备文件通常包含在 `/dev` 目录中。表 13-7 显示了一些设备文件的属性。注意字符设备和块设备有独立的编号，因此，块设备(3,0)不同于字符设备(3,0)。

表 13-7: 设备文件的例子

设备名	类型	主设备号	次设备号	说明
<code>/dev/fd0</code>	块设备	2	0	软盘
<code>/dev/hda</code>	块设备	3	0	第一个 IDE 磁盘
<code>/dev/hda2</code>	块设备	3	2	第一个 IDE 磁盘上的第二个主分区
<code>/dev/hdb</code>	块设备	3	64	第二个 IDE 磁盘
<code>/dev/hdb3</code>	块设备	3	67	第二个 IDE 磁盘上的第三个主分区
<code>/dev/tty0</code>	字符设备	3	0	终端
<code>/dev/console</code>	字符设备	5	1	控制台
<code>/dev/lp1</code>	字符设备	6	1	并口打印机
<code>/dev/ttyS0</code>	字符设备	4	64	第一个串口
<code>/dev/rtc</code>	字符设备	10	135	实时时钟
<code>/dev/null</code>	字符设备	1	3	空设备 (黑洞)

设备文件通常与硬件设备（如硬盘 `/dev/hda`），或硬件设备的某一物理或逻辑分区（如磁盘分区 `/dev/hda2`）相对应。但在某些情况下，设备文件不会和任何实际的硬件对应，而是表示一个虚拟的逻辑设备。例如，`/dev/null` 就是一个和“黑洞”对应的设备文件，所有写入这个文件的数据都被简单地丢弃，因此，该文件看起来总为空。

就内核所关心的内容而言，设备文件名是无关紧要的。如果你建立了一个名为 `/tmp/disk` 的设备文件，类型为“块”，主设备号是 3，次设备号是 0，那么这个设备文件就和表 13-7 中的 `/dev/hda` 等价。另一方面，对某些应用程序来说，设备文件名可能就很有意义。例如，通信程序可能假设第一个串口和 `/dev/ttyS0` 设备文件对应。但是，通常可以把大部分应用程序设定为随意地与指定的设备文件进行交互。

设备文件的用户态处理

传统的 Unix 系统中（以及 Linux 的早期版本中），设备文件的主设备号和次设备号都是 8 位长。因此，最多只能有 65536 个块设备文件和 65536 个字符设备文件。你可能认为这些已经足够了，但遗憾的是它们并不够用。

真正的问题是设备文件被分配一次且永远保存在 `/dev` 目录中，因此，系统中的每个逻辑设备都应该有一个与其相对应的、明确定义了设备号的设备文件。`Documentation/devices.txt`

文件存放了官方注册的已分配设备号和 `/dev` 目录节点；`include /linux/major.h` 文件也可能包含设备的主设备号对应的宏。

不幸的是，如今各种不同的硬件设备数量惊人，几乎分配了所有的设备号。官方注册的设备号对于一般的 Linux 系统还能胜任；然而，它却不能很好地适用于大规模的系统。此外，高端系统可能使用数百或数千的同类型磁盘，因而 8 位的次设备号是远远不够的。例如，注册表为 16 个 SCSI 磁盘保留了设备号，而每个 SCSI 磁盘拥有 15 个分区；如果一个高端系统拥有多于 16 个的 SCSI 磁盘，那么必须改变原先主设备号和次设备号的标准分配——这是一个非常繁琐的工作，它需要改变内核源代码并且使得系统难以维护。

为了解决上述问题，Linux 2.6 已经增加了设备号的编码大小：目前主设备号的编码为 12 位，次设备号的编码为 20 位。通常把这两个参数合并成一个 32 位的 `dev_t` 变量；`MAJOR` 宏和 `MINOR` 宏可以从 `dev_t` 中分别提取主设备号和次设备号，而 `MKDEV` 宏可以把主设备号和次设备号合并成一个 `dev_t` 值。为了实现向后兼容，内核仍然可以正确地处理设备号编码为 16 位的老式设备文件。

官方注册表不能静态地分配这些附加的可用设备号，只有在处理设备号的特殊要求时才允许使用。事实上，对分配设备号和创建设备文件来说，如今更倾向的做法是高度动态地处理设备文件。

动态分配设备号

每个设备驱动程序在注册阶段都会指定它将要处理的设备号范围（参见后面的“注册设备驱动程序”一节）。然而，驱动程序可以只指定设备号的分配范围，无需指定精确的值：在这种情形下，内核会分配一个合适的设备号范围给驱动程序。

因此，新的硬件设备驱动程序不再需要从官方注册表中分配的一个设备号；它们可以仅仅使用当前系统中空闲的设备号。

然而，在这种情形下，就不能永久性地创建设备文件；它只在设备驱动程序初始化一个主设备号和次设备号时才创建。因此，这就需要有一个标准的方法将每个驱动程序所使用的设备号输出到用户态应用程序中。正如我们在前面“设备驱动程序模型的组件”一节所看到的，设备驱动程序模型提供了一个非常好的解决办法：把主设备号和次设备号存放在 `/sys/class` 子目录下的 `dev` 属性中。

动态创建设备文件

Linux 内核可以动态地创建设备文件：它无需把每一个可能想到的硬件设备的设备文件都填充到 `/dev` 目录下，因为设备文件可以按照需要来创建。由于设备驱动程序模型的存

在，Linux 2.6 内核提供了一个非常简单的方法来处理这个问题。系统中必须安装一组称为 *udev* 工具集的用户态程序。当系统启动时，*/dev* 目录是清空的，这时 *udev* 程序将扫描 */sys/class* 子目录来寻找 *dev* 文件。对每一个这样的文件（主设备号和次设备号的组合表示一个内核所支持的逻辑设备文件），*udev* 程序都会在 */dev* 目录下为它创建一个相应的设备文件。*udev* 程序也会根据配置文件为其分配一个文件名并创建一个符号链接，该方法类似于 Unix 设备文件的传统命名模式。最后，*/dev* 目录里只存放了系统中内核所支持的所有设备的设备文件，而没有任何其他的文件。

通常在系统初始化后才创建设备文件。它要么发生在加载设备驱动程序（系统尚未支持该设备）所在的模块时，要么发生在一个热拔插的设备（如 USB 外围设备）加入系统中时。*udev* 工具集可以自动地创建相应的设备文件，因为设备驱动程序模型支持设备的热插拔。当发现一个新的设备时，内核会产生一个新的进程来执行用户态 shell 脚本文件 */sbin/hotplug*（注 3），并将新设备上的有用信息作为环境变量传递给 shell 脚本。用户态脚本文件读取配置文件信息并关注完成新设备初始化所必需的任何操作。如果安装了 *udev* 工具集，脚本文件也会在 */dev* 目录下创建适当的设备文件。

设备文件的 VFS 处理

虽然设备文件也在系统的目录树中，但是它们和普通文件以及目录文件有根本的不同。当进程访问普通文件时，它会通过文件系统访问磁盘分区中的一些数据块；而在进程访问设备文件时，它只要驱动硬件设备就可以了。例如，进程可以访问一个设备文件以从连接到计算机的温度计读取房间的温度。为应用程序隐藏设备文件与普通文件之间的差异正是 VFS 的责任。

为了做到这点，VFS 在设备文件打开时改变其缺省文件操作；因此，可以把设备文件的每个系统调用都转换成与设备相关的函数的调用，而不是对主文件系统相应函数的调用。与设备相关的函数对硬件设备进行操作以完成进程所请求的操作（注 4）。

让我们假定进程在设备文件（块或字符类型）上执行 *open()* 系统调用。这个系统调用所执行的操作已经在第十二章“*open()* 系统调用”一节进行了描述。从本质上说，相应的服务例程解析到设备文件的路径名，并建立相应的索引节点对象、目录项对象和文件对象。

注 3： 可以通过写 */proc/sys/kernel/hotplug* 文件改变在发生热插拔事件时所调用的用户态程序的路径名。

注 4： 注意，根据第十二章中的“路径名查找”一节中介绍的命名解析机制，指向设备文件的符号链接与设备文件的作用相同。

通过适当的文件系统函数（通常为 `ext2_read_inode()` 或 `ext3_read_inode()`；参见第十八章）读取磁盘上的相应索引节点来对索引节点对象进行初始化。当这个函数确定磁盘索引节点与设备文件对应时，则调用 `init_special_inode()`，该函数把索引节点对象的 `i_rdev` 字段初始化为设备文件的主设备号和次设备号，而把索引节点对象的 `i_fop` 字段设置为 `def_blk_fops` 或者 `def_chr_fops` 文件操作表的地址（根据设备文件的类型）。因此，`open()` 系统调用的服务例程也调用 `dentry_open()` 函数，后者分配一个新的文件对象并把其 `f_op` 字段设置为 `i_fop` 中存放的地址，即再一次指向 `def_blk_fops` 或 `def_chr_fops` 的地址。正是这两个表的引入，才使得在设备文件上所发出的任何系统调用都将激活设备驱动程序的函数而不是基本文件系统的函数。

设备驱动程序

设备驱动程序是内核例程的集合，它使得硬件设备响应控制设备的编程接口，而该接口是一组规范的 VFS 函数集（`open`，`read`，`lseek`，`ioctl` 等等）。这些函数的实际实现由设备驱动程序全权负责。由于每个设备都有一个唯一的 I/O 控制器，因此就有唯一的命令和唯一的状态信息，所以大部分 I/O 设备都有自己的驱动程序。

设备驱动程序的种类有很多。它们在对用户态应用程序提供支持的级别上有很大的不同，也对来自硬件设备的数据采集有不同的缓冲策略。这些选择极大地影响了设备驱动程序的内部结构，我们将在“直接内存访问（DMA）”和“字符设备的缓冲策略”两节进行讨论。

设备驱动程序并不仅仅由实现设备文件操作的函数组成。在使用设备驱动程序之前，有几个活动是肯定要发生的。我们将在下面几节考察它们。

注册设备驱动程序

我们知道在设备文件上发出的每个系统调用都由内核转化为对相应设备驱动程序的对应函数的调用。为了完成这个操作，设备驱动程序必须注册自己。换句话说，注册一个设备驱动程序意味着分配一个新的 `device_driver` 描述符，将其插入到设备驱动程序模型的数据结构中（参见“设备驱动程序模型的组件”一节），并把它与对应的设备文件（可能是多个设备文件）连接起来。如果设备文件对应的驱动程序以前没有注册，则对该设备文件的访问会返回错误码 `-ENODEV`。

如果设备驱动程序被静态地编译进内核，则它的注册在内核初始化阶段进行。相反，如果驱动程序是作为一个内核模块来编译的（参见附录二），则它的注册在模块装入时进行。在后一种情况下，设备驱动程序也可以在模块卸载时注销自己。

例如，我们考虑一个通用的PCI设备。为了能正确地对其进行处理，其设备驱动程序必须分配一个pci_driver类型的描述符，PCI内核层使用该描述符来处理设备。初始化描述符的一些字段后，设备驱动程序就会调用pci_register_driver()函数。事实上，pci_driver描述符包括一个内嵌的device_driver描述符（参见前面的“设备驱动程序模型的组件”一节）；pci_register_driver()函数仅仅初始化内嵌的驱动程序描述符中的字段，然后调用driver_register()函数把驱动程序插入设备驱动程序模型的数据结构中。

注册设备驱动程序时，内核会寻找可能由该驱动程序处理但还尚未获得支持的硬件设备。为了做到这点，内核主要依靠相关的总线类型描述符bus_type的match方法，以及device_driver对象的probe方法。如果探测到可被驱动程序处理的硬件设备，内核会分配一个设备对象，然后调用device_register()函数把设备插入设备驱动程序模型中。

初始化设备驱动程序

对设备驱动程序进行注册和初始化是两件不同的事。设备驱动程序应当尽快被注册，以便用户态应用程序能通过相应的设备文件使用它。相反，设备驱动程序在最后可能的时刻才被初始化。事实上，初始化驱动程序意味着分配宝贵的系统资源，这些资源因此就对其他驱动程序不可用了。

我们已经在第四章“I/O中断处理”一节看到一个例子：把IRQ分配给设备通常是自动进行的，这正好发生在使用设备之前，因为多个设备可能共享同一条IRQ线。其他可以在最后时刻被分配的资源是用于DMA传送缓冲区的页框和DMA通道本身（用于像软盘驱动器那样的老式非PCI设备）。

为了确保资源在需要时能够获得，在获得后不再被请求，设备驱动程序通常采用下列模式：

- 引用计数器记录当前访问设备文件的进程数。在设备文件的open方法中计数器被增加，在release方法中被减少（注5）。
- open方法在增加引用计数器的值之前先检查它。如果计数器为0，则设备驱动程序必须分配资源并激活硬件设备上的中断和DMA。
- release方法在减少使用计数器的值之后检查它。如果计数器为0，说明已经没有进程使用这个硬件设备。如果是这样，该方法将禁止I/O控制器上的中断和DMA，然后释放所分配的资源。

注5：更确切地说，引用计数器记录引用设备文件的文件对象的个数，因为子进程可能共享文件对象。

监控 I/O 操作

I/O 操作的持续时间通常是不可预知的。这可能和机械装置的情况有关（对于要传送的数据块来说是磁头的当前位置），和实际的随机事件有关（数据包什么时候到达网卡），还和人为因素有关（用户在键盘上按下一个键或者发现打印机夹纸了）。在任何情况下，启动 I/O 操作的设备驱动程序都必须依靠一种监控技术在 I/O 操作终止或超时时发出信号。

在终止操作的情况下，设备驱动程序读取 I/O 接口状态寄存器的内容来确定 I/O 操作是否成功执行。在超时的情况下，驱动程序知道一定出了问题，因为完成操作所允许的最大时间间隔已经用完，但什么也没做。

监控 I/O 操作结束的两种可用技术分别称为轮询模式 (*polling mode*) 和中断模式 (*interrupt mode*)。

轮询模式

CPU 依照这种技术重复检查（轮询）设备的状态寄存器，直到寄存器的值表明 I/O 操作已经完成为止。我们已经在第五章的“自旋锁”一节中提到一种基于轮询的技术：当处理器试图获得一个繁忙的自旋锁时，它就重复地查询变量的值，直到该值变成 0 为止。但是，应用到 I/O 操作中的轮询技术更加巧妙，这是因为驱动程序还必须记住检查可能的超时。下面是轮询的一个简单例子：

```
for (;;) {
    if (read_status(device) & DEVICE_END_OPERATION) break;
    if (--count == 0) break;
}
```

在进入循环之前，count 变量已被初始化，每次循环都对 count 的值减 1，因此就可以使用这个变量实现一种粗略的超时机制。另外，更精确的超时机制可以通过这样的方法实现：在每次循环时读取节拍计数器 jiffies 的值（请参看第六章中的“更新时间和日期”一节），并将它与开始等待循环之前读取的原值进行比较。

如果完成 I/O 操作需要的时间相对较多，比如说毫秒级，那么这种模式就变得低效，因为 CPU 花费宝贵的机器周期去等待 I/O 操作的完成。在这种情况下，在每次轮询操作之后，可以通过把 schedule() 的调用插入到循环内部来自愿放弃 CPU。

中断模式

如果 I/O 控制器能够通过 IRQ 线发出 I/O 操作结束的信号，那么中断模式才能被使用。

我们现在通过一个简单的例子说明中断模式如何工作。假定我们想实现一个简单的输入

字符设备的驱动程序。当用户在相应的设备文件上发出 `read()` 系统调用时，一条输入命令被发往设备的控制寄存器。在一个不可预知的长时间间隔后，设备把一个字节的数据放进输入寄存器。设备驱动程序然后将这个字节作为 `read()` 系统调用的结果返回。

这是一个用中断模式实现驱动程序的典型例子。实质上，驱动程序包含两个函数：

1. 实现文件对象 `read` 方法的 `foo_read()` 函数。
2. 处理中断的 `foo_interrupt()` 函数。

只要用户读设备文件，`foo_read()` 函数就被触发：

```
ssize_t foo_read(struct file *filp, char *buf, size_t count,
                 loff_t *ppos)
{
    foo_dev_t * foo_dev = filp->private_data;
    if (down_interruptible(&foo_dev->sem)
        return -ERESTARTSYS;
    foo_dev->intr = 0;
    outb(DEV_FOO_READ, DEV_FOO_CONTROL_PORT);
    wait_event_interruptible(foo_dev->wait, (foo_dev->intr= 1));
    if (put_user(foo_dev->data, buf))
        return -EFAULT;
    up(&foo_dev->sem);
    return 1;
}
```

设备驱动程序依赖类型为 `foo_dev_t` 的自定义描述符，它包含信号量 `sem`（保护硬件设备免受并发访问）、等待队列 `wait`、标志 `intr`（当设备发出一个中断时设置）及单个字节缓冲区 `data`（由中断处理程序写入且由 `read` 方法读取）。一般而言，所有使用中断的 I/O 驱动程序都依赖中断处理程序及 `read` 和 `write` 方法均访问的数据结构。`foo_dev_t` 描述符的地址通常存放在设备文件的文件对象的 `private_data` 字段中或一个全局变量中。

`foo_read()` 函数的主要操作如下：

1. 获取 `foo_dev->sem` 信号量，因此确保没有其他进程访问该设备。
2. 清 `intr` 标志。
3. 对 I/O 设备发出读命令。
4. 执行 `wait_event_interruptible` 以挂起进程，直到 `intr` 标志变为 1。这个宏已在第三章“等待队列”一节描述过。

一定时间后，我们的设备发出中断信号以通知 I/O 操作已经完成，数据已经放在适当的 `DEV_FOO_DATA_PORT` 数据端口。中断处理程序置 `intr` 标志并唤醒进程。当调度程序决定重新执行这个进程时，`foo_read()` 的第二部分被执行，步骤如下：

1. 把准备在 `foo_dev->data` 变量中的字符拷贝到用户地址空间。
2. 释放 `foo_dev->sem` 信号量后终止。

为了简单起见，我们没有包含任何超时控制。一般来说，超时控制是通过静态或动态定时器实现的（参见第六章），定时器必须设置为启动 I/O 操作后正确的时间，并在操作结束时删除。

让我们来看一下 `foo_interrupt()` 函数的代码：

```
void foo_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    foo->data = inb(DEV_FOO_DATA_PORT);
    foo->intr = 1;
    wake_up_interruptible(&foo->wait);
    return 1;
}
```

中断处理程序从设备的输入寄存器中读字符，并把它存放在 `foo` 全局变量指向的驱动程序描述符 `foo_dev_t` 的 `data` 字段中。然后设置 `intr` 标志，并调用 `wake_up_interruptible()` 函数唤醒在 `foo->wait` 等待队列上阻塞的进程。

注意，三个参数中没有一个是被中断处理程序使用，这是相当普遍的情况。

访问 I/O 共享存储器

根据设备和总线的类型，PC 体系结构里的 I/O 共享存储器可以被映射到不同的物理地址范围。主要有：

对于连接到 ISA 总线上的大多数设备

I/O 共享存储器通常被映射到 `0xa0000~0xfffff` 的 16 位物理地址范围；这就在 640 KB 和 1 MB 之间留出了一段空间，就是我们在第二章的“物理内存布局”一节中所介绍的那个“空洞”。

对于连接到 PCI 总线上的设备

I/O 共享存储器被映射到接近 4 GB 的 32 位物理地址范围。这种类型的设备更容易处理。

几年以前，Intel 引入了图形加速端口 (AGP) 标准，该标准是适合于高性能图形卡的 PCI 的增强。这种卡除了有自己的 I/O 共享存储器外，还能够通过图形地址再映像表 (GART) 这个特殊的硬件电路直接对主板的 RAM 部分进行寻址。GART 电路能够使 AGP 卡比老式的 PCI 卡具有更高的数据传输速率。然而，从内核的观点看，物理存储器位于何处根本没有什么关系，GART 映射的存储器与其他种类 I/O 共享存储器的处理方式完全一样。

设备驱动程序如何访问一个I/O共享存储器单元？让我们从比较简单的PC体系结构开始入手，之后再扩展到其他体系结构。

不要忘了内核程序作用于线性地址，因此I/O共享存储器单元必须表示成大于PAGE_OFFSET的地址。在后面的讨论中，我们假设PAGE_OFFSET等于0xc0000000，也就是说，内核线性地址是在第4个GB。

设备驱动程序必须把I/O共享存储器单元的物理地址转换成内核空间的线性地址。在PC体系结构中，这可以简单地把32位的物理地址和0xc0000000常量进行或运算得到。例如，假设内核需要把物理地址为0x000b0fe4的I/O单元的值存放在t1中，把物理地址为0xfc000000的I/O单元的值存放在t2中。你可能认为使用下面的表达式就可以完成这项工作：

```
t1 = *((unsigned char *) (0xc00b0fe4));
t2 = *((unsigned char *) (0xfc000000));
```

在初始化阶段，内核已经把可用的RAM物理地址映射到线性地址空间第4个GB的开始部分。因此，分页单元把出现在第一个语句中的线性地址0xc00b0fe4映射回到原来的I/O物理地址0x000b0fe4，这正好落在从640KB到1MB的这段“ISA洞”中（请参看第二章的“Linux中的分页”一节）。这工作得很好。

但是，对于第二个语句来说，这里有一个问题，因为其I/O物理地址超过了系统RAM的最大物理地址。因此，线性地址0xfc000000就不需要与物理地址0xfc000000相对应。在这种情况下，为了在内核页表中包括对这个I/O物理地址进行映射的线性地址，必须对页表进行修改。这可以通过调用ioremap()或ioremap_nocache()函数来实现。第一个函数与vmalloc()函数类似，都调用get_vm_area()为所请求的I/O共享存储器区的大小建立一个新的vm_struct描述符（请参看第八章中的“非连续内存器区的描述符”一节）。然后，这两个函数适当地更新常规内核页表中的对应页表项。ioremap_nocache()不同于ioremap()，因为前者在适当地引用再映射的线性地址时还使硬件高速缓存内容失效。

因此，第二个语句的正确形式应该为：

```
io_mem = ioremap(0xf0000000, 0x200000);
t2 = *((unsigned char *) (io_mem + 0x100000));
```

第一条语句建立一个2MB的新的线性地址区间，该区间映射了从0xf0000000开始的物理地址；第二条语句读取地址为0xfc000000的内存单元。设备驱动程序以后要取消这种映射，就必须使用iounmap()函数。

在其他体系结构（PC 之外的体系结构）上，简单地间接引用物理内存单元的线性地址并不能正确访问 I/O 共享存储器。因此，Linux 定义了下列依赖于体系结构的函数，当访问 I/O 共享存储器时来使用它们：

`readb()`, `readw()`, `readl()`

分别从 I/O 共享存储器单元读取 1、2 或者 4 个字节

`writeb()`, `writew()`, `writel()`

分别向 I/O 共享存储器单元写入 1、2 或者 4 个字节

`memcpy_fromio()`, `memcpy_toio()`

把一个数据块从一个 I/O 共享存储器单元拷贝到动态内存中，另一个函数正好相反

`memset_io()`

用一个固定的值填充一个 I/O 共享存储器区域

因此，对于 `0xfc000000` I/O 单元的访问推荐使用这样的方法：

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = readb(io_mem + 0x100000);
```

正是由于这些函数，就可以隐藏不同平台访问 I/O 共享存储器所用方法的差异。

直接内存访问（DMA）

在最初的 PC 体系结构中，CPU 是系统中唯一的总线主控器，也就是说，为了提取和存储 RAM 存储单元的值，CPU 是唯一可以驱动地址 / 数据总线的硬件设备。随着更多诸如 PCI 这样的现代总线体系结构的出现，如果提供合适的电路，每一个外围设备都可以充当总线主控器。因此，现在所有的 PC 都包含一个辅助的 DMA 电路，它可以用来控制在 RAM 和 I/O 设备之间数据的传送。DMA 一旦被 CPU 激活，就可以自行传送数据；当数据传送完成之后，DMA 发出一个中断请求。当 CPU 和 DMA 同时访问同一内存单元时，所产生的冲突由一个名为内存仲裁器（参见第五章中的“原子操作”一节）的硬件电路来解决。

使用 DMA 最多的是磁盘驱动器和其他需要一次传送大量字节的设备。因为 DMA 的设置时间相当长，所以在传送数量很少的数据时直接使用 CPU 效率更高。

原来的 ISA 总线所使用的 DMA 电路非常复杂，难于对其进行编程，并且限于物理内存的低 16MB。PCI 和 SCSI 总线所使用的最新 DMA 电路依靠总线中的专用硬件电路，这就简化了设备驱动程序开发人员的开发工作。

同步 DMA 和异步 DMA

设备驱动程序可以采用两种方式使用 DMA，分别是同步 DMA 和异步 DMA。第一种方式，数据的传送是由进程触发的；而第二种方式，数据的传送是由硬件设备触发的。

采用同步 DMA 传送的例子如声卡，它可以播放电影音乐。用户态应用程序将声音数据（称为样本）写入一个与声卡的数字信号处理器（DSP）相对应的设备文件中。声卡的驱动程序把写入的这些样本收集在内核缓冲区中。同时，驱动程序命令声卡把这些样本从内核缓冲区拷贝到预先定时的 DSP 中。当声卡完成数据传送时，就会引发一个中断，然后驱动程序会检查内核缓冲区是否还有要播放的样本，如果没有，驱动程序就再启动一次 DMA 数据传送。

采用异步 DMA 传送的例子如网卡，它可以从一个 LAN 中接收帧（数据包）。网卡将接收到的帧存储在自己的 I/O 共享存储器中，然后引发一个中断。其驱动程序确认该中断后，命令网卡将接收到的帧从 I/O 共享存储器拷贝到内核缓冲区。当数据传送完成后，网卡会引发新的中断，然后驱动程序将这个新帧通知给上层内核层。

DMA 传送的辅助函数

当为使用 DMA 传送方式的设备设计驱动程序时，开发者编写的代码应该与体系结构和总线（就 DMA 传送方式来说）二者都不相关。由于内核提供了丰富的 DMA 辅助函数，因而现在上述目标是切实可行的。这些辅助函数隐藏了不同硬件体系结构的 DMA 实现机制的差异。

这是 DMA 辅助函数的两个子集：老式的子集为 PCI 设备提供了与体系结构无关的函数，新的子集则保证了与总线和体系结构两者都无关。我们现在将介绍其中的一些函数，同时指出 DMA 的一些硬件特性。

总线地址

DMA 的每次数据传送（至少）需要一个内存缓冲区，它包含硬件设备要读出或写入的数据。一般而言，启动一次数据传送前，设备驱动程序必须确保 DMA 电路可以直接访问 RAM 内存单元。

到现在为止，我们已区分了三类存储器地址：逻辑地址、线性地址以及物理地址，前两个在 CPU 内部使用，最后一个为 CPU 从物理上驱动数据总线所用的存储器地址。但是，还有第四种存储器地址，称为总线地址（*bus address*），它是除 CPU 之外的硬件设备驱动数据总线时所用的存储器地址。

从根本上说，内核为什么应该关心总线地址呢？这是因为在 DMA 操作中，数据传送不

需要CPU的参与；I/O设备和DMA电路直接驱动数据总线。因此，当内核开始DMA操作时，必须把所涉及的内存缓冲区总线地址或写入DMA适当的I/O端口，或写入I/O设备适当的I/O端口。

在80x86体系结构中，总线地址与物理地址是一致的。然而，其他的体系结构例如Sun公司的SPARC和HP的Alpha都包括一个所谓的I/O存储器管理单元(IO-MMU)的硬件电路，它类似于微处理器的分页单元，将物理地址映射为总线地址。使用DMA的所有I/O驱动程序在启动一次数据传送前必须设置好IO-MMU。

不同的总线具有不同的总线地址大小。例如，ISA的总线地址是24位长，因此，在80x86体系结构中，可以在物理内存的低16 MB中完成DMA传送——这就是为什么DMA使用的内存缓冲区分配在ZONE_DMA内存区中（设置了GFP_DMA标志）。原来的PCI标准定义了32位的总线地址；但是，一些PCI硬件设备最初是为ISA总线而设计的，因此它们仍然访问不了物理地址0x00ffffff以上的RAM内存单元。新的PCI-X标准采用64位的总线地址并允许DMA电路可以直接寻址更高的内存。

在Linux中，数据类型dma_addr_t代表一个通用的总线地址。在80x86体系结构中，dma_addr_t对应一个32位长的整数，除非内核支持PAE[参见第二章的“物理地址扩展(PAE)分页机制”一节]，在这种情形下，dma_addr_t代表一个64位的整数。

pci_set_dma_mask()和dma_set_mask()两个辅助函数用于检查总线是否可以接收给定大小的总线地址(mask)，如果可以，则通知总线层给定的外围设备将使用该大小的总线地址。

高速缓存的一致性

系统体系结构没有必要在硬件级为硬件高速缓存与DMA电路之间提供一个一致性协议，因此，执行DMA映射操作时，DMA辅助函数必须考虑硬件高速缓存。为了弄清楚这是为什么，假设设备驱动程序把一些数据填充到内存缓冲区中，然后立刻命令硬件设备利用DMA传送方式读取该数据。如果DMA访问这些物理RAM内存单元，而相应的硬件高速缓存行的内容还没有写入RAM中，那么硬件设备所读取的值就是内存缓冲区中的旧值。

设备驱动程序开发人员可以采用两种方法来处理DMA缓冲区，他们分别使用两类不同的辅助函数来完成。用Linux的术语来说，开发人员在下面两种DMA映射类型中进行选择：

一致性DMA映射

使用这种映射方式时，内核必须保证内存与硬件设备间高速缓存一致性不是什么问

题，也就是说CPU在RAM内存单元上所执行的每个写操作对硬件设备而言都是立即可见的，反过来也一样。这种映射方式也称为“同步的”或“一致的”。

流式DMA映射

使用这种映射方式时，设备驱动程序必须了解高速缓存一致性问题，这可以使用适当的同步辅助函数来解决。这种映射方式也称为“异步的”或“非一致性的”。

在80x86体系结构中使用DMA时，从不存在高速缓存一致性问题根本不是什么问题，因为硬件设备驱动程序本身会“窥探”所访问的硬件高速缓存。因此，80x86体系结构中为硬件设备所设计的驱动程序会从前述的两种DMA映射方式中选择一个：它们二者在本质上是等价的。另一方面，在诸如MIPS、SPARC以及PowerPC的一些模型等许多其他的体系结构中，硬件设备通常不窥探硬件高速缓存，因而就会产生高速缓存一致性问题。总的来说，为与体系结构无关的驱动程序选择一个合适的DMA映射方式是很重要的。

一般来说，如果CPU和DMA处理器以不可预知的方式去访问一个缓冲区，那么必须强制使用一致性DMA映射方式（例如，SCSI适配器的command数据结构的缓冲区）。其他情形下，流式DMA映射方式更可取，因为在一些体系结构中处理一致性DMA映射是很麻烦的，并且可能导致更低的系统性能。

一致性DMA映射的辅助函数

通常，设备驱动程序在初始化阶段会分配内存缓冲区并建立一致性DMA映射；在卸载时释放映射和缓冲区。为了分配内存缓冲区和建立一致性DMA映射，内核提供了依赖体系结构的`pci_alloc_consistent()`和`dma_alloc_coherent()`两个函数。它们均返回新缓冲区的线性地址和总线地址。在80x86体系结构中，它们返回新缓冲区的线性地址和物理地址。为了释放映射和缓冲区，内核提供了`pci_free_consistent()`和`dma_free_coherent()`两个函数。

流式DMA映射的辅助函数

流式DMA映射的内存缓冲区通常在数据传送之前被映射，在传送之后被取消映射。也有可能几次DMA传送过程中保持相同的映射，但是在这种情况下，设备驱动程序开发人员必须知道位于内存和外围设备之间的硬件高速缓存。

为了启动一次流式DMA数据传送，驱动程序必须首先利用分区页框分配器（参见第八章的“分区页框分配器”一节）或通用内存分配器（参见第八章的“通用对象”一节）来动态地分配内存缓冲区。然后，驱动程序调用`pci_map_single()`函数或者`dma_map_single()`函数建立流式DMA映射，这两个函数接收缓冲区的线性地址作为其参数并返回相应的总线地址。为了释放该映射，驱动程序调用相应的`pci_unmap_single()`函数或`dma_unmap_single()`函数。

为了避免高速缓存一致性问题，驱动程序在开始从RAM到设备的DMA数据传送之前，如果有必要，应该调用`pci_dma_sync_single_for_device()`函数或`dma_sync_single_for_device()`函数刷新与DMA缓冲区对应的高速缓存行。同样地，从设备到RAM的一次DMA数据传送完成之前设备驱动程序是不可以访问内存缓冲区的；相反，如果有必要，在读缓冲区之前，驱动程序应该调用`pci_dma_sync_single_for_cpu()`函数或`dma_sync_single_for_cpu()`函数使相应的硬件高速缓存行无效。在80x86体系结构中，上述函数几乎不做任何事情，因为硬件高速缓存和DMA之间的一致性是由硬件来维护的。

即使是高端内存的缓冲区（参见第八章的“高端内存页框的内核映射”一节）也可以用于DMA传送；开发人员使用`pci_map_page()`或`dma_map_page()`函数，给其传递的参数为缓冲区所在页的描述符地址和页中缓冲区的偏移地址。相应地，为了释放高端内存缓冲区的映射，开发人员使用`pci_unmap_page()`或`dma_unmap_page()`函数。

内核支持的级别

Linux内核并不完全支持所有可能存在的I/O设备。一般来说，事实上有三种可能的方式支持硬件设备：

根本不支持

应用程序使用适当的in和out汇编语言指令直接与设备的I/O端口进行交互。

最小支持

内核不识别硬件设备，但能识别它的I/O接口。用户程序把I/O接口视为能够读写字符流的顺序设备。

扩展支持

内核识别硬件设备，并处理I/O接口本身。事实上，这种设备可能就没有对应的设备文件。

第一种方式与内核设备驱动程序毫无关系，最常见的例子是X Window系统对图形显示的传统处理方式。这种方法效率很高，尽管它限制了X服务器使用I/O设备产生的硬件中断。为了让X服务器访问所请求的I/O端口，这种方法还需要做一些其他努力。正如第三章的“任务状态段”一节中所介绍的那样，`iopl()`和`ioperm()`系统调用给进程授权访问I/O端口。只有具有root权限的用户才可以调用这两个系统调用。但是通过设置可执行文件的`setuid`标志，普通用户也可以使用这些程序（参见第二十章中的“进程的信任状和权能”一节）。

新近的Linux版本支持几种广泛使用的图形卡。`/dev/fb`设备文件为图形卡的帧缓冲区提供了一种抽象，并允许应用软件无需知道图形接口的I/O端口的任何事情就可以访问它。

此外，内核提供了直接绘制基本架构（Direct Rendering Infrastructure, DRI），DRI 允许应用软件充分挖掘 3D 加速图形卡的硬件特性。不管怎样，传统的“自己动手配置”X Window 系统服务器还依然被广泛采用。

最小支持方法是用来处理连接到通用 I/O 接口上的外部硬件设备的。内核通过提供设备文件（由此而提供一个设备驱动程序）来处理 I/O 接口；应用程序通过读写设备文件来处理外部硬件设备。

最小支持优于扩展支持，因为它保持内核尽可能小。但是，在基于 PC 的通用 I/O 接口之中，只有串口和并口的处理使用了这种方法。因此，诸如 X 服务器之类的应用程序可以直接控制串口鼠标，而串口调制解调器通常都需要一个诸如 Minicom、Seyon 或 PPP（点对点协议）守护进程之类的通信程序。

最小支持的应用范围是有限的，因为当外部设备必须频繁地与内核内部数据结构进行交互时不能使用这种方法。例如，考虑一个连到通用 I/O 接口上的可移动硬盘。应用程序不能和所有的内核数据结构进程交互，也不能与识别磁盘所需要的函数和装载文件系统所需要的函数进行交互，因此，这种情况下就必须使用扩展支持。

一般情况下，直接连接到 I/O 总线上的任何硬件设备（如内置硬盘）都要根据扩展支持方法进行处理：内核必须为每个这样的设备提供一个设备驱动程序。通用串行总线（USB）、笔记本电脑上的 PCMCIA 接口或者 SCSI 接口——简而言之，除串口和并口之外的所有通用 I/O 接口之上连接的外部设备都需要扩展支持。

值得注意的是，与标准文件相关的系统调用，如 `open()`、`read()` 和 `write()`，并不总让应用程序完全控制底层硬件设备。事实上，VFS 的“最小公分母（lowest-common-denominator）”方法没有包含某些设备所需的特殊命令，或不让应用程序检查设备是否处于某一特殊的内部状态。

已引入的 `ioctl()` 系统调用可以满足这样的需要。这个系统调用除了设备文件的文件描述符和另一个表示请求的 32 位参数之外，还可以接收任意多个额外的参数。例如，特殊的 `ioctl()` 请求可以用来获得 CD-ROM 的音量或者弹出 CD-ROM 介质。应用程序可以用这类 `ioctl()` 请求提供一个 CD 播放器的用户接口。

字符设备驱动程序

处理字符设备相对比较容易，因为通常并不需要复杂的缓冲策略，也不涉及磁盘高速缓存。当然，字符设备在它们的需求方面有所不同：有些必须实现复杂的通信协议以驱动硬件设备，而有些仅仅需要从硬件设备的一对 I/O 端口读几个值。例如，多端口串口卡设备（一个硬件设备提供多个串口）的驱动程序比总线鼠标的设备驱动程序要复杂得多。

另一方面，块设备驱动程序本身就比字符设备驱动程序复杂得多。事实上，应用程序可以反复地要求读或写同一个数据块。此外，访问这些设备通常是很慢的。这些特性对磁盘驱动程序的结构产生了深刻的影响。然而，就如我们将在下一章看到的，内核提供了诸如页面高速缓存和块 I/O 子系统这些高级组件去处理驱动程序。在本章剩下的部分中我们把注意力集中于字符设备驱动程序。

字符设备驱动程序是由一个 `cdev` 结构描述的，其字段如表 13-8 所示。

表 13-8: `cdev` 结构中的字段

类型	字段	说明
<code>struct kobject</code>	<code>kobj</code>	内嵌的 <code>kobject</code>
<code>struct module *</code>	<code>owner</code>	指向实现驱动程序模块（如果有的话）的指针
<code>struct file_operations *</code>	<code>ops</code>	指向设备驱动程序文件操作表的指针
<code>struct list_head</code>	<code>list</code>	与字符设备文件对应的索引节点链表的头
<code>dev_t</code>	<code>dev</code>	给设备驱动程序所分配的初始主设备号和次设备号
<code>unsigned int</code>	<code>count</code>	给设备驱动程序所分配的设备号范围的大小

`list` 字段是双向循环链表的首部，该链表用于收集相同字符设备驱动程序所对应的字符设备文件的索引节点。可能很多设备文件具有相同的设备号，并对应于相同的字符设备。此外，一个设备驱动程序对应的设备号可以是一个范围，而不仅仅是一个号，设备号位于同一范围内的所有设备文件均由同一个字符设备驱动程序处理。设备号范围的大小存放在 `count` 字段中。

`cdev_alloc()` 函数的功能是动态地分配 `cdev` 描述符，并初始化内嵌的 `kobject` 数据结构，因此在引用计数器的值变为 0 时会自动释放该描述符。

`cdev_add()` 函数的功能是在设备驱动程序模型中注册一个 `cdev` 描述符。它初始化 `cdev` 描述符中的 `dev` 和 `count` 字段，然后调用 `kobj_map()` 函数。`kobj_map()` 则依次建立设备驱动程序模型的数据结构，把设备号范围复制到设备驱动程序的描述符中。

设备驱动程序模型为字符设备定义了一个 `kobject` 映射域，该映射域由一个 `kobj_map` 类型的描述符描述，并由全局变量 `cdev_map` 引用。`kobj_map` 描述符包括一个散列表，它有 255 个表项，并由 0~255 范围的主设备号进行索引。散列表存放 `probe` 类型的对象，每个对象都拥有一个已注册的主设备号和次设备号，其中各字段如表 13-9 所示。

表 13-9: probe 对象中的字段

类型	字段	说明
struct probe *	next	散列冲突链表中的下一个元素
dev_t	dev	设备号范围的初始设备号 (主、次设备号)
unsigned long	range	设备号范围的大小
struct module *	owner	如果有的话, 指向实现设备驱动程序模块的指针
struct kobject *(*)(dev_t, int *, void *)	get	探测谁拥有这个设备号范围
int (*)(dev_t, void *)	lock	增加设备号范围内拥有者的引用计数器
void *	data	设备号范围内拥有者的私有数据

调用 `kobj_map()` 函数时, 把指定的设备号范围加入到散列表中。相应的 `probe` 对象的 `data` 字段指向设备驱动程序的 `cdev` 描述符。执行 `get` 和 `lock` 方法时把 `data` 字段的值传递给它们。在这种情况下, `get` 方法通过一个简捷函数实现, 其返回值为 `cdev` 描述符中内嵌的 `kobject` 数据结构的地址; 相反, `lock` 方法本质上用于增加内嵌的 `kobject` 数据结构的引用计数器的值。

`kobj_lookup()` 函数接收 `kobject` 映射域和设备号作为输入参数; 它搜索散列表, 如果找到, 则返回该设备号所在范围的拥有者的 `kobject` 的地址。当这个函数应用到字符设备的映射域时, 就返回设备驱动程序描述符 `cdev` 中所嵌入的 `kobject` 的地址。

分配设备号

为了记录目前已经分配了哪些字符设备号, 内核使用散列表 `chrdevs`, 表的大小不超过设备号范围。两个不同的设备号范围可能共享同一个主设备号, 但是范围不能重叠, 因此它们的次设备号应该完全不同。 `chrdevs` 包含 255 个表项, 由于散列函数屏蔽了主设备号的高四位——因此, 主设备号的个数少于 255 个, 它们被散列到不同的表项中。每个表项指向冲突链表的第一个元素, 而该链表是按主、次设备号的递增顺序进行排序的。

冲突链表中的每个元素是一个 `char_device_struct` 结构, 其各字段如表 13-10 所示。

表 13-10: `char_device_struct` 描述符中的字段

类型	字段	说明
unsigned char_device_struct *	next	指向散列冲突链表中下一个元素的指针
unsigned int	major	设备号范围内的主设备号
unsigned int	baseminor	设备号范围内的初始次设备号

表 13-10: char_device_struct 描述符中的字段 (续)

类型	字段	说明
int	minorct	设备号范围的大小
const char *	name	处理设备号范围内的设备驱动程序的名称
struct file_operations *	fops	没有使用
struct cdev *	cdev	指向字符设备驱动程序描述符的指针

本质上可以采用两种方法为字符设备驱动程序分配一个范围内的设备号。所有新的设备驱动程序使用第一种方法，该方法使用 `register_chrdev_region()` 函数和 `alloc_chrdev_region()` 函数为驱动程序分配任意范围内的设备号。例如，为了获得从 `dev`（类型为 `dev_t`）开始的大小为 `size` 的一个设备号范围：

```
register_chrdev_region(dev, size, "foo");
```

上述函数并不执行 `cdev_add()`，因此设备驱动程序在所要求的设备号范围被成功分配时必须执行 `cdev_add()` 函数。

第二种方法使用 `register_chrdev()` 函数，它分配一个固定的设备号范围，该范围包含唯一一个主设备号以及 0~255 的次设备号。在这种情形下，设备驱动程序不必调用 `cdev_add()` 函数。

register_chrdev_region() 函数和 alloc_chrdev_region() 函数

`register_chrdev_region()` 函数接收三个参数：初始的设备号（主设备号和次设备号）、请求的设备号范围大小（与次设备号的大小一样）以及这个范围内的设备号对应的设备驱动程序的名称。该函数检查请求的设备号范围是否跨越一些次设备号，如果是，则确定其主设备号以及覆盖整个区间的相应设备号范围，然后，在每个相应设备号范围上调用 `__register_chrdev_region()` 函数（参见下文）。

`alloc_chrdev_region()` 函数与 `register_chrdev_region()` 相似，但它可以动态地分配一个主设备号，因此，该函数接收的参数为设备号范围内的初始次设备号、范围的大小以及设备驱动程序的名称。结束时它也调用 `__register_chrdev_region()` 函数。

`__register_chrdev_region()` 函数执行以下步骤：

1. 分配一个新的 `char_device_struct` 结构，并用 0 填充。
2. 如果设备号范围内的主设备号为 0，那么设备驱动程序请求动态分配一个主设备号。

函数从散列表的末尾表项开始继续向后寻找一个与尚未使用的主设备号对应的空冲突链表（NULL 指针）。若没有找到空表项，则返回一个错误码（注 6）。

3. 初始化 `char_device_struct` 结构中的初始设备号、范围大小以及设备驱动程序名称。
4. 执行散列函数计算与主设备号对应的散列表索引。
5. 遍历冲突链表，为新的 `char_device_struct` 结构寻找正确的位置。同时，如果找到与请求的设备号范围重叠的一个范围，则返回一个错误码。
6. 将新的 `char_device_struct` 描述符插入冲突链表中。
7. 返回新的 `char_device_struct` 描述符的地址。

register_chrdev()函数

驱动程序使用 `register_chrdev()` 函数时需要一个老式的设备号范围：一个单独的主设备号和 0~255 的次设备号范围。该函数接收的参数为：请求的主设备号 `major`（如果是 0 则动态分配）、设备驱动程序的名称 `name` 和一个指针 `fops`（它指向设备号范围内的特定字符设备文件的文件操作表）。该函数执行下列操作：

1. 调用 `__register_chrdev_region()` 函数分配请求的设备号范围。如果返回一个错误码（不能分配该范围），函数将终止运行。
2. 为设备驱动程序分配一个新的 `cdev` 结构。
3. 初始化 `cdev` 结构：
 - a. 将内嵌的 `kobject` 类型设置为 `ktype_cdev_dynamic` 类型的描述符（参见前面的“`kobject`”一节）。
 - b. 将 `owner` 字段设置为 `fops->owner` 的内容。
 - c. 将 `ops` 字段设置为文件操作表的地址 `fops`。
 - d. 将设备驱动程序的名称拷贝到内嵌的 `kobject` 结构里的 `name` 字段中。
4. 调用 `cdev_add()` 函数（在前面解释过）。
5. 将 `__register_chrdev_region()` 函数在第 1 步中返回的 `char_device_struct` 描述符的 `cdev` 字段设置为设备驱动程序的 `cdev` 描述符的地址。
6. 返回分配的设备号范围的主设备号。

注 6： 注意，内核可以动态分配一个小于 255 的主设备号，而且在某些情况下，即使存在一个小于 255 的未被使用的主设备号，分配也可能失败。我们可以期待将来会改变这种限制。

访问字符设备驱动程序

我们在“设备文件的 VFS 处理”一节中曾提到，由 `open()` 系统调用服务例程触发的 `dentry_open()` 函数定制字符设备文件的文件对象的 `f_op` 字段，以使它指向 `def_chr_fops` 表。这个表几乎为空；它仅仅定义了 `chrdev_open()` 函数作为设备文件的打开方法。这个方法由 `dentry_open()` 直接调用。

`chrdev_open()` 函数接收的参数为索引节点的地址 `inode`、指向所打开文件对象的指针 `filp`。本质上它执行以下操作：

1. 检查指向设备驱动程序的 `cdev` 描述符的指针 `inode->i_cdev`。如果该字段不为空，则 `inode` 结构已经被访问；增加 `cdev` 描述符的引用计数器值并跳转到第 6 步。
2. 调用 `kobj_lookup()` 函数搜索包括该设备号在内的范围。如果该范围不存在，则返回一个错误码；否则，函数计算与该范围相对应的 `cdev` 描述符的地址。
3. 将 `inode` 对象的 `inode->i_cdev` 字段设置为 `cdev` 描述符的地址。
4. 将 `inode->i_cindex` 字段设置为设备驱动程序的设备号范围内的设备号的相关索引（设备号范围内的第一个次设备号的索引值为 0，第二个为 1，依此类推）。
5. 将 `inode` 对象加入到由 `cdev` 描述符的 `list` 字段所指向的链表中。
6. 将 `filp->f_ops` 文件操作指针初始化为 `cdev` 描述符的 `ops` 字段的值。
7. 如果定义了 `filp->f_ops->open` 方法，`chrdev_open()` 函数就会执行该方法。若设备驱动程序处理一个以上的设备号，则 `chrdev_open()` 一般会再次设置 `file` 对象的文件操作，这样可以为所访问的设备文件安装合适的文件操作。
8. 成功时返回 0 结束。

字符设备的缓冲策略

传统的类 Unix 操作系统把硬件设备划分为块设备和字符设备。但是，这种分类并不能说明整个事实。某些设备在一次单独的 I/O 操作中能够传送大量的数据，而有些设备则只能传送几个字符。

例如，PS/2 鼠标驱动程序在每次读操作中获得几个字节——它们对应鼠标按钮的状态和屏幕上鼠标的指针。这种设备是最容易处理的。首先从设备的输入寄存器中一次读一个字符的输入数据，并存放在合适的内核数据结构中；然后，在空闲时把这个数据拷贝到进程的地址空间。同理，把输出数据首先从进程的地址空间拷贝到合适的内核数据结构中，然后，再一次一个字符地写到 I/O 设备的输出寄存器。显然，这种设备的 I/O 驱

动程序没有使用 DMA，因为 CPU 建立 DMA I/O 操作所花费的时间跟把数据移到 I/O 端口所花费的时间差不多。

另一方面，内核也必须准备处理在每次 I/O 操作中产生大量字节的设备，这些设备或者是诸如声卡或网卡的顺序设备，或者是诸如各类磁盘（软盘、光盘、SCSI 磁盘等）的随机访问设备。

例如，假定你已经为自己的计算机配置了声卡，以便能够录下来自麦克风的聲音。声卡以固定的频率（比如说 44.14 kHz）对来自麦克风的电信号进行采样，并产生一个 16 位数的输入数据块的流。声卡驱动程序必须能处理所有可能情况下这种蜂拥而至的数据，即使当 CPU 暂时忙于运行某个其他进程也不例外。

这可以结合两种不同的技术做到：

- 使用 DMA 方式传送数据块。
- 运用两个或多个元素的循环缓冲区，每个元素具有一个数据块的大小。当一个中断（发送一个信号表明新的数据块已被读入）发生时，中断处理程序把指针移到循环缓冲区的下一个元素，以便将来的数据会存放在一个空元素中。相反，只要驱动程序把数据成功地拷贝到用户地址空间，就释放循环缓冲区中的元素，以便用它来保存从硬件设备传送来的新数据。

循环缓冲区的作用是消除 CPU 负载的峰值，即使接收数据的用户态应用程序因为其他高优先级任务而慢下来，DMA 也要能够继续填充循环缓冲区中的元素，因为中断处理程序代表当前运行的进程执行。

当接收来自网卡的数据包时有类似的情况发生，只是在这种情况下，进入的数据流都是异步的。数据包被互相独立地接收，且两个连续的数据包之间到达的时间间隔是不可预测的。

总而言之，顺序设备的缓冲区是容易处理的，因为同一缓冲区从不会被重用：音频应用程序不能要求麦克风重新传送同一数据块。

我们将在第十五章中看到对随机访问设备（各种各样的磁盘）进行缓冲是相当复杂的。



块设备驱动程序

本章主要讨论块设备（例如各类磁盘）的 I/O 驱动程序。块设备的主要特点是，CPU 和总线读写数据所花时间与磁盘硬件的速度不匹配。块设备的平均访问时间很高。每个操作都需要几个毫秒才能完成，主要是因为磁盘控制器必须在磁盘表面将磁头移动到记录数据的确切位置。但是，当磁头到达正确位置时，数据传送就可以稳定在每秒几十 MB 的速率。

Linux 块设备处理程序的组织是相当复杂的。我们不可能对内核的块设备 I/O 子系统所包含的所有函数都进行详细讨论；但是，我们会提纲挈领地介绍一般软件体系结构。与前一章一样，我们的目标是描述 Linux 如何支持各种块设备驱动程序的实现，而不只说明如何实现一个具体的驱动程序。

我们将在“块设备的处理”一节首先说明 Linux 块设备 I/O 子系统的一般体系结构。然后将在“通用块层”、“I/O 调度程序”和“块设备驱动程序”这几节中描述块设备 I/O 子系统的主要组件。在最后的“打开块设备文件”一节中，我们简要地介绍一下打开一个块设备文件时内核所执行的步骤。

块设备的处理

块设备驱动程序上的每个操作都涉及很多内核组件；其中最重要的一些如图 14-1 所示。

例如，我们假设一个进程在某个磁盘文件上发出一个 `read()` 系统调用——我们将会看到处理 `write` 请求本质上采用同样的方式。下面是内核对进程请求给予回应的一般步骤：

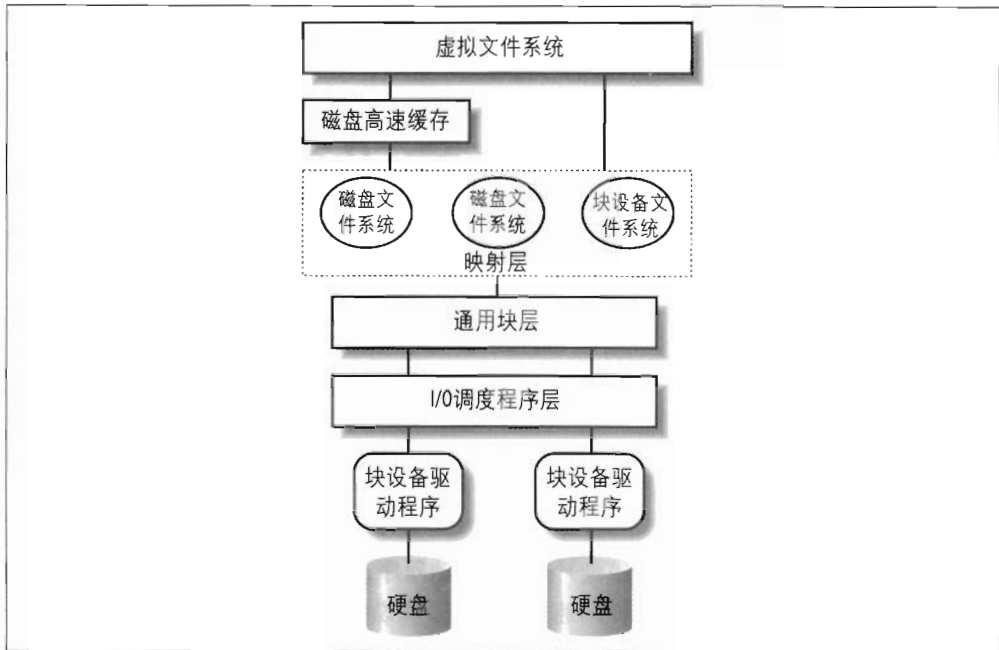


图 14-1: 一个块设备操作所涉及的内核组件

1. `read()` 系统调用的服务例程调用一个适当的 VFS 函数，将文件描述符和文件内的偏移量传递给它。虚拟文件系统位于块设备处理体系结构的上层，它提供一个通用的文件模型，Linux 支持的所有文件系统均采用该模型。我们在第十二章已经详细介绍了 VFS 层。
2. VFS 函数确定所请求的数据是否已经存在，如果有必要的话，它决定如何执行 read 操作。有时候没有必要访问磁盘上的数据，因为内核将大多数最近从块设备读出或写入其中的数据保存在 RAM 中。第十五章介绍了磁盘高速缓存机制，而第十六章详细说明了 VFS 如何处理磁盘操作以及如何与磁盘高速缓存和文件系统交互。
3. 我们假设内核从块设备读数据，那么它就必须确定数据的物理位置。为了做到这点，内核依赖映射层 (*mapping layer*)，主要执行下面两步：
 - a. 内核确定该文件所在文件系统的块大小，并根据文件块的大小计算所请求数据的长度。本质上，文件被看作拆分成许多块，因此内核确定请求数据所在的块号（文件开始位置的相对索引）。
 - b. 接下来，映射层调用一个具体文件系统的函数，它访问文件的磁盘节点，然后根据逻辑块号确定所请求数据在磁盘上的位置。事实上，磁盘也被看作拆分成许多块，因此内核必须确定存放所请求数据的块对应的号（磁盘或分区开始位

置的相对索引)。由于一个文件可能存储在磁盘上的不连续块中,因此存放在磁盘索引节点中的数据结构将每个文件块号映射为一个逻辑块号(注1)。

我们将在第十六章中说明映射层的功能,在第十八章中将介绍一些典型的磁盘文件系统。

4. 现在内核可以对块设备发出读请求。内核利用通用块层 (*generic block layer*) 启动 I/O 操作来传送所请求的数据。一般而言,每个 I/O 操作只针对磁盘上一组连续的块。由于请求的数据不必位于相邻的块中,所以通用块层可能启动几次 I/O 操作。每次 I/O 操作是由一个“块 I/O”(简称“bio”)结构描述,它收集底层组件需要的所有信息以满足所发出的请求。

通用块层为所有的块设备提供了一个抽象视图,因而隐藏了硬件块设备间的差异性。几乎所有的块设备都是磁盘,所以通用块层也提供了一些通用数据结构来描述“磁盘”或“磁盘分区”。我们将在本章的“通用块层”一节中讨论通用块层和 bio 数据结构。

5. 通用块层下面的“I/O 调度程序”根据预先定义的内核策略将待处理的 I/O 数据传送请求进行归类。调度程序的作用是把物理介质上相邻的数据请求聚集在一起。我们将在本章后面的“I/O 调度程序”一节中介绍调度程序。
6. 最后,块设备驱动程序向磁盘控制器的硬件接口发送适当的命令,从而进行实际的数据传送。我们将在后面的“块设备驱动程序”一节介绍通用块设备驱动程序的总体组织结构。

如你所见,块设备中的数据存储涉及了许多内核组件,每个组件采用不同长度的块来管理磁盘数据:

- 硬件块设备控制器采用称为“扇区”的固定长度的块来传送数据。因此,I/O 调度程序和块设备驱动程序必须管理数据扇区。
- 虚拟文件系统、映射层和文件系统将磁盘数据存放在称为“块”的逻辑单元中。一个块对应文件系统中一个最小的磁盘存储单元。
- 我们很快会看到,块设备驱动程序应该能够处理数据的“段”:一个段就是一个内存页或内存页的一部分,它们包含磁盘上物理相邻的数据块。
- 磁盘高速缓存作用于磁盘数据的“页”上,每页正好装在一个页框中。
- 通用块层将所有的上层和下层的组件组合在一起,因此它了解数据的扇区、块、段以及页。

注1: 但是,如果是从原始块设备文件进行读访问,映射层就不调用具体文件系统的方法,而是把块设备文件中的偏移量转换成在磁盘或在对应设备文件的磁盘分区中的位置。

即使有许多不同的数据块，它们通常也是共享相同的物理 RAM 单元。例如，图 14-2 显示了一个具有 4096 字节的页的构造。上层内核组件将页看成是由 4 个 1024 字节组成的块缓冲区。块设备驱动程序正在传送页中的后 3 个块，因此这 3 块被插入到涵盖了后 3072 字节的段中。硬盘控制器将该段看成是由 6 个 512 字节的扇区组成。

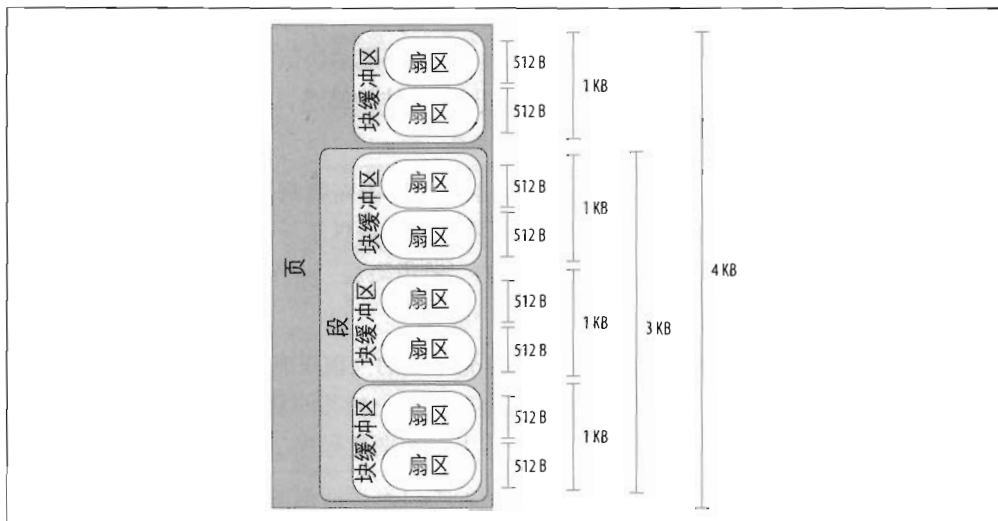


图 14-2：包含磁盘数据的页的典型构造

本章我们介绍处理块设备的下层内核组件：通用块层、I/O 调度程序以及块设备驱动程序，因此我们将注意力集中在扇区、块和段上。

扇区

为了达到可接受的性能，硬盘和类似的设备快速传送几个相邻字节的数据。块设备的每次数据传送操作都作用于一组称为扇区的相邻字节。在下面的讨论中，我们假定字节按相邻的方式记录在磁盘表面，这样一次搜索操作就可以访问到它们。尽管磁盘的物理构造很复杂，但是硬盘控制器接收到的命令将磁盘看成一大组扇区。

在大部分磁盘设备中，扇区的大小是 512 字节，但是一些设备使用更大的扇区（1024 和 2048 字节）。注意，应该把扇区作为数据传送的基本单元，不允许传送少于一个扇区的数据，尽管大部分磁盘设备都可以同时传送几个相邻的扇区。

在 Linux 中，扇区大小按惯例都设为 512 字节；如果一个块设备使用更大的扇区，那么相应的底层块设备驱动程序将做些必要的变换。因此，对存放在块设备中的一组数据是通过它们在磁盘上的位置来标识，即其首个 512 字节扇区的下标以及扇区的数目。扇区的下标存放在类型为 `sector_t` 的 32 位或 64 位的变量中。

块

扇区是硬件设备传送数据的基本单位，而块是VFS和文件系统传送数据的基本单位。例如，内核访问一个文件的内容时，它必须首先从磁盘上读文件的磁盘索引节点所在的块（参见第十二章的“索引节点对象”一节）。该块对应磁盘上一个或多个相邻的扇区，而VFS将其看成是一个单一的数据单元。

在Linux中，块大小必须是2的幂，而且不能超过一个页框。此外，它必须是扇区大小的整数倍，因为每个块必须包含整数个扇区。因此，在 80×86 体系结构中，允许块的大小为512、1024、2048和4096字节。

块设备的块大小不是唯一的。创建一个磁盘文件系统时，管理员可以选择合适的块大小。因此，同一个磁盘上的几个分区可能使用不同的块大小。此外，对块设备文件的每次读或写操作是一种“原始”访问，因为它绕过了磁盘文件系统；内核通过使用最大的块（4096字节）执行该操作。

每个块都需要自己的块缓冲区，它是内核用来存放块内容的RAM内存区。当内核从磁盘读出一个块时，就用从硬件设备中所获得的值来填充相应的块缓冲区；同样，当内核向磁盘中写入一个块时，就用相关块缓冲区的实际值来更新硬件设备上相应的一组相邻字节。块缓冲区的大小通常要与相应块的大小相匹配。

缓冲区首部是一个与每个缓冲区相关的buffer_head类型的描述符。它包含内核处理缓冲区需要了解的所有信息；因此，在对每个缓冲区进行操作之前，内核都要首先检查其缓冲区首部。我们将在第十五章中详细介绍缓冲区首部中的所有字段值；但是在本章中我们仅仅介绍其中的一些字段：`b_page`、`b_data`、`b_blocknr`和`b_bdev`。

`b_page`字段存放的是块缓冲区所在页框的页描述符地址。如果页框位于高端内存中，那么`b_data`字段存放页中块缓冲区的偏移量；否则，`b_data`存放块缓冲区本身的起始线性地址。`b_blocknr`字段存放的是逻辑块号（例如磁盘分区中的块索引）。最后，`b_bdev`字段标识使用缓冲区首部的块设备（参见本章后面的“块设备”一节）。

段

我们知道对磁盘的每个I/O操作就是在磁盘与一些RAM单元之间相互传送一些相邻扇区的内容。大多数情况下，磁盘控制器直接采用DMA方式进行数据传送[参见第十三章中的“直接内存访问(DMA)”一节]。块设备驱动程序只要向磁盘控制器发送一些适当的命令就可以触发一次数据传送；一旦完成数据的传送，控制器就会发出一个中断通知块设备驱动程序。

DMA 方式传送的是磁盘上相邻扇区的数据。这是一个物理约束：磁盘控制器允许 DMA 传送不相邻的扇区数据，但是这种方式的传送速率很低，因为在磁盘表面上移动读/写磁头是相当慢的。

老式的磁盘控制器仅仅支持“简单的”DMA 传送方式：在这种传送方式中，磁盘必须与 RAM 中的连续内存单元相互传送数据。但是，新的磁盘控制器也支持所谓的分散-聚集 (scatter-gather) DMA 传送方式：此种方式中，磁盘可以与一些非连续的内存区相互传送数据。

启动一次分散-聚集 DMA 传送，块设备驱动程序需要向磁盘控制器发送：

- 要传送的起始磁盘扇区号和总的扇区数
- 内存区的描述符链表，其中链表的每项包含一个地址和一个长度

磁盘控制器负责整个数据传送；例如，在读操作中控制器从相邻磁盘扇区中获得数据，然后将它们存放到不同的内存区中。

为了使用分散-聚集 DMA 传送方式，块设备驱动程序必须能够处理称为段的数据存储单元。一个段就是一个内存页或内存页中的一部分，它们包含一些相邻磁盘扇区中的数据。因此，一次分散-聚集 DMA 操作可能同时传送几个段。

注意，块设备驱动程序不需要知道块、块大小以及块缓冲区。因此，即使高层将段看成是由几个块缓冲区组成的页，块设备驱动程序也不用对此给予关注。

正如我们所见，如果不同的段在 RAM 中相应的页框正好是连续的并且在磁盘上相应的数据块也是相邻的，那么通用块层可以合并它们。通过这种合并方式产生的更大的内存区就称为物理段。

然而，在多种体系结构上还允许使用另一个合并方式：通过使用一个专门的总线电路 [如 IO-MMU，参见第十三章中的“直接内存访问 (DMA)”一节] 来处理总线地址与物理地址间的映射。通过这种合并方式产生的内存区称为硬件段。由于我们将注意力集中在 80×86 体系结构上，它在总线地址和物理地址之间不存在动态的映射，因此在本章剩余部分我们假定硬件段总是对应物理段。

通用块层

通用块层是一个内核组件，它处理来自系统中的所有块设备发出的请求。由于该层所提供的函数，内核可以容易地做到：

- 将数据缓冲区放在高端内存——仅当 CPU 访问其数据时，才将页框映射为内核中的线性地址空间，并在数据访问完后取消映射。
- 通过一些附加的手段，实现一个所谓的“零-复制”模式，将磁盘数据直接存放在用户态地址空间而不是首先复制到内核内存区；事实上，内核为 I/O 数据传送使用的缓冲区所在的页框就映射在进程的用户态线性地址空间中。
- 管理逻辑卷，例如由 LVM（逻辑卷管理器）和 RAID（廉价磁盘冗余阵列）使用的逻辑卷：几个磁盘分区，即使位于不同的块设备中，也可以被看作是一个单一的分区。
- 发挥大部分新磁盘控制器的高级特性，例如大主板磁盘高速缓存、增强的 DMA 性能、I/O 传送请求的相关调度等等。

bio 结构

通用块层的核心数据结构是一个称为 *bio* 的描述符，它描述了块设备的 I/O 操作。每个 *bio* 结构都包含一个磁盘存储区标识符（存储区中的起始扇区号和扇区数目）和一个或多个描述与 I/O 操作相关的内存区的段。*bio* 由 *bio* 数据结构描述，其各字段如表 14-1 所示。

表 14-1: *bio* 结构中的字段

类型	字段	说明
<code>sector_t</code>	<code>bi_sector</code>	块 I/O 操作的第一个磁盘扇区
<code>struct bio *</code>	<code>bi_next</code>	链接到请求队列中的下一个 <i>bio</i>
<code>struct block_device *</code>	<code>bi_bdev</code>	指向块设备描述符的指针
<code>unsigned long</code>	<code>bi_flags</code>	<i>bio</i> 的状态标志
<code>unsigned long</code>	<code>bi_rw</code>	I/O 操作标志
<code>unsigned short</code>	<code>bi_vcnt</code>	<i>bio</i> 的 <code>bio_vec</code> 数组中段的数目
<code>unsigned short</code>	<code>bi_idx</code>	<i>bio</i> 的 <code>bio_vec</code> 数组中段的当前索引值
<code>unsigned short</code>	<code>bi_phys_segments</code>	合并之后 <i>bio</i> 中物理段的数目
<code>unsigned short</code>	<code>bi_hw_segments</code>	合并之后硬件段的数目
<code>unsigned int</code>	<code>bi_size</code>	需要传送的字节数
<code>unsigned int</code>	<code>bi_hw_front_size</code>	硬件段合并算法使用
<code>unsigned int</code>	<code>bi_hw_back_size</code>	硬件段合并算法使用
<code>unsigned int</code>	<code>bi_max_vecs</code>	<i>bio</i> 的 <code>bio_vec</code> 数组中允许的最大段数
<code>struct bio_vec *</code>	<code>bi_io_vec</code>	指向 <i>bio</i> 的 <code>bio_vec</code> 数组中的段的指针

表 14-1: bio 结构中的字段 (续)

类型	字段	说明
bio_end_io_t *	bi_end_io	bio 的 I/O 操作结束时调用的方法
atomic_t	bi_cnt	bio 的引用计数器
void *	bi_private	通用块层和块设备驱动程序的 I/O 完成方法使用的指针
bio_destructor_t *	bi_destructor	释放 bio 时调用的析构方法 (通常是 bio_destructor() 方法)

bio 中的每个段是由一个 bio_vec 数据结构描述的, 其中各字段如表 14-2 所示。bio 中的 bi_io_vec 字段指向 bio_vec 数据结构的第一个元素, bi_vcnt 字段则存放了 bio_vec 数组中当前的元素个数。

表 14-2: bio_vec 结构中的字段

类型	字段	说明
struct page *	bv_page	指向段的页框中页描述符的指针
unsigned int	bv_len	段的字节长度
unsigned int	bv_offset	页框中段数据的偏移量

在块 I/O 操作期间 bio 描述符的内容一直保持更新。例如, 如果块设备驱动程序在一次分散-聚集 DMA 操作中不能完成全部的数据传送, 那么 bio 中的 bi_idx 字段会不断更新来指向待传送的第一个段。为了从索引 bi_idx 指向的当前段开始不断重复 bio 中的段, 设备驱动程序可以执行宏 bio_for_each_segment。

当通用块层启动一次新的 I/O 操作时, 调用 bio_alloc() 函数分配一个新的 bio 结构。通常, bio 结构是由 slab 分配器分配的, 但是, 当内存不足时, 内核也会使用一个备用的 bio 小内存池 (参见第八章的“内存池”一节)。内核也为 bio_vec 结构分配内存池——毕竟, 分配一个 bio 结构而不能分配其中的段描述符也是没有什么意义的。相应地, bio_put() 函数减少 bio 中引用计数器 (bi_cnt) 的值, 如果该值等于 0, 则释放 bio 结构以及相关的 bio_vec 结构。

磁盘和磁盘分区的表示

磁盘是一个由通用块层处理的逻辑块设备。通常一个磁盘对应一个硬件块设备, 例如硬盘、软盘或光盘。但是, 磁盘也可以是一个虚拟设备, 它建立在几个物理磁盘分区之上或一些 RAM 专用页中的内存区上。在任何情形中, 借助通用块层提供的服务, 上层内核组件可以以同样的方式工作在所有的磁盘上。

磁盘是由 `gendisk` 对象描述的，其中各字段如表 14-3 所示。

表 14-3: `gendisk` 对象中的字段

类型	字段	说明
<code>int</code>	<code>major</code>	Major 磁盘主设备号
<code>int</code>	<code>first_minor</code>	与磁盘关联的第一个次设备号
<code>int</code>	<code>minors</code>	与磁盘关联的次设备号范围
<code>char [32]</code>	<code>disk_name</code>	磁盘的标准名称(通常是相应设备文件的规范名称)
<code>struct hd_struct **</code>	<code>part</code>	磁盘的分区描述符数组
<code>struct block_device_operations *</code>	<code>fops</code>	指向块设备操作表的指针
<code>struct request_queue *</code>	<code>queue</code>	指向磁盘请求队列的指针(参见本章后面的“请求队列描述符”一节)
<code>void *</code>	<code>private_data</code>	块设备驱动程序的私有数据
<code>sector_t</code>	<code>capacity</code>	磁盘内存区的大小(扇区数目)
<code>int</code>	<code>flags</code>	描述磁盘类型的标志(见下文)
<code>char [64]</code>	<code>devfs_name</code>	<code>devfs</code> 特殊文件系统(现在已不赞成使用)中的设备文件名称
<code>int</code>	<code>number</code>	不再使用
<code>struct device *</code>	<code>driverfs_dev</code>	指向磁盘的硬件设备的 <code>device</code> 对象的指针(参见第十三章的“设备驱动程序模型的组件”一节)
<code>struct kobject</code>	<code>kobj</code>	内嵌的 <code>kobject</code> 结构(参见第十三章的“ <code>kobject</code> ”一节)
<code>struct timer_rand_state *</code>	<code>random</code>	该指针指向的这个数据结构记录磁盘中断的定时,由内核内置的随机数发生器使用
<code>int</code>	<code>policy</code>	如果磁盘是只读的,则置为 1(写操作禁止),否则为 0
<code>atomic_t</code>	<code>sync_io</code>	写入磁盘的扇区数计数器,仅为 RAID 使用
<code>unsigned long</code>	<code>stamp</code>	统计磁盘队列使用情况的时间戳
<code>unsigned long</code>	<code>stamp_id1</code>	同上
<code>int</code>	<code>in_flight</code>	正在进行的 I/O 操作数
<code>struct disk_stats *</code>	<code>dkstats</code>	统计每个 CPU 使用磁盘的情况

flags字段存放了关于磁盘的信息。其中最重要的标志是GENHD_FL_UP: 如果设置它, 那么磁盘将被初始化并可以使用。另一个相关的标志是GENHD_FL_REMOVABLE, 如果是诸如软盘或光盘这样可移动的磁盘, 那么就要设置该标志。

gendisk对象的fops字段指向一个表block_device_operations, 该表为块设备的主要操作存放了几个定制的方法(如表14-4所示)。

表 14-4: 块设备的方法

方法	触发事件
open	打开块设备文件
release	关闭对块设备文件的最后一个引用
ioctl	在块设备文件上发出 ioctl() 系统调用 (使用大内核锁)
compat_ioctl	在块设备文件上发出 ioctl() 系统调用 (不使用大内核锁)
media_changed	检查可移动介质是否已经变化 (例如软盘)
revalidate_disk	检查块设备是否持有有效数据

通常硬盘被划分成几个逻辑分区。每个块设备文件要么代表整个磁盘, 要么代表磁盘中的某一个分区。例如, 一个主设备号为3、次设备号为0的设备文件 /dev/hda 代表的可能是一个主 EIDE 磁盘; 该磁盘中的前两个分区分别由设备文件 /dev/hda1 和 /dev/hda2 代表, 它们的主设备号都是3, 而次设备号分别为1和2。一般而言, 磁盘中的分区是由连续的次设备号来区分的。

如果将一个磁盘分成了几个分区, 那么其分区表保存在hd_struct结构的数组中, 该数组的地址存放在gendisk对象的part字段中。通过磁盘内分区的相对索引对该数组进行索引。hd_struct描述符中的字段如表14-5所示。

表 14-5: hd_struct描述符中的字段

类型	字段	说明
sector_t	start_sect	磁盘中分区的起始扇区
sector_t	nr_sects	分区的长度 (扇区数)
struct kobject	kobj	内嵌的kobject (参见第十三章的“kobject”一节)
unsigned int	reads	对分区发出的读操作次数
unsigned int	read_sectors	从分区读取的扇区数
unsigned int	writes	对分区发出的写操作次数
unsigned int	write_sectors	写进分区的扇区数

表 14-5: hd_struct 描述符中的字段 (续)

类型	字段	说明
int	policy	如果分区是只读的, 则置为 1, 否则为 0
int	partno	磁盘中分区的相对索引

当内核发现系统中一个新的磁盘时(在启动阶段, 或将一个可移动介质插入一个驱动器中时, 或在运行期附加一个外置式磁盘时), 就调用 `alloc_disk()` 函数, 该函数分配并初始化一个新的 `gendisk` 对象, 如果新磁盘被分成了几个分区, 那么 `alloc_disk()` 还会分配并初始化一个适当的 `hd_struct` 类型的数组。然后, 内核调用 `add_disk()` 函数将新的 `gendisk` 对象插入到通用块层的数据结构中(参见本章后面的“注册和初始化设备驱动程序”一节)。

提交请求

我们介绍一下当向通用块层提交一个 I/O 操作请求时, 内核所执行的步骤顺序。我们假设被请求的数据块在磁盘上是相邻的, 并且内核已经知道了它们的物理位置。

第一步是执行 `bio_alloc()` 函数分配一个新的 `bio` 描述符。然后, 内核通过设置一些字段值来初始化 `bio` 描述符:

- 将 `bi_sector` 设为数据的起始扇区号(如果块设备分成了几个分区, 那么扇区号是相对于分区的起始位置的)。
- 将 `bi_size` 设为涵盖整个数据的扇区数目。
- 将 `bi_bdev` 设为块设备描述符的地址(参见本章后面的“块设备”一节)。
- 将 `bi_io_vec` 设为 `bio_vec` 结构数组的起始地址, 数组中的每个元素描述了 I/O 操作中的一个段(内存缓存); 此外, 将 `bi_vcnt` 设为 `bio` 中总的段数。
- 将 `bi_rw` 设为被请求操作的标志。其中最重要的标志指明数据传送的方向: `READ (0)` 或 `WRITE (1)`。
- 将 `bi_end_io` 设为当 `bio` 上的 I/O 操作完成时所执行的完成程序的地址。

一旦 `bio` 描述符被进行了适当的初始化, 内核就调用 `generic_make_request()` 函数, 它是通用块层的主要入口点。该函数主要执行下列操作:

1. 检查 `bio->bi_sector` 没有超过块设备的扇区数。如果超过, 则将 `bio->bi_flags` 设置为 `BIO_EOF` 标志, 然后打印一条内核错误信息, 调用 `bio_endio()` 函数, 并终止。 `bio_endio()` 更新 `bio` 描述符中的 `bi_size` 和 `bi_sector` 值, 然后调用 `bio`

的 `bi_end_io` 方法。`bi_end_io` 函数的实现本质上依赖于触发 I/O 数据传送的内核组件；我们将在下面的章节中看到 `bi_end_io` 方法的一些例子。

2. 获取与块设备相关的请求队列 `q`（参见本章后面的“请求队列描述符”一节）；其地址存放在块设备描述符的 `bd_disk` 字段中，其中的每个元素由 `bio->bi_bdev` 指向。
3. 调用 `block_wait_queue_running()` 函数检查当前正在使用的 I/O 调度程序是否可以被动态取代；若可以，则让当前进程睡眠直到启动一个新的 I/O 调度程序（参见下一节“I/O 调度程序”）。
4. 调用 `blk_partition_remap()` 函数检查块设备是否指的是一个磁盘分区（`bio->bi_bdev` 不等于 `bio->bi_dev->bd_contains`；参见本章后面的“块设备”一节）。如果是，则从 `bio->bi_bdev` 获取分区的 `hd_struct` 描述符，从而执行下面的子操作：
 - a. 根据数据传送的方向，更新 `hd_struct` 描述符中的 `read_sectors` 和 `reads` 值，或 `write_sectors` 和 `writes` 值。
 - b. 调整 `bio->bi_sector` 值使得把相对于分区的起始扇区号转变为相对于整个磁盘的扇区号。
 - c. 将 `bio->bi_bdev` 设置为整个磁盘的块设备描述符（`bio->bd_contains`）。

从现在开始，通用块层、I/O 调度程序以及设备驱动程序将忘记磁盘分区的存在，直接作用于整个磁盘。

5. 调用 `q->make_request_fn` 方法将 `bio` 请求插入请求队列 `q` 中。
6. 返回。

在本章后面的“向 I/O 调度程序发出请求”一节中我们将讨论 `make_request_fn` 方法典型实现。

I/O 调度程序

虽然块设备驱动程序一次可以传送一个单独的扇区，但是块 I/O 层并不会为磁盘上每个被访问的扇区都单独执行一次 I/O 操作；这会导致磁盘性能的下降，因为确定磁盘表面上扇区的物理位置是相当费时的。取而代之的是，只要可能，内核就试图把几个扇区合并在一起，并作为一个整体来处理，这样就减少了磁头的平均移动时间。

当内核组件要读或写一些磁盘数据时，实际上创建一个块设备请求。从本质上说，请求描述的是所请求的扇区以及要对它执行的操作类型（读或写）。然而，并不是请求一发出，内核就满足它——I/O 操作仅仅被调度，执行会向后推迟。这种人为的延迟是提高

块设备性能的关键机制。当请求传送一个新的数据块时，内核检查能否通过稍微扩展前一个一直处于等待状态的请求而满足新请求（也就是说，能否不用进一步的寻道操作就能满足新请求）。由于磁盘的访问大都是顺序的，因此这种简单机制就非常高效。

延迟请求复杂化了块设备的处理。例如，假设某个进程打开了一个普通文件，然后，文件系统的驱动程序就要从磁盘读取相应的索引节点。块设备驱动程序把这个请求加入一个队列，并把这个进程挂起，直到存放索引节点的块被传送为止。然而，块设备驱动程序本身不会被阻塞，因为试图访问同一磁盘的任何其他进程也可能被阻塞。

为了防止块设备驱动程序被挂起，每个 I/O 操作都是异步处理的。特别是块设备驱动程序是中断驱动的（参见第十三章的“监控 I/O 操作”一节）：通用块层调用 I/O 调度程序产生一个新的块设备请求或扩展一个已有的块设备请求，然后终止。随后激活的块设备驱动程序会调用一个所谓的策略例程（*strategy routine*）选择一个待处理的请求，并向磁盘控制器发出一条适当的命令来满足这个请求。当 I/O 操作终止时，磁盘控制器就产生一个中断，如果需要，相应的中断处理程序就又调用策略例程去处理队列中的另一个请求。

每个块设备驱动程序都维持着自己的请求队列，它包含设备待处理的请求链表。如果磁盘控制器正在处理几个磁盘，那么通常每个物理块设备都有一个请求队列。在每个请求队列上单独执行 I/O 调度，这样可以提高磁盘的性能。

请求队列描述符

请求队列是由一个大的数据结构 `request_queue` 表示的，其字段如表 14-6 所示。

表 14-6：请求队列描述符中的字段

类型	字段	说明
<code>struct list_head</code>	<code>queue_head</code>	待处理请求的链表
<code>struct request *</code>	<code>last_merge</code>	指向队列中首先可能合并的请求描述符
<code>elevator_t *</code>	<code>elevator</code>	指向 <code>elevator</code> 对象的指针（参见后面的“ <i>I/O 调度算法</i> ”一节）
<code>struct request_list</code>	<code>rq</code>	为分配请求描述符所使用的数据结构
<code>request_fn_proc *</code>	<code>request_fn</code>	实现驱动程序的策略例程入口点的方法
<code>merge_request_fn *</code>	<code>back_merge_fn</code>	检查是否可能将 <code>bio</code> 合并到请求队列的最后一个请求中的方法
<code>merge_request_fn *</code>	<code>front_merge_fn</code>	检查是否可能将 <code>bio</code> 合并到队列的第一个请求中的方法

表 14-6: 请求队列描述符中的字段 (续)

类型	字段	说明
merge_requests_fn *	merge_requests_fn	试图合并请求队列中两个相邻请求的方法
make_request_fn *	make_request_fn	将一个新请求插入请求队列时调用的方法
prep_rq_fn *	prep_rq_fn	该方法把这个处理请求的命令发送给硬件设备
unplug_fn *	unplug_fn	去掉块设备的方法 (参见本章后面的“激活块设备驱动程序”一节)
merge_bvec_fn *	merge_bvec_fn	当增加一个新段时, 该方法返回可插入到某个已存在的 bio 结构中的字节数 (通常未定义)
activity_fn *	activity_fn	将某个请求加入请求队列时调用的方法 (通常未定义)
issue_flush_fn *	issue_flush_fn	刷新请求队列时调用的方法 (通过连续处理所有的请求清空队列)
struct timer_list	unplug_timer	插入设备时使用的动态定时器 (参见后面的“激活块设备驱动程序”一节)
int	unplug_thresh	如果请求队列中待处理请求数大于该值, 将立即去掉请求设备 (缺省值是 4)
unsigned long	unplug_delay	去掉设备之前的时间延迟 (缺省值是 3ms)
struct work_struct	unplug_work	去掉设备时使用的操作队列 (参见后面的“激活块设备驱动程序”一节)
struct backing_dev_info	backing_dev_info	参见本表后面的正文
void *	queuedata	指向块设备驱动程序的私有数据的指针
void *	activity_data	activity_fn 方法使用的私有数据
unsigned long	bounce_pfn	在大于该页框号时必须使用缓冲区回弹 (参见本章前面的“提交请求”一节)
int	bounce_gfp	回弹缓冲区的内存分配标志
unsigned long	queue_flags	描述请求队列状态的标志
spinlock_t *	ueue_lock	指向请求队列锁的指针
struct kobject	kobj	请求队列的内嵌 kobject 结构
unsigned long	nr_requests	请求队列中允许的最大请求数

表 14-6: 请求队列描述符中的字段 (续)

类型	字段	说明
unsigned int	nr_congestion_on	如果待处理请求数超出了该阈值, 则认为该队列是拥挤的
unsigned int	nr_congestion_off	如果待处理请求数在这个阈值的范围内, 则认为该队列是不拥挤的
unsigned int	nr_batching	即使队列已满, 仍可以由特殊进程“batcher”提交的待处理请求的最大值 (通常为 32)
unsigned short	max_sectors	单个请求所能处理的最大扇区数 (可调的)
unsigned short	max_hw_sectors	单个请求所能处理的最大扇区数 (硬约束)
unsigned short	max_phys_segments	单个请求所能处理的最大物理段数
unsigned short	max_hw_segments	单个请求所能处理的最大硬段数 (分散-聚集 DMA 操作中的最大不同内存区数)
unsigned short	hardsect_size	扇区中以字节为单位的大小
unsigned int	max_segment_size	物理段的最大长度 (以字节为单位)
unsigned long	seg_boundary_mask	段合并的内存边界屏蔽字
unsigned int	dma_alignment	DMA 缓冲区的起始地址和长度的对齐位图 (缺省值是 511)
struct	queue_tags	空闲/忙标记的位图 (用于带标记的请求) blk_queue_tag *
atomic_t	refcnt	请求队列的引用计数器
unsigned int	in_flight	请求队列中待处理请求数
unsigned int	sg_timeout	用户定义的命令超时 (仅由 SCSI 通用块设备使用)
unsigned int	sg_reserved_size	基本上没有使用
struct list_head	drain_list	临时延时的请求链表的首部, 直到 I/O 调度程序被动态取代

实质上, 请求队列是一个双向链表, 其元素就是请求描述符 (也就是 request 数据结构, 参见下一节)。请求队列描述符中的 queue_head 字段存放链表的头 (第一个伪元素), 而请求描述符中 queuelist 字段的指针把任一请求链接到链表的前一个和后一个元素之间。

队列链表中元素的排序方式对每个块设备驱动程序是特定的，然而，I/O 调度程序提供了几种预先确定好的元素排序方式，这将在后面的“I/O 调度算法”一节中讨论。

backing_dev_info 字段是一个 backing_dev_info 类型的小对象，它存放了关于基本硬件块设备的 I/O 数据流量的信息。例如，它保存了关于预读以及关于请求队列拥塞状态的信息。

请求描述符

每个块设备的待处理请求都是用一个请求描述符来表示的，请求描述符存放在如表 14-7 所示的 request 数据结构中。

表 14-7: 请求描述符的字段

类型	字段	说明
struct list_head	queuelist	请求队列链表的指针
unsigned long	flags	请求标志（参见下面）
sector_t	sector	要传送的下一个扇区号
unsigned long	nr_sectors	整个请求中要传送的扇区数
unsigned int	current_nr_sectors	当前 bio 的当前段中要传送的扇区数
sector_t	hard_sector	要传送的下一个扇区号
unsigned long	hard_nr_sectors	整个请求中要传送的扇区数（由通用块层更新）
unsigned int	hard_cur_sectors	当前 bio 的当前段中要传送的扇区数（由通用块层更新）
struct bio *	bio	请求中第一个没有完成传送操作的 bio
struct bio *	biotail	请求链表中末尾的 bio
void *	elevator_private	指向 I/O 调度程序私有数据的指针
int	rq_status	请求状态：实际上，或者是 RQ_ACTIVE, 或者是 RQ_INACTIVE
struct gendisk *	rq_disk	请求所引用的磁盘描述符
int	errors	用于记录当前传送中发生的 I/O 失败次数的计数器
unsigned long	start_time	请求的起始时间（用 jiffies 表示）
unsigned short	nr_phys_segments	请求的物理段数
unsigned short	nr_hw_segments	请求的硬段数

表 14-7: 请求描述符的字段 (续)

类型	字段	说明
int	tag	与请求相关的标记 (只适合支持多次数据传送的硬件设备)
char *	buffer	指向当前数据传送的内存缓冲区的指针 (如果缓冲区是高端内存区, 则为 NULL)
int	ref_count	请求的引用计数器
request_queue_t *	q	指向包含请求的请求队列描述符的指针
struct request_list *	rl	指向 request_list 结构的指针
struct completion *	waiting	等待数据传送终止的 Completion 结构 (参见第五章的“补充原语”一节)
void *	special	对硬件设备发出“特殊”命令的请求所使用的数据的指针
unsigned int	cmd_len	cmd 字段中命令的长度
unsigned char []	cmd	由请求队列的 prep_rq_fn 方法准备好的预先内置命令所在的缓冲区
unsigned int	data_len	通常, 由 data 字段指向的缓冲区中数据的长度
void *	data	设备驱动程序为了跟踪所传送的数据而使用的指针
unsigned int	sense_len	由 sense 字段指向的缓冲区的长度 (如果 sense 是 NULL, 则为 0)
void *	sense	指向输出 sense 命令的缓冲区的指针
unsigned int	timeout	请求的超时
struct request_pm_state *	pm	指向电源管理命令所使用的数据结构

每个请求包含一个或多个 bio 结构。最初, 通用块层创建一个仅包含一个 bio 结构的请求。然后, I/O 调度程序要么向初始的 bio 中增加一个新段, 要么将另一个 bio 结构链接到请求中, 从而“扩展”该请求。可能存在新数据与请求中已存在的数据物理相邻的情况。请求描述符的 bio 字段指向请求中的第一个 bio 结构, 而 biotail 字段则指向最后一个 bio 结构。rq_for_each_bio 宏执行一个循环, 从而遍历请求中的所有 bio 结构。

请求描述符中的几个字段值可能是动态变化的。例如, 一旦 bio 中引用的数据块全部传

送完毕，bio 字段立即更新从而指向请求链表中的下一个 bio。在此期间，新的 bio 可能被加入到请求链表的尾部，所以 biotail 的值也可能改变。

当磁盘数据块正在传送时，请求描述符的其它几个字段的值由 I/O 调度程序或设备驱动程序修改。例如，nr_sectors 存放整个请求还需传送的扇区数，current_nr_sectors 存放当前 bio 结构中还需传送的扇区数。

flags 中存放了很多标志，如表 14-8 中所示。到目前为止，最重要的一个标志是 REQ_RW，它确定数据传送的方向。

表 14-8：请求描述符的标志

标志	说明
REQ_RW	数据传送的方向：READ (0) 或 WRITE (1)
REQ_FAILFAST	万一出错请求申明不再重试 I/O 操作
REQ_SOFTBARRIER	请求相当于 I/O 调度程序的屏障
REQ_HARDBARRIER	请求相当于 I/O 调度程序和设备驱动程序的屏障 —— 应当在旧请求与新请求之间处理该请求
REQ_CMD	包含一个标准的读或写 I/O 数据传送的请求
REQ_NOMERGE	不允许扩展或与其它请求合并的请求
REQ_STARTED	正处理的请求
REQ_DONTPREP	不调用请求队列中的 prep_rq_fn 方法预先准备把命令发送给硬件设备
REQ_QUEUED	请求被标记 —— 也就是说，与该请求相关的硬件设备可以同时管理很多未完成数据的传送
REQ_PC	请求包含发送给硬件设备的直接命令
REQ_BLOCK_PC	与前一个标志功能相同，但发送的命令包含在 bio 结构中
REQ_SENSE	请求包含一个“sense”请求命令 (SCSI 和 ATAPI 设备使用)
REQ_FAILED	当请求中的 sense 或 direct 命令的操作与预期的不一致时设置该标志
REQ_QUIET	万一 I/O 操作出错请求申明不产生内核消息
REQ_SPECIAL	请求包含对硬件设备的特殊命令 (例如，重设驱动器)
REQ_DRIVE_CMD	请求包含对 IDE 磁盘的特殊命令
REQ_DRIVE_TASK	请求包含对 IDE 磁盘的特殊命令
REQ_DRIVE_TASKFILE	请求包含对 IDE 磁盘的特殊命令
REQ_PREEMPT	请求取代位于请求队列前面的请求 (仅对 IDE 磁盘而言)

表 14-8: 请求描述符的标志 (续)

标志	说明
REQ_PM_SUSPEND	请求包含一个挂起硬件设备的电源管理命令
REQ_PM_RESUME	请求包含一个唤醒硬件设备的电源管理命令
REQ_PM_SHUTDOWN	请求包含一个切断硬件设备的电源管理命令
REQ_BAR_PREFLUSH	请求包含一个要发送给磁盘控制器的“刷新队列”命令
REQ_BAR_POSTFLUSH	请求包含一个已发送给磁盘控制器的“刷新队列”命令

对请求描述符的分配进行管理

在重负载和磁盘操作频繁的情况下,固定数目的动态空闲内存将成为进程想要把新请求加入请求队列 `q` 的瓶颈。为了解决这种问题,每个 `request_queue` 描述符包含一个 `request_list` 数据结构,其中包括:

- 一个指针,指向请求描述符的内存池(参见第八章的“内存池”一节)。
- 两个计数器,分别用于记录分配给 `READ` 和 `WRITE` 请求的请求描述符数。
- 两个标志,分别用于标记为读或写请求的分配是否失败。
- 两个等待队列,分别存放了为获得空闲的读和写请求描述符而睡眠的进程。
- 一个等待队列,存放等待一个请求队列被刷新(清空)的进程。

`blk_get_request()` 函数试图从一个特定请求队列的内存池中获得一个空闲的请求描述符;如果内存区不足并且内存池已经用完,则要么挂起当前进程,要么返回 `NULL` (如果不能阻塞内核控制路径)。如果分配成功,则将请求队列的 `request_list` 数据结构的地址存放在请求描述符的 `rl` 字段中。`blk_put_request()` 函数则释放一个请求描述符;如果该描述符的引用计数器的值为 0,则将描述符归还回它原来所在的内存池。

避免请求队列拥塞

每个请求队列都有一个允许处理的最大请求数。请求队列描述符的 `nr_requests` 字段存放了每个数据传送方向所允许处理的最大请求数。缺省情况下,一个队列至多有 128 个待处理读请求和 128 个待处理写请求。如果待处理的读(写)请求数超过了 `nr_requests` 值,那么通过设置请求队列描述符的 `queue_flags` 字段的 `QUEUE_FLAG_READFULL` (`QUEUE_FLAG_WRITEFULL`) 标志将该队列标记为已满,试图把请求加入到某个传送方向的可阻塞进程被放置到 `request_list` 结构所对应的等待队列中睡眠。

一个填满的请求队列对系统性能有负面影响，因为它会强制许多进程去睡眠以等待 I/O 数据传送的完成。因此，如果给定传送方向上的待处理请求数超过了存放在请求描述符的 `nr_congestion_on` 字段中的值（缺省值为 113），那么内核认为该队列是拥塞的，并试图降低新请求的创建速率。当待处理请求数小于 `nr_congestion_off` 的值（缺省值为 111）时，拥塞的请求队列才变为不拥塞。`blk_congestion_wait()` 函数挂起当前进程，直到所有请求队列都变为不拥塞或超时已到。

激活块设备驱动程序

正如我们在前面已经看到的一样，延迟激活块设备驱动程序有利于把相邻块请求进行集中。这种延迟是通过所谓的设备插入和设备拔出技术（注 2）来实现的。在块设备驱动程序被插入时，该驱动程序并不被激活，即使在驱动程序队列中有待处理的请求。

`blk_plug_device()` 函数的功能是插入一个块设备——更准确地说，插入到某个块设备驱动程序处理的请求队列中。本质上，该函数接收一个请求队列描述符的地址 `q` 作为其参数。它设置 `q->queue_flags` 字段中的 `QUEUE_FLAG_PLUGGED` 位；然后，重新启动 `q->unplug_timer` 字段中的内嵌动态定时器。

`blk_remove_plug()` 则拔去一个请求队列 `q`：清除 `QUEUE_FLAG_PLUGGED` 标志并取消 `q->unplug_timer` 动态定时器的执行。当“视线中”所有可合并的请求都被加入请求队列时，内核就会显式地调用该函数。此外，如果请求队列中待处理的请求数超过了请求队列描述符的 `unplug_thresh` 字段中存放的值（缺省值为 4），那么 I/O 调度程序也会去掉该请求队列。

如果一个设备保持插入的时间间隔为 `q->unplug_delay`（通常为 3ms），那么说明由 `blk_plug_device()` 函数激活的动态定时器的时间已用完，因此就会执行 `blk_unplug_timeout()` 函数。因而，唤醒内核线程 `kblockd` 所操作的工作队列 `kblockd_workqueue`（参见第四章的“工作队列”一节）。`kblockd` 执行 `blk_unplug_work()` 函数，其地址存放在 `q->unplug_work` 结构中。接着，该函数会调用请求队列中的 `q->unplug_fn` 方法，通常该方法是由 `generic_unplug_device()` 函数实现的。`generic_unplug_device()` 函数的功能是拔出块设备：首先，检查请求队列是否仍然活跃；然后，调用 `blk_remove_plug()` 函数；最后，执行策略例程 `request_fn` 方法来开始处理请求队列中的下一个请求（参见本章后面的“注册和初始化设备驱动程序”一节）。

注 2： 如果“插入”和“拔出”这两个术语使你费解，你可以把它们分别理解为“激活”和“撤消”。

I/O 调度算法

当向请求队列增加一条新的请求时，通用块层会调用 I/O 调度程序来确定请求队列中新请求的确切位置。I/O 调度程序试图通过扇区将请求队列排序。如果顺序地从链表中提取要处理的请求，那么就会明显减少磁头寻道的次数，因为磁头是按照直线的方式从内磁道移向外磁道（反之亦然），而不是随意地从一个磁道跳跃到另一个磁道。这可以从电梯算法中得到启发，回想一下，电梯算法处理来自不同层的上下请求。电梯是往一个方向移动的，当朝一个方向上的最后一个预定层到达时，电梯就会改变方向而开始向相反的方向移动。因此，I/O 调度程序也被称为电梯算法（*elevator*）。

在重负载情况下，严格遵循扇区号顺序的 I/O 调度算法运行的并不是很好。在这种情形下，数据传送的完成时间主要取决于磁盘上数据的物理位置。因此，如果设备驱动程序处理的请求位于队列的首部（小扇区号），并且拥有小扇区号的新请求不断被加入队列中，那么队列末尾的请求就很容易会饿死。因而 I/O 调度算法会非常复杂。

当前，Linux 2.6 中提供了四种不同类型的 I/O 调度程序或电梯算法，分别为“预期（Anticipatory）”算法、“最后期限（Deadline）”算法、“CFQ（Complete Fairness Queueing，完全公平队列）”算法以及“noop（No Operation）”算法。对大多数块设备而言，内核使用的缺省电梯算法可在引导时通过内核参数 `elevator=<name>` 进行再设置，其中 `<name>` 值可取下列任何一个：`as`、`deadline`、`cfq` 和 `noop`。如果没有给定引导参数，那么内核缺省使用“预期” I/O 调度程序。总之，设备驱动程序可以用任何一个调度程序取代缺省的电梯算法，设备驱动程序也可以自己定制 I/O 调度算法，但是这种情况很少见。

此外，系统管理员可以在运行时为一个特定的块设备改变 I/O 调度程序。例如，为了改变第一个 IDE 通道的主磁盘所使用的 I/O 调度程序，管理员可把一个电梯算法的名称写入 `sysfs` 特殊文件系统的 `/sys/block/hda/queue/scheduler` 文件中（参见第十三章中的“`sysfs` 文件系统”一节）。

请求队列中使用的 I/O 调度算法是由一个 `elevator_t` 类型的 `elevator` 对象表示的，该对象的地址存放在请求队列描述符的 `elevator` 字段中。`elevator` 对象包含了几种方法，它们覆盖了 `elevator` 所有可能的操作：链接和断开 `elevator`，增加和合并队列中的请求，从队列中删除请求，获得队列中下一个待处理的请求等等。`elevator` 对象也存放了一个表的地址，表中包含了处理请求队列所需的所有信息。而且，每个请求描述符包含一个 `elevator_private` 字段，该字段指向一个由 I/O 调度程序用来处理请求的附加数据结构。

现在我们从易到难简要地介绍一下四种 I/O 调度算法。注意，设计一个 I/O 调度程序与设计一个 CPU 调度程序很相似（参见第七章）：启发算法和采用的常量值是测试和基准外延量的结果。

一般而言，所有的算法都使用一个调度队列 (*dispatch queue*)，队列中包含的所有请求按照设备驱动程序应当处理的顺序进行排序——也即设备驱动程序要处理的下一个请求通常是调度队列中的第一个元素。调度队列实际上是由请求队列描述符的 `queue_head` 字段所确定的请求队列。几乎所有的算法都使用另外的队列对请求进行分类和排序。它们允许设备驱动程序将 `bio` 结构增加到已存在请求中，如果需要，还要合并两个“相邻的”请求。

“Noop” 算法

这是最简单的 I/O 调度算法。它没有排序的队列：新的请求通常被插在调度队列的开头或末尾，下一个要处理的请求总是队列中的第一个请求。

“CFQ” 算法

“CFQ (完全公平队列)”算法的主要目标是在触发 I/O 请求的所有进程中确保磁盘 I/O 带宽的公平分配。为了达到这个目标，算法使用许多个排序队列——缺省为 64——它们存放了不同进程发出的请求。当算法处理一个请求时，内核调用一个散列函数将当前进程的线程组标识符 (通常，它对应其 PID，参见第三章“标识一个进程”一节) 转换为队列的索引值；然后，算法将一个新的请求插入该队列的末尾。因此，同一个进程发出的请求通常被插入相同的队列中。

为了再填充调度队列，算法本质上采用轮询方式扫描 I/O 输入队列，选择第一个非空队列，然后将该队列中的一组请求移动到调度队列的末尾。

“最后期限” 算法

除了调度队列外，“最后期限”算法还使用了四个队列。其中的两个排序队列分别包含读请求和写请求，其中的请求是根据起始扇区数排序的。另外两个最后期限队列包含相同的读和写请求，但这是根据它们的“最后期限”排序的。引入这些队列是为了避免请求饿死，由于电梯策略优先处理与上一个所处理的请求最近的请求，因而就会对某个请求忽略很长一段时间，这时就会发生这种情况。请求的最后期限本质上就是一个超时定时器，当请求被传给电梯算法时开始计时。缺省情况下，读请求的超时时间是 500ms，写请求的超时时间是 5s——读请求优先于写请求，因为读请求通常阻塞发出请求的进程。最后期限保证了调度程序照顾等待很长一段时间的那个请求，即使它位于排序队列的末尾。

当算法要补充调度队列时，首先确定下一个请求的数据方向。如果同时要调度读和写两个请求，算法会选择“读”方向，除非该“写”方向已经被放弃很多次了 (为了避免写请求饿死)。

接下来，算法检查与被选择方向相关的最后期限队列：如果队列中的第一个请求的最后期限已用完，那么算法将该请求移到调度队列的末尾，也可以从超时的那个请求开始移动来自排序队列的一组请求。如果将要移动的请求在磁盘上物理相邻，那么组的长度会变长，否则就变短。

最后，如果没有请求超时，算法对来自于排序队列的最后一个请求之后的一组请求进行调度。当指针到达排序队列的末尾时，搜索又从头开始（“单方向算法”）。

“预期”算法

“预期”算法是 Linux 提供的最复杂的一种 I/O 调度算法。基本上，它是“最后期限”算法的一个演变，借用了“最后期限”算法的基本机制：两个最后期限队列和两个排序队列；I/O 调度程序在读和写请之间交互扫描排序队列，不过更倾向于读请求。扫描基本上是连续的，除非有某个请求超时。读请求的缺省超时时间是 125ms，写请求的缺省超时时间是 250ms。但是，该算法还遵循一些附加的启发式准则：

- 有些情况下，算法可能在排序队列当前位置之后选择一个请求，从而强制磁头从后搜索。这种情况通常发生在这个请求之后的搜索距离小于在排序队列当前位置之后对该请求搜索距离的一半时。
- 算法统计系统中每个进程触发的 I/O 操作的种类。当刚刚调度的由某个进程 p 发出的一个读请求之后，算法马上检查排序队列中的下一个请求是否来自同一个进程 p。如果是，立即调度下一个请求。否则，查看关于该进程 p 的统计信息：如果确定进程 p 可能很快发出另一个读请求，那么就延迟一小段时间（缺省大约为 7ms）。因此，算法预测进程 p 发出的读请求与刚被调度的请求在磁盘上可能是“近邻”。

向 I/O 调度程序发出请求

正如我们在本章前面的“提交请求”一节中所看到的，`generic_make_request()` 函数调用请求队列描述符的 `make_request_fn` 方法向 I/O 调度程序发送一个请求。通常该方法是由 `__make_request()` 函数实现的；该函数接收一个 `request_queue` 类型的描述符 `q` 和一个 `bio` 结构的描述符 `bio` 作为其参数，然后执行如下操作：

1. 如果需要，调用 `blk_queue_bounce()` 函数建立一个回弹缓冲区（参见后面）。如果回弹缓冲区被建立，`__make_request()` 函数将对该缓冲区而不是原先的 `bio` 结构进行操作。
2. 调用 I/O 调度程序的 `elv_queue_empty()` 函数检查请求队列中是否存在待处理请求——注意，调度队列可能是空的，但是 I/O 调度程序的其他队列可能包含待处

理请求。如果没有待处理请求，那么调用 `blk_plug_device()` 函数插入请求队列（参见本章前面的“激活块设备驱动程序”一节），然后跳转到第 5 步。

3. 插入的请求队列包含待处理请求。调用 I/O 调度程序的 `elv_merge()` 函数检查新的 bio 结构是否可以并入已存在的请求中。该函数将返回三个可能值：
 - `ELEVATOR_NO_MERGE`: 已经存在的请求中不能包含 bio 结构；这种情况下，跳转到第 5 步。
 - `ELEVATOR_BACK_MERGE`: bio 结构可作为末尾的 bio 而插入到某个请求 `req` 中；这种情形下，函数调用 `q->back_merge_fn` 方法检查是否可以扩展该请求。如果不行，则跳转到第 5 步。否则，将 bio 描述符插入 `req` 链表的末尾并更新 `req` 的相应字段值。然后，函数试图将该请求与其后面的请求合并（新的 bio 可能填充在两个请求之间）。
 - `ELEVATOR_FRONT_MERGE`: bio 结构可作为某个请求 `req` 的第一个 bio 被插入；这种情形下，函数调用 `q->front_merge_fn` 方法检查是否可以扩展该请求。如果不行，则跳转到第 5 步。否则，将 bio 描述符插入 `req` 链表的首部并更新 `req` 的相应字段值。然后，试图将该请求与其前面的请求合并。
4. bio 已经被并入存在的请求中，跳转到第 7 步终止函数。
5. bio 必须被插入到一个新的请求中。分配一个新的请求描述符。如果没有空闲的内存，那么挂起当前进程，直到设置了 `bio->bi_rw` 中的 `BIO_RW_AHEAD` 标志，该标志表明这个 I/O 操作是一次预读（参见第十六章）；在这种情形下，函数调用 `bio_endio()` 并终止；此时将不会执行数据传送。对 `bio_endio()` 函数的描述参见 `generic_make_request()` 函数执行的第 1 步操作（参见前面的“提交请求”一节）。
6. 初始化请求描述符中的字段。主要有：
 - a. 根据 bio 描述符的内容初始化各个字段，包括扇区数、当前 bio 以及当前段。
 - b. 设置 `flags` 字段中的 `REQ_CMD` 标志（一个标准的读或写操作）。
 - c. 如果第一个 bio 段的页框存放在低端内存，则将 `buffer` 字段设置为缓冲区的线性地址。
 - d. 将 `rq_disk` 字段设置为 `bio->bi_bdev->bd_disk` 的地址。
 - e. 将 bio 插入请求链表。
 - f. 将 `start_time` 字段设置为 `jiffies` 的值。
7. 所有操作全部完成。但是，在终止之前，检查是否设置了 `bio->bi_rw` 中的

BIO_RW_SYNC标志。如果是，则对“请求队列”调用generic_unplug_device()函数以卸载设备驱动程序（参见本章前面的“激活块设备驱动程序”一节）。

8. 函数终止。

如果在调用__make_request()函数之前请求队列不是空的，那么说明该请求队列要么已经被拔掉过，要么很快将被拔掉——因为每个拥有待处理请求的插入请求队列q都有一个正在运行的动态定时器q->unplug_timer。另一方面，如果请求队列是空的，则__make_request()函数插入请求队列。或迟（最坏的情况是当拔出的定时器到期了）或早（从__make_request()中退出时，如果设置了bio的BIO_RW_SYNC标志），该请求队列都会被拔掉。任何情形下，块设备驱动程序策略例程最后都将处理调度队列中的请求（参见本章后面的“注册和初始化设备驱动程序”一节）。

blk_queue_bounce()函数

blk_queue_bounce()函数的功能是查看q->bounce_gfp中的标志以及q->bounce_pfn中的阈值，从而确定回弹缓冲区（buffer bouncing）是否是必需的。通常当请求中的一些缓冲区位于高端内存而硬件设备不能访问它们时发生这种情况。

ISA总线使用的老式DMA方式只能处理24位的物理地址。因此，回弹缓冲区的上限设为16 MB，也就是说，页框号为4096。然而，当处理老式设备时，块设备驱动程序通常不依赖回弹缓冲区；相反，它们更倾向于直接在ZONE_DMA内存区中分配DMA缓冲区。

如果硬件设备不能处理高端内存中的缓冲区，则blk_queue_bounce()函数检查bio中的一些缓冲区是否真的必须是回弹的。如果是，则将bio描述符复制一份，接着创建一个回弹bio；然后，当段中的页框号等于或大于q->bounce_pfn的值时，执行下列操作：

1. 根据分配的标志，在ZONE_NORMAL或ZONE_DMA内存区中分配一个页框。
2. 更新回弹bio中段的bv_page字段的值，使其指向新页框的描述符。
3. 如果bio->bio_rw代表一个写操作，则调用kmap()临时将高端内存页映射到内核地址空间中，然后将高端内存页复制到低端内存页上，最后调用kunmap()释放该映射。

然后blk_queue_bounce()函数设置回弹bio中的BIO_BOUNCED标志，为其初始化一个特定的bi_end_io方法，最后将它存放在回弹bio的bi_private字段中，该字段是指向初始bio的指针。当在回弹bio上的I/O数据传送终止时，函数执行bi_end_io方法将数据复制到高端内存区中（仅适合读操作），并释放该回弹bio结构。

块设备驱动程序

块设备驱动程序是Linux块子系统的最底层组件。它们从I/O调度程序中获得请求，然后按要求处理这些请求。

当然，块设备驱动程序是设备驱动程序模型的组成部分（参见第十三章中的“设备驱动程序模型”一节）。因此，每个块设备驱动程序对应一个 `device_driver` 类型的描述符；此外，设备驱动程序处理的每个磁盘都与一个 `device` 描述符相关联。但是，这些描述符没有什么特别的：块 I/O 子系统必须为系统中的每个块设备存放附加信息。

块设备

一个块设备驱动程序可能处理几个块设备。例如，IDE 设备驱动程序可以处理几个 IDE 磁盘，其中的每个都是一个单独的块设备。而且，每个磁盘通常是被分区的，每个分区又可以被看作是一个逻辑块设备。很明显，块设备驱动程序必须处理在块设备对应的块设备文件上发出的所有 VFS 系统调用。

每个块设备都是由一个 `block_device` 结构的描述符来表示的，其字段如表 14-9 所示。

表 14-9：块设备描述符中的字段

类型	字段	说明
<code>dev_t</code>	<code>bd_dev</code>	块设备的主设备号和次设备号
<code>struct inode *</code>	<code>bd_inode</code>	指向 <code>bdev</code> 文件系统中块设备对应的文件索引节点的指针
<code>int</code>	<code>bd_openers</code>	计数器，统计块设备已经被打开了多少次
<code>struct semaphore</code>	<code>bd_sem</code>	保护块设备的打开和关闭的信号量
<code>struct semaphore</code>	<code>bd_mount_sem</code>	禁止在块设备上安装新设备的信号量
<code>struct list_head</code>	<code>bd_inodes</code>	已打开的块设备文件的索引节点链表的首部
<code>void *</code>	<code>bd_holder</code>	块设备描述符的当前所有者
<code>int</code>	<code>bd_holders</code>	计数器，统计对 <code>bd_holder</code> 字段多次设置的次数
<code>struct block_device *</code>	<code>bd_contains</code>	如果块设备是一个分区，则指向整个磁盘的块设备描述符；否则，指向该块设备描述符
<code>unsigned</code>	<code>bd_block_size</code>	块大小
<code>struct hd_struct *</code>	<code>bd_part</code>	指向分区描述符的指针（如果该块设备不是一个分区，则为 <code>NULL</code> ）

表 14-9: 块设备描述符中的字段 (续)

类型	字段	说明
unsigned	bd_part_count	计数器, 统计包含在块设备中的分区已经被打开了多少次
int	bd_invalidated	当需要读块设备的分区表时设置的标志
struct gendisk *	bd_disk	指向块设备中基本磁盘的gendisk结构的指针
struct list_head *	bd_list	用于块设备描述符链表的指针
struct backing_dev_info *	bd_inode_backing_dev_info	指向块设备的专门描述符 backing_dev_info 的指针 (通常为 NULL)
unsigned long	bd_private	指向块设备持有者的私有数据的指针

所有的块设备描述符被插入一个全局链表中, 链表首部是由变量 `all_bdevs` 表示的; 链表链接所用的指针位于块设备描述符的 `bd_list` 字段中。

如果块设备描述符对应一个磁盘分区, 那么 `bd_contains` 字段指向与整个磁盘相关的块设备描述符, 而 `bd_part` 字段指向 `hd_struct` 分区描述符 (参见本章前面的“磁盘和磁盘分区的表示”一节)。否则, 若块设备描述符对应整个磁盘, 那么 `bd_contains` 字段指向块设备描述符本身, `bd_part_count` 字段用于记录磁盘上的分区已经被打开了多少次。

`bd_holder` 字段存放代表块设备持有者的线性地址。持有者并不是进行 I/O 数据传送的块设备驱动程序; 准确地说, 它是一个内核组件, 使用设备并拥有独一无二的特权 (例如, 它可以自由使用块设备描述符的 `bd_private` 字段)。典型地, 块设备的持有者是安装在该设备上的文件系统。当块设备文件被打开进行互斥访问时, 另一个普遍的问题出现了: 持有者就是对应的文件对象。

`bd_claim()` 函数将 `bd_holder` 字段设置为一个特定的地址; 相反, `bd_release()` 函数将该字段重新设置为 `NULL`。然而, 值得注意的是, 同一个内核组件可以多次调用 `bd_claim()` 函数, 每调用一次都增加 `bd_holders` 的值。为了释放块设备, 内核组件必须调用 `bd_release()` 函数 `bd_holders` 次。

图 14-3 对应的是一个整盘, 它说明了块设备描述符是如何被链接到块 I/O 子系统的其他重要数据结构上的。

访问块设备

当内核接收一个打开块设备文件的请求时, 必须首先确定该设备文件是否已经是打开的。事实上, 如果文件已经是打开的, 内核就没有必要创建并初始化一个新的块设备描述符;

相反，内核应该更新这个已经存在的块设备描述符。然而，真正的复杂性在于具有相同主设备号和次设备号但有不同路径名的块设备文件被VFS看作不同的文件，但是它们实际上指向同一个块设备。因此，内核无法通过简单地在一个对象的索引节点高速缓存中检查块设备文件的存在就确定相应的块设备已经在使用。

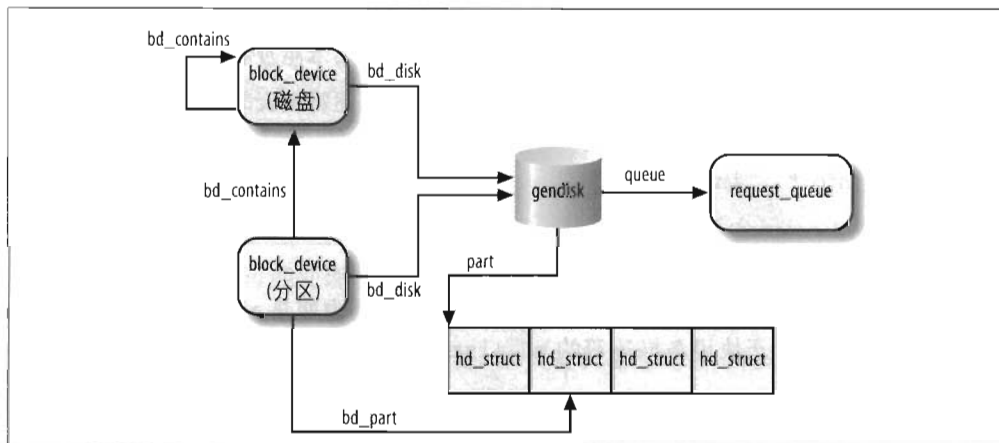


图 14-3：块设备描述符与块子系统其他结构的链接

主、次设备号和相应的块设备描述符之间的关系是通过 *bdev* 特殊文件系统（参见第十二章的“特殊文件系统”一节）来维护的。每个块设备描述符都对应一个 *bdev* 特殊文件：块设备描述符的 *bd_inode* 字段指向相应的 *bdev* 索引节点；而该索引节点则将块设备的主、次设备号和相应描述符的地址进行编码。

bdget() 接收块设备的主设备号和次设备号作为其参数：在 *bdev* 文件系统中查寻相关的索引节点，如果不存在这样的节点，那么就分配一个新索引节点和新块设备描述符。在任何情形下，函数都返回一个与给定主、次设备号对应的块设备描述符的地址。

一旦找到了块设备的描述符，那么内核通过检查 *bd_openers* 字段的值来确定块设备当前是否在使用：如果值是正的，说明块设备已经在使用（可能通过不同的设备文件）。同时内核也维护一个与已打开的块设备文件对应的索引节点对象的链表。该链表存放在块设备描述符的 *bd_inodes* 字段中；索引节点对象的 *i_devices* 字段存放用于链接链表中的前后元素的指针。

注册和初始化设备驱动程序

现在我们来说明一下为一个块设备设计一个新的驱动程序所涉及的基本步骤。显然，其描述是比较简单的，但是理解何时并怎样初始化块 I/O 子系统使用的主要数据结构是很有用的。

我们省略了所有块设备驱动程序需要的但在第十三章中已经讲过的步骤。例如，我们跳过了注册一个驱动程序本身的所有步骤（参见第十三章中的“设备驱动程序模型”一节）。通常，块设备属于一个诸如PCI或SCSI这样的标准总线体系结构，内核提供了相应的辅助函数，作为一个辅助作用，就是在驱动程序模型中注册驱动程序。

自定义驱动程序描述符

首先，设备驱动程序需要一个 `foo_dev_t` 类型的自定义描述符 `foo`，它拥有驱动硬件设备所需的数据。该描述符存放每个设备的相关信息，例如操作设备时使用的 I/O 端口、设备发出中断的 IRQ 线、设备的内部状态等等。同时它也包含块 I/O 子系统所需的一些字段：

```
struct foo_dev_t {
    [...]
    spinlock_t lock;
    struct gendisk *gd;
    [...]
} foo;
```

`lock` 字段是用来保护 `foo` 描述符中字段值的自旋锁，通常将其地址传给内核辅助函数，从而保护对驱动程序而言特定的块 I/O 子系统的数据结构。`gd` 字段是指向 `gendisk` 描述符的指针，该描述符描述由这个驱动程序处理的整个块设备（磁盘）。

预订主设备号

设备驱动程序必须自己预订一个主设备号。传统上，该操作通过调用 `register_blkdev()` 函数完成：

```
err = register_blkdev(FOO_MAJOR, "foo");
if (err) goto error_major_is_busy;
```

该函数类似于第十三章的“分配设备号”一节中出现的 `register_chrdev()` 函数：预订主设备号 `FOO_MAJOR` 并将设备名称 `foo` 赋给它。注意，这不能分配次设备号范围，因为没有类似的 `register_chrdev_region()` 函数；此外，预订的主设备号和驱动程序的数据结构之间也没有建立链接。`register_blkdev()` 函数产生的唯一可见的效果是包含一个新条目，该条目位于 `/proc/devices` 特殊文件的已注册主设备号列表中。

初始化自定义描述符

在使用驱动程序之前必须适当地初始化 `foo` 描述符中的所有字段。为了初始化与块 I/O 子系统相关的字段，设备驱动程序主要执行如下操作：

```
spin_lock_init(&foo.lock);
```



```
foo.gd = alloc_disk(16);
if (!foo.gd) goto error_no_gendisk;
```

驱动程序首先初始化自旋锁，然后分配一个磁盘描述符。正如在前面的图 14-3 中所看到的，gendisk 结构是块 I/O 子系统中最重要的数据结构，因为它涉及许多其他的数据结构。alloc_disk() 函数也分配一个存放磁盘分区描述符的数组。该函数所需要的参数是数组中 hd_struct 结构的元素个数，16 表示驱动程序可以支持 16 个磁盘，而每个磁盘可以包含 15 个分区（0 分区不使用）。

初始化 gendisk 描述符

接下来，驱动程序初始化 gendisk 描述符的一些字段：

```
foo.gd->private_data = &foo;
foo.gd->major = FOO_MAJOR;
foo.gd->first_minor = 0;
foo.gd->minors = 16;
set_capacity(foo.gd, foo_disk_capacity_in_sectors);
strcpy(foo.gd->disk_name, "foo");
foo.gd->fops = &foo_ops;
```

foo 描述符的地址存放在 gendisk 结构的 private_data 字段中，因此被块 I/O 子系统当作方法调用的低级驱动程序函数可以迅速地查找到驱动程序描述符——如果驱动程序可以并发地处理多个磁盘，那么这种方式可以提高效率。set_capacity() 函数将 capacity 字段初始化为以 512 字节扇区为单位的磁盘大小——这个值也可能在探测硬件并询问磁盘参数时确定。

初始化块设备操作表

gendisk 描述符的 fops 字段被初始化为自定义的块设备方法表的地址（参见本章前面的表 14-4）（注 3）。类似地，设备驱动程序的 foo_ops 表中包含设备驱动程序的特有函数。例如，如果硬件设备支持可移动磁盘，通用块层将调用 media_changed 方法检查自从最后一次安装或打开该块设备以来磁盘是否被更换。通常通过向硬件控制器发送一些低级命令来完成该检查，因此，每个设备驱动程序所实现的 media_changed 方法都是不同的。

类似地，仅当通用块层不知道如何处理 ioctl 命令时才调用 ioctl 方法。例如，当一个 ioctl() 系统调用询问磁盘构造时，也就是磁盘使用的柱面数、磁道数、扇区数以及磁头数时，通常调用该方法。因此，每个设备驱动程序所实现的 ioctl 方法也都是不同的。

注 3： 不应该把块设备方法和块设备文件操作混为一谈，参见本章稍后“打开块设备文件”一节。

分配和初始化请求队列

我们勇敢的设备驱动程序设计者现在将要建立一个请求队列,该队列用于存放等待处理的请求。可以通过如下操作轻松地建立请求队列:

```
foo.gd->rq = blk_init_queue(foo_strategy, &foo.lock);
if (!foo.gd->rq) goto error_no_request_queue;
blk_queue_hardsect_size(foo.gd->rd, foo_hard_sector_size);
blk_queue_max_sectors(foo.gd->rd, foo_max_sectors);
blk_queue_max_hw_segments(foo.gd->rd, foo_max_hw_segments);
blk_queue_max_phys_segments(foo.gd->rd, foo_max_phys_segments);
```

`blk_init_queue()`函数分配一个请求队列描述符并将其中许多字段初始化为缺省值。它接收的参数为设备描述符的自旋锁的地址 (`foo.gd->rq->queue_lock` 字段值) 和设备驱动程序的策略例程 (参见下一节“策略例程”) 的地址 (`foo.gd->rq->request_fn` 字段值)。该函数也初始化 `foo.gd->rq->elevator` 字段, 并强制驱动程序使用缺省的 I/O 调度算法。如果设备驱动程序想要使用其他的调度算法, 可在稍后覆盖 `elevator` 字段的地址。

接下来, 使用几个辅助函数将请求队列描述符的不同字段设为设备驱动程序的特征值 (参考表 14-6 中的类似字段)。

设置中断处理程序

正如第四章的“I/O 中断处理”一节中所介绍的, 设备驱动程序需要为设备注册 IRQ 线。这可以通过如下操作完成:

```
request_irq(foo_irq, foo_interrupt,
            SA_INTERRUPT|SA_SHIRQ, "foo", NULL);
```

`foo_interrupt()` 函数是设备的中断处理程序, 我们将在本章稍后的“中断处理程序”一节中讨论它的一些特性。

注册磁盘

最后, 设备驱动程序的所有数据结构已经准备好了: 初始化阶段的最后一步就是“注册”和激活磁盘。这可以简单地通过执行下面的操作完成:

```
add_disk(foo.gd);
```

`add_disk()` 函数接收 `gendisk` 描述符的地址作为其参数, 主要执行下列操作:

1. 设置 `gd->flags` 的 `GENHD_FL_UP` 标志。
2. 调用 `kobj_map()` 建立设备驱动程序和设备的主设备号 (连同相关范围内的次设备

- 号)之间的连接(参见第十三章的“字符设备驱动程序”一节;注意,在这种情况下, `kobject` 映射域由 `bdev_map` 变量表示)。
3. 注册设备驱动程序模型的 `gendisk` 描述符中的 `kobject` 结构,它作为设备驱动程序处理的一个新设备(例如 `/sys/block/foo`)。
 4. 如果需要,扫描磁盘中的分区表;对于查找到的每个分区,适当地初始化 `foo.gd->part` 数组中相应的 `hd_struct` 描述符。同时注册设备驱动程序模型中的分区(例如 `/sys/block/foo/fool`)。
 5. 注册设备驱动程序模型的请求队列描述符中内嵌的 `kobject` 结构(例如 `/sys/block/foo/queue`)。

一旦 `add_disk()` 返回,设备驱动程序就可以工作了。进行初始化的函数终止,策略例程和中断处理程序开始处理 I/O 调度程序传送给设备驱动程序的每个请求。

策略例程

策略例程是块设备驱动程序的一个函数或一组函数,它与硬件块设备之间相互作用以满足调度队列中所汇集的请求。通过请求队列描述符中的 `request_fn` 方法可以调用策略例程——例如前面一节介绍的 `foo_strategy()` 函数, I/O 调度程序层将请求队列描述符 `q` 的地址传递给该函数。

如前所述,在把新的请求插入到空的请求队列后,策略例程通常才被启动。只要块设备驱动程序被激活,就应该对队列中的所有请求都进行处理,直到队列为空才结束。

策略例程的简单实现如下:对于调度队列中的每个元素,与块设备控制器相互作用共同为请求服务,等待直到数据传送完成,然后把已经服务过的请求从队列中删除,继续处理调度队列中的下一个请求。

这种实现效率并不高。即使假设可以使用 DMA 传送数据,策略例程在等待 I/O 操作完成的过程中也必须自行挂起。也就是说策略例程应该在一个专门的内核线程上执行(我们不想处罚毫不相关的用户进程)。而且,这样的驱动程序也不能支持可以一次处理多个 I/O 数据传送的现代磁盘控制器。

因此,很多块设备驱动程序都采用如下策略:

- 策略例程处理队列中的第一个请求并设置块设备控制器,以便在数据传送完成时可以产生一个中断。然后策略例程就终止。
- 当磁盘控制器产生中断时,中断控制器重新调用策略例程(通常是直接的,有时也通过激活一个工作队列)。策略例程要么为当前请求再启动一次数据传送,要么当

请求的所有数据块已经传送完成时,把该请求从调度队列中删除然后开始处理下一个请求。

请求是由几个 bio 结构组成的,而每个 bio 结构又是由几个段组成的。基本上,块设备驱动程序以以下两种方式使用 DMA:

- 驱动程序建立不同的 DMA 传送方式,为请求的每个 bio 结构中的每个段进行服务。
- 驱动程序建立一种单独的分散 - 聚集 DMA 传送方式,为请求的所有 bio 中的所有段服务。

最后,设备驱动程序策略例程的设计依赖块控制器的特性。每个物理块设备都有不同于其他物理块设备的固有特性(例如,软盘驱动程序把磁道上的块分组为磁道,一次单独的 I/O 操作传送整个磁道),因此对设备驱动程序怎样为每个请求进行服务而做一般假设并没有多大意义。

在我们的例子中, `foo_strategy()` 策略例程应该执行以下操作:

1. 通过调用 I/O 调度程序的辅助函数 `elv_next_request()` 从调度队列中获取当前的请求。如果调度队列为空,就结束这个策略例程:

```
req = elv_next_request(q);
if (!req) return;
```

2. 执行 `blk_fs_request` 宏检查是否设置了请求的 `REQ_CMD` 标志,也即,请求是否包含一个标准的读或写操作:

```
if (!blk_fs_request(req))
    goto handle_special_request;
```

3. 如果块设备控制器支持分散 - 聚集 DMA,那么对磁盘控制器进行编程,以便为整个请求执行数据传送并在传送完成时产生一个中断。`blk_rq_map_sg()` 辅助函数返回一个可以立即被用来启动数据传送的分散 - 聚集链表。

4. 否则,设备驱动程序必须一段一段地传送数据。在这种情形下,策略例程执行 `rq_for_each_bio` 和 `bio_for_each_segment` 两个宏,分别遍历 bio 链表和每个 bio 中的段链表。

```
rq_for_each_bio(bio, rq)
    bio_for_each_segment(bvec, bio, i) {
        /* transfer the i-th segment bvec */
        local_irq_save(flags);
        addr = kmap_atomic(bvec->bv_page, KM_BIO_SRC_IRQ);
        foo_start_dma_transfer(addr+bvec->bv_offset, bvec->bv_len);
        kunmap_atomic(bvec->bv_page, KM_BIO_SRC_IRQ);
        local_irq_restore(flags);
    }
```

如果要传送的数据在高端内存中，那么 `kmap_atomic()` 和 `kunmap_atomic()` 两个函数就是必需的。`foo_start_dma_transfer()` 函数对硬件设备进行编程，以便启动 DMA 数据传送并在 I/O 操作完成时产生一个中断。

5. 返回。

中断处理程序

块设备驱动程序的中断处理程序是在 DMA 数据传送结束时被激活的。它检查是否已经传送完请求的所有数据块。如果是，中断处理程序就调用策略例程处理调度队列中的下一个请求。否则，中断处理程序更新请求描述符的相应字段并调用策略例程处理还没有完成的数据传送。

我们的设备驱动程序 `foo` 的中断处理程序的一个典型片段如下：

```
irqreturn_t foo_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct foo_dev_t *p = (struct foo_dev_t *) dev_id;
    struct request_queue *rq = p->gd->rq;
    [...]
    if (!end_that_request_first(rq, uptodate, nr_sectors)) {
        blkdev_dequeue_request(rq);
        end_that_request_last(rq);
    }
    rq->request_fn(rq);
    [...]
    return IRQ_HANDLED;
}
```

`end_that_request_first()` 和 `end_that_request_last()` 两个函数共同承担结束一个请求的任务。

`end_that_request_first()` 函数接收的参数为一个请求描述符、一个指示 DMA 数据传送成功完成的标志以及 DMA 所传送的扇区数（`end_that_request_chunk()` 函数类似，只不过该函数接收的是传送的字节数而不是扇区数）。本质上，它扫描请求中的 bio 结构以及每个 bio 中的段，然后采用如下方式更新请求描述符的字段值：

- 修改 bio 字段使其指向请求中的第一个未完成的 bio 结构。
- 修改未完成 bio 结构的 `bi_idx` 字段使其指向第一个未完成的段。
- 修改未完成段的 `bv_offset` 和 `bv_len` 两个字段使其指定仍需传送的数据。

该函数也在每个已经完成数据传送的 bio 结构上调用 `bio_endio()` 函数。

如果已经传送完请求中的所有数据块，那么 `end_that_request_first()` 返回 0，否则

返回 1。如果返回值是 1，则中断处理程序重新调用策略例程，继续处理该请求。否则，中断处理程序把请求从请求队列中删除（主要由 `blkdev_dequeue_request()` 完成），然后调用 `end_that_request_last()` 辅助函数，并再次调用策略例程处理调度队列中的下一个请求。

`end_that_request_last()` 函数的功能是更新一些磁盘使用统计数，把请求描述符从 I/O 调度程序 `rq->elevator` 的调度队列中删除，唤醒等待请求描述符完成的任一睡眠进程，并释放删除的那个描述符。

打开块设备文件

通过描述打开一个块设备文件时 VFS 所执行的操作，我们将总结本章的内容。

每当一个文件系统被映射到磁盘或分区上时，每当激活一个交换分区时，每当用户态进程向块设备文件发出一个 `open()` 系统调用时，内核都会打开一个块设备文件。在所有情况下，内核本质上执行相同的操作：寻找块设备描述符（如果块设备没有在使用，那么就分配一个新的描述符），为即将开始的数据传送设置文件操作方法。

我们已经在第十三章的“设备文件的 VFS 处理”一节描述了当一个设备文件被打开时，`dentry_open()` 函数是如何定制文件对象的方法。它的 `f_op` 字段设置为表 `def_blk_fops` 的地址，该表的内容如表 14-10 所示。

表 14-10：缺省的块设备文件操作（表 `def_blk_fops`）

方法	用于块设备文件的函数
<code>open</code>	<code>blkdev_open()</code>
<code>release</code>	<code>blkdev_close()</code>
<code>llseek</code>	<code>block_llseek()</code>
<code>read</code>	<code>generic_file_read()</code>
<code>write</code>	<code>blkdev_file_write()</code>
<code>aio_read</code>	<code>generic_file_aio_read()</code>
<code>aio_write</code>	<code>blkdev_file_aio_write()</code>
<code>mmap</code>	<code>generic_file_mmap()</code>
<code>fsync</code>	<code>block_fsync()</code>
<code>ioctl</code>	<code>block_ioctl()</code>
<code>compat_ioctl</code>	<code>compat_blkdev_ioctl()</code>
<code>readv</code>	<code>generic_file_readv()</code>

表 14-10: 缺省的块设备文件操作 (表 def_blk_fops) (续)

方法	用于块设备文件的函数
writev	generic_file_write_nolock()
sendfile	generic_file_sendfile()

我们仅仅考虑 open 方法，它由 dentry_open() 函数调用。blkdev_open() 接收 inode 和 filp 作为其参数，它们分别存放了索引节点和文件对象的地址；该函数本质上执行下列操作：

1. 执行 bd_acquire(inode) 从而获得块设备描述符 bdev 的地址。该函数接收索引节点对象的地址并执行下列主要步骤：
 - a. 检查索引节点对象的 inode->i_bdev 字段是否不为 NULL；如果是，表明块设备文件已经打开了，该字段存放了相应块描述符的地址。在这种情况下，增加与块设备相关联的 bdev 特殊文件系统的 inode->i_bdev->bd_inode 索引节点的引用计数器的值，并返回描述符 inode->i_bdev 的地址。
 - b. 块设备文件没有被打开的情况。根据块设备文件的主设备号和次设备号（参见本章前面的“块设备”一节），执行 bdget(inode->i_rdev) 获取块设备描述符的地址。如果描述符不存在，bdget() 就分配一个；但是，要注意的是描述符可能已经存在，例如其他块设备文件已经访问了该块设备。
 - c. 将块设备描述符的地址存放在 inode->i_bdev 中，以便加速将来对相同块设备文件的打开操作。
 - d. 将 inode->i_mapping 字段设置为 bdev 索引节点中相应字段的值。该字段指向地址空间对象，我们将在第十五章的“address_space 对象”一节中介绍这个对象。
 - e. 把索引节点插入到由 bdev->bd_inodes 确立的块设备描述符的已打开索引节点链表中。
 - f. 返回描述符 bdev 的地址。
2. 将 filp->i_mapping 字段设置为 inode->i_mapping 的值（参见前面的第 1d 步）。
3. 获取与这个块设备相关的 gendisk 描述符的地址：

```
disk = get_gendisk(bdev->bd_dev, &part);
```

如果被打开的块设备是一个分区，则返回的索引值存放在本地变量 part 中；否则，part 为 0。get_gendisk() 函数在 kobject 映射域 bdev_map 上简单地调用

`kobj_lookup()`来传递设备的主设备号和次设备号（参见本章前面的“注册和初始化设备驱动程序”一节）。

4. 如果 `bdev->bd_openers` 的值不等于 0，表明块设备已经被打开了。检查 `bdev->bd_contains` 字段：
 - a. 如果值等于 `bdev`，那么块设备是一个整盘：调用块设备方法 `bdev->bd_disk->fops->open`（如果定义了），然后检查 `bdev->bd_invalidated` 字段的值，如果需要，调用 `rescan_partitions()` 函数（参见后面的第 6a 步和第 6c 步）。
 - b. 如果不等于 `bdev`，那么块设备是一个分区：增加 `bdev->bd_contains->bd_part_count` 计数器的值。然后跳到第 8 步。
5. 这里块设备是第一次被访问。初始化 `bdev->bd_disk` 为 `gendisk` 描述符的地址 `disk`。
6. 如果块设备是一个整盘（`part` 等于 0），则执行下列子步骤：
 - a. 如果定义了 `disk->fops->open` 块设备方法，就执行它：该方法是由块设备驱动程序定义的定制函数，它执行任何特定的最后一分钟初始化。
 - b. 从 `disk->queue` 请求队列的 `hardsect_size` 字段中获取扇区大小（字节数），使用该值适当地设置 `bdev->bd_block_size` 和 `bdev->bd_inode->i_blkbits` 两个字段。同时用从 `disk->capacity` 中计算来的磁盘大小设置 `bdev->bd_inode->i_size` 字段。
 - c. 如果设置了 `bdev->bd_invalidated` 标志，那么调用 `rescan_partitions()` 扫描分区表并更新分区描述符。该标志是由 `check_disk_change` 块设备方法设置的，仅适用于可移动设备。
7. 否则如果块设备是一个分区，则执行下列子步骤：
 - a. 再次调用 `bdget()` —— 这次是传递 `disk->first_minor` 次设备号 —— 获取整盘的块描述符地址 `whole`。
 - b. 对整盘的块设备描述符重复第 3 步 ~ 第 6 步，如果需要则初始化该描述符。
 - c. 将 `bdev->bd_contains` 设置为整盘描述符的地址。
 - d. 增加 `whole->bd_part_count` 的值从而说明磁盘分区上新的打开操作。
 - e. 用 `disk->part[part-1]` 中的值设置 `bdev->bd_part`，它是分区描述符 `hd_struct` 的地址。同样，执行 `kobject_get(&bdev->bd_part->kobj)` 增加分区引用计数器的值。
 - f. 与第 6b 步中的一样，设置索引节点中表示分区大小和扇区大小的字段。
8. 增加 `bdev->bd_openers` 计数器的值。

9. 如果块设备文件以独占方式被打开（设置了 `filp->f_flags` 中的 `O_EXCL` 标志），那么调用 `bd_claim(bdev, filp)` 设置块设备的持有者（参见本章前面的“块设备”一节）。万一出错——块设备已经有一个拥有者——释放该块设备描述符并返回一个错误码 `-EBUSY`。
10. 返回 0（成功）终止。

`blkdev_open()` 函数一旦终止，`open()` 系统调用如往常一样继续进行。对已打开的文件上将来发出的每个系统调用都将触发一个缺省的块设备文件操作。正如我们将在第十六章中看到的，采用向通用块层提交请求的方式进行块设备的每次数据传送都是高效率的。



页高速缓存

正如在第十二章的“通用文件模型”一节中提到的那样，磁盘高速缓存是一种软件机制，它允许系统把通常存放在磁盘上的一些数据保留在RAM中，以便对那些数据的进一步访问不用再访问磁盘而能尽快得到满足。

因为对同一磁盘数据的反复访问频繁发生，所以磁盘高速缓存对系统性能至关重要。与磁盘交互的用户态进程有权反复请求读或写同一磁盘数据。此外，不同的进程可能也需要在不同的时间访问相同的磁盘数据。例如，你可以使用 *cp* 命令拷贝一个文本文件，然后调用你喜欢的编辑器修改它。为了满足你的请求，命令 shell 将创建两个不同的进程，它们在不同的时间访问同一个文件。

我们曾在第十二章提到过其他的磁盘高速缓存：目录项高速缓存和索引节点高速缓存，前者存放的是描述文件系统路径名的目录项对象，而后者存放的是描述磁盘索引节点的索引节点对象。不过要注意，目录项对象和索引节点对象不只是存放一些磁盘块内容的缓冲区；由此而知，目录项高速缓存和索引节点高速缓存就像磁盘高速缓存那样相当独特。

本章介绍页高速缓存——一种对完整的数据页进行操作的磁盘高速缓存。我们在第一节介绍页高速缓存。然后，我们在“把块存放在页高速缓存中”一节中讨论如何使用页高速缓存快速检索单独的数据块（例如超级块和索引节点），这是提高虚拟文件系统和磁盘文件系统速度的关键特征。接下来，我们在“把脏页写入磁盘”一节描述如何把页高速缓存中的脏页写回到磁盘中。最后，我们在“*sync()*、*fsync()*和*fdatasync()*系统调用”一节中介绍一些系统调用，让用户刷新页高速缓存的内容来更新磁盘内容。

页高速缓存

页高速缓存 (*page cache*) 是 Linux 内核所使用的主要磁盘高速缓存。在绝大多数情况下, 内核在读写磁盘时都引用页高速缓存。新页被追加到页高速缓存以满足用户态进程的读请求。如果页不在高速缓存中, 新页就被加到高速缓存中, 然后用从磁盘读出的数据填充它。如果内存有足够的空闲空间, 就让该页在高速缓存中长期保留, 使其他进程再使用该页时不再访问磁盘。

同样, 在把一页数据写到块设备之前, 内核首先检查对应的页是否已经在高速缓存中, 如果不在, 就要先在其中增加一个新项, 并用要写到磁盘中的数据填充该项。I/O 数据的传送并不是马上开始, 而是要延迟几秒之后才对磁盘进行更新, 从而使进程有机会对要写入磁盘的数据做进一步的修改 (换句话说, 就是内核执行延迟的写操作)。

内核的代码和内核数据结构不必从磁盘读, 也不必写入磁盘 (注 1), 因此, 页高速缓存中的页可能是下面的类型:

- 含有普通文件数据的页。在第十六章我们将描述内核如何处理它们的读、写和内存映射操作。
- 含有目录的页。在第十八章我们将会看到, Linux 采用与普通文件类似的方式操作目录文件。
- 含有直接从块设备文件 (跳过文件系统层) 读出的数据的页。正如我们将在第十六章讨论的那样, 内核处理这种页与处理含有普通文件的页使用相同的函数集合。
- 含有用户态进程数据的页, 但页中的数据已经被交换到磁盘。在第十七章我们将会看到, 内核可能会强行在页高速缓存中保留一些页面, 而这些页面中的数据已经被写到交换区 (可能是普通文件或磁盘分区)。
- 属于特殊文件系统文件的页, 如共享内存的进程间通信 (Interprocess Communication, IPC) 所使用的特殊文件系统 *shm* (参见第十九章)。

正如你所看到的, 页高速缓存中的每个页所包含的数据肯定属于某个文件。这个文件 (或者更准确地说 是文件的索引节点) 就称为页的所有者 (*owner*)。 (在第十七章我们会了解到, 含有换出数据的页都属于同一个所有者, 即使它们涉及不同的交换区。)

注 1: 如果要在关机后恢复系统的所有状态 (其实几乎不会出现这种情况), 可以执行“挂起到磁盘”操作 (*hibernation*), 把 RAM 的全部内容保存到交换区, 对此我们不做更多的讨论。

几乎所有的文件读和写操作都依赖于页高速缓存。只有在`O_DIRECT`标志被置位而进程打开文件的情况下才会出现例外：此时，I/O数据的传送绕过了页高速缓存而使用了进程用户态地址空间的缓冲区（参见第十六章“直接I/O传送”一节）；少数数据库应用软件为了能采用自己的磁盘高速缓存算法而使用了`O_DIRECT`标志。

内核设计者实现页高速缓存主要为了满足下面两种需要：

- 快速定位含有给定所有者相关数据的特定页。为了尽可能充分发挥页高速缓存的优势，对它应该采用高速的搜索操作。
- 记录在读或写页中的数据时应当如何处理高速缓存中的每个页。例如，从普通文件、块设备文件或交换区读一个数据页必须用不同的实现方式，因此内核必须根据页的所有者选择适当的操作。

页高速缓存中的信息单位显然是一个完整的数据页。在第十八章我们会看到，一个页中包含的磁盘块在物理上不一定是相邻的，所以不能用设备号和块号来识别它，取而代之的是，通过页的所有者和所有者数据中的索引（通常是一个索引节点和在相应文件中的偏移量）来识别页高速缓存中的页。

address_space 对象

页高速缓存的核心数据结构是`address_space`对象，它是一个嵌入在页所有者的索引节点对象中的数据结构（注2）。高速缓存中的许多页可能属于同一个所有者，从而可能被链接到同一个`address_space`对象。该对象还在所有者的页和对这些页的操作之间建立起链接关系。

每个页描述符都包括把页链接到页高速缓存的两个字段`mapping`和`index`（参见第八章“页描述符”一节）。`mapping`字段指向拥有页的索引节点的`address_space`对象，`index`字段表示在所有者的地址空间中以页大小为单位的偏移量，也就是在所有者的磁盘映像中页中数据的位置。在页高速缓存中查找页时使用这两个字段。

值得庆幸的是，页高速缓存可以包含同一磁盘数据的多个副本。例如，可以用下述方式访问普通文件的同一4KB的数据块：

- 读文件；因此，数据就包含在普通文件的索引节点所拥有的页中。

注2： 页被换出可能会引起缺页异常。在第十七章我们将看到，这些被换出的页拥有不在任何索引节点中的公共`address_space`对象。

- 从文件所在的设备文件（磁盘分区）读取块，因此，数据就包含在块设备文件的主索引节点所拥有的页中。

因此，两个不同 `address_space` 对象所引用的两个不同的页中出现了相同的磁盘数据。

`address_space` 对象包含如表 15-1 所示的字段。

表 15-1: `address_space` 对象的字段

类型	字段	说明
<code>struct inode *</code>	<code>host</code>	指向拥有该对象的索引节点的指针（如果存在）
<code>struct radix_tree_root</code>	<code>page_tree</code>	表示拥有者页的基树（radix tree）的根
<code>spinlock_t</code>	<code>tree_lock</code>	保护基树的自旋锁
<code>unsigned int</code>	<code>i_mmap_writable</code>	地址空间中共享内存映射的个数
<code>struct prio_tree_root</code>	<code>i_mmap</code>	radix 优先搜索树的根（参见第十七章）
<code>struct list_head</code>	<code>i_mmap_nonlinear</code>	地址空间中非线性内存区的链表
<code>spinlock_t</code>	<code>i_mmap_lock</code>	保护 radix 优先搜索树的自旋锁
<code>unsigned int</code>	<code>truncate_count</code>	截断文件时使用的顺序计数器
<code>unsigned long</code>	<code>nropages</code>	所有者的页总数
<code>unsigned long</code>	<code>writeback_index</code>	最后一次回写操作所作用的页的索引
<code>struct address_space_operations *</code>	<code>a_ops</code>	对所有者页进行操作的方法
<code>unsigned long</code>	<code>flags</code>	错误位和内存分配器的标志
<code>struct backing_dev_info *</code>	<code>backing_dev_info</code>	指向拥有所有者数据的块设备的 <code>backing_dev_info</code> 的指针
<code>spinlock_t</code>	<code>private_lock</code>	通常是管理 <code>private_list</code> 链表时使用的自旋锁
<code>struct list_head</code>	<code>private_list</code>	通常是与索引节点相关的间接块的脏缓冲区的链表
<code>struct address_space *</code>	<code>assoc_mapping</code>	通常是指向间接块所在块设备的 <code>address_space</code> 对象的指针

如果页高速缓存中页的所有者是一个文件，`address_space` 对象就嵌入在 VFS 索引节点对象的 `i_data` 字段中。索引节点的 `i_mapping` 字段总是指向索引节点的数据页所有者的 `address_space` 对象。`address_space` 对象的 `host` 字段指向其所有者的索引节点对象。

因此，如果页属于一个文件（存放在Ext3文件系统中），那么页的所有者就是文件的索引节点，而且相应的 `address_space` 对象存放在 VFS 索引节点对象的 `i_data` 字段中。索引节点的 `i_mapping` 字段指向同一个索引节点的 `i_data` 字段，而 `address_space` 对象的 `host` 字段也指向这个索引节点。

不过，有些时候情况会更复杂。如果页中包含的数据来自块设备文件，即页含有存放着块设备的“原始”数据，那么就把 `address_space` 对象嵌入到与该块设备相关的特殊文件系统 `bdev` 中文件的“主”索引节点中（块设备描述符的 `bd_inode` 字段引用这个索引节点，参见第十四章“块设备”一节）。因此，块设备文件对应索引节点的 `i_mapping` 字段指向主索引节点中的 `address_space` 对象。相应地，`address_space` 对象的 `host` 字段指向主索引节点。这样，从块设备读取数据的所有页具有相同的 `address_space` 对象，即使这些数据位于不同的块设备文件。

`i_mmap`、`i_mmap_writable`、`i_mmap_nonlinear` 和 `i_mmap_lock` 字段涉及内存映射和反映射，我们将在第十六、十七章讨论这些主题。

`backing_dev_info` 字段指向 `backing_dev_info` 描述符，后者是对所有者的数据所在块设备进行有关描述的数据结构。正如在第十四章“请求队列描述符”一节所描述的，`backing_dev_info` 结构通常嵌入在块设备的请求队列描述符中。

`private_list` 字段是普通链表的首部，文件系统在实现其特定功能时可以随意使用。例如，Ext2 文件系统利用这个链表收集与索引节点相关的“间接”块的脏缓冲区（参见第十八章“数据块寻址”一节）。当刷新操作把索引节点强行写入磁盘时，内核也同时刷新该链表中的所有缓冲区。此外，Ext2 文件系统在 `assoc_mapping` 字段中存放指向间接块所在块设备的 `address_space` 对象，并使用 `assoc_mapping->private_lock` 自旋锁保护多处理器系统中的间接块链表。

`address_space` 对象的关键字段是 `a_ops`，它指向一个类型为 `address_space_operations` 的表，表中定义了对所有者的页进行处理的各种方法。这些方法如表 15-2 所示。

表 15-2: `address_space` 对象的方法

方法	说明
<code>writepage</code>	写操作（从页写到所有者的磁盘映像）
<code>readpage</code>	读操作（从所有者的磁盘映像读到页）
<code>sync_page</code>	如果对所有者页进行的操作已准备好，则立刻开始 I/O 数据的传输
<code>writepages</code>	把指定数量的所有者脏页写回磁盘
<code>set_page_dirty</code>	把所有者的页设置为脏页

表 15-2: `address_space` 对象的方法 (续)

方法	说明
<code>readpages</code>	从磁盘中读所有者页的链表
<code>prepare_write</code>	为写操作做准备 (由磁盘文件系统使用)
<code>commit_write</code>	完成写操作 (由磁盘文件系统使用)
<code>bmap</code>	从文件块索引中获取逻辑块号
<code>invalidatepage</code>	使所有者的页无效(截断文件时使用)
<code>releasepage</code>	由日志文件系统使用以准备释放页
<code>direct_IO</code>	所有者页的直接 I/O 传输 (绕过页高速缓存)

最重要的方法是 `readpage`、`writpage`、`prepare_write` 和 `commit_write`。我们将在第十六章对它们进行讨论。在绝大多数情况下，这些方法把所有者的索引节点对象和访问物理设备的低级驱动程序联系起来。例如，为普通文件的索引节点实现 `readpage` 方法的函数知道如何确定文件页的对应块在物理磁盘设备上的位置。不过，我们不必在本章进一步讨论 `address_space` 的方法。

基树

Linux 支持大到几个 TB 的文件。访问大文件时，页高速缓存中可能充满太多的文件页，以至于顺序扫描这些页要消耗大量的时间。为了实现页高速缓存的高效查找，Linux 2.6 采用了大量的搜索树，其中每个 `address_space` 对象对应一棵搜索树。

`address_space` 对象的 `page_tree` 字段是基树 (*radix tree*) 的根，它包含指向所有者的页描述符的指针。给定的页索引表示页在所有者磁盘映像中的位置，内核能够通过快速搜索操作来确定所需要的页是否在页高速缓存中。当查找所需要的页时，内核把页索引转换为基树中的路径，并快速找到页描述符所 (或应当) 在的位置。如果找到，内核可以从基树获得页描述符，而且还可以很快确定所找到的页是否是脏页 (也就是应当被刷新到磁盘的页)，以及其数据的 I/O 传送是否正在进行。

基树的每个节点可以有多到 64 个指针指向其他节点或页描述符。底层节点存放指向页描述符的指针 (叶子节点)，而上层的节点存放指向其他节点 (孩子节点) 的指针。每个节点由 `radix_tree_node` 数据结构表示，它包括三个字段：`slots` 是包括 64 个指针的数组，`count` 是记录节点中空指针数量的计数器，`tags` 是二维的标志数组，在本章稍后“基树的标记”一节将对其进行讨论。树根由 `radix_tree_root` 数据结构表示，它有三个字段：`height` 表示树的当前深度 (不包括叶子节点的层数)，`gfp_mask` 指定为新节点请求内存时所用的标志，`rnode` 指向与树中第一层节点相应的数据结构 `radix_tree_node` (如果有的话)。

我们来看一个简单的例子。如果树中没有索引大于63，那么树的深度就等于1，因为可能存在的64个叶子可以都存放在第一层的节点中 [如图 15-1 (a) 所示]。不过，如果与索引131相应的新页的描述符肯定存放在页高速缓存中，那么树的深度就增加为2，这样基树就可以查找多达 4 095 个索引 [如图 15-1 (b) 所示]。

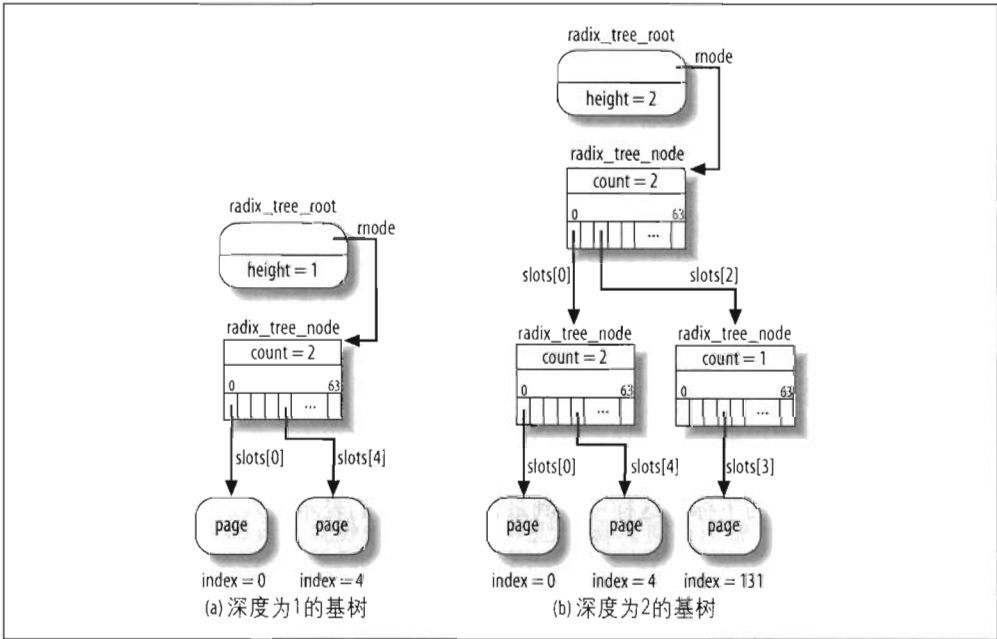


图 15-1：基树的两个例子

表 15-3 显示了页索引的最大值和基于 32 位体系结构的基树中与每个给定深度相应的文件的最大长度。在这里，基树的最大深度是 6，当然系统中的页高速缓存不大可能使用那么大的基树。因为页索引存放在 32 位变量中，当树的深度为 6 时，最高层的节点最多可以有 4 个孩子节点。

表 15-3：每个基树深度对应的最大索引值和文件的最大长度

基树深度	最大索引值	文件的最大长度
0	0	0 字节
1	$2^6 - 1 = 63$	256 KB
2	$2^{12} - 1 = 4\,095$	16 MB
3	$2^{18} - 1 = 262\,143$	1 GB
4	$2^{24} - 1 = 16\,777\,215$	64 GB

表 15-3: 每个基树深度对应的最大索引值和文件的最大长度 (续)

基树深度	最大索引值	文件的最大长度
5	$2^{30}-1 = 1\ 073\ 741\ 823$	4 TB
6	$2^{32}-1 = 4\ 294\ 967\ 295$	16 TB

回顾一下分页系统是如何利用页表实现线性地址到物理地址转换的,从而理解如何实现页查找。正如第二章“常规分页”一节所讨论的,线性地址最高 20 位分成两个 10 位的字段:第一个字段是页目录中的偏移量,而第二个字段是某个页目录项所指向的页表中的偏移量。

基树中使用类似的方法。页索引相当于线性地址,不过页索引中要考虑的字段的数量依赖于基树的深度。如果基树的深度为 1,就只能表示从 0~63 范围的索引,因此页索引的低 6 位被解释为 slots 数组的下标,每个下标对应第一层的一个节点。如果基树的深度为 2,就可以表示从 0~4095 范围的索引,页索引的低 12 位分成两个 6 位的字段,高位的字段用于表示第一层节点数组的下标,而低位的字段用于表示第二层节点数组的下标。依此类推,如果深度等于 6,页索引的最高两位表示第一层节点数组的下标,接下来的 6 位表示第二层节点数组的下标,这样一直到最低 6 位,它们表示第六层节点数组的下标。

如果基树的最大索引小于应该增加的页的索引,那么内核相应地增加树的深度;基树的中间节点依赖于页索引的值(例子参见图 15-1)。

页高速缓存的处理函数

对页高速缓存操作的基本高级函数有查找、增加和删除页。在以上函数的基础上还有另一个函数确保高速缓存包含指定页的最新版本。

查找页

函数 `find_get_page()` 接收的参数为指向 `address_space` 对象的指针和偏移量。它获取地址空间的自旋锁,并调用 `radix_tree_lookup()` 函数搜索拥有指定偏移量的基树的叶子节点。该函数根据偏移量值中的位依次从树根开始并向下搜索,如上节所述。如果遇到空指针,函数返回 `NULL`;否则,返回叶子节点的地址,也就是所需要的页描述符指针。如果找到了所需要的页,`find_get_page()` 函数就增加该页的使用计数器,释放自旋锁,并返回该页的地址;否则,函数就释放自旋锁并返回 `NULL`。

函数 `find_get_pages()` 与 `find_get_page()` 类似,但它实现在高速缓存中查找一组具有相邻索引的页。它接收的参数是:指向 `address_space` 对象的指针、地址空间中相对

于搜索起始位置的偏移量、所检索到页的最大数量、指向由该函数赋值的页描述符数组的指针。`find_get_pages()` 依赖 `radix_tree_gang_lookup()` 函数实现查找操作，`radix_tree_gang_lookup()` 函数为指针数组赋值并返回找到的页数。尽管由于一些页可能不在页高速缓存中而会出现空缺的页索引，但所返回的页还是递增的索引值。

还有另外几个函数实现页高速缓存上的查找操作。例如，`find_lock_page()` 函数与 `find_get_page()` 类似，但它增加返回页的使用计数器，并调用 `lock_page()` 设置 `PG_locked` 标志，从而当函数返回时调用者能够以互斥的方式访问返回的页。随后，如果页已经被加锁，`lock_page()` 函数就阻塞当前进程。最后，它在 `PG_locked` 位置位时调用 `__wait_on_bit_lock()` 函数。后面的函数把当前进程置为 `TASK_UNINTERRUPTIBLE` 状态，把进程描述符存入等待队列，执行 `address_space` 对象的 `sync_page` 方法以取消文件所在块设备的请求队列，最后调用 `schedule()` 函数来挂起进程，直到把 `PG_locked` 标志清 0。内核使用 `unlock_page()` 函数对页进行解锁，并唤醒在等待队列上睡眠的进程。

函数 `find_trylock_page()` 与 `find_lock_page()` 类似，仅有一点不同，就是 `find_trylock_page()` 从不阻塞：如果被请求的页已经上锁，函数就返回错误码。最后要说明的是，函数 `find_or_create_page()` 执行 `find_lock_page()`；不过，如果找不到所请求的页，就分配一个新页并把它插入页高速缓存。

增加页

函数 `add_to_page_cache()` 把一个新页的描述符插入到页高速缓存。它接收的参数有：页描述符的地址 `page`、`address_space` 对象的地址 `mapping`、表示在地址空间内的页索引的值 `offset` 和为基树分配新节点时所使用的内存分配标志 `gfp_mask`。函数执行以下操作：

1. 调用 `radix_tree_preload()` 函数，它禁用内核抢占，并把一些空的 `radix_tree_node` 结构赋给每 CPU 变量 `radix_tree_preloads`。`radix_tree_node` 结构的分配由 slab 分配器高速缓存 `radix_tree_node_cachep` 来完成。如果 `radix_tree_preload()` 预分配 `radix_tree_node` 结构不成功，函数 `add_to_page_cache()` 就终止并返回错误码 `-ENOMEM`。否则，如果 `radix_tree_preload()` 成功地完成预分配，`add_to_page_cache()` 函数肯定不会因为缺乏空闲内存或因为文件的大小达到了 64GB 而无法完成新页描述符的插入。
2. 获取 `mapping->tree_lock` 自旋锁——注意，`radix_tree_preload()` 函数已经禁用了内核抢占。
3. 调用 `radix_tree_insert()` 在树中插入新节点，该函数执行下述操作：

- a. 调用 `radix_tree_maxindex()` 获得最大索引，该索引可能被插入具有当前深度的基树；如果新页的索引不能用当前深度表示，就调用 `radix_tree_extend()` 通过增加适当数量的节点来增加树的深度（例如，对图 15-1 (a) 所示的基树，`radix_tree_extend()` 在它的顶端增加一个节点）。分配新节点是通过执行 `radix_tree_node_alloc()` 函数实现的，该函数试图从 slab 分配器高速缓存获得 `radix_tree_node` 结构，如果分配失败，就从存放在 `radix_tree_preloads` 中的预分配的结构池中获得 `radix_tree_node` 结构。
 - b. 根据页索引的偏移量，从根节点 (`mapping->page_tree`) 开始遍历树，直到叶子节点，如上一节所述。如果需要，就调用 `radix_tree_node_alloc()` 分配新的中间节点。
 - c. 把页描述符地址存放在对基树所遍历的最后节点的适当位置，并返回 0。
4. 增加页描述符的使用计数器 `page->_count`。
 5. 由于页是新的，所以其内容无效：函数设置页框的 `PG_locked` 标志，以阻止其他的内核路径并发访问该页。
 6. 用 `mapping` 和 `offset` 参数初始化 `page->mapping` 和 `page->index`。
 7. 递增在地址空间所缓存页的计数器 (`mapping->npages`)。
 8. 释放地址空间的自旋锁。
 9. 调用 `radix_tree_preload_end()` 重新启用内核抢占。
 10. 返回 0 (成功)。

删除页

函数 `remove_from_page_cache()` 通过下述步骤从页高速缓存中删除页描述符：

1. 获取自旋锁 `page->mapping->tree_lock` 并关中断。
2. 调用 `radix_tree_delete()` 函数从树中删除节点。该函数接收树根的地址 (`page->mapping->page_tree`) 和要删除的页索引作为参数，并执行下述步骤：
 - a. 如上节所述，根据页索引从根节点开始遍历树，直到到达叶子节点。遍历时，建立 `radix_tree_path` 结构的数组，描述从根到与要删除的页相应的叶子节点的路径构成。
 - b. 从最后一个节点（包含指向页描述符的指针）开始，对路径数组中的节点开始循环操作。对每个节点，把指向下一个节点（或页描述符）位置数组的元素置为 `NULL`，并递减 `count` 字段。如果 `count` 变为 0，就从树中删除节点并把

`radix_tree_node`结构释放给slab分配器高速缓存。然后继续循环处理路径数组中的节点。否则，如果 `count` 不等于0，继续执行下一步。

- c. 返回已经从树中删除的页描述符指针。
3. 把 `page->mapping` 字段置为 `NULL`。
4. 把所缓存页的 `page->mapping->npages` 计数器的值减1。
5. 释放自旋锁 `page->mapping->tree_lock`，打开中断，函数终止。

更新页

函数 `read_cache_page()` 确保高速缓存中包括最新版本的指定页。它的参数是指向 `address_space` 对象的指针 `mapping`、表示所请求页的偏移量的值 `index`、指向从磁盘读页数据的函数的指针 `filler`（通常是实现地址空间 `readpage` 方法的函数）以及传递给 `filler` 函数的指针 `data`（通常为 `NULL`），下面是对这个函数的简单说明：

1. 调用函数 `find_get_page()` 检查页是否已经在页高速缓存中。
2. 如果页不在页高速缓存中，则执行下述子步骤：
 - a. 调用 `alloc_pages()` 分配一个新页框。
 - b. 调用 `add_to_page_cache()` 在页高速缓存中插入相应的页描述符。
 - c. 调用 `lru_cache_add()` 把页插入该管理区的非活动 LRU 链表中 [参见第十七章“最近最少使用的链表 (LRU)”一节]。
3. 此时，所请求的页已经在页高速缓存中了。调用 `mark_page_accessed()` 函数记录页已经被访问过的事实 [参见第十七章“最近最少使用 (LRU) 链表”一节]。
4. 如果页不是最新的 (`PG_uptodate` 标志为0)，就调用 `filler` 函数从磁盘读该页。
5. 返回页描述符的地址。

基树的标记

前面我们曾强调，页高速缓存不仅允许内核快速获得含有块设备中指定数据的页，还允许内核从高速缓存中快速获得给定状态的页。

例如，我们假设内核必须从高速缓存获得属于指定所有者的所有页和脏页（即其内容还没有写回磁盘）。存放在页描述符中的 `PG_dirty` 标志表示页是否是脏的，但是，如果绝大多数页都不是脏页，遍历整个基树以顺序访问所有叶子节点（页描述符）的操作就太慢了。

相反，为了能快速搜索脏页，基树中的每个中间节点都包含一个针对每个孩子节点（或叶子节点）的脏标记，当有且只有至少有一个孩子节点的脏标记被置位时这个标记被设置。最底层节点的脏标记通常是页描述符的 `PG_dirty` 标志的副本。通过这种方式，当内核遍历基树搜索脏页时，就可以跳过脏标记为0的中间节点的所有子树：中间节点的脏标记为0说明其子树中的所有页描述符都不是脏的。

同样的想法应用到了 `PG_writeback` 标志，该标志表示页正在被写回磁盘。这样，为基树的每个结点引入两个页描述符的标志：`PG_dirty`和`PG_writeback`（参见第八章“页描述符”一节）。每个结点的 `tags` 字段中有两个64位的数组来存放这两个标志。`tags[0]`（`PAGECACHE_TAG_DIRTY`）数组是脏标记，而 `tags[1]`（`PAGECACHE_TAG_WRITEBACK`）数组是写回标记。

设置页高速缓存中页的 `PG_dirty` 或 `PG_writeback` 标志时调用函数 `radix_tree_tag_set()`，它作用于三个参数：基树的根、页的索引以及要设置的标记的类型（`PAGECACHE_TAG_DIRTY`或`PAGECACHE_TAG_WRITEBACK`）。函数从树根开始并向下搜索到与指定索引对应的叶子结点，对于从根通往叶子路径上的每一个节点，函数利用指向路径中下一个结点的指针设置标记。然后，函数返回页描述符的地址。结果是，从根结点到叶子结点的路径中的所有结点都以适当的方式被加上了标记。

清除页高速缓存中页的 `PG_dirty` 或 `PG_writeback` 标志时调用函数 `radix_tree_tag_clear()`，它的参数与函数 `radix_tree_tag_set()` 的参数相同。函数从树根开始并向下到叶子结点，建立描述路径的 `radix_tree_path` 结构的数组。然后，函数从叶子结点到根结点向后进行操作：清除底层结点的标记，然后检查是否结点数组中所有标记都被清0，如果是，函数把上层父结点的相应标记清0，并如此继续上述操作。最后，函数返回页描述符的地址。

从基树删除页描述符时，必须更新从根结点到叶子结点的路径中结点的相应标记。函数 `radix_tree_delete()` 可以正确地完成这个工作（尽管我们在上一节没有提到这一点）。而函数 `radix_tree_insert()` 不更新标记，因为插入基树的所有页描述符的 `PG_dirty` 和 `PG_writeback` 标志都被认为是清零的。如果需要，内核可以随后调用函数 `radix_tree_tag_set()`。

函数 `radix_tree_tagged()` 利用树的所有结点的标志数组来测试基树是否至少包括一个指定状态的页。函数通过执行下面的代码轻松地完成这一任务（`root` 是指向基树的 `radix_tree_root` 结构的指针，`tag` 是要测试的标记）：

```
for (idx = 0; idx < 2; idx++) {
    if (root->rnode->tags[tag][idx])
        return 1;
}
return 0;
```

因为可能假设基树所有结点的标记都正确地更新过，所以 `radix_tree_tagged()` 函数只需要检查第一层的标记。使用该函数的一个例子是：确定一个包含脏页的索引节点是否要写回磁盘。注意，函数在每次循环时要测试在无符号长整型的32个标志中，是否有被设置的标志。

函数 `find_get_pages_tag()` 和 `find_get_pages()` 类似，只有一点不同，就是前者返回的只是那些用 `tag` 参数标记的页。正如我们将在“把脏页写入磁盘”一节所见的，该函数对快速找到一个索引节点的所有脏页是非常关键的。

把块存放在页高速缓存中

我们在第十四章“块设备的处理”一节已经看到，VFS（映射层）和各种文件系统以叫做“块”的逻辑单位组织磁盘数据。

在Linux内核的旧版本中，主要有两种不同的磁盘高速缓存：页高速缓存和缓冲区高速缓存，前者用来存放访问磁盘文件内容时生成的磁盘数据页，后者把通过VFS（管理磁盘文件系统）访问的块的内容保留在内存中。

从2.4.10的稳定版本开始，缓冲区高速缓存其实就不存在了。事实上，由于效率的原因，不再单独分配块缓冲区；相反，把它们存放在叫做“缓冲区页”的专门页中，而缓冲区页保存在页高速缓存中。

缓冲区页在形式上就是与称做“缓冲区首部”的附加描述符相关的数据页，其主要目的是快速确定页中的一个块在磁盘中的地址。实际上，页高速缓存内的页中的一大块数据在磁盘上的地址不一定是相邻的。

块缓冲区和缓冲区首部

每个块缓冲区都有 `buffer_head` 类型的缓冲区首部描述符。该描述符包含内核必须了解的、有关如何处理块的所有信息。因此，在对所有块操作之前，内核检查缓冲区首部。缓冲区首部的字段在表 15-4 中列出。

表 15-4：缓冲区首部的字段

类型	字段	说明
<code>unsigned long</code>	<code>b_state</code>	缓冲区状态标志
<code>struct buffer_head *</code>	<code>b_this_page</code>	指向缓冲区页的链表中的下一个元素的指针

表 15-4: 缓冲区首部的字段 (续)

类型	字段	说明
struct page *	b_page	指向拥有该块的缓冲区页的描述符的指针
atomic_t	b_count	块使用计数器
u32	b_size	块大小
sector_t	b_blocknr	与块设备相关的块号 (逻辑块号)
char *	b_data	块在缓冲区页内的位置
struct block_device *	b_bdev	指向块设备描述符的指针
bh_end_io_t *	b_end_io	I/O 完成方法
void *	b_private	指向 I/O 完成方法数据的指针
struct list_head	b_assoc_buffers	为与某个索引节点相关的间接块的链表提供的指针 (参见本章前面“address_space 对象”一节)

缓冲区首部的两个字段编码表示块的磁盘地址: b_bdev 字段表示包含块的块设备 (参见第十四章“块设备”一节), 通常是磁盘或分区; 而 b_blocknr 字段存放逻辑块号, 即块在磁盘或分区中的编号。

b_data 字段表示块缓冲区在缓冲区页中的位置。实际上, 这个位置的编号依赖于页是否在高端内存。如果页在高端内存, 则 b_data 字段存放的是块缓冲区相对于页的起始位置的偏移量, 否则, b_data 存放的是块缓冲区的线性地址。

b_state 字段可以存放几个标志。其中一些标志是通用的, 把它们列在表 15-5 中。每个文件系统还可以定义自己的私有缓冲区首部标志。

表 15-5: 缓冲区首部的通用标志

标志	说明
BH_Uptodate	缓冲区包含有效数据时被置位
BH_Dirty	如果缓冲区脏就置位 (表示缓冲区中的数据必须写回块设备)
BH_Lock	如果缓冲区加锁就置位, 通常发生在缓冲区进行磁盘传输时
BH_Req	如果已经为初始化缓冲区而请求数据传输就置位
BH_Mapped	如果缓冲区被映射到磁盘就置位, 即: 如果相应的缓冲区首部的 b_bdev 和 b_blocknr 是有效的就置位
BH_New	如果相应的块刚被分配而还没有被访问过就置位

表 15-5: 缓冲区首部的通用标志 (续)

标志	说明
BH_Async_Read	如果在异步地读缓冲区就置位
BH_Async_Write	如果在异步地写缓冲区就置位
BH_Delay	如果还没有在磁盘上分配缓冲区就置位
BH_Boundary	如果两个相邻的块在其中一个提交之后不再相邻就置位
BH_Write_EIO	如果写块时出现 I/O 错误就置位
BH_Ordered	如果必须严格地把块写到在它之前提交的块的后面就置位 (用于日志文件系统)
BH_Eopnotsupp	如果块设备的驱动程序不支持所请求的操作就置位

管理缓冲区首部

缓冲区首部有它们自己的 slab 分配器高速缓存, 其描述符 `kmem_cache_s` 存在变量 `bh_cachep` 中。 `alloc_buffer_head()` 和 `free_buffer_head()` 函数分别用于获取和释放缓冲区首部。

缓冲区首部的 `b_count` 字段是相应的块缓冲区的引用计数器。在每次对块缓冲区进行操作之前递增计数器并在操作之后递减它。除了周期性地检查保存在页高速缓存中的块缓冲区之外, 当空闲内存变得很少时也要对它进行检查, 只有引用计数器等于 0 的块缓冲区才可以被回收 (参见第十七章)。

当内核控制路径希望访问块缓冲区时, 应该先递增引用计数器。确定块在页高速缓存中的位置的函数 (`__getblk()`, 参见本章稍后“在页高速缓存中搜索块”一节) 自动完成这项工作, 因此, 高层函数通常不增加块缓冲区的引用计数器。

当内核控制路径停止访问块缓冲区时, 应该调用 `__brelse()` 或 `__bforget()` 递减相应的引用计数器。这两个函数之间的不同是 `__bforget()` 还从间接块链表 (缓冲区首部的 `b_assoc_buffers` 字段, 参见前面的“块缓冲区和缓冲区首部”一节) 中删除块, 并把该缓冲区标记为干净的, 因此强制内核忽略对缓冲区所做的任何修改, 但实际上缓冲区依然必须被写回磁盘。

缓冲区页

只要内核必须单独地访问一个块, 就要涉及存放块缓冲区的缓冲区页, 并检查相应的缓冲区首部。

下面是内核创建缓冲区页的两种普通情况：

- 当读或写的文件页在磁盘块中不相邻时。发生这种情况是因为文件系统为文件分配了非连续的块，或因为文件有“洞”（参见第十八章“文件的洞”一节）。
- 当访问一个单独的磁盘块时（例如，当读超级块或索引节点块时）。

在第一种情况下，把缓冲区页的描述符插入普通文件的基树；保存好缓冲区首部，因为其中存有重要的信息，即存有数据在磁盘中位置的块设备和逻辑块号。在第十六章我们将了解内核如何利用这种类型的缓冲区页。

在第二种情况下，把缓冲区页的描述符插入基树，树根是与块设备相关的特殊 *bdev* 文件系统中索引节点的 *address_space* 对象（参见本章前面“*address_space* 对象”一节）。这种缓冲区页必须满足很强的约束条件，就是所有的块缓冲区涉及的块必须是在块设备上相邻存放的。

这种情况的一个应用实例是：如果虚拟文件系统要读大小为 1024 个字节的索引节点块（包含给定文件的索引节点）。内核并不是只分配一个单独的缓冲区，而是必须分配一个整页，从而存放四个缓冲区；这些缓冲区将存放块设备上相邻的 4 块数据，其中包括所请求的索引节点块。

本章我们将重点讨论第二种类型的缓冲区页，即所谓的块设备缓冲区页（有时简称为块设备页）。

在一个缓冲区页内的所有块缓冲区大小必须相同，因此，在 80x86 体系结构上，根据块的大小，一个缓冲区页可以包括 1~8 个缓冲区。

如果一个页作为缓冲区页使用，那么与它的块缓冲区相关的所有缓冲区首部都被收集在一个单向循环链表中。缓冲区页描述符的 *private* 字段指向页中第一个块的缓冲区首部（注 3），每个缓冲区首部存放在 *b_this_page* 字段中，该字段是指向链表中下一个缓冲区首部的指针。此外，每个缓冲区首部还把缓冲区页描述符的地址存放在 *b_page* 字段中。图 15-2 显示了一个缓冲区页，其中包含四个块缓冲区和对应的缓冲区首部。

注 3： 由于 *private* 字段包含有效数据，而且页的 *PG_private* 标志被设置，因此，如果页中包含磁盘数据并且设置了 *PG_private* 标志，该页就是一个缓冲区页。注意，尽管如此，其他与块 I/O 子系统无关的内核组件也因为别的用途而使用 *private* 和 *PG_private* 字段。

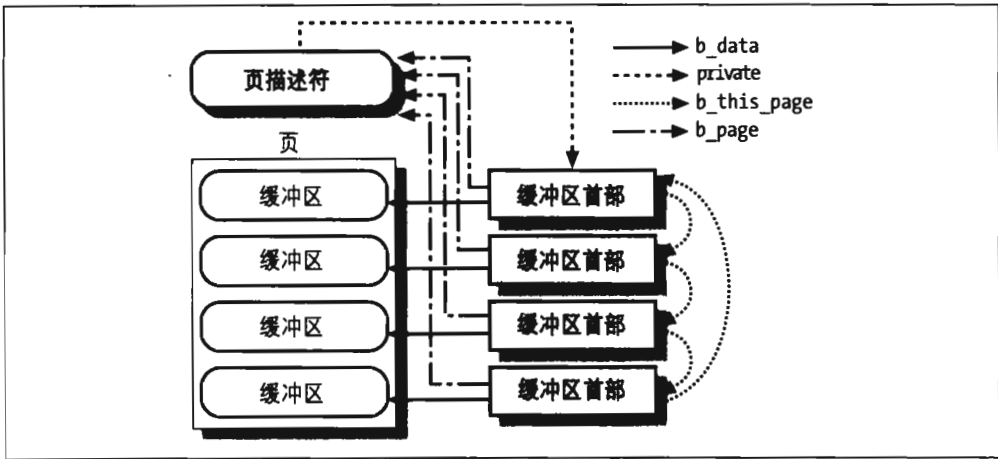


图 15-2：一个缓冲区页，其中包含四个块缓冲区和对应的缓冲区首部

分配块设备缓冲区页

当内核发现指定块的缓冲区所在的页不在页高速缓存中时，就分配一个新的块设备缓冲区页（参见本章稍后“在页高速缓存中搜索块”一节）。特别是，对块的查找操作会由于下述原因而失败：

1. 包含数据块的页不在块设备的基树中：这种情况下，必须把新页的描述符加到基树中。
2. 包含数据块的页在块设备的基树中，但这个页不是缓冲区页：在这种情况下，必须分配新的缓冲区首部，并将它链接到所属的页，从而把它变成块设备缓冲区页。
3. 包含数据块的缓冲区页在块设备的基树中，但页中块的大小与所请求的块大小不相同：这种情况下，必须释放旧的缓冲区首部，分配经过重新赋值的缓冲区首部并将它链接到所属的页。

内核调用函数 `grow_buffers()` 把块设备缓冲区页添加到页高速缓存中，该函数接收三个标识块的参数：

- `block_device` 描述符的地址 `bdev`。
- 逻辑块号 `block`（块在块设备中的位置）。
- 块大小 `size`。

该函数本质上执行下列操作：

1. 计算数据页在所请求块的块设备中的偏移量 `index`。
2. 如果需要，就调用 `grow_dev_page()` 创建新的块设备缓冲区页。该函数依次执行下列子步骤：
 - a. 调用函数 `find_or_create_page()`，传递给它的参数有：块设备的 `address_space` 对象(`bdev->bd_inode->i_mapping`)、页偏移 `index` 以及 `GFP_NOFS` 标志。正如在前面“页高速缓存的处理函数”一节所描述的，`find_or_create_page()` 在页高速缓存中搜索需要的页，如果需要，就把新页插入高速缓存。
 - b. 此时，所请求的页已经在页高速缓存中，而且函数获得了它的描述符地址。函数检查它的 `PG_private` 标志，如果为空，说明页还不是一个缓冲区页（没有相关的缓冲区首部），就跳到第 2e 步。
 - c. 页已经是缓冲区页。从页描述符的 `private` 字段获得第一个缓冲区首部的地址 `bh`，并检查块大小 `bh->size` 是否等于所请求的块大小；如果大小相等，在页高速缓存中找到的页就是有效的缓冲区页，因此跳到第 2g 步。
 - d. 如果页中块的大小有错误，就调用 `try_to_free_buffers()`（参见下一节）释放缓冲区页的上一个缓冲区首部。
 - e. 调用函数 `alloc_page_buffers()` 根据页中所请求的块大小分配缓冲区首部，并把它们插入由 `b_this_page` 字段实现的单向循环链表。此外，函数用页描述符的地址初始化缓冲区首部的 `b_page` 字段，用块缓冲区在页内的线性地址或偏移量初始化 `b_data` 字段。
 - f. 在字段 `private` 中存放第一个缓冲区首部的地址，把 `PG_private` 字段置位，并递增页的使用计数器（页中的块缓冲区被算作一个页用户）。
 - g. 调用 `init_page_buffers()` 函数初始化连接到页的缓冲区首部的字段 `b_bdev`、`b_blocknr` 和 `b_bstate`。因为所有的块在磁盘上都是相邻的，因此逻辑块号是连续的，而且很容易从块得出。
 - h. 返回页描述符地址。
3. 为页解锁（函数 `find_or_create_page()` 曾为页加了锁）。
4. 递减页的使用计数器（函数 `find_or_create_page()` 曾递增了计数器）。
5. 返回 1（成功）。

释放块设备缓冲区页

就像我们在第十七章将要了解的那样，当内核试图获得更多的空闲内存时，就释放块设

备缓冲区页。显然，不可能释放有脏缓冲区或上锁的缓冲区的页。内核调用函数 `try_to_release_page()` 释放缓冲区页，该函数接收页描述符的地址 `page`，并执行下述步骤（注4）：

1. 如果设置了页的 `PG_writeback` 标志，则返回 0（因为正在把页写回磁盘，所以不可能释放该页）。
2. 如果已经定义了块设备 `address_space` 对象的 `releasepage` 方法，就调用它（通常没有为块设备定义的 `releasepage` 方法）。
3. 调用函数 `try_to_free_buffers()` 并返回它的错误代码。

函数 `try_to_free_buffers()` 依次扫描链接到缓冲区页的缓冲区首部，它本质上执行下列操作：

1. 检查页中所有缓冲区的缓冲区首部的标志。如果有些缓冲区首部的 `BH_Dirty` 或 `BH_Locked` 标志被置位，说明函数不可能释放这些缓冲区，所以函数终止并返回 0（失败）。
2. 如果缓冲区首部在间接缓冲区的链表中（参见本章前面“块缓冲区和缓冲区首部”一节），该函数就从链表中删除它。
3. 清除页描述符的 `PG_private` 标记，把 `private` 字段设置为 `NULL`，并递减页的使用计数器。
4. 清除页的 `PG_dirty` 标记。
5. 反复调用 `free_buffer_head()`，以释放页的所有缓冲区首部。
6. 返回 1（成功）。

在页高速缓存中搜索块

当内核需要读或写一个单独的物理设备块时（例如一个超级块），必须检查所请求的块缓冲区是否已经在页高速缓存中。在页高速缓存中搜索指定的块缓冲区（由块设备描述符的地址 `bdev` 和逻辑块号 `nr` 表示）的过程分成三个步骤：

1. 获取一个指针，让它指向包含指定块的块设备的 `address_space` 对象（`bdev->bd_inode->i_mapping`）。

注4： 还可以对普通文件所拥有的缓冲区页调用 `try-to-release-page()` 函数。

2. 获得设备的块大小 (`bdev->bd_block_size`), 并计算包含指定块的页索引。这需要在逻辑块号上进行位移操作。例如, 如果块的大小是 1024 字节, 每个缓冲区页包含四个块缓冲区, 那么页的索引是 `nr/4`。
3. 在块设备的基树中搜索缓冲区页。获得页描述符之后, 内核访问缓冲区首部, 它描述了页中块缓冲区的状态。

不过, 实现的细节要更为复杂。为了提高系统性能, 内核维持一个小磁盘高速缓存数组 `bh_lrus` (每个 CPU 对应一个数组元素), 即所谓的最近最少使用 (LRU) 块高速缓存。每个磁盘高速缓存有 8 个指针, 指向被指定 CPU 最近访问过的缓冲区首部。对每个 CPU 数组的元素排序, 使指向最后被使用过的那个缓冲区首部的指针索引为 0。相同的缓冲区首部可能出现在几个 CPU 数组中 (但是同一个 CPU 数组中不会有相同的缓冲区首部)。在 LRU 块高速缓存中每出现一次缓冲区首部, 该缓冲区首部的使用计数器 `b_count` 就加 1。

__find_get_block() 函数

函数 `__find_get_block()` 的参数有: `block_device` 描述符地址 `bdev`、块号 `block` 和块大小 `size`。函数返回页高速缓存中的块缓冲区对应的缓冲区首部的地址; 如果不存在指定的块, 就返回 `NULL`。该函数本质上执行下面的操作:

1. 检查执行 CPU 的 LRU 块高速缓存数组中是否有一个缓冲区首部, 其 `b_bdev`、`b_blocknr` 和 `b_size` 字段分别等于 `bdev`、`block` 和 `size`。
2. 如果缓冲区首部在 LRU 块高速缓存中, 就刷新数组中的元素, 以便让指针指在第一个位置 (索引为 0) 刚找到的缓冲区首部, 递增它的 `b_count` 字段, 并跳转到第 8 步。
3. 如果缓冲区首部不在 LRU 块高速缓存中, 根据块号和块大小得到与块设备相关的页的索引:

```
index = block >> (PAGE_SHIFT - bdev->bd_inode->i_blkbits)
```

4. 调用 `find_get_page()` 确定存有所请求的块缓冲区的缓冲区页的描述符在页高速缓存中的位置。该函数传递的参数有: 指向块设备的 `address_space` 对象的指针 (`bdev->bd_inode->i_mapping`) 和页索引。页索引用于确定存有所请求的块缓冲区的缓冲区页的描述符在页高速缓存中的位置。如果高速缓存中没有这样的页, 就返回 `NULL` (失败)。
5. 此时, 函数已经得到了缓冲区页描述符的地址: 它扫描链接到缓冲区页的缓冲区首部链表, 查找逻辑块号等于 `block` 的块。

6. 递减页描述符的 `count` 字段 (`find_get_page()` 曾递增它的值)。
7. 把 LRU 块高速缓存中的所有元素向下移动一个位置, 并把指向所请求块的缓冲区首部的指针插入到第一个位置。如果一个缓冲区首部已经不在 LRU 块高速缓存中, 就递减它的引用计数器 `b_count`。
8. 如果需要, 就调用 `mark_page_accessed()` 把缓冲区页移至适当的 LRU 链表中 [参见第十七章“最近最少使用 (LRU) 链表”一节]。
9. 返回缓冲区首部指针。

`__getblk()` 函数

函数 `__getblk()` 与 `__find_get_block()` 接收相同的参数, 也就是 `block_device` 描述符的地址 `bdev`、块号 `block` 和块大小 `size`, 并返回与缓冲区对应的缓冲区首部的地址。即使块根本不存在, 该函数也不会失败, `__getblk()` 友好地分配块设备缓冲区页并返回将要描述块的缓冲区首部的指针。注意, `__getblk()` 返回的块缓冲区不必存有有效数据——缓冲区首部的 `BH_Uptodate` 标志可能被清 0。

函数 `__getblk()` 本质上执行下面的步骤:

1. 调用 `__find_get_block()` 检查块是否已经在页高速缓存中。如果找到块, 则函数返回其缓冲区首部的地址。
2. 否则, 调用 `grow_buffers()` 为所请求的页分配一个新的缓冲区页 (参见本章前面“分配块设备缓冲区页”一节)。
3. 如果 `grow_buffers()` 分配这样的页时失败, `__getblk()` 试图通过调用函数 `free_more_memory()` 回收一部分内存 (参见第十七章)。
4. 跳转到第 1 步。

`__bread()` 函数

函数 `__bread()` 接收与 `__getblk()` 相同的参数, 即 `block_device` 描述符的地址 `bdev`、块号 `block` 和块大小 `size`, 并返回与缓冲区对应的缓冲区首部的地址。与 `__getblk()` 相反的是, 如果需要的话, 在返回缓冲区首部之前函数 `__bread()` 从磁盘读块。函数 `__bread()` 执行下述步骤:

1. 调用 `__getblk()` 在页高速缓存中查找与所请求的块相关的缓冲区页, 并获得指向相应的缓冲区首部的指针。
2. 如果块已经在页高速缓存中并包含有效数据 (`BH_Uptodate` 标志被置位), 就返回缓冲区首部的地址。

3. 否则，递增缓冲区首部的引用计数器。
4. 把 `end_buffer_read_sync()` 的地址赋给 `b_end_io` 字段（参见下一节）。
5. 调用 `submit_bh()` 把缓冲区首部传送到通用块层（参见下一节）。
6. 调用 `wait_on_buffer()` 把当前进程插入等待队列，直到 I/O 操作完成，即直到缓冲区首部的 `BH_Lock` 标志被清 0。
7. 返回缓冲区首部的地址。

向通用块层提交缓冲区首部

一对 `submit_bh()` 和 `ll_rw_block()` 函数，允许内核对缓冲区首部描述的一个或多个缓冲区进行 I/O 数据传送。

`submit_bh()` 函数

内核利用 `submit_bh()` 函数向通用块层传递一个缓冲区首部，并由此请求传输一个数据块。它的参数是数据传输的方向（本质上就是 `READ` 或 `WRITE`）和指向描述块缓冲区的缓冲区首部的指针 `bh`。

`submit_bh()` 函数假设缓冲区首部已经被彻底初始化，尤其是，必须正确地为 `b_bdev`、`b_blocknr` 和 `b_size` 字段赋值以标识包含所请求数据的磁盘上的块。如果块缓冲区在块设备缓冲区页中，就由 `__find_get_block()` 完成对缓冲区首部的初始化，就像在上一节所描述的。不过，我们将在下一章看到，还可以对普通文件所有的缓冲区页中的块调用 `submit_bh()`。

`submit_bh()` 函数只是一个起连接作用的函数，它根据缓冲区首部的内容创建一个 `bio` 请求，并随后调用 `generic_make_request()`（参见第十四章“提交请求”一节）。函数执行的主要步骤如下：

1. 设置缓冲区首部的 `BH_Req` 标志以表示块至少被访问过一次。此外，如果数据传输的方向是 `WRITE`，就将 `BH_Write_EIO` 标志清 0。
2. 调用 `bio_alloc()` 分配一个新的 `bio` 描述符（参见第十四章“`bio` 结构”一节）。
3. 根据缓冲区首部的内容初始化 `bio` 描述符的字段：
 - a. 把块中的第一个扇区的号 (`bh->b_blocknr*bh->b_size/512`) 赋给 `bi_sector` 字段。
 - b. 把块设备描述符的地址 (`bh->b_bdev`) 赋给 `bi_bdev` 字段。

- c. 把块大小 (`bh->b_size`) 赋给 `bi_size` 字段。
 - d. 初始化 `bi_io_vec` 数组的第一个元素 以使该段对应于块缓冲区:把 `bh->b_page` 赋给 `bi_io_vec[0].bv_page`, 把 `bh->b_size` 赋给 `bi_io_vec[0].bv_len`, 并把块缓冲区在页中的偏移量 `bh->b_data` 赋给 `bi_io_vec[0].bv_offset`。
 - e. 把 `bi_vcnt` 置为 1 (只有一个涉及 bio 的段), 并把 `bi_idx` 置为 0 (将要传输的是当前段)。
 - f. 把 `end_bio_bh_io_sync()` 的地址赋给 `bi_end_io` 字段, 并把缓冲区首部的地址赋给 `bi_private` 字段; 数据传输结束时调用函数 (见下面)。
4. 递增 bio 的引用计数器 (它变为 2)。
 5. 调用 `submit_bio()`, 把 `bi_rw` 标志设置为数据传输的方向, 更新每 CPU 变量 `page_states` 以表示读和写的扇区数, 并对 bio 描述符调用 `generic_make_request()` 函数。
 6. 递减 bio 的使用计数器; 因为 bio 描述符现在已经被插入 I/O 调度程序的队列, 所以没有释放 bio 描述符。
 7. 返回 0 (成功)。

当针对 bio 上的 I/O 数据传输终止的时候, 内核执行 `bi_end_io` 方法, 具体来说执行 `end_bio_bh_io_sync()` 函数。后者本质上从 bio 的 `bi_private` 字段获取缓冲区首部的地址, 然后调用缓冲区首部 (在调用 `submit_bh()` 之前已为它正确赋值) 的方法 `b_end_io`, 最后调用 `bio_put()` 释放 bio 结构。

ll_rw_block()函数

有些时候内核必须立刻触发几个数据块的数据传输, 这些数据块不一定物理上相邻。`ll_rw_block()` 函数接收的参数有数据传输的方向 (本质上就是 READ 或 WRITE)、要传输的数据块的块号以及指向块缓冲区所对应的缓冲区首部的指针数组。该函数在所有缓冲区首部上进行循环, 每次循环执行下面的操作:

1. 检查并设置缓冲区首部的 `BH_Lock` 标志; 如果缓冲区已经被锁住, 而另外一个内核控制路径已经激活了数据传输, 就不处理这个缓冲区, 而跳转到第 9 步。
2. 把缓冲区首部的使用计数器 `b_count` 加 1。
3. 如果数据传输的方向是 `WRITE`, 就让缓冲区首部的 `b_end_io` 指向函数 `end_buffer_write_sync()` 的地址, 否则让 `b_end_io` 指向 `end_buffer_read_sync()` 函数的地址。

4. 如果数据传输的方向是 WRITE, 就检查并清除缓冲区首部的 BH_Dirty 标志。如果该标志没有置位, 就不必把块写入磁盘, 因此跳转到第 7 步。
5. 如果数据传输的方向是 READ 或 READA(向前读), 检查缓冲区首部的 BH_Uptodate 标志是否被置位; 如果是, 就不必从磁盘读块, 因此跳转到第 7 步。
6. 此时必须读或写数据块: 调用 submit_bh() 函数把缓冲区首部传递到通用块层, 然后跳转到第 9 步。
7. 通过清除 BH_Lock 标志为缓冲区首部解锁, 然后唤醒所有等待块解锁的进程。
8. 递减缓冲区首部的 b_count 字段。
9. 如果数组中还有其他的缓冲区首部要处理, 就选择下一个缓冲区首部并跳转回到第 1 步, 否则, 就结束。

注意, 如果函数 ll_rw_block() 把缓冲区首部传递到通用块层, 而留下加了锁的缓冲区和增加了的引用计数器, 这样, 在完成数据传输之前就不可能访问该缓冲区, 也不可能释放这个缓冲区。当块的数据传送结束时, 内核执行缓冲区首部的 b_end_io 方法。假设没有 I/O 错误, end_buffer_write_sync() 和 end_buffer_read_sync() 函数只是简单地把缓冲区首部的 BH_Uptodate 字段置位, 为缓冲区解锁, 并递减它的引用计数器。

把脏页写入磁盘

正如我们所了解的, 内核不断用包含块设备数据的页填充页高速缓存。只要进程修改了数据, 相应的页就被标记为脏页, 即把它的 PG_dirty 标志置位。

Unix 系统允许把脏缓冲区写入块设备的操作延迟执行, 因为这种策略可以显著地提高系统的性能。对高速缓存中的页的几次写操作可能只需对相应的磁盘块进行一次缓慢的物理更新就可以满足。此外, 写操作没有读操作那么紧迫, 因为进程通常是不会由于延迟写而挂起, 而大部分情况都因为延迟读而挂起。正是由于延迟写, 使得任一物理块设备平均为读请求提供的服务将多于写请求。

一个脏页可能直到最后一刻(即直到系统关闭时)都一直逗留在主存中。然而, 从延迟写策略的局限性来看, 它有两个主要的缺点:

- 如果发生了硬件错误或电源掉电的情况, 那么就无法再获得 RAM 的内容, 因此, 从系统启动以来对文件进行的很多修改就丢失了。
- 页高速缓存的大小(由此存放它所需的 RAM 的大小)就可能要很大——至少要与所访问块设备的大小相同。

因此，在下列条件下把脏页刷新（写入）到磁盘：

- 页高速缓存变得太满，但还需要更多的页，或者脏页的数量已经太多。
- 自从页变成脏页以来已过去太长时间。
- 进程请求对块设备或者特定文件任何待定的变化都进行刷新。通过调用 `sync()`、`fsync()` 或 `fdatasync()` 系统调用来实现（参见本章稍后“`sync()`、`fsync()`和 `fdatasync()`系统调用”一节）。

缓冲区页的引入使问题更加复杂。与每个缓冲区页相关的缓冲区首部使内核能够了解每个独立块缓冲区的状态。如果至少有一个缓冲区首部的 `BH_Dirty` 标志被置位，就应该设置相应缓冲区页的 `PG_dirty` 标志。当内核选择要刷新的缓冲区页时，它扫描相应的缓冲区首部，并只把脏块的内容有效地写到磁盘。一旦内核把缓冲区的所有脏页刷新到磁盘，就把页的 `PG_dirty` 标记清 0。

pdflush 内核线程

早期版本的 Linux 使用 `bdflush` 内核线程系统地扫描页高速缓存以搜索要刷新的脏页，并且使用另一个内核线程 `kupdate` 来保证所有的页不会“脏”太长的时间。Linux 2.6 用一组通用内核线程 `pdflush` 代替上述两个线程。

这些内核线程结构灵活，它们作用于两个参数：一个指向线程要执行的函数的指针和一个函数要用的参数。系统中 `pdflush` 内核线程的数量是要动态调整的：`pdflush` 线程太少时就创建，太多时就杀死。因为这些内核线程所执行的函数可以阻塞，所以创建多个而不是一个 `pdflush` 内核线程可以改善系统性能。

根据下面的原则控制 `pdflush` 线程的产生和消亡：

- 必须有至少两个，最多八个 `pdflush` 内核线程。
- 如果到最近的 1s 期间没有空闲 `pdflush`，就应该创建新的 `pdflush`。
- 如果最近一次 `pdflush` 变为空闲的时间超过了 1s，就应该删除一个 `pdflush`。

所有的 `pdflush` 内核线程都有 `pdflush_work` 描述符（如表 15-6 所示）。空闲 `pdflush` 内核线程的描述符都集中在 `pdflush_list` 链表中；在多处理器系统中，`pdflush_lock` 自旋锁保护该链表不会被并发访问。`nr_pdflush_threads` 变量（注 5）存放 `pdflush` 内核线程（空闲的或忙的）的总数。最后，`last_empty_jifs` 变量存放 `pdflush` 线程的 `pdflush_list` 链表变为空的时间（以 `jiffies` 表示）。

注 5： 可以从文件 `/proc/sys/vm/nr_pdflush_threads` 中读出这个变量的值。

表 15-6: `pdflush_work` 描述符的字段

类型	字段	说明
<code>struct task_struct</code>	<code>who</code>	指向内核线程描述符的指针
<code>void(*) (unsigned long)</code>	<code>fn</code>	内核线程所执行的回调函数
<code>unsigned long</code>	<code>arg0</code>	给回调函数的参数
<code>struct list head</code>	<code>list</code>	<code>pdflush_list</code> 链表的链接
<code>unsigned long</code>	<code>when_i_went_</code> <code>to_sleep</code>	当内核线程可用时的时间(以jiffies表示)

所有 `pdflush` 内核线程都执行函数 `__pdflush()`，它本质上循环执行一直到内核线程死亡。我们不妨假设 `pdflush` 内核线程是空闲的，而进程正在 `TASK_INTERRUPTIBLE` 状态睡眠。一旦内核线程被唤醒，`__pdflush()` 就访问其 `pdflush_work` 描述符，并执行字段 `fn` 中的回调函数，把 `arg0` 字段中的参数传递给该函数。当回调函数结束时，`__pdflush()` 检查 `last_empty_jifs` 变量的值：如果不存在空闲 `pdflush` 内核线程的时间已经超过 1s，而且 `pdflush` 内核线程的数量不到 8 个，函数 `__pdflush()` 就创建另外一个内核线程。相反，如果 `pdflush_list` 链表中的最后一项对应的 `pdflush` 内核线程空闲时间超过了 1s，而且系统中有两个以上的 `pdflush` 内核线程，函数 `__pdflush()` 就终止：就像在第三章“内核线程”一节所描述的，相应的内核线程执行 `_exit()` 系统调用，并因此而被撤消。否则，如果系统中 `pdflush` 内核线程不多于两个，`__pdflush()` 就把内核线程的 `pdflush_work` 描述符重新插入到 `pdflush_list` 链表中，并使内核线程睡眠。

`pdflush_operation()` 函数用来激活空闲的 `pdflush` 内核线程。该函数作用于两个参数：一个指针 `fn`，指向必须执行的函数；以及参数 `arg0`。函数执行下面的步骤：

1. 从 `pdflush_list` 链表中获取 `pdf` 指针，它指向空闲 `pdflush` 内核线程的 `pdflush_work` 描述符。如果链表为空，就返回 -1。如果链表中仅剩一个元素，就把 `jiffies` 的值赋给变量 `last_empty_jifs`。
2. 把参数 `fn` 和 `arg0` 分别赋给 `pdf->fn` 和 `pdf->arg0`。
3. 调用 `wake_up_process()` 唤醒空闲的 `pdflush` 内核线程，即 `pdf->who`。

把哪些工作委托给 `Pdflush` 内核线程来完成呢？其中一些工作与脏数据的刷新相关。尤其是，`pdflush` 通常执行下面的回调函数之一：

- `background_writeout()`：系统地扫描页高速缓存以搜索要刷新的脏页(参见下一节“搜索要刷新的脏页”)。

- `wb_kupdate()`：检查页高速缓存中是否有“脏”了很长时间的页（参见本章稍后“回写陈旧的脏页”一节）。

搜索要刷新的脏页

所有的基树都可能要有要刷新的脏页。为了得到所有这些页，就要彻底搜索与在磁盘上有映像的索引节点相应的所有 `address_space` 对象。由于页高速缓存可能有大量的页，如果用一个单独的执行流来扫描整个高速缓存，会令 CPU 和磁盘长时间繁忙。因此，Linux 使用一种复杂的机制把对页高速缓存的扫描划分为几个执行流。

`wakeup_bdflush()` 函数接收页高速缓存中应该刷新的脏页数量作为参数，0 值表示高速缓存中的所有脏页都应该写回磁盘。该函数调用 `pdflush_operation()` 唤醒 `pdflush` 内核线程（参见上一节），并委托它执行回调函数 `background_writeout()`，后者有效地从页高速缓存获得指定数量的脏页，并把它们写回磁盘。

当内存不足或用户显式地请求刷新操作时执行 `wakeup_bdflush()` 函数。特别是在下述情况下会调用该函数：

- 用户态进程发出 `sync()` 系统调用（参见本章稍后“`sync()`、`fsync()`和 `fdatasync()` 系统调用”一节）
- `grow_buffers()` 函数分配一个新缓冲区页时失败（参见前面“分配块设备缓冲区页”一节）
- 页框回收算法调用 `free_more_memory()` 或 `try_to_free_pages()`（参见第十七章）
- `mempool_alloc()` 函数分配一个新的内存池元素时失败（参见第八章“内存池”一节）

此外，执行 `background_writeout()` 回调函数的 `pdflush` 内核线程是由满足以下两个条件的进程唤醒的：一是对页高速缓存中的页内容进行了修改，二是引起脏页部分增加到超过某个脏背景阈值（*background threshold*）。背景阈值通常设置为系统中所有页的 10%，不过可以通过修改文件 `/proc/sys/vm/dirty_background_ratio` 来调整这个值。

`background_writeout()` 函数依赖于作为双向通信设备的 `writeback_control` 结构：一方面，它告诉辅助函数 `writeback_inodes()` 要做什么；另一方面，它保存写回磁盘的页的数量的统计值。下面是这个结构最重要的字段：

`sync_mode`

表示同步模式：`WB_SYNC_ALL` 表示如果遇到一个上锁的索引节点，必须等待而不能略过它；`WB_SYNC_HOLD` 表示把上锁的索引节点放入稍后涉及的链表中；`WB_SYNC_NONE` 表示简单地略过上锁的索引节点。

bdi

如果不为空，就指向 `backing_dev_info` 结构。此时，只有属于基本块设备的脏页将会被刷新。

older_than_this

如果不为空，就表示应该略过比指定值还新的索引节点。

nr_to_write

当前执行流中仍然要写的脏页的数量。

nonblocking

如果这个标志被置位，就不能阻塞进程。

`background_writeout()` 函数只作用于一个参数 `nr_pages`，表示应该刷新到磁盘的最少页数。它本质上执行下述步骤：

1. 从每CPU变量 `page_state` 中读当前页高速缓存中页和脏页的数量。如果脏页所占的比例低于给定的阈值，而且已经至少有 `nr_pages` 页被刷新到磁盘，该函数就终止。这个阈值通常大约是系统中总页数的 40%，可以通过写文件 `/proc/sys/vm/dirty_ratio` 来调整这个值。
2. 调用 `writeback_inodes()` 尝试写 1024 个脏页（见下面）。
3. 检查有效写过的页的数量，并减少需要写的页的个数。
4. 如果已经写过的页少于 1024 页，或略过了一些页，则可能块设备的请求队列处于拥塞状态：此时，`background_writeout()` 函数使当前进程在特定的等待队列上睡眠 100ms，或使当前进程睡眠到队列变得不拥塞。
5. 返回到第 1 步。

`writeback_inodes()` 函数只作用于一个参数，就是指针 `wbc`，它指向 `writeback_control` 描述符。该描述符的 `nr_to_write` 字段存有要刷新到磁盘的页数。函数返回时，该字段存有要刷新到磁盘的剩余页数，如果一切顺利，则该字段的值被赋为 0。

我们假设 `writeback_inodes()` 函数被调用的条件为：指针 `wbc->bdi` 和 `wbc->older_than_this` 被置为 `NULL`，`WB_SYNC_NONE` 同步模式和 `wbc->nonblocking` 标志置位（这些值都由 `background_writeout()` 函数设置）。函数 `writeback_inodes()` 扫描在 `super_blocks` 变量中建立的超级块链表（参见第十二章“超级块对象”一节）。当遍历完整个链表或刷新的页数达到预期数量时，就停止扫描。对每个超级块 `sb`，函数执行下述步骤：

1. 检查 `sb->s_dirty` 或 `sb->s_io` 链表是否为空：第一个链表集中了超级块的脏索引节点，而第二个链表集中了等待被传输到磁盘的索引节点（见下面）。如果两个链表都为空，说明相应文件系统的索引节点没有脏页，因此函数处理链表中的下一个超级块。
2. 此时，超级块有脏索引节点。对超级块 `sb` 调用 `sync_sb_inodes()`，该函数执行下面的操作：
 - a. 把 `sb->s_dirty` 的所有索引节点插入 `sb->s_io` 指向的链表，并清空脏索引节点链表。
 - b. 从 `sb->s_io` 获得下一个索引节点的指针。如果该链表为空，就返回。
 - c. 如果 `sync_sb_inodes()` 函数开始执行后，索引节点变为脏节点，就略过这个索引节点的脏页并返回。注意，`sb->s_io` 链表中可能残留一些脏索引节点。
 - d. 如果当前进程是 `pdflush` 内核线程，`sync_sb_inodes()` 就检查运行在另一个 CPU 上的 `pdflush` 内核线程是否已经试图刷新这个块设备文件的脏页。这是通过一个原子测试和对索引节点的 `backing_dev_info` 的 `BDI_pdflush` 标志的设置操作来完成的。本质上，它对同一个请求队列上有多个 `pdflush` 内核线程是毫无意义的（参见本章前面“`pdflush` 内核线程”一节）。
 - e. 把索引节点的引用计数器加 1。
 - f. 调用 `__writeback_single_inode()` 回写与所选择的索引节点相关的脏缓冲区：
 - (1) 如果索引节点被锁定，就把它移到脏索引节点链表中 (`inode->i_sb->s_dirty`) 并返回 0。（因为我们假定 `wbc->sync_mode` 字段不等于 `WB_SYNC_ALL`，所以函数不会因为等待索引节点解锁而阻塞。）
 - (2) 使用索引节点地址空间的 `writepages` 方法，或者在没有这个方法的情况下使用 `mpage_writepages()` 函数来写 `wbc->nr_to_write` 个脏页。该函数调用 `find_get_pages_tag()` 函数快速获得索引节点地址空间的所有脏页（参见本章前面“基树的标记”一节），细节将在下一章描述。
 - (3) 如果索引节点是脏的，就用超级块的 `write_inode` 方法把索引节点写到磁盘。实现该方法的函数通常依靠 `submit_bh()` 来传输一个数据块（参见本章前面“向通用块层提交缓冲区首部”一节）。
 - (4) 检查索引节点的状态。如果索引节点还有脏页，就把索引节点移回 `sb->s_dirty` 链表；如果索引节点引用计数器为 0，就把索引节点移到 `inode_unused` 链表中，否则就把索引节点移到 `inode_in_use` 链表中。（参见第十二章“索引节点对象”一节）。

- (5) 返回在第 2f(2)步所调用的函数的错误代码。
- g. 回到 `sync_sb_inodes()` 函数中。如果当前进程是 *pdflush* 内核线程，就把在第 2d 步设置的 `BDI_pdflush` 标志清 0。
 - h. 如果略过了刚处理的索引节点中的一些页，那么该索引节点包括锁定的缓冲区：把 `sb->s_io` 链表中的所有剩余索引节点移回到 `sb->s_dirty` 链表中，以后将重新处理它们。
 - i. 把索引节点的引用计数器减 1。
 - j. 如果 `wbc->nr_to_write` 大于 0，则回到第 2b 步搜索同一个超级块的其他脏索引节点。否则，`sync_sb_inodes()` 函数终止。
3. 回到 `writeback_inodes()` 函数中。如果 `wbc->nr_to_write` 大于 0，就跳转到第 1 步，并继续处理全局链表中的下一个超级块。否则，就返回。

回写陈旧的脏页

如前所述，内核试图避免当一些页很久没有被刷新时发生饥饿危险。因此，脏页在保留一定时间后，内核就显式地开始进行 I/O 数据的传输，把脏页的内容写到磁盘。

回写陈旧脏页的工作委托给了被定期唤醒的 *pdflush* 内核线程。在内核初始化期间，`page_writeback_init()` 函数建立 `wb_timer` 动态定时器，以便定时器的到期时间发生在 `dirty_writeback_centisecs` 文件中所规定的几百分之一秒之后（通常是 500 分之一秒，不过可以通过修改 `/proc/sys/vm/dirty_writeback_centisecs` 文件调整这个值）。定时器函数 `wb_timer_fn()` 本质上调用 `pdflush_operation()` 函数，传递给它的参数是回调函数 `wb_kupdate()` 的地址。

`wb_kupdate()` 函数遍历页高速缓存搜索陈旧的脏索引节点，它执行下面的步骤：

1. 调用 `sync_supers()` 函数把脏的超级块写到磁盘中（参见下一节）。虽然这与页高速缓存中的页刷新没有很密切的关系，但对 `sync_supers()` 的调用确保了任何超级块脏的时间通常不会超过 5s。
2. 把当前时间减 30s 所对应的值（用 jiffies 表示）的指针存放在 `writeback_control` 描述符的 `older_than_this` 字段中。允许一个页保持脏状态的最长时间是 30s。
3. 根据每 CPU 变量 `page_state` 确定当前在页高速缓存中脏页的大概数量。
4. 反复调用 `writeback_inodes()`，直到写入磁盘的页数等于上一步所确定的值，或直到把所有保持脏状态时间超过 30s 的页都写到磁盘。如果在循环的过程中一些请求队列变得拥塞，函数就可能去睡眠。

5. 用 `mod_timer()` 重新启动 `wb_timer` 动态定时器：一旦从调用该函数开始经历过文件 `dirty_writeback_centisecs` 中规定的几分之一秒时间后，定时器到期（或者如果本次执行的时间太长，就从现在开始 1s 后到期）。

sync()、fsync()和fdatasync()系统调用

在本节我们简要介绍用户应用程序把脏缓冲区刷新到磁盘会用到的三个系统调用：

`sync()`

允许进程把所有的脏缓冲区刷新到磁盘。

`fsync()`

允许进程把属于特定打开文件的所有块刷新到磁盘。

`fdatasync()`

与 `fsync()` 非常相似，但不刷新文件的索引节点块。

sync ()系统调用

`sync()` 系统调用的服务例程 `sys_sync()` 调用一系列辅助函数：

```
wakeup_bdflush(0);
sync_inodes(0);
sync_supers();
sync_filesystems(0);
sync_filesystems(1);
sync_inodes(1);
```

正如上一节所描述的，`wakeup_bdflush()` 启动 `pdflush` 内核线程，把页高速缓存中的所有脏页刷新到磁盘。

`sync_inodes()` 函数扫描超级块的链表以搜索要刷新的脏索引节点；它作用于参数 `wait`，该参数表示在执行完刷新之前函数是否必须等待。函数扫描当前已安装的所有文件系统的超级块；对于每个包含脏索引节点的超级块，`sync_inodes()` 首先调用 `sync_sb_inodes()` 刷新相应的脏页（我们在前面“搜索要刷新的脏页”一节曾对该函数进行过说明），然后调用 `sync_blockdev()` 显式刷新该超级块所在块设备的脏缓冲区页。这一步之所以能完成是因为许多磁盘文件系统的 `write_inode` 超级块方法仅仅把磁盘索引节点对应的块缓冲区标记为“脏”；函数 `sync_blockdev()` 确保把 `sync_sb_inodes()` 所完成的更新有效地写到磁盘。

函数 `sync_supers()` 把脏超级块写到磁盘，如果需要，也可以使用适当的 `write_super` 超级块操作。最后，`sync_filesystems()` 为所有可写的文件系统执行 `sync_fs` 超级块

方法。该方法只不过是提供给文件系统的一个“钩子”，在需要对每个同步执行一些特殊操作时使用，只有像 Ext3(参见第十八章)这样的日志文件系统使用这个方法。

注意，`sync_inodes()`和`sync_filesystems()`都是被调用两次，一次是参数`wait`等于0时，另一次是`wait`等于1时。这样做的目的是：首先，它们把未上锁的索引节点快速刷新到磁盘；其次，它们等待所有上锁的索引节点被解锁，然后把它们逐个地写到磁盘。

`fsync ()`和`fdatasync ()` 系统调用

系统调用`fsync()`强制内核把文件描述符参数`fd`所指定文件的所有脏缓冲区写到磁盘中（如果需要，还包括存有索引节点的缓冲区）。相应的服务例程获得文件对象的地址，并随后调用`fsync`方法。通常这个方法以调用函数`__writeback_single_inode()`结束，该函数把与被选中的索引节点相关的脏页和索引节点本身都写回磁盘（参见本章前面“搜索要刷新的脏页”一节）。

系统调用`fdatasync()`与`fsync()`非常相似，但是它只把包含文件数据而不是那些包含索引节点信息的缓冲区写到磁盘。由于Linux 2.6没有提供专门的`fdatasync()`文件方法，该系统调用使用`fsync`方法，因此与`fsync()`是相同的。



访问基于磁盘的文件是一种复杂的活动，既涉及 VFS 抽象层（第十二章）、块设备的处理（第十四章），也涉及磁盘高速缓存的使用（第十五章）。本章介绍内核如何使用这些技术实现文件的读及写。本章所涵盖的主题既应用于磁盘文件系统的普通文件，也应用于块设备文件，将这两种文件系统都简单地统称为“文件”。

本章所介绍的内容是调用了读或写方法之后（如第十二章中所描述）系统所处的阶段。在这里，我们会说明每个读操作最终是如何把所需要的数据传递给用户态进程的，以及每个写操作最终又是如何把数据标志为“就绪”以传送到磁盘上的。其他传送过程是使用第十四章和第十五章中描述的技术来处理的。

访问文件的模式有多种。我们在本章考虑如下几种情况：

规范模式

规范模式下文件打开后，标志 `O_SYNC` 与 `O_DIRECT` 清 0，而且它的内容是由系统调用 `read()` 和 `write()` 来存取。系统调用 `read()` 将阻塞调用进程，直到数据被拷贝进用户态地址空间（内核允许返回的字节数少于要求的字节数）。但系统调用 `write()` 不同，它在数据被拷贝到页高速缓存（延迟写）后就马上结束。这会在“读写文件”这一节详细阐述。

同步模式

同步模式下文件打开后，标志 `O_SYNC` 置 1 或稍后由系统调用 `fcntl()` 对其置 1。这个标志只影响写操作（读操作总是会阻塞），它将阻塞调用进程，直到数据被有效地写入磁盘。这也会在“读写文件”这一节详细阐述。

内存映射模式

内存映射模式下文件打开后,应用程序发出系统调用`mmap()`将文件映射到内存中。因此,文件就成为RAM中的一个字节数组,应用程序就可以直接访问数组元素,而不需用系统调用`read()`、`write()`或`lseek()`。这将在“内存映射”这一节详细阐述。

直接 I/O 模式

直接 I/O 模式下文件打开后,标志`O_DIRECT`置 1。任何读写操作都将数据在用户态地址空间与磁盘间直接传送而不通过页高速缓存。这将在“直接 I/O 传送”这一节详细阐述。(标志`O_SYNC`和`O_DIRECT`的值可以有四种组合。)

异步模式

异步模式下,文件的访问可以有两种方法,即通过一组 POSIX API 或 Linux 特有的系统调用来实现。所谓异步模式就是数据传输请求并不阻塞调用进程,而是在后台执行,同时应用程序继续它的正常运行。这将在“异步 I/O”这一节详细阐述。

读写文件

在第十二章的“`read()`和`write()`系统调用”一节中已经说明了`read()`和`write()`系统调用是如何实现的。相应的服务例程最终会调用文件对象的`read`和`write`方法,这两个方法可能依赖文件系统。对磁盘文件系统来说,这些方法能够确定正被访问的数据所在物理块的位置,并激活块设备驱动程序开始数据传送。

读文件是基于页的,内核总是一次传送几个完整的数据页。如果进程发出`read()`系统调用来读取一些字节,而这些数据还不在RAM中,那么,内核就要分配一个新页框,并使用文件的适当部分来填充这个页,把该页加入页高速缓存,最后把所请求的字节拷贝到进程地址空间中。对于大部分文件系统来说,从文件中读取一个数据页就等同于在磁盘上查找所请求的数据存放在哪些块上。只要这个过程完成了,内核就可以通过向通用块层提交适当的 I/O 操作来填充这些页。事实上,大多数磁盘文件系统的`read`方法是由名为`generic_file_read()`的通用函数实现的。

对基于磁盘的文件来说,写操作的处理相当复杂,因为文件大小可以改变,因此内核可能会分配磁盘上的一些物理块。当然,这个过程到底如何实现要取决于文件系统的类型。不过,很多磁盘文件系统是通过通用函数`generic_file_write()`实现它们的`write`方法的。这样的文件系统如 Ext2、System V/Coherent/Xenix 及 Minix。另一方面,还有几个文件系统(如日志文件系统和网络文件系统)通过自定义的函数实现它们的`write`方法。

从文件中读取数据

让我们讨论一下 `generic_file_read()` 函数，该函数实现了几乎所有磁盘文件系统中的普通文件及任何块设备文件的 `read` 方法。该函数作用于以下参数：

<code>filp</code>	文件对象的地址
<code>buf</code>	用户态线性区的线性地址，从文件中读出的数据必须存放在这里
<code>count</code>	要读取的字符个数
<code>ppos</code>	指向一个变量的指针，该变量存放读操作开始处的文件偏移量（通常为 <code>filp</code> 文件对象的 <code>f_pos</code> 字段）

第一步，函数初始化两个描述符。第一个描述符存放在类型为 `iovec` 的局部变量 `local_iov` 中；它包含用户态缓冲区的地址 (`buf`) 与长度 (`count`)，该缓冲区用来存放待读文件中的数据。第二个描述符存放在类型为 `kiocb` 的局部变量 `kiocb` 中；它用来跟踪正在运行的同步和异步 I/O 操作的完成状态。`kiocb` 描述符的主要字段描述如表 16-1 所示。

表 16-1: `kiocb` 描述符的主要字段

类型	字段	说明
<code>struct list_head</code>	<code>ki_run_list</code>	以后要重新操作的 I/O 链表指针
<code>long</code>	<code>ki_flags</code>	<code>kiocb</code> 描述符的标志
<code>int</code>	<code>ki_users</code>	<code>kiocb</code> 描述符的引用计数器
<code>unsigned int</code>	<code>ki_key</code>	异步 I/O 操作标识符，同步 I/O 操作标识符为 <code>KIOCB_SYNC_KEY (0xffffffff)</code>
<code>struct file *</code>	<code>ki_filp</code>	与正在进行的 I/O 操作相关的文件对象指针
<code>struct kiocb *</code>	<code>ki_ctx</code>	异步 I/O 环境描述符指针（参见本章后面的“异步 I/O”一节）
<code>int (*)</code> (<code>struct kiocb *</code> , <code>struct io_event *</code>)	<code>ki_cancel</code>	当取消异步 I/O 操作时所调用的方法
<code>ssize_t (*)</code> (<code>struct kiocb *</code>)	<code>ki_retry</code>	当重试异步 I/O 操作时所调用的方法

表 16-1: kiocb 描述符的主要字段 (续)

类型	字段	说明
void (*) (struct kiocb *)	ki_dtor	当清除 kiocb 描述符时所调用的方法
struct list_head	ki_list	在异步操作环境下, 当前进行的 I/O 操作链表的指针
union	ki_obj	对于同步操作, 它是指向发出该操作的进程描述符的指针; 对于异步操作, 它是指向用户态数据结构 iocb 的指针
__u64	ki_user_data	给用户态进程返回的值
loff_t	ki_pos	正在进行 I/O 操作的当前文件位置
unsigned short	ki_opcode	操作类型(read、write 或 sync)
size_t	ki_nbytes	被传输的字节数
char *	ki_buf	用户态缓冲区的当前位置
size_t	ki_left	待传输的字节数
wait_queue_t	ki_wait	异步 I/O 操作等待队列
void *	private	由文件系统层自由使用

函数 `generic_file_read()` 通过执行宏 `init_sync_kiocb` 来初始化描述符 `kiocb`, 并设置一个同步操作对象的有关字段。具体地说就是, 该宏设置 `ki_key` 字段为 `KIOCB_SYNC_KEY`、`ki_filp` 字段为 `filp`、`ki_obj` 字段为 `current`。

然后, `generic_file_read()` 调用 `__generic_file_aio_read()` 并将刚填完的 `iovec` 和 `kiocb` 描述符地址传给它。后面这个函数返回一个值, 这个值通常就是从文件有效读入的字节数。`generic_file_read()` 返回值后结束。

函数 `__generic_file_aio_read()` 是所有文件系统实现同步和异步读操作所使用的通用例程。该函数接受四个参数: `kiocb` 描述符的地址 `kiocb`、`iovec` 描述符数组的地址 `iov`、数组的长度和存放文件当前指针的一个变量的地址 `ppos`。`iovec` 描述符数组被函数 `generic_file_read()` 调用时只有一个元素, 该元素描述待接收数据的用户态缓冲区(注 1)。

注 1: `read()` 系统调用的一个叫做 `readv()` 的变体允许应用程序定义多个用户态缓冲区, 从文件读出的数据分散存放在其中; `__generic_file_aio_read()` 函数也实现这种功能。下面我们要假设从文件读出的数据将只拷贝到一个用户态缓冲区, 不过, 可以想象, 使用多个缓冲区虽然简单, 但需要执行更多的步骤。

我们现在来说明函数 `_generic_file_aio_read()` 的操作。为简单起见，我们只针对最常见的情形，即对页高速缓存文件的系统调用 `read()` 所引发的同步操作。本章后面我们会阐述该函数执行的其他情形。同样，我们不讨论如何对错误和异常的处理。

该函数执行的步骤如下：

1. 调用 `access_ok()` 来检查 `iovec` 描述符所描述的用户态缓冲区是否有效。因为起始地址和长度已经从 `sys_read()` 服务例程得到，因此在使用前需要对它们进行检查（参见第十章“验证参数”一节）。如果参数无效，则返回错误代码 `-EFAULT`。
2. 建立一个读操作描述符，也就是一个 `read_descriptor_t` 类型的数据结构。该结构存放与单个用户态缓冲相关的文件读操作的当前状态。该描述符的字段参见表 16-2。
3. 调用函数 `do_generic_file_read()`，传送给它文件对象指针 `filp`、文件偏移量指针 `ppos`、刚分配的读操作描述符的地址和函数 `file_read_actor()` 的地址（后面还会阐述）。
4. 返回拷贝到用户态缓冲区的字节数，即 `read_descriptor_t` 数据结构中 `written` 字段的值。

表 16-2: 读操作描述符的字段

类型	字段	说明
<code>size_t</code>	<code>written</code>	已经拷贝到用户态缓冲区的字节数
<code>size_t</code>	<code>count</code>	待传送的字节数
<code>char *</code>	<code>buf</code>	在用户态缓冲区中的当前位置
<code>int</code>	<code>error</code>	读操作的错误码（0 表示无错误）

函数 `do_generic_file_read()` 从磁盘读入所请求的页并把它们拷贝到用户态缓冲区。具体执行如下步骤：

1. 获得要读取的文件对应的 `address_space` 对象，它的地址存放在 `filp->f_mapping`。
2. 获得地址空间对象的所有者，即索引节点对象，它将拥有填充了文件数据的页面。它的地址存放在 `address_space` 对象的 `host` 字段中。如果所读文件是块设备文件，那么所有者就不是由 `filp->f_dentry->d_inode` 所指向的索引节点对象，而是 `bdev` 特殊文件系统中的索引节点对象。
3. 把文件看作细分的数据页（每页 4096 字节），并从文件指针 `*ppos` 导出第一个请求字节所在页的逻辑号，即地址空间中的页索引，并把它存放在 `index` 局部变量中。也把第一个请求字节在页内的偏移量存放在 `offset` 局部变量中。

4. 开始一个循环来读入包含请求字节的所有页，要读数据的字节数存放在 `read_descriptor_t` 描述符的 `count` 字段中。在一次单独的循环期间，函数通过执行下列的子步骤来传送一个数据页：
 - a. 如果 `index*4096+offset` 超过存放在索引节点对象的 `i_size` 字段中的文件大小，则从循环退出，并跳到第 5 步。
 - b. 调用 `cond_resched()` 来检查当前进程的标志 `TIF_NEED_RESCHED`。如果该标志置位，则调用函数 `schedule()`。
 - c. 如果有预读的页，则调用 `page_cache_readahead()` 读入这些页面。我们在后面“文件的预读”一节讨论预读。
 - d. 调用 `find_get_page()`，并传入指向 `address_space` 对象的指针及索引值作为参数；它将查找页高速缓存以找到包含所请求数据的页描述符（如果有的话）。
 - e. 如果 `find_get_page()` 返回 `NULL` 指针，则所请求的页不在页高速缓存中。如果这样，它将执行如下步骤：
 - (1) 调用 `handle_ra_miss()` 来调整预读系统的参数。
 - (2) 分配一个新页。
 - (3) 调用 `add_to_page_cache()` 插入该新页描述符到页高速缓存中。记住该函数将新页的 `PG_locked` 标志置位。
 - (4) 调用 `lru_cache_add()` 插入新页描述符到 LRU 链表（参见第十七章）。
 - (5) 跳到第 4j 步，开始读文件数据。
 - f. 如果函数已运行至此，说明页已经位于页高速缓存中。检查标志 `PG_uptodate`，如果置位，则页所存数据是最新的，因此无需从磁盘读数据。跳到第 4m 步。
 - g. 页中的数据是无效的，因此必须从磁盘读取。函数通过调用 `lock_page()` 函数获取对页的互斥访问。正如第十五章“页高速缓存的处理函数”一节中所描述的，如果 `PG_locked` 已经置位，则 `lock_page()` 阻塞当前进程直到标志被清 0。
 - h. 现在页已由当前进程锁定。然而，另一个进程也许会在上一步之前已从页高速缓存中删除该页，那么，它就要检查页描述符的 `mapping` 字段是否为 `NULL`。在这种情形下，它将调用 `unlock_page()` 来解锁页，减少它的引用计数（`find_get_page()` 增加计数），并跳回第 4a 步来重读同一页。
 - i. 如果函数已运行至此，说明页已被锁定且在页高速缓存中。再次检查标志 `PG_uptodate`，因为另一个内核控制路径可能已经完成第 4f 步和第 4g 步的必要读操作。如果标志置位，则调用 `unlock_page()` 并跳至第 4m 步来跳过读操作。

- j. 现在真正的I/O操作可以开始了,调用文件的address_space对象之readpage方法。相应的函数会负责激活磁盘到页之间的I/O数据传输。我们以后再讨论该函数对普通文件与块设备文件都会做些什么。
- k. 如果标志PG_uptodate还没有置位,则它会等待直到调用lock_page()函数后页被有效读入。该页在第4g步中锁定,一旦读操作完成就被解锁。因此当前进程在I/O数据传输完成时才停止睡眠。
- l. 如果index超出文件包含的页数(该数是通过将inode对象的i_size字段的值除以4096得到的),那么它将减少页的引用计数器,并跳出循环至第5步。这种情况发生在这个正被本进程读的文件同时有其他进程正在删减它的时候。
- m. 将应被拷入用户态缓冲区的页中的字节数存放在局部变量nr中。这个值应该等于页的大小(4096字节),除非offset非0(这只发生在读请求书的首尾页时)或请求数据不全在该文件中。
- n. 调用mark_page_accessed()将标志PG_referenced或PG_active置位,从而表示该页正被访问并且不应该被换出(参见第十七章)。如果同一文件(或它的一部分)在do_generic_file_read()的后续执行中要读几次,那么这个步骤只在第一次读时执行。
- o. 现在到了把页中的数据拷贝到用户态缓冲区的时候了。为了这么做,do_generic_file_read()调用file_read_actor()函数,该函数的地址作为参数传递。file_read_actor()执行下列步骤:
 - (1) 调用kmap(),该函数为处于高端内存中的页建立永久的内核映射(参见第八章“高端内存页框的内核映射”一节)。
 - (2) 调用__copy_to_user(),该函数把页中的数据拷贝到用户态地址空间(参见第十章“访问进程地址空间”一节)。注意,这个操作在访问用户态地址空间时如果有缺页异常将会阻塞进程。
 - (3) 调用kunmap()来释放页的任一永久内核映射。
 - (4) 更新read_descriptor_t描述符的count、written和buf字段。
- p. 根据传入用户态缓冲区的有效字节数来更新局部变量index和count。一般情况下,如果页的最后一个字节已拷贝到用户态缓冲区,那么index的值加1而offset的值清0;否则,index的值不变而offset的值被设为已拷贝到用户态缓冲区的字节数。
- q. 减少页描述符的引用计数器。
- r. 如果read_descriptor_t描述符的count字段不为0,那么文件中还有其他数据要读,跳至第4a步继续循环来读文件中的下一页数据。

5. 所有请求的或者说可以读到的数据已读完。函数更新预读数据结构 `filp->f_ra` 来标记数据已被顺序从文件读入（参见下一节“文件的预读”）。
6. 把 `index*4096+offset` 值赋给 `*ppos`，从而保存以后调用 `read()` 和 `write()` 进行顺序访问的位置。
7. 调用 `update_atime()` 把当前时间存放在文件的索引节点对象的 `i_atime` 字段中，并把它标记为脏后返回。

普通文件的 `readpage` 方法

我们从前一节看到，`do_generic_file_read()` 反复使用 `readpage` 方法把一个个页从磁盘读到内存中。

`address_space` 对象的 `readpage` 方法存放的是函数地址，这种函数有效地激活从物理磁盘到页高速缓存的 I/O 数据传送。对于普通文件，这个字段通常指向调用 `mpage_readpage()` 函数的封装函数。例如，Ext3 文件系统的 `readpage` 方法由下列函数实现：

```
int ext3_readpage(struct file *file, struct page *page)
{
    return mpage_readpage(page, ext3_get_block);
}
```

需要封装函数是因为 `mpage_readpage()` 函数接收的参数为待填充页的页描述符 `page` 及有助于 `mpage_readpage()` 找到正确块的函数的地址 `get_block`。封装函数依赖文件系统并因此能提供适当的函数来得到块。这个函数把相对于文件开始位置的块号转换为相对于磁盘分区中块位置的逻辑块号（例子参见第十八章）。当然，后一个参数依赖于普通文件所在文件系统的类型；在前面的例子中，这个参数就是 `ext3_get_block()` 函数的地址。所传递的 `get_block` 函数总是用缓冲区首部来存放有关重要信息，如块设备（`b_dev` 字段）、设备上请求数据的位置（`b_blocknr` 字段）和块状态（`b_state` 字段）。

函数 `mpage_readpage()` 在从磁盘读入一页时可选择两种不同的策略。如果包含请求数据的块在磁盘上是连续的，那么函数就用单个 `bio` 描述符向通用块层发出读 I/O 操作。而如果不连续，函数就对页上的每一块用不同的 `bio` 描述符来读。`get_block` 函数依赖于文件系统，它的一个重要作用就是：确定文件中的下一块在磁盘上是否也是下一块。

具体地说，`mpage_readpage()` 函数执行下列步骤：

1. 检查页描述符的 `PG_private` 字段：如果置位，则该页是缓冲区页，也就是该页与描述组成该页的块的缓冲区首部链表相关（参见第十五章“把块存放在页高速缓存

中”一节)。这意味着该页过去已从磁盘读入过,而且页中的块在磁盘上不是相邻的。跳到第 11 步,用一次读一块的方式读该页。

2. 得到块的大小(存放在 `page->mapping->host->i_blkbits` 索引节点字段),然后计算出访问该页的所有块所需要的两个值,即页中的块数及页中第一块的文件块号,也就是相对于文件起始位置页中第一块的索引。
3. 对于页中的每一块,调用依赖于文件系统的 `get_block` 函数,作为参数传递过去,得到逻辑块号,即相对于磁盘或分区开始位置的块索引。页中所有块的逻辑块号存放在一个本地数组中。
4. 在执行上一步的同时,检查可能发生的异常条件。具体有这几种情况:当一些块在磁盘上不相邻时,或某块落入“文件洞”内(参见第十八章的“文件的洞”一节)时,或一个块缓冲区已经由 `get_block` 函数写入时。那么跳到第 11 步,用一次读一块的方式读该页。
5. 如果函数运行至此,说明页中的所有块在磁盘上是相邻的。然而,它可能是文件中的最后一页,因此页中的一些块可能在磁盘上没有映像。如果这样的话,它将页中相应的块缓冲区填上 0;如果不是这样,它将页描述符的标志 `PG_mappedtodisk` 置位。
6. 调用 `bio_alloc()` 分配包含单一段的一个新 `bio` 描述符,并且分别用块设备描述符地址和页中第一个块的逻辑块号来初始化 `bi_bdev` 字段和 `bi_sector` 字段。这两个信息已在上面的第 3 步中得到。
7. 用页的起始地址、所读数据的首字节偏移量(0)和所读的字节总数设置 `bio` 段的 `bio_vec` 描述符。
8. 将 `mpage_end_io_read()` 函数的地址赋给 `bio->bi_end_io` 字段(见下面)。
9. 调用 `submit_bio()`,它将用数据传输的方向设定 `bi_rw` 标志,更新每 CPU 变量 `page_states` 来跟踪所读扇区数,并在 `bio` 描述符上调用 `generic_make_request()` 函数(参见第十四章的“向 I/O 调度程序发出请求”一节)。
10. 返回 0 (成功)。
11. 如果函数跳到这里,则页中含有的块在磁盘上不连续。如果页是最新的(`PG_uptodate` 置位),函数就调用 `unlock_page()` 来对该页解锁;否则调用 `block_read_full_page()` 用一次读一块的方式读该页(见下面)。
12. 返回 0 (成功)。

函数 `mpage_end_io_read()` 是 `bio` 的完成方法,一旦 I/O 数据传输结束它就开始执行。假定没有 I/O 错误,该函数将页描述符的标志 `PG_uptodate` 置位,调用 `unlock_page()` 来对该页解锁并唤醒任何因为该事件而睡眠的进程,然后调用 `bio_put()` 来清除 `bio` 描述符。

块设备文件的 readpage 方法

在第十三章“设备文件的 VFS 处理”一节和第十四章的“打开块设备文件”一节中，我们讨论了内核如何处理请求以打开块设备文件。我们还看到 `init_special_inode()` 函数如何建立设备的索引节点及 `blkdev_open()` 如何完成其打开阶段。

在 `bdev` 特殊文件系统中，块设备使用 `address_space` 对象，该对象存放在对应块设备索引节点的 `i_data` 字段。不像普通文件（在 `address_space` 对象中它的 `readpage` 方法依赖于文件所属的文件系统的类型），块设备文件的 `readpage` 方法总是相同的。它是由 `blkdev_readpage()` 函数实现的，该函数调用 `block_read_full_page()`：

```
int blkdev_readpage(struct file * file, struct * page page)
{
    return block_read_full_page(page, blkdev_get_block);
}
```

正如你看到的，这个函数又是一个封装函数，这里是 `block_read_full_page()` 函数的封装函数。这一次，第二个参数也指向一个函数，该函数把相对于文件开始处的文件块号转换为相对于块设备开始处的逻辑块号。不过，对于块设备文件来说，这两个数是一致的；因此，`blkdev_get_block()` 函数执行下列步骤：

1. 检查页中第一个块的块号是否超过块设备的最后一块的索引值（存放在 `bdev->bd_inode->i_size` 中的块设备大小除以存放在 `bdev->bd_block_size` 中的块大小得到该索引值；`bdev` 指向块设备描述符）。如果超过，那么对于写操作它返回 `-EIO`，而对于读操作它返回 0。（超出块设备读也是不允许的，但不返回错误代码。内核可以对块设备的最后数据试着发出读请求，而得到的缓冲区页只被部分映射）。
2. 设置缓冲区首部的 `b_dev` 字段为 `b_dev`。
3. 设置缓冲区首部的 `b_blocknr` 字段为文件块号，它将被作为参数传给本函数。
4. 把缓冲区首部的 `BH_Mapped` 标志置位，以表明缓冲区首部的 `b_dev` 和 `b_blocknr` 字段是有效的。

函数 `block_read_full_page()` 以一次读一块的方式读一页数据。正如我们已看到的，当读块设备文件和磁盘上块不相邻的普通文件时都使用该函数。它执行如下步骤：

1. 检查页描述符的标志 `PG_private`，如果置位，则该页与描述组成该页的块的缓冲区首部链表相关（参见第十五章的“把块存放在页高速缓存中”一节）；否则，调用 `create_empty_buffers()` 来为该页所含所有块缓冲区分配缓冲区首部。页中第一个缓冲区的缓冲区首部地址存放在 `page->private` 字段中。每个缓冲区首部的 `b_this_page` 字段指向该页中下一个缓冲区的缓冲区首部。

2. 从相对于页的文件偏移量 (`page->index` 字段) 计算出页中第一块的文件块号。
3. 对该页中每个缓冲区的缓冲区首部, 执行如下子步骤:
 - a. 如果标志 `BH_Uptodate` 置位, 则跳过该缓冲区继续处理该页的下一个缓冲区。
 - b. 如果标志 `BH_Mapped` 未置位, 并且该块未超出文件尾, 则调用依赖于文件系统的 `get_block` 函数, 该函数的地址已被作为参数得到。对于普通文件, 该函数在文件系统的磁盘数据结构中查找, 得到相对于磁盘或分区开始处的缓冲区逻辑块号。对于块设备文件, 不同的是该函数把文件块号当作逻辑块号。对这两种情形, 函数都将逻辑块号存放在相应缓冲区首部的 `b_blocknr` 字段中, 并将标志 `BH_Mapped` 置位 (注 2)。
 - c. 再检查标志 `BH_Uptodate`, 因为依赖于文件系统的 `get_block` 函数可能已触发块 I/O 操作而更新了缓冲区。如果 `BH_Uptodate` 置位, 则继续处理该页的下一个缓冲区。
 - d. 将缓冲区首部的地址存放在局部数组 `arr` 中, 继续该页的下一个缓冲区。
4. 假如上一步中没遇到“文件洞”, 则将该页的标志 `PG_mappedtodisk` 置位。
5. 现在局部变量 `arr` 中存放了一些缓冲区首部的地址, 与其对应的缓冲区的内容不是最新的。如果数组为空, 那么页中的所有缓冲区都是有效的, 因此, 该函数设置页描述符的 `PG_uptodate` 标志, 调用 `unlock_page()` 对该页解锁并返回。
6. 局部数组 `arr` 非空。对数组中的每个缓冲区首部, `block_read_full_page()` 执行下列子步骤:
 - a. 将 `BH_Lock` 标志置位。该标志一旦置位, 函数将一直等到该缓冲区释放。
 - b. 将缓冲区首部的 `b_end_io` 字段设为 `end_buffer_async_read()` 函数的地址 (见下面), 并将缓冲区首部的 `BH_Async_Read` 标志置位。
7. 对局部数组 `arr` 中的每个缓冲区首部调用 `submit_bh()`, 将操作类型设为 `READ`。就像我们在前面看到的那样, 该函数触发了相应块的 I/O 数据传输。
8. 返回 0。

注 2: 访问普通文件时, 如果一个数据块处于“文件洞”中, `get_block` 函数就可能找不到这个块 (参见第十八章“文件的洞”一节)。此时, 函数用 0 填充这个块缓冲区并设置缓冲区首部的 `BH_Uptodate` 标志。

函数 `end_buffer_async_read()` 是缓冲区首部的完成方法。对块缓冲区的 I/O 数据传输一结束，它就执行。假定没有 I/O 错误，函数将缓冲区首部的 `BH_Uptodate` 标志置位而将 `BH_Async_Read` 标志清 0。那么，函数就得到包含块缓冲区的缓冲区页描述符（它的地址存放在缓冲区首部的 `b_page` 字段中），同时检查是否页中所有块是最新的；如果是，函数将该页的 `PG_uptodate` 标志置位并调用 `unlock_page()`。

文件的预读

很多磁盘的访问都是顺序的。我们在第十八章会看到，普通文件以相邻扇区成组存放在磁盘上，因此很少移动磁头就可以快速检索到文件。当程序读或拷贝一个文件时，它通常从第一个字节到最后一个字节顺序地访问文件。因此，在处理进程对同一文件的一系列读请求时，可以从磁盘上很多相邻的扇区读取。

预读 (*read-ahead*) 是一种技术，这种技术在于在实际请求前读普通文件或块设备文件的几个相邻的数据页。在大多数情况下，预读能极大地提高磁盘的性能，因为预读使磁盘控制器处理较少的命令，其中的每条命令都涉及一大组相邻的扇区。此外，预读还能提高系统的响应能力。顺序读取文件的进程通常不需要等待请求的数据，因为请求的数据已经在 RAM 中了。

但是，预读对于随机访问的文件是没有用的；在这种情况下，预读实际上是有害的，因为它用无用的信息浪费了页高速缓存的空间。因此，当内核确定出最近所进行的 I/O 访问与前一次 I/O 访问不是顺序的时就减少或停止预读。

文件的预读需要更复杂的算法，这是由于以下几个原因：

- 由于数据是逐页进行读取的，因此预读算法不必考虑页内偏移量，只要考虑所访问的页在文件内部的位置就可以了。
- 只要进程持续地顺序访问一个文件，预读就会逐渐增加。
- 当前的访问与上一次访问不是顺序的时（随机访问），预读就会逐渐减少乃至禁止。
- 当一个进程重复地访问同一页（即只使用文件的很小一部分）时，或者当几乎所有的页都已在页高速缓存内时，预读就必须停止。
- 低级 I/O 设备驱动程序必须在合适的时候激活，这样当将来进程需要时，页已传送完毕。

如果请求的第一页紧跟上次访问所请求的最后一页，那么相对于上次的文件访问，内核把文件的这次访问看作是顺序的。

当访问给定文件时，预读算法使用两个页面集，各自对应文件的一个连续区域。这两个页面集分别叫做当前窗 (*current window*) 和预读窗 (*ahead window*)。

当前窗内的页是进程请求的页和内核预读的页，且位于页高速缓存内（当前窗内的页不必是最新的，因为 I/O 数据传输仍可能在运行中）。当前窗包含进程顺序访问的最后一页，且可能有内核预读但进程未请求的页。

预读窗内的页紧接着当前窗内的页，它们是内核正在预读的页。预读窗内的页都不是进程请求的，但内核假定进程会迟早请求。

当内核认为是顺序访问而且第一页在当前窗内时，它就检查是否建立了预读窗。如果没有，内核创建一个预读窗并触发相应页的读操作。理想情况下，进程继续从当前窗请求页，同时预读窗的页则正在传送。当进程请求的页在预读窗，那么预读窗就成为当前窗。

预读算法使用的主要数据结构是 `file_ra_state` 描述符，它的字段见表 16-3。每个文件对象在它的 `f_ra` 字段中存放这样的一个描述符。

表 16-3: `file_ra_state` 描述符的字段

类型	字段	说明
unsigned long	start	当前窗内第一页的索引
unsigned long	size	当前窗内的页数（当临时禁止预读时为 -1, 0 表示当前窗空）
unsigned long	flags	控制预读的一些标志
unsigned long	cache_hit	连续高速缓存命中数（进程请求的页同时又在页高速缓存内）
unsigned long	prev_page	进程请求的最后一页的索引
unsigned long	ahead_start	预读窗内第一页的索引
unsigned long	ahead_size	预读窗的页数（0 表示预读窗口空）
unsigned long	ra_pages	预读窗的最大页数（0 表示预读窗永久禁止）
unsigned long	mmap_hit	预读命中计数器（用于内存映射文件）
unsigned long	mmap_miss	预读失败计数器（用于内存映射文件）

当一个文件被打开时，在它的 `file_ra_state` 描述符中，除了 `prev_page` 和 `ra_pages` 这两个字段，其他的所有字段都置为 0。

`prev_page` 字段存放着进程在上一次读操作中所请求页的最后一页的索引。它的初值是 -1。

ra_pages字段表示当前窗的最大页数,即对该文件允许的最大预读量。该字段的初始值(缺省值)存放在该文件所在块设备的backing_dev_info描述符中(参见第十四章的“请求队列描述符”一节)。一个应用可以修改一个打开文件的ra_pages字段从而调整预读算法;具体的实现方法是调用posix_fadvise()系统调用,并传给它命令POSIX_FADV_NORMAL(设最大预读量为缺省值,通常是32页)、POSIX_FADV_SEQUENTIAL(设最大预读量为缺省值的两倍)和POSIX_FADV_RANDOM(最大预读量为0,从而永久禁止预读)。

flags字段内有两个重要的字段RA_FLAG_MISS和RA_FLAG_INCACHE。如果已被预读的页不在页高速缓存内(可能的原因是内核为了释放内存而加以收回了,参见第十七章),则第一个标志置位,这时候下一个要创建的预读窗大小将被缩小。当内核确定进程请求的最后256页都在页高速缓存内时(连续高速缓存命中数存放在ra->cache_hit字段中),第二个标志置位,这时内核认为所有的页都已在页高速缓存内,进而关闭预读。

何时执行预读算法?这有下列几种情形:

- 当内核用用户态请求来读文件数据的页时。这一事件触发page_cache_readahead()函数的调用(参见本章前面“从文件中读取数据”一节有关do_generic_file_read()描述的第4c步)。
- 当内核为文件内存映射分配一页时(参见本章后面“内存映射的请求调页”一节中的filemap_nopage()函数,它再次调用page_cache_readahead()函数)。
- 当用户态应用执行readahead()系统调用时,它会对某个文件描述符显式触发某预读活动。
- 当用户态应用使用POSIX_FADV_NOREUSE或POSIX_FADV_WILLNEED命令执行posix_fadvise()系统调用时,它会通知内核,某个范围的文件页不久将要被访问。
- 当用户态应用使用MADV_WILLNEED命令执行madvise()系统调用时,它会通知内核,某个文件内存映射区域中的给定范围的文件页不久将要被访问。

page_cache_readahead() 函数

page_cache_readahead()函数处理没有被特殊系统调用显式触发的所有预读操作。它填写当前窗和预读窗,根据预读命中数更新当前窗和预读窗的大小,也就是根据过去对文件访问预读策略的成功程度来调整。

当内核必须满足对某个文件一页或多页的读请求时,函数就被调用,该函数有下面五个参数:

mapping	描述页所有者的 address_space 对象指针
ra	包含该页的文件 file_ra_state 描述符指针
filp	文件对象地址
offset	文件内页的偏移量
req_size	要完成当前读操作还需要读的页数（注3）

图 16-1 是 page_cache_readahead() 的流程图。该函数基本上作用于 file_ra_state 描述符的字段，因此，尽管流程图中的行为描述不很正规，你还是能很容易地确定函数执行的实际步骤。例如，为了检查请求页是否与刚读的页相同，函数检查 ra->prev_page 字段的值和 offset 参数的值是否一致（见前面的表 16-3）。

当进程第一次访问一个文件，并且其第一个请求页是文件中偏移量为 0 的页时，函数假定进程要进行顺序访问。那么，函数从第一页创建一个新的当前窗。初始当前窗的长度（总是为 2 的幂）与进程第一个读操作所请求的页数有一定的联系。请求页数越大，当前窗越大，一直到最大值，最大值存放在 ra->ra_pages 字段。反之，当进程第一次访问文件，但其第一个请求页在文件中的偏移量不为 0 时，函数假定进程不是执行顺序读。那么，函数暂时禁止预读（ra->size 字段设为 -1）。但是当预读暂时被禁止而函数又认为需要顺序访问时，将建立一个新的当前窗。

如果预读窗不存在，一旦函数认为在当前窗内进程执行了顺序读，则预读窗将被建立。预读窗总是从当前窗的最后一页开始。但它的长度与当前窗的长度相关：如果 RA_FLAG_MISS 标志置位，则预读窗长度是当前窗长度减 2，小于 4 时设为 4；否则，预读窗长度是当前窗长度的 4 倍或 2 倍。如果进程继续顺序访问文件，最终预读窗成为新的当前窗，新的预读窗被创建。这样，随着进程顺序地读文件，预读会大大地增强。

注 3：实际上，如果读操作要读的页数大于预读窗的最大尺寸，就会多次调用 page_cache_readahead() 函数。因此，req_size 参数可能比完成读操作还需要读的页数小。

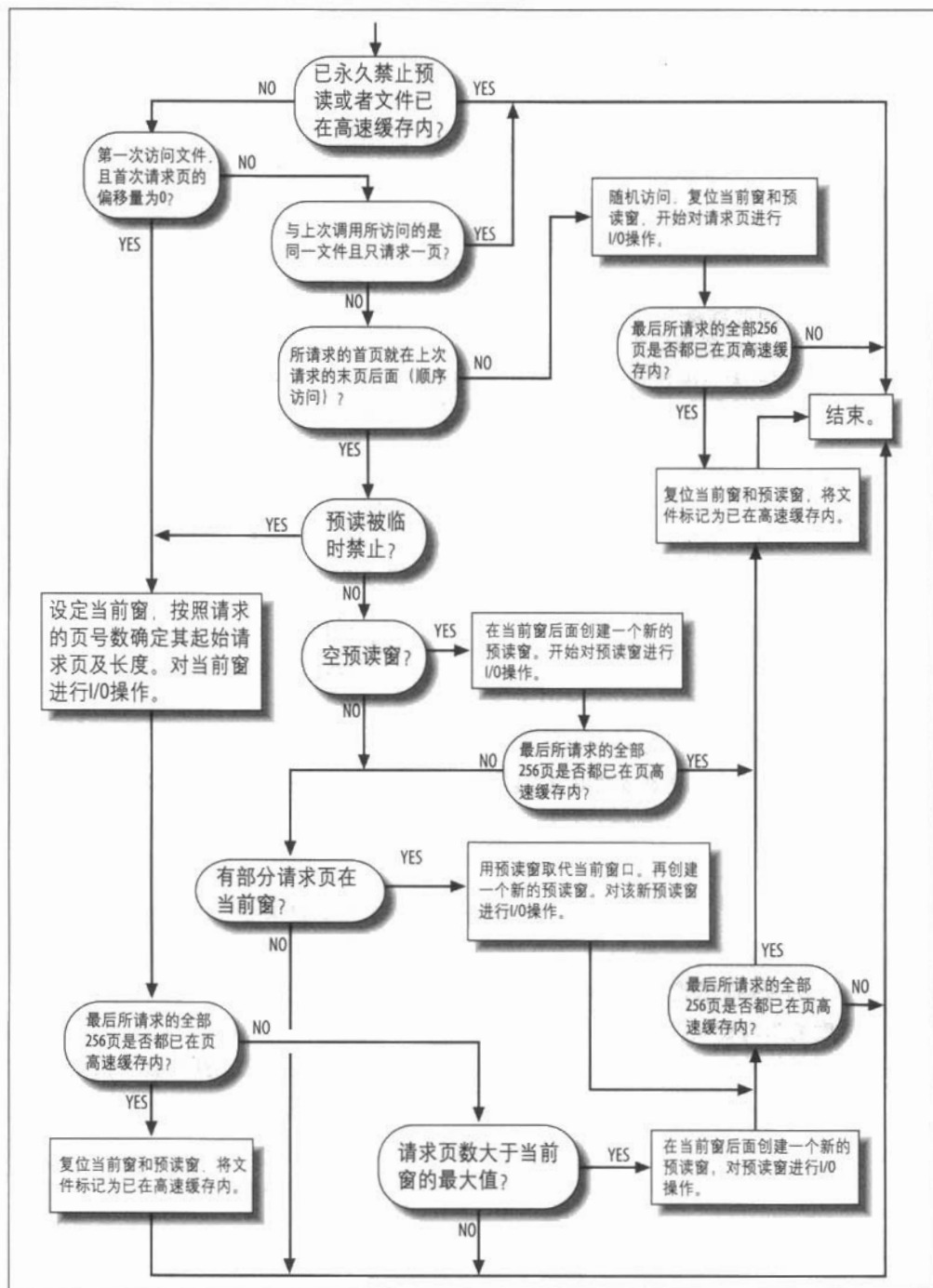


图 16-1: 函数 page_cache_readahead() 的流程图

一旦函数认识到对文件的访问相对于上一次不是顺序的，当前窗与预读窗就被清空，预读被暂时禁止。当进程的读操作相对于上一次文件访问为顺序时，预读将重新开始。

每次 `page_cache_readahead()` 创建一个新窗，它就开始对所包含页的读操作。为了读一大组页，函数 `page_cache_readahead()` 调用 `blockable_page_cache_readahead()`。为减少内核开销，后面这个函数采用下面灵活的方法：

- 如果服务于块设备的请求队列是读拥塞的，就不进行读操作。
- 将要读的页与页高速缓存进行比较，如果该页已在页高速缓存内，跳过即可。
- 在从磁盘进行读之前，读请求所需的全部页框是一次性分配的。如果不能一次性得到全部页框，预读操作就只在可以得到的页上进行。而且把预读推迟至所有页框都得到时再进行并没有多大意义。
- 只要可能，通过使用多段 `bio` 描述符向通用块层发出读操作（参见第十四章“段”一节）。这通过 `address_space` 对象专用的 `readpages` 方法实现（假如已定义）；如果没有定义，就通过反复调用 `readpage` 方法来实现。`readpage` 方法在前面“从文件中读取数据”一节中对于单段情形有详细描述，但稍作修改就可以很容易地将其用于多段情形。

`handle_ra_miss()` 函数

在某些情况下，预读策略似乎不是十分有效，内核就必须修正预读参数。让我们考虑本章前面“从文件中读取数据”一节中描述的 `do_generic_file_read()` 函数。在第 4c 步中调用函数 `page_cache_readahead()`。图 16-1 中展示了两种情形：请求页在当前窗或预读窗表明它已经被预先读入了，或者还没有，则调用 `blockable_page_cache_readahead()` 来读入。在这两种情形下，函数 `do_generic_file_read()` 应该在第 4d 步中就在页高速缓存中找到了该页，如果没有，就表示该页框已被收回算法从高速缓存中删除。在这种情形下，`do_generic_file_read()` 调用 `handle_ra_miss()` 函数，这个函数会通过将 `RA_FLAG_MISS` 标志置位与 `RA_FLAG_INCACHE` 标志清 0 来调整预读算法。

写入文件

回想一下，`write()` 系统调用涉及把数据从调用进程的用户态地址空间中移动到内核数据结构中，然后再移动到磁盘上。文件对象的 `write` 方法允许每种文件类型都定义一个专用的写操作。在 Linux 2.6 中，每个磁盘文件系统的 `write` 方法都是一个过程，该过程主要标识写操作所涉及的磁盘块，把数据从用户态地址空间拷贝到页高速缓存的某些页中，然后把这些页中的缓冲区标记成脏。

许多文件系统（包括Ext2或JFS）通过generic_file_write()函数来实现文件对象的write方法。它有如下参数：

file

文件对象指针

buf

用户态地址空间中的地址，必须从这个地址获取要写入文件的字符

count

要写入的字符个数

ppos

存放文件偏移量的变量地址，必须从这个偏移量处开始写入

该函数执行以下操作：

1. 初始化iovec类型的一个局部变量，它包含用户态缓冲区的地址与长度（参见本章前面“从文件读取数据”一节中对generic_file_read()函数的描述）。
2. 确定所写文件索引节点对象的地址inode(file->f_mapping->host)和获得信号量(inode->i_sem)。有了这个信号量，一次只能有一个进程对某个文件发出write()系统调用。
3. 调用宏init_sync_kiocb初始化kiocb类型的局部变量。就像本章前面“从文件读取数据”一节中描述的那样，该宏将ki_key字段设置为KIOCB_SYNC_KEY（同步I/O操作）、ki_filp字段设置为filp、ki_obj字段设置为current。
4. 调用__generic_file_aio_write_nolock()函数（见下面）将涉及的页标记为脏，并传递相应的参数：iovec和kiocb类型的局部变量地址、用户态缓冲区的段数（这里只有一个）和ppos。
5. 释放inode->i_sem信号量。
6. 检查文件的O_SYNC标志、索引节点的S_SYNC标志及超级块的MS_SYNCHRONOUS标志。如果至少一个标志置位，则调用函数sync_page_range()来强制内核将页高速缓存中第4步涉及的所有页刷新，阻塞当前进程直到I/O数据传输结束。然后依次地，sync_page_range()先执行address_space对象的writepages方法（如果有定义）或mpage_writepages()函数来开始这些脏页的I/O传输（参见本章后面“将脏页写到磁盘”一节），然后调用generic_osync_inode()将索引节点和相关的缓冲区刷新到磁盘，最后调用wait_on_page_bit()挂起当前进程一直到全部所刷新页的PG_writeback标志清0。

7. 将 `__generic_file_aio_write_nolock()` 函数的返回值返回，通常是写入的有效字节数。

函数 `__generic_file_aio_write_nolock()` 接收四个参数：`kiocb` 描述符的地址 `iocb`、`iovec` 描述符数组的地址 `iov`、该数组的长度以及存放文件当前指针的变量的地址 `ppos`。当被 `generic_file_write()` 调用时，`iovec` 描述符数组只有一个元素，该元素描述待写数据的用户态缓冲区（注4）。

我们现在来解释 `__generic_file_aio_write_nolock()` 函数的行为。为简单起见，我们只讨论最常见的情形，即对有页高速缓存的文件进行 `write()` 系统调用的一般情况。我们在本章后面会讨论该函数在其他情况下的行为。我们不讨论如何处理错误和异常条件。

该函数执行如下步骤：

1. 调用 `access_ok()` 确定 `iovec` 描述符所描述的用户态缓冲区是有效的（起始地址和长度已从服务例程 `sys_write()` 得到，因此使用前必须对其检查。参见第十章“验证参数”一节）。如果参数无效，则返回错误 `-EFAULT`。
2. 确定待写文件 (`file->f_mapping->host`) 索引节点对象的地址 `inode`。记住：如果文件是一个块设备文件，这就是一个 `bdev` 特殊文件系统的索引节点（参见第十四章）。
3. 将文件 (`file->f_mapping->backing_dev_info`) 的 `backing_dev_info` 描述符的地址设为 `current->backing_dev_info`。实际上，即使相应请求队列是拥塞的，这个设置也会允许当前进程写回由 `file->f_mapping` 拥有的脏页（参见第十七章）。
4. 如果 `file->flags` 的 `O_APPEND` 标志置位而且文件是普通文件（非块设备文件），它将 `*ppos` 设为文件尾，从而新数据将都追加到文件的后面。
5. 对文件大小进行几次检查。比如，写操作不能把一个普通文件增大到超过每用户的上限或文件系统的上限，每用户上限存放在 `current->signal->rlim[RLIMIT_FSIZE]`（参见第三章“进程资源限制”一节），文件系统上限存放在 `inode->i_sb->s_maxbytes`。另外，如果文件不是“大型文件”（当 `file->f_flags` 的 `O_LARGEFILE` 标志清0时），那么它的大小不能超出2GB。如果没有设定所述限制，它就减少待写字节数。

注4：系统调用 `write()` 的一个叫做 `writew()` 的变体允许应用程序定义多个用户态缓冲区，从中可以获取待写入文件的数据。函数 `generic_file_aio_write_nolock()` 也具有这种功能。下面我们假设将从一个用户缓冲区取数据，不过我们可以想象，使用多个缓冲区虽然简单，但需要执行更多的步骤。

6. 如果设定, 则将文件的 `suid` 标志清 0, 而且如果是可执行文件的话就将 `sgid` 标志也清 0 (参见第一章“访问权限和文件模式”一节)。我们并不要用户能修改 `setuid` 文件。
7. 将当前时间存放在 `inode->mtime` 字段 (文件写操作的最新时间) 中, 也存放在 `inode->ctime` 字段 (修改索引节点的最新时间) 中, 而且将索引节点对象标记为脏。
8. 开始循环以更新写操作中涉及的所有文件页。在每次循环期间, 执行下列子步骤:
 - a. 调用 `find_lock_page()` 在页高速缓存中搜索该页 (参见第十五章“页高速缓存的处理函数”一节)。如果函数找到了该页, 则增加引用计数并将 `PG_locked` 标志置位。
 - b. 如果该页不在页高速缓存中, 则分配一个新页框并调用 `add_to_page_cache()` 在页高速缓存内插入此页。正如第十五章“页高速缓存的处理函数”一节所述的那样, 这个函数也会增加引用计数并将 `PG_locked` 标志置位。另外函数还在内存管理区的非活动链表中插入一页 (参见第十七章)。
 - c. 调用索引节点 (`file->f-mapping`) 中 `address_space` 对象的 `prepare_write` 方法。对应的函数会为该页分配和初始化缓冲区首部。我们在后面的章节中再讨论该函数对于普通文件和块设备文件做些什么。
 - d. 如果缓冲区在高端内存中, 则建立用户态缓冲区的内核映射 (参见第八章的“高端内存页框的内核映射”一节), 然后它调用 `__copy_from_user()` 把用户态缓冲区中的字符拷贝到页中, 并且释放内核映射。
 - e. 调用索引节点 (`file->f-mapping`) 中 `address_space` 对象的 `commit_write` 方法。对应的函数把基础缓冲区标记为脏, 以便随后把它们写到磁盘。我们在后面两节讨论该函数对于普通文件和块设备文件做些什么。
 - f. 调用 `unlock_page()` 清 `PG_locked` 标志, 并唤醒等待该页的任何进程。
 - g. 调用 `mark_page_accessed()` 来为内存回收算法更新页状态 [参见第十七章“最近最少使用 (LRU) 链表”一节]。
 - h. 减少页引用计数来撤销第 8a 或 8b 步中的增加值。
 - i. 在这一步, 还有另一页被标记为脏, 它检查页高速缓存中脏页比例是否超过一个固定的阈值 (通常为系统中页的 40%)。如果这样, 则调用 `writeback_inodes()` 来刷新几十页到磁盘 (参见第十五章的“搜索要刷新的脏页”一节)。

- j. 调用 `cond_resched()` 来检查当前进程的 `TIF_NEED_RESCHED` 标志。如果该标志置位，则调用 `schedule()` 函数。
9. 现在，在写操作中所涉及的文件的所有页都已处理。更新 `*ppos` 的值，让它正好指向最后一个被写入的字符之后的位置。
10. 设置 `current->backing_dev_info` 为 `NULL`（参见第 3 步）。
11. 返回写入文件的有效字符数后结束。

普通文件的 `prepare_write` 和 `commit_write` 方法

`address_space` 对象的 `prepare_write` 和 `commit_write` 方法专用于由 `generic_file_write()` 实现的通用写操作，这个函数适用于普通文件和块设备文件。对文件的受写操作影响的每一页，调用一次这两个方法。

每个磁盘文件系统都定义了自己的 `prepare_write` 方法。与读操作类似，这个方法只不过是普通函数的一个封装函数。例如，`Ext2` 文件系统通过下列函数实现 `prepare_write` 方法：

```
int ext2_prepare_write(struct file *file, struct page *page, unsigned
from, unsigned to)
{
    return block_prepare_write(page, from, to, ext2_get_block);
}
```

在前面“从文件读取数据”一节已经提到 `ext2_get_block()` 函数；它把相对于文件的块号转换为逻辑块号（表示数据在物理块设备上的位置）。

`block_prepare_write()` 函数通过执行下列步骤为文件页的缓冲区和缓冲区首部做准备：

1. 检查某页是否是一个缓冲区页（如果是则 `PG_Private` 标志置位）；如果该标志清 0，则调用 `create_empty_buffers()` 为页中所有的缓冲区分配缓冲区首部（参见第十五章“缓冲区页”一节）。
2. 对与页中包含的缓冲区对应的每个缓冲区首部，及受写操作影响的每个缓冲区首部，执行下列操作：
 - a. 如果 `BH_New` 标志置位，则将它清 0（参见下面）。
 - b. 如果 `BH_New` 标志已清 0，则函数执行下列子步骤：
 - (1) 调用依赖于文件系统的函数，该函数的地址 `get_block` 以参数形式传递过来。查看这个文件系统磁盘数据结构并查找缓冲区的逻辑块号（相对于磁

盘分区的起始位置而不是普通文件的起始位置)。与文件系统相关的函数把这个数存放在对应缓冲区首部的**b_blocknr**字段,并设置它的**BH_Mapped**标志。与文件系统相关的函数可能为文件分配一个新的物理块(例如,如果访问的块掉进普通文件的一个“洞”中,参见第十八章“文件的洞”一节)。在这种情况下,设置**BH_New**标志。

- (2) 检查 **BH_New** 标志的值; 如果它被置位, 则调用 `unmap_underlying_metadata()` 来检查页高速缓存内的某个块设备缓冲区页是否包含指向磁盘同一块的一个缓冲区(注5)。该函数实际上调用 `__find_get_block()` 在页高速缓存内查找一个旧块(参见第十五章“在页高速缓存中搜索块”一节)。如果找到一块, 函数将 **BH_Dirty** 标志清0并等待直到该缓冲区的I/O数据传输完毕。此外, 如果写操作不对整个缓冲区进行重写, 则用0填充未写区域。然后考虑页中的下一个缓冲区。
 - c. 如果写操作不对整个缓冲区进行重写且它的**BH_Delay**和**BH_Uptodate**标志未置位(也就是说, 已在磁盘文件系统数据结构中分配了块, 但是RAM中的缓冲区并没有有效的数据映像), 函数对该块调用 `ll_rw_block()` 从磁盘读取它的内容(参见第十五章“向通用块层提交缓冲区首部”一节)。
3. 阻塞当前进程, 直到在第2c步触发的所有读操作全部完成。
4. 返回0。

一旦 `prepare_write` 方法返回, `generic_file_write()` 函数就用存放在用户态地址空间中的数据更新页。接下来, 调用 `address_space` 对象的 `commit_write` 方法。这个方法由 `generic_commit_write()` 函数实现, 几乎适用于所有非日志型磁盘文件系统。

`generic_commit_write()` 函数执行下列步骤:

1. 调用 `__block_commit_write()` 函数, 然后依次执行如下步骤:
 - a. 考虑页中受写操作影响的所有缓冲区; 对于其中的每个缓冲区, 将对应缓冲区首部的 **BH_Uptodate** 和 **BH_Dirty** 标志置位。
 - b. 标记相应索引节点为脏, 正如第十五章“搜索要刷新的脏页”一节中所述, 这需要将索引节点加入超级块脏的索引节点链表。
 - c. 如果缓冲区页中的所有缓冲区是最新的, 则将 **PG_uptodate** 标志置位。

注5: 尽管可能性不大, 但在一个用户直接向块设备文件写数据块时, 还是会出现这种情况, 从而越过文件系统。

- d. 将页的 PG_dirty 标志置位，并在基树中将页标记成脏（参见第十五章“基树”一节）。
2. 检查写操作是否将文件增大。如果增大，则更新文件索引节点对象的 i_size 字段。
3. 返回 0。

块设备文件的 prepare_write 和 commit_write 方法

写入块设备文件的操作非常类似于对普通文件的相应操作。事实上，块设备文件的 address_space 对象的 prepare_write 方法通常是由下列函数实现的：

```
int blkdev_prepare_write(struct file *file, struct page *page, unsigned
from, unsigned to)
{
    return block_prepare_write(page, from, to, blkdev_get_block);
}
```

你可以看到，这个函数只不过是前一节讨论过的 block_prepare_write() 函数的封装函数。当然，唯一的差异是第二个参数，它是一个指向函数的指针，该函数必须把相对于文件开始处的文件块号转换为相对于块设备开始处的逻辑块号。回想一下，对于块设备文件来说，这两个数是一致的（参见前面“从文件读取数据”一节中对 blkdev_get_block() 函数的讨论）。

用于块设备文件的 commit_write 方法是由下列简单的封装函数实现的：

```
int blkdev_commit_write(struct file *file, struct page *page, unsigned
from, unsigned to)
{
    return block_commit_write(page, from, to);
}
```

正如你所看到的，用于块设备的 commit_write 方法与用于普通文件的 commit_write 方法本质上做同样的事情（我们在前一节描述了 block_commit_write() 函数）。唯一的差异是这个方法不检查写操作是否扩大了文件，你根本不可能在块设备文件的末尾追加字符来扩大它。

将脏页写到磁盘

系统调用 write() 的作用就是修改页高速缓存内一些页的内容，如果页高速缓存内没有所要的页则分配并追加这些页。某些情况下（例如文件带 O_SYNC 标志打开），I/O 数据传输立即启动（参见本章前面“写入文件”一节中 generic_file_write() 函数的第 6 步）。但是通常 I/O 数据传输是延迟进行的，这在第十五章的“把脏页写入磁盘”一节中描述过。

当内核要有效启动 I/O 数据传输时，就要调用文件 `address_space` 对象的 `writepages` 方法，它在基树中寻找脏页，并把它们刷新到磁盘。例如 Ext2 文件系统通过下面的函数实现 `writepages` 方法：

```
int ext2_writepages(struct address_space *mapping, struct
writeback_control *wbc)
{
    return mpage_writepages(mapping, wbc, ext2_get_block);
}
```

你可以看到，该函数是通用 `mpage_writepages()` 的一个简单的封装函数。事实上，若文件系统没有定义 `writepages` 方法，内核则直接调用 `mpage_writepages()` 并把 `NULL` 传给第三个参数。`ext2_get_block()` 函数在前面“从文件读取数据”一节中已讲到过，这是一个依赖于文件系统的函数，它将文件块号转换成逻辑块号。

`writeback_control` 数据结构是一个描述符，它控制 `writeback` 写回操作如何执行，我们在第十五章“搜索要被刷新的脏页”一节中已有描述。

`mpage_writepages()` 函数执行下列步骤：

1. 如果请求队列写阻塞，但进程不希望阻塞，则不向磁盘写任何页就返回。
2. 确定文件的首页，如果 `writeback_control` 描述符给定一个文件内的初始位置，函数将把它转换成页索引。否则，如果 `writeback_control` 描述符指定进程无需等待 I/O 数据传输结束，它将 `mapping->writeback_index` 的值设为初始页索引（即从上一步写回操作的最后一页开始扫描）。最后，如果进程必须等待 I/O 数据传输完毕，则从文件的第一页开始扫描。
3. 调用 `find_get_pages_tag()` 在页高速缓存中查找脏页描述符（参见第十五章“基树的标记”一节）。
4. 对上一步得到的每个页描述符，执行如下步骤：
 - a. 调用 `lock_page()` 来锁定该页。
 - b. 确认页是有效的并在页高速缓存内（因为另一个内核控制路径可能已在第 3 步与第 4a 步间作用于该页）。
 - c. 检查页的 `PG_writeback` 标志。如果置位，表明页已被刷新到磁盘。如果进程必须等待 I/O 数据传输完毕，则调用 `wait_on_page_bit()` 在 `PG_writeback` 清 0 之前一直阻塞当前进程；当函数结束时，以前运行的任何 `writeback` 操作都被终止。否则，如果进程无需等待，它将检查 `PG_dirty` 标志：如果 `PG_dirty` 标志现已清 0，则正在运行的写回操作将处理该页，将它解锁并跳回第 4a 步继续下一页。

- d. 如果 `get_block` 的参数是 `NULL` (没有定义 `writpages` 方法), 它将调用文件 `address_space` 对象的 `mapping->writepage` 方法将页刷新到磁盘。否则, 如果 `get_block` 的参数不是 `NULL`, 它就调用 `mpage_writepage()` 函数。详见第 8 步。
5. 调用 `cond_resched()` 来检查当前进程的 `TIF_NEED_RESCHED` 标志, 如果该标志置位就调用 `schedule()` 函数。
6. 如果函数没有扫描完给定范围内的所有页, 或者写到磁盘的有效页数小于 `writeback_control` 描述符中原来的给定值, 那么跳回第 3 步。
7. 如果 `writeback_control` 描述符没有给定文件内的初始位置, 它将最后一个扫描页的索引值赋给 `mapping->writeback_index` 字段。
8. 如果在第 4d 步中调用了 `mpage_writepage()` 函数, 而且返回了 `bio` 描述符地址, 那么调用 `mpage_bio_submit()` (见下面)。

像 Ext2 这样的典型文件系统所实现的 `writepage` 方法是一个通用的 `block_write_full_page()` 函数的封装函数, 并将依赖于文件系统的 `get_block` 函数的地址作为参数传给它。就像本章前面“从文件读取数据”一节描述的 `block_read_full_page()` 一样, `block_write_full_page()` 函数也依次执行: 分配页缓冲区首部 (如果还不在于缓冲区页中), 对每页调用 `submit_bh()` 函数来指定写操作。就块设备文件而言, 就用 `block_write_full_page()` 的封装函数 `blkdev_writepage()` 实现 `writepage` 方法。

许多非日志型文件系统依赖于 `mpage_writepage()` 函数而不是自定义的 `writepage` 方法。这样能改善性能, 因为 `mpage_writepage()` 函数进行 I/O 传输时, 在同一个 `bio` 描述符中聚集尽可能多的页。这就使得块设备驱动程序能利用现代硬盘控制器的 DMA 分散-聚集能力。

长话短说, `mpage_writepage()` 函数将检查: 待写页包含的块在磁盘上是否不相邻, 该页是否包含文件洞, 页上的某块是否没有脏或不是最新的。如果以上情况至少一条成立, 函数就像上面那样仍然用依赖于文件系统的 `writepage` 方法。否则, 将页追加为 `bio` 描述符的一段。 `bio` 描述符的地址将作为参数被传给函数, 如果该地址为 `NULL`, `mpage_writepage()` 将初始化一个新的 `bio` 描述符并将地址返回给调用函数, 调用函数转而在未来调用 `mpage_writepage()` 时再将该地址传回来。这样, 同一个 `bio` 可以加载几个页。如果 `bio` 中某页与上一个加载页不相邻, `mpage_writepage()` 就调用 `mpage_bio_submit()` 开始该 `bio` 的 I/O 数据传输, 并为该页分配一个新的 `bio`。

`mpage_bio_submit()` 函数将 `bio` 的 `bi_end_io` 方法设为 `mpage_end_io_write()` 的地址; 然后调用 `submit_bio()` 开始传输 (参见第十五章“向通用块层提交缓冲区首部”

一节)。一旦数据传输成功结束，完成函数 `mpage_end_io_write()` 就唤醒那些等待页传输结束的进程，并清除 `bio` 描述符。

内存映射

正如我们在第九章的“线性区”一节中已经介绍过的一样，一个线性区可以和磁盘文件系统的普通文件的某一部分或者块设备文件相关联。这就意味着内核把对区线性中页内某个字节的访问转换成对文件中相应字节的操作。这种技术称为内存映射 (*memory mapping*)。

有两种类型的内存映射：

共享型

在线性区页上的任何写操作都会修改磁盘上的文件，而且，如果进程对共享映射中的一个页进行写，那么这种修改对于其他映射了这同一文件的所有进程来说都是可见的。

私有型

当进程创建的映射只是为读文件，而不是写文件时才会使用此种映射。出于这种目的，私有映射的效率要比共享映射的效率更高。但是对私有映射页的任何写操作都会使内核停止映射该文件中的页。因此，写操作既不会改变磁盘上的文件，对访问相同文件的其他进程也不可见。但是私有内存映射中还没有被进程改变的页面因为其他进程进行的文件更新而更新。

进程可以发出一个 `mmap()` 系统调用来创建一个新的内存映射（参见本章后面的“创建内存映射”一节）。程序员必须指定一个 `MAP_SHARED` 标志或 `MAP_PRIVATE` 标志作为这个系统调用的参数。正如你可以猜到的那样，前一种情况下，映射是共享的，而后一种情况下，映射是私有的。一旦创建了这种映射，进程就可以从这个新线性区的内存单元读取数据，也就等价于读取了文件中存放的数据。如果这个内存映射是共享的，那么进程可以通过对相同的内存单元进行写而达到修改相应文件的目的。为了撤消或者缩小一个内存映射，进程可以使用 `munmap()` 系统调用（参见后面的“撤消内存映射”一节）。

作为一条通用规则，如果一个内存映射是共享的，相应的线性区就设置了 `VM_SHARED` 标志；如果一个内存映射是私有的，那么相应的线性区就清除了 `VM_SHARED` 标志。正如我们在后面会看到的一样，对于只读共享内存映射来说，有一个特例不符合本规则。

内存映射的数据结构

内存映射可以用下列数据结构的组合来表示：

- 与所映射的文件相关的索引节点对象
- 所映射文件的 address_space 对象
- 不同进程对一个文件进行不同映射所使用的文件对象
- 对文件进行每一不同映射所使用的 vm_area_struct 描述符
- 对文件进行映射的线性区所分配的每个页框所对应的页描述符

图 16-2 说明了这些数据结构是如何链接在一起的。图的左边给出了标识文件的索引节点。每个索引节点对象的 i_mapping 字段指向文件的 address_space 对象。每个 address_space 对象的 page_tree 字段又指向该地址空间的页的基树（参见第十五章“基树”一节），而 i_mmap 字段指向第二棵树，叫做 radix 优先级搜索树（priority search tree, PST），这种树由地址空间的线性区组成。PST 的主要作用是为了执行“反向映射”，这是为了快速标识共享一页的所有进程。我们将在下一章中详细讨论 PST，因为它们用于页框回收。对同一文件的文件对象和索引节点之间链接的建立是通过 f_mapping 字段达到的。

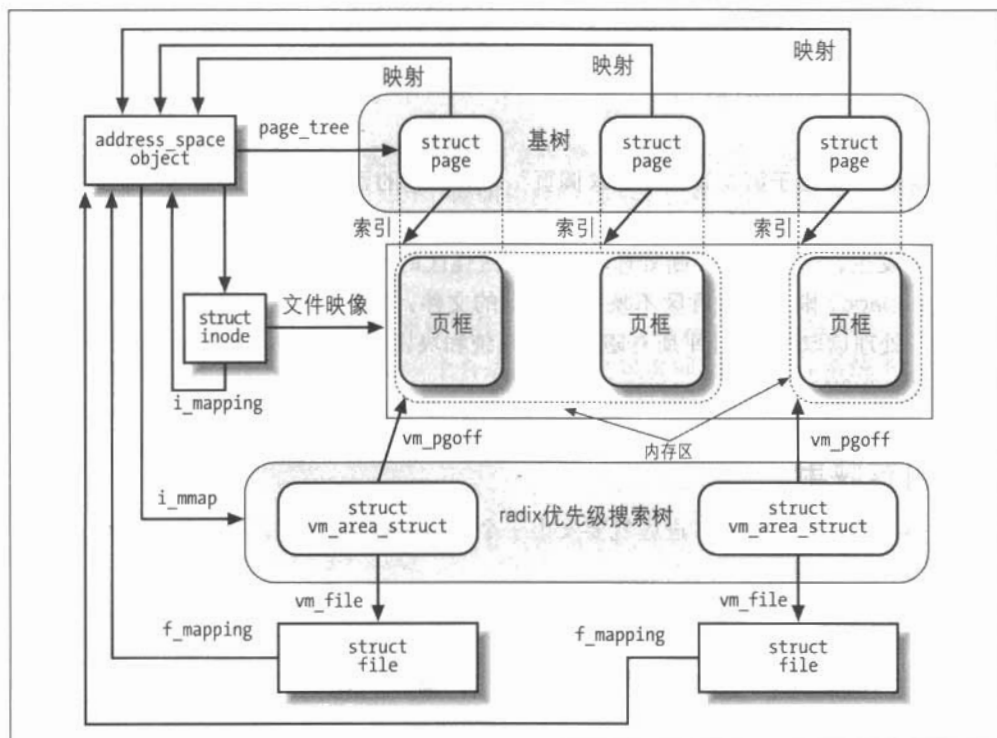


图 16-2: 文件内存映射的数据结构

每个线性区描述符都有一个 `vm_file` 字段，与所映射文件的文件对象链接（如果该字段为 `NULL`，则线性区没有用于内存映射）。第一个映射单元的位置存放在线性区描述符的 `vm_pgoff` 字段，它表示以页大小为单位的偏移量。所映射的文件那部分的长度就是线性区的大小，这可以从 `vm_start` 和 `vm_end` 字段计算出来。

共享内存映射的页通常都包含在页高速缓存中；私有内存映射的页只要还没有被修改，也都包含在页高速缓存中。当进程试图修改一个私有内存映射的页时，内核就把该页框进行复制，并在进程页表中用复制的页来替换原来的页框，这是第八章中介绍的写时复制机制的应用之一。虽然原来的页框还仍然在页高速缓存中，但不再属于这个内存映射，这是由于被复制的页框替换了原来的页框。依次类推，这个复制的页框不会被插入到页高速缓存中，因为其中所包含的数据不再是磁盘上表示那个文件的有效数据。

图16-2还显示了包含在页高速缓存中的几个指向内存映射文件的页的页描述符。注意图中的第一个线性区有三页，但是只为它分配了两个页框；猜想一下，大概是拥有这个线性区的进程从没有访问过第三页。

对每个不同的文件系统，内核提供了几个钩子（hook）函数来定制其内存映射机制。内存映射实现的核心委托给文件对象的 `mmap` 方法。对于大多数磁盘文件系统和块设备文件，这个方法是由叫做 `generic_file_mmap()` 的通用函数实现的，该函数将在下一节进行描述。

文件内存映射依赖于第九章的“请求调页”一节描述的请求调页机制。事实上，一个新建立的内存映射就是一个不包含任何页的线性区。当进程引用线性区中的一个地址时，缺页异常发生，缺页异常中断处理程序检查线性区的 `nopage` 方法是否被定义。如果没有定义 `nopage`，则说明线性区不映射磁盘上的文件；否则，进行映射，这个方法通过访问块设备处理读取的页。几乎所有磁盘文件系统和块设备文件都通过 `filemap_nopage()` 函数实现 `nopage` 方法。

创建内存映射

要创建一个新的内存映射，进程就要发出一个 `mmap()` 系统调用，并向该函数传递以下参数：

- 文件描述符，标识要映射的文件。
- 文件内的偏移量，指定要映射的文件部分的第一个字符。
- 要映射的文件部分的长度。

- 一组标志。进程必须显式地设置MAP_SHARED标志或MAP_PRIVATE标志来指定所请求的内存映射的种类（注6）。
- 一组权限，指定对线性区进行访问的一种或者多种权限：读访问（PROT_READ）、写访问（PROT_WRITE）或执行访问（PROT_EXEC）。
- 一个可选的线性地址，内核把该地址作为新线性区应该从哪里开始的一个线索。如果指定了MAP_FIXED标志，且内核不能从指定的线性地址开始分配新线性区，那么这个系统调用失败。

mmap()系统调用返回新线性区中第一个单元位置的线性地址。为了兼容起见，在80 × 86体系结构中，内核在系统调用表中为mmap()保留两个表项：一个在索引90处，一个在索引192处。前一个表项对应于old_mmap()服务例程（由老的C库使用），而后一个表项对应于sys_mmap2()服务例程（由新近的C库使用）。这两个服务例程仅在如何传递系统调用的第6个参数时有所差异。这两个服务例程都调用do_mmap_pgoff()函数（参见第九章“分配线性地址区间”一节）。我们现在就详细介绍当创建对文件进行映射的线性区时执行的步骤。我们所讨论的是do_mmap_pgoff()的file参数（文件对象指针）非空的情形。为清楚起见，我们要列举描述do_mmap_pgoff()的步骤，并指出在新条件下执行的其他步骤。

步骤1

检查是否为要映射的文件定义了mmap文件操作。如果没有，就返回一个错误码。文件操作表中的mmap值为NULL说明相应的文件不能被映射（例如，因为这是一个目录）。

步骤2

函数get_unmapped_area()调用文件对象的get_unmapped_area方法，如果已定义，就为文件的内存映射分配一个合适的线性地址区间。磁盘文件系统不会定义这个方法，那么像第九章“线性区的处理”一节描述的那样，get_unmapped_area()函数就调用内存描述符的get_unmapped_area方法。

步骤3

除了进行正常的一致性检查之外，还要对所请求的内存映射的种类（存放在mmap()

注6： 进程可以设置MAP_ANONYMOUS标志来指定新线性区是匿名的，也就是说，与任何基于磁盘的文件都无关（参见第九章的“请求调页”一节）。进程也可以创建具有MAP_SHARED和MAP_ANONYMOUS标志的线性区。在这种情况下，线性区在tmpfs文件系统中映射一个特殊的文件（参见第十九章的“IPC共享内存”一节），这个文件可以由创建进程的所有后代来访问。

系统调用的 `flags` 参数中) 与在打开文件时所指定的标志 (存放在 `file → f-mode` 字段中) 进行比较。根据这两个消息源, 执行以下的检查:

- 如果请求一个共享可写的内存映射, 就检查文件是为写入而打开的, 而不是以追加模式打开的 (`open()` 系统调用的 `O_APPEND` 标志)。
- 如果请求一个共享内存映射, 就检查文件上没有强制锁 (参见第十二章中的“文件加锁”一节)。
- 对于任何种类的内存映射, 都要检查文件是为读操作而打开的。

如果以上这些条件都不能满足, 就返回一个错误码。

另外, 当初始化非线性区描述符的 `vm_flags` 字段时, 要根据文件的访问权限和所请求的内存映射的种类设置 `VM_READ`、`VM_WRITE`、`VM_EXEC`、`VM_SHARED`、`VM_MAYREAD`、`VM_MAYWRITE`、`VM_MAYEXEC` 和 `VM_MAYSHARE` 标志 (参见第九章“线性区访问权限”一节)。最佳情况下, 对于不可写共享内存映射, 标志 `VM_SHARED` 和 `VM_MAYWRITE` 清 0。可以这样处理是因为不允许进程写入这个线性区的页, 因此, 这种映射的处理就与私有映射的处理相同。但是, 内核实际上允许共享该文件的其他进程读这个线性区中的页。

步骤 4

用文件对象的地址初始化线性区描述符的 `vm_file` 字段, 并增加文件的引用计数器。对映射的文件调用 `mmap` 方法, 将文件对象地址和线性区描述符地址作为参数传给它。对于大多数文件系统, 该方法由 `generic_file_mmap()` 实现, 它执行下列步骤:

- a. 将当前时间赋给文件索引节点对象的 `i_atime` 字段, 并将该索引节点标记为脏。
- b. 用 `generic_file_vm_ops` 表的地址初始化线性区描述符的 `vm_ops` 字段。在这个表中的方法, 除了 `nopage` 和 `populate` 方法外, 其他所有都为空。 `nopage` 方法由 `filemap_nopage()` 实现, 而 `populate` 方法由 `filemap_populate()` 实现 (参见本章后面的“非线性内存映射”一节)。

步骤 5

增加文件索引节点对象 `i_writecount` 字段的值, 该字段就是写进程的引用计数器。

撤消内存映射

当进程准备撤消一个内存映射时, 就调用 `munmap()`; 该系统调用还可用于减少每种内存区的大小。给它传递的参数如下:

- 要删除的线性地址区间中第一个单元的地址。
- 要删除的线性地址区间的长度。

该系统调用的 `sys_munmap()` 服务例程实际上是调用 `do_munmap()` 函数，该函数在第九章的“释放线性地址区间”一节已有描述。注意，不需要将待撤销可写共享内存映射中的页刷新到磁盘。实际上，因为这些页仍然在页高速缓存内，因此继续起磁盘高速缓存的作用。

内存映射的请求调页

出于效率的原因，内存映射创建之后并没有立即把页框分配给它，而是尽可能向后推迟到不能再推迟——也就是说，当进程试图对其中的一页进行寻址时，就产生一个“缺页”异常。

我们在第九章中的“缺页异常处理程序”一节中已经看到，内核是如何验证缺页所在的地址是否包含在某个进程的线性区中的。如果是这样，那么内核就检查这个地址所对应的页表项，如果表项为空就调用 `do_no_page()` 函数（参见第九章的“请求调页”一节）。

`do_no_page()` 函数执行对请求调页的所有类型都通用的操作，例如分配页框和更新页表。它还检查所涉及的线性区是否定义了 `nopage` 方法。在第九章的“请求调页”一节中，我们已经介绍了这个方法没有定义的情况（匿名线性区）。现在我们讨论当 `nopage` 方法被定义时，`do_no_page()` 所执行的主要操作：

1. 调用 `nopage` 方法，它返回包含所请求页的页框的地址。
2. 如果进程试图对页进行写入而该内存映射是私有的，则通过把刚读取的页拷贝一份并把它插入页的非活动链表中以避免进一步的“写时复制”异常（参见第十七章）。如果私有内存映射区域还没有一个包含新页的被动匿名线性区（slave anonymous memory region），它要么追加一个新的被动匿名线性区，要么增大现有的（参见第九章的“线性区”一节）。在下面的步骤中，该函数使用新页而不是 `nopage` 方法返回的页，所以后者不会被用户态进程修改。
3. 如果某个其他进程删改或作废了该页（`address_space` 描述符的 `truncate_count` 字段就是用于这种检查的），函数将跳回第 1 步，尝试再次获得该页。
4. 增加进程内存描述符的 `rss` 字段，表示一个新页框已分配给进程。
5. 用新页框的地址以及线性区的 `vm_page_prot` 字段中所包含的页访问权来设置缺页所在的地址对应的页表项。
6. 如果进程试图对这个页进行写入，则把页表项的 `Read/Write` 和 `Dirty` 位强制置为

1. 在这种情况下，或者把这个页框互斥地分配给进程，或者让页成为共享；在这两种情况下，都应该允许对这个页进行写入。

请求调页算法的核心在于线性区的 `nopage` 方法。一般来说，该方法必须返回进程所访问页所在的页框地址。其实现依赖于页所在线性区的种类。

在处理对磁盘文件进行映射的线性区时，`nopage` 方法必须首先在页高速缓存中查找所请求的页。如果没有找到相应的页，这个方法就必须将其从磁盘上读入。大部分文件系统都是使用 `filemap_nopage()` 函数来实现 `nopage` 方法的，该函数接收三个参数：

`area`

所请求页所在线性区的描述符地址。

`address`

所请求页的线性地址。

`type`

存放函数侦测到的缺页类型（`VM_FAULT_MAJOR` 或 `VM_FAULT_MINOR`）的变量的指针。

`filemap_nopage()` 函数执行以下步骤：

1. 从 `area->vm_file` 字段得到文件对象地址 `file`；从 `file->f_mapping` 得到 `address_space` 对象地址；从 `address_space` 对象的 `host` 字段得到索引节点对象地址。
2. 用 `area` 的 `vm_star` 和 `vm_pgoff` 字段来确定从 `address` 开始的页对应的数据在文件中的偏移量。
3. 检查文件偏移量是否大于文件大小。如果是，就返回 `NULL`，这就意味着分配新页失败，除非缺页是由调试程序通过 `ptrace()` 系统调用跟踪另一个进程引起的，我们打算讨论这种特殊情况。
4. 如果线性区的 `VM_RAND_READ` 标志置位（见下面），我们假定进程以随机方式读内存映射中的页，那么它忽略预读，跳到第 10 步。
5. 如果线性区的 `VM_SEQ_READ` 标志置位（见下面），我们假定进程以严格顺序方式读内存映射中的页，那么它调用 `page_cache_readahead()` 从缺页处开始预读（参见本章前面“文件的预读”一节）。
6. 调用 `find_get_page()`，在页高速缓存内存中寻找由 `address_space` 对象和文件偏移量标识的页。如果没找到这样的页，跳到第 11 步。

7. 如果函数运行至此,说明没在页高速缓存内找到页,检查内存区的VM_SEQ_READ标志:
 - a. 如果标志置位,内核将强行预读线性区中的页,从而预读算法失败,它就调用handle_ra_miss()来调整预读参数(参见本章前面“文件的预读”一节),并跳到第10步。
 - b. 否则,如果标志未置位,将文件file_ra_state描述符中的mmap_miss计数器加1。如果失败数远大于命中数(存放在mmap_hit计数器内),它将忽略预读,跳到第10步。
8. 如果预读没有永久禁止(file_ra_state描述符的ra_pages字段大于0),它将调用do_page_cache_readahead(),读入围绕请求页的一组页。
9. 调用find_get_page()来检查请求页是否在页高速缓存中,如果在,则跳到第11步。
10. 调用page_cache_read()。这个函数检查请求页是否在页高速缓存中,如果不在,则分配一个新页框,把它追加到页高速缓存,执行mapping->a_ops->readpage方法,安排一个I/O操作从磁盘读入该页内容。
11. 调用grab_swap_token()函数,尽可能为当前进程分配一个交换标记(参见第十七章“交换标记”一节)。
12. 请求页现已在页高速缓存内,将文件file_ra_state描述符的mmap_hit计数器加1。
13. 如果页不是最新的(标志PG_uptodate flag未置位),就调用lock_page()锁定该页,执行mapping->a_ops->readpage方法来触发I/O数据传输,调用wait_on_page_bit()后睡眠,一直等到该页被解锁,就是说等到数据传输完成。
14. 调用mark_page_accessed()来标记请求页为访问过(参见下一章)。
15. 如果在页高速缓存内找到该页的最新版,将*type设为VM_FAULT_MINOR,否则设为VM_FAULT_MAJOR。
16. 返回请求页地址。

用户态进程可以通过madvise()系统调用来调整filemap_nopage()函数的预读行为。MADV_RANDOM命令将线性区的VM_RAND_READ标志置位,从而指定以随机方式访问线性区的页。MADV_SEQUENTIAL命令将线性区的VM_SEQ_READ标志置位,从而指定以严格顺序方式访问页。最后,MADV_NORMAL命令将复位VM_RAND_READ和VM_SEQ_READ标志,从而指定以不确定的顺序访问页。

把内存映射的脏页刷新到磁盘

进程可以使用msync()系统调用把属于共享内存映射的脏页刷新到磁盘。这个系统调用

所接收的参数为：一个线性地址区间的起始地址、区间的长度以及具有下列含义的一组标志。

MS_SYNC

要求这个系统调用挂起进程，直到 I/O 操作完成为止。在这种方式中，调用进程就可以假设当系统调用完成时，这个内存映射中的所有页都已经被刷新到磁盘。

MS_ASYNC (对 MS_SYNC 的补充)

要求系统调用立即返回，而不用挂起调用进程。

MS_INVALIDATE

要求系统调用使同一文件的其他内存映射无效（没有真正实现，因为在 Linux 中无用）。

对线性地址区间中所包含的每个线性区，`sys_msync()` 服务例程都调用 `msync_interval()`。后者依次执行以下操作：

1. 如果线性区描述符的 `vm_file` 字段为 `NULL`，或者如果 `VM_SHARED` 标志清 0，就返回 0（说明这个线性区不是文件的可写共享内存映射）。
2. 调用 `filemap_sync()` 函数，该函数扫描包含在线性区中的线性地址区间所对应的页表项。对于找到的每个页，重设对应页表项的 `Dirty` 标志，调用 `flush_tlb_page()` 刷新相应的转换后援缓冲器（translation lookaside buffer, TLB）。然后设置页描述符中的 `PG-dirty` 标志，把页标记为脏。
3. 如果 `MS_ASYNC` 标志置位，它就返回。因此，`MS_ASYNC` 标志的实际作用就是将线性区的页标志 `PG_dirty` 置位。该系统调用并没有实际开始 I/O 数据传输。
4. 如果函数运行至此，则 `MS_SYNC` 标志置位，因此函数必须将内存区的页刷新到磁盘，而且，当前进程必须睡眠一直到所有 I/O 数据传输结束。为做到这一点，函数要得到文件索引节点的信号量 `i_sem`。
5. 调用 `filemap_fdatawrite()` 函数，该函数接收的参数为文件的 `address_space` 对象的地址。该函数必须用 `WB_SYNC_ALL` 同步模式建立一个 `writeback_control` 描述符，而且要检查地址空间是否有内置的 `writpages` 方法。如果有，则调用这个函数后返回。如果没有，就执行 `mpage_writpages()` 函数（参见本章前面“将脏页写到磁盘”一节）。
6. 检查文件对象的 `fsync` 方法是否已定义，如果是，就执行它。对于普通文件，这个方法限制自己把文件的索引节点对象刷新到磁盘。然而，对于块设备文件，这个方法调用 `sync_blockdev()`，它会激活该设备所有脏缓冲区的 I/O 数据传输。

7. 执行 `filemap_fdatawait()` 函数。在第十五章的“基树的标记”一节中讲过，页高速缓存中的基树标识了所有通过 `PAGECACHE_TAG_WRITEBACK` 标记正在往磁盘写的页。函数快速地扫描覆盖给定线性地址区间的这一部分基树来寻找 `PG_writeback` 标志置位的页。函数调用 `wait_on_page_bit()` 使其中每一页睡眠，一直到 `PG_writeback` 标志清 0，也就是等到正在进行的该页的 I/O 数据传输结束。
8. 释放文件的信号量 `i_sem` 并返回。

非线性内存映射

对普通文件，Linux 2.6 内核还提供一个访问方法，即非线性内存映射。非线性内存映射基本上还是前面所述的文件内存映射，但它的内存页映射的并不是文件的顺序页，而是每一内存页都映射的是文件数据的随机页（任意页）。

当然，一个用户态应用每次针对同一文件的不同 4096 字节部分重复调用 `mmap()` 系统调用，也可以得到同样的结果。然而，因为每个映射需要一个独立的线性区，所以这种方法对于大文件的非线性映射是非常低效的。

为了实现非线性映射，内核使用了另外的一些数据结构。首先，线性区描述符的 `VM_NONLINEAR` 标志用于表示线性区存在一个非线性映射。给定文件的所有非线性映射线性区描述符都存放在一个双向循环链表，该链表根植于 `address_space` 对象的 `i_mmap_nonlinear` 字段。

为创建一个非线性内存映射，用户态进程首先以 `mmap()` 系统调用创建一个常规的共享内存映射。应用然后调用 `remap_file_pages()` 来重新映射内存映射中的一些页。该系统调用的 `sys_remap_file_pages()` 服务例程有下面几个参数：

```
start
    调用进程共享文件内存映射区域内的线性地址
size
    文件重新映射部分的字节数
prot
    未用(必须为 0)
pgoff
    待映射文件初始页的页索引
flags
    控制非线性映射的标志
```

该服务例程用线性地址 `start`、页索引 `pgoff` 和映射尺寸 `size` 所确定的文件数据部分进行重新映射。如果线性区非共享或不能容纳要映射的所有页，则系统调用失败并返回错误码。实际上，该服务例程把线性区插入文件的 `i_mmap_nonlinear` 链表，并调用该线性区的 `populate` 方法。

对于所有普通文件，`populate` 方法是由 `filemap_populate()` 函数实现的。它执行以下步骤：

1. 检查 `remap_file_pages()` 系统调用的 `flags` 参数中 `MAP_NONBLOCK` 标志是否清 0。如果是，则调用 `do_page_cache_readahead()` 预读待映射文件的页。
2. 对重新映射的每一页，执行下列步骤：
 - a. 检查页描述符是否已在页高速缓存内，如果不在且 `MAP_NONBLOCK` 未置位，那从磁盘读入该页。
 - b. 如果页描述符在页高速缓存内，它将更新对应线性地址的页表项来指向该页框，并更新线性区描述符的页引用计数器。
 - c. 否则，如果没有在页高速缓存内找到该页描述符，它将文件页的偏移量存放在该线性地址对应的页表项的最高 32 位，并将页表项的 `Present` 位清 0、`Dirty` 位置位。

正如第九章“请求调页”一节所述，当处理请求调页错误时，`handle_pte_fault()` 函数检查页表项的 `Present` 和 `Dirty` 位。如果它们的值对应一个非线性内存映射，则 `handle_pte_fault()` 调用 `do_file_page()` 函数，从页表项的高位中取出所请求文件页的索引，然后，`do_file_page()` 函数调用线性区的 `populate` 方法从磁盘读入页并更新页表项本身。

因为非线性内存映射的内存页是按照相对于文件开始处的页索引存放在页高速缓存中，而不是按照相对于线性区开始处的索引存放的，所以非线性内存映射刷新到磁盘的方式与线性内存映射是一样的（参见本章前面“把内存映射的脏页刷新到磁盘”一节）。

直接 I/O 传送

我们已经看到，在 Linux 2.6 版本中，通过文件系统与通过引用基本块设备文件上的块，甚至与通过建立文件内存映射访问一个普通文件之间没有什么本质的差异。但是，还是有一些非常复杂的程序（自缓存的应用程序，*self-caching application*）更愿意具有控制 I/O 数据传送机制的全部权力。例如，考虑高性能数据库服务器：它们大都实现了自己的高速缓存机制，以充分挖掘对数据库独特的查询机制。对于这些类型的程序，内核页高速缓存毫无帮助；相反，因为以下原因它可能是有害的：

- 很多页框浪费在复制已在 RAM 中的磁盘数据上（在用户级磁盘高速缓存中）。
- 处理页高速缓存和预读的多余指令降低了 `read()` 和 `write()` 系统调用的执行效率，也降低了与文件内存映射相关的分页操作。
- `read()` 和 `write()` 系统调用不是在磁盘和用户存储器之间直接传送数据，而是分两次传送：在磁盘和内核缓冲区之间和在内核缓冲区与用户存储器之间。

因为必须通过中断和直接内存访问（DMA）处理块硬件设备，而且这只能在内核态完成，因此，最终需要某种内核支持来实现自缓存的应用程序。

Linux 提供了绕过页高速缓存的简单方法：直接 I/O 传送。在每次 I/O 直接传送中，内核对磁盘控制器进行编程，以便在自缓存的应用程序的用户态地址空间中的页与磁盘之间直接传送数据。

我们知道，任何数据传送都是异步进行的。当数据传送正在进行时，内核可能切换当前进程，CPU 可能返回到用户态，产生数据传送的进程的页可能被交换出去，等等。这对于普通 I/O 数据传送没有什么影响，因为它们涉及磁盘高速缓存中的页，磁盘高速缓存由内核拥有，不能被换出去，并且对内核态的所有进程都是可见的。

另一方面，直接 I/O 传送应当在给定进程的用户态地址空间的页内移动数据。内核必须当心这些页是由内核态的任一进程访问的，当数据传送正在进行时不能把它们交换出去。让我们看看这是如何实现的。

当自缓存的应用程序要直接访问文件时，它以 `O_DIRECT` 标志置位的方式打开文件（参见第十二章“`open()`系统调用”一节）。在运行 `open()` 系统调用时，`dentry_open()` 函数检查打开文件的 `address_space` 对象是否有已实现的 `direct_IO` 方法，如果没有则返回错误码。对一个已打开的文件也可以由 `fcntl()` 系统调用的 `F_SETFL` 命令把 `O_DIRECT` 置位。

让我们首先看第一种情况，这里自缓存应用程序对一个以 `O_DIRECT` 标志置位的方式打开的文件调用 `read()` 系统调用。在本章前面“从文件中读取数据”一节中提到过，文件的 `read` 方法通常是由 `generic_file_read()` 函数实现的，它初始化 `iovec` 和 `kiocb` 描述符并调用 `__generic_file_aio_read()`。后面这个函数检查 `iovec` 描述符描述的用户态缓冲区是否有效，然后检查文件的 `O_DIRECT` 标志是否置位。当被 `read()` 调用时，该函数执行的代码段实际上等效于下面的代码：

```
if (filp->f_flags & O_DIRECT) {
    if (count == 0 || *ppos > filp->f_mapping->host->i_size)
        return 0;
    retval = generic_file_direct_IO(READ, iocb, iov, *ppos, 1);
    if (retval > 0)
```

```

        *ppos += retval;
        file_accessed(filp);
        return retval;
    }

```

函数检查文件指针的当前值、文件大小与请求的字符数，然后调用 `generic_file_direct_IO()` 函数，传给它 `READ` 操作类型、`kiocb` 描述符、`iovec` 描述符、文件指针的当前值以及 `io_vec` 描述符中指定的用户态缓冲区号 (1)。当 `generic_file_direct_IO()` 结束时，`__generic_file_aio_read()` 更新文件指针，设置对文件索引节点的访问时间戳，然后返回。

对一个以 `O_DIRECT` 标志置位打开的文件调用 `write()` 系统调用时，情况类似。在本章前面“写入文件”一节讲到过，文件的 `write` 方法就是调用 `generic_file_aio_write_nolock()`。该函数检查 `O_DIRECT` 标志是否置位，如果置位，则调用 `generic_file_direct_IO()` 函数，而这次限定的是 `WRITE` 操作类型。

`generic_file_direct_IO()` 函数有以下参数：

`rw`

操作类型: `READ` 或 `WRITE`

`kiocb`

`kiocb` 描述符指针 (参见表 16-1)

`iovec`

`iovec` 描述符数组指针 (参见本章前面“从文件中读取数据”一节)

`offset`

文件偏移量

`nr_segs`

`iovec` 数组中 `iovec` 描述符数

`generic_file_direct_IO()` 函数的执行步骤如下：

1. 从 `kiocb` 描述符的 `ki_filp` 字段得到文件对象的地址 `file`，从 `file->f_mapping` 字段得到 `address_space` 对象的地址 `mapping`。
2. 如果操作类型为 `WRITE`，而且一个或多个进程已创建了与文件的某个部分关联的内存映射，那么它调用 `unmap_mapping_range()` 取消该文件所有页的内存映射。如果任何取消映射的页所对应的页表项，其 `Dirty` 位置位，则该函数也确保它在页高速缓存内的相应页被标记为脏。
3. 如果根植于 `mapping` 的基树不为空 (`mapping->nropages` 大于 0)，则调用

`filemap_fdatawrite()`和`filemap_fdatawait()`函数刷新所有脏页到磁盘，并等待I/O操作结束（参见本章前面“把内存映射的脏页刷新到磁盘”一节）。（即使自缓存应用程序是直接访问文件的，系统中还可能有通过页高速缓存访问文件的其他应用程序。为了避免数据的丢失，在启动直接I/O传送之前，磁盘映像要与页高速缓存进行同步）。

4. 调用 `mapping` 地址空间的 `direct_IO` 方法（参见下面的段落）。
5. 如果操作类型为 `WRITE`，则调用 `invalidate_inode_pages2()` 扫描 `mapping` 基树中的所有页并释放它们。该函数同时也清空指向这些页的用户态页表项。

大多数情况下，`direct_IO`方法都是`__blockdev_direct_IO()`函数的封装函数。这个函数相当复杂，它调用大量的辅助数据结构和函数，但是实际上它所执行的操作与本章所描述的操作一样：对存放在相应块中要读或写的数据进行拆分，确定数据在磁盘上的位置，并添加一个或多个用于描述要进行的I/O操作的`bio`描述符。当然，数据将被直接从`iov`数组中`iovec`描述符确定的用户态缓冲区读写。调用`submit_bio()`函数将`bio`描述符提交给通用块层（参见第十五章“向通用块层提交缓冲区首部”一节）。通常情况下，`__blockdev_direct_IO()`函数并不立即返回，而是等所有的直接I/O传送都已完成才返回；因此，一旦`read()`或`write()`系统调用返回，自缓存应用程序就可以安全地访问含有文件数据的缓冲区。

异步 I/O

POSIX 1003.1 标准为异步方式访问文件定义了一套库函数（如表 16-4 所示）。“异步”实际上就是：当用户态进程调用库函数读写文件时，一旦读写操作进入队列函数就结束，甚至有可能真正的I/O数据传输还没有开始。这样调用进程可以在数据正在传输时继续自己的运行。

表 16-4：异步 I/O 的 POSIX 库函数

函数	说明
<code>aio_read()</code>	从文件异步读数据
<code>aio_write()</code>	向文件异步写数据
<code>aio_fsync()</code>	请求刷新所有正在运行的异步 I/O 操作（不阻塞）
<code>aio_error()</code>	获得正在运行的异步 I/O 操作的错误代码
<code>aio_return()</code>	获得一个已完成异步 I/O 操作的返回码
<code>aio_cancel()</code>	取消正在运行的异步 I/O 操作
<code>aio_suspend()</code>	挂起进程直到至少其中一个正在运行的异步 I/O 操作完成

使用异步 I/O 很简单，应用程序还是通过 `open()` 系统调用打开文件，然后用描述请求操作的信息填充 `struct aiocb` 类型的控制块。`struct aiocb` 控制块最常用的字段有：

`aio_fildes`

文件的文件描述符（由 `open()` 系统调用返回）

`aio_buf`

文件数据的用户态缓冲区

`aio_nbytes`

待传输的字节数

`aio_offset`

读写操作在文件中的起始位置（与“同步”文件指针无关）

最后，应用程序将控制块地址传给 `aio_read()` 或 `aio_write()`。一旦请求的 I/O 数据传输已由系统库或内核送进队列，这两个函数就结束。应用程序稍后可以调用 `aio_error()` 检查正在运行的 I/O 操作的状态。如数据传输仍在进行当中，则返回 `EINPROGRESS`；如果成功完成，则返回 0；如果失败，则返回一个错误码。`aio_return()` 函数返回已完成异步 I/O 操作的有效读写字节数，或者如果失败，返回 -1。

Linux 2.6 中的异步 I/O

异步 I/O 可以由系统库实现而完全不需要内核支持。实际上 `aio_read()` 或 `aio_write()` 库函数克隆当前进程，让子进程调用同步的 `read()` 或 `write()` 系统调用，然后父进程结束 `aio_read()` 或 `aio_write()` 函数并继续程序的执行。因此，它不用等待由子进程启动的同步操作完成。但是，这个“穷人版”POSIX 函数比内核层实现的异步 I/O 要慢很多。

Linux 2.6 内核版运用一组系统调用实现异步 I/O。但在 Linux 2.6.11 中，这个功能还在实现中，异步 I/O 只能用于打开 `O_DIRECT` 标志置位的文件（见上一节）。表 16-5 列出了异步 I/O 的系统调用。

表 16-5：异步 I/O 的 Linux 系统调用

系统调用	说明
<code>io_setup()</code>	为当前进程初始化一个异步环境
<code>io_submit()</code>	提交一个或多个异步 I/O 操作
<code>io_getevents()</code>	获得正在运行的异步 I/O 操作的完成状态
<code>io_cancel()</code>	取消一个正在运行的异步 I/O 操作
<code>io_destroy()</code>	删除当前进程的异步环境

异步 I/O 环境

如果一个用户态进程调用 `io_submit()` 系统调用开始异步 I/O 操作，那么它必须预先创建一个异步 I/O 环境。

基本上，一个异步 I/O 环境（简称 AIO 环境）就是一组数据结构，这个数据结构用于跟踪进程请求的异步 I/O 操作的运行情况。每个 AIO 环境与一个 `kiocx` 对象关联，它存放了与该环境有关的所有信息。一个应用可以创建多个 AIO 环境。一个给定进程的所有的 `kiocx` 描述符存放在一个单向链表中，该链表位于内存描述符的 `iocx_list` 字段（参见第九章中的表 9-2）。

我们不再详细讨论 `kiocx` 对象。但是我们应当注意一个 `kiocx` 对象使用的重要的数据结构：AIO 环。

AIO 环是用户态进程中地址空间的内存缓冲区，它也可以由内核态的所有进程访问。`kiocx` 对象中的 `ring_info.mmap_base` 和 `ring_info.mmap_size` 字段分别存放 AIO 环的用户态起始地址和长度。`ring_info.ring_pages` 字段则存放有一个数组指针，该数组存放所有含 AIO 环的页框的描述符。

AIO 环实际上是一个环形缓冲区，内核用它来写正运行的异步 I/O 操作的完成报告。AIO 环的第一个字节有一个首部（`struct aio_ring` 数据结构），后面的所有字节是 `io_event` 数据结构，每个都表示一个已完成的异步 I/O 操作。因为 AIO 环的页映射至进程的用户态地址空间，应用可以直接检查正运行的异步 I/O 操作的情况，从而避免使用相对较慢的系统调用。

`io_setup()` 系统调用为调用进程创建一个新的 AIO 环境。它有两个参数：正在运行的异步 I/O 操作的最大数目（这将确定 AIO 环的大小）和一个存放环境句柄的变量指针。这个句柄也是 AIO 环的基地址。`sys_io_setup()` 服务例程实际上是调用 `do_mmap()` 为进程分配一个存放 AIO 环的新匿名线性区（参见第九章“分配线性地址区间”一节），然后创建和初始化描述该 AIO 环境的 `kiocx` 对象。

相反地，`io_destroy()` 系统调用删除 AIO 环境，还删除含有对应 AIO 环的匿名线性区。这个系统调用阻塞当前进程直到所有正在运行的异步 I/O 操作结束。

提交异步 I/O 操作

为开始异步 I/O 操作，应用要调用 `io_submit()` 系统调用。该系统调用有三个参数：

`ctx_id`

由 `io_setup()`（标识 AIO 环境）返回的句柄

`iocbpp`

`iocb` 类型描述符的指针数组的地址，其中描述符的每项描述一个异步 I/O 操作

`nr`

`iocbpp` 指向的数组长度

`iocb` 数据结构与 POSIX `aio_cb` 描述符有同样的字段 `aio_fildes`、`aio_buf`、`aio_nbytes`、`aio_offset`，另外还有 `aio_lio_opcode` 字段存放请求操作的类型（典型地有：`read`、`write` 或 `sync`）。

`sys_io_submit()` 服务例程执行下列步骤：

1. 验证 `iocb` 描述符数组的有效性。
2. 在内存描述符的 `ioctx_list` 字段所对应的链表中查找 `ctx_id` 句柄对应的 `kiocb` 对象。
3. 对数组中的每一个 `iocb` 描述符，执行下列子步骤：
 - a. 获得 `aio_fildes` 字段中的文件描述符对应的文件对象地址。
 - b. 为该 I/O 操作分配和初始化一个新的 `kiocb` 描述符。
 - c. 检查 AIO 环中是否有空闲位置来存放操作的完成情况。
 - d. 根据操作类型设置 `kiocb` 描述符的 `ki_retry` 方法（见下面）。
 - e. 执行 `aio_run_iocb()` 函数，它实际上调用 `ki_retry` 方法为相应的异步 I/O 操作启动数据传输。如果 `ki_retry` 方法返回 `-EIOCBRETRY`，则表示异步 I/O 操作已提交但还没有完全成功：稍后在这个 `kiocb` 上，`aio_run_iocb()` 函数会被再次调用（见下面）；否则，调用 `aio_complete()`，为异步 I/O 操作在 AIO 环境的环中追加完成事件。

如果异步 I/O 操作是一个读请求，那么对应 `kiocb` 描述符的 `ki_retry` 方法是由 `aio_pread()` 实现的。该函数实际上执行的是文件对象的 `aio_read` 方法，然后按照 `aio_read` 方法的返回值更新 `kiocb` 描述符的 `ki_buf` 和 `ki_left` 字段（参见本章前面的表 16-1）。最后 `aio_pread()` 返回从文件读入的有效字节数，或者，如果函数确定请求的字节没有传输完，则返回 `-EIOCBRETRY`。对于大部分文件系统，文件对象的 `aio_read` 方法就是调用 `__generic_file_aio_read()` 函数。假如文件的 `O_DIRECT` 标志置位，函数就调用 `generic_file_direct_IO()` 函数，这在上一节描述过。但在这种情况下，`__blockdev_direct_IO()` 函数不是阻塞当前进程使之等待 I/O 数据传输完毕，而是立即返回。因为异步 I/O 操作仍在运行，`aio_run_iocb()` 会被再次调用，而这一次

的调用者是 aio_wq 工作队列的 aio 内核线程。kiocb 描述符跟踪 I/O 数据传输的运行。终于所有数据传输完毕，将完成结果追加到 AIO 环。

类似地，如果异步 I/O 操作是一个写请求，那么对应 kioceb 描述符的 ki_retry 方法是由 aio_pwrite() 实现的。该函数实际上执行的是文件对象的 aio_write 方法，然后按照 aio_write 方法的返回值更新 kioceb 描述符的 ki_buf 和 ki_left 字段（参见本章前面的表 16-1）。最后 aio_pwrite() 返回写入文件的有效字节数，或者，如果函数确定请求的字节没有完全传输完，则返回 -EIOCBRETRY。对于大部分文件系统，文件对象的 aio_write 方法就是调用 generic_file_aio_write_nolock() 函数。假如文件的 O_DIRECT 标志置位，跟上面一样，函数就调用 generic_file_direct_IO() 函数。

第十七章



回收页框

在前面的章节中，我们说明了内核如何通过记录空闲和占用的页框来处理动态内存。我们还讨论了用户态的每个进程怎样拥有自己的线性地址空间，又是怎样以每次一页的方式得到所请求的内存，以及如何在万不得已时才把页框分配给进程。最后，我们也讨论了如何使用动态内存实现内存与磁盘高速缓存。

在本章，我们通过讨论页框的回收完成对虚拟内存子系统的描述。在第一节“页框回收算法”中，我们阐述了内核回收页框的原因与策略；然后在“反向映射”一节做一个技术补充，介绍了内核使用的一种数据结构，借助这个结构，内核可以快速定位指向同一个页框的所有页表项；而“PFRA 实现”一节则介绍 Linux 使用的页框回收算法；最后一节“交换”，几乎可以自成一章，这一节讲述了交换子系统，它是将匿名页（并非文件的映射数据）保存到磁盘的内核部件。

页框回收算法

Linux 中有一点很有意思，在为用户态进程与内核分配动态内存时，所作的检查是马马虎虎的。

比如，对单个用户所创建进程的 RAM 使用总量并不作严格检查（第三章的“进程资源限制”一节提到的限制只针对单个进程）；对内核使用的许多磁盘高速缓存和内存高速缓存大小也同样不作限制。

减少控制是一种设计选择，这使内核以最好的可行方式使用可用的 RAM。当系统负载较低时，RAM 的大部分由磁盘高速缓存占用，很少正在运行的进程可以从中获益。但是，

当系统负载增加时，RAM的大部分则由进程页占用，高速缓存就会缩小从而给后来的进程让出空间。

我们在前面的章节中看到，内存及磁盘高速缓存抓取了那么多的页框但从未释放任何页框。这是合理的，因为高速缓存系统并不知道进程是否（什么时候）会重新使用某些缓存的数据，因此不能确定高速缓存的哪些部分应该释放。此外，正是有了第九章描述的请求调页机制，只要用户态进程继续执行，它们就能获得页框；然而，请求调页没有办法强制进程释放不再使用的页框。

因此，迟早所有空闲内存将被分配给进程和高速缓存。Linux内核的页框回收算法（*page frame reclaiming algorithm, PFRA*）采取从用户态进程和内核高速缓存“窃取”页框的办法补充伙伴系统的空闲块列表。

实际上，在用完所有空闲内存之前，就必须执行页框回收算法。否则，内核很可能陷入一种内存请求的僵局中，并导致系统崩溃。也就是说，要释放一个页框，内核就必须把页框的数据写入磁盘；但是，为了完成这一操作，内核却要请求另一个页框（例如，为I/O数据传送分配缓冲区首部）。因为不存在空闲页框，因此，不可能释放页框。

页框回收算法的目标之一就是保存最少的空闲页框池以便内核可以安全地从“内存紧缺”的情形中恢复过来。

选择目标页

页框回收算法（*PFRA*）的目标就是获得页框并使之空闲。显然，*PFRA*选取的页框肯定不是空闲的，即这些页框原本不在伙伴系统的任何一个 `free_area` 数组中（参见第八章的“伙伴系统算法”一节）。

*PFRA*按照页框所含内容，以不同的方式处理页框。我们将它们区分成：不可回收页、可交换页、可同步页和可丢弃页，如表 17-1 所示。

表 17-1：PFRA处理的页框类型

页类型	说明	回收操作
不可回收页	空闲页（包含在伙伴系统列表中）	（不允许也无需回收）
	保留页（ <code>PG_reserved</code> 标志置位）	
	内核动态分配页	
	进程内核态堆栈页	
	临时锁定页（ <code>PG_locked</code> 标志置位）	
	内存锁定页（在线性区中且 <code>VM_LOCKED</code> 标志置位）	

表 17-1: PFRA 处理的页框类型 (续)

页类型	说明	回收操作
可交换页	用户态地址空间的匿名页 tmpfs 文件系统的映射页 (如 IPC 共享内存的页)	将页的内容保存在交换区
可同步页	用户态地址空间的映射页 存有磁盘文件数据且在页高速缓存中的页 块设备缓冲区页 某些磁盘高速缓存的页 (如索引节点高速缓存)	必要时, 与磁盘映像同步这些页
可丢弃页	内存高速缓存中的未使用页 (如 slab 分配器高速缓存) 目录项高速缓存的未使用页	无需操作

在表 17-1 中, 所谓“映射页”是指该页映射了一个文件的某个部分。比如, 属于文件内存映射的用户态地址空间中所有页都是映射页, 页高速缓存中的任何其他页也是映射页。映射页差不多都是可同步的: 为回收页框, 内核必须检查页是否为脏, 而且必要时将页的内容写到相应的磁盘文件中。

相反, 所谓的“匿名页”是指它属于一个进程的某匿名线性区 (例如, 进程的用户态堆和堆栈中的所有页为匿名页)。为回收页框, 内核必须将页中内容保存到一个专门的磁盘分区或磁盘文件, 叫做“交换区” (参见后面“交换”一节)。因此, 所有匿名页都是可交换的。

通常, 特殊文件系统中的页是不可回收的。唯一的例外是 *tmpfs* 特殊文件系统的页, 它可以被保存在交换区后被回收。在第十九章中我们将看到 *tmpfs* 特殊文件系统用于 IPC 共享内存机制。

当 PFRA 必须回收属于某进程用户态地址空间的页框时, 它必须考虑页框是否为共享的。共享页框属于多个用户态地址空间, 而非共享页框属于单个用户态地址空间。注意, 非共享页框可能属于几个轻量级进程, 这些进程使用同一个内存描述符。

当进程创建子进程时, 就建立了共享页框。正如第九章“写时复制”一节所述, 子进程页表都从父进程中复制过来的, 父子进程因此共享同一个页框。共享页框的另一个常见情形是: 一个或多个进程以共享内存映射的方式访问同一个文件 (参见第十六章的“内存映射”一节) (注 1)。

注 1: 应该注意的是, 尽管如此, 当一个单独的进程通过共享内存映射访问文件时, 相应的页对 PFRA 来说却是非共享的。同样, PFRA 可能把属于私有内存映射的页作为共享页 (例如: 两个文件读同一个文件的某个部分, 但都不修改这部分的数据所在页的内容)。

PFRA 设计

尽管很容易确定回收内存的候选页（粗略地说，任何属于磁盘和内存高速缓存的页，以及属于进程用户态地址空间的页），但是选择合适的目标页可能是内核设计中最精巧的问题。

事实上，对处理虚拟内存子系统的开发者来说，其最难的工作在于找到一种合适的算法，这种算法既能确保台式计算机有可接受的性能（在这种计算机上内存的需要是相当有限的，而对系统响应的要求则是十分严格的），也能确保像大型数据库服务器那样的高级计算机有可接受的性能（在这种计算机上对内存的需要则巨大无比）。

不幸的是，找到一种较佳的页框回收算法是一种相当经验性的工作，很少有理论的支持。这种情形类似于对决定进程动态优先级的因素进行评估：主要目的是调整参数以达到较好的性能，不要问太多的为什么。通常情况下，这仅仅是“让我们试一试这种方法，看看会发生什么……”这么回事。这种经验主义方法的负面效果就是代码变化快。因此我们无法保证：在你阅读本章时，这里讲的 Linux 2.6.11 使用的内存回收算法与 Linux 2.6 的最新版本中所使用的内存回收算法完全一致。但是，这里所讲的一般原则和主要的启发式准则还会继续使用。

一叶障目，不见泰山。因此，让我们先看看 PFRA 采用的几个总的原则，这些原则包含在本章后面介绍的几个函数中。

首先释放“无害”页

在进程用户态地址空间的页回收之前，必须先回收没有被任何进程使用的磁盘与内存高速缓存中的页。实际上，回收磁盘与内存高速缓存的页框并不需要修改任何页表项。我们在本章后面“最近最少使用（LRU）链表”一节会看到，在使用“交换倾向因子（swap tendency factor）”后，这个准则可以做出一些调整。

将用户态进程的所有页定为可回收页

除了锁定页，PFRA 必须能够窃得任何用户态进程页，包括匿名页。这样，睡眠较长时间的进程将逐渐失去所有页框。

同时取消引用一个共享页框的所有页表项的映射，就可以回收该共享页框

当 PFRA 要释放几个进程共享的页框时，它就清空引用该页框的所有页表项，然后回收该页框。

只回收“未用”页

使用简化的最近最少使用（Least Recently Used, LRU）置换算法，PFRA 将页分

为“在用 (*in_use*)”与“未用 (*unused*)”(注2)。如果某页很长时间没有被访问,那么它将来被访问的可能性较小,就可以将它看作未用;另一方面,如果某页最近被访问过,那么它将来被访问的可能性较大,就必须将它看作在用。PFRA只回收未用页。这就是第二章中“硬件高速缓存”一节所讲局部性原则的另一个应用。

LRU算法的主要思想就是用一个计数器来存放RAM中每一页的页年龄,即上一次访问该页到现在已经过的时间。这个计数器可使PFRA只回收任何进程的最旧页。一些计算机平台提供较为成熟的LRU算法(注3)。不幸的是,80x86处理器不提供这样的硬件功能,因此Linux内核不能依赖页计数器记录每页的页年龄。为解决这一问题, Linux使用每个页表项中的访问标志位(*Accessed*),在页被访问时,该标志位由硬件自动置位;而且,页年龄由页描述符在链表(两个不同的链表之一)中的位置来表示[参见本章后面“最近最少使用(LRU)链表”一节]。

因此,页框回收算法是几种启发式方法的混合:

- 谨慎选择检查高速缓存的顺序。
- 基于页年龄的变化排序(在释放最近访问的页之前,应当释放最近最少使用的页)。
- 区别对待不同状态的页(例如,不脏的页与脏页之间,最好把前者换出,因为前者不必写磁盘)。

反向映射

正如上一节所述,PFRA的目标之一是能释放共享页框。为达到这个目的, Linux 2.6内核能够快速定位指向同一页框的所有页表项。这个过程就叫做反向映射(*reverse mapping*)。

反向映射方法的简单解决之道,就是在页描述符中引入附加字段,从而将某页描述符所确定的页框中对应的所有页表项联接起来。但是,保持更新这样的链表将会大大增加系统开销,因此,就有更成熟的方法设计出来了。Linux 2.6就有叫做“面向对象的反向映射”的技术。实际上,对任何可回收的用户态页,内核保留系统中该页所在所有线性区(“对象”)的反向链接,每个线性区描述符存放一个指针指向一个内存描述符,而该内存描述符又包含一个指针指向一个页全局目录(*Page Global Directory*)。因此,这

注2: 还可以认为PFRA是“曾经使用过的”算法,它的思想来源于T.Johnson和D.Shasha在1994年提出的2Q缓冲区管理置换算法。

注3: 例如,一些大型机的CPU自动更新每个页表项中表示相应页年龄的计数器。

些反向链接使得PFRA能够检索引用某页的所有页表项。因为线性区描述符比页描述符少，所以更新共享页的反向链接就比较省时间。我们来看看这一方法是如何实现的。

首先，PFRA必须要确定待回收页是共享的或是非共享的，以及是映射页或是匿名页。为做到这一点，内核要查看页描述符的两个字段：`_mapcount`和`mapping`。

`_mapcount`字段存放引用页框的页表项数目。计数器的起始值为-1，这表示没有页表项引用该页框；如果值为0，就表示页是非共享的；而如果值大于0，则表示页是共享的。`page_mapcount`函数接收页描述符地址，返回值为`_mapcount+1`（这样，如返回值为1，表明是某个进程的用户态地址空间中存放的一个非共享页）。

页描述符的`mapping`字段用于确定页是映射的或匿名的。说明如下：

- 如果`mapping`字段空，则该页属于交换高速缓存（参见本章后面“交换高速缓存”一节）。
- 如果`mapping`字段非空，且最低位是1，表示该页为匿名页；同时`mapping`字段中存放的是指向`anon_vma`描述符的指针（参见下一节“匿名页的反向映射”）。
- 如果`mapping`字段非空，且最低位是0，表示该页为映射页；同时`mapping`字段指向对应文件的`address_space`对象（参见第十五章的“`address_space`对象”一节）。

Linux的`address_space`对象在RAM中是对齐的，所以其起始地址是4的倍数。因此其`mapping`字段的最低位可以用作一个标志位来表示该字段的指针是指向`address_space`对象还是`anon_vma`描述符。这是一个不规范的编程技巧，但内核要使用大量的页描述符，所以这些数据结构必须尽可能的小。`PageAnon()`函数接收页描述符地址作为参数，如果`mapping`字段的最低位置位，则函数返回1；否则返回0。

`try_to_unmap()`函数接收页描述符指针作为参数，尝试清空所有引用该页描述符对应页框的页表项。如果从页表项中成功清除所有对该页框的应用，函数返回`SWAP_SUCCESS`(0)；如果有些引用不能清除，函数返回`SWAP_AGAIN`(1)；如果出错，函数返回`SWAP_FAIL`(2)。这个函数很短：

```
int try_to_unmap(struct page *page)
{
    int ret;
    if (PageAnon(page))
        ret = try_to_unmap_anon(page);
    else
        ret = try_to_unmap_file(page);
    if (!page_mapped(page))
        ret = SWAP_SUCCESS;
    return ret;
}
```

函数 `try_to_unmap_anon()` 和 `try_to_unmap_file()` 分别处理匿名页和映射页。后面会对这两个函数加以说明。

匿名页的反向映射

匿名页经常是由几个进程共享的。最为常见的情形是：创建新进程，这在第九章中“写时复制”一节里已有描述，父进程的所有页框，包括匿名页，同时也分配给子进程。另外（但不常见），进程创建线性区时使用两个标志 `MAP_ANONYMOUS` 和 `MAP_SHARED`，表明这个区域内的页将由该进程后面的子进程共享。

将引用同一个页框的所有匿名页链接起来的策略非常简单，即将该页框所在的匿名线性区存放在一个双向循环链表中。要注意的是：即使一个匿名线性区存有不同页，也始终只有一个反向映射链表用于该区域中的所有页框。

当为一个匿名线性区分配第一页时，内核创建一个新的 `anon_vma` 数据结构，它只有两个字段：`lock` 和 `head`。`lock` 字段是竞争条件下保护链表的自旋锁；`head` 字段是线性区描述符双向循环链表的头部。然后，内核将匿名线性区的 `vm_area_struct` 描述符插入 `anon_vma` 链表。为实现这个目的，`vm_area_struct` 数据结构中包含有对应该链表的两个字段：`anon_vma_node` 和 `anon_vma`。`anon_vma_node` 字段存放指向链表中前一个和后一个元素的指针，而 `anon_vma` 字段指向 `anon_vma` 数据结构。最后，按前面所述，内核将 `anon_vma` 数据结构的地址存放在匿名页描述符的 `mapping` 字段。如图 17-1 所示。

当已被一个进程引用的页框插入另一个进程的页表项时（例如调用 `fork()` 系统调用时，参见第三章中“`clone()`、`fork()` 及 `vfork()` 系统调用”一节），内核只是将第二个进程的匿名线性区插入 `anon_vma` 数据结构的双向循环链表，而第一个进程线性区的 `anon_vma` 字段指向该 `anon_vma` 数据结构。因此每个 `anon_vma` 链表通常包含不同进程的线性区（注 4）。

如图 17-1 所示，借助 `anon_vma` 链表，内核可以快速定位引用同一匿名页框的所有页表项。实际上，每个区域描述符在 `vm_mfn` 字段中存放内存描述符地址，而该内存描述符又有一个 `pgd` 字段，其中存有进程的页全局目录。这样，页表项就可以从匿名页的起始线性地址得到，而该线性地址可以由线性区描述符以及页描述符的 `index` 字段得到。

注 4： `anon_vma` 的链表还可以包括同一个进程的几个相邻的匿名线性区。通常在系统调用 `mprotect()` 把一个匿名线性区划分成两个或更多个线性区时就会出现这种情况。

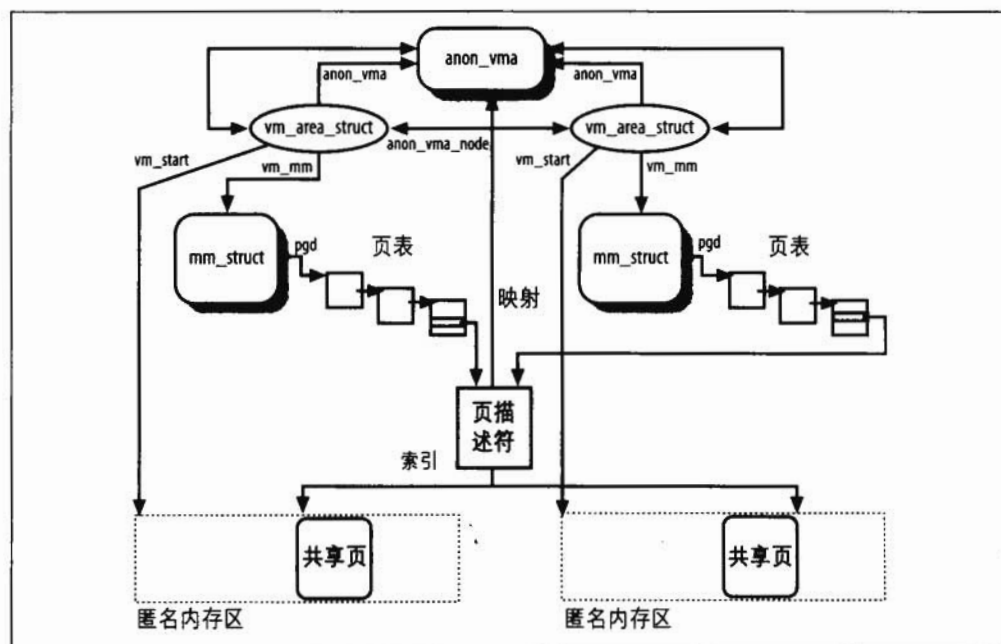


图 17-1: 匿名页的面向对象反向映射

try_to_unmap_anon()函数

当回收匿名页框时，PFRA 必须扫描 anon_vma 链表中的所有线性区，仔细检查是否每个区域都存有一个匿名页，而其对应的页框就是目标页框。这一工作就是通过 try_to_unmap_anon() 函数实现的，它接收目标页框描述符作为参数，执行的主要步骤如下：

1. 获得 anon_vma 数据结构的自旋锁，页描述符的 mapping 字段指向该数据结构。
2. 扫描线性区描述符的 anon_vma 链表。对该链表中的每一个 vma 线性区描述符，调用 try_to_unmap_one() 函数，传给它参数 vma 和页描述符（参见下面）。如果由于某种原因返回值为 SWAP_FAIL，或如果页描述符的 _mapcount 字段表明已找到所有引用该页框的页表项，那么停止扫描，而不用扫描到链表底部。
3. 释放第 1 步得到的自旋锁。
4. 返回最后调用 try_to_unmap_one() 函数得到的值：SWAP_AGAIN（部分成功）或 SWAP_FAIL（失败）。

try_to_unmap_one()函数

try_to_unmap_one()函数由try_to_unmap_anon()和try_to_unmap_file()重复调用。它有两个参数：page和vma。page是一个指向目标页描述符的指针；而vma是指向线性区描述符的指针。该函数执行的主要步骤如下：

1. 计算出待回收页的线性地址，所依据的参数有：线性区的起始线性地址（vma->vm_start）、被映射文件的线性区偏移量（vma->vm_pgoff）和被映射文件内的页偏移量（page->index）。对于匿名页，vma->vm_pgoff字段是0或者vm_start/PAGE_SIZE；相应地，page->index字段是区域内的页索引或是页的线性地址除以PAGE_SIZE。
2. 如果目标页是匿名页，则检查页的线性地址是否在线性区内。如果不是，则结束并返回SWAP_AGAIN（在介绍匿名页的反向映射时，我们讲过anon_vma链表可能存有不包含目标页的线性区）。
3. 从vma->vm_mm得到内存描述符地址，并获得保护页表的自旋锁vma->vm_mm->page_table_lock。
4. 成功调用pgd_offset()、pud_offset()、pmd_offset()和pte_offset_map()以获得对应目标页线性地址的页表项地址。
5. 执行一些检查来验证目标页可有效回收。下面的检查步骤中，如果任何一步失败，函数跳到第12步，结束并返回一个有关的错误码：SWAP_AGAIN或SWAP_FAIL。
 - a. 检查指向目标页的页表项。如果不成功，则函数返回SWAP_AGAIN。这可能在以下几种情形下发生：
 - 指向页框的页表项与COW关联，而vma标识的匿名线性区仍然属于原页框的anon_vma链表。
 - mremap()系统调用可重新映射线性区，并通过直接修改页表项将页移到用户态地址空间。这种特殊情况下，因为页描述符的index字段不能用于确定页的实际线性地址，所以面向对象的反向映射就不能使用了。
 - 文件内存映射是非线性的（参见第十六章的“非线性内存映射”一节）。
 - b. 验证线性区不是锁定（VM_LOCKED）或保留（VM_RESERVED）的。如果有锁定（VM_LOCKED）或保留情况之一出现，函数就返回SWAP_FAIL。
 - c. 验证页表项中的访问标志位（Accessed）被清0。如果没有，该函数将它清0，并返回SWAP_FAIL。访问标志位位置位表示页在用，因此不能被回收。
 - d. 检查页是否属于交换高速缓存（参见本章后面“交换高速缓存”一节），且此时

它正由 `get_user_pages()` 处理（参见第九章的“分配线性地址区间”一节）。在这种情形下，为避免恶性竞争条件，函数返回 `SWAP_FAIL`。

6. 页可以被回收。如果页表项的 `Dirty` 标志位置位，则将页的 `PG_dirty` 标志置位。
7. 清空页表项，刷新相应的 TLB。
8. 如果是匿名页，函数将换出页（swapped-out page）标识符插入页表项，以便将来访问时将该页换入（参见本章后面“交换”一节）。而且，递减存放在内存描述符 `anon_rss` 字段中的匿名页计数器。
9. 递减存放在内存描述符 `rss` 字段中的页框计数器。
10. 递减页描述符的 `_mapcount` 字段，因为对用户态页表项中页框的引用已被删除。
11. 递减存放在页描述符 `_count` 字段中的页框使用计数器。如果计数器变为负数，则从活动或非活动链表中删除页描述符[参见本章后面“最近最少使用 (LRU) 链表”一节]，而且调用 `free_hot_page()` 释放页框（参见第八章的“每 CPU 页框高速缓存”一节）。
12. 调用 `pte_unmap()` 释放临时内核映射，因为第 4 步中的 `pte_offset_map()` 可能分配了一个这样的映射（参见第八章的“高端内存页框的内核映射”一节）。
13. 释放第 3 步中获得的自旋锁 `vma->vm_mm->page_table_lock`。
14. 返回相应的错误码（成功时返回 `SWAP_AGAIN`）。

映射页的反向映射

与匿名页相比，映射页的面向对象反向映射所基于的思想很简单：我们总是可以获得指向一个给定页框的页表项，方法就是访问相应映射页所在的线性区描述符。因此，反向映射的关键就是一个精巧的数据结构，这个数据结构可以存放与给定页框有关的所有线性区描述符。

我们在上一节看到，匿名线性区描述符存放在双向循环链表中。获得引用给定页框的所有页表项，就是对该链表中的元素进行线性扫描。共享匿名页框的数量不是很大，因此这个方法工作得很好。

与匿名页相反，映射页经常是共享的，这是因为不同的进程常会共享同一个程序代码。例如，几乎所有进程都会共享包含标准 C 库代码的页（参见第二十章的“库”一节）。因此，Linux 2.6 依靠叫做“优先搜索树（priority search tree）”的结构来快速定位引用同一页框的所有线性区。

每个文件对应一个优先搜索树。它存放在 `address_space` 对象的 `i_mmap` 字段中，该 `address_space` 对象包含在文件的索引节点对象中。因为映射页描述符的 `mapping` 字段指向 `address_space` 对象，所以总是能够快速检索搜索树的根。

优先搜索树

Linux 2.6 使用的优先搜索树 (PST) 是基于 Edward McCreight 于 1985 年提出的一种数据结构，用于表示一组相互重叠的区间。McCreight 树是一个堆和对称搜索树的混合体，且用于对一个区间集进行查询。例如，“在一个给定区间内有哪些区间？”和“哪些区间与给定区间相交？”这种查询所花的时间与树的高度和结果区间的数量成正比。

PST 中的每一个区间相当于一个树的节点，它由基索引 (*radix index*) 和堆索引 (*heap index*) 两个索引来标识。基索引表示区间的起始点而堆索引表示终点。PST 实际上是一个依赖于基索引的搜索树，并附加一个类堆属性，即一个节点的堆索引不会小于其子节点的堆索引。

Linux 优先搜索树与 McCreight 数据结构的不同有两个重要方面：第一，Linux 树不总是对称的（对称算法要耗费很多的系统空间和执行时间）；第二，Linux 树被修改成存放线性区而不是线性区间。

每个线性区可以被看成是文件页的一个区间，并由在文件中的起始位置（基索引）和终点位置（堆索引）所确定。但是，线性区通常是从同一页开始（通常从页索引 0 开始）。不幸的是，McCreight 的原数据结构不能存放起始位置完全一样的区间。补充解决方案是：除了基索引和堆索引，PST 的每个节点附带一个大小索引 (*size index*)。该大小索引的值为线性区大小（页数）减 1。该大小索引使搜索程序能够区分同一起始文件位置的不同线性区。

然而，大小索引会大大增加不同的节点数，会使 PST 溢出。特别是，当有很多节点具有相同的基索引但堆索引不同时，PST 就无法全部容下它们。为了解决这个问题，PST 可以包括溢出子树 (*overflow subtree*)，该子树以 PST 的叶为根，且包含具有相同基索引的节点。

此外，不同进程拥有的线性区可能是映射了相同文件的相同部分（如上面提及的标准 C 库）。在这种情况下，对应这些区域的所有节点具有相同的基索引、堆索引和大小索引。当必须在 PST 中插入一个与现存某个节点具有相同索引值（基索引、堆索引和大小索引都相同）的线性区时，内核将该线性区描述符插入一个以原 PST 节点为根的双向循环列表。

图 17-2 所示是一个简单的优先搜索树。在图的左侧，我们看到有七个线性区覆盖着一个文件的前六页。每个区间都标有基索引、堆索引和大小索引。在图的右侧，则是对应的

PST。注意，子节点的堆索引都不大于相应父节点的堆索引。而且我们可以看到，任意一个节点的左子节点基索引也都不大于右子节点基索引，如果基索引相等，则按照大小索引排序。让我们假定：PFRA 搜索包含某页（索引为 5）的全部线性区。搜索算法从根（0, 5, 5）开始，因为相应区间包含该页，那么这就是得到的第一个线性区。然后算法搜索根的左子节点（0, 4, 4），比较堆索引（4）和页索引，因为堆索引较小，所以区间不包括该页。而且，有了 PST 的类堆属性，该节点的所有子节点都不包括该页。因此，算法直接跳到根的右子节点（2, 3, 5），其相应区间包含该页，因此得到这个区间。然后，算法搜索子节点（1, 2, 3）和（2, 0, 2），但它们都不包含该页。

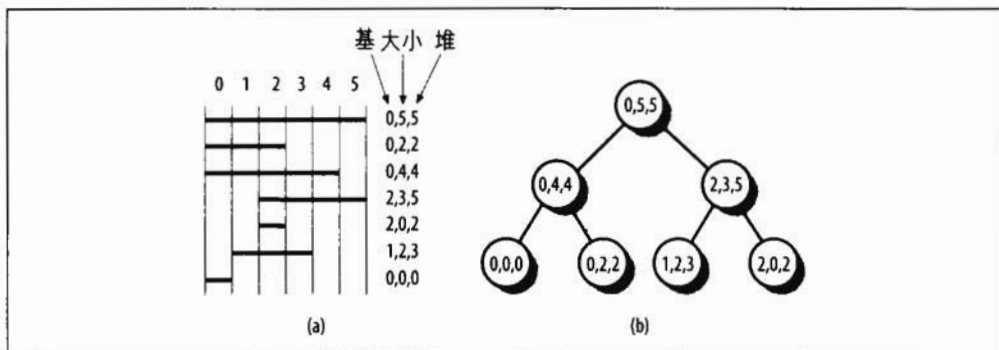


图 17-2: 简单的优先搜索树

因篇幅有限，我们对实现 Linux PST 的数据结构与函数无法作详尽阐述。我们只讨论由 `prio_tree_node` 数据结构表示的一个 PST 节点。该数据结构在每个线性区描述符的 `shared.prio_tree_node` 字段中。`shared.vm_set` 数据结构作为 `shared.prio_tree_node` 的替代品，可以用来将线性区描述符插入一个 PST 节点的链表副本。可以用 `vma_prio_tree_insert()` 和 `vma_prio_tree_remove()` 函数分别插入和删除 PST 节点。两个函数的参数都是线性区描述符地址与 PST 根地址。对 PST 的搜索可调用 `vma_prio_tree_foreach` 宏来实现，该宏循环搜索所有线性区描述符，这些描述符在给定范围的线性地址中包含至少一页。

try_to_unmap_file() 函数

`try_to_unmap_file()` 函数由 `try_to_unmap()` 调用，并执行映射页的反向映射。当为线性内存映射时，该函数就很容易描述（参见第十六章的“内存映射”一节）。这种情况下，它执行的步骤如下：

1. 获得 `page->mapping->i_mmap_lock` 自旋锁。
2. 对搜索树应用 `vma_prio_tree_foreach()` 宏，搜索树的根存放在 `page->mapping`

->i_mmap 字段。对宏发现的每个 vm_area_struct 描述符，函数调用 try_to_unmap_one()，尝试对该页所在的线性区页表项清 0（参见前面“匿名页的反向映射”一节）。如果由于某种原因，返回 SWAP_FAIL，或者如果页描述符的 _mapcount 字段表明引用该页框的所有页表项都已找到，则搜索过程马上结束。

3. 释放 page->mapping->i_mmap_lock 自旋锁。
4. 根据所有的页表项清 0 与否，返回 SWAP_AGAIN 或 SWAP_FAIL。

如果映射是非线性的（参见第十六章的“非线性内存映射”一节），那么 try_to_unmap_one() 函数可能无法清 0 某些页表项，这是因为页描述符的 index 字段（该字段存放文件中页的位置）不再对应线性区中的页位置，try_to_unmap_one() 函数就无法确定页的线性地址，也就无法得到页表项地址。

唯一的解决方法是对文件非线性线性区的穷尽搜索。双向链表以文件的所有非线性线性区的描述符所在的 page->mapping 文件的 address-space 对象的 i_mmap_nonlinear 字段为根。对每个这样的线性区，try_to_unmap_file() 函数调用 try_to_unmap_cluster()，而 try_to_unmap_cluster() 函数会扫描该线性区线性地址所对应的所有页表项，并尝试将它们清 0。

因为搜索可能很费时，所以执行有限扫描，而且通过试探法决定扫描线性区的哪一部分，vma_area_struct 描述符的 vm_private_data 字段存有当前扫描的当前指针。因此，try_to_unmap_file() 函数在某些情况下可能会找不到待停止映射的页。出现这种情况时，try_to_unmap() 函数发现页仍然是映射的，那么返回 SWAP_AGAIN 而不是 SWAP_SUCCESS。

PFRA 实现

页框回收算法必须处理多种属于用户态进程、磁盘高速缓存和内存高速缓存的页，而且必须遵照几条试探法准则。因此，PFRA 有很多函数也就不奇怪了。图 17-3 列出了 PFRA 的主要函数，箭头表示函数调用。例如，try_to_free_pages() 函数调用 shrink_caches()、shrink_slab() 和 out_of_memory() 三个函数。

正如你所看到的，PFRA 有几个入口（entry point）。实际上，页框回收算法的执行有三种基本情形：

内存紧缺回收

内核发现内存紧缺

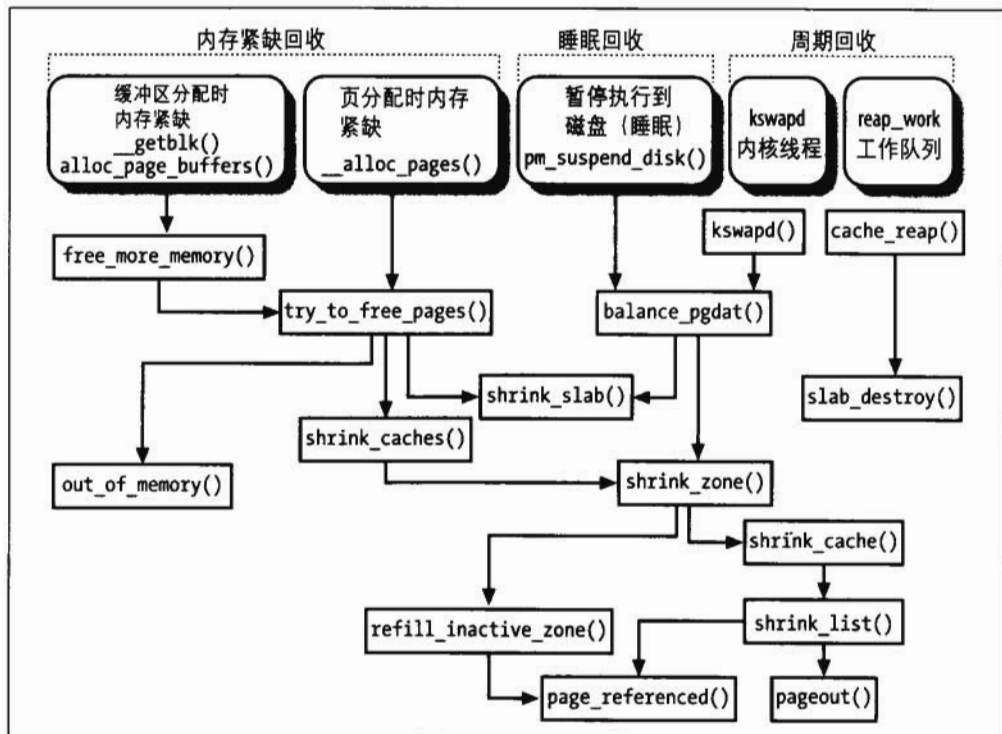


图 17-3: PFRA 的主要函数

睡眠回收

在进入 suspend-to-disk 状态时，内核必须释放内存（我们不再进一步讨论这种情形）

周期回收

必要时，周期性激活内核线程执行内存回收算法

内存紧缺回收在下列几种情形下激活：

- `grow_buffers()` 函数（由 `__getblk()` 调用）无法获得新的缓冲区页（参见第十五章的“在页高速缓存中搜索块”一节）。
- `alloc_page_buffers()` 函数（由 `create_empty_buffers()` 调用）无法获得页临时缓冲区首部（参见第十六章的“读写文件”一节）。
- `__alloc_pages()` 函数无法在给定的内存管理区（memory zone）中分配一组连续页框（参见第八章中“伙伴系统算法”一节）。

周期回收由下面两种不同的内核线程激活：

- *kswapd*内核线程，它检查某个内存管理区中空闲页框数是否已低于`pages_high`值的标高（参见后面的“周期回收”一节）。
- *events*内核线程，它是预定义工作队列的工作者线程（参见第四章的“工作队列”一节）；PFRA周期性地调度预定义工作队列中的一个任务执行，从而回收slab分配器处理的位于内存高速缓存中的所有空闲slab（参见第八章的“slab分配器”一节）。

我们现在详细讨论页框回收算法的各个部分，也包括图 17-3 中的所有函数。

最近最少使用（LRU）链表

属于进程用户态地址空间或页高速缓存的所有页被分成两组：活动链表与非活动链表。它们被统称为LRU链表。前面一个链表用于存放最近被访问过的页；后面的则存放有一段时间没有被访问过的页。显然，页必须从非活动链表中窃取。

页的活动链表和非活动链表是页框回收算法的核心数据结构。这两个双向链表的头分别存放在每个zone描述符（参见第八章的“内存管理区”一节）的`active_list`和`inactive_list`字段，而该描述符的`nr_active`和`nr_inactive`字段表示存放在两个链表中的页数。最后，`lru_lock`字段是一个自旋锁，保护两个链表免受SMP系统上的并发访问。

如果页属于LRU链表，则设置页描述符中的`PG_lru`标志。此外，如果页属于活动链表，则设置`PG_active`标志，而如果页属于非活动链表，则清除`PG_active`标志。页描述符的`lru`字段存放指向LRU链表中下一个元素和前一个元素的指针。

另外有几个辅助函数处理LRU链表：

```
add_page_to_active_list()
```

将页加入管理区的活动链表头部并递增管理区描述符的`nr_active`字段。

```
add_page_to_inactive_list()
```

将页加入管理区的非活动链表头部并递增管理区描述符的`nr_inactive`字段。

```
del_page_from_active_list()
```

从管理区的活动链表中删除页并递减管理区描述符的`nr_active`字段。

```
del_page_from_inactive_list()
```

从管理区的非活动链表中删除页并递减管理区描述符的`nr_inactive`字段。

```
del_page_from_lru()
```

检查页的`PG_active`标志。依据检查结果，将页从活动或非活动链表中删除，递减

管理区描述符的 `nr_active` 或 `nr_inactive` 字段，且如有必要，将 `PG_active` 标志清 0。

`activate_page()`

检查 `PG_active` 标志，如果未置位（页在非活动链表中），将页移到活动列表中，依次调用 `del_page_from_inactive_list()` 和 `add_page_to_active_list()`，最后将 `PG_active` 标志置位。在移动页之前，获得管理区的 `lru_lock` 自旋锁。

`lru_cache_add()`

如果页不在 LRU 链表中，将 `PG_lru` 标志置位，得到管理区的 `lru_lock` 自旋锁，调用 `add_page_to_inactive_list()` 把页插入管理区的非活动链表。

`lru_cache_add_active()`

如果页不在 LRU 链表中，将 `PG_lru` 和 `PG_active` 标志置位，得到管理区的 `lru_lock` 自旋锁，调用 `add_page_to_active_list()` 把页插入管理区的活动链表。

事实上，最后两个函数，`lru_cache_add()` 和 `lru_cache_add_active()` 稍有些复杂。这两个函数实际上并没有立刻把页移到 LRU，而是在 `pagevec` 类型的临时数据结构中聚集这些页，每个结构可以存放多达 14 个页描述符指针。只有当一个 `pagevec` 结构写满了，页才真正被移到 LRU 链表中。这种机制可以改善系统性能，这是因为只当 LRU 链表实际修改后才获得 LRU 自旋锁。

在 LRU 链表之间移动页

PFRA 把最近访问过的页集中放在活动链表中，以便当查找要回收的页框时不扫描这些页。相反，PFRA 把很长时间没有访问的页集中放在非活动链表中。当然，应该根据页是否正被访问，把页从非活动链表移到活动链表或者退回。

显然，两个状态（“活动”和“非活动”）是不足以描述所有可能的访问模式的。例如，假定日志进程每隔 1 小时把一些数据写入一个页中。尽管这个页是“不活动的”已经很长时间，但是访问使它变为“活动的”，因此即使这一页在整整 1 小时内没有被访问，也不回收相应的页框。当然，对这种问题并没有通用的解决方法，因为 PFRA 没有办法预测用户态进程的行为；不过，页不应该在每次单独的访问中就改变自己的状态似乎是合理的。

在页描述符中的 `PG_referenced` 标志用来把一个页从非活动链表移到活动链表所需的访问次数加倍；也把一个页从活动链表移到非活动链表所需的“丢失访问”次数加倍（见下面）。例如，假定在非活动链表中的一个页其 `PG_referenced` 标志置为 0。第一次访问把这个标志置为 1，但是这一页仍然留在非活动链表中。第二次对该页访问时发现这

一标志被设置，因此，把页移到活动链表。但是，如果第一次访问之后在给定的时间间隔内第二次访问没有发生，那么页框回收算法就可能重置 PG_referenced 标志。

如图17-4所示，PFRA使用 mark_page_accessed()、page_referenced() 和 refill_inactive_zone() 函数在 LRU 链表之间移动页。在图中，包含有页的 LRU 链表由 PG_active 标志的状态表示。

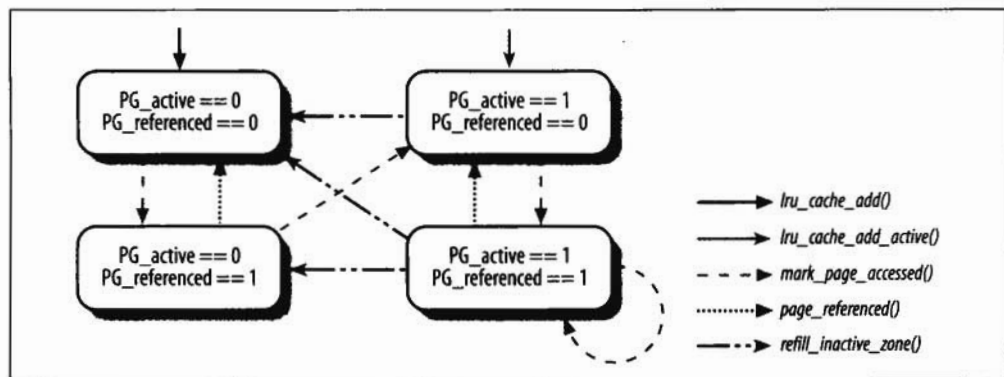


图 17-4: 在 LRU 链表之间移动页

mark_page_accessed() 函数

当内核必须把一个页标记为访问过时，就调用 mark_page_accessed() 函数。每当内核决定一个页是被用户态进程、文件系统层还是设备驱动程序引用时，这种情况就会发生。例如，在下列情况下调用 mark_page_accessed()：

- 当按需装入进程的一个匿名页时（由 do_anonymous_page() 函数执行；参见第九章“请求调页”一节）。
- 当按需装入内存映射文件的一个页时（由 filemap_nopage() 函数执行；参见第十六章“内存映射的请求调页”一节）。
- 当按需装入 IPC 共享内存区的一个页时（由 shmem_nopage() 函数执行；参见第十九章“IPC 共享内存”一节）。
- 当从文件读取数据页时（由 do_generic_file_read() 函数执行；参见第十六章“从文件中读取数据”一节）。
- 当换入一个页时（由 do_swap_page() 函数执行；参见后面的“换入页”一节）。
- 当在页高速缓存中搜索一个缓冲区页时（参见第十五章“在页高速缓存中搜索块”一节中介绍的 __find_get_block() 函数）。

mark_page_accessed()函数执行下列代码片段：

```
if (!PageActive(page) && PageReferenced(page) && PageLRU(page)) {
    activate_page(page);
    ClearPageReferenced(page);
} else if (!PageReferenced(page))
    SetPageReferenced(page);
```

如图 17-4 所示，该函数调用前，只有当 PG_referenced 标志置位，它才把页从非活动链表移到活动链表。

page_referenced()函数

PFRA 扫描一页调用一次 page_referenced()函数，如果 PG_referenced 标志或页表项中的某些 Accessed 标志位置位，则该函数返回 1，否则返回 0。该函数首先检查页描述符的 PG_referenced 标志。如果标志置位则清 0。然后使用面向对象的反向映射方法，对引用该页的所有用户态页表项中的 Accessed 标志位进行检查并清 0。为此，函数用到三个辅助函数：page_referenced_anon()、page_referenced_file()和 page_referenced_one()，这与本章前面“反向映射”一节中的 try_to_unmap_xxx()函数类似。page_referenced()函数还会用到交换标记 (swap token，参见本章后面“交换标记”一节)。

从活动链表到非活动链表移动页不是由 page_referenced()函数，而是由 refill_inactive_zone()函数实施的。实际上，refill_inactive_zone()函数除此之外还有其他很多功能，因此我们要进行深入的讨论。

refill_inactive_zone()函数

如图 17-3 所示，refill_inactive_zone()函数由 shrink_zone()调用，而 shrink_zone()函数对页高速缓存和用户态地址空间进行页回收（参见本章后面“内存紧缺回收”一节）。此函数有两个参数：zone 和 sc。指针 zone 指向一个内存管理区描述符，指针 sc 指向一个 scan_control 结构。PFRA 广泛使用 scan_control 这个数据结构，该结构存放着回收操作执行时的有关信息。表 17-2 中列出了它的字段。

表 17-2: scan_control 描述符的字段

类型	字段	说明
unsigned long	nr_to_scan	活动链表中待扫描的目标页数
unsigned long	nr_scanned	当前迭代中扫描过的非活动页数
unsigned long	nr_reclaimed	当前迭代中回收的页数
unsigned long	nr_mapped	用户态地址空间引用的页数

表 17-2: scan_control 描述符的字段 (续)

类型	字段	说明
int	nr_to_reclaim	待回收的目标页数
unsigned int	priority	扫描优先级, 范围从 12 到 0, 低优先级表示扫描更多的页
unsigned int	gfp_mask	调用进程传来的 GFP 掩码
int	may_writepage	如果置位, 则允许将脏页写到磁盘 (只针对便携情形)

refill_inactive_zone() 函数的工作至关重要, 因为, 从活动链表将页移到非活动链表就意味着页迟早要被 PFRA 捕获。如果函数的掠夺性过强, 就会有过多的页从活动链表被移动到非活动链表。因此, PFRA 就会回收大量的页框, 系统性能会受到影响。反过来, 如果函数太懒惰, 就没有足够的未用页来补充非活动链表, PFRA 就不能回收内存。为此, 该函数可以调整自己的行为: 开始时, 对每次调用, 扫描非活动链表中少量的页, 但是当 PFRA 很难回收内存时, refill_inactive_zone() 在每次调用时就逐渐增加扫描的活动页数。scan_control 数据结构中 priority 字段的值控制该函数的行为 (低值表示更紧迫的优先级)。

还有一个试探法可以调整 refill_inactive_zone() 函数行为。LRU 链表中有两类页: 属于用户态地址空间的页、不属于任何用户态进程且在页高速缓存中的页。如前所述, PFRA 倾向于压缩页高速缓存, 而将用户态进程的页留在 RAM 中。然而, 每一种策略中都没有一个固定的黄金法则保证系统的高性能, 所以 refill_inactive_zone() 函数使用交换倾向 (swap tendency) 经验值, 由它确定函数是移动所有的页还是只移动不属于用户态地址空间的页 (注 5)。函数按如下公式计算交换倾向值:

$$\text{交换倾向值} = \text{映射比率} / 2 + \text{负荷值} + \text{交换值}$$

映射比率 (mapped ratio) 是用户态地址空间所有内存管理区的页 (sc->nr_mapped) 占所有可分配页框数的百分比。mapped_ratio 的值大表示动态内存大部分用于用户态进程, 而值小则表示大部分用于页高速缓存。

负荷值 (distress) 用于表示 PFRA 在管理区中回收页框的效率。其依据是前一次 PFRA

注 5: “交换倾向”这一表述可能引起误解, 因为用户态地址空间的页可以是“可交换的”、“可同步的”以及“可丢弃的”。不过, 交换倾向值确实控制了 PFRA 实现的交换量, 因为几乎所有可交换的页都属于用户态地址空间。

运行时管理区的扫描优先级，这个优先级存放在管理区描述符的 `prev_priority` 字段。负荷值与管理区前一次优先级的对应关系如下：

管理区前一次优先级	12...7	6	5	4	3	2	1	0
负荷值	0	1	3	6	12	25	50	100

最后，交换值 (*swappiness*) 是一个用户定义常数，通常为60。系统管理员可以在 `/proc/sys/vm/swappiness` 文件内修改这个值，或用相应的 `sysctl()` 系统调用调整这个值。

只有当管理区交换倾向值大于等于100时，页才从进程地址空间回收。那么当系统管理员将交换值设为0时，PFRA就不会从用户态地址空间回收页，除非管理区的前一次优先级为0（这不大可能发生）。如果系统管理员将交换值设为100，那么PFRA每次调用该函数时都会从用户态地址空间回收页。

下面是 `refill_inactive_zone()` 函数执行步骤的一个简要说明：

1. 调用 `lru_add_drain()`，把仍留在 `pagevec` 数据结构中的所有页移入活动与非活动链表。
2. 获得 `zone->lru_lock` 自旋锁。
3. 对 `zone->active_list` 中的页进行首次扫描，从链表的底部开始向上，一直执行下去，直到链表为空或 `sc->nr_to_scan` 的页扫描完毕。在这一次循环中每扫描一页，函数就将引用计数器加1，从 `zone->active_list` 中删除页描述符，把它放在临时局部链表 `l_hold` 中。但是如果页框引用计数器是0，则将该页放回活动链表。实际上，引用计数器为0的页框一定属于管理区的伙伴系统，但释放页框时，首先递减使用计数器，然后将页框从LRU链表删除并插入伙伴系统链表。因此在一个很小的时间段，PFRA可能会发现LRU链表中的空闲页。
4. 把已扫描的活动页数追加到 `zone->pages_scanned`。
5. 从 `zone->nr_active` 中减去移入局部链表 `l_hold` 中的页数。
6. 释放 `zone->lru_lock` 自旋锁。
7. 计算交换倾向值（见上面）。
8. 对局部链表 `l_hold` 中的页运行第二次循环。这次循环的目的是：把其中的页分到两个子链表 `l_active` 和 `l_inactive` 中。属于某个进程用户态地址空间的页（即 `page->mapcount` 为非负数的页）被加入 `l_active` 的条件是：交换倾向值小于100，或者是匿名页但又没有激活的交换区，或者应用于该页的 `page_referenced()` 函

- 数返回正数（正数表示该页最近被访问过）。而任何其他情形下，页被加入 `l_inactive` 链表（注6）。
9. 获得 `zone->lru_lock` 自旋锁。
 10. 对局部链表 `l_inactive` 中的页执行第三次循环。把页移入 `zone->inactive_list` 链表，更新 `zone->nr_inactive` 字段，同时递减被移页框的使用计数器，从而抵消第3步中增加的值。
 11. 对局部链表 `l_active` 中的页执行第四次也是最后一次循环。把页移入 `zone->active_list` 链表，更新 `zone->nr_active` 字段，同时递减被移页框的使用计数器，从而抵消第3步中增加的值。
 12. 释放自旋锁 `zone->lru_lock` 并返回。

注意，`refill_inactive_zone()` 只检查用户态地址空间页的 `PG_referenced` 标志（见第8步）。相反的情况是，页在活动链表的底部，也就是较长时间以前被访问过，那么不大可能会在近期被访问。另外，如果页属于某个用户态进程且最近被使用过，那么函数也不会将页从活动链表删除。

内存紧缺回收

当内存分配失败时激活内存紧缺回收。在图 17-3 中，在分配 VFS 缓冲区或缓冲区首部时，内核调用 `free_more_memory()`；而当从伙伴系统分配一个或多个页框时，调用 `try_to_free_pages()`。

`free_more_memory()` 函数

`free_more_memory()` 函数执行如下操作：

1. 调用 `wakeup_bdflush()` 唤醒一个 `pdflush` 内核线程，并触发页高速缓存中 1024 个脏页的写操作（参见第十五章的“`pdflush` 内核线程”一节）。写脏页到磁盘的操作将最终使包含缓冲区、缓冲区首部和其他 VFS 数据结构的页框成为可释放的。
2. 调用 `sched_yield()` 系统调用的服务例程，为 `pdflush` 内核线程提供执行机会。
3. 对系统的所有内存节点，启动一个循环[参见第八章的“非一致内存访问(NUMA)”

注6： 注意，不属于任何用户态地址空间的页被移动到非活动链表中，但是由于它的 `PG_referenced` 标志没有清0，所以对该页的第一次访问导致函数 `mark_page_accessed()` 把该页移回活动链表中（参见图 17-4）。

一节]。对每一个节点，调用 `try_to_free_pages()` 函数，传给它的参数是一个“紧缺”内存管理区链表（在 80x86 体系结构中是 `ZONE_DMA` 和 `ZONE_NORMAL`，参见第八章的“内存管理区”一节）。

`try_to_free_pages()` 函数

`try_to_free_pages()` 函数接收如下三个参数：

`zones`

要回收的页所在的内存管理区链表（参见第八章的“内存管理区”一节）。

`gfp_mask`

用于失败的内存分配的一组分配标志（参见第八章的“分区页框分配器”一节）。

`order`

没有使用。

该函数的目标就是通过重复调用 `shrink_caches()` 和 `shrink_slab()` 函数释放至少 32 个页框，每次调用后优先级会比前一次提高。有关的辅助函数可以获得 `scan_control` 类型描述符中的优先级，以及正在进行的扫描操作的其他参数（见前面的表 17-2）。最低的、也是初始的优先级是 12，而最高的、也是最终的优先级是 0。如果 `try_to_free_pages()` 没能在某次（共 13 次）调用 `shrink_caches()` 和 `shrink_slab()` 函数时成功回收至少 32 个页框，PFRA 就要黔驴技穷了。最后一招：删除一个进程，释放它的所有页框。这一操作由 `out_of_memory()` 函数执行（参见本章后面“内存不足删除程序”一节）。

该函数主要执行如下步骤：

1. 分配和初始化一个 `scan_control` 描述符，具体说就是把分配掩码 `gfp_mask` 存入 `gfp_mask` 字段。
2. 对 `zones` 链表中的每个管理区，将管理区描述符的 `temp_priority` 字段设为初始优先级 12，而且计算管理区 LRU 链表中的总页数。
3. 从优先级 12 到 0，执行最多 13 次的循环，每次迭代执行如下子步骤：
 - a. 更新 `scan_control` 描述符的一些字段。具体地，把用户态进程的总页数存入 `nr_mapped` 字段，把本次迭代的当前优先级存入 `priority` 字段。而且将 `nr_scanned` 和 `nr_reclaimed` 字段设为 0。
 - b. 调用 `shrink_caches()`，传给它 `zones` 链表和 `scan_control` 描述符地址作为参数。这个函数扫描管理区的非活动页（见下面）。

- c. 调用 `shrink_slab()` 从可压缩内核高速缓存中回收页（参见后面“回收可压缩磁盘高速缓存的页”一节）。
 - d. 如果 `current->reclaim_state` 非空，则将 slab 分配器高速缓存中回收的页数（该数存放在一个由进程描述符字段指向的小型数据结构中）追加到 `scan_control` 描述符的 `nr_reclaimed` 字段。在调用 `try_to_free_pages()` 函数之前，`_alloc_pages()` 函数建立 `current->reclaim_state` 字段，并在结束后马上清除该字段（不可思议的是，`free_more_memory()` 不设置这个字段）。
 - e. 如果已达目标（`scan_control` 描述符的 `nr_reclaimed` 字段大于等于 32），则跳出循环到第 4 步。
 - f. 如果未达目标，但已扫描完成至少 49 页，函数则调用 `wakeup_bdflush()` 激活 `pdflush` 内核线程，并将页高速缓存中的一些脏页写入磁盘（参见第十五章的“搜索要刷新的脏页”一节）。
 - g. 如果函数已完成 4 次迭代而又未达目标，则调用 `blk_congestion_wait()` 挂起进程，一直到没有拥塞的 WRITE 请求队列或 100ms 超时已过（参见第十四章的“请求描述符”一节）。
4. 把每个管理区描述符的 `prev_priority` 字段设为上一次调用 `shrink_caches()` 使用的优先级，该值存放在管理区描述符的 `temp_priority` 字段。
 5. 如果成功回收则返回 1，否则返回 0。

shrink_caches() 函数

`shrink_caches()` 函数由 `try_to_free_pages()` 调用，它有两个参数：内存管理区链表 `zones` 和 `scan_control` 描述符地址 `sc`。

该函数的目的只是对 `zones` 链表中的每个管理区调用 `shrink_zone()` 函数。但对给定管理区调用 `shrink_zone()` 之前，`shrink_caches()` 函数用 `sc->priority` 字段的值更新管理区描述符的 `temp_priority` 字段，这就是扫描操作的当前优先级。而且如果 PFRA 的上一次调用优先级高于当前优先级，即这个管理区进行页框回收变得更难了，那么 `shrink_caches()` 把当前优先级拷贝到管理区描述符的 `prev_priority`。最后，如果管理区描述符中的 `all_unreclaimable` 标志置位，且当前优先级小于 12，则 `shrink_caches()` 不调用 `shrink_zone()`，也就是说，在 `try_to_free_pages()` 的第一迭代中不调用 `shrink_caches()`。当 PFRA 确定一个管理区都是不可回收页，扫描该管理区的页纯粹是浪费时间时，则将 `all_unreclaimable` 标志置位。

shrink_zone()函数

shrink_zone()函数有两个参数：zone和sc。zone是指向struct_zone描述符的指针；sc是指向scan_control描述符的指针。该函数的目标是从管理区非活动链表回收32页。它每次在更大的一段管理区非活动链表上重复调用辅助函数shrink_cache()，以期达到目标。而且shrink_zone()重复调用refill_inactive_zone()函数来补充管理区非活动链表[参见前面“最近最少使用(LRU)链表”一节]。

管理区描述符的nr_scan_active和nr_scan_inactive字段在这里起到很重要的作用。为提高效率，函数每批处理32页。因此如果函数在低优先级运行(对应sc->priority的高值)，且某个LRU链表中没有足够的页，函数就跳过对这个链表的扫描。但因此跳过的活动或不活动页数就分别存放在nr_scan_active或nr_scan_inactive中，这样函数下次执行时再处理这些跳过的页。

shrink_zone()函数的具体执行步骤如下：

1. 递增zone->nr_scan_active，增量是活动链表(zone->nr_active)的一小部分。实际增量取决于当前优先级，其范围是：zone->nr_active/2¹²到zone->nr_active/2⁰(即管理区内的总活动页数)。
2. 递增zone->nr_scan_inactive，增量是非活动链表(zone->nr_inactive)的一小部分。实际增量取决于当前优先级，其范围是：zone->nr_inactive/2¹²到zone->nr_inactive。
3. 如果zone->nr_scan_active字段大于等于32，函数就把该值赋给局部变量nr_active，并把该字段设为0，否则把nr_active设为0。
4. 如果zone->nr_scan_inactive字段大于等于32，函数就把该值赋给局部变量nr_inactive，并把该字段设为0，否则把nr_inactive设为0。
5. 设定scan_control描述符的sc->nr_to_reclaim字段为32。
6. 如果nr_active和nr_inactive都为0，则无事可做，函数结束。这不常见，用户态进程没有被分配到任何页时才可能出现这种情形。
7. 如果nr_active为正，则补充管理区非活动链表：

```
sc->nr_to_scan = min(nr_active, 32)
nr_active -= sc->nr_to_scan
refill_inactive_zone(zone, sc)
```

8. 如果nr_inactive为正，则尝试从非活动链表回收最多32页：

```
sc->nr_to_scan = min(nr_inactive, 32)
nr_inactive -= sc->nr_to_scan
shrink_cache(zone, sc)
```

9. 如果 `shrink_zone()` 成功回收 32 页（现在 `sc->nr_to_reclaim` 小于等于 0），则结束；否则，跳回第 6 步。

`shrink_cache()` 函数

`shrink_cache()` 函数又是一个辅助函数，它的主要目的就是管理区非活动链表取出一组页，把它们放入一个临时链表，然后调用 `shrink_list()` 函数对这个链表中的每一页进行有效的页框回收操作。`shrink_cache()` 函数的参数与 `shrink_zones()` 一样，都是 `zone` 和 `sc`，执行的主要步骤如下：

1. 调用 `lru_add_drain()`，把仍然在 `pagevec` 数据结构中的页移入活动与非活动链表[参见本章前面“最近最少使用 (LRU) 链表”一节]。
2. 获得 `zone->lru_lock` 自旋锁。
3. 处理非活动链表中的页（最多 32 页），对于每一页，函数递增使用计数器；检查该页是否不会被释放到伙伴系统（参见 `refill_inactive_zone()` 的第 3 步的讨论），把页从管理区非活动链表移入一个局部链表。
4. 把 `zone->nr_inactive` 计数器的值减去从非活动链表中删除的页数。
5. 递增 `zone->pages_scanned` 计数器的值，增量为在非活动链表中有效检查的页数。
6. 释放 `zone->lru_lock` 自旋锁。
7. 调用 `shrink_list()` 函数，传给它上面第 3 步中搜集的页（在局部链表中）。下面将详细讨论（你一定很期盼的讨论）。
8. 把 `sc->nr_to_reclaim` 字段的值减去由 `shrink_list()` 实际回收的页数。
9. 再次获取 `zone->lru_lock` 自旋锁。
10. 把局部链表中 `shrink_list()` 没有成功释放的页放回非活动或活动链表。注意，`shrink_list()` 有可能置位 `PG_active` 标志，从而将某页标记为活动页。这一操作使用 `pagevec` 数据结构对一组页进行处理[参见本章前面“最近最少使用 (LRU) 链表”一节]。
11. 如果函数扫描的页数至少是 `sc->nr_to_scan`，且如果没有成功回收目标页数（即 `sc->nr_to_reclaim` 仍然大于 0），则跳回第 3 步。
12. 释放 `zone->lru_lock` 自旋锁并结束。

`shrink_list()` 函数

我们现在讨论页框回收算法的核心部分。从 `try_to_free_pages()` 到 `shrink_cache()` 函数，前面所述这些函数的目的就是找到一组适合回收的候选页。`shrink_list()` 函数则从

参数 `page_list` 链表中尝试回收这些页，该函数的第二个参数 `sc` 是指向 `scan_control` 描述符的指针。当 `shrink_list()` 返回时，`page_list` 链表中剩下的是无法回收的页。

函数执行步骤如下：

1. 如果当前进程的 `need_resched` 字段置位，则调用 `schedule()`。
2. 执行一个循环，处理 `page_list` 链表中的每一页。对其中每个元素，从链表中删除页描述符并尝试回收该页框。如果由于某种原因页框不能释放，则把该页描述符插入一个局部链表。
3. 现在 `page_list` 已空，函数再把页描述符从局部链表移回 `page_list` 链表。
4. 递增 `sc->nr_reclaimed` 字段，增量为第 2 步中回收的页数，并返回这个数。

当然，`shrink_list()` 函数尝试回收页框的代码确实很有意思。图 17-5 是这段代码的流程图。

`shrink_list()` 处理的每个页框只可能有三种结果：

- 调用 `free_cold_page()` 函数，把页释放到管理区伙伴系统（参见第八章中“每 CPU 页框高速缓存”一节），因此被有效回收。
- 页没有被回收，因此被重新插入 `page_list` 链表。但是 `shrink_list()` 假设不久还能回收该页。因此函数让页描述符的 `PG_active` 标志保持清 0，这样页将被放回内存管理区的非活动链表（参见前面 `shrink_cache()` 函数描述的第 9 步）。这种情况对应于图 17-5 中标为“INACTIVE”的小方框。
- 页没有被回收，因此被重新插入 `page_list` 链表。但是，或是页正被使用，或是 `shrink_list()` 假设近期无法回收该页。函数将页描述符的 `PG_active` 标志置位，这样页将被放回内存管理区的活动链表。这种情况对应于图 17-5 中标为“ACTIVE”的小方框。

`shrink_list()` 函数不会去回收锁定页（`PG_locked` 置位）与写回页（`PG_writeback` 置位）。`shrink_list()` 调用 `page_referenced()` 函数检查该页是否最近被引用过，参见本章前面“最近最少使用（LRU）链表”一节中的描述。

要回收匿名页，就必须把它加入交换高速缓存，那么就必须在交换区为它保留一个新页槽（slot）。参见本章后面“交换”一节的详细讨论。

如果页在某个进程的用户态地址空间（页描述符的 `_mapcount` 字段大于等于 0），则 `shrink_list()` 调用 `try_to_unmap()` 寻找引用该页框的所有页表项（参见本章前面“反向映射”一节）。当然，只有当这个函数返回 `SWAP_SUCCESS` 时，回收才可继续。

如果是脏页，则写回磁盘前不能回收。为此，`shrink_list()`使用`pageout()`函数（后面会加以说明）。只有当`pageout()`不必进行写操作或写操作不久将结束时，回收才可继续。

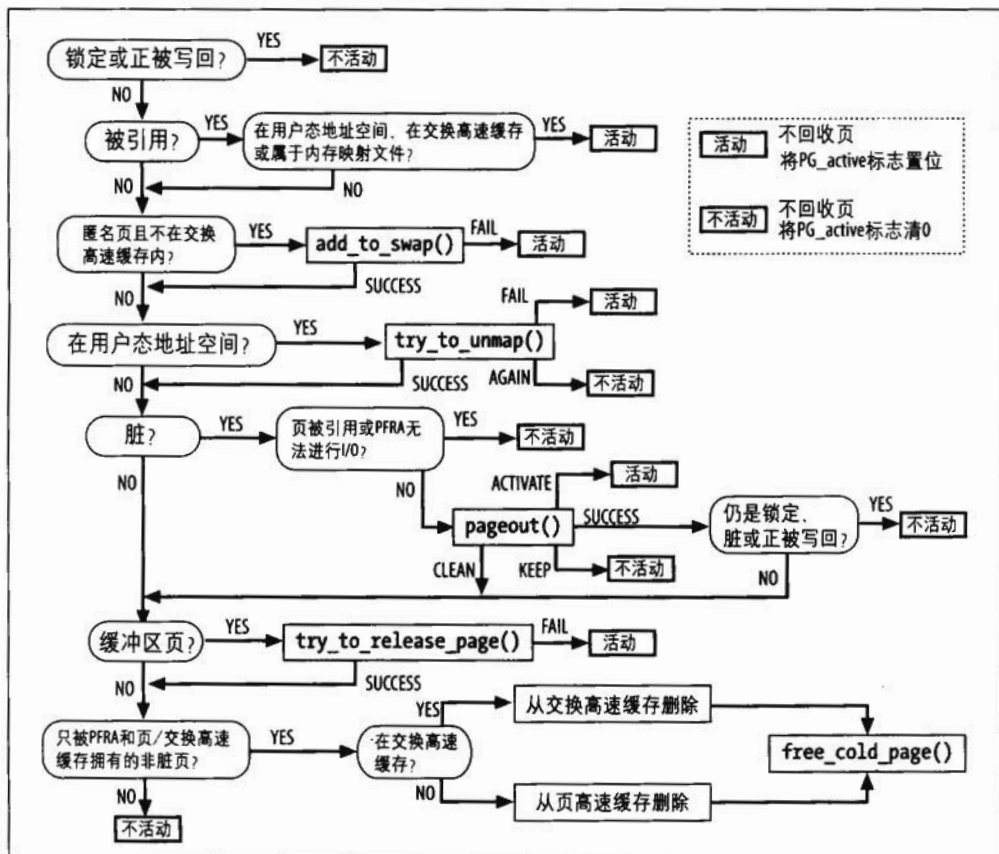


图 17-5: `shrink_list()`函数的页框回收流程

如果页包含 VFS 缓冲区，则 `shrink_list()`调用 `try_to_release_page()`释放关联的缓冲区首部（参见第十五章中“释放块设备缓冲区页”一节）。

最后，如果一切顺利，`shrink_list()`就检查页的引用计数器。若等于 2，那么这两个拥有者就是：页高速缓存（如果是匿名页，则为交换高速缓存）和 PFRA 自己（`shrink_cache()`函数中第 3 步中会递增引用计数器，参见前面）。这种情况下，如果页仍然不为脏，则页可以回收。为此，首先根据页描述符的 `PG_swapcache` 标志的值，从页高速缓存或交换高速缓存删除该页，然后，执行函数 `free_cold_page()`。

pageout()函数

当一个脏页必须写回磁盘时，`shrink_list()`调用`pageout()`函数。函数执行的主要步骤如下：

1. 检查页存放在页高速缓存还是交换高速缓存中（参见本章后面“交换高速缓存”一节）。进一步检查该页是否由页高速缓存（或交换高速缓存）与PFRA拥有。如果检查失败，则返回`PAGE_KEEP`（如果没有被`shrink_list()`回收，则写页到磁盘就没有意义了）。
2. 检查`address_space`对象的`writepage`方法是否已定义。如果没有，则返回`PAGE_ACTIVATE`。
3. 检查当前进程是否可以向块设备（与`address_space`对象对应）请求队列发出写请求。实际上，`kswapd`和`pdflush`内核线程总会发出写请求；而普通进程只有在请求队列不拥塞的情况下才能发出写请求，除非`current->backing_dev_info`字段指向块设备的`backing_dev_info`数据结构（参见第十六章“写入文件”一节中`generic_file_aio_write_nolock()`函数描述的第3步）。
4. 检查是否仍然是脏页。如果不是则返回`PAGE_CLEAN`。
5. 建立一个`writeback_control`描述符，调用`address_space`对象的`writepage`方法以启动一个写回操作（参见第十六章中“将脏页写到磁盘”一节）。
6. 如果`writepage`方法返回错误码，则函数返回`PAGE_ACTIVATE`。
7. 返回`PAGE_SUCCESS`。

回收可压缩磁盘高速缓存的页

我们从前面的章节中知道，内核在页高速缓存之外还使用其他磁盘高速缓存，例如，目录项高速缓存与索引节点高速缓存（参见第十二章“目录项高速缓存”）。当要回收其中的页框时，PFRA就必须检查这些磁盘高速缓存是否可压缩。

PFRA处理的每个磁盘高速缓存在初始化时必须注册一个`shrinker`函数。`shrinker`函数有两个参数：待回收页框数和一组GFP分配标志。函数按照要求从磁盘高速缓存回收页，然后返回仍然留在高速缓存内的可回收页数。

`set_shrinker()`函数向PFRA注册一个`shrinker`函数。该函数分配一个`shrinker`类型的描述符，在该描述符中存放`shrinker`函数的地址，然后把描述符插入一个全局链表，该链表存放在`shrinker_list`全局变量中。`set_shrinker()`函数还初始化`shrinker`描述符的`seeks`字段，通俗地说，这个字段表示：在高速缓存中的元素一旦被删除，那么重建一个所需的代价。

在 Linux 2.6.11 中，向 PFRA 注册的磁盘高速缓存很少。除了目录项高速缓存和索引节点高速缓存之外，注册 shrinker 函数的只有磁盘限额层、文件系统元信息块高速缓存（主要用于文件系统扩展属性）和 XFS 日志文件系统。

从可压缩磁盘高速缓存回收页的 PFRA 函数叫作 `shrink_slab()`（函数名有点误导，因为该函数与 slab 分配器高速缓存没什么关系）。它由 `try_to_free_pages()`（在前面“内存紧缺回收”一节中有描述）和 `balance_pgdat()` 调用（在后面的“周期回收”一节会有描述）。

对于从可压缩磁盘高速缓存回收的代价与及从 LRU 链表回收的代价（由 `shrink_list()` 执行）之间，`shrink_slab()` 函数试图作出一种权衡。实际上，函数扫描 shrinker 描述符的链表，调用这些 shrinker 函数并得到磁盘高速缓存中总的可回收页数。然后，函数再一次扫描 shrinker 描述符的链表，对于每个可压缩磁盘高速缓存，函数推算出待回收页框数。推算考虑的因素有：磁盘高速缓存中总的可回收页数、在磁盘高速缓存中重建一页的相关代价、LRU 链表中的页数。然后再调用 shrinker 函数尝试回收一组页（至少 128 页）。

因篇幅所限，我们只简单讨论目录项高速缓存和索引节点高速缓存的 shrinker 函数。

从目录项高速缓存回收页框

`shrink_dcache_memory()` 函数是目录项高速缓存的 shrinker 函数。它搜索高速缓存中的未用目录项对象，即没有被任何进程引用的目录项对象，然后将它们释放（参见第十二章的“目录项对象”一节）。

由于目录项高速缓存对象是通过 slab 分配器分配的，因此 `shrink_dcache_memory()` 函数可能导致一些 slab 变成空闲的，这样有些页框就可以被 `cache_reap()` 回收（参见本章后面“周期回收”一节）。此外，目录项高速缓存起索引节点高速缓存控制器的作用，因此，当一个目录项对象被释放时，存放相应索引节点对象的页就可以变为未用，而最终被释放。

`shrink_dcache_memory()` 函数接收两个参数：待回收页框数和 GFP 掩码。一开始，它检查 GFP 掩码中的 `__GFP_FS` 标志位是否清 0，如果是则返回 -1，因为释放目录项可能触发基于磁盘文件系统的操作。通过调用 `prune_dcache()`，就可以有效地进行页框回收。该函数扫描未用目录项链表（该链表的头部存放在 `dentry_unused` 变量中），一直到获得请求数量的释放对象或整个链表扫描完毕。对每个最近未被引用的对象，函数执行如下步骤：

1. 把目录项对象从目录项散列表、从其父目录中的目录项对象链表、从拥有者索引节点的目录项对象链表中删除。

2. 调用 `d_iput` 目录项方法（如果定义）或者 `iput()` 函数减少目录项的索引节点的引用计数器。
3. 调用目录项对象的 `d_release` 方法（如果定义）。
4. 调用 `call_rcu()` 函数以注册一个会删除目录项对象的回调函数[参见第五章“读-拷贝-更新 (RCU)”一节]，该回调函数又调用 `kmem_cache_free()` 把对象释放给 slab 分配器（参见第八章“从高速缓存中释放 slab”一节）。
5. 减少父目录的引用计数器。

最后，依据仍然留在目录项高速缓存中的未用目录项数，`shrink_dcache_memory()` 返回一个值。更准确地说，返回值是未用目录项数乘以 100 除以 `sysctl_vfs_cache_pressure` 全局变量的值。该变量的系统默认值是 100，因此返回值实际就是未用目录项数。但是通过修改文件 `/proc/sys/vm/vfs_cache_pressure` 或通过有关的 `sysctl()` 系统调用，系统管理员可以改变这个变量值。把值改为小于 100，则使 `shrink_slab()` 从目录项高速缓存（与索引节点高速缓存，见下一节）回收的页少于从 LRU 链表中回收的页。反之，如把值改为大于 100，则使 `shrink_slab()` 从目录项高速缓存回收的页多于从 LRU 链表中回收的页。

从索引节点高速缓存回收页框

`shrink_ichache_memory()` 函数被调用来从索引节点高速缓存删除未用索引节点对象。“未用”就是指索引节点不再有一个控制目录项对象。这个函数非常类似于刚描述的 `shrink_dcache_memory()`。它检查 `gfp_mask` 参数的 `__GFP_FS` 位，然后调用 `prune_ichache()`，最后与前面一样，依据仍然留在索引节点高速缓存中的未用索引节点数和 `sysctl_vfs_cache_pressure` 变量的值，返回一个值。

`prune_ichache()` 函数又扫描 `inode_unused` 链表（参见第十二章“索引节点对象”一节）。要释放一个索引节点，函数必须释放与该索引节点关联的任何私有缓冲区，它使页高速缓存内（引用该索引节点的）不再使用的干净页框无效，然后通过调用 `clear_inode()` 和 `destroy_inode()` 函数来删除索引节点对象。

周期回收

PFRA 用两种机制进行周期回收：`kswapd` 内核线程和 `cache_reap` 函数。前者调用 `shrink_zone()` 和 `shrink_slab()` 从 LRU 链表中回收页，后者则被周期性地调用以便从 slab 分配器中回收未用的 slab。

kswapd 内核线程

*kswapd*内核线程是激活内存回收的另外一种机制。为什么还需要这个内核线程呢？当空闲内存变得紧缺并且发出另一个内存分配请求时，调用 `try_to_free_pages()` 还不足够吗？

遗憾的是，实际情形并非如此。有些内存分配请求是由中断和异常处理程序执行的，它们不会阻塞等待释放页框的当前进程；还有，有些内存分配请求是由已经获得对临界资源互斥访问权限，因此就不能激活 I/O 数据传送的内核控制路径实现的。在极少数的情况下，所有的内存分配请求都是由这种内核控制路径完成的，因此内核将永远不能释放空闲内存。

*kswapd*利用机器空闲的时间保持内存空闲也对系统性能有良好的影响，进程因此能很快获得自己的页。

每个内存节点对应各自的 *kswapd* 内核线程[参见第八章中“非一致内存访问 (NUMA)”一节]。每个这样的线程通常睡眠在等待队列中，该等待队列以节点描述符的 `kswapd_wait` 字段为头部。但是，如果 `_alloc_pages()` 发现所有适合内存分配的内存管理区包含的空闲页框数低于“警告”阈值(一个依据内存管理区描述符的 `pages_low` 和 `protection` 字段推算出来的值)时，那么相应内存节点的 *kswapd* 内核线程被激活(参见第八章“管理区分配器”一节)。从本质上说，为了避免更多紧张的“内存紧缺”的情形，内核才开始回收页框。

正如第八章“保留的页框池”一节所述，每个管理区描述符还包括字段 `pages_min` 和 `pages_high`。前者表示必须保留的最小空闲页框数阈值；后者表示“安全”空闲页框数阈值，即空闲页框数大于该阈值时，应该停止页框回收。

kswapd 内核线程执行 `kswapd()` 函数。内核线程被初始化的内容是：把线程绑定到访问内存节点的 CPU；再把 `reclaim_state` 描述符地址存入进程描述符的 `current->reclaim_state` 字段(参见本章前面 `try_to_free_pages()` 函数的描述中的第 3d 步)；把 `current->flags` 字段的 `PF_MEMALLOC` 和 `PF_KSWAP` 标志置位，其含义是进程将回收内存，运行时允许使用全部可用空闲内存。每当 *kswapd* 内核线程被唤醒，`kswapd()` 函数执行下列主要操作：

1. 调用 `finish_wait()` 从节点的 `kswapd_wait` 等待队列删除内核线程(参见第三章中“如何组织进程”一节)。
2. 调用 `balance_pgdat()` 对 *kswapd* 的内存节点进行内存回收(见下面)。
3. 调用 `prepare_to_wait()` 把进程设成 `TASK_INTERRUPTIBLE` 状态，并让它在节点的 `kswapd_wait` 等待队列中睡眠。

4. 调用 `schedule()` 让 CPU 处理一些其他可运行进程。

`balance_pgdat()` 函数又执行下面的主要步骤：

1. 建立 `scan_control` 描述符（参见本章前面的表 17-2）。
2. 把内存节点的每个管理区描述符中的 `temp_priority` 字段设为 12（最低优先级）。
3. 执行一个循环，从 12 到 0 最多 13 次迭代。每次迭代执行下列子步骤：
 - a. 扫描内存管理区，寻找空闲页框数不足的最高管理区（从 `ZONE_DMA` 到 `ZONE_HIGHMEM`）。由 `zone_watermark_ok()` 函数进行每次的检测（参见第八章中“管理区分配器”一节的描述）。如果所有管理区都有大量空闲页框，则跳到第 4 步。
 - b. 对一部分管理区再一次进行扫描，范围是从 `ZONE_DMA` 到第 3a 步找到的管理区。对每个管理区，必要时用当前优先级更新管理区描述符的 `prev_priority` 字段，且连续调用 `shrink_zone()` 以回收管理区中的页（参见前面“内存紧缺回收”一节）。然后，调用 `shrink_slab()` 从可压缩磁盘高速缓存回收页（参见前面“回收可压缩磁盘高速缓存的页”一节）。
 - c. 如果已有至少 32 页被回收，则跳出循环至第 4 步。
4. 用各自 `temp_priority` 字段的值更新每个管理区描述符的 `prev_priority` 字段。
5. 如果仍有“内存紧缺”管理区存在，且如果进程的 `need_resched` 字段置位，则调用 `schedule()`。当再一次执行时，跳到第 1 步。
6. 返回回收的页数。

`cache_reap()` 函数

PFRA 还必须回收 slab 分配器高速缓存的页（参见第八章“内存区管理”一节）。为此，它使用 `cache_reap()` 函数，该函数周期性（差不多每两秒一次）地在预定事件工作队列（参见第四章“工作队列”一节）中被调度。它的地址存放在每 CPU 变量 `reap_work` 的 `func` 字段，该变量为 `work_struct` 类型。

`cache_reap()` 函数主要执行如下步骤：

1. 尝试获得 `cache_chain_sem` 信号量，该信号量保护 slab 高速缓存描述符链表。如果信号量已取得，就调用 `schedule_delayed_work()` 去调度该函数的下一次执行，然后结束。
2. 否则，扫描存放在 `cache_chain` 链表中的 `kmem_cache_t` 描述符。对找到的每一个高速缓存描述符，函数执行以下步骤：

- a. 如果高速缓存描述符的 `SLAB_NO_REAP` 标志置位，则页框回收被禁止，因此处理链表中的下一个高速缓存。
 - b. 清空局部 slab 高速缓存（参见第八章的“空闲 slab 对象的本地高速缓存”一节），则会有新的 slab 被释放。
 - c. 每个高速缓存都有“收割时间（reap time）”，该值存放在高速缓存描述符中 `kmem_list3` 结构的 `next_reap` 字段（参见第八章的“高速缓存描述符”一节）。如果 `jiffies` 值仍然小于 `next_reap`，则继续处理链表中的下一个高速缓存。
 - d. 把存放在 `next_reap` 字段的下一次“收割时间”设为：从现时起的 4s。
 - e. 在多处处理器系统中，函数清空 slab 共享高速缓存（参见第八章中“空闲 slab 对象的本地高速缓存”一节），那么会有新的 slab 被释放。
 - f. 如有新的 slab 最近被加入高速缓存，即高速缓存描述符中 `kmem_list3` 结构的 `free_touched` 标志置位，那么跳过这个高速缓存，继续处理链表中的下一个高速缓存。
 - g. 根据经验公式计算要释放的 slab 数量。基本上，这个数取决于高速缓存中空闲对象数的上限和能装入单个 slab 的对象数。
 - h. 对高速缓存空闲 slab 链表中的每个 slab，重复调用 `slab_destroy()`，一直到链表为空或者已回收目标数量的空闲 slab。
 - i. 调用 `cond_resched()` 检查当前进程的 `TIF_NEED_RESCHED` 标志，如果该标志置位，则调用 `schedule()`。
3. 释放 `cache_chain_sem` 信号量。
 4. 调用 `schedule_delayed_work()` 去调度该函数的下一次执行，然后结束。

内存不足删除程序

尽管 PFRA 尽量保留一定的空闲页框数，但虚拟内存子系统的压力可能变得很高，以至于所有可用内存耗尽。这很快会造成系统内的所有工作冻结。为满足一些紧迫请求，内核总试图释放内存，但是无法成功，这是因为交换区已满且所有磁盘高速缓存已被压缩。因此，没有进程可以继续执行，也就没有进程会释放它所拥有的页框。

为应对这种突发情况，PFRA 使用所谓的内存不足（*out of memory*, OOM）删除程序，该程序选择系统中的一个进程，强行删除它并释放页框。OOM 删除程序就像是外科大夫，为挽救一个人的生命而进行截肢。失去手脚当然是坏事，但这是不得已而为之。

当空闲内存十分紧缺且 PFRA 又无法成功回收任何页时，`__alloc_pages()` 调用 `out_of_`

`memory()` 函数（参见第八章中“管理区分配器”一节）。函数调用 `select_bad_process()` 在现有进程中选择一个“牺牲品”，然后调用 `oom_kill_process()` 删除该进程。

当然，`select_bad_process()` 并不是随机挑选进程的。被选进程应满足下列条件：

- 它必须拥有大量页框，从而可以释放出大量内存（为应对“子母弹”进程，函数计算母进程所属所有子进程的内存占用总量）。
- 删除它只损失少量工作成果（删除一个工作了几个小时或几天的批处理进程就不是个好主意）。
- 它应具有较低的静态优先级，用户通常给不太重要的进程赋予较低的优先级。
- 它不应是有 `root` 特权的进程，特权进程的工作通常比较重要。
- 它不应直接访问硬件块设备（如 X Window 服务器），因为硬件不能处在一个无法预知的状态。
- 它不能是 `swapper`（进程 0）、`init`（进程 1）和任何其他内核线程。

`select_bad_process()` 函数扫描系统中的每一个进程，根据以上准则用经验公式计算一个值，这个值表示选择这个进程的有利程度，然后返回最有利的被选进程描述符的地址。`out_of_memory()` 函数再调用 `oom_kill_process()` 并发出死亡信号（通常是 `SIGKILL`，参见第十一章），该信号发给该进程的一个子进程，或如果做不到，就发给该进程本身。`oom_kill_process()` 同时也删除与被选进程共享内存描述符的所有克隆进程。

交换标记

在阅读本章时，你可能认识到 Linux VM 子系统的代码太复杂，尤其是 PFRA，以致于无法预测任意负荷下它的行为。而且在有些情形下，VM 子系统表现出了一些病态行为。交换失效（*swap thrashing*）现象就是其中一例：当系统内存不足时，PFRA 全力把页写入磁盘以释放内存并从一些进程窃取相应的页框；而同时，这些进程要继续执行，也全力访问它们的页。因此内核把 PFRA 刚释放的页框又分配给这些进程，并从磁盘读回其内容。其结果就是页被无休止地写入磁盘并且再从磁盘读回。大部分的时间耗在访问磁盘上，从而没有进程能实质性地运行下去。

为减少交换失效的发生，一种由 Jiang 和 Zhang 在 2004 年提出的技术在内核版本 2.6.9 中得到实现。即把所谓的交换标记（*swap token*）赋给系统中的单个进程，该标记可以使该进程免于页框回收，所以进程可以实质性地运行，而且即使内存十分稀少，也有希望运行至结束。

交换标记的具体实现形式是 `swap_token_mm` 内存描述符指针。当进程拥有交换标记时，`swap_token_mm` 被设为进程内存描述符的地址。

页框回收算法的免除以如此简洁的方式实现了。我们在“最近最少使用 (LRU) 链表”一节看到，只当最近没有被引用时，一页才可从活动链表移入非活动链表。`page_referenced()` 函数进行这一检查。如果该页属于一个线性区，该区域所在进程拥有交换标记，那么该函数认可这个交换标记并返回 1 (被引用)。实际上，交换标记在几种情况下不予考虑：PFRA 代表一个拥有交换标记的进程运行，以及 PFRA 达到页框回收的最难优先级 (0 级)。

`grab_swap_token()` 函数决定是否将交换标记赋给当前进程。对每个主缺页 (major page fault) 调用该函数，这只有两种情形：

- 当 `filemap_nopage()` 函数发现请求页不在页高速缓存中时 (参见第十六章中“内存映射的请求调页”一节)。
- 当 `do_swap_page()` 函数从交换区读入一个新页时 (参见本章后面“换入页”一节)。

`grab_swap_token()` 函数在分配交换标记之前要进行一些检查，具体地说，就是要满足下列条件才可赋予交换标记：

- 上次调用 `grab_swap_token()` 后，至少已过了 2s。
- 在上一次调用 `grab_swap_token()` 后，当前拥有交换标记的进程没再提出主缺页，或该进程拥有交换标记的时间超出 `swap_token_default_timeout` 个节拍。
- 当前进程最近没有获得过交换标记。

交换标记的持有时间最好长一些，甚至以分钟为单位，因为其目标就是允许进程完成其执行。在 Linux 2.6.11 中，交换标记的持有时间默认值很小，即一个节拍。但是，通过编辑 `/proc/sys/vm/swap_token_default_timeout` 文件或发出相应的 `sysctl()` 系统调用，系统管理员可以修改 `swap_token_default_timeout` 变量的值。

当删除一个进程时，内核检查该进程是否拥有交换标记。如果是则放开它。这由 `mmapput()` 函数实现 (参见第九章的“内存描述符”一节)。

交换

交换 (swapping) 用来为非映射页在磁盘上提供备份。从前面的讨论我们知道有三类页必须由交换子系统处理：

- 属于进程匿名线性区（例如，用户态堆栈和堆）的页。
- 属于进程私有内存映射的脏页。
- 属于 IPC 共享内存区的页（参见第十九章的“IPC 共享内存”一节）。

就像请求调页，交换对于程序必须是透明的。换句话说，不需要在代码中嵌入与交换有关的特别指令。为了理解这是如何实现的，回想一下第二章的“常规分页”一节，我们知道每个页表项包含一个 Present 标志。内核利用这个标志来通知属于某个进程地址空间的页已被换出。在这个标志之外，Linux 还利用页表项中的其他位存放换出页标识符 (swapped-out page identifier)。该标识符用于编码换出页在磁盘上的位置。当缺页异常发生时，相应的异常处理程序可以检测到该页不在 RAM 中，然后调用函数从磁盘换入该缺页。

交换子系统的主要功能总结如下：

- 在磁盘上建立交换区 (swap area)，用于存放没有磁盘映像的页。
- 管理交换区空间。当需求发生时，分配与释放页槽 (page slot)。
- 提供函数用于从 RAM 中把页换出 (swap out) 到交换区或从交换区换入 (swap in) 到 RAM 中。
- 利用页表项 (现已被换出的换出页表项) 中的换出页标识符跟踪数据在交换区中的位置。

总之，交换是页框回收的一个最高级特性。如果我们要确保进程的所有页框都能被 PFRA 随意回收，而不仅仅是回收有磁盘映像的页，那么就必须使用交换。当然，你可以用 *swapon* 命令关闭交换，但此时随着磁盘系统负载增加，很快就會发生磁盘系统瘫痪。

我们还需指出，交换可以用来扩展内存地址空间，使之被用户态进程有效地使用。事实上，一个大交换区可允许内核运行几个大需求量的应用，它们的内存总需求量超过系统中安装的物理内存量。但是，就性能而言，RAM 的仿真还是比不上 RAM 本身。进程对当前换出页的每一次访问，与对 RAM 中页的访问比起来，要慢几个数量级。简而言之，如果性能重要，那么交换仅仅作为最后一个方案；为了解决不断增长的计算需求增加 RAM 芯片的容量仍然是一个最好的方法。

交换区

从内存中换出的页存放在交换区 (swap area) 中，交换区的实现可以使用自己的磁盘分区，也可以使用包含在大型分区中的文件。可以定义几种不同的交换区，最大个数由 MAX_SWAPFILES 宏（通常被设置成 32）确定。

如果有多个交换区，就允许系统管理员把大的交换空间分布在几个磁盘上，以使硬件可以并发操作这些交换区；这样处理还允许在系统运行时不用重新启动系统就可以扩大交换空间的大小。

每个交换区都由一组页槽 (*page slot*) 组成，也就是说，由一组 4096 字节大小的块组成，每块中包含一个换出的页。交换区的第一个页槽用来永久存放有关交换区的信息，其格式由 `swap_header` 联合体（由两个结构 `info` 和 `magic` 组成）来描述。`magic` 结构提供了一个字符串，用来把磁盘某部分明确地标记成交换区，它只含有一个字段 `magic.magic`，这个字段含有一个 10 字符的“magic”字符串。`magic` 结构从根本上允许内核明确地把一个文件或分区标记成交换区，这个字符串的内容就是“SWAPSPACE2”。该字段通常位于第一个页槽的末尾。

`info` 结构包括以下字段：

`bootbits`

交换算法不使用该字段。该字段对应于交换区的第一个 1024 字节，可以存放分区数据、磁盘标签等等。

`version`

交换算法的版本。

`last_page`

可有效使用的最后一个页槽。

`nr_badpages`

有缺陷的页槽的个数。

`padding[125]`

填充字节。

`badpages[1]`

一共 637 个数字，用来指定有缺陷页槽的位置。

创建与激活交换区

只要系统是打开的，存放在交换区中的数据就是有意义的。当系统被关闭时，所有的进程都被杀死，因此，进程存放在交换区中的数据也被丢弃。基于这个原因，交换区包含很少的控制信息，实际上包含交换区类型和有缺陷页槽的链表。这种控制信息很容易存放在一个单独的 4KB 页中。

通常，系统管理员在创建 Linux 系统中的其他分区时都创建一个交换分区，然后使用 `mkswap` 命令把这个磁盘区设置成一个新的交换区。该命令对刚才介绍的第一个页槽中

的字段进行初始化。由于磁盘中可能会有一些坏块，这个程序还可以对其他所有的页槽进行检查从而确定有缺陷页槽的位置。但是执行 `mkswap` 命令会把交换区设置成非激活的状态。每个交换区都可以在系统启动时在脚本文件中被激活，也可以在系统运行之后动态激活。

每个交换区由一个或多个交换子区 (*swap extent*) 组成，每个交换子区由一个 `swap_extent` 描述符表示，每个子区对应一组页（更准确地说，是一组页槽），它们在磁盘上是物理相邻的。`swap_extent` 描述符由下面这几部分组成：交换区的子区首页索引、子区的页数和子区的起始磁盘扇区号。当激活交换区自身的同时，组成交换区的有序子区链表也被创建。存放在磁盘分区中的交换区只有一个子区；但是，存放在普通文件中的交换区则可能有多个子区，这是因为文件系统有可能没把该文件全部分配在磁盘的一组连续块中。

如何在交换区中分布页

当换出时，内核尽力把换出的页存放在相邻的页槽中，从而减少在访问交换区时磁盘的寻道时间，这是高效交换算法的一个重要因素。

但是，如果系统使用了多个交换区，事情就变得更加复杂了。快速交换区（也就是存放在快速磁盘中的交换区）可以获得比较高的优先级。当查找一个空闲页槽时，要从优先级最高的交换区中开始搜索。如果优先级最高的交换区不止一个，为了避免超负荷地使用其中一个，应该循环选择相同优先级的交换区。如果在优先级最高的交换区中没有找到空闲页槽，就在优先级次高的交换区中继续进行搜索，依此类推。

交换区描述符

每个活动的交换区在内存中都有自己的 `swap_info_struct` 描述符，其字段如表 17-3 所示。

表 17-3: 交换区描述符的字段

类型	字段	说明
unsigned int	flags	交换区标志
spinlock_t	sdev_lock	保护交换区的自旋锁
struct file *	swap_file	指针，指向存放交换区的普通文件或设备文件的文件对象
struct block_device *	bdev	存放交换区的块设备描述符
struct list head	extent_list	组成交换区的子区链表的头部

表 17-3: 交换区描述符的字段 (续)

类型	字段	说明
int	nr_extents	组成交换区的子区数量
struct swap_extent *	curr_swap_extent	指向最近使用的子区描述符的指针
unsigned int	old_block_size	存放交换区的磁盘分区自然块大小
unsigned short *	swap_map	指向计数器数组的指针, 交换区的每个页槽对应一个数组元素
unsigned int	lowest_bit	在搜索一个空闲页槽时要扫描的第一个页槽
unsigned int	highest_bit	在搜索一个空闲页槽时要扫描的最后一个页槽
unsigned int	cluster_next	在搜索一个空闲页槽时要扫描的下一个页槽
unsigned int	cluster_nr	在从头重新开始扫描之前空闲页槽的分配次数
int	prio	交换区优先级
int	pages	可用页槽的个数
unsigned long	max	交换区的大小, 以页为单位
unsigned long	inuse_pages	交换区内已用页槽数
int	next	指向下一个交换区描述符的指针

flags 字段包括三个重叠的子字段:

SWP_USED

如果交换区是活动的, 该值就是 1; 如果交换区不是活动的, 该值就是 0。

SWP_WRITEOK

如果可以写入交换区, 该值就是 1; 如果交换区只读, 该值就是 0 (可以是活动的或不是活动的)。

SWP_ACTIVE

这个两位的字段实际上是 SWP_USED 和 SWP_WRITEOK 的组合。如果前面两个标志置位, 那么 SWP_ACTIVE 标志置位。

swap_map 字段指向一个计数器数组, 交换区的每个页槽对应一个元素。如果计数器值等于 0, 那么这个页槽就是空闲的; 如果计数器为正数, 那么换出页就填充了这个页槽。实际上, 页槽计数器的值就表示共享换出页的进程数。如果计数器的值为 SWAP_MAP_MAX (等于 32767), 那么存放在这个页槽中的页就是“永久”的, 并且不能从相应的页槽中

删除。如果计数器的值是 `SWAP_MAP_BAD` (等于 32768), 那么就认为这个页槽是有缺陷的, 也就是不可用的 (注 7)。

`prio` 字段是一个有符号的整数, 表示交换子系统依据这个值考虑每个交换区的次序。

`sdev_lock` 字段是一个自旋锁, 它防止 SMP 系统上对交换区数据结构 (主要是交换描述符) 的并发访问。

`swap_info` 数组包括 `MAX_SWAPFILES` 个交换区描述符。只有那些设置了 `SWP_USED` 标志的交换区才被使用, 因为它们是活动区域。图 17-6 说明了 `swap_info` 数组、一个交换区和相应的计数器数组的情况。

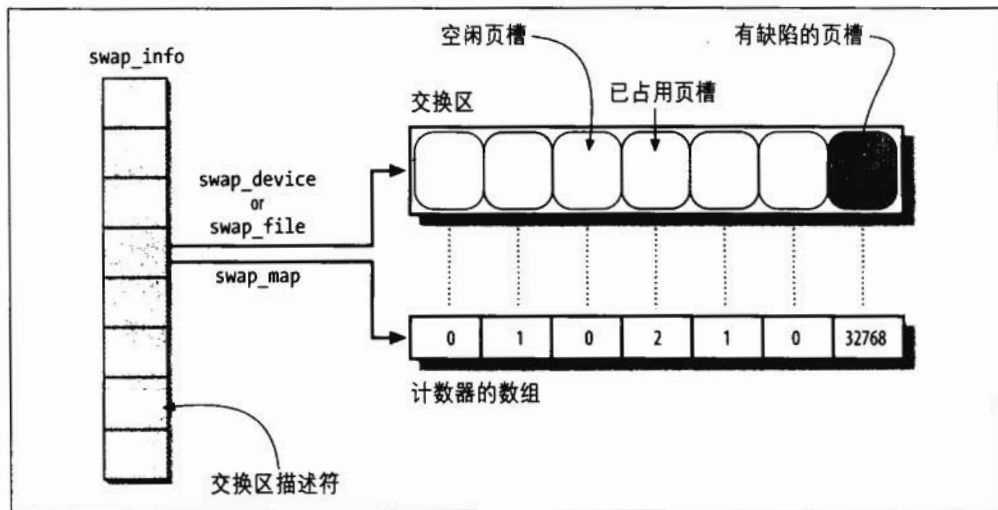


图 17-6: 交换区数据结构

`nr_swapfiles` 变量存放数组中包含或已包含所使用交换区描述符的最后一个元素的索引。这个变量有些名不符实, 它并没有包含活动交换区的个数。

活动交换区描述符也被插入按交换区优先级排序的链表中。该链表是通过交换区描述符的 `next` 字段实现的, `next` 字段存放的是 `swap_info` 数组中下一个描述符的索引。该字段作为索引的这种用法与我们已经见过的很多名为 `next` 字段的用法有所不同, 后者通常都是指针。

注 7: “永久”页槽防止 `swap_map` 计数器溢出。没有这些“永久”页槽, 如果有效的页槽被多次引用, 它们就会变得“有缺陷”, 从而导致数据丢失。但是, 谁也不真正期望一个页槽计数器达到 32768。这仅仅是一条权宜之计。

swap_list_t 类型的 swap_list 变量包括以下字段：

head

第一个链表元素在 swap_info 数组中的下标。

next

为换出页所选中的下一个交换区的描述符在 swap_info 数组中的下标。该字段用于在具有空闲页槽的最大优先级的交换区之间实现轮询算法。

swaponlock 自旋锁防止在多处理器系统对链表的并发访问。

交换区描述符的 max 字段存放以页为单位交换区的大小，而 pages 字段存放可用页槽的数目。这两个数字之所以不同是因为 pages 字段并没有考虑第一个页槽和有缺陷的页槽。

最后，nr_swap_pages 变量包含所有活动交换区中可用的（空闲并且无缺陷）页槽数目，而 total_swap_pages 变量包含无缺陷页槽的总数。

换出页标识符

可以很简单地而又唯一地标识一个换出页，这是通过在 swap_info 数组中指定交换区的索引和在交换区内指定页槽的索引实现的。由于交换区的第一个页（索引为 0）留给 swap_header 联合体，第一个可用页槽的索引就为 1。换出页标识符的格式如图 17-7 所示。

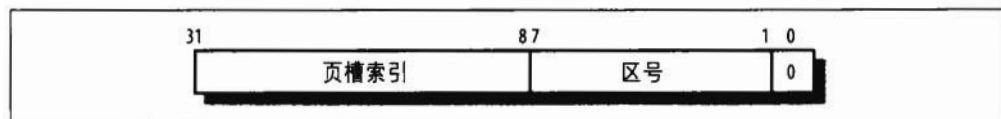


图 17-7：换出页标识符

swp_entry (type, offset) 宏负责从交换区索引 type 和页槽索引 offset 中构造换出页标识符。swp_type 和 swp_offset 宏正好相反，它们分别从换出页标识符中提取出交换区索引和页槽索引。

当页被换出时，其标识符就作为页的表项插入页表中，这样在需要时就可以再找到这个页。要注意这种标识符的最低位与 Present 标志对应，通常被清除来说明该页目前不在 RAM 中。但是，剩余 31 位中至少有一位被置位，因为没有页存放在交换区 0 的页槽 0 中。这样就可以从一个页表项中区分三种不同的情况：

空项

该页不属于进程的地址空间，或相应的页框还没有分配给进程（请求调页）。

前31个最高位不全等于0，最后一位等于0

该页现在被换出。

最低位等于1

该页包含在RAM中。

注意，交换区的最大值由表示页槽的可用位数决定。在80x86体系结构上，有24位可用，这就限制了交换区的大小为 2^{24} 个页槽（也就是64GB）。

由于一个页可以属于几个进程的地址空间（参见前面的“反向映射”一节），所以它可能从一个进程的地址空间中被换出，但是仍旧保留在内存中；因此可能把同一个页换出多次。当然，一个页在物理上只被换出并存储一次，但是后来每次试图换出该页都会增加 `swap_map` 计数器的值。

在试图换出一个已经换出的页时就会调用 `swap_duplicate()` 函数。该函数只是验证以参数传递的换出页标识符是否有效，并增加相应的 `swap_map` 计数器的值。更确切地说，该函数执行以下操作：

1. 使用 `swp_type` 和 `swp_offset` 宏从参数中提取出交换区号 `type` 和页槽索引 `offset`。
2. 检查交换区是否被激活；如果不是，则返回 0（无效的标识符）。
3. 检查页槽是否有效且是否不为空闲（`swap_map` 计数器大于 0 且小于 `SWAP_MAP_BAD`）；如果不是，则返回 0（无效的标识符）。
4. 否则，换出页的标识符确定出一个有效页的位置。如果页槽的 `swap_map` 计数器还没有达到 `SWAP_MAP_MAX`，则增加它的值。
5. 返回 1（有效的标识符）。

激活和禁用交换区

一旦交换区被初始化，超级用户（或者更确切地说是任何具有 `CAP_SYS_ADMIN` 权能的用户，有关内容将在第二十章中的“进程的信任状和权能”一节中介绍）就可以分别使用 `swapon` 和 `swapoff` 程序激活和禁用交换区。这两个程序分别使用了 `swapon()` 和 `swapoff()` 系统调用，我们将简要介绍相应的服务例程。

`sys_swapon()` 服务例程

`sys_swapon()` 服务例程接收如下参数：

specialfile

这个参数指向设备文件（或分区）的路径名（在用户态地址空间），或指向实现交换区的普通文件的路径名。

swap_flags

这个参数由一个单独的 SWAP_FLAG_PREFER 位加上交换区优先级的 31 位组成（只有在 SWAP_FLAG_PREFER 位置位时，优先级位才有意义）。

sys_swapon() 函数对创建交换区时放入第一个页槽中的 swap_header 联合体字段进行检查。其执行的主要步骤有：

1. 检查当前进程是否具有 CAP_SYS_ADMIN 权能。
2. 在交换区描述符 swap_info 数组的前 nr_swapfiles 个元素中查找 SWP_USED 标志为 0（即对应的交换区不是活动的）的第一个描述符。如果找到一个不活动交换区，则跳到第 4 步。
3. 新交换区数组索引等于 nr_swapfiles；它检查保留给交换区索引的位数是否足够用于编码新索引。如果不够，则返回错误代码；如果足够，就将 nr_swapfiles 的值加 1。
4. 找到未用交换区索引：它初始化这个描述符的字段，即把 flags 置为 SWP_USED，把 lowest_bit 和 highest_bit 置为 0。
5. 如果 swap_flags 参数为新交换区指定了优先级，则设置描述符的 prio 字段。否则，就把所有活动交换区中最低的优先级减 1 后赋给这个字段（这样就假设最后一个被激活的交换区在最慢的块设备上）。如果没有其他交换区是活动的，就把该字段设置成 -1。
6. 从用户态地址空间复制由 specialfile 参数所指向的字符串。
7. 调用 filp_open() 打开由 specialfile 参数指定的文件（参见第十二章的“open() 系统调用”一节）。
8. 把 filp_open() 返回的文件对象地址存放在交换区描述符的 swap_file 字段。
9. 检查 swap_info 中其他的活动交换区，以确认该交换区还未被激活。具体就是，检查交换区描述符的 swap_file->f_mapping 字段中存放的 address_space 对象地址。如果交换区已被激活，则返回错误码。
10. 如果 specialfile 参数标识一个块设备文件，则执行下列子步骤：
 - a. 调用 bd_claim() 把交换子系统设置成块设备的占有者（参见第十四章的“块设备”一节）。如果块设备已有一个占有者，则返回错误码。

- b. 把 `block_device` 描述符地址存入交换区描述符的 `bdev` 字段。
 - c. 把设备的当前块大小存放在交换区描述符的 `old_block_size` 字段，然后把设备的块大小设成 4096 字节（即页的大小）。
11. 如果 `specialfile` 参数标识一个普通文件，则执行下列子步骤：
 - a. 检查文件索引节点 `i_flags` 字段中的 `S_SWAPFILE` 字段。如果该标志置位，说明文件已被用作交换区，返回错误码。
 - b. 把该文件所在块设备的描述符地址存入交换区描述符的 `bdev` 字段。
 12. 读入存放在交换区页槽 0 中的 `swap_header` 描述符。为达到这个目的，它调用 `read_cache_page()`，并传入参数：由 `swap_file->f_mapping` 指向的 `address_space` 对象、页索引 0、文件 `readpage` 方法的地址（存放在 `swap_file->f_mapping->a_ops->readpage`）和指向文件对象 `swap_file` 的指针。然后等待直到页被读入内存。
 13. 检查交换区中第一页的最后 10 个字符中的魔术字符串是否等于“SWAPSPACE2”。如果不是，就返回一个错误码。
 14. 根据存放在 `swap_header` 联合体的 `info.last_page` 字段中的交换区的大小，初始化交换区描述符的 `lowest_bit` 和 `highest_bit` 字段。
 15. 调用 `vmalloc()` 来创建与新交换区相关的计数器数组，并把它的地址存放在交换描述符的 `swap_map` 字段中。还要根据 `swap_header` 联合体的 `info.bad_pages` 字段中存放的有缺陷的页槽链表把这个数组的元素初始化成 0 或 `SWAP_MAP_BAD`。
 16. 通过访问第一个页槽中的 `info.last_page` 和 `info.nr_badpages` 字段计算可用页槽的个数，并把它存入交换区描述符的 `pages` 字段。而且把交换区中的总页数赋给 `max` 字段。
 17. 为新交换区建立子区链表 `extent_list`（如果交换区建立在磁盘分区上，则只有一个子区），并相应地设定交换区描述符的 `nr_extents` 和 `curr_swap_extent` 字段。
 18. 把交换区描述符的 `flags` 字段设为 `SWP_ACTIVE`。
 19. 更新 `nr_good_pages`、`nr_swap_pages` 和 `total_swap_pages` 三个全局变量。
 20. 把新交换区描述符插入 `swap_list` 变量所指向的链表中。
 21. 返回 0（成功）。

sys_swapoff()服务例程

`sys_swapoff()` 服务例程使 `specialfile` 参数所指定的交换区无效。`sys_swapoff()` 比 `sys_swapon()` 复杂得多，也更加耗时，因为使之无效的这个分区现在可能仍然还包含

几个进程的页。因此，强制该函数扫描交换区并把所有现有的页都换入。由于每个换入操作都需要一个新的页框，因此如果现在没有空闲页框，这个操作就可能失败。在这种情况下，该函数就返回一个错误码。所有这些操作都是通过执行以下主要步骤实现的：

1. 验证当前进程是否具有 `CAP_SYS_ADMIN` 权能。
2. 拷贝内核空间中 `specialfile` 所指向的字符串。
3. 调用 `filp_open()`，打开 `specialfile` 参数确定的文件。与往常一样，该函数返回文件对象的地址。
4. 扫描交换区描述符链表 `swap_list`，比较由 `filp_open()` 返回的文件对象地址与活动交换区描述符的 `swap_file` 字段中的地址，如果不一致，说明传给函数的是一个无效参数，则返回一个错误码。
5. 调用 `cap_vm_enough_memory()`，检查是否有足够的空闲页框把交换区上存放的所有页换入。如果不够，交换区就不能禁用，然后释放文件对象，返回错误码。这只是个粗略的检查，但可使内核免于许多无用的磁盘操作。当执行这项检查时，`cap_vm_enough_memory()` 要考虑由 `slab` 高速缓存分配且 `SLAB_RECLAIM_ACCOUNT` 标志置位的页框（参见第八章中“`slab` 分配器与分区页框分配器的接口”一节），这样的页（被认为是可回收的这些页）的数量存放在 `slab_reclaim_pages` 变量中。
6. 从 `swap_list` 链表中删除该交换区描述符。
7. 从 `nr_swap_pages` 和 `total_swap_pages` 的值中减去存放在交换区描述符的 `pages` 字段中的值。
8. 把交换区描述符 `flags` 字段中的 `SWP_WRITEOK` 标志清 0。这可禁止 PFRA 向交换区换出更多的页。
9. 调用 `try_to_unuse()` 函数（见下面）强制把这个交换区中剩余的所有页都移到 RAM 中，并相应地修改使用这些页的进程的页表。当执行该函数时，当前进程（即运行 `swaponoff` 的进程）的 `PF_SWAPOFF` 标志置位。该标志置位只有一个结果：如页框严重不足，`select_bad_process()` 函数就会被强制选择并删除该进程（参见本章前面“内存不足删除程序”一节）。
10. 一直等到交换区所在的块设备驱动器被卸载（参见第十四章“激活块设备驱动程序”一节）。这样在交换区被禁用之前，`try_to_unuse()` 发出的读请求会被驱动器处理。
11. 如果在分配所有请求的页框时 `try_to_unuse()` 函数失败，那么就不能禁用这个交换区。因此，`sys_swaponoff()` 执行下列子步骤：
 - a. 把这个交换区描述符重新插入 `swap_list` 链表，并把它的 `flags` 字段置为 `SWP_WRITEOK`。

- b. 把交换区描述符中 `pages` 字段的值加到 `nr_swap_pages` 和 `total_swap_pages` 变量以恢复其原值。
 - c. 调用 `filp_close()` 关闭在第 3 步中打开的文件（参见第十二章“`close()` 系统调用”一节），并返回错误码。
12. 否则，所有已用的页槽都已经被成功传送到 RAM 中。因此，执行下列子步骤：
- a. 释放存有 `swap_map` 数组和子区描述符的内存区域。
 - b. 如果交换区存放在磁盘分区，则把块大小恢复到原值，该原值存放在交换区描述符的 `old_block_size` 字段。而且，调用 `bd_release()` 函数，使交换子系统不再占有该块设备（参见 `sys_swapon()` 函数第 10a 步的描述）。
 - c. 如果交换区存放在普通文件中，则把文件索引节点的 `S_SWAPFILE` 标志清 0。
 - d. 调用 `filp_close()` 两次，第一次针对 `swap_file` 文件对象，第二次针对第 3 步中 `filp_open()` 返回的对象。
 - e. 返回 0（成功）。

try_to_unuse() 函数

`try_to_unuse()` 函数使用一个索引参数，该参数标识待清空的交换区。该函数换入页并更新已换出页的进程的所有页表。因此，该函数从 `init_mm` 内存描述符（用作标记）开始，访问所有内核线程和进程的地址空间。这是一个相当耗时的函数，通常以开中断运行。因此，与其他进程的同步也是关键的。

`try_to_unuse()` 函数扫描交换区的 `swap_map` 数组。当它找到一个“在用”页槽时，首先换入其中的页，然后开始查找引用该页的进程。这两个操作的顺序对避免竞争条件是至关重要的。当 I/O 数据传送正在进行时，页被加锁，因此没有进程可以访问它。一旦 I/O 数据传送完成，页又被 `try_to_unuse()` 加锁，以使它不会被另一个内核控制路径再次换出。因为每个进程在开始进行换入或换出操作之前查找页高速缓存，所以这也可以避免竞争条件（参见后面“交换高速缓存”一节）。最后，由 `try_to_unuse()` 所考虑的交换区被标记为不可写（`SWP_WRITEOK` 标志被清 0），因此，没有进程可以对这个交换区的页槽执行换出。

但是，可能强迫 `try_to_unuse()` 对交换区引用计数器的 `swap_map` 数组扫描几次。这是因为对换出页引用的线性区可能在一次扫描中消失，而在随后又出现在进程链表中。

例如，回想 `do_munmap()` 函数的描述（在第九章“释放线性地址区间”一节）：只要进程释放一个线性地址区间，`do_munmap()` 就从进程链表中删除所有受影响线性地址所在的线性区；随后，该函数把只是部分解除映射的那部分线性区重新插入进程链表中。

`do_munmap()`还要负责释放属于已释放线性地址区间的换出页；但是，如果换出的页属于重新插入进程链表的线性区，则最好不要释放它们。

因此，`try_to_unuse()`对引用给定页槽的进程进行查找时可能失败，因为相应的线性区暂时没有包含在进程链表中。为了处理这种情况，`try_to_unuse()`一直对`swap_map`数组进行扫描，直到所有的引用计数器都变为空。引用了换出页的“神出鬼没”的线性区最终会重新出现在进程链表中，因此，`try_to_unuse()`终将会成功释放所有页槽。

让我们现在来描述`try_to_unuse()`所执行的主要操作。传递给它的参数为交换区`swap_map`数组的引用计数器，该函数在这个引用计数器上执行连续循环。如果当前进程接收到一个信号，则循环会中断，函数返回错误码。对于数组中的每个引用计数器，`try_to_unuse()`执行下列步骤：

1. 如果计数器等于0（没有页存放在这里）或者等于`SWAP_MAP_BAD`，则对下一个页槽继续处理。
2. 否则，调用`read_swap_cache_async()`函数（参见本章后面“换入页”一节）换入该页。这包括分配一个新页框（如果必要），用存放在页槽中的数据填充新页框并把该页存放在交换高速缓存。
3. 等待，直到用磁盘中的数据适当地更新了该新页，然后锁住它。
4. 当正在执行前一步时，进程有可能被挂起。因此，还要检查这个页槽的引用计数器是否变为空，如果是，说明这个交换页可能被另一个内核控制路径释放，然后继续处理下一个页槽。
5. 对于以`init_mm`为头部的双向链表（参见第九章“内存描述符”一节）中的每个内存描述符，调用`unuse_process()`。这个耗时的函数扫描拥有内存描述符的进程的所有页表项，并用这个新页框的物理地址替换页表中每个出现的换出页标识符。为了反映这种移动，还要把`swap_map`数组中的页槽计数器减1（除非计数器等于`SWAP_MAP_MAX`），并增加这个页框的引用计数器。
6. 调用`shmem_unuse()`检查换出的页是否用于IPC共享内存资源，并适当地处理那种情况（参见第十九章“IPC共享内存”一节）。
7. 检查页的引用计数器。如果它的值等于`SWAP_MAP_MAX`，则页槽是“永久的”。为了释放它，则把引用计数器强制置为1。
8. 交换高速缓存可能也拥有该页（它对引用计数器的值起作用）。如果页属于交换高速缓存，就调用`swap_writepage()`函数把页的内容刷新到磁盘（如果页为脏），调用`delete_from_swap_cache()`从交换高速缓存删去页，并把页的引用计数减1。

9. 设置页描述符的 `PG_dirty` 标志, 并打开页框的锁, 递减它的引用计数器 (取消第 5 步的增量)。
10. 检查当前进程的 `need_resched` 字段; 如果它被设置, 则调用 `schedule()` 放弃 CPU。禁用交换区是一件冗长的工作, 内核必须保证系统中的其他进程仍然继续执行。只要这个进程再次被调度程序选中, `try_to_unuse()` 函数就从这一步继续执行。
11. 继续到下一个页槽, 从第 1 步开始。

`try_to_unuse()` 继续执行, 直到 `swap_map` 数组中的每个引用计数器都为空。回想一下, 即使这个函数已经开始检查下一个页槽, 但是前一个页槽的引用计数器有可能仍然为正。事实上, 一个“神出鬼没”的进程可能还在引用这个页, 典型的原因是某些线性区已经被临时从第 5 步所扫描的进程链表中删除。`try_to_unuse()` 最终会捕获到每个引用。但是, 在此期间, 页不再位于交换高速缓存, 它的锁被打开, 并且页的一个拷贝仍然包含在要禁用的交换区的页槽中。

一般会认为这种情形可能导致数据丢失。例如, 假定某个“神出鬼没”的进程访问页槽, 并开始换入其中的页。因为页不再位于交换高速缓存, 因此, 进程用从磁盘读取的数据填充一个新的页框。但是, 这个页框可能不同于与“神出鬼没”进程共享页的那些进程曾经拥有的页框。

当禁用交换区时这个问题不会发生, 因为只有换出的页属于私有匿名内存映射时 (注 8), 对“神出鬼没”进程的干涉才会发生。在这种情况下, 使用第九章描述的“写时复制”机制来处理页框, 所以, 把不同的页框分配给引用了同一页的进程是完全合法的。但是, `try_to_unuse()` 函数将页标记为“脏” (第 9 步)。否则, `shrink_list()` 函数可能随后从某个进程的页表中删除这一页, 而并不把它保存在另一个交换区中 (参见后面“换出页”一节)。

分配和释放页槽

正如我们将在后面看到的那样, 在释放内存时, 内核要在很短的时间内把很多页都交换出去。因此尽力把这些页存放在相邻的页槽中非常重要, 这样就减少了在访问交换区时磁盘的寻道时间。

搜索空闲页槽的第一种方法可以选择下列两种既简单而又有些极端的策略之一:

- 总是从交换区的开头开始。这种方法在换出操作过程中可能会增加平均寻道时间, 因为空闲页槽可能已经被弄得凌乱不堪。

注 8: 事实上, 页可能属于 IPC 共享内存区, 第十九章将讨论这种情况。

- 总是从最后一个已分配的页槽开始。如果交换区的大部分空间都是空闲的（这是最通常的情况），那么这种方法在换入操作过程中会增加平均寻道时间，因为所占用的为数不多的页槽可能是零散存放的。

Linux 采用了一种混合的方法。除非发生以下这些条件之一，否则 Linux 总是从最后一个已分配的页槽开始查找。

- 已经到达交换区的末尾。
- 上次从交换区的开头重新分配之后，已经分配了 `SWAPFILE_CLUSTER`（通常是 256）个空闲页槽。

`swap_info_struct` 描述符的 `cluster_nr` 字段存放已分配的空闲页槽数。当函数从交换区的开头重新分配时该字段被重置为 0。`cluster_next` 字段存放在下一次分配时要检查的第一个页槽的索引（注 9）。

为了加速对空闲页槽的搜索，内核要保证每个交换区描述符的 `lowest_bit` 和 `highest_bit` 字段是最新的。这两个字段定义了第一个和最后一个可能为空的页槽，换言之，所有低于 `lowest_bit` 和高于 `highest_bit` 的页槽都被认为已经分配过。

scan_swap_map() 函数

`scan_swap_map()` 函数用来在给定的交换区中查找一个空闲页槽。该函数只作用于一个参数，该参数指向交换区描述符并返回一个空闲页槽的索引。如果交换区不含有任何空闲页槽，就返回 0。该函数执行以下步骤：

1. 首先试图使用当前的簇。如果交换区描述符的 `cluster_nr` 字段是正数，就从 `cluster_next` 索引处的元素开始对计数器的 `swap_map` 数组进行扫描，查找一个空项。如果找到一个空项，就减少 `cluster_nr` 字段的值并转到第 4 步。
2. 如果执行到这儿，那么，或者 `cluster_nr` 字段为空，或者从 `cluster_next` 开始搜索后没有在 `swap_map` 数组中找到空项。现在就应该开始第二阶段的混合查找。把 `cluster_nr` 重新初始化成 `SWAPFILE_CLUSTER`，并从 `lowest_bit` 索引处开始重新扫描这个数组，以便试图找到有 `SWAPFILE_CLUSTER` 个空闲页槽的一个组。如果找到这样的一个组，就转到第 4 步。
3. 不存在 `SWAPFILE_CLUSTER` 个空闲页槽的组。从 `lowest_bit` 索引处开始重新开始扫描这个数组，以便试图找到一个单独的空闲页槽。如果没有找到空项，就把

注 9：正如你可能已经注意到的一样，Linux 数据结构的名字并不都是恰当的。在这种情况下，内核并不会真正“聚簇（cluster）”交换区的页槽。

lowest_bit 字段置为数组的最大索引, highest_bit 字段置为 0, 并返回 0 (交换区已满)。

4. 已经找到空项。把 1 放在空项中, 减少 nr_swap_pages 的值, 如果需要就修改 lowest_bit 和 highest_bit 字段, 把 inuse_page 字段的值加 1, 并把 cluster_next 字段设置成刚才分配的页槽的索引加 1。
5. 返回刚才分配的页槽的索引。

get_swap_page() 函数

get_swap_page() 函数通过搜索所有活动的交换区来查找一个空闲页槽。它返回一个新近分配页槽的换出页标识符, 如果所有的交换区都填满, 就返回 0, 该函数要考虑活动交换区的不同优先级。

该函数需要经过两遍扫描, 以便在容易发现页槽时节约运行时间。第一遍是部分的, 只适用于只有相同优先级的交换区。该函数以轮询的方式在这种交换区中查找一个空闲页槽。如果没有找到空闲页槽, 就从交换区链表的起始位置开始进行第二遍扫描。在第二遍扫描中, 要对所有的交换区都进行检查。更确切地说, 该函数执行以下步骤:

1. 如果 nr_swap_pages 为空或者如果没有活动的交换区, 就返回 0。
2. 首先考虑 swap_list.next 所指向的交换区 (回想一下, 交换区链表是按照优先级从高到低的顺序排列的)。
3. 如果交换区是活动的, 就调用 scan_swap_map() 来获得一个空闲页槽。如果 scan_swap_map() 函数返回一个页槽索引, 该函数的任务基本上就完成了, 但是它还要准备下一次被调用。因此, 如果下一个交换区的优先级和这个交换区的优先级相同 (即轮询使用这些交换区), 该函数就把 swap_list.next 修改成指向交换区链表中的下一个交换区。如果下一个交换区的优先级和当前交换区的优先级不同, 该函数就把 swap_list.next 设置成交换区链表中的第一个交换区 (这样下一次搜索操作就会从优先级最高的交换区开始)。该函数最终返回刚才分配的页槽所对应的换出页标识符。
4. 或者交换区是不可写的, 或者交换区中没有空闲页槽。如果交换区链表中的下一个交换区的优先级和当前交换区的优先级相同, 就把下一个交换区设置成当前交换区并跳转到第 3 步。
5. 此时, 交换区链表中的下一个交换区的优先级小于前一个交换区的优先级。下一步操作取决于该函数正在进行哪一遍扫描。
 - a. 如果这是第一遍 (局部) 扫描, 就考虑链表中的第一个交换区并跳转到第 3 步, 这样就开始第二遍扫描。

- b. 否则，就检查交换区链表中是否有下一个元素。如果有，就考虑这个元素并跳转到第3步。
6. 此时，第二遍对链表的扫描已经完成，并没有发现空闲页槽，返回0。

swap_free()函数

当换入页时，调用 `swap_free()` 函数以对相应的 `swap_map` 计数器进行减1操作（参见表17-3）。当相应的计数器达到0时，由于页槽的标识符不再包含在任何页表项中，因此页槽就变成空闲。但是，我们将在后面“交换高速缓存”一节看到，交换高速缓存也记入页槽拥有者的个数。

该函数只作用于一个参数 `entry`，`entry` 表示换出页标识符。函数执行以下步骤：

1. 从 `entry` 参数导出交换区索引和页槽索引 `offset`，并获得交换区描述符的地址。
2. 检查交换区是否是活动的。如果不是，就立即返回。
3. 如果正在释放的页槽对应的 `swap_map` 计数器小于 `SWAP_MAP_MAX`，就减少这个计数器的值。回想一下，值为 `SWAP_MAP_MAX` 的项都被认为是永久的（不可删除的）。
4. 如果 `swap_map` 计数器变成0，就增加 `nr_swap_pages` 的值，减少 `inuse_pages` 字段的值，如果需要就修改这个交换区描述符的 `lowest_bit` 和 `highest_bit` 字段。

交换高速缓存

向交换区来回传送页面会引发很多竞争条件，具体地说，交换子系统必须仔细处理下面的情形：

多重换入

两个进程可能同时要换入同一个共享匿名页。

同时换入换出

一个进程可能换入正由 PFRA 换出的页。

交换高速缓存 (*swap cache*) 的引入就是为了解决这类同步问题。关键的原则是，没有检查交换高速缓存是否已包括了所涉及的页，就不能进行换入或换出操作。有了交换高速缓存，涉及同一页的并发交换操作总是作用于同一个页框的。因此，内核可以安全地依赖页描述符的 `PG_locked` 标志，以避免任何竞争条件。

考虑一下共享同一换出页的两个进程这种情形。当第一个进程试图访问页时，内核开始换入页操作，第一步就是检查页框是否在交换高速缓存中，我们假定页框不在交换高速缓存中，那么内核就分配一个新页框并把它插入交换高速缓存，然后开始 I/O 操作，从

交换区读入页的数据；同时，第二个进程访问该共享匿名页，与上面相同，内核开始换入操作，检查涉及的页框是否在交换高速缓存中。现在页框是在交换高速缓存，因此内核只是访问页框描述符，在 `PG_locked` 标志清 0 之前（即 I/O 数据传输完毕之前），让当前进程睡眠。

当换入换出操作同时出现时，交换高速缓存起着至关重要的作用。在本章前面“内存紧缺回收”一节我们描述过，`shrink_list()` 函数要开始换出一个匿名页，就必须当 `try_to_unmap()` 从进程（所有拥有该页的进程）的用户态页表中成功删除了该页后才可以。但是当换出的写操作还在执行的时候，这些进程中可能有某个进程要访问该页，而产生换入操作。

在写入磁盘前，待换出页由 `shrink_list()` 存放在交换高速缓存。考虑页 P 由两个进程（A 和 B）共享。最初，两个进程的页表项都引用该页框，该页有两个拥有者，如图 17-8 (a) 所示。当 PFRA 选择回收页时，`shrink_list()` 把页框插入交换高速缓存。如图 17-8 (b) 所示，现在页框有三个拥有者，而交换区中的页槽只被交换高速缓存引用。然后 PFRA 调用 `try_to_unmap()` 从这两个进程的页表项中删除对该页框的引用。一旦这个函数结束，该页框就只有交换高速缓存引用它，而引用页槽的有这两个进程和交换高速缓存，如图 17-8 (c) 所示。假定：当页中的数据写入磁盘时，进程 B 访问该页，即它要用该页内部的线性地址访问内存单元。那么，缺页异常处理程序发现页框在交换高速缓存，并把物理地址放回进程 B 的页表项，如图 17-8 (d) 所示。相反地，如果换出操作结束，而没有并发换入操作，`shrink_list()` 函数则从交换高速缓存删除该页框并把它释放到伙伴系统，如图 17-8 (e) 所示。

你可以认为交换高速缓存是一个临时区域，该区域存有正在被换入或换出的匿名页描述符。当换入或换出结束时（对于共享匿名页，换入换出操作必须对共享该页的所有进程进行），匿名页描述符就可以从交换高速缓存删除（注 10）。

交换高速缓存的实现

交换高速缓存由页高速缓存数据结构和过程实现（在第十五章的“页高速缓存”一节中有描述）。回想一下，页高速缓存的核心就是一组基树，借助基树，算法就可以从 `address_space` 对象地址（即该页的拥有者）和偏移量值推算出页描述符的地址。

在交换高速缓存中页的存放方式是隔页存放，并有下列特征：

注 10： 有些情况下，交换高速缓存还能够提高系统性能：如服务器后台程序通过创建于进程提供服务请求时。系统负载很高时，页可能从父进程换出，而且再也不会成为父进程的页。如果不使用交换高速缓存，每个被创建的子进程就需要从交换区选择调入的页。

- 页描述符的 mapping 字段为 NULL。
- 页描述符的 PG_swapcache 标志置位。
- private 字段存放与该页有关的换出页标识符。

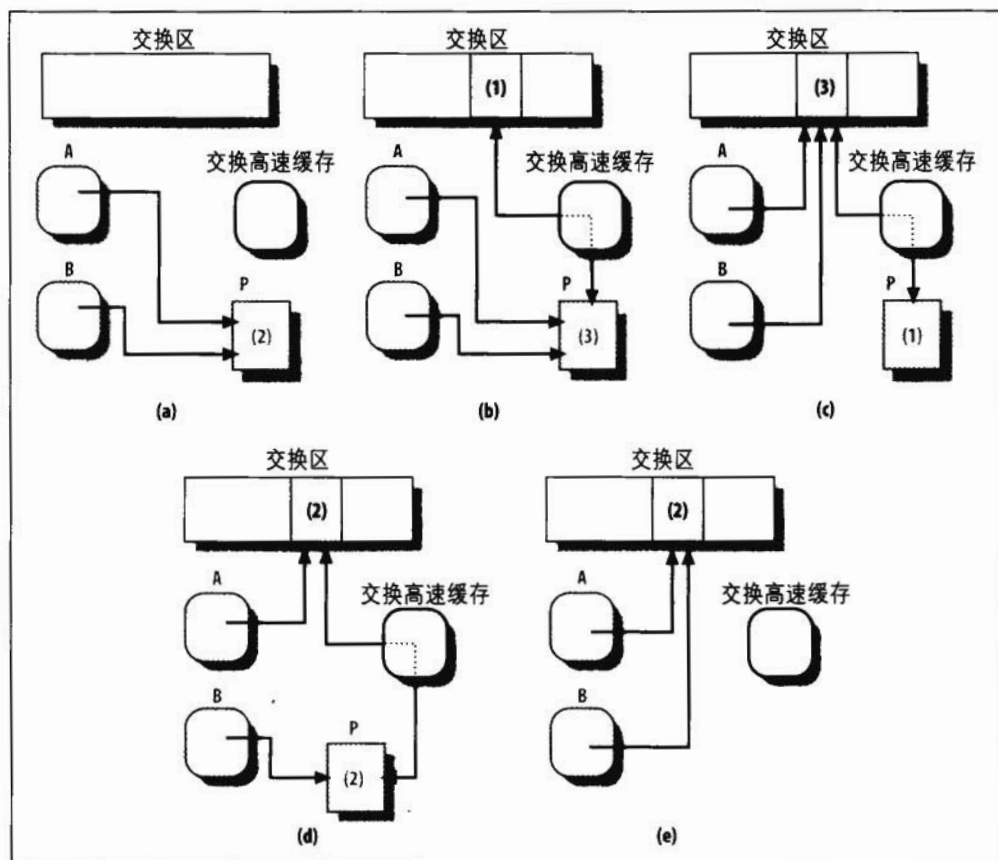


图 17-8: 交换高速缓存的作用

此外, 当页被放入交换高速缓存时, 则页描述符的 count 字段和页槽引用计数器的值都增加, 因为交换高速缓存既要使用页框, 也要使用页槽。

最后, 交换高速缓存中的所有页只使用一个 swapper_space 地址空间, 因此只有一个基树 (由 swapper_space.page_tree 指向) 对交换高速缓存中的页进行寻址。swapper_space 地址空间的 nrpages 字段存放交换高速缓存中的页数。

交换高速缓存的辅助函数

内核使用几个函数来处理交换高速缓存，它们主要是基于第十五章的“页高速缓存”一节中所讨论的那些函数。稍后我们将说明这些相对低层的函数是如何被高层函数调用来按需换入和换出页的。

处理交换高速缓存的函数主要有：

`lookup_swap_cache()`

通过传递来的参数(换出页标识符)在交换高速缓存中查找页并返回页描述符的地址。如果该页不在交换高速缓存中，就返回0。该函数调用`radix_tree_lookup()`函数，把指向`swapper_space.page_tree`的指针(用于交换高速缓存中页的基树)和换出页标识符作为参数传递，以查找所需要的页。

`add_to_swap_cache()`

把页插入交换高速缓存中。它本质上调用`swap_duplicate()`检查作为参数传递来的页槽是否有效，并增加页槽引用计数器；然后调用`radix_tree_insert()`把页插入高速缓存；最后递增页引用计数器并将`PG_swapcache`和`PG_locked`标志置位。

`__add_to_swap_cache()`

与`add_to_swap_cache()`类似，但是，在把页框插入交换高速缓存前，这个函数不调用`swap_duplicate()`。

`delete_from_swap_cache()`

调用`radix_tree_delete()`从交换高速缓存中删除页，递减`swap_map`中相应的使用计数器，递减页引用计数器。

`free_page_and_swap_cache()`

如果除了当前进程外，没有其它用户态进程正在引用相应的页槽，则从交换高速缓存中删除该页，并递减页使用计数器。

`free_pages_and_swap_cache()`

与`free_page_and_swap_cache()`相似，但它是对一组页操作。

`free_swap_and_cache()`

释放一个交换表项，并检查该表项引用的页是否在交换高速缓存。如果没有用户态进程(除了当前进程之外)引用该页，或者超过50%的交换表项在用，则从交换高速缓存中释放该页。

换出页

我们从本章前面“内存紧缺回收”一节可看到，`PFRA`是如何确定一个给定的匿名页是否该被换出。在这一节，我们描述内核如何执行换出操作。

向交换高速缓存插入页框

换出操作的第一步就是准备交换高速缓存。如果 `shrink_list()` 函数确认某页为匿名页 (`PageAnon()` 函数返回1) 而且交换高速缓存中没有相应的页框 (页描述符的 `PG_swapcache` 标志清0), 内核就调用 `add_to_swap()` 函数。

`add_to_swap()` 函数在交换区中分配一个新页槽, 并把一个页框 (其页描述符地址作为参数传递) 插入交换高速缓存。函数执行如下主要步骤:

1. 调用 `get_swap_page()` 函数分配一个新页槽 (参见本章前面“分配和释放页槽”一节)。如果失败 (例如没有发现空闲页槽), 则返回0。
2. 调用 `__add_to_page_cache()`, 传给它页槽索引、页描述符地址和一些分配标志。
3. 将页描述符中的 `PG_uptodate` 和 `PG_dirty` 标志置位, 从而强制 `shrink_list()` 函数把页写入磁盘 (见下一节)。
4. 返回1 (成功)。

更新页表项

一旦 `add_to_swap()` 结束, `shrink_list()` 就调用 `try_to_unmap()`, 它确定引用匿名页的每个用户态页表项地址, 然后将换出页标识符写入其中 (参见本章前面“匿名页的反向映射”一节)。

将页写入交换区

为完成换出操作需执行的下一个步骤是将页的数据写入交换区。这一 I/O 传输是由 `shrink_list()` 函数激活的, 它检查页框的 `PG_dirty` 标志是否置位, 然后执行 `pageout()` 函数 (参见本章前面的图 17-5)。

在本章前面的“内存紧缺回收”一节我们描述过, `pageout()` 函数建立一个 `writeback_control` 描述符, 且调用页 `address_space` 对象的 `writepage` 方法。而 `swapper_state` 对象的 `writepage` 方法是由 `swap_writepage()` 函数实现的。

`swap_writepage()` 函数所执行的主要步骤如下:

1. 检查是否至少有一个用户态进程引用该页。如果没有, 则从交换高速缓存删除该页, 并返回0。这一检查之所以必须做, 是因为一个进程可能会与 PFRA 发生竞争并在 `shrink_list()` 检查完后释放一页。
2. 调用 `get_swap_bio()` 分配并初始化一个 `bio` 描述符 (参见第十四章“bio 结构”一节)。函数从换出页标识符算出交换区描述符地址。然后它搜索交换子区链表, 以

找到页槽的初始磁盘扇区。bio描述符将包含一个单页数据请求（页槽），其完成方法设为 `end_swap_bio_write()` 函数。

3. 置位页描述符的 `PG_writeback` 标志和交换高速缓存基树的 `writeback` 标记（参见第十五章的“基树的标记”一节）。此外函数还清 `OPG_locked` 标志。
4. 调用 `submit_bio()`，传给它 `WRITE` 命令和 `bio` 描述符地址。
5. 返回 0。

一旦 I/O 数据传输结束，就执行 `end_swap_bio_write()` 函数。实际上，这个函数唤醒正等待页 `PG_writeback` 标志清 0 的所有进程，清除 `PG_writeback` 标志和基树中的相关标记，并释放用于 I/O 传输的 `bio` 描述符。

从交换高速缓存中删除页框

换出操作的最后一步还是由 `shrink_list()` 执行。如果它验证在 I/O 数据传输时没有进程试图访问该页框，它实际就调用 `delete_from_swap_cache()` 从交换高速缓存中删除该页框。因为交换高速缓存是该页的唯一拥有者，该页框被释放到伙伴系统。

换入页

当进程试图对一个已被换出到磁盘的页进行寻址时，必然会发生页的换入。在以下条件发生时，缺页异常处理程序就会触发一个换入操作（参见第九章中的“处理地址空间内的错误地址”一节）：

- 引起异常的地址所在的页是一个有效的页，也就是说，它属于当前进程的一个线性区。
- 页不在内存中，也就是说，页表项中的 `Present` 标志被清除。
- 与页有关的页表项不为空，但是 `Dirty` 位清 0，这意味着页表项包含一个换出页标识符（参见第九章的“请求调页”一节）。

如果上面的所有条件满足，则 `handle_pte_fault()` 调用相对简易的 `do_swap_page()` 函数换入所需页。

`do_swap_page()` 函数

`do_swap_page()` 函数作用于如下参数：

`mm`

引起缺页异常的进程的内存描述符地址

vma	address 所在的线性区的线性区描述符地址
address	引起异常的线性地址
page_table	映射 address 的页表项的地址
Pmd	映射 address 的页中间目录的地址
orig_pte	映射 address 的页表项的内容
write_access	一个标志, 表示试图执行的访问是读操作还是写操作

与其他函数相反, `do_swap_page()` 从不返回 0。如果页已经在交换高速缓存中就返回 1 (次错误), 如果页已经从交换区读入就返回 2 (主错误), 如果在进行换入时发生错误就返回 -1。该函数本质上执行下列步骤:

1. 从 `orig_pte` 获得换出页标识符。
2. 调用 `pte_unmap()` 释放任何页表的临时内核映射, 该页表由 `handle_mm_fault()` 函数建立 (参见第九章的“处理地址空间内的错误地址”一节)。正如第八章“高端内存页框的内核映射”一节所述, 访问高端内存页表需要进行内核映射。
3. 释放内存描述符的 `page_table_lock` 自旋锁 (它是由调用者函数 `handle_pte_fault()` 获取的)。
4. 调用 `lookup_swap_cache()` 检查交换高速缓存是否已经含有换出页标识符对应的页; 如果页已经在交换高速缓存中, 就跳到第 6 步。
5. 调用 `swpin_readahead()` 函数从交换区读取至多有 $2n$ 个页的一组页, 其中包括所请求的页。值 n 存放在 `page_cluster` 变量中, 通常等于 3 (注 11)。其中的每个页是通过调用 `read_swap_cache_async()` 函数读入的 (参见下面)。
6. 再一次调用 `read_swap_cache_async()` 换入由引起缺页异常的进程所访问的那一页。这一步可能看起来有点多余, 但其实不然。 `swpin_readahead()` 函数可能在读取请求的页时失败——例如, 因为 `page_cluster` 被置为 0, 或者该函数试图读取一组含

注 11: 系统管理员通过写 `/proc/sys/vm/page-cluster` 文件可以调整这个值。通过把 `page-cluster` 置为 0 可以禁止提前换入页。

有空闲或有缺陷页槽 (SWAP_MAP_BAD) 的页。另一方面, 如果 `swapon_readahead()` 成功, 这次对 `read_swap_cache_async()` 的调用就很快结束, 因为它在交换高速缓存找到了页。

7. 尽管如此, 如果请求的页还是没有被加到交换高速缓存, 那么, 另一个内核控制路径可能已经代表这个进程的一个子进程换入了所请求的页。这种情况的检查可以通过临时获取 `page_table_lock` 自旋锁, 并把 `page_table` 所指向的表项与 `orig_pte` 进行比较来实现。如果二者有差异, 则说明这一页已经被某个其他的内核控制路径换入, 因此, 函数返回 1 (次错误); 否则, 返回 -1 (失败)。
8. 函数执行到此, 我们知道页已经在高速缓存中。如果页已被换入 (主错误), 函数就调用 `grab_swap_token()` 试图获得一个交换标记 (参见本章前面“交换标记”一节)。
9. 调用 `mark_page_accessed()` [参见前面“最近最少使用 (LRU) 链表”一节] 并对页加锁。
10. 获取 `page_table_lock` 自旋锁。
11. 检查另一个内核控制路径是否代表这个进程的一个子进程换入了所请求的页。如果是, 就释放 `page_table_lock` 自旋锁, 打开页上的锁, 并返回 1 (次错误)。
12. 调用 `swap_free()` 减少 `entry` 对应的页槽的引用计数器。
13. 检查交换高速缓存是否至少占满 50% (`nr_swap_pages` 小于 `total_swap_pages` 的一半)。如果是, 则检查页是否仅被引起异常的进程 (或其一个子进程) 拥有; 如果是这样, 则从交换高速缓存中删去这一页。
14. 增加进程的内存描述符的 `rss` 字段。
15. 更新页表项以便进程能找到这一页。这一操作的实现是通过把所请求页的物理地址和在线性区的 `vm_page_prot` 字段所找到的保护位写入 `page_table` 所指向的页表项中来达到的。此外, 如果引起缺页的访问是一个写访问, 且造成缺页的进程是页的唯一拥有者, 那么, 函数还要设置 `Dirty` 和 `Read/Write` 标志以防止无用的写时复制错误。
16. 打开页上的锁。
17. 调用 `page_add_anon_rmap()` 把匿名页插入面向对象的反向映射数据结构 (参见本章前面“匿名页的反向映射”一节)。
18. 如果 `write_access` 参数等于 1, 则函数调用 `do_wp_page()` 复制一份页框 (参见第九章的“写时复制”一节)。
19. 释放 `mm->page_table_lock` 自旋锁, 并返回 1 (次错误) 或 2 (主错误)。

read_swap_cache_async() 函数

只要内核必须换入一个页，就调用 `read_swap_cache_async()` 函数，它接收的参数为：

`entry`

换出页标识符

`vma`

指向该页所在线性区的指针

`addr`

页的线性地址

我们知道，在访问交换分区之前，该函数必须检查交换高速缓存是否已经包含了所要的页框。因此，该函数本质上执行下列操作：

1. 调用 `radix_tree_lookup()`，搜索 `swapper_space` 对象的基树，寻找由换出页标识符 `entry` 给出位置的页框。如果找到该页，递增它的引用计数器，返回它的描述符地址。
2. 页不在交换高速缓存。调用 `alloc_page()` 分配一个新的页框。如果没有空闲的页框可用，则返回 0（表示系统没有足够的内存）。
3. 调用 `add_to_swap_cache()` 把新页框的页描述符插入交换高速缓存。正如前面“交换高速缓存的辅助函数”一节提到的那样，这个函数也对页加锁。
4. 如果 `add_to_swap_cache()` 在交换高速缓存找到页的一个副本，则前一步可能失败。例如，进程可能在第 2 步阻塞，因此允许另一个进程在同一个页槽上开始换入操作。在这种情况下，该函数释放在第 2 步分配的页框，并从第 1 步重新开始。
5. 调用 `lru_cache_add_active()` 把页插入 LRU 的活动链表[参见本章前面“最近最少使用 (LRU) 链表”一节]。
6. 新页框的页描述符现已在交换高速缓存。调用 `swap_readpage()` 从交换区读入该页数据。这个函数与前面“换出页”一节所描述的 `swap_writepage()` 函数很相似，它将页描述符的 `PG_uptodate` 标志清 0，调用 `get_swap_bio()` 为 I/O 传输分配与初始化一个 `bio` 描述符，再调用 `submit_bio()` 向块设备子系统层发出 I/O 请求。
7. 返回页描述符的地址。



Ext2 和 Ext3 文件系统

在本章，我们通过当与一个特定的文件系统交互时内核所关注的细节来结束对 I/O 和文件系统进行的广泛讨论。因为第二扩展文件系统（Ext2）是 Linux 所固有的，事实上已在每个 Linux 系统中得以使用，因此我们自然要对 Ext2 进行讨论。此外，Ext2 在对现代文件系统的高性能支持方面也显示出很多良好的实践性。固然，Linux 支持的其他文件系统有很多有趣的特点，但因篇幅限制，我们并不对此一一讨论。

在“Ext2 的一般特征”一节中引入 Ext2 后，我们会像在其他章节中一样，接着描述所需的数据结构。因为我们着眼于磁盘上存放数据的特定方式，因此必须考虑同一数据结构的两种形式：“Ext2 磁盘数据结构”一节说明把 Ext2 存放在磁盘上的数据结构，而“Ext2 的内存数据结构”一节说明如何把磁盘上的数据结构复制到内存中。

然后，我们讨论 Ext2 文件系统上所执行的操作。在“创建 Ext2 文件系统”一节，我们讨论如何在磁盘分区创建 Ext2。接下来的一节描述使用磁盘时内核所执行的操作。其中的大部分操作是为索引节点和数据块分配磁盘空间，这些操作相对比较低级。

在最后一节，我们将对 Ext3 文件系统给予简短描述，Ext3 是 Ext2 文件系统的发展。

Ext2 的一般特征

类 Unix 操作系统使用多种文件系统。尽管所有这些文件系统都有少数 POSIX API（如 `state()`）所需的共同的属性子集，但每种文件系统的实现方式是不同的。

Linux 的第一个版本是基于 MINIX 文件系统的。当 Linux 成熟时，引入了扩展文件系统 (*Extended Filesystem, Ext FS*)，它包含了几个重要的扩展但提供的性能不令人满意。在 1994 年引入了第二扩展文件系统 (Ext2)，它除了包含几个新的特点外，还相当高效和稳定，Ext2 及它的下代文件系统 Ext3 已成为广泛使用的 Linux 文件系统。

下列特点有助于 Ext2 的效率：

- 当创建 Ext2 文件系统时，系统管理员可以根据预期的文件平均长度来选择最佳块大小（从 1024B ~ 4096B）。例如，当文件的平均长度小于几千字节时，块的大小为 1024B 是最佳的，因为这会产生较少的内部碎片——也就是文件长度与存放它的磁盘分区有较少的不匹配（参见第八章中的“内存区管理”一节，在那里讨论了动态内存的内部碎片）。另一方面，大的块对于大于几千字节的文件通常比较合适，因为这样的磁盘传送较少，因而减轻了系统的开销。
- 当创建 Ext2 文件系统时，系统管理员可以根据在给定大小的分区上预计存放的文件数来选择给该分区分配多少个索引节点。这可以有效地利用磁盘的空间。
- 文件系统把磁盘块分为组。每组包含存放在相邻磁道上的数据块和索引节点。正是这种结构，使得可以用较少的磁盘平均寻道时间对存放在一个单独块组中的文件进行访问。
- 在磁盘数据块被实际使用之前，文件系统就把这些块预分配给普通文件。因此，当文件的大小增加时，因为物理上相邻的几个块已被保留，这就减少了文件的碎片。
- 支持快速符号链接（参见第一章中的“硬链接和软链接”一节）。如果符号链接表示一个短路径名（小于或等于 60 个字符），就把它存放在索引节点中而不用通过读一个数据块进行转换。

此外，Ext2 还包含了一些使它既健壮又灵活的特点：

- 文件更新策略的谨慎实现将系统崩溃的影响减到最少。例如，当给文件创建一个新的硬链接时，首先增加磁盘索引节点中的硬链接计数器，然后把这个名字加到合适的目录中。在这种方式下，如果在更新索引节点后而改变这个目录之前出现一个硬件故障，这样即使索引节点的计数器产生错误，但目录是一致的。因此，尽管删除文件时无法自动收回文件的数据块，但并不导致灾难性的后果。如果这种操作的顺序相反（更新索引节点前改变目录），同样的硬件故障将会导致危险的不一致：删除原始的硬链接就会从磁盘删除它的数据块，但新的目录项将指向一个不存在的索引节点。如果那个索引节点号以后又被另外的文件所使用，那么向这个旧目录项的写操作将毁坏这个新的文件。
- 在启动时支持对文件系统的状态进行自动的一致性检查。这种检查是由外部程序

e2fsck 完成的, 这个外部程序不仅可以在系统崩溃之后被激活, 也可以在一个预定义的文件系统安装数 (每次安装操作之后对计数器加 1) 之后被激活, 或者在自从最近检查以来所花的预定义时间之后被激活。

- 支持不可变 (*immutable*) 的文件 (不能修改、删除和更名) 和仅追加 (*append-only*) 的文件 (只能把数据追加在文件尾)。
- 既与 Unix System V Release 4 (SVR4) 相兼容, 也与新文件的用户组 ID 的 BSD 语义相兼容。在 SVR4 中, 新文件采用创建它的进程的用户组 ID; 而在 BSD 中, 新文件继承包含它的目录的用户组 ID。Ext2 包含一个安装选项, 由你指定采用哪种语义。

即使 Ext2 文件系统是如此成熟、稳定的程序, 也还要考虑引入另外几个特性。一些特性已被实现并以外部补丁的形式来使用。另外一些还仅仅处于计划阶段, 但在一些情况下, 已经在 Ext2 的索引节点中为这些特性引入新的字段。最重要的一些特点如下:

块片 (*block fragmentation*)

系统管理员对磁盘的访问通常选择较大的块, 因为计算机应用程序常常处理大文件。因此, 在大块上存放小文件就会浪费很多磁盘空间。这个问题可以通过把几个文件存放在同一块的不同片上来解决。

透明地处理压缩和加密文件

这些新的选项 (创建一个文件时必须指定) 将允许用户透明地在磁盘上存放压缩和 (或) 加密的文件版本。

逻辑删除

一个 *undelete* 选项将允许用户在必要时很容易恢复以前已删除的文件内容。

日志

日志避免文件系统在突然卸载 (例如, 作为系统崩溃的后果) 时对其自动进行的耗时检查。

实际上, 这些特点没有一个正式地包含在 Ext2 文件系统中。有人可能说 Ext2 是这种成功的牺牲品; 直到几年前, 它仍然是大多数 Linux 发布公司采用的首选文件系统, 每天有成千上万的用户在使用它, 这些用户会对用其他文件系统来代替 Ext2 的任何企图产生质疑。

Ext2 中缺少的最突出的功能就是日志, 日志是高可用服务器必需的功能。为了平顺过渡, 日志没有引入到 Ext2 文件系统; 但是, 我们在后面 “Ext3 文件系统” 一节会讨论, 完全与 Ext2 兼容的一种新文件系统已经创建, 这种文件系统提供了日志。不真正需要日

志的用户可以继续使用良好而老式的Ext2文件系统,而其他用户可能采用这种新的文件系统。现在发行的大部分系统采用Ext3作为标准的文件系统。

Ext2 磁盘数据结构

任何Ext2分区中的第一个块从不受Ext2文件系统的管理,因为这一块是为分区的引导扇区所保留的(参见附录一)。Ext2分区的其余部分分成块组(block group),每个块组的分布图如图18-1所示。正如你从图中所看到的,一些数据结构正好可以放在一块中,而另一些可能需要更多的块。在Ext2文件系统的所有块组大小相同并被顺序存放,因此,内核可以从块组的整数索引很容易地得到磁盘中一个块组的位置。

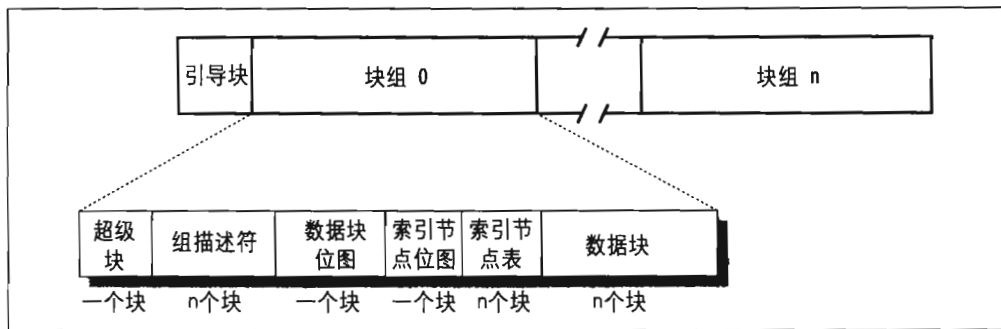


图 18-1: Ext2 分区和 Ext2 块组的分布图

由于内核尽可能地把属于一个文件的数据块存放在同一块组中,所以块组减少了文件的碎片。块组中的每个块包含下列信息之一:

- 文件系统的超级块的一个拷贝
- 一组块组描述符的拷贝
- 一个数据块位图
- 一个索引节点位图
- 一个索引节点表
- 属于文件的一大块数据,即数据块

如果一个块中不包含任何有意义的信息,就说这个块是空闲的。

从图18-1中可以看出,超级块与组描述符被复制到每个块组中。只有块组0中所包含的超级块和组描述符才由内核使用,而其余的超级块和组描述符保持不变;事实上,内核甚至不考虑它们。当e2fsck程序对Ext2文件系统的状态执行一致性检查时,就引用存放

在块组 0 中的超级块和组描述符，然后把它们拷贝到其他所有的块组中。如果出现数据损坏，并且块组 0 中的主超级块和主描述符变为无效，那么，系统管理员就可以命令 *e2fsck* 引用存放在某个块组（除了第一个块组）中的超级块和组描述符的旧拷贝。通常情况下，这些多余的拷贝所存放的信息足以让 *e2fsck* 把 Ext2 分区带回到一个一致的状态。

有多少块组呢？这取决于分区的大小和块的大小。其主要限制在于块位图，因为块位图必须存放在一个单独的块中。块位图用来标识一个组中块的占用和空闲状况。所以，每组中至多可以有 $8 \times b$ 个块， b 是以字节为单位的块大小。因此，块组的总数大约是 $s / (8 \times b)$ ，这里 s 是分区所包含的总块数。

举例说明，让我们考虑一下 32GB 的 Ext2 分区，块的大小为 4KB。在这种情况下，每个 4KB 的块位图描述 32K 个数据块，即 128MB。因此，最多需要 256 个块组。显然，块的大小越小，块组数越大。

超级块

Ext2 在磁盘上的超级块存放在一个 `ext2_super_block` 结构中，它的字段在表 18-1 中列出（注 1）。`__u8`、`__u16` 及 `__u32` 数据类型分别表示长度为 8、16 及 32 位的无符号数，而 `__s8`、`__s16` 及 `__s32` 数据类型表示长度为 8、16 及 32 位的有符号数。为清晰地表示磁盘上字或双字中字节的存放顺序，内核又使用了 `__le16`、`__le32`、`__be16` 和 `__be32` 数据类型，前两种类型分别表示字或双字的“小尾 (*little-endian*)”排序方式（低阶字节在高位地址），而后两种类型分别表示字或双字的“大尾 (*big-endian*)”排序方式（高阶字节在高位地址）。

表 18-1: Ext2 超级块的字段

类型	字段	说明
<code>__le32</code>	<code>s_inodes_count</code>	索引节点的总数
<code>__le32</code>	<code>s_blocks_count</code>	以块为单位的文件系统的大小
<code>__le32</code>	<code>s_r_blocks_count</code>	保留的块数
<code>__le32</code>	<code>s_free_blocks_count</code>	空闲块计数器
<code>__le32</code>	<code>s_free_inodes_count</code>	空闲索引节点计数器
<code>__le32</code>	<code>s_first_data_block</code>	第一个使用的块号（总为 1）
<code>__le32</code>	<code>s_log_block_size</code>	块的大小

注 1: 为了确保 Ext2 和 Ext3 文件系统之间的兼容性，`ext2_super_block` 数据结构包含了一些 Ext3 特有的字段，它们并未在表 18-1 中列出。

表 18-1: Ext2 超级块的字段 (续)

类型	字段	说明
__le32	s_log_frag_size	片的大小
__le32	s_blocks_per_group	每组中的块数
__le32	s_frags_per_group	每组中的片数
__le32	s_inodes_per_group	每组中的索引节点数
__le32	s_mtime	最后一次安装操作的时间
__le32	s_wtime	最后一次写操作的时间
__le16	s_mnt_count	安装操作计数器
__le16	s_max_mnt_count	检查之前安装操作的次数
__le16	s_magic	魔术签名
__le16	s_state	状态标志
__le16	s_errors	当检测到错误时的行为
__le16	s_minor_rev_level	次版本号
__le32	s_lastcheck	最后一次检查的时间
__le32	s_checkinterval	两次检查之间的时间间隔
__le32	s_creator_os	创建文件系统的操作系统
__le32	s_rev_level	版本号
__le16	s_def_resuid	保留块的缺省 UID
__le16	s_def_resgid	保留块的缺省用户组 ID
__le32	s_first_ino	第一个非保留的索引节点号
__le16	s_inode_size	磁盘上索引节点结构的大小
__le16	s_block_group_nr	这个超级块的块组号
__le32	s_feature_compat	具有兼容特点的位图
__le32	s_feature_incompat	具有非兼容特点的位图
__le32	s_feature_ro_compat	只读兼容特点的位图
__u8 [16]	s_uuid	128 位文件系统标识符
char [16]	s_volume_name	卷名
char [64]	s_last_mounted	最后一个安装点的路径名
__le32	s_algorithm_usage_bitmap	用于压缩
__u8	s_prealloc_blocks	预分配的块数
__u8	s_prealloc_dir_blocks	为目录预分配的块数
__u16	s_padding1	按字对齐
__u32 [204]	s_reserved	用 null 填充 1024 字节

`s_inodes_count` 字段存放索引节点的个数，而 `s_blocks_count` 字段存放 Ext2 文件系统的块的个数。

`s_log_block_size` 字段以 2 的幂次方表示块的大小，用 1024 字节作为单位。因此，0 表示 1024 字节的块，1 表示 2048 字节的块，如此等等。目前 `s_log_frag_size` 字段与 `s_log_block_size` 字段相等，因为块片还没有实现。

`s_blocks_per_group`、`s_frags_per_group` 与 `s_inodes_per_group` 字段分别存放每个块组中的块数、片数及索引节点数。

一些磁盘块保留给超级用户（或由 `s_def_resuid` 和 `s_def_resgid` 字段挑选给某一其他用户或用户组）。即使当普通用户没有空闲块可用时，系统管理员也可以用这些块继续使用 Ext2 文件系统。

`s_mnt_count`、`s_max_mnt_count`、`s_lastcheck` 及 `s_checkinterval` 字段使系统启动时自动地检查 Ext2 文件系统。在预定义的安装操作数完成之后，或自最后一次一致性检查以来预定义的时间已经用完，这些字段就导致 `e2fsck` 执行（两种检查可以一起进行）。如果 Ext2 文件系统还没有被全部卸载（例如系统崩溃以后），或内核在其中发现一些错误，则一致性检查在启动时要强制进行。如果 Ext2 文件系统被安装或未被全部卸载，则 `s_state` 字段存放的值为 0；如果被正常卸载，则这个字段的值为 1；如果包含错误，则值为 2。

组描述符和位图

每个块组都有自己的组描述符，它是一个 `ext2_group_desc` 结构，它的字段在表 18-2 中列出。

表 18-2: Ext2 组描述符的字段

类型	字段	说明
__le32	<code>bg_block_bitmap</code>	块位图的块号
__le32	<code>bg_inode_bitmap</code>	索引节点位图的块号
__le32	<code>bg_inode_table</code>	第一个索引节点表块的块号
__le16	<code>bg_free_blocks_count</code>	组中空闲块的个数
__le16	<code>bg_free_inodes_count</code>	组中空闲索引节点的个数
__le16	<code>bg_used_dirs_count</code>	组中目录的个数
__le16	<code>bg_pad</code>	按字对齐
__le32 [3]	<code>bg_reserved</code>	用 null 填充 24 个字节

当分配新索引节点和数据块时,会用到**bg_free_blocks_count**、**bg_free_inodes_count**和**bg_used_dirs_count**字段。这些字段确定在最合适的块中给每个数据结构进行分配。位图是位的序列,其中值0表示对应的索引节点块或数据块是空闲的,1表示占用。因为每个位图必须存放在一个单独的块中,又因为块的大小可以是1024、2048或4096字节,因此,一个单独的位图描述8192、16384或32768个块的状态。

索引节点表

索引节点表由一连串连续的块组成,其中每一块包含索引节点的一个预定义号。索引节点表第一个块的块号存放在组描述符的**bg_inode_table**字段中。

所有索引节点的大小相同,即128字节。一个1024字节的块可以包含8个索引节点,一个4096字节的块可以包含32个索引节点。为了计算出索引节点表占用了多少块,用一个组中的索引节点总数(存放在超级块的**s_inodes_per_group**字段中)除以每块中的索引节点数。

每个Ext2索引节点为**ext2_innode**结构,其字段如表18-3所示。

表 18-3: Ext2 磁盘索引节点的字段

类型	字段	说明
__le16	i_mode	文件类型和访问权限
__le16	i_uid	拥有者标识符
__le32	i_size	以字节为单位的文件长度
__le32	i_atime	最后一次访问文件的时间
__le32	i_ctime	索引节点最后改变的时间
__le32	i_mtime	文件内容最后改变的时间
__le32	i_dtime	文件删除的时间
__le16	i_gid	用户组标识符
__le16	i_links_count	硬链接计数器
__le32	i_blocks	文件的数据块数
__le32	i_flags	文件标志
union	osd1	特定的操作系统信息
__le32 [EXT2_N_BLOCKS]	i_block	指向数据块的指针
__le32	i_generation	文件版本(当网络文件系统访问文件时使用)
__le32	i_file_acl	文件访问控制列表

表 18-3: Ext2 磁盘索引节点的字段 (续)

类型	字段	说明
__le32	i_dir_acl	目录访问控制列表
__le32	i_faddr	片的地址
union	osd2	特定的操作系统信息

与 POSIX 规范相关的很多字段类似于 VFS 索引节点对象的相应字段，这已在第十二章的“索引节点对象”一节中讨论过。其余的字段与 Ext2 的特殊实现相关，主要处理块的分配。

特别地，`i_size` 字段存放以字节为单位的文件的有效长度，而 `i_blocks` 字段存放已分配给文件的数据块数（以 512 字节为单位）。

`i_size` 和 `i_blocks` 的值没有必然的联系。因为一个文件总是存放在整数块中，一个非空文件至少接受一个数据块（因为还没实现片）且 `i_size` 可能小于 $512 \times i_blocks$ 。另一方面，我们将在本章后面的“文件的洞”一节中看到，一个文件可能包含有洞。在那种情况下，`i_size` 可能大于 $512 \times i_blocks$ 。

`i_blocks` 字段是具有 `EXT2_N_BLOCKS`（通常是 15）个指针元素的一个数组，每个元素指向分配给文件的数据块（参见本章后面的“数据块寻址”一节）。

留给 `i_size` 字段的 32 位把文件的大小限制到 4GB。事实上，`i_size` 字段的最高位没有使用，因此，文件的最大长度限制为 2GB。然而，Ext2 文件系统包含一种“脏技巧”，允许像 AMD 的 Opteron 和 IBM 的 PowerPC G5 这样的 64 位体系结构使用大型文件。从本质上说，索引节点的 `i_dir_acl` 字段（普通文件没有使用）表示 `i_size` 字段的 32 位扩展。因此，文件的大小作为 64 位整数存放在索引节点中。Ext2 文件系统的 64 位版本与 32 位版本在某种程度上兼容，因为在 64 位体系结构上创建的 Ext2 文件系统可以安装在 32 位体系结构上，反之亦然。但是，在 32 位体系结构上不能访问大型文件，除非以 `O_LARGEFILE` 标志打开文件（参见第十二章“`open()` 系统调用”一节）。

回忆一下，VFS 模型要求每个文件有不同的索引节点号。在 Ext2 中，没有必要在磁盘上存放文件的索引节点号与相应块号之间的转换，因为后者的值可以从块组号和它在索引节点表中的相对位置而得出。例如，假设每个块组包含 4096 个索引节点，我们想知道索引节点 13021 在磁盘上的地址。在这种情况下，这个索引节点属于第三个块组，它的磁盘地址存放在相应索引节点表的第 733 个表项中。正如你看到的，索引节点号是 Ext2 例程用来快速搜索磁盘上合适的索引节点描述符的一个关键字。

索引节点的增强属性

Ext2索引节点的格式对于文件系统设计者就好像一件紧身衣，索引节点的长度必须是2的幂，以免造成存放索引节点表的块内碎片。实际上，一个Ext2索引节点的128个字符空间中充满了信息，只有少许空间可以增加新的字段。另一方面，将索引节点的长度增加至256不仅相当浪费，而且使用不同索引节点长度的Ext2文件系统之间还会造成兼容问题。

引入增强属性 (*extended attribute*) 就是要克服上面的问题。这些属性存放在索引节点之外的磁盘块中。索引节点的 `i_file_acl` 字段指向一个存放增强属性的块。具有同样增强属性的不同索引节点可以共享同一个块。

每个增强属性有一个名称和值。两者都编码为变长字符数组，并由 `ext2_xattr_entry` 描述符来确定。图 18-2 表示 Ext2 中增强属性块的结构。每个属性分成两部分：在块首部的是 `ext2_xattr_entry` 描述符与属性名，而属性值则在块尾部。块前面的表项按照属性名称排序，而值的位置是固定的，因为它们是由属性的分配次序决定的。

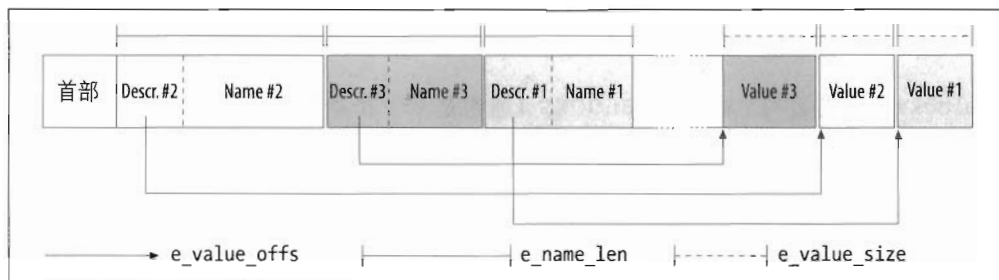


图 18-2: 含增强属性的块的结构

有很多系统调用用来设置、取得、列表和删除一个文件的增强属性。系统调用 `setxattr()`、`lsetxattr()` 和 `fsetxattr()` 设置文件的增强属性，它们在符号链接的处理与文件限定的方式（或者传递路径名或者是文件描述符）上根本不同。类似地，系统调用 `getxattr()`、`lgetxattr()` 和 `fgetxattr()` 返回增强属性的值。系统调用 `listxattr()`、`llistxattr()` 和 `flistxattr()` 则列出一个文件的所有增强属性。最后，系统调用 `removexattr()`、`lremovexattr()` 和 `fremovexattr()` 从文件删除一个增强属性。

访问控制列表

很早以前访问控制列表就被建议用来改善 Unix 文件系统的保护机制。不是将文件的用户分成三类：拥有者、组和其他，访问控制列表 (*access control list, ACL*) 可以与每个

文件关联。有了这种列表，用户可以为他的文件限定可以访问的用户（或用户组）名称以及相应的权限。

Linux 2.6 通过索引节点的增强属性完整实现 ACL。实际上，增强属性主要就是为了支持 ACL 才引入的。因此，能让你处理文件 ACL 的库函数 `chacl()`、`setfacl()` 和 `getfacl()` 就是通过上一节中介绍的 `setxattr()` 和 `getxattr()` 系统调用实现的。

不幸的是，在 POSIX 1003.1 系列标准内，定义安全增强属性的工作组所完成的成果从没有正式成为新的 POSIX 标准。因此现在，不同的类 Unix 文件系统都支持 ACL，但不同的实现之间有一些微小的差别。

各种文件类型如何使用磁盘块

Ext2 所认可的文件类型（普通文件、管道文件等）以不同的方式使用数据块。有些文件不存放数据，因此根本就不需要数据块。本节讨论每种文件类型的存储要求，如表 18-4 所示。

表 18-4：Ext2 文件类型

文件类型	说明
0	未知
1	普通文件
2	目录
3	字符设备
4	块设备
5	命名管道
6	套接字
7	符号链接

普通文件

普通文件是最常见的情况，本章主要关注它。但普通文件只有在开始有数据时才需要数据块。普通文件在刚创建时是空的，并不需要数据块，也可以用 `truncate()` 或 `open()` 系统调用清空它。这两种情况是相同的，例如，当你发出一个包含字符串 `>filename` 的 shell 命令时，shell 创建一个空文件或截断一个现有文件。

目录

Ext2以一种特殊的文件实现了目录，这种文件的数据块把文件名和相应的索引节点号存放在一起。特别说明的是，这样的数据块包含了类型为 `ext2_dir_entry_2` 的结构。表 18-5 列出了这个结构的字段。因为该结构最后一个 `name` 字段是最大为 `EXT2_NAME_LEN`（通常是 255）个字符的变长数组，因此这个结构的长度是可变的。此外，因为效率的原因，目录项的长度总是 4 的倍数，并在必要时用 `null` 字符（`\0`）填充文件名的末尾。`name_len` 字段存放实际的文件名长度（参见图 18-3）。

表 18-5: Ext2 目录项中的字段

类型	字段	说明
<code>__le32</code>	<code>inode</code>	索引节点号
<code>__le16</code>	<code>rec_len</code>	目录项长度
<code>__u8</code>	<code>name_len</code>	文件名长度
<code>__u8</code>	<code>file_type</code>	文件类型
<code>char [EXT2_NAME_LEN]</code>	<code>name</code>	文件名

`file_type` 字段存放指定文件类型的值（见表 18-4）。`rec_len` 字段可以被解释为指向下一个有效目录项的指针：它是偏移量，与目录项的起始地址相加就得到下一个有效目录项的起始地址。为了删除一个目录项，把它的 `inode` 字段置为 0 并适当地增加前一个有效目录项 `rec_len` 字段的值就足够了。仔细看一下图 18-3 的 `rec_len` 字段，你会发现 `oldfile` 项已被删除，因为 `usr` 的 `rec_len` 字段被置为 `12+16`（`usr` 和 `oldfile` 目录项的长度）。

	inode	rec_len	name_len	file_type	name
0	21	12	1	2	. \0 \0 \0
12	22	12	2	2	. \0 \0
24	53	16	5	2	h o m e \0 \0 \0
40	67	28	3	2	u s r \0
52	0	16	7	1	o l d f i l e \0
68	34	12	4	2	s b i n

图 18-3: Ext2 目录的一个例子

符号链接

如前所述，如果符号链接的路径名小于等于 60 个字符，就把它存放在索引节点的 `i_blocks` 字段，该字段是由 15 个 4 字节整数数组成的数组，因此无需数据块。但是，如果路径名大于 60 个字符，就需要一个单独的数据块。

设备文件、管道和套接字

这些类型的文件不需要数据块。所有必要的信息都存放在索引节点中。

Ext2 的内存数据结构

为了提高效率，当安装 Ext2 文件系统时，存放在 Ext2 分区的磁盘数据结构中的大部分信息被拷贝到 RAM 中，从而使内核避免了后来的很多读操作。那么一些数据结构如何经常更新呢？让我们考虑一些基本的操作：

- 当一个新文件被创建时，必须减少 Ext2 超级块中 `s_free_inodes_count` 字段的值和相应的组描述符中 `bg_free_inodes_count` 字段的值。
- 如果内核给一个现有的文件追加一些数据，以使分配给它的数据块数因此也增加，那么就on须修改 Ext2 超级块中 `s_free_blocks_count` 字段的值和组描述符中 `bg_free_blocks_count` 字段的值。
- 即使仅仅重写一个现有文件的部分内容，也要对 Ext2 超级块的 `s_wtime` 字段进行更新。

因为所有的 Ext2 磁盘数据结构都存放在 Ext2 分区的块中，因此，内核利用页高速缓存来保持它们最新（参见第十五章中的“把脏页写入磁盘”一节）。

对于与 Ext2 文件系统以及文件相关的每种数据类型，表 18-6 详细说明了在磁盘上用来表示数据的数据结构、在内存中内核所使用的数据结构以及决定使用多大容量高速缓存的经验方法。频繁更新的数据总是存放在高速缓存，也就是说，这些数据一直存放在内存并包含在页高速缓存中，直到相应的 Ext2 分区被卸载。内核通过让缓冲区的引用计数器一直大于 0 来达到此目的。

表 18-6: Ext2 数据结构 VFS 映像

类型	磁盘数据结构	内存数据结构	缓存模式
超级块	<code>ext2_super_block</code>	<code>ext2_sb_info</code>	总是缓存
组描述符	<code>ext2_group_desc</code>	<code>ext2_group_desc</code>	总是缓存

表 18-6: Ext2 数据结构的 VFS 映像 (续)

类型	磁盘数据结构	内存数据结构	缓存模式
块位图	块中的位数组	缓冲区中的位数组	动态
索引节点位图	块中的位数组	缓冲区中的位数组	动态
索引节点	ext2_inode	ext2_inode_info	动态
数据块	字节数组	VFS 缓冲区	动态
空闲索引节点	ext2_inode	无	从不缓存
空闲块	字节数组	无	从不缓存

在任何高速缓存中不保存“从不缓存”的数据，因为这种数据表示无意义的信息。相反，“总是缓存”的数据也总在 RAM 中，这样就不必从磁盘读数据了（但是，数据必须周期性地写回磁盘）。除了这两种极端模式外，还有一种动态模式。在动态模式下，只要相应的对象（索引节点、数据块或位图）还在使用，它就保存在高速缓存中；而当文件关闭或数据块被删除后，页框回收算法会从高速缓存中删除有关数据。

有意思的是，索引节点与块位图并不永久保存在内存里，而是需要时从磁盘读。有了页高速缓存，最近使用的磁盘块保存在内存里，这样可以避免很多磁盘读（参见第十五章“把块存放在页高速缓存中”一节）（注 2）。

Ext2 的超级块对象

在第十二章“超级块对象”一节我们介绍过，VFS 超级块的 `s_fs_info` 字段指向一个包含文件系统信息的数据结构。对于 Ext2，该字段指向 `ext2_sb_info` 类型的结构，它包含如下信息：

- 磁盘超级块中的大部分字段
- `s_sbh` 指针，指向包含磁盘超级块的缓冲区的缓冲区首部
- `s_es` 指针，指向磁盘超级块所在的缓冲区
- 组描述符的个数 `s_desc_per_block`，可以放在一个块中
- `s_group_desc` 指针，指向一个缓冲区（包含组描述符的缓冲区）首部数组（通常一项就够）

注 2: 在 Linux 2.4 和早期的版本中，最近使用的索引节点和块位图被保存在特殊高速缓存的有限空间里。

- 其他与安装状态、安装选项等有关的数据

图 18-4 表示的是与 Ext2 超级块和组描述符有关的缓冲区与缓冲区首部和 `ext2_sb_info` 数据结构之间的关系。

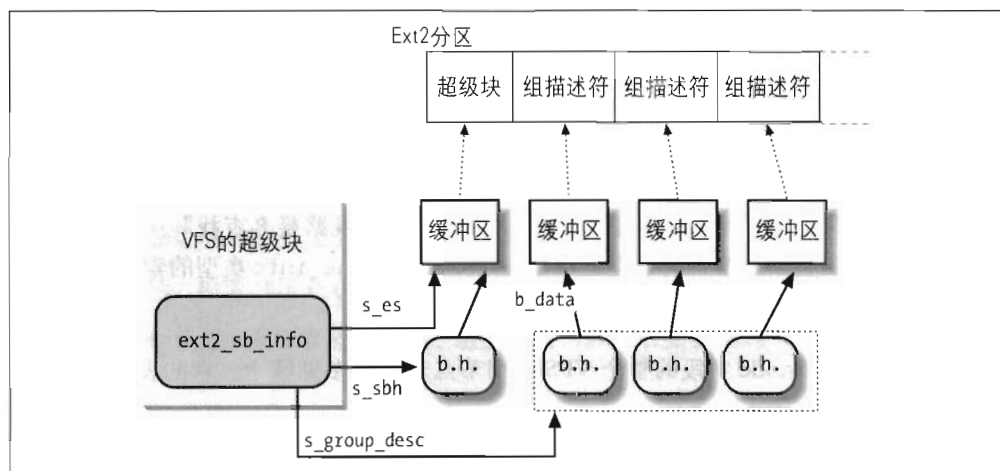


图 18-4: `ext2_sb_info` 数据结构

当内核安装 Ext2 文件系统时，它调用 `ext2_fill_super()` 函数来为数据结构分配空间，并写入从磁盘读取的数据（参见第十二章“安装普通文件系统”一节）。这里是对该函数的一个简要说明，只强调缓冲区与描述符的内存分配。

1. 分配一个 `ext2_sb_info` 描述符，将其地址当作参数传递并存放在超级块的 `s_fs_info` 字段。
2. 调用 `__bread()` 在缓冲区页中分配一个缓冲区和缓冲区首部。然后从磁盘读入超级块存放在缓冲区中。在第十五章的“在页高速缓存中搜索块”一节我们讨论过，如果一个块已在页高速缓存的缓冲区页而且是最新的，那么无需再分配。将缓冲区首部地址存放在 Ext2 超级块对象的 `s_sbh` 字段。
3. 分配一个字节数组，每组一个字节，把它的地址存放在 `ext2_sb_info` 描述符的 `s_debts` 字段（参见本章后面的“创建索引节点”一节）。
4. 分配一个数组用于存放缓冲区首部指针，每个组描述符一个，把该数组地址存放在 `ext2_sb_info` 的 `s_group_desc` 字段。
5. 重复调用 `__bread()` 分配缓冲区，从磁盘读入包含 Ext2 组描述符的块。把缓冲区首部地址存放在上一步得到的 `s_group_desc` 数组中。

6. 为根目录分配一个索引节点和目录项对象，为超级块建立相应的字段，从而能够从磁盘读入根索引节点对象。

很显然，`ext2_fill_super()`函数返回后，分配的所有数据结构都保存在内存里，只有当Ext2文件系统卸载时才会被释放。当内核必须修改Ext2超级块的字段时，它只要把新值写入相应缓冲区内的相应位置然后将该缓冲区标记为脏即可。

Ext2 的索引节点对象

在打开文件时，要执行路径名查找。对于不在目录项高速缓存内的路径名元素，会创建一个新的目录项对象和索引节点对象（参见第十二章“标准路径名查找”一节）。当VFS访问一个Ext2磁盘索引节点时，它会创建一个`ext2_inode_info`类型的索引节点描述符。该描述符包含下列信息：

- 存放在`vfs_inode`字段的整个VFS索引节点对象（参见第十二章的表12-3）
- 磁盘索引节点对象结构中的大部分字段（不保存在VFS索引节点中）
- 索引节点对应的`i_block_group`块组索引（参见本章前面“Ext2磁盘数据结构”一节）
- `i_next_alloc_block`和`i_next_alloc_goal`字段，分别存放着最近为文件分配的磁盘块的逻辑块号与物理块号
- `i_prealloc_block`和`i_prealloc_count`字段，用于数据块预分配（参见本章后面“分配数据块”一节）
- `xattr_sem`字段，一个读写信号量，允许增强属性与文件数据同时读入
- `i_acl`和`i_default_acl`字段，指向文件的ACL

当处理Ext2文件时，`alloc_inode`超级块方法是由`ext2_alloc_inode()`函数实现的。它首先从`ext2_inode_cache` slab分配器高速缓存得到一个`ext2_inode_info`描述符，然后返回在这个`ext2_inode_info`描述符中的索引节点对象的地址。

创建 Ext2 文件系统

在磁盘上创建一个文件系统通常有两个阶段。第二步格式化磁盘，以使磁盘驱动程序可以读和写磁盘上的块。现在的硬磁盘已经由厂家预先格式化，因此不需要重新格式化；在Linux上可以使用`superformat`或`fdformat`等实用程序对软盘进行格式化。第二步才涉及创建文件系统，这意味着建立本章前面详细描述的结构。

Ext2 文件系统是由实用程序 *mke2fs* 创建的。*mke2fs* 采用下列缺省选项，用户可以用命令行的标志修改这些选项：

- 块大小：1024 字节（小文件系统的缺省值）
- 片大小：块的大小（块的分片还没有实现）
- 所分配的索引节点个数：每 8192 字节的组分配一个索引节点
- 保留块的百分比：5%

mke2fs 程序执行下列操作：

1. 初始化超级块和组描述符。
2. 作为选择，检查分区是否包含有缺陷的块，如果有，就创建一个有缺陷块的链表。
3. 对于每个块组，保留存放超级块、组描述符、索引节点表及两个位图所需要的所有磁盘块。
4. 把索引节点位图和每个块组的数据映射位图都初始化为 0。
5. 初始化每个块组的索引节点表。
6. 创建 */root* 目录。
7. 创建 *lost+found* 目录，由 *e2fsck* 使用这个目录把丢失和找到的缺陷块连接起来。
8. 在前两个已经创建的目录所在的块组中，更新块组中的索引节点位图和数据块位图。
9. 把有缺陷的块（如果存在）组织起来放在 *lost+found* 目录中。

让我们看一下 *mke2fs* 是如何以缺省选项初始化 Ext2 的 1.44 MB 软盘的。

软盘一旦被安装，VFS 就把它看作由 1412 个块组成的一个卷，每块大小为 1024 字节。为了查看磁盘的内容，我们可以执行如下 Unix 命令：

```
$ dd if=/dev/fd0 bs=1k count=1440 | od -tx1 -Ax > /tmp/dump_hex
```

从而获得了 */tmp* 目录下的一个文件，这个文件包含十六进制的软盘内容的转储（注 3）。

通过查看 *dump_hex* 文件我们可以看到，由于软盘有限的容量，一个单独的块组描述符就足够了。我们还注意到保留的块数为 72（1440 块的 5%），并且根据缺省选项，索引节点表必须为每 8192 个字节设置一个索引节点，也就是有 184 个索引节点存放在 23 个块中。

注 3： 使用 *dumpe2fs* 和 *debugfs* 实用程序也可以获得有关 Ext2 文件系统的一些信息。

表 18-7 总结了按缺省选项如何在软盘上建立 Ext2 文件系统。

表 18-7: 软盘的 Ext2 块分配

块	内容
0	引导块
1	超级块
2	包含一个单独的块组描述符的块
3	数据块位图
4	索引节点位图
5~27	索引节点表: 5~10 —— 保留 (2 是 root); 11 —— lost+found; 12~184 —— 空闲
28	根目录 (包括 “.”、 “..” 及 “lost+found”)
29	lost+found 目录 (包括 “.” 和 “..”)
30~40	给 lost+found 目录预分配保留的块
41~1439	空闲块

Ext2 的方法

在第十二章所描述的关于 VFS 的很多方法在 Ext2 都有相应的实现。因为对所有的方法都进行描述将需要整整一本书, 因此我们仅仅简单地回顾一下在 Ext2 中所实现的方法。一旦你真正搞明白了磁盘和内存数据结构, 你就应当能理解实现这些方法的 Ext2 函数的代码。

Ext2 超级块的操作

很多 VFS 超级块操作在 Ext2 中都有具体的实现, 这些方法为 `alloc_inode`、`destroy_inode`、`read_inode`、`write_inode`、`delete_inode`、`put_super`、`write_super`、`statfs`、`remount_fs` 和 `clear_inode`。超级块方法的地址存放在 `ext2_sops` 指针数组中。

Ext2 索引节点的操作

一些 VFS 索引节点的操作在 Ext2 中都有具体的实现, 这取决于索引节点所指的文件类型。

Ext2 的普通文件和目录文件的索引节点操作见表 18-8。每个方法的目的是在第十二章的“索

引节点对象”一节有介绍。表中没有列出普通文件和目录中未定义的方法（NULL 指针）。回忆一下，如果方法未定义，VFS 要么调用通用函数，要么什么也不做。普通文件与目录的 Ext2 方法地址分别存放在 `ext2_file_inode_operations` 和 `ext2_dir_inode_operations` 表中。

表 18-8: Ext2 普通文件与目录的索引节点操作

VFS 索引节点操作	普通文件	目录
<code>create</code>	NULL	<code>ext2_create()</code>
<code>lookup</code>	NULL	<code>ext2_lookup()</code>
<code>link</code>	NULL	<code>ext2_link()</code>
<code>unlink</code>	NULL	<code>ext2_unlink()</code>
<code>symlink</code>	NULL	<code>ext2_symlink()</code>
<code>mkdir</code>	NULL	<code>ext2_mkdir()</code>
<code>rmdir</code>	NULL	<code>ext2_rmdir()</code>
<code>mknod</code>	NULL	<code>ext2_mknod()</code>
<code>rename</code>	NULL	<code>ext2_rename()</code>
<code>truncate</code>	<code>ext2_truncate()</code>	NULL
<code>permission</code>	<code>ext2_permission()</code>	<code>ext2_permission()</code>
<code>setattr</code>	<code>ext2_setattr()</code>	<code>ext2_setattr()</code>
<code>setxattr</code>	<code>generic_setxattr()</code>	<code>generic_setxattr()</code>
<code>getxattr</code>	<code>generic_getxattr()</code>	<code>generic_getxattr()</code>
<code>listxattr</code>	<code>ext2_listxattr()</code>	<code>ext2_listxattr()</code>
<code>removexattr</code>	<code>generic_removexattr()</code>	<code>generic_removexattr()</code>

Ext2 的符号链接的索引节点操作见表 18-9（省略未定义的方法）。实际上有两种符号链接：快速符号链接（路径名全部存放在索引节点内）与普通符号链接（较长的路径名）。因此，有两套索引节点操作，分别存放在 `ext2_fast_symlink_inode_operations` 和 `ext2_symlink_inode_operations` 表中。

表 18-9: Ext2 的快速及普通符号链接的索引节点操作

VFS 索引节点操作	快速符号链接	普通符号链接
<code>readlink</code>	<code>generic_readlink()</code>	<code>generic_readlink()</code>
<code>follow_link</code>	<code>ext2_follow_link()</code>	<code>page_follow_link_light()</code>
<code>put_link</code>	NULL	<code>page_put_link()</code>

表 18-9: Ext2 的快速及普通符号链接的索引节点操作 (续)

VFS 索引节点操作	快速符号链接	普通符号链接
setxattr	generic_setxattr()	generic_setxattr()
getxattr	generic_getxattr()	generic_getxattr()
listxattr	ext2_listxattr()	ext2_listxattr()
removexattr	generic_removexattr()	generic_removexattr()

如果索引节点指的是一个字符设备文件、块设备文件或命名管道（参见第十九章中的“FIFO”一节），那么这种索引节点的操作不依赖于文件系统，其操作分别位于 `chrdev_inode_operations`、`blkdev_inode_operations` 和 `fifo_inode_operations` 表中。

Ext2 的文件操作

表 18-10 列出了 Ext2 文件系统特定的文件操作。正如你看到的，一些 VFS 方法是由很多文件系统共用的通用函数实现的。这些方法的地址存放在 `ext2_file_operations` 表中。

表 18-10: Ext2 文件操作

VFS 文件操作	Ext2 方法
llseek	generic_file_llseek()
read	generic_file_read()
write	generic_file_write()
aio_read	generic_file_aio_read()
aio_write	generic_file_aio_write()
ioctl	ext2_ioctl()
mmap	generic_file_mmap()
open	generic_file_open()
release	ext2_release_file()
fsync	ext2_sync_file()
readv	generic_file_readv()
writev	generic_file_writev()
sendfile	generic_file_sendfile()

注意，Ext2 的 read 和 write 方法是分别通过 generic_file_read() 和 generic_file_write() 函数实现的。这两个函数在第十五章的“从文件中读取数据”和“写入文件”两节进行了描述。

管理 Ext2 磁盘空间

文件在磁盘的存储不同于程序员所看到的文件，这表现在两个方面：块可以分散在磁盘上（尽管文件系统尽力保持块连续存放以提高访问速度），以及程序员看到的文件似乎比实际的文件大，这是因为程序可以把洞引入文件（通过 lseek() 系统调用）。

在本节，我们将介绍 Ext2 文件系统如何管理磁盘空间，也就是说，如何分配和释放索引节点和数据块。有两个主要的问题必须考虑：

- 空间管理必须尽力避免文件碎片，也就是说，避免文件在物理上存放于几个小的、不相邻的盘块上。文件碎片增加了对文件的连续读操作的平均时间，因为在读操作期间，磁头必须频繁地重新定位（注 4）。这个问题类似于在第八章的“伙伴系统算法”一节中所讨论的 RAM 的外部碎片问题。
- 空间管理必须考虑效率，也就是说，内核应该能从文件的偏移量快速地导出 Ext2 分区上相应的逻辑块号。为了达到此目的，内核应该尽可能地限制对磁盘上寻址表的访问次数，因为对该表的访问会极大地增加文件的平均访问时间。

创建索引节点

ext2_new_inode() 函数创建 Ext2 磁盘的索引节点，返回相应的索引节点对象的地址（或失败时为 NULL）。该函数谨慎地选择存放该新索引节点的块组；它将无联系的目录散放在不同的组，而且同时把文件存放在父目录的同一组。为了平衡普通文件数与块组中的目录数，Ext2 为每一个块组引入“债（debt）”参数。

该函数作用于两个参数：dir，一个目录对应的索引节点对象的地址，新创建的索引节点必须插入到这个目录中；mode，要创建的索引节点的类型。后一个参数还包含一个 MS_SYNCHRONOUS 标志，该标志请求当前进程一直挂起，直到索引节点被分配。该函数执行如下操作：

1. 调用 new_inode() 分配一个新的 VFS 索引节点对象，并把它的 i_sb 字段初始化为存放在 dir->i_sb 中的超级块地址。然后把它追加到正在用的索引节点链表与超级块链表中（参见第十二章“索引节点对象”一节）。

注 4： 请注意，把一个文件跨过块组进行分片是一件坏事，而为了在一个块中存放多个文件把块进行分片（还没实现）是一件好事，二者之间是不同的。

2. 如果新的索引节点是一个目录,函数就调用 `find_group_orlov()` 为目录找到一个合适的块组(注5)。该函数执行如下试探法:
 - a. 以文件系统根 `root` 为父目录的目录应该分散在各个组。这样,函数在这些块组去查找一个组,它的空闲索引节点数和空闲块数比平均值高。如果没有这样的组则跳到第 2c 步。
 - b. 如果满足下列条件,嵌套目录(父目录不是文件系统根 `root`) 就应被存放到父目录组:
 - 该组没有包含太多的目录
 - 该组有足够多的空闲索引节点
 - 该组有一点小“债”。(块组的债存放在一个 `ext2_sb_info` 描述符的 `s_debts` 字段所指向的计数器数组中。每当一个新目录加入,债加一;每当其他类型的文件加入,债减一)如果父目录组不满足这些条件,那么选择第一个满足条件的组。如果没有满足条件的组,则跳到第 2c 步。
 - c. 这是一个“退一步”原则,当找不到合适的组时使用。函数从包含父目录的块组开始选择第一个满足条件的块组,这个条件是:它的空闲索引节点数比每块组空闲索引节点数的平均值大。
3. 如果新索引节点不是个目录,则调用 `find_group_other()`,在有空闲索引节点的块组中给它分配一个。该函数从包含父目录的组开始往下找。具体如下:
 - a. 从包含父目录 `dir` 的块组开始,执行快速的对数查找。这种算法要查找 $\log(n)$ 个块组,这里 n 是块组总数。该算法一直向前查找直到找到一个可用的块组,具体如下:如果我们把开始的块组称为 i ,那么,该算法要查找的块组为 $i \bmod (n)$, $i+1 \bmod (n)$, $i+1+2 \bmod (n)$, $i+1+2+4 \bmod (n)$, 等等。
 - b. 如果该算法没有找到含有空闲索引节点的块组,就从包含父目录 `dir` 的块组开始执行彻底的线性查找。
4. 调用 `read_inode_bitmap()` 得到所选块组的索引节点位图,并从中寻找第一个空位,这样就得到了第一个空闲磁盘索引节点号。
5. 分配磁盘索引节点:把索引节点位图中的相应位置位,并把含有这个位图的缓冲区标记为脏。此外,如果文件系统安装时指定了 `MS_SYNCHRONOUS` 标志(参见第十二章中的“安装普通文件系统”一节),则调用 `sync_dirty_buffer()` 开始 I/O 写操作并等待,直到写操作终止。

注5: 安装 Ext2 文件系统还可以带有一个选项标志,它强制内核使用一个更简单、更老式的分配策略,该策略是由 `find_group_dir()` 函数实现的。

6. 减小组描述符的 `bg_free_inodes_count` 字段。如果新的索引节点是一个目录，则增加 `bg_used_dirs_count` 字段，并把含有这个组描述符的缓冲区标记为脏。
7. 依据索引节点指向的是普通文件或目录，相应增减超级块内 `s_debts` 数组中的组计数器。
8. 减小 `ext2_sb_info` 数据结构中的 `s_freeinodes_counter` 字段；而且如果新索引节点是目录，则增大 `ext2_sb_info` 数据结构的 `s_dirs_counter` 字段。
9. 将超级块的 `s_dirt` 标志置 1，并把包含它的缓冲区标记为脏。
10. 把 VFS 超级块对象的 `s_dirt` 字段置 1。
11. 初始化这个索引节点对象的字段。特别是，设置索引节点号 `i_no`，并把 `xtime.tv_sec` 的值拷贝到 `i_atime`、`i_mtime` 及 `i_ctime`。把这个块组的索引赋给 `ext2_inode_info` 结构的 `i_block_group` 字段。关于这些字段的含义请参考表 18-3。
12. 初始化这个索引节点对象的访问控制列表（ACL）。
13. 将新索引节点对象插入散列表 `inode_hashtable`，调用 `mark_inode_dirty()` 把该索引节点对象移进超级块脏索引节点链表（参见第十二章“索引节点对象”一节）。
14. 调用 `ext2_preread_inode()` 从磁盘读入包含该索引节点的块，将它存入页高速缓存。进行这种预读是因为最近创建的索引节点可能会被很快写入。
15. 返回新索引节点对象的地址。

删除索引节点

用 `ext2_free_inode()` 函数删除一个磁盘索引节点，把磁盘索引节点表示为索引节点对象，其地址作为参数来传递。内核在进行一系列的清除操作（包括清除内部数据结构和文件中的数据）之后调用这个函数。具体来说，它在下列操作完成之后才执行：索引节点对象已经从散列表中删除，指向这个索引节点的最后一个硬链接已经从适当的目录中删除，文件的长度截为 0 以回收它的所有数据块（参见本章后面“释放数据块”一节）。函数执行下列操作：

1. 调用 `clear_inode()`，它依次执行如下步骤：
 - a. 删除与索引节点关联的“间接”脏缓冲区（参见后面“数据块寻址”一节）。它们都存放在一个链表中，该链表的首部在 `address_space` 对象 `inode->i_data` 的 `private_list` 字段（参见第十五章“`address_space` 对象”一节）。
 - b. 如果索引节点的 `I_LOCK` 标志置位，则说明索引节点中的某些缓冲区正处于 I/O 数据传送中；于是，函数挂起当前进程，直到这些 I/O 数据传送结束。

- c. 调用超级块对象的 `clear_inode` 方法（如果已定义），但 Ext2 文件系统没有定义这个方法。
 - d. 如果索引节点指向一个设备文件，则从设备的索引节点链表中删除索引节点对象，这个链表要么在 `cdev` 字符设备描述符的 `cdev` 字段（参见第十三章“字符设备驱动程序”一节），要么在 `block_device` 块设备描述符的 `bd_inodes` 字段（参见第十四章“块设备”一节）。
 - e. 把索引节点的状态置为 `I_CLEAR`（索引节点对象的内容不再有意义）。
2. 从每个块组的索引节点号和索引节点数计算包含这个磁盘索引节点的块组的索引。
 3. 调用 `read_inode_bitmap()` 得到索引节点位图。
 4. 增加组描述符的 `bg_free_inodes_count` 字段。如果删除的索引节点是一个目录，那么也要减小 `bg_used_dirs_count` 字段。把这个组描述符所在的缓冲区标记为脏。
 5. 如果删除的索引节点是一个目录，就减小 `ext2_sb_info` 结构的 `s_dirs_counter` 字段，把超级块的 `s_dirt` 标志置 1，并把它所在的缓冲区标记为脏。
 6. 清除索引节点位图中这个磁盘索引节点对应的位，并把包含这个位图的缓冲区标记为脏。此外，如果文件系统以 `MS_SYNCHRONIZE` 标志安装，则调用 `sync_dirty_buffer()` 并等待，直到在位图缓冲区上的写操作终止。

数据块寻址

每个非空的普通文件都由一组数据块组成。这些块或者由文件内的相对位置（它们的文件块号）来标识，或者由磁盘分区内的位置（它们的逻辑块号）来标识（参见第十四章的“块设备的处理”一节）。

从文件内的偏移量 f 导出相应数据块的逻辑块号需要两个步骤：

1. 从偏移量 f 导出文件的块号，即在偏移量 f 处的字符所在的块索引。
2. 把文件的块号转化为相应的逻辑块号。

因为 Unix 文件不包含任何控制字符，因此，导出文件的第 f 个字符所在的文件块号是相当容易的，只是用 f 除以文件系统块的大小，并取整即可。

例如，让我们假定块的大小为 4KB。如果 f 小于 4096，那么这个字符就在文件的第一个数据块中，其文件的块号为 0。如果 f 等于或大于 4096 而小于 8192，则这个字符就在文件块号为 1 的数据块中，以此类推。

只用关注文件的块号确实不错。但是，由于 Ext2 文件的数据块在磁盘上不必是相邻的，因此把文件的块号转化为相应的逻辑块号可不是这么直截了当的。

因此，Ext2 文件系统必须提供一种方法，用这种方法可以在磁盘上建立每个文件块号与相应逻辑块号之间的关系。在索引节点内部部分实现了这种映射（回到了 AT&T Unix 的早期版本）。这种映射也涉及一些包含额外指针的专用块，这些块用来处理大型文件的索引节点的扩展。

磁盘索引节点的 `i_block` 字段是一个有 `EXT2_N_BLOCKS` 个元素且包含逻辑块号的数组。在下面的讨论中，我们假定 `EXT2_N_BLOCKS` 的默认值为 15。如图 18-5 所示，这个数组表示一个大型数据结构的初始化部分。正如从图中所看到的，数组的 15 个元素有 4 种不同的类型：

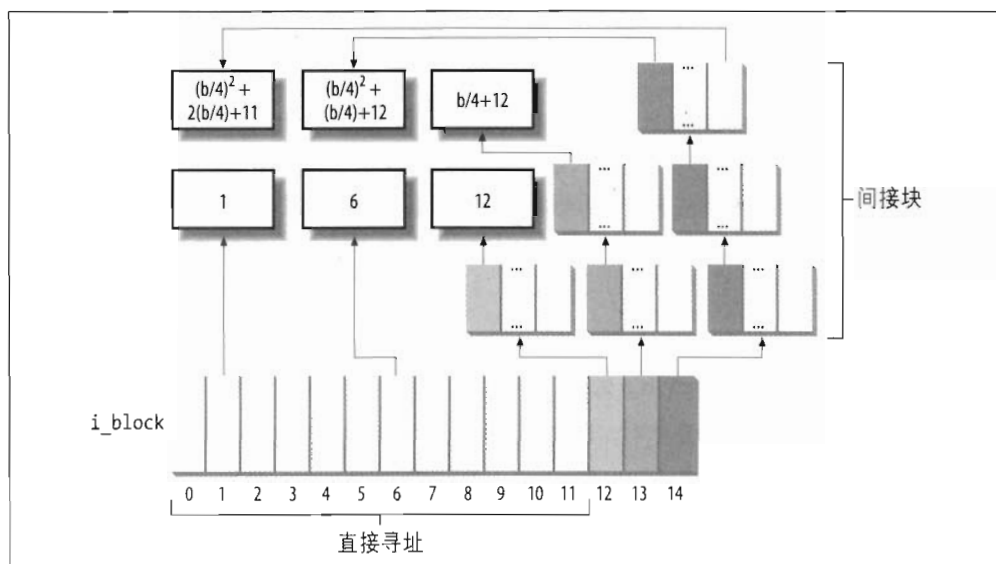


图 18-5：对文件的数据块进行寻址的数据结构

- 最初的 12 个元素产生的逻辑块号与文件最初的 12 个块对应，即对应的文件块号从 0~11。
- 下标 12 中的元素包含一个块的逻辑块号（叫做间接块），这个块表示逻辑块号的一个二级数组。这个数组的元素对应的文件块号从 12~ $b/4+11$ ，这里 b 是文件系统的块大小（每个逻辑块号占 4 个字节，因此我们在式子中用 4 作除数）。因此，内核为了查找指向一个块的指针必须先访问这个元素，然后，在这个块中找到另一个指向最终块（包含文件内容）的指针。

- 下标 13 中的元素包含一个间接块的逻辑块号，而这个块包含逻辑块号的一个二级数组，这个二级数组的数组项依次指向三级数组，这个三级数组存放的才是文件块号对应的逻辑块号，范围从 $b/4+12 \sim (b/4)^2+(b/4)+11$ 。
- 最后，下标 14 中的元素使用三级间接索引，第四级数组中存放的才是文件块号对应的逻辑块号，范围从 $(b/4)^2+(b/4)+12 \sim (b/4)^3+(b/4)^2+(b/4)+11$ 。

在图 18-5 中，块内的数字表示相应的文件块号。箭头（表示存放在数组元素中的逻辑块号）指示了内核如何通过间接块找到包含文件实际内容的块。

注意这种机制是如何支持小文件的。如果文件需要的数据块小于 12，那么两次磁盘访问就可以检索到任何数据：一次是读磁盘索引节点 `i_block` 数组的一个元素，另一次是读所需要的数据块。对于大文件来说，可能需要三四次的磁盘访问才能找到需要的块。实际上，这是一种最坏的估计，因为目录项、索引节点、页高速缓存都有助于极大地减少实际访问磁盘的次数。

还要注意文件系统的块大小是如何影响寻址机制的，因为大的块允许 Ext2 把更多的逻辑块号存放在一个单独的块中。表 18-11 显示了对每种块大小和每种寻址方式所存放文件大小的上限。例如，如果块的大小是 1024 字节，并且文件包含的数据最多为 268KB，那么，通过直接映射可以访问文件最初的 12KB 数据，通过简单的间接映射可以访问剩余的 13~268KB 的数据。大于 2GB 的大型文件通过指定 `O_LARGEFILE` 打开标志必须在 32 位体系结构上进行打开。

表 18-11：数据块寻址的文件大小上界

块大小	直接	一次间接	二次间接	三次间接
1024	12 KB	268 KB	64.26 MB	16.06 GB
2048	24 KB	1.02 MB	513.02 MB	256.5 GB
4096	48 KB	4.04 MB	4 GB	~ 4 TB

文件的洞

文件的洞(*file hole*)是普通文件的一部分，它是一些空字符但没有存放在磁盘的任何数据块中。洞是 Unix 文件一直存在的一个特点。例如，下列的 Unix 命令创建了第一个字节是洞的文件。

```
$ echo -n "X" | dd of=/tmp/hole bs=1024 seek=6
```

现在，`/tmp/hole` 有 6145 个字符（6144 个空字符加一个 X 字符），然而，这个文件在磁盘上只占一个数据块。

引入文件的洞是为了避免磁盘空间的浪费。它们被广泛地用在数据库应用中，更一般地说，用于在文件上进行散列的所有应用。

文件洞在 Ext2 中的实现是基于动态数据块的分配的：只有当进程需要向一个块写数据时，才真正把这个块分配给文件。每个索引节点的 `i_size` 字段定义程序所看到的文件大小，包括洞，而 `i_blocks` 字段存放分配给文件有效的数据块数（以 512 字节为单位）。

在前面 `dd` 命令的例子中，假定 `/tmp/hole` 文件创建在块大小为 4096 的 Ext2 分区上。其相应磁盘索引节点的 `i_size` 字段存放的数为 6144，而 `i_blocks` 字段存放的数为 8（因为每 4096 字节的块包含 8 个 512 字节的块）。`i_block` 数组的第二个元素（对应块的文件块号为 1）存放已分配块的逻辑块号，而数组中的其他元素都为空（参看图 18-6）。

分配数据块

当内核要分配一个数据块来保存 Ext2 普通文件的数据时，就调用 `ext2_get_block()` 函数。如果块不存在，该函数就自动为文件分配块。请记住，每当内核在 Ext2 普通文件上执行读或写操作时就调用这个函数（参见第十六章“从文件中读取数据”和“写入文件”两节）；显然，这个函数只在页高速缓存内没有相应的块时才被调用。

`ext2_get_block()` 函数处理在“数据块寻址”一节描述的数据结构，并在必要时调用 `ext2_alloc_block()` 函数在 Ext2 分区真正搜索一个空闲块。如果需要，该函数还为间接寻址分配相应的块（参见图 18-5）。

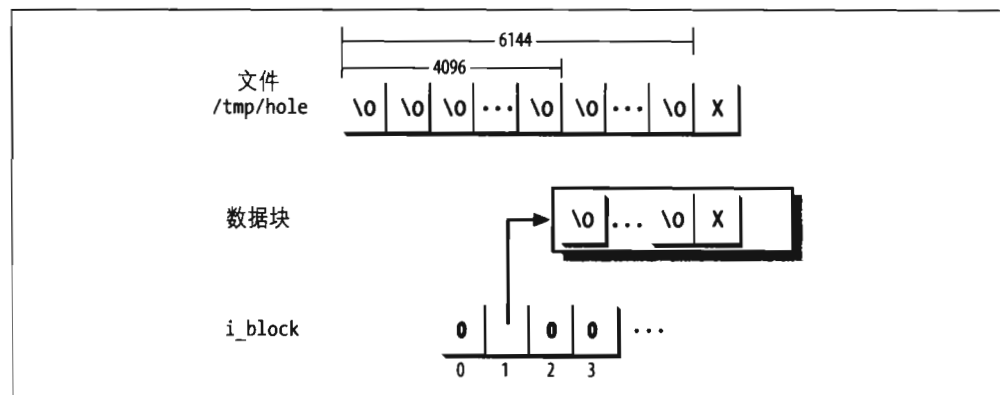


图 18-6：起始部分有洞的文件

为了减少文件的碎片，Ext2 文件系统尽力在已分配给文件的最后一个块附近找一个新块

分配给该文件。如果失败，Ext2文件系统又在包含这个文件索引节点的块组中搜寻一个新的块。作为最后一个办法，可以从其他一个块组中获得空闲块。

Ext2文件系统使用数据块的预分配策略。文件并不仅仅获得所需要的块，而是获得一组多达8个邻接的块。ext2_inode_info结构的i_prealloc_count字段存放预分配给某一文件但还没有使用的数据块数，而i_prealloc_block字段存放下一次要使用的预分配块的逻辑块号。当下列情况发生时，释放预分配而一直没有使用的块：当文件被关闭时，当文件被缩短时，或者当一个写操作相对于引发块预分配的写操作不是顺序的时。

ext2_alloc_block()函数接收的参数为指向索引节点对象的指针、目标(goal)和存放错误码的变量地址。目标是一个逻辑块号，表示新块的首选位置。ext2_getblk()函数根据下列的试探法设置目标参数：

1. 如果正被分配的块与前面已分配的块有连续的文件块号，则目标就是前一块的逻辑块号加1。这很有意义，因为程序所看到的连续的块在磁盘上将会是相邻的。
2. 如果第一条规则不适用，并且至少给文件已分配了一个块，那么目标就是这些块的逻辑块号中的一个。更确切地说，目标是已分配块的逻辑块号，位于文件中待分配块之前。
3. 如果前面的规则都不适用，那么目标就是文件索引节点所在的块组中第一个块的逻辑块号（不必空闲）。

ext2_alloc_block()函数检查目标是否指向文件的预分配块中的一块。如果是，就分配相应的块并返回它的逻辑块号；否则，丢弃所有剩余的预分配块并调用ext2_new_block()。

ext2_new_block()函数用下列策略在Ext2分区内搜寻一个空闲块：

1. 如果传递给ext2_alloc_block()的首选块（目标块）是空闲的，就分配它。
2. 如果目标为忙，就检查首选块后的其余块之中是否有空闲的块。
3. 如果在首选块附近没有找到空闲块，就从包含目标的块组开始，查找所有的块组。对每个块组：
 - a. 寻找至少有8个相邻空闲块的一个组块。
 - b. 如果没有找到这样的一组块，就寻找一个单独的空闲块。

只要找到一个空闲块，搜索就结束。在结束前，ext2_new_block()函数还尽力在找到的空闲块附近的块中找8个空闲块进行预分配，并把磁盘索引节点的i_prealloc_block和i_prealloc_count字段置为适当的块位置及块数。

释放数据块

当进程删除一个文件或把它的长度截为 0 时，其所有数据块必须回收。这是通过调用 `ext2_truncate()` 函数（其参数是这个文件的索引节点对象的地址）来完成的。实际上，这个函数扫描磁盘索引节点的 `i_block` 数组，以确定所有数据块的位置和间接寻址用的块的位置。然后反复调用 `ext2_free_blocks()` 函数释放这些块。

`ext2_free_blocks()` 函数释放一组含有一个或多个相邻块的数据块。除 `ext2_truncate()` 调用它外，当丢弃文件的预分配块时也主要调用它（参见前面的“分配数据块”一节）。函数参数如下：

`inode`

文件的索引节点对象的地址。

`block`

要释放的第一个块的逻辑块号。

`count`

要释放的相邻块数。

这个函数对每个要释放的块执行下列操作：

1. 获得要释放块所在块组的块位图。
2. 把块位图中要释放的块的对应位清 0，并把位图所在的缓冲区标记为脏。
3. 增加块组描述符的 `bg_free_blocks_count` 字段，并把相应的缓冲区标记为脏。
4. 增加磁盘超级块的 `s_free_blocks_count` 字段，并把相应的缓冲区标记为脏，把超级块对象的 `s_dirt` 标记置位。
5. 如果 Ext2 文件系统安装时设置了 `MS_SYNCHRONOUS` 标志，则调用 `sync_dirty_buffer()` 并等待，直到对这个位图缓冲区的写操作终止。

Ext3 文件系统

在本节我们将简单描述从 Ext2 发展而来的增强型文件系统，即 Ext3。这个新的文件系统在设计时曾秉持两个简单的概念：

- 成为一个日志文件系统（参见下一节）
- 尽可能与原来的 Ext2 文件系统兼容

Ext3 完全达到了这两个目标。尤其是，它很大程度上是基于 Ext2 的，因此，它在磁盘上的数据结构从本质上与 Ext2 文件系统的结构是相同的。事实上，如果 Ext3 文件系统已经被彻底卸载，那么就可以把它作为 Ext2 文件系统来重新安装；反之，创建 Ext2 文件系统的日志并把它作为 Ext3 文件系统来重新安装，也是一种简单、快速的操作。

由于 Ext3 与 Ext2 之间的兼容性，本章前面几节的很多描述也适用于 Ext3。因此，本节我们集中于 Ext3 所提供的新特点——“日志”。

日志文件系统

随着磁盘变得越来越大，传统 Unix 文件系统（像 Ext2）的一种设计选择证明是不相称的。从第十四章我们已经知道，对文件系统块的更新可能在内存保留相当长的时间后才刷新到磁盘。因此，像断电故障或系统崩溃这样不可预测的事件可能导致文件系统处于不一致状态。为了克服这个问题，每个传统的 Unix 文件系统在安装之前都要进行检查，如果它没有被正常卸载，那么，就有一个特定的程序执行彻底、耗时的检查，并修正磁盘上文件系统的所有数据结构。

例如，Ext2 文件系统的状态存放在磁盘上超级块的 `s_mount_state` 字段中。由启动脚本调用 `e2fsck` 实用程序检查存放在这个字段中的值；如果它不等于 `EXT2_VALID_FS`，说明文件系统没有正常卸载，因此，`e2fsck` 开始检查文件系统的所有磁盘数据结构。

显然，检查文件系统一致性所花费的时间主要取决于要检查的文件数和目录数；因此，它也取决于磁盘的大小。如今，随着文件系统达到几百个 GB，一次一致性检查就可能花费数个小时。造成的停机时间对任何生产环境和高可用服务器都是无法接受的。

日志文件系统的目标就是避免对整个文件系统进行耗时的一致性检查，这是通过查看一个特殊的磁盘区达到的，因为这种磁盘区包含所谓日志 (*journal*) 的最新磁盘写操作。系统出现故障后，安装日志文件系统只不过是几秒钟的事。

Ext3 日志文件系统

Ext3 日志所隐含的思想就是对文件系统进行的任何高级修改都分两步进行。首先，把待写块的一个副本存放在日志中；其次，当发往日志的 I/O 数据传送完成时（简而言之，把数据提交到日志），块就被写入文件系统。当发往文件系统的 I/O 数据传送终止时（把数据提交给文件系统），日志中的块副本就被丢弃。

当从系统故障中恢复时，`e2fsck` 程序区分下列两种情况：

提交到日志之前系统故障发生。与高级修改相关的块副本或者从日志中丢失，或者是不完整的；在这两种情况下，*e2fsck* 都忽略它们。

提交到日志之后系统故障发生。块的副本是有效的，且 *e2fsck* 把它们写入文件系统。

在第一种情况下，对文件系统的高级修改被丢失，但文件系统的状态还是一致的。在第二种情况下，*e2fsck* 应用于整个高级修改，因此，修正由于把未完成的 I/O 数据传送到文件系统而造成的任何不一致。

不要对日志文件系统有太多的期望。它只能确保系统调用级的一致性。例如，当你正在发出几个 `write()` 系统调用拷贝一个大型文件时发生了系统故障，这将会使拷贝操作中中断，因此，复制的文件就会比原来的文件短。

因此，日志文件系统通常不把所有的块都拷贝到日志中。事实上，每个文件系统都由两种块组成：包含所谓元数据 (*metadata*) 的块和包含普通数据的块。在 Ext2 和 Ext3 的情形中，有六种元数据：超级块、块组描述符、索引节点、用于间接寻址的块 (间接块)、数据位图块和索引节点位图块。其他的文件系统可能使用不同的元数据。

很多日志文件系统 (如 SGI 的 XFS 以及 IBM 的 JFS) 都限定自己把影响元数据的操作记入日志。事实上，元数据的日志记录足以恢复磁盘文件系统数据结构的一致性。然而，因为文件的数据块不记入日志，因此就无法防止系统故障造成的文件内容的损坏。

不过，可以把 Ext3 文件系统配置为把影响文件系统元数据的操作和影响文件数据块的操作都记入日志。因为把每种写操作都记入日志会导致极大的性能损失，因此，Ext3 让系统管理员决定应当把什么记入日志；具体来说，它提供三种不同的日志模式：

日志 (*Journal*)

文件系统所有数据和元数据的改变都被记入日志。这种模式减少了丢失每个文件修改的机会，但是它需要很多额外的磁盘访问。例如，当一个新文件被创建时，它的所有数据块都必须复制一份作为日志记录。这是最安全和最慢的 Ext3 日志模式。

预定 (*Ordered*)

只有对文件系统元数据的改变才被记入日志。然而，Ext3 文件系统把元数据和相关的数据块进行分组，以便在元数据之前把数据块写入磁盘。这样，就可以减少文件内数据损坏的机会；例如，确保增大文件的任何写访问都完全受日志的保护。这是缺省的 Ext3 日志模式。

写回 (*Writeback*)

只有对文件系统元数据的改变才被记入日志，这是在其他日志文件系统中发现的方法，也是最快的模式。

Ext3 文件系统的日志模式由 *mount* 系统命令的一个选项来指定。例如，为了在 */jdisk* 安装点对存放在 */dev/sda2* 分区上的 Ext3 文件系统以“写回”模式进行安装，系统管理员可以键入如下命令：

```
# mount -t ext3 -o data=writeback /dev/sda2 /jdisk
```

日志块设备层

Ext3 日志通常存放在名为 *.journal* 的隐藏文件中，该文件位于文件系统的根目录。

Ext3 文件系统本身不处理日志，而是利用所谓日志块设备 (*Journaling Block Device*, *JBD*) 的通用内核层。现在，只有 Ext3 使用 JBD 层，而其他文件系统可能在将来才使用它。

JBD 层是相当复杂的软件部分。Ext3 文件系统调用 JBD 例程，以确保在系统万一出现故障时它的后续操作不会损坏磁盘数据结构。然而，JBD 典型地使用同一磁盘来把 Ext3 文件系统所做的改变记入日志，因此，它与 Ext3 一样易受系统故障的影响。换言之，JBD 也必须保护自己免受任何系统故障引起的日志损坏。

因此，Ext3 与 JBD 之间的交互本质上基于三个基本单元：

日志记录

描述日志文件系统一个磁盘块的一次更新。

原子操作处理

包括文件系统的一次高级修改对应的日志记录；一般来说，修改文件系统的每个系统调用都引起一次单独的原子操作处理。

事务

包括几个原子操作处理，同时，原子操作处理的日志记录对 *e2fsck* 标记为有效。

日志记录

日志记录 (*log record*) 本质上是文件系统将要发出的一个低级操作的描述。在某些日志文件系统中，日志记录只包括操作所修改的字节范围及字节在文件系统中的起始位置。然而，JBD 层使用的日志记录由低级操作所修改的整个缓冲区组成。这种方式可能浪费很多日志空间（例如，当低级操作仅仅改变位图的一个位时），但是，它还是相当快的，因为 JBD 层直接对缓冲区和缓冲区首部进行操作。

因此，日志记录在日志内部表示为普通的数据块（或元数据）。但是，每个这样的块都是与类型为 *journal_block_tag_t* 的小标签相关联的，这种小标签存放块在文件系统中的逻辑块号和几个状态标志。

随后, 只要一个缓冲区得到 JBD 的关注, 或者因为它属于日志记录, 或者因为它是一个数据块, 该数据块应当在相应的元数据之前刷新到磁盘 (处于“预定”模式), 那么, 内核把 `journal_head` 数据结构加入到缓冲区首部。在这种情况下, 缓冲区首部的 `b_private` 字段存放 `journal_head` 数据结构的地址, 并把 `BH_JBD` 标志置位 (参见第十五章“块缓冲区和缓冲区首部”一节)。

原子操作处理

修改文件系统的任一系统调用通常都被划分为操纵磁盘数据结构的一系列低级操作。

例如, 假定 Ext3 必须满足用户把一个数据块追加到普通文件的请求。文件系统层必须确定文件的最后一个块, 定位文件系统中的空闲块, 更新适当块组内的数据块位图, 存放新块的逻辑块号在文件的索引节点或间接寻址块中, 写新块的内容, 并在最后更新索引节点的几个字段。你可以看到, 追加操作转换为对文件系统数据块和元数据块很多低级的操作。

现在, 仅仅想象一下, 如果在追加操作的中间一些低级操作已经执行, 另一些还没有执行, 而系统出现了故障会发生什么事情。当然, 对于影响两个或多个文件的高级操作 (例如, 把文件从一个目录移到另一个目录), 情况会更糟。

为了防止数据损坏, Ext3 文件系统必须确保每个系统调用以原子的方式进行处理。原子操作处理 (*atomic operation handle*) 是对磁盘数据结构的一组低级操作, 这组低级操作对应一个单独的高级操作。当从系统故障中恢复时, 文件系统确保要么整个高级操作起作用, 要么没有一个低级操作起作用。

任何原子操作处理都用类型为 `handle_t` 的描述符来表示。为了开始一个原子操作, Ext3 文件系统调用 `journal_start()` JBD 函数, 该函数在必要时分配一个新的原子操作处理并把它插入到当前的事务中 (见下一节)。因为对磁盘的任何低级操作都可能挂起进程, 因此, 活动原子操作处理的地址存放在进程描述符的 `journal_info` 字段中。为了通知原子操作已经完成, Ext3 文件系统调用 `journal_stop()` 函数。

事务

出于效率的原因, JBD 层对日志的处理采用分组的方法, 即把属于几个原子操作处理的日志记录分组放在一个单独的事务 (*transaction*) 中。此外, 与一个处理相关的所有日志记录都必须包含在同一个事务中。

一个事务的所有日志记录存放在日志的连续块中。JBD 层把每个事务作为整体来处理。

例如,只有当包含在一个事务的日志记录中的所有数据都提交给文件系统时才回收该事务所使用的块。

事务一旦被创建,它就能接受新处理的日志记录。当下列情况之一发生时,事务就停止接受新处理:

- 固定的时间已经过去,典型情况下为 5s。
- 日志中没有空闲块留给新处理

事务是由类型为 `transaction_t` 的描述符来表示的。其最重要的字段为 `t_state`,该字段描述事务的当前状态。

从本质上说,事务可以是:

完成的

包含在事务中的所有日志记录都已经从物理上写入日志。当从系统故障中恢复时, `e2fsck` 考虑日志中每个完成的事务,并把相应的块写入文件系统。在这种情况下, `t_state` 字段存放值 `T_FINISHED`。

未完成的

包含在事务中的日志记录至少还有一个没有从物理上写入日志,或者新的日志记录还正在追加到事务中。在系统故障的情况下,存放在日志中的事务映像很可能不是最新的。因此,当从系统故障中恢复时, `e2fsck` 不信任日志中未完成的事务,并跳过它们。在这种情况下, `i_state` 存放下列值之一:

`T_RUNNING`

还在接受新的原子操作处理。

`T_LOCKED`

不接受新的原子操作处理,但其中的一些还没有完成。

`T_FLUSH`

所有的原子操作处理都已完成,但一些日志记录还正在写入日志。

`T_COMMIT`

原子操作处理的所有日志记录都已经写入磁盘,但在日志中,事务仍然被标记为完成。

在任何时刻,日志可能包含多个事务,但其中只有一个处于 `T_RUNNING` 状态,即它是活动事务 (*active transaction*)。所谓活动事务就是正在接受由 Ext3 文件系统发出的新原子操作处理的请求。

日志中的几个事务可能是未完成的，因为包含相关日志记录的缓冲区还没有写入日志。

如果事务完成，说明所有日志记录已被写入日志，但是一部分相应的缓冲区还没有写入文件系统。只有当 JDB 层确认日志记录描述的所有缓冲区都已成功写入 Ext3 文件系统时，一个完成的事务才能从日志中删除。

日志如何工作

让我们用一个例子来试图解释日志如何工作：Ext3 文件系统层接受向普通文件写一些数据块的请求。

你可能很容易猜到，我们打算详细描述 Ext3 文件系统层和 JDB 层的每个单独操作。那将会涉及太多问题！但是，我们描述本质的操作：

1. `write()` 系统调用服务例程触发与 Ext3 普通文件相关的文件对象的 `write` 方法。对于 Ext3 来说，这个方法是由 `generic_file_write()` 函数实现的，这已在第十六章“写入文件”一节进行了描述。
2. `generic_file_write()` 函数几次调用 `address_space` 对象的 `prepare_write` 方法，写方法涉及的每个数据页都调用一次。对 Ext3 来说，这个方法是由 `ext3_prepare_write()` 函数实现的。
3. `ext3_prepare_write()` 函数调用 `journal_start()` JBD 函数开始一个新的原子操作。这个原子操作处理被加到活动事务中。实际上，原子操作处理是在第一次调用 `journal_start()` 函数时创建的。后续的调用确认进程描述符的 `journal_info` 字段已经被置位，并使用这个处理。
4. `ext3_prepare_write()` 函数调用第十六章已描述过的 `block_prepare_write()` 函数，传递给它的参数为 `ext3_get_block()` 函数的地址。回想一下，`block_prepare_write()` 负责准备文件页的缓冲区和缓冲区首部。
5. 当内核必须确定 Ext3 文件系统的逻辑块号时，就执行 `ext3_get_block()` 函数。这个函数实际上类似于 `ext2_get_block()`，后者在前面“分配数据块”一节已经描述。但是，有一个主要的差异在于 Ext3 文件系统调用 JDB 层的函数来确保低级操作记入日志：
 - 在对 Ext3 文件系统的元数据块发出低级写操作之前，该函数调用 `journal_get_write_access()`。后一个函数主要把元数据缓冲区加入到活动事务的链表中。但是，它也必须检查元数据是否包含在日志的一个较老的未完成的事务中；在这种情况下，它把缓冲区复制一份以确保老的事务以老的内容提交。

- 在更新元数据块所在的缓冲区之后,Ext3文件系统调用`journal_dirty_metadata()`把元数据缓冲区移到活动事务的适当脏链表中,并在日志中记录这一操作。

注意,由JDB层处理的元数据缓冲区通常并不包含在索引节点的缓冲区的脏链表中,因此,这些缓冲区并不由第十五章描述的正常磁盘高速缓存的刷新机制写入磁盘。

6. 如果Ext3文件系统已经以“日志”模式安装,则`ext3_prepare_write()`函数在写操作触及的每个缓冲区上也调用`journal_get_write_access()`。
7. 控制权回到`generic_file_write()`函数,该函数用存放在用户态地址空间的数据更新页,并调用`address_space`对象的`commit_write`方法。对于Ext3,函数如何实现这个方法取决于Ext3文件的安装方式:
 - 如果Ext3文件系统已经以“日志”模式安装,那么`commit_write`方法是由`ext3_journalled_commit_write()`函数实现的,它对页中的每个数据(不是元数据)缓冲区调用`journal_dirty_metadata()`。这样,缓冲区就包含在活动事务的适当脏链表中,但不包含在拥有者索引节点的脏链表中;此外,相应的日志记录写入日志。最后,`ext3_journalled_commit_write()`调用`journal_stop`通知JBD层原子操作处理已关闭。
 - 如果Ext3文件系统已经以“预定”模式安装,那么`commit_write`方法是由`ext3_ordered_commit_write()`函数实现的,它对页中的每个数据缓冲区调用`journal_dirty_data()`函数以把缓冲区插入到活动事务的适当链表中。JBD层确保在事务中的元数据缓冲区写入之前这个链表中的所有缓冲区写入磁盘。没有日志记录写入日志。然后,`ext3_ordered_commit_write()`函数执行第十五章描述的常规`generic_commit_write()`函数,该函数把数据缓冲区插入拥有者索引节点的脏缓冲区链表中。最后,`ext3_ordered_commit_write()`调用`journal_stop()`通知JBD层原子操作处理已关闭。
 - 如果Ext3文件系统已经以“写回”模式安装,那么`commit_write`方法是由`ext3_writeback_commit_write()`函数实现的,它执行第十五章描述的常规`generic_commit_write()`函数,该函数把数据缓冲区插入拥有者索引节点的脏缓冲区链表中。然后,`ext3_writeback_commit_write()`调用`journal_stop()`通知JBD层原子操作处理已关闭。
8. `write()`系统调用的服务例程到此结束。但是,JBD层还没有完成它的工作。终于,当事务的所有日志记录都物理地写入日志时,我们的事务才完成。然后,执行`journal_commit_transaction()`。
9. 如果Ext3文件系统已经以“预定”模式安装,则`journal_commit_transaction()`函数为事务链表包含的所有数据缓冲区激活I/O数据传送,并等待直到数据传送终止。

10. `journal_commit_transaction()` 函数为包含在事务中的所有元数据缓冲区激活 I/O 数据传送（如果 Ext3 以“日志”模式安装，则也为所有的数据缓冲区激活 I/O 数据传送）。
11. 内核周期性地为日志中每个完成的事务激活检查点活动。检查点主要验证由 `journal_commit_transaction()` 触发的 I/O 数据传送是否已经成功结束。如果是，则从日志中删除事务。

当然，除非发生系统故障，否则日志中的日志记录根本就没有什么积极作用。事实上，只有在系统发生故障时，`e2fsck` 实用程序才扫描存放在文件系统中的日志，并重新安排完成的事务中的日志记录所描述的所有写操作。

第十九章

进程通信



本章介绍用户态的进程之间如何进行同步和交换数据。在第五章我们已经介绍了很多同步的主题，但是参与的对象是内核控制路径，而不是用户态程序。我们在详细讨论了I/O管理和文件系统之后，就可以继续讨论用户态进程的同步。这些进程都要依靠内核来实现彼此之间的同步以及通信。

正如我们在第十二章“Linux文件加锁”一节已经看到的那样，通过创建一个文件（可能是空文件）并使用适当的VFS系统调用对该文件加锁和解锁就可以在用户态进程之间实现某种同步。通过把数据存放在使用锁保护的临时文件中就可以在进程之间实现类似的数据共享，然而这种方法的代价很高，因为它需要访问磁盘文件系统。出于这个原因，所有的Unix内核都包含一组系统调用，这些系统调用不用与文件系统打交道就可以支持进程通信；而且，已经开发了几个封装函数并将其加入到适当的库来加速进程对内核发出同步请求。

通常，应用程序员有使用不同通信机制的各种需求。这里列出Unix系统提供的进程间通信的基本机制：

管道和FIFO（命名管道）

最适合在进程之间实现生产者/消费者的交互。有些进程向管道中写入数据，而另外一些进程则从管道中读出数据。这将在“管道”与“FIFO”一节讨论。

信号量

顾名思义，这是在第五章中的“信号量”一节讨论过的内核信号量的用户态版本。这将在“System V IPC”一节讨论。

消息

允许进程在预定义的消息队列中读和写消息来交换消息（小块数据）。Linux 内核提供两种不同的消息版本：System V IPC 消息（参见“System V IPC”一节）和 POSIX 消息一节（参见“POSIX 消息队列”一节）。

共享内存区

允许进程通过共享内存块来交换信息。在必须共享大量数据的应用中，这可能是最高效的进程通信形式。这将在“System V IPC”一节讨论。

套接字

允许不同计算机上的进程通过网络交换数据。套接字还可以用作相同主机上的进程之间的通信工具；例如，X Window 系统图形接口就是使用套接字来允许客户端和 X 服务器交换数据的。

管道

管道 (*pipe*) 是所有 Unix 都愿意提供的一种进程间通信机制。管道是进程之间的一个单向数据流：一个进程写入管道的所有数据都由内核定向到另一个进程，另一个进程由此就可以从管道中读取数据。

在 Unix 的命令 shell 中，可以使用“|”操作符来创建管道。例如，下面的语句通知 shell 创建两个进程，并使用一个管道把这两个进程连接在一起：

```
$ ls | more
```

① 第一个进程（执行 *ls* 程序）的标准输出被重定向到管道中；② 第二个进程（执行 *more* 程序）从这个管道中读取输入。

注意，执行下面这两条命令也可以得到相同的结果：

```
$ ls > temp  
$ more < temp
```

第一个命令把 *ls* 的输出重定向到一个普通文件中；接下来，第二个命令强制 *more* 从这个普通文件中读取输入。当然，通常使用管道比使用临时文件更方便，这是因为：

- shell 语句比较短，也比较简单。
- 没有必要创建将来还必须删除的临时普通文件。

使用管道

管道被看作是打开的文件，但在已安装的文件系统中没有相应的映像。可以使用pipe()系统调用来创建一个新管道，这个系统调用返回一对文件描述符，然后进程通过fork()把这两个描述符传递给它的子进程，由此与子进程共享管道。进程可以在read()系统调用中使用第一个文件描述符从管道中读取数据，同样也可以在write()系统调用中使用第二个文件描述符向管道中写入数据。

POSIX 只定义了半双工的管道，因此即使pipe()系统调用返回了两个描述符，每个进程在使用一个文件描述符之前仍得把另外一个文件描述符关闭。如果所需要的是双向数据流，那么进程必须通过两次调用pipe()来使用两个不同的管道。

有些 Unix 系统，例如 System V Release 4，实现了全双工的管道。在全双工管道中，允许两个文件描述符既可以被写入也可以被读取，这就有两个双向信息通道。Linux 采用了另外一种解决方法：每个管道的文件描述符仍然都是单向的，但是在使用一个描述符之前不必把另外一个描述符关闭。

让我们回顾一下前面的那个例子。当 shell 命令对 `ls | more` 语句进行解释时，实际上要执行以下操作：

1. 调用 `pipe()` 系统调用；让我们假设 `pipe()` 返回文件描述符 3（管道的读通道）和 4（管道的写通道）。
2. 两次调用 `fork()` 系统调用。
3. 两次调用 `close()` 系统调用来释放文件描述符 3 和 4。

第一个子进程必须执行 `ls` 程序，它执行以下操作：

1. 调用 `dup2(4, 1)` 把文件描述符 4 拷贝到文件描述符 1。从现在开始，文件描述符 1 就代表该管道的写通道。
2. 两次调用 `close()` 系统调用来释放文件描述符 3 和 4。
3. 调用 `execve()` 系统调用来执行 `ls` 程序（参见第二十章“exec 函数”一节）。缺省情况下，这个程序要把自己的输出写到文件描述符为 1 的那个文件（标准输出）中，也就是说，写入管道中。

第二个子进程必须执行 `more` 程序；因此，该进程执行以下操作：

1. 调用 `dup2(3, 0)` 把文件描述符 3 拷贝到文件描述符 0。从现在开始，文件描述符 0 就代表管道的读通道。

2. 两次调用 `close()` 系统调用来释放文件描述符 3 和 4。
3. 调用 `execve()` 系统调用来执行 *more* 程序。缺省情况下，这个程序要从文件描述符为 0 的那个文件（标准输入）中读取输入，也就是说，从管道中读取输入。

在这个简单的例子中，管道完全被两个进程使用。但是，由于管道的这种实现方式，一个管道可以供任意个进程使用（注1）。显然，如果两个或者更多个进程对同一个管道进行读写，那么这些进程必须使用文件加锁机制（参见第十二章中的“Linux文件加锁”一节）或者 IPC 信号量机制（参见本章后面的“IPC 信号量”一节）对自己的访问进行显式的同步。

除了 `pipe()` 系统调用之外，很多 Unix 系统都提供了两个名为 `popen()` 和 `pclose()` 的封装函数来处理在使用管道的过程中产生的所有脏工作。只要使用 `popen()` 函数创建一个管道，就可以使用包含在 C 函数库中的高级 I/O 函数（`fprintf()`，`fscanf()` 等等）对这个管道进行操作。

在 Linux 中，`popen()` 和 `pclose()` 都包含在 C 函数库中。`popen()` 函数接收两个参数：可执行文件的路径名 `filename` 和定义数据传输方向的字符串 `type`。该函数返回一个指向 FILE 数据结构的指针。`popen()` 函数实际上执行以下操作：

1. 使用 `pipe()` 系统调用创建一个新管道。
2. 创建一个新进程，该进程又执行以下操作：
 - a. 如果 `type` 是 `r`，就把与管道的写通道相关的文件描述符拷贝到文件描述符 1（标准输出）；否则，如果 `type` 是 `w`，就把与管道的读通道相关的文件描述符拷贝到文件描述符 0（标准输入）。
 - b. 关闭 `pipe()` 返回的文件描述符。
 - c. 调用 `execve()` 系统调用执行 `filename` 所指定的程序。
3. 如果 `type` 是 `r`，就关闭与管道的写通道相关的文件描述符；否则，如果 `type` 是 `w`，就关闭与管道的读通道相关的文件描述符。
4. 返回 FILE 文件指针所指向的地址，这个指针指向仍然打开的管道所涉及的任一文件描述符。

注 1：由于大部分 shell 都提供只连接两个进程的管道，所以应用程序要通过管道连接多于两个的进程就必须使用诸如 C 之类的编程语言自行编写。

在 `popen()` 函数被调用之后，父进程和子进程就可以通过管道交换信息：父进程可以使用该函数所返回的 `FILE` 指针来读（如果 `type` 是 `r`）写（如果 `type` 是 `w`）数据。子进程所执行的程序分别把数据写入标准输出或从标准输入中读取数据。

`pclose()` 函数接收 `popen()` 所返回的文件指针作为参数，它会简单地调用 `wait4()` 系统调用并等待 `popen()` 所创建的进程结束。

管道数据结构

我们现在又一次在系统调用的层次考虑问题。只要管道一被创建，进程就可以使用 `read()` 和 `write()` 这两个 VFS 系统调用来访问管道。因此，对于每个管道来说，内核都要创建一个索引节点对象和两个文件对象，一个文件对象用于读，另外一个对象用于写。当进程希望从管道中读取数据或向管道中写入数据时，必须使用适当的文件描述符。

当索引节点指的是管道时，其 `i_pipe` 字段指向一个如表 19-1 所示的 `pipe_inode_info` 结构。

表 19-1: `pipe_inode_info` 结构

类型	字段	说明
<code>struct wait_queue *</code>	<code>wait</code>	管道 /FIFO 等待队列
<code>unsigned int</code>	<code>nrbufs</code>	包含待读数据的缓冲区数
<code>unsigned int</code>	<code>curbuf</code>	包含待读数据的第一个缓冲区的索引
<code>struct pipe_buffer [16]</code>	<code>bufs</code>	管道缓冲区描述符数组
<code>struct page *</code>	<code>tmp_page</code>	高速缓存页框指针
<code>unsigned int</code>	<code>start</code>	当前管道缓冲区读的位置
<code>unsigned int</code>	<code>readers</code>	读进程的标志（或编号）
<code>unsigned int</code>	<code>writers</code>	写进程的标志（或编号）
<code>unsigned int</code>	<code>waiting_writers</code>	在等待队列中睡眠的写进程的个数
<code>unsigned int</code>	<code>r_counter</code>	与 <code>readers</code> 类似，但当等待读取 FIFO 的进程时使用
<code>unsigned int</code>	<code>w_counter</code>	与 <code>writers</code> 类似，但当等待写入 FIFO 的进程时使用
<code>struct fasync_struct *</code>	<code>fasync_readers</code>	用于通过信号进行的异步 I/O 通知
<code>struct fasync_struct *</code>	<code>fasync_writers</code>	用于通过信号进行的异步 I/O 通知

除了一个索引节点对象和两个文件对象之外，每个管道都还有自己的管道缓冲区 (*pipe buffer*)。实际上，它是一个单独的页，其中包含了已经写入管道等待读出的数据。在 Linux 2.6.10 以前，每个管道一个管道缓冲区。而 2.6.11 内核中，管道 (与 FIFO) 的数据缓冲区已有很大改变，每个管道可以使用 16 个管道缓冲区。这个改变大大增强了向管道写大量数据的用户态应用的性能。

`pipe_inode_info` 数据结构的 `bufs` 字段存放一个具有 16 个 `pipe_buffer` 对象的数组，每个对象代表一个管道缓冲区。该对象的字段如表 19-2 所示。

表 19-2: `pipe_buffer` 对象的字段

类型	字段	说明
<code>struct page *</code>	<code>page</code>	管道缓冲区页框的描述符地址
<code>unsigned int</code>	<code>offset</code>	页框内有效数据的当前位置
<code>unsigned int</code>	<code>len</code>	页框内有效数据的长度
<code>struct pipe_buf_operations*</code>	<code>ops</code>	管道缓冲区方法表的地址 (管道缓冲区空时为 <code>NULL</code>)

`ops` 字段指向管道缓冲区方法表 `anon_pipe_buf_ops`，它是一个类型为 `pipe_buf_operations` 的数据结构。实际上，它有三个方法：

`map`

在访问缓冲区数据之前调用。它只在管道缓冲区在高端内存时对管道缓冲区页框调用 `kmap()` (参见第八章“高端内存页框的内核映射”一节)。

`unmap`

不再访问缓冲区数据时调用。它对管道缓冲区页框调用 `kunmap()`。

`release`

当释放管道缓冲区时调用。该方法实现了一个单页内存高速缓存：释放的不是存放缓冲区的那个页框，而是由 `pipe_inode_info` 数据结构 (如果不是 `NULL`) 的 `tmp_page` 字段指向的高速缓存页框。存放缓冲区的页框变成新的高速缓存页框。

16 个缓冲区可以被看作一个整体环形缓冲区：写进程不断向这个大缓冲区追加数据，而读进程则不断移出数据。所有管道缓冲区中当前写入而等待读出的字节数就是所谓的管道大小。为提高效率，仍然要读的数据可以分散在几个未填满的管道缓冲区内：事实上，在上一个管道缓冲区没有足够空间存放新数据时，每个写操作都可能会把数据拷贝到一个新的空管道缓冲区。因此，内核必须记录：

- 下一个待读字节所在的管道缓冲区、页框中的对应偏移量。该管道缓冲区的索引存放在 `pipe_inode_info` 数据结构的 `curbuf` 字段，而偏移量在相应 `pipe_buffer` 对象的 `offset` 字段。
- 第一个空管道缓冲区。它可以通过增加当前管道缓冲区的索引得到（模为 16），并存放在 `pipe_inode_info` 数据结构的 `curbuf` 字段，而存放有效数据的管道缓冲区号存放在 `nrbufs` 字段。

为了避免对管道数据结构的竞争条件，内核使用包含在索引节点对象中的 `i_sem` 信号量。

pipefs 特殊文件系统

管道是作为一组 VFS 对象来实现的，因此没有对应的磁盘映象。在 Linux 2.6 中，把这些 VFS 对象组织为 *pipefs* 特殊文件系统以加速它们的处理（参见第十二章“特殊文件系统”一节）。因为这种文件系统在系统目录树中没有安装点，因此用户根本看不到它。但是，有了 *pipefs*，管道完全被整合到 VFS 层，内核就可以以命名管道或 FIFO 的方式处理它们，FIFO 是以终端用户认可的文件而存在的（参见后面“FIFO”一节）。

`init_pipe_fs()` 函数（一般是在内核初始化期间执行）注册 *pipefs* 文件系统并安装它（参见第十二章“安装普通文件系统”一节）：

```
struct file_system_type pipe_fs_type;
pipe_fs_type.name = "pipefs";
pipe_fs_type.get_sb = pipefs_get_sb;
pipe_fs_type.kill_sb = kill_anon_super;
register_filesystem(&pipe_fs_type);
pipe_mnt = do_kern_mount("pipefs", 0, "pipefs", NULL);
```

表示 *pipefs* 根目录的已安装文件系统对象存放在 `pipe_mnt` 变量中。

创建和撤消管道

`pipe()` 系统调用由 `sys_pipe()` 函数处理，后者又会调用 `do_pipe()` 函数。为了创建一个新的管道，`do_pipe()` 函数执行以下操作：

1. 调用 `get_pipe_inode()` 函数，该函数为 *pipefs* 文件系统管道分配一个索引节点对象并对其进行初始化。具体来说，该函数执行下列操作：
 - a. 在 *pipefs* 文件系统中分配一个新的索引节点。
 - b. 分配 `pipe_inode_info` 数据结构，并把它的地址存放在索引节点的 `i_pipe` 字段。
 - c. 设置 `pipe_inode_info` 的 `curbuf` 和 `nrbufs` 字段为 0，并将 `bufs` 数组中的管道缓冲区对象的所有字段都清 0。

- d. 把 `pipe_inode_info` 结构的 `r_counter` 和 `w_counter` 字段初始化为 1。
 - e. 把 `pipe_inode_info` 结构的 `readers` 和 `writers` 字段初始化为 1。
2. 为管道的读通道分配一个文件对象和一个文件描述符，并把这个文件对象的 `f_flag` 字段设置成 `O_RDONLY`，把 `f_op` 字段初始化成 `read_pipe_fops` 表的地址。
 3. 为管道的写通道分配一个文件对象和一个文件描述符，并把这个文件对象的 `flag` 字段设置成 `O_WRONLY`，把 `f_op` 字段初始化成 `write_pipe_fops` 表的地址。
 4. 分配一个目录项对象，并使用它把两个文件对象和索引节点对象连接在一起（参见第十二章的“通用文件模型”一节）；然后，把新的索引节点插入 `pipefs` 特殊文件系统中。
 5. 把两个文件描述符返回给用户态进程。

发出一个 `pipe()` 系统调用的进程是最初唯一一个可以读写访问新管道的进程。为了表示该管道实际上既有一个读进程，又有一个写进程，就要把 `pipe_inode_info` 数据结构的 `readers` 和 `writers` 字段都初始化为 1。通常，只要相应管道的文件对象仍然由某个进程打开，这两个字段中的每个字段就应该都被设置成 1；如果相应的文件对象已经被释放，那么这个字段就被设置成 0，因为不会再有任何进程访问这个管道。

创建一个新进程并不增加 `readers` 和 `writers` 字段的值，因此这两个值从不超过 1（注 2）。但是，父进程仍然使用的所有文件对象的引用计数器的值都会增加（参见第三章“`clone()`、`fork()`及`vfork()`系统调用”一节）。因此，即使父进程死亡时这个对象都不会被释放，管道仍会一直打开供子进程使用。

只要进程对与管道相关的一个文件描述符调用 `close()` 系统调用，内核就对相应的文件对象执行 `fput()` 函数，这会减少它的引用计数器的值。如果这个计数器变成 0，那么该函数就调用这个文件操作的 `release` 方法（参见第十二章的“`close()`系统调用”和“与进程相关的文件”两节）。

根据文件是与读通道还是与写通道关联，`release` 方法或者由 `pipe_read_release()` 或者由 `pipe_write_release()` 函数来实现。这两个函数都调用 `pipe_release()`，后者把 `pipe_inode_info` 结构的 `readers` 字段或 `writers` 字段设置成 0。`pipe_release()` 还要检查 `readers` 和 `writers` 是否都等于 0。如果是，就调用所有管道缓冲区的 `release` 方法，向伙伴系统（buddy system）释放所有管道缓冲区页框；此外，函数还释放由 `tmp_page` 字段指向的高速缓存页框。否则，`readers` 或者 `writers` 字段不为 0，函数唤醒在管道的等待队列上睡眠的任一进程，以使它们可以识别管道状态的变化。

注 2：正如我们将看到的，当与 FIFO 相关时，`readers` 和 `writers` 字段用作计数器而不是标志。

从管道中读取数据

希望从管道中读取数据的进程发出一个 `read()` 系统调用，为管道的读端指定一个文件描述符。正如在第十二章的“`read()`和`write()`系统调用”一节中描述的那样，内核最终调用与这个文件描述符相关的文件操作表中所找到的`read`方法。在管道的情况下，`read`方法在 `read_pipe_fops` 表中的表项指向 `pipe_read()` 函数。

`pipe_read()` 相当复杂，因为 POSIX 标准定义了管道的读操作的一些要求。表 19-3 概述了所期望的 `read()` 系统调用的行为，该系统调用从一个管道大小（管道缓冲区中待读的字节数）为 p 的管道中读取 n 个字节。

表 19-3: 从一个管道中读取 n 个字节

	至少有一个写进程		非阻塞读	没有写进程
	阻塞读	无睡眠的写者进程		
管道大小 p	睡眠的写者进程	等待某一数据、拷贝它并返回它的大小	返回 <code>-EAGAIN</code>	返回 0
$p=0$	拷贝 n 个字节并返回 n ，当管道缓冲区为空时	等待某一数据、拷贝它并返回它的大小	返回 <code>-EAGAIN</code>	返回 0
$0 < p < n$	等待数据	拷贝 p 个字节并返回 p ；在管道的缓冲区中还剩 0 个字节		
$p \geq n$	拷贝 n 个字节并返回 n ；在管道的缓冲区中还剩 $p-n$ 个字节			

这个系统调用可能以两种方式阻塞当前进程：

- 当系统调用开始时管道缓冲区为空。
- 管道缓冲区没有包含所有请求的字节，写进程在等待缓冲区的空间时曾被置为睡眠。

注意，读操作可以是非阻塞的。在这种情况下，只要所有可用的字节（即使是 0 个）一旦被拷贝到用户地址空间中，读操作就完成（注 3）。

还要注意，只有在管道为空而且当前没有进程正在使用与管道的写通道相关的文件对象时，`read()` 系统调用才会返回 0。

注 3: 非阻塞操作通常都是通过通过在 `open()` 系统调用中指定 `O_NONBLOCK` 标志进行请求。这个方法并不适合管道，因为管道不能被打开；但是，进程可以通过对相应的文件描述符发出一个 `fcntl()` 系统调用来请求对管道执行非阻塞操作。

pipe_read()函数执行以下操作：

1. 获取索引节点的 `i_sem` 信号量。
2. 确定存放在 `pipe_inode_info` 结构 `nrbufs` 字段中的管道大小是否为 0。如果是，说明所有管道缓冲区为空。这时还要确定函数必须返回还是进程在等待时必须被阻塞，直到其他进程向管道中写入一些数据（参见表 19-3）。I/O 操作的类型（阻塞或非阻塞）是通过文件对象的 `f_flags` 字段中的 `O_NONBLOCK` 标志来表示的。如果当前进程必须被阻塞，则函数执行下列操作：
 - a. 调用 `prepare_to_wait()` 把 `current` 加到管道的等待队列 (`pipe_inode_info` 结构的 `wait` 字段)。
 - b. 释放索引节点的信号量。
 - c. 调用 `schedule()`。
 - d. 一旦 `current` 被唤醒，就调用 `finish_wait()` 把它从等待队列中删除，再次获取 `i_sem` 索引节点信号量，然后跳回第 2 步。
3. 从 `pipe_inode_info` 数据结构的 `curbuf` 字段得到当前管道缓冲区索引。
4. 执行管道缓冲区的 `map` 方法。
5. 从管道缓冲区拷贝请求的字节数（如果较小，就是管道缓冲区可用字节数）到用户地址空间。
6. 执行管道缓冲区的 `unmap` 方法。
7. 更新相应 `pipe_buffer` 对象的 `offset` 和 `len` 字段。
8. 如果管道缓冲区已空（`pipe_buffer` 对象的 `len` 字段现在等于 0），则调用管道缓冲区的 `release` 方法释放对应的页框，把 `pipe_buffer` 对象的 `ops` 字段置为 `NULL`，增加在 `pipe_inode_info` 数据结构的 `curbuf` 字段中存放的当前管道缓冲区索引，并减小 `nrbufs` 字段中非空管道缓冲区计数器的值。
9. 如果所有请求字节拷贝完毕，则跳至第 12 步。
10. 目前，还没有把所有请求字节拷贝到用户态地址空间。如果管道大小大于 0（`pipe_inode_info` 的 `nrbufs` 字段不是 `NULL`），则跳到第 3 步。
11. 管道缓冲区内已没有剩余字节。如果至少有一个写进程正在睡眠（即 `pipe_inode_info` 数据结构的 `waiting_writers` 字段大于 0），且读操作是阻塞的，那么调用 `wake_up_interruptible_sync()` 唤醒在管道等待队列中所有睡眠的进程，然后跳至第 2 步。
12. 释放索引节点的 `i_sem` 信号量。

13. 调用 `wake_up_interruptible_sync()` 函数唤醒在管道的等待队列中所有睡眠的写者进程。
14. 返回拷贝到用户地址空间的字节数。

向管道中写入数据

希望向管道中写入数据的进程发出一个 `write()` 系统调用，为管道的写端指定一个文件描述符。内核通过调用适当文件对象的 `write` 方法来满足这个请求；`write_pipe_fops` 表中相应的项指向 `pipe_write()` 函数。

表 19-4 概述了由 POSIX 标准所定义的 `write()` 系统调用的行为，该系统调用请求把 n 个字节写入一个管道中，而该管道在它的缓冲区中有 u 个未用的字节。具体地说，该标准要求涉及少量字节数的写操作必须原子地执行。更确切地说，如果两个或者多个进程并发地在写入一个管道，那么任何少于 4096 个字节（管道缓冲区的大小）的写操作都必须单独完成，而不能与唯一进程对同一个管道的写操作交叉进行。但是，超过 4096 个字节的写操作是可分割的，也可以强制调用进程睡眠。

表 19-4：把 n 个字节写入管道

可用缓冲区的空间 u	至少有一个读进程		没有读进程
	阻塞写	非阻塞写	
$u < n \leq 4096$	等待，直到有 $n-u$ 个字节被释放为止，拷贝 n 个字节，并返回 n	返回 <code>-EAGAIN</code>	发送 <code>SIGPIPE</code> 信号并返回 <code>-EPIPE</code>
$n > 4096$	拷贝 n 个字节（必要时等待）并返回 n	如果 $u > 0$ ，就拷贝 u 个字节并返回 u ，否则就返回 <code>-EAGAIN</code>	
$u \geq n$	拷贝 n 个字节并返回 n		

还有，如果管道没有读进程（也就是说，如果管道的索引节点对象的 `readers` 字段的值是 0），那么任何对管道执行的写操作都会失败。在这种情况下，内核会向写进程发送一个 `SIGPIPE` 信号，并停止 `write()` 系统调用，使其返回一个 `-EPIPE` 错误码，这个错误码就表示我们熟悉的“Broken pipe（损坏的管道）”消息。

`pipe_write()` 函数执行以下操作：

1. 获取索引节点的 `i_sem` 信号量。
2. 检查管道是否至少有一个读进程。如果不是, 就向当前进程发送一个 `SIGPIPE` 信号, 释放索引节点信号量并返回 `-EPIPE` 值。
3. 将 `pipe_inode_info` 数据结构 `curbuf` 和 `nrbufs` 字段相加并减一得到最后写入的管道缓冲区索引。如果该管道缓冲区有足够空间存放待写字节, 就拷入这些数据:
 - a. 执行管道缓冲区的 `map` 方法。
 - b. 把所有字节拷贝到管道缓冲区。
 - c. 执行管道缓冲区的 `unmap` 方法。
 - d. 更新相应 `pipe_buffer` 对象的 `len` 字段。
 - e. 跳至第 11 步。
4. 如果 `pipe_inode_info` 数据结构的 `nrbufs` 字段等于 16, 就表明没有空闲管道缓冲区来存放待写字节。这种情况下:
 - a. 如果写操作是非阻塞的, 跳至第 11 步, 结束并返回错误码 `-EAGAIN`。
 - b. 如果写操作是阻塞的, 将 `pipe_inode_info` 结构的 `waiting_writers` 字段加 1, 调用 `prepare_to_wait()` 将当前操作加入管道等待队列 (`pipe_inode_info` 结构的 `wait` 字段), 释放索引节点信号量, 调用 `schedule()`。一旦唤醒, 则调用 `finish_wait()` 从等待队列中移出当前操作, 重新获得索引节点信号量, 递减 `waiting_writers` 字段, 然后跳回第 4 步。
5. 现在至少有一个空缓冲区, 将 `pipe_inode_info` 数据结构的 `curbuf` 和 `nrbufs` 字段相加得到第一个空管道缓冲区索引。
6. 除非 `pipe_inode_info` 数据结构的 `tmp_page` 字段不是 `NULL`, 否则从伙伴系统中分配一个新页框。
7. 从用户态地址空间拷贝多达 4096 个字节到页框 (如果必要, 在内核态线性地址空间作临时映射)。
8. 更新与管道缓冲区关联的 `pipe_buffer` 对象的字段: 将 `page` 字段设为页框描述符的地址, `ops` 字段设为 `anon_pipe_buf_ops` 表的地址, `offset` 字段设为 0, `len` 字段设为写入的字节数。
9. 增加非空管道缓冲区计数器的值, 该缓冲区计数器存放在 `pipe_inode_inf` 结构的 `nrbufs` 字段。
10. 如果所有请求的字节还没有写完, 则跳至第 4 步。

11. 释放索引节点信号量。
12. 唤醒在管道等待队列上睡眠的所有读进程。
13. 返回写入管道缓冲区的字节数（如果无法写入，则返回错误码）。

FIFO

虽然管道是一种十分简单、灵活、有效的通信机制，但它们有一个主要的缺点，也就是无法打开已经存在的管道。这就使得任意的两个进程不可能共享同一个管道，除非管道由一个共同的祖先进程创建。

这个缺点在很多应用程序中都存在。例如，考虑一个数据库引擎服务器，该服务器连续地轮流询问发出查询请求的客户端进程，并把数据库查询的结果返回客户端进程。服务器和给定客户端之间的每次交互都可以使用一个管道进行处理。但是，当用户显式查询数据库时，通常由shell命令根据需要创建客户端进程；因此，服务器进程和客户端进程就不能方便地共享管道。

为了突破这种限制，Unix系统引入了一种称为命名管道(*named pipe*)或者FIFO[FIFO代表“先进先出(first in, first out)”：最先写入文件的字节总是被最先读出]的特殊文件类型。FIFO在这几个方面都非常类似于管道：在文件系统中不拥有磁盘块，打开的FIFO总是与一个内核缓冲区相关联，这一缓冲区中临时存放两个或多个进程之间交换的数据。

然而，有了磁盘索引节点，使得任何进程都可以访问FIFO，因为FIFO文件名包含在系统的目录树中。因此，在前面那个数据库的例子中，服务器和客户端之间的通信可以很容易地使用FIFO而不是管道。服务器在启动时创建一个FIFO，由客户端程序用来发出自己的请求。每个客户端程序在建立连接之前都另外创建一个FIFO，并在自己对服务器发出的最初请求中包含这个FIFO的名字，服务器程序就可以把查询结果写入这个FIFO。

在Linux 2.6中，FIFO和管道几乎是相同的，并使用相同的pipe_inode_info结构。事实上，FIFO的read和write操作就是由前面“从管道中读取数据”和“向管道中写入数据”这两节描述的pipe_read()和pipe_write()函数实现的。事实上，只有两点主要的差别：

- FIFO索引节点出现在系统目录树上而不是pipefs特殊文件系统中。
- FIFO是一种双向通信管道；也就是说，可能以读/写模式打开一个FIFO。

因此，为了完成我们的描述，我们仅说明如何创建和打开FIFO。

创建并打开 FIFO

进程通过执行 `mknod()` (注 4) 系统调用创建一个 FIFO (参见第十三章的“设备文件”一节), 传递的参数是新 FIFO 的路径名以及 `S_IFIFO` (`0x10000`) 与这个新文件的权限位掩码进行逻辑或的结果。POSIX 引入了一个名为 `mkfifo()` 的系统调用专门用来创建 FIFO。这个系统调用在 Linux 以及 System V Release 4 中是作为调用 `mknod()` 的 C 库函数实现的。

FIFO 一旦被创建, 就可以使用普通的 `open()`、`read()`、`write()` 和 `close()` 系统调用访问 FIFO, 但是 VFS 对 FIFO 的处理方法比较特殊, 因为 FIFO 的索引节点及文件操作都是专用的, 并且不依赖于 FIFO 所在的文件系统。

POSIX 标准定义了 `open()` 系统调用对 FIFO 的操作; 这种操作本质上与所请求的访问类型、I/O 操作的种类 (阻塞或非阻塞) 以及其他正在访问 FIFO 的进程的存在状况有关。

进程可以为读操作、写操作或者读写操作打开一个 FIFO。根据这三种情况, 把与相应的文件对象相关的文件操作设置成特定的方法。

当进程打开一个 FIFO 时, VFS 就执行一些与设备文件所执行的操作相同的操作 (参见第十三章的“设备文件的 VFS 处理”一节)。与打开的 FIFO 相关的索引节点对象是由依赖于文件系统的 `read_inode` 超级块方法进行初始化的。这个方法总要检查磁盘上的索引节点是否表示一个特殊文件, 并在必要时调用 `init_special_inode()` 函数。这个函数又把索引节点对象的 `i_fop` 字段设置为 `def_fifo_fops` 表的地址。随后, 内核把文件对象的文件操作表设置为 `def_fifo_fops`, 并执行它的 `open` 方法, 这个方法由 `fifo_open()` 实现。

`fifo_open()` 函数初始化专用于 FIFO 的数据结构; 具体来说, 它执行下列操作:

1. 获取 `i_sem` 索引节点信号量。
2. 检查索引节点对象的 `i_pipe` 字段; 如果为 `NULL`, 则分配并初始化一个新的 `pipe_inode_info` 结构, 这与本章前面“创建和撤销管道”一节的第 1b~1e 步相同。
3. 根据 `open()` 系统调用的参数中指定的访问模式, 用合适的文件操作表的地址初始化文件对象的 `f_op` 字段 (如表 I9-5 所示)。

注 4: 实际上, 用 `mknod()` 几乎可以创建任何种类的文件, 如块设备文件、字符设备文件、FIFO 甚至是普通文件 (但是该函数不能创建目录和套接字)。

表 19-5: FIFO 的文件操作

访问类型	文件操作	读方法	写方法
只读	read_fifo_fops	pipe_read()	bad_pipe_w()
只写	write_fifo_fops	bad_pipe_r()	pipe_write()
读 / 写	rdwr_fifo_fops	pipe_read()	pipe_write()

4. 如果访问模式或者为只读或者为读 / 写, 则把 1 加到 pipe_inode_info 结构的 readers 字段和 r_counter 字段。此外, 如果访问模式是只读的, 且没有其他的读进程, 则唤醒等待队列上的任何写进程。
5. 如果访问模式或者为只写或者为读 / 写, 则把 1 加到 pipe_inode_info 结构的 writers 字段和 w_counter 字段。此外, 如果访问模式是只写的, 且没有其他的写进程, 则唤醒等待队列上的任何读进程。
6. 如果没有读进程或没有写进程, 则确定函数是应当阻塞还是返回一个错误码而终止 (如表 19-6 所示)。

表 19-6: fifo_open() 函数的行为

访问类型	阻塞	非阻塞
只读, 有写者	成功返回	成功返回
只读, 无写者	等待一个写者	成功返回
只写, 有读者	成功返回	成功返回
只写, 无读者	等待一个读者	返回 -ENXIO
读 / 写	成功返回	成功返回

7. 释放索引节点信号量, 并终止, 返回 0 (成功)。

FIFO 的三个专用文件操作表的主要区别是 read 和 write 方法的实现不同。如果访问类型允许读操作, 那么 read 方法是使用 pipe_read() 函数实现的; 否则, read 方法就是使用 bad_pipe_r() 函数实现的, 该函数只是返回一个错误码。类似地, 如果访问类型允许写操作, 那么 write 方法就是使用 pipe_write() 函数实现的; 否则, write 方法就是使用 bad_pipe_w() 函数实现的, 该函数也只是返回一个错误代码。

System V IPC

IPC 是进程间通信 (Interprocess Communication) 的缩写, 通常指允许用户态进程执行下列操作的一组机制:

- 通过信号量与其他进程进行同步
- 向其他进程发送消息或者从其他进程接收消息
- 和其他进程共享一段内存区

System V IPC 最初是在一个名为“Columbus Unix”的开发版 Unix 变体中引入的，之后在 AT&T 的 System III 中采用。现在在大部分 Unix 系统（包括 Linux）中都可以找到。

IPC 数据结构是在进程请求 IPC 资源（信号量、消息队列或者共享内存区）时动态创建的。每个 IPC 资源都是持久的：除非被进程显式地释放，否则永远驻留在内存中（直到系统关闭）。IPC 资源可以由任一进程使用，包括那些不共享祖先进程所创建的资源的进程。

由于一个进程可能需要同类型的多个 IPC 资源，因此每个新资源都是使用一个 32 位的 IPC 关键字来标识的，这和系统的目录树中的文件路径名类似。每个 IPC 资源都有一个 32 位的 IPC 标识符，这与和打开文件相关的文件描述符有些类似。IPC 标识符由内核分配给 IPC 资源，在系统内部是唯一的，而 IPC 关键字可以由程序员自由地选择。

当两个或者更多的进程要通过一个 IPC 资源进行通信时，这些进程都要引用该资源的 IPC 标识符。

使用 IPC 资源

根据新资源是信号量、消息队列还是共享内存区，分别调用 `semget()`、`msgget()` 或者 `shmget()` 函数创建 IPC 资源。

这三个函数的主要目的都是从 IPC 关键字（作为第一个参数传递）中导出相应的 IPC 标识符，进程以后就可以使用这个标识符对资源进行访问。如果还没有 IPC 资源和 IPC 关键字相关联，就创建一个新的资源。如果一切都顺利，那么函数就返回一个正的 IPC 标识符；否则，就返回一个如表 19-7 所示的错误码。

表 19-7：当请求 IPC 标识符时返回的错误码

错误码	说明
EACCESS	进程没有适当的访问权限
EEXIST	进程试图创建一个和已有的关键字相同的 IPC 资源
EINVAL	在 <code>semget()</code> 、 <code>msgget()</code> 或 <code>shmget()</code> 函数中有非法参数
ENOENT	不存在具有所请求的关键字的 IPC 资源，而且进程没有请求创建这个资源
ENOMEM	没有更多的存储空间供 IPC 另外的资源使用
ENOSPC	已经超过了 IPC 资源最大数目的限制

假设两个独立的进程想共享一个公共的 IPC 资源。这可以使用两种方法来达到：

- 这两个进程统一使用固定的、预定义的 IPC 关键字。这是最简单的情况，对于由很多进程实现的任一复杂的应用程序也工作得很好。然而，另外一个无关的程序也可能使用了相同的 IPC 关键字。在这种情况下，IPC 函数可能被成功地调用，但返回错误资源的 IPC 标识符（注 5）。
- 一个进程通过指定 `IPC_PRIVATE` 作为自己的 IPC 关键字来调用 `semget()`、`msgget()` 或 `shmget()` 函数。一个新的 IPC 资源因此而被分配，这个进程或者可以与应用程序中的另一个进程共享自己的 IPC 标识符（注 6），或者自己创建另一个进程。这种方法确保 IPC 资源不会偶然被其他应用程序使用。

`semget()`、`msgget()` 和 `shmget()` 函数的最后一个参数可以包括三个标志。`IPC_CREAT` 说明如果 IPC 资源不存在，就必须创建它；`IPC_EXCL` 说明如果资源已经存在而且设置了 `IPC_CREAT` 标志，那么函数就必定失败；`IPC_NOWAIT` 说明访问 IPC 资源时进程从不阻塞（典型的情况如取得消息或获取信号量）。

即使进程使用了 `IPC_CREAT` 和 `IPC_EXCL` 标志，也没有办法保证对一个 IPC 资源进行排它访问，因为其他进程也可能用自己的 IPC 标识符引用了这个资源。

为了把不正确地引用错误资源的风险降到最小，内核不会在 IPC 标识符一空闲时就再利用它。相反，分配给资源的 IPC 标识符总是大于给同类型的前一个资源所分配的标识符（唯一的例外发生在 32 位的 IPC 标识符溢出时）。每个 IPC 标识符都是通过结合使用与资源类型相关的位置使用序号（*slot usage sequence number*）、已分配资源的任意位置索引（*slot index*）以及内核中为可分配资源所选定的最大值而计算出来的。如果我们使用 s 来代表位置使用序号， M 来代表可分配资源的最大数目， i 来代表位置索引，此处 $0 \leq i < M$ ，则每个 IPC 资源的 ID 都可以按如下公式来计算：

$$\text{IPC 标识符} = s \times M + i$$

在 Linux 2.6 中， M 的值设为 32768（`IPC_MNI` 宏）。位置使用序号 s 被初始化成 0，每次分配资源时增加 1。当 s 达到预定的阈值时（这取决于 IPC 资源类型），它从 0 重新开始。

注 5：`ftok()` 函数试图从作为参数传递的文件路径名和一个 8 位对象标识符中获得一个新关键字。但是这并不能担保是一个唯一关键字，因为也有可能使用不同路径名和对象标识符的两个不同应用程序会返回同一个 IPC 关键字，不过这种机会很小。

注 6：当然，这就意味着进程之间另一个通信通道的存在并不基于 IPC。

IPC资源的每种类型（信号量、消息队列和共享内存区）都拥有 `ipc_ids` 数据结构，该结构包括的字段如表 19-8 所示。

表 19-8: `ipc_ids` 数据结构的字段

类型	字段	说明
<code>int</code>	<code>in_use</code>	已分配 IPC 资源数
<code>int</code>	<code>max_id</code>	在使用的最大位置索引
<code>unsigned short</code>	<code>seq</code>	下一个分配的位置使用序号
<code>unsigned short</code>	<code>seq_max</code>	最大位置使用序号
<code>struct semaphore</code>	<code>sem</code>	保护 <code>ipc_ids</code> 数据结构的信号量
<code>struct ipc_id_ary</code>	<code>nullentry</code>	如果 IPC 资源无法初始化，则 <code>entries</code> 字段指向伪数据结构（一般不使用）
<code>struct ipc_id_ary *</code>	<code>entries</code>	指向资源的 <code>ipc_id_ary</code> 数据结构的指针

`ipc_id_ary` 数据结构有两个字段：`p` 和 `size`。`p` 字段是一个指向 `kern_ipc_perm` 数据结构的指针数组，每个结构对应一个可分配资源。`size` 字段是这个数组的大小。最初，数组为共享内存区、消息队列与信号量分别存放 1、16 或 128 个指针。当太小时，内核动态地增大数组。但是每种资源都有个上限。系统管理员可以修改 `/proc/sys/kernel/sem`、`/proc/sys/kernel/msgmni` 和 `/proc/sys/kernel/shmmni` 这三个文件以改变这些上限。

每个 `kern_ipc_perm` 数据结构与一个 IPC 资源相关联，并且包含如表 19-9 所示的字段。`uid`、`gid`、`cuid` 和 `cgid` 分别存放资源的创建者的用户标识符和组标识符以及当前资源属主的用户标识符和组标识符。`mode` 位掩码包括六个标志，分别存放资源的属主、组以及其他用户的读、写访问权限。IPC 访问许可权和第一章的“访问权限和文件模式”一节中介绍的文件访问许可权类似，唯一不同的是这里没有执行许可权标志。

表 19-9: `kern_ipc_perm` 结构中的字段

类型	字段	说明
<code>spinlock_t</code>	<code>lock</code>	保护 IPC 资源描述符的自旋锁
<code>int</code>	<code>deleted</code>	如果资源已被释放，则设置该标志
<code>int</code>	<code>key</code>	IPC 关键字
<code>unsigned int</code>	<code>uid</code>	属主用户 ID
<code>unsigned int</code>	<code>gid</code>	属主组 ID
<code>unsigned int</code>	<code>cuid</code>	创建者用户 ID
<code>unsigned int</code>	<code>cgid</code>	创建者组 ID

表 19-9: kern_ipc_perm 结构中的字段 (续)

类型	字段	说明
unsigned short	mode	许可权位掩码
unsigned long	seq	位置使用序号
void*	security	安全结构指针 (用于 SELinux)

kern_ipc_perm 数据结构也包括一个 key 字段和一个 seq 字段, 前者指的是相应资源的 IPC 关键字, 后者存放的是用来计算该资源的 IPC 标识符所使用的位置使用序号。

semctl()、msgctl()和 shmctl()函数都可以用来处理 IPC 资源。IPC_SET 子命令允许进程改变属主的用户标识符和组标识符以及 ipc_perm 数据结构中的许可权位掩码。IPC_STAT 和 IPC_INFO 子命令取得和资源有关的信息。最后, IPC_RMID 子命令释放 IPC 资源。根据 IPC 资源的种类不同, 还可以使用其他专用的子命令 (注 7)。

一旦一个 IPC 资源被创建, 进程就可以通过一些专用函数对这个资源进行操作。进程可以执行 semop()函数获得或释放一个 IPC 信号量。当进程希望发送或接收一个 IPC 消息时, 就分别使用msgsnd()和msgrcv()函数。最后, 进程可以分别使用shmat()和shmdt()函数把一个共享内存区附加到自己的地址空间中或者取消这种附加关系。

ipc()系统调用

所有的 IPC 函数都必须通过适当的 Linux 系统调用实现。实际上, 在 80 × 86 体系结构中, 只有一个名为 ipc()的 IPC 系统调用。当进程调用一个 IPC 函数时, 比如说msgget(), 该函数实际上调用 C 库中的一个封装函数, 该函数又通过传递 msgget()的所有参数加上一个适当的子命令代码 (在本例中是 MSGGET) 来调用 ipc()系统调用。sys_ipc()服务例程检查子命令代码, 并调用内核函数实现所请求的服务。

ipc()“多路复用”系统调用是从早期的 Linux 版本中继承而来的, 早期 Linux 版本把 IPC 代码包含在动态模块中 (参见附录二)。在 system_call 表中为可能未实现的内核部件保留几个系统调用入口并没有什么意义, 因此内核设计者就采用了这种多路复用的方法。

现在, System V IPC 不再作为动态模块被编译, 因此也就没有理由使用单个 IPC 系统调用。事实上, Linux 在 HP 的 Alpha 体系结构和 Intel 的 IA-64 上为每个 IPC 函数都提供了一个系统调用。

注 7: IPC 在设计上的一个缺陷是用户态进程不能原子地创建和初始化一个 IPC 信号量, 因为这两个操作是由两个不同的 IPC 函数执行的。

IPC 信号量

IPC 信号量和在第五章中介绍的内核信号量非常类似：二者都是计数器，用来为多个进程共享的数据结构提供受控访问。

如果受保护的资源是可用的，那么信号量的值就是正数；如果受保护的资源现不可用，那么信号量的值就是 0。要访问资源的进程试图把信号量的值减 1，但是，内核阻塞这个进程，直到在这个信号量上的操作产生一个正值。当进程释放受保护的资源时，就把信号量的值增加 1；在这样处理的过程中，其他所有正在等待这个信号量的进程就都被唤醒。

实际上，IPC 信号量比内核信号量的处理更复杂是由于两个主要的原因：

- 每个 IPC 信号量都是一个或者多个信号量值的集合，而不像内核信号量一样只有一个值。这意味着同一个 IPC 资源可以保护多个独立、共享的数据结构。在资源正在被分配的过程中，必须把每个 IPC 信号量中的信号量的个数指定为 `semget()` 函数的一个参数。从现在开始，我们就把信号量内部的计数器作为原始信号量 (*primitive semaphore*) 来引用。IPC 信号量资源的个数和单个 IPC 资源内原始信号量的个数都有界限，其缺省值前者为 128，后者为 250；不过，系统管理员可以通过 `/proc/sys/kernel/sem` 文件很容易地修改这两个界限。
- System V IPC 信号量提供了一种失效安全机制，这是用于进程不能取消以前对信号量执行的操作就死亡的情况的。当进程选择使用这种机制时，由此引起的操作就是所谓的可取消的 (*undoable*) 信号量操作。当进程死亡时，所有 IPC 信号量都可以恢复成原来的值，就好像从来都没有开始它的操作。这有助于防止出现这种情况：由于正在结束的进程不能手工取消它的信号量操作，其他使用相同信号量的进程无限地停留在阻塞状态。

首先我们简要描绘一下，当进程想访问 IPC 信号量所保护的一个或者多个资源时所执行的典型步骤：

1. 调用 `semget()` 封装函数来获得 IPC 信号量标识符，作为参数指定对共享资源进行保护的 IPC 信号量的 IPC 关键字。如果进程希望创建一个新的 IPC 信号量，则还要指定 `IPC_CREATE` 或者 `IPC_PRIVATE` 标志以及所需要的原始信号量（参见本章前面的“使用 IPC 资源”一节）。
2. 调用 `semop()` 封装函数来测试并递减所有原始信号量所涉及的值。如果所有的测试全部成功，就执行递减操作，结束函数并允许这个进程访问受保护的资源。如果有些信号量正在使用，那么进程通常都会被挂起，直到某个其他进程释放这个资源为

止。函数接收的参数为IPC信号量标识符、用来指定对原始信号量所进行的原子操作的一组整数以及这种操作的个数。作为选项，进程也可以指定SEM_UNDO标志，这个标志通知内核：如果进程没有释放原始信号量就退出，那么撤消那些操作。

3. 当放弃受保护的资源时，就再次调用semop()函数来原子地增加所有有关的原始信号量。
4. 作为选择，调用semctl()封装函数，在参数中指定IPC_RMID命令把这个IPC信号量从系统中删除。

现在我们可以讨论内核是如何实现IPC信号量的。有关的数据结构如图19-1所示。sem_ids变量存放IPC信号量资源类型的ipc_ids数据结构，对应的ipc_id_ary数据结构包含一个指针数组，它指向sem_array数据结构，每个元素对应一个IPC信号量资源。

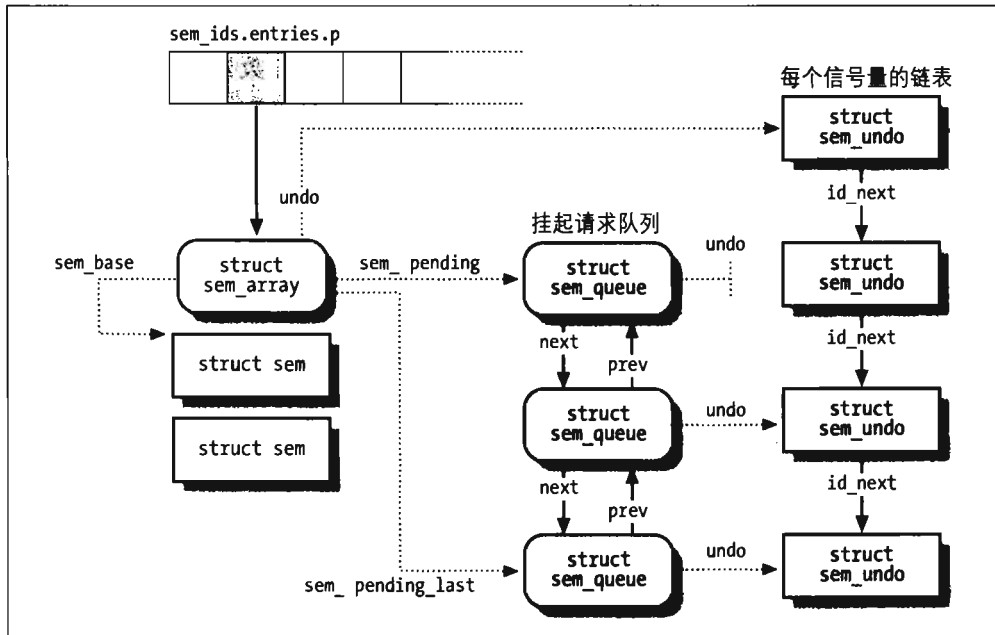


图 19-1：IPC 信号量数据结构

从形式上说，这个数组存放指向 kern_ipc_perm 数据结构的指针，但是每个结构只不过是 sem_array 数据结构的第一个字段。sem_array 数据结构的所有字段如表 19-10 所示。

表 19-10: sem_array 数据结构中的字段

类型	字段	说明
struct kern_ipc_perm	sem_perm	kern_ipc_perm 数据结构
long	sem_otime	最后一次调用 semop() 的时间戳
long	sem_ctime	最后一次修改的时间戳
struct sem *	sem_base	指向第一个 sem 结构的指针
struct sem_queue *	sem_pending	挂起操作
struct sem_queue **	sem_pending_last	最后一次挂起操作
struct sem_undo *	undo	取消请求
unsigned short	sem_nsems	数组中信号量的个数

sem_base 字段指向 sem 数据结构的数组，每个元素对应一个 IPC 原始信号量。sem 数据结构只包括两个字段：

semval

信号量的计数器的值。

sempid

最后一个访问信号量的进程的 PID。进程可以使用 semctl() 封装函数查询该值。

可取消的信号量操作

如果一个进程突然放弃执行，那么它就不能取消已经开始执行的操作（例如，释放自己保留的信号量）；因此通过把这些操作定义成可取消的，进程就可以让内核把信号量返回到一致状态并允许其他进程继续执行。进程可以在 semop() 函数中指定 SEM_UNDO 标志来请求可取消的操作。

为了有助于内核撤销给定进程对给定的 IPC 信号量资源所执行的可撤销操作，有关的信息存放在 sem_undo 数据结构中。这个结构实际上包含信号量的 IPC 标识符及一个整数数组，这个数组表示由进程执行的所有可取消操作对原始信号量值引起的修改。

有一个简单的例子可以说明如何使用这种 sem_undo 元素。考虑一个进程使用具有 4 个原始信号量的一个 IPC 信号量资源，并假设该进程调用 semop() 函数把第一个计数器的值增加 1 并把第二个计数器的值减 2。如果该函数指定了 SEM_UNDO 标志，sem_undo 数据结构中的第一个数组元素中的整数值就被减少 1，而第二个元素就被增加 2，其他两个整数都保持不变。同一进程对这个 IPC 信号量执行的更多的可取消操作将相应地改变存放在 sem_undo 结构中的整数值。当进程退出时，该数组中的任何非零值就表示对相应

原始信号量的一个或者多个错乱操作,内核只简单地给相应的原始信号量计数器增加这个非零值来取消这些操作。换言之,把异常中断的进程所做的修改退回,而其他进程所做的修改仍然能反映信号量的状态。

对于每个进程来说,内核都要记录以可取消操作处理的所有信号量资源,这样如果进程意外退出,就可以回滚这些操作。还有,内核还必须对每个信号量都记录它所有的 `sem_undo` 结构,这样只要进程使用 `semctl()` 来强行给一个原始信号量的计数器赋一个明确的值或者撤消一个 IPC 信号量资源时,内核就可以快速访问这些结构。

正是由于两个链表(我们称之为每个进程的链表和每个信号量的链表),使得内核可以有效地处理这些任务。第一个链表记录给定进程以可取消操作处理的所有信号量。第二个链表记录对以可取消操对给定信号量进行操作的所有进程。更确切地说:

- 每个进程链表包含所有的 `sem_undo` 数据结构,该结构对应于进程执行了可取消操作的 IPC 信号量。进程描述符的 `sysvsem.undo_list` 字段指向一个 `sem_undo_list` 类型的数据结构,而该结构又包含了指针指向该链表的第一个元素。每个 `sem_undo` 数据结构的 `proc_next` 字段指向该链表的下一个元素(在第三章“`clone()`、`fork()` 及 `vfork()` 系统调用”一节我们讲过,因为都共享一个 `sem_undo_list` 描述符,将 `CLONE_SYSVSEM` 标志传给 `clone()` 系统调用而克隆的进程都共享同一个可取消信号量操作链表)。
- 每个信号量链表包含的所有 `sem_undo` 数据结构对应于在该信号量上执行可取消操作的进程。`sem_array` 数据结构的 `undo` 字段指向链表的第一个元素,而每个 `sem_undo` 数据结构的 `id_next` 字段指向链表的下一个元素。

当进程结束时,每个进程的链表才被使用。`exit_sem()` 函数由 `do_exit()` 调用,后者会遍历这个链表,并为进程所涉及的每个 IPC 信号量平息错乱操作产生的影响。与此相对照,当进程调用 `semctl()` 函数强行给一个原始信号量赋一个明确的值时,每个信号量的链表才被使用。内核把指向 IPC 信号量资源的所有 `sem_undo` 数据结构中的数组的相应元素都设置成 0,因为撤消原始信号量的一个可取消操作不再有任何意义。此外,在 IPC 信号量被清除时,每个信号量链表也被使用。通过把 `semid` 字段设置成 -1 而使所有有关的 `sem_undo` 数据结构都变为无效(注 8)。

注 8: 注意仅仅是使这些数据结构无效而已,并没有释放它们,因为从所有进程的每个进程链表中删除这些数据结构的代价太高了。

挂起请求的队列

内核给每个 IPC 信号量都分配了一个挂起请求队列，用来标识正在等待数组中的一个（或多个）信号量的进程。这个队列是一个 `sem_queue` 数据结构的双向链表，其字段如表 19-11 所示。队列中的第一个和最后一个挂起请求分别由 `sem_array` 结构中的 `sem_pending` 和 `sem_pending_last` 字段所指向。这最后一个字段允许把链表作为一个 FIFO 进行简单的处理。新的挂起请求都被追加到链表的末尾，这样就可以稍后得到服务。挂起请求最重要的字段是 `nsops` 和 `sops`，前者存放挂起操作所涉及的原始信号量的个数，后者指向描述每个信号量操作的整型数组。`sleeper` 字段存放发出请求操作的睡眠进程的描述符地址。

表 19-11: `sem_queue` 数据结构中的字段

类型	字段	说明
<code>struct sem_queue *</code>	<code>next</code>	指向下一个队列元素的指针
<code>struct sem_queue **</code>	<code>prev</code>	指向上一个队列元素的指针
<code>struct wait_queue *</code>	<code>sleeper</code>	指向请求信号量操作的睡眠进程
<code>struct sem_undo *</code>	<code>undo</code>	指向 <code>sem_undo</code> 结构的指针
<code>int</code>	<code>pid</code>	进程标识符
<code>int</code>	<code>status</code>	操作的完成状态
<code>struct sem_array *</code>	<code>sma</code>	IPC 信号量的描述符指针
<code>int</code>	<code>id</code>	IPC 信号量的位置索引
<code>struct sembuf *</code>	<code>sops</code>	指向挂起操作的数组的指针
<code>int</code>	<code>nsops</code>	挂起操作的个数
<code>int</code>	<code>alter</code>	标志，表示操作是否修改了信号量数组

图 19-1 显示有三个挂起请求的一个 IPC 信号量。第二个和第三个请求涉及可取消操作，因此 `sem_queue` 数据结构的 `undo` 字段指向相应的 `sem_undo` 结构；第一个挂起请求的 `undo` 字段为 `NULL`，因为相应的操作是不可取消的。

IPC 消息

进程彼此之间可以通过 IPC 消息进行通信。进程产生的每条消息都被发送到一个 IPC 消息队列中，这个消息一直存放在队列中直到另一个进程将其读走为止。

消息是由固定大小的首部和可变长度的正文组成的，可以使用一个整数值（消息类型）标识消息，这就允许进程有选择地从消息队列中获取消息（注9）。只要进程从IPC消息队列中读出一条消息，内核就把这个消息删除；因此，只能有一个进程接收一条给定的消息。

为了发送一条消息，进程要调用 `msgsnd()` 函数，传递给它以下参数：

- 目标消息队列的 IPC 标识符
- 消息正文的大小
- 用户态缓冲区的地址，缓冲区中包含消息类型，之后紧跟消息正文

进程要获得一条消息就要调用 `msgrcv()` 函数，传递给它如下参数：

- IPC 消息队列资源的 IPC 标识符
- 指向用户态缓冲区的指针，消息类型和消息正文应该到被拷贝这个缓冲区
- 缓冲区的大小
- 一个值 t ，指定应该获得什么消息

如果 t 的值为 0，就返回队列中的第一条消息。如果 t 为正数，就返回队列中类型等于 t 的第一条消息。最后，如果 t 为负数，就返回消息类型小于等于 t 绝对值的最小的第一条消息。

为了避免资源耗尽，IPC 消息队列资源在这几个方面是有限制的：IPC 消息队列数（缺省为 16），每个消息的大小（缺省为 8192 字节）及队列中全部信息的大小（缺省为 16384 字节）。不过和前面类似，系统管理员可以分别修改 `/proc/sys/kernel/msgmni`、`/proc/sys/kernel/msgmnb` 和 `/proc/sys/kernel/msgmax` 文件调整这些值。

与 IPC 消息队列有关的数据结构如图 19-2 所示。`msg_ids` 变量存放 IPC 消息队列资源类型的 `ipc_ids` 数据结构，相应的 `ipc_id_ary` 数据结构包含一个指向 `shmid_kernel` 数据结构的指针数组——每个 IPC 消息资源对应一个元素。从形式上看，数组中存放指向 `kern_ipc_perm` 数据结构的指针，但是，每个这样的结构只不过是 `msg_queue` 数据结构的第一个字段。`msg_queue` 数据结构的所有字段如表 19-12 所示。

注9： 我们将看到，消息队列是使用一个链表实现的。因为消息可以按照非“先进先出”的顺序获得，因此“消息队列”这个名字并不恰当。不过，新消息通常都放在链表的末尾。

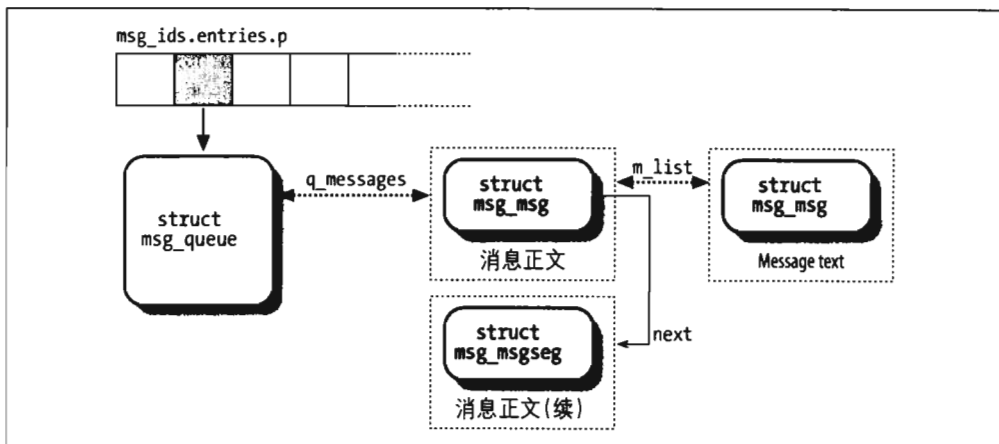


图 19-2: IPC 消息队列数据结构

表 19-12: msg_queue 数据结构

类型	字段	说明
struct kern_ipc_perm	q_perm	kern_ipc_perm 数据结构
long	q_stime	最后调用 msgsnd() 的时间
long	q_rtime	最后调用 msgrcv() 的时间
long	q_ctime	最后修改的时间
unsigned long	q_qcbytes	队列中的字节数
unsigned long	q_qnum	队列中的消息数
unsigned long	q_qbytes	队列中的最大字节数
int	q_lspid	最后调用 msgsnd() 的 PID
int	q_lrpid	最后调用 msgrcv() 的 PID
struct list_head	q_messages	队列中的消息链表
struct list_head	q_receivers	接收消息的进程链表
struct list_head	q_senders	发送消息的进程链表

最重要的字段是 `q_messages`，它表示包含队列中当前所有消息的双向循环链表的首部（也就是第一个哑元素）。

每条消息分开存放在一个或多个动态分配的页中。第一页的起始部分存放消息头，消息头是一个 `msg_msg` 类型的数据结构，它的字段如表 19-13 所示。`m_list` 字段指向队列中前一条和后一条消息的指针。消息的正文正好从 `msg_msg` 描述符之后开始；如果消息（页

的大小减去 `msg_msg` 描述符的大小) 大于 4072 字节, 就继续放在另一页, 它的地址存放在 `msg_msg` 描述符的 `next` 字段中。第二个页框以 `msg_msgseg` 类型的描述符开始, 这个描述符只包含一个 `next` 指针, 该指针存放可选的第三个页, 以此类推。

表 19-13: `msg_msg` 数据结构

类型	字段	说明
<code>struct list_head</code>	<code>m_list</code>	用于消息链表的指针
<code>long</code>	<code>m_type</code>	消息类型
<code>int</code>	<code>m_ts</code>	消息正文的大小
<code>struct msg_msgseg *</code>	<code>next</code>	消息的下一部分
<code>void *</code>	<code>security</code>	安全数据结构指针 (用于 SELinux)

当消息队列满时 (或者达到了最大消息数, 或者达到了队列最大字节数), 则试图让新消息入队的进程可能被阻塞。`msg_queue` 数据结构的 `q_senders` 字段是所有阻塞的发送进程的描述符形成的链表的头。

当消息队列为空时 (或者当进程指定的一条消息类型不在队列中时), 则接收进程也会被阻塞。`msg_queue` 数据结构的 `q_receivers` 字段是 `msg_receiver` 数据结构链表的头, 每个阻塞的接收进程对应其中一个元素。其中的每个结构本质上都包含一个指向进程描述的指针、一个指向消息的 `msg_msg` 结构的指针和所请求的消息类型。

IPC 共享内存

最有用的 IPC 机制是共享内存, 这种机制允许两个或多个进程通过把公共数据结构放入一个共享内存区 (*IPC shared memory region*) 来访问它们。如果进程要访问这种存放在共享内存区的数据结构, 就必须在自己的地址空间中增加一个新内存区 (参见第九章的“线性区”一节), 它将映射与这个共享内存区相关的页框。这样的页框可以很容易地由内核通过请求调页进行处理 (参见第九章的“请求调页”一节)。

与信号量以及消息队列一样, 调用 `shmget()` 函数来获得一个共享内存区的 IPC 标识符, 如果这个共享内存区不存在, 就创建它。

调用 `shmat()` 函数把一个共享内存区“附加 (attach)”到一个进程上。该函数使用 IPC 共享内存资源的标识符作为参数, 并试图把一个共享内存区加入到调用进程的地址空间中。调用进程可以获得这个内存区域的起始线性地址, 但是这个地址通常并不重要, 访问这个共享内存区域的每个进程都可以使用自己地址空间中的不同地址。`shmat()` 函数

不修改进程的页表。我们稍后会介绍在进程试图访问属于新内存区域的页时内核究竟怎样进行处理。

调用 `shmdt()` 函数来“分离”由 IPC 标识符所指定的共享内存区域，也就是说把相应的共享内存区域从进程的地址空间中删除。回想一下，IPC 共享内存资源是持久的；即使现在没有进程在使用它，相应的页也不能被丢弃，但是可以被换出。

与 IPC 资源的其他类型一样，为了避免用户态进程过分使用内存，也有一些限制施加于所允许的 IPC 共享内存区域数（缺省为 4096）、每个共享段的大小（缺省为 32MB）以及所有共享段的最大字节数（缺省为 8GB）。不过，系统管理员照样可以调整这些值，这是通过分别修改 `/proc/sys/kernel/shmmni`、`/proc/sys/kernel/shmmax` 和 `/proc/sys/kernel/shmall` 文件完成的。

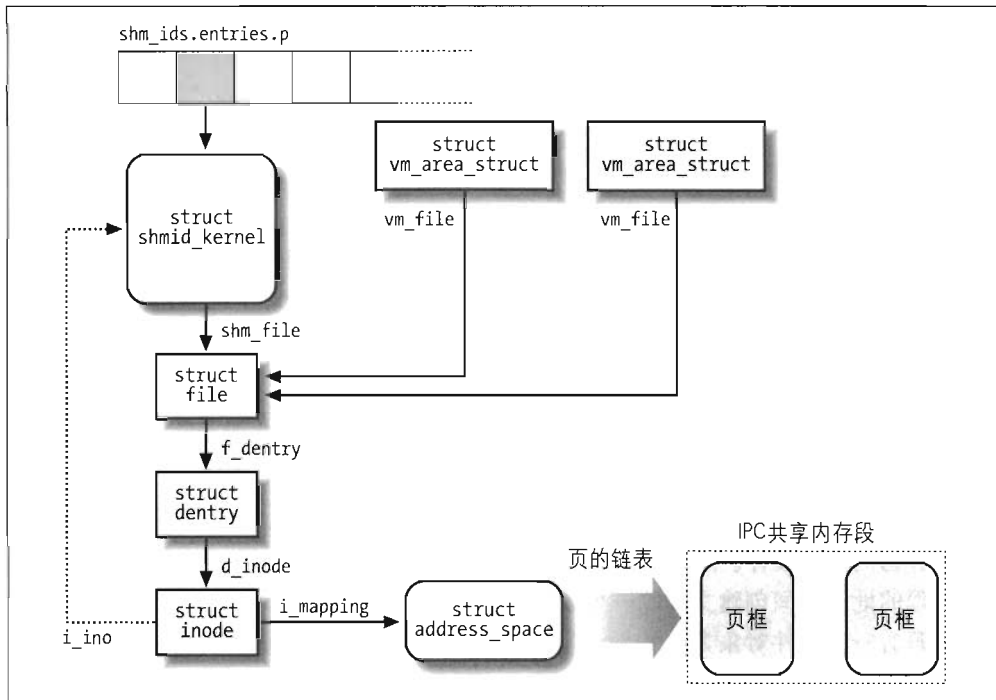


图 19-3: IPC 共享内存数据结构

图 19-3 显示与 IPC 共享内存区相关的数据结构。`shm_ids` 变量存放 IPC 共享内存资源类型 `ipc_ids` 的数据结构，相应的 `ipc_id_ary` 数据结构包含一个指向 `shm_id_kernel` 数据结构的指针数组，每个 IPC 共享内存资源对应一个数组元素。从形式上看，这个数组

存放指向 `kern_ipc_perm` 数据结构指针，但是每个这样的结构只不过是 `msg_queue` 数据结构的第一个字段。 `shmid_kernel` 数据结构的所有字段如表 19-14 所示。

表 19-14：在 `shmid_kernel` 数据结构中的字段

类型	字段	说明
<code>struct kern_ipc_perm</code>	<code>shm_perm</code>	<code>kern_ipc_perm</code> 数据结构
<code>struct file *</code>	<code>shm_file</code>	共享段的特殊文件
<code>int</code>	<code>id</code>	共享段的位置索引
<code>unsigned long</code>	<code>shm_nattch</code>	当前附加的内存区数
<code>unsigned long</code>	<code>shm_segsz</code>	内存区字节数
<code>long</code>	<code>shm_atime</code>	最后访问时间
<code>long</code>	<code>shm_dtime</code>	最后分离时间
<code>long</code>	<code>shm_ctime</code>	最后修改时间
<code>int</code>	<code>shm_cprid</code>	创建者的 PID
<code>int</code>	<code>shm_lprid</code>	最后访问进程的 PID
<code>struct user_struct *</code>	<code>mlock_user</code>	锁定在共享内存 RAM 中的用户的 <code>user_struct</code> 描述符的指针（参见第三章“ <code>clone()</code> 、 <code>fork()</code> 及 <code>vfork()</code> 系统调用”一节）

最重要的字段是 `shm_file`，该字段存放文件对象的地址。这反映 Linux 2.6 中 IPC 共享内存与 VFS 的紧密结合。具体来说，每个 IPC 共享内存区与属于 `shm` 特殊文件系统的—个普通文件相关联（参见第十二章“特殊文件系统”一节）。

因为 `shm` 文件系统在系统目录树中没有安装点，因此，用户不能通过普通的 VFS 系统调用打开并访问它的文件。但是，只要进程“附加”一个内存段，内核就调用 `do_mmap()`，并在进程的地址空间创建文件的一个新的共享内存映射。因此，属于 `shm` 特殊文件系统的文件只有—个文件对象方法 `mmap`，该方法是由 `shm_mmap()` 函数实现的。

如图 19-3 所示，与 IPC 共享内存区对应的内存区是用 `vm_area_struct` 对象描述的（参见第十六章“内存映射”一节）；它的 `vm_file` 字段指回特殊文件的文件对象，而特殊文件又依次引用目录项对象和索引节点对象。存放在索引节点 `i_ino` 字段的索引节点号实际上是 IPC 共享内存区的位置索引，因此，索引节点对象间接引用 `shmid_kernel` 描述符。

同样，对于任何共享内存映射，通过 `address_space` 对象把页框包含在页高速缓存中，而 `address_space` 对象包含在索引节点中而且被索引节点的 `i_mapping` 字段引用（你也

可以参看图 16-2)。万一页框属于 IPC 共享内存区，`address_space`对象的方法就存放在全局变量 `shmem_aops` 中。

换出 IPC 共享内存区的页

内核在把包含在共享内存区的页换出时一定要谨慎，并且交换高速缓存的作用是至关重要的（这个主题已经在第十七章“交换高速缓存”一节讨论过）。

因为 IPC 共享内存区映射的是在磁盘上没有映像的特殊索引节点，因此其页是可交换的（而不是可同步的，参见第十七章的表 17-1）。因此为了回收 IPC 共享内存区的页，内核必须把它写入交换区。因为 IPC 共享内存区是持久的——也就是说即使内存段不附加到进程，也必须保留这些页。因此即使这些页没有进程在使用，内核也不能简单地删除它们。

让我们看看 PFRA 是如何回收 IPC 共享内存区页框的。一直到 `shrink_list()` 函数处理页之前，都与第十七章“内存紧缺回收”一节所描述的一样。因为这个函数并不为 IPC 共享内存区域作任何检查，因此它会调用 `try_to_unmap()` 函数从用户态地址空间删除对页框的每个引用。正如第十七章“反向映射”一节描述的一样，相应的页表项就被删除。

然后，`shrink_list()` 函数检查页的 `PG_dirty` 标志，调用 `pageout()`（当 IPC 共享内存区域的页框在分配时总是被标记为脏，因此 `pageout()` 总是被调用）。而 `pageout()` 函数又调用所映射文件的 `address_space` 对象的 `writepage` 方法。

`shmem_writepage()` 函数实现了 IPC 共享内存区页的 `writepage` 方法。它实际上给交换区域分配一个新页槽（page slot），然后将它从页高速缓存移到交换高速缓存（实际上就是改变页所有者的 `address_space` 对象）。该函数还在 `shmem_inode_info` 结构中存放换出页标识符，这个结构包含了 IPC 共享内存区的索引节点对象，它再次设置页的 `PG_dirty` 标志。如第十七章的表 17-5 所示，`shrink_list()` 函数检查 `PG_dirty` 标志，并通过把页留在非活动链表而中断回收过程。

迟早，PFRA 还会处理该页框。`shrink_list()` 又一次调用 `pageout()` 尝试将页刷新到磁盘。但这一次，页已在交换高速缓存内，因而它的所有者是交换子系统的 `address_space` 对象，即 `swapper_space`。相应的 `writepage` 方法 `swap_writepage()` 开始有效地向交换区进行写入操作（参见第十七章“换出页”一节）。一旦 `pageout()` 结束，`shrink_list()` 确认该页已干净，于是从交换高速缓存删除页并释放给伙伴系统。

IPC 共享内存区的请求调页

通过 `shmat()` 加入进程的页都是哑元页（dummy page），该函数把一个新内存区加入一

个进程的地址空间中，但是它不修改该进程的页表。此外，我们已经看到，IPC 共享内存区的页可以被换出。因此，可以通过请求调页机制来处理这些页。

我们知道，当进程试图访问IPC 共享内存区的一个单元，而其基本的页框还没有分配时则发生缺页异常。相应的异常处理程序确定引起缺页的地址是在进程的地址空间内，且相应的页表项为空；因此，它就调用 `do_no_page()` 函数（参见第九章中的“请求调页”一节）。这个函数又检查是否为此内存区定义了 `nopage` 方法。然后调用这个方法，并将页表项设置成所返回的地址（参见第十六章“内存映射的请求调页”一节）。

IPC 共享内存所使用的内存区通常都定义了 `nopage` 方法。这是通过 `shmem_nopage()` 函数实现的，该函数执行以下操作：

1. 遍历 VFS 对象的指针链表，并导出 IPC 共享内存资源的索引节点对象的地址（参见图 19-3）。
2. 从内存区域描述符的 `vm_start` 字段和请求的地址计算共享段内的逻辑页号。
3. 检查页是否已经在交换高速缓存中，如果是，则结束并返回该描述符的地址。
4. 检查页是否在交换高速缓存内且是否最新，如果是，则结束并返回该描述符的地址。
5. 检查内嵌在索引节点对象的 `shmem_inode_info` 是否存放着逻辑页号对应的换出页标识符。如果是，就调用 `read_swap_cache_async()` 执行换入操作（参见第十七章“换入页”一节），并一直等到数据传送完成，然后结束并返回页描述符的地址。
6. 否则，页不在交换区中，因此就从伙伴系统分配一个新页框，把它插入页高速缓存，并返回它的地址。

`do_no_page()` 函数对引起缺页的地址在进程的页表中所对应的表项进行设置，以使该函数指向 `nopage` 方法所返回的页框。

POSIX 消息队列

POSIX 标准（IEEE Std 1003.1-2001）基于消息队列定义了一个 IPC 机制，就是大家知道的 POSIX 消息队列。它很像本章前面“IPC 消息”一节介绍的 System V IPC 消息队列。但是 POSIX 消息队列比老的队列具有许多优点：

- 更简单的基于文件的应用接口
- 完全支持消息优先级（优先级最终决定队列中消息的位置）

- 完全支持消息到达的异步通知，这通过信号或是线程创建实现
- 用于阻塞发送与接收操作的超时机制

POSIX 消息队列通过一套库函数来实现，参见表 19-15。

表 19-15: POSIX 消息队列的库函数

函数名	说明
<code>mq_open()</code>	打开（或创建）POSIX 消息队列
<code>mq_close()</code>	关闭 POSIX 消息队列（并不删除）
<code>mq_unlink()</code>	删除 POSIX 消息队列
<code>mq_send()</code>	给 POSIX 消息队列发送一个消息
<code>mq_timedsend()</code>	在操作时限内给 POSIX 消息队列发送一个消息
<code>mq_receive()</code>	从 POSIX 消息队列接收一个消息
<code>mq_timedreceive()</code>	在操作时限内从 POSIX 消息队列接收一个消息
<code>mq_notify()</code>	在空 POSIX 消息队列中，为消息到达建立异步通知机制
<code>mq_getattr()</code>	获得 POSIX 消息队列的属性（实际上就是发送和接收操作应当是阻塞还是非阻塞）
<code>mq_setattr()</code>	设置 POSIX 消息队列的属性（实际上就是发送和接收操作应当是阻塞还是非阻塞）

我们来看看应用如何典型地使用这些函数。首先，应用调用 `mq_open()` 库函数打开一个 POSIX 消息队列。函数的第一个参数是一个指定队列名字的字符串，这与文件名类似，而且必须以“/”开始。该库函数接收一个 `open()` 系统调用的标志子集：`O_RDONLY`、`O_WRONLY`、`O_RDWR`、`O_CREAT`、`O_EXCL` 和 `O_NONBLOCK`（用于非阻塞发送与接收）。注意应用可以通过指定一个 `O_CREAT` 标志来创建一个新的 POSIX 消息队列。`mq_open()` 函数返回一个队列描述符，与 `open()` 系统调用返回的文件描述符非常类似。

一旦 POSIX 消息队列打开，应用可以通过库函数 `mq_send()` 和 `mq_receive()` 来发送与接收消息，并传给它们 `mq_open()` 返回的队列描述符作为参数。应用也可以通过 `mq_timedsend()` 和 `mq_timedreceive()` 指定应用程序等待发送与接收操作完成所需的最长时间。

应用除了在 `mq_receive()` 上阻塞，或者如果 `O_NONBLOCK` 标志置位则继续在消息队列上轮询外，还可以通过执行 `mq_notify()` 库函数建立异步通知机制。实际上当一个消息插入空队列时，应用可以要求：要么给指定进程发出信号，要么创建一个新线程。

最后，当应用使用完消息队列，它调用 `mq_close()` 库函数，传给它队列描述符。注意这个函数并不删除队列，这与 `close()` 系统调用不会删除文件一样。要删除队列，应用需要调用 `mq_unlink()` 函数。

在 Linux 2.6 中，POSIX 消息队列的实现是简单的。已经引入了一个叫做 *mqueue* 的特殊文件系统（参见第十二章“特殊文件系统”一节），每个现存队列在其中都有一个相应的索引节点。内核提供了几个系统调用：`mq_open()`、`mq_unlink()`、`mq_timedsend()`、`mq_timedreceive()`、`mq_notify()` 和 `mq_getsetattr()`。这些系统调用大略对应前面表 19-15 中的库函数。这些系统调用透明地对 *mqueue* 文件系统的文件进行操作，而大部分工作交由 VFS 层处理。例如，注意到内核不提供 `mq_close()` 函数，而事实上返回给应用的队列描述符实际上是一个文件描述符，因此 `mq_close()` 的工作由 `close()` 系统调用来做就可以了。

mqueue 特殊文件系统不能安装在系统目录树中。但是如果安装了，用户可以通过使用文件系统根目录中的文件来创建 POSIX 消息队列，也可以读入相应文件来得到队列的有关信息。最后，应用可以使用 `select()` 和 `poll()` 获得队列状态变化的通知。

每个队列有一个 `mqueue_inode_info` 描述符，它包含有 `inode` 对象，该对象与 *mqueue* 特殊文件系统的文件相对应。当 POSIX 消息队列系统调用接收一个队列描述符作为参数时，它就调用 VFS 的 `fget()` 函数计算出对应文件对象的地址。然后，系统调用得到 *mqueue* 文件系统中文件的索引节点对象。最后，就可以得到该索引节点对象所对应的 `mqueue_inode_info` 描述符地址。

队列中挂起的消息被收集到 `mqueue_inode_info` 描述符中的一个单向链表。每个消息由一个 `msg_msg` 类型的描述符来表示，这与 System V IPC 中使用的消息描述符是完全一样的（参见本章前面“IPC 消息”一节）。



程序的执行

第三章所描述的“进程”概念在 Unix 中是用来表示正在运行的一组程序竞争系统资源的行为。本章作为最后的一章，将集中讨论程序和进程之间的关系。我们会专门描述内核如何通过程序文件的内容建立进程的执行上下文。尽管把一组指令装入内存并让 CPU 执行看起来并不是什么大问题，但内核还必须灵活处理以下几方面的问题：

不同的可执行文件格式

Linux 的出色表现之一就是能执行为其他操作系统编译的二进制代码。具体地说，Linux 可以在 64 位版本的机器上执行 32 位可执行代码。例如，在 Pentium 上创建的可执行代码可以在 64 位的 AMD Opteron CPU 平台上运行。

共享库

很多可执行文件并不包含执行程序所需的所有代码，而是期望内核在运行时从共享库中加载函数。

执行上下文的其他信息

这包括程序员熟悉的命令行参数与环境变量。

程序是以可执行文件 (*executable file*) 的形式存放在磁盘上的，可执行文件既包括被执行函数的目标代码，也包括这些函数所使用的数据。程序中的很多函数是所有程序员都可使用的服务例程，它们的目标代码包含在所谓“库”的特殊文件中。实际上，一个库函数的代码或被静态地拷贝到可执行文件中（静态库），或在运行时被连接到进程（共享库，因为它们的代码由很多独立的进程所共享）。

当装入并运行一个程序时，用户可以提供影响程序执行方式的两种信息：命令行参数和

环境变量。用户在 shell 提示符下紧跟文件名输入的就是命令行参数。环境变量（例如 HOME 和 PATH）是从 shell 继承来的，但用户在装入并运行程序前可以修改任何环境变量。

我们在“可执行文件”一节将解释一个程序的执行上下文到底是什么。在“可执行格式”一节我们会提及一些 Linux 所支持的可执行格式，并说明 Linux 如何改变它的“个性”以执行其他操作系统所编译的程序。最后，在“exec 函数”一节会描述执行一个新程序的进程所需的系统调用。

可执行文件

在第一章中我们把进程定义为“执行上下文”。这就意味着进行特定的计算需要收集必要的信息，包括所访问的页，打开的文件，硬件寄存器的内容等等。可执行文件是一个普通文件，它描述了如何初始化一个新的执行上下文，也就是如何开始一个新的计算。

假定一位用户想在当前目录下显示文件，他知道在 shell 提示符下只要简单地敲入外部命令 `/bin/ls`（注 1）就可得到这个结果。命令 shell 创建一个新进程，新进程又调用系统调用 `execve()`（参看本章后面的“exec 函数”一节），其中传递的一个参数就是 `ls` 可执行文件的全路径名，在本例中即 `/bin/ls`。`sys_execve()` 服务例程找到相应的文件，检查可执行格式，并根据存放在其中的信息修改当前进程的执行上下文。因此，当这个系统调用终止时，新进程开始执行存放在可执行文件中的代码，也就是执行目录显示。

当进程开始执行一个新程序时，它的执行上下文发生很大的变化，这是因为在进程的前一个计算执行期间所获得的大部分资源会被抛弃。在前面的例子中，当进程开始执行 `/bin/ls` 时，它用 `execve()` 系统调用传递来的新参数代替 shell 的参数，并获得一个新的 shell 环境（参见后面的“命令行参数和 shell 环境”一节）；从父进程继承的所有页（并通过写时复制机制实现共享）被释放，以便在一个新的用户态地址空间开始执行新的计算；甚至进程的特权都可能改变（参看后面的“进程的信任状和权能”一节）。然而，进程的 PID 不改变，并且新的计算从前一个计算继承所有打开的文件描述符，当然这些文件描述符是在执行 `execve()` 系统调用时还没有自动关闭的描述符（注 2）。

注 1: 在 Linux 中，可执行文件的路径不是固定的，这取决于所使用的发布版本。对于所有的 Unix 系统，已经提议了几个标准的命名模式，如 *FHS (Filesystem Hierarchy standard)*。

注 2: 默认情况下，在发出 `execve()` 系统调用后由进程已经打开的文件仍然是打开的。但是，如果进程把 `files_struct` 结构的 `close_on_exec` 字段的相应位置置位，则文件自动关闭（参见第十二章的表 12-7）；这是通过 `fcntl()` 系统完成的。

进程的信任状和权能

从传统上看，Unix 系统与每个进程的一些信任状 (*credential*) 相关，信任状把进程与一个特定的用户或用户组捆绑在一起。信任状在多用户系统上尤为重要，因为信任状可以决定每个进程能做什么，不能做什么，这样既保证了每个用户的个人数据的完整性，也保证了系统整体上的稳定性。

信任状的使用既需要在进程的数据结构方面给予支持，也需要在被保护的资源方面给予支持。文件就是一种显而易见的资源。因此，在 Ext2 文件系统中，每个文件都属于一个特定的用户，并被捆绑于某个用户组。文件的拥有者可以决定对某个文件允许哪些操作，以在文件的拥有者、文件的用户组及其他所有用户之间做出区别。当某个进程试图访问一个文件时，VFS 总是根据文件的拥有者和进程的信任状所建立的许可权检查访问的合法性。

进程的信任状存放在进程描述符的几个字段中，如表 20-1 所示。这些字段包括系统中用户和用户组的标识符，与之可以相比较的通常是存放在所访问文件索引节点中的标识符。

表 20-1：传统的进程信任状

名字	说明
uid, gid	用户和组的实际标识符
euid, egid	用户和组的有效标识符
fsuid, fsgid	文件访问的用户和组的有效标识符
groups	补充的组标识符
suid, sgid	用户和组保存的标识符

值为 0 的 UID 指定给 root 超级用户，而值为 0 的用户 GID 指定给 root 超级组。只要有有关进程的信任状存放了一个零值，则内核将放弃权限检查，始终允许这个进程做任何事情，如涉及系统管理或硬件处理的那些操作，而这些操作对于非特权进程是不允许的。

当一个进程被创建时，总是继承父进程的信任状。不过，这些信任状以后可以被修改，这发生在当进程开始执行一个新程序时，或者当进程发出合适的系统调用时。通常情况下，进程的 uid、euid、fsuid 及 suid 字段具有相同的值。然而，当进程执行 setuid 程序时，即可执行文件的 setuid 标志被设置时，euid 和 fsuid 字段被置为这个文件拥有者的标识符。几乎所有的检查都涉及这两个字段中的一个：fsuid 用于与文件相关的操作，而 euid 用于其他所有的操作。这也同样适用于组标识符的 gid、egid、fsgid 及 sgid 字段。

我们用一个例子来说明如何使用 `fsuid` 字段，考虑一下当用户想改变她的口令时的典型情况。所有的口令都存放在一个公共文件中，但用户不能直接编辑这样的文件，因为它是受保护的。因此，用户调用一个名为 `/usr/bin/passwd` 的系统程序，它可以设置 `setuid` 标志，而且它的拥有者是超级用户。当 shell 创建的进程执行这样一个程序时，进程的 `uid` 和 `fsuid` 字段被置为 0，即超级用户的 PID。现在，这个进程可以访问这个文件，因为当内核执行访问控制表时在 `fsuid` 字段发现了值 0。当然，`usr/bin/passwd` 程序除了让用户改变自己的口令外，并不允许做其他任何事情。

从 Unix 的历史发展可以得出一个教训，即 `setuid` 程序是相当危险的：恶意用户可以以这样的方式触发代码中的一些编程错误 (bug)，从而强迫 `setuid` 程序执行程序的最初设计者从未安排的操作。这可能常常危及整个系统的安全。为了减少这样的风险，Linux 与所有现代 Unix 操作系统一样，让进程只有在必要时才获得 `setuid` 特权，并在不需要时取消它们。可以证明，当以几个保护级别实现用户应用程序时，这种特点是很有用的。进程描述符包含一个 `suid` 字段，在 `setuid` 程序执行以后在该字段中正好存放有效标识符 (`uid` 和 `fsuid`) 的值。进程可以通过 `setuid()`、`setresuid()`、`setfsuid()` 和 `setreuid()` 系统调用改变有效标识符 (注 3)。

表 20-2 显示了这些系统调用是怎样影响进程的信任状的。请注意，如果调用进程还没有超级用户特权，即它的 `uid` 字段不为 0，那么，只能用这些系统调用来设置在这个进程的信任状字段已有的值。例如，一个普通用户进程可以通过调用系统调用 `setfsuid()` 强迫它的 `fsuid` 值为 500，但这只有在其他信任状字段中有一个字段已经有相同的值 500 时才行。

表 20-2：设置进程信任状的系统调用语义

字段	setuid (e)		setresuid (u,e,s)	setreuid (u,e)	setfsuid (f)
	uid=0	uid ≠ 0			
uid	设置为 e	不改变	设置为 u	设置为 u	不改变
uid	设置为 e	设置为 e	设置为 e	设置为 e	不改变
fsuid	设置为 e	设置为 e	设置为 e	设置为 e	设置为 f
suid	设置为 e	不改变	设置为 s	设置为 e	不改变

为了理解四个用户 ID 字段之间的复杂关系，让我们考虑一下 `setuid()` 系统调用的效果。

注 3：通过发出相应的 `setgid()`、`setresgid()`、`setfsgid()` 和 `setregid()` 系统调用，可以改变组的有效信任状。

这些操作是不同的，这取决于调用者进程的 `uid` 字段是否被置为 0（即进程有超级用户特权）或被置为一个正常的 UID。

如果 `uid` 字段为 0，这个系统调用就把调用进程的所有信任状字段(`uid`、`uid`、`fsuid` 及 `suid`)置为参数 `e` 的值。超级用户进程因此就可以删除自己的特权而变为由普通用户拥有的一个进程。例如，在用户登录时，系统以超级用户特权创建一个新进程，但这个进程通过调用 `setuid()` 系统调用删除自己的特权，然后开始执行用户的 `login shell` 程序。

如果 `uid` 字段不为 0，那么这个系统调用只修改存放在 `uid` 和 `fsuid` 中的值，让其他两个字段保持不变。当运行 `setuid` 程序来提高和降低进程有效权限时（这些权限存放在 `uid` 和 `fsuid` 字段），该系统调用的这种功能是非常有用的。

进程的权能

POSIX.1e 草案（现已撤销）用“权能（capability）”一词引入进程信任状的另一种模型。Linux 内核支持 POSIX 权能，但是大部分 Linux 的发行版本不用它。

一种权能仅仅是一个标志，它表明是否允许进程执行一个特定的操作或一组特定的操作。这个模型不同于传统的“超级用户 VS 普通用户”模型，在后一种模型中，一个进程要么能做任何事情，要么什么也不能做，这取决于它的有效 UID。如表 20-3 所示，在 Linux 内核中已包含了很多权能。

表 20-3: Linux 的权能

名字	说明
<code>CAP_AUDIT_WRITE</code>	通过在 <code>netlink</code> 套接字进行写入而产生审计消息
<code>CAP_AUDIT_CONTROL</code>	通过 <code>netlink</code> 套接字控制内核审计操作
<code>CAP_CHOWN</code>	忽略对文件和组的拥有者进行改变的限制
<code>CAP_DAC_OVERRIDE</code>	忽略文件的访问许可权
<code>CAP_DAC_READ_SEARCH</code>	忽略文件 / 目录读和搜索的许可权
<code>CAP_FOWNER</code>	一般是忽略对文件拥有者的权限检查
<code>CAP_FSETID</code>	忽略对文件 <code>setid</code> 和 <code>setgid</code> 标志设置的限制
<code>CAP_KILL</code>	产生信号时绕过权限检查
<code>CAP_LINUX_IMMUTABLE</code>	允许修改仅追加和不可变的 Ext2/Ext3 文件
<code>CAP_IPC_LOCK</code>	允许页加锁和共享内存段加锁
<code>CAP_IPC_OWNER</code>	跳过 IPC 拥有者检查
<code>CAP_LEASE</code>	允许对文件进行租借（参看第十二章“Linux 文件加锁”一节）

表 20-3: Linux 的权能 (续)

名字	说明
CAP_MKNOD	允许有特权的 <code>mknod()</code> 操作
CAP_NET_ADMIN	允许一般的联网管理
CAP_NET_BIND_SERVICE	允许绑定到低于 1024 的 TCP/UDP 套接字
CAP_NET_BROADCAST	允许广播与组播
CAP_NET_RAW	允许使用 RAW 和 PACKET 套接字
CAP_SETGID	忽略对组进程信任状操作的限制
CAP_SETPCAP	允许对其他进程进行权能操作
CAP_SETUID	忽略对用户进程信任状操作的限制
CAP_SYS_ADMIN	允许一般的系统管理
CAP_SYS_BOOT	允许使用 <code>reboot()</code>
CAP_SYS_CHROOT	允许使用 <code>chroot()</code>
CAP_SYS_MODULE	允许内核模块的插入和删除
CAP_SYS_NICE	跳过 <code>nice()</code> 和 <code>setpriority()</code> 系统调用的权限检查, 并允许创建实时进程
CAP_SYS_PACCT	允许配置进程的记账
CAP_SYS_PTRACE	允许对任何进程使用 <code>ptrace()</code>
CAP_SYS_RAWIO	允许通过 <code>ioperm()</code> 和 <code>iopl()</code> 访问 I/O 端口
CAP_SYS_RESOURCE	允许增加资源限制
CAP_SYS_TIME	允许系统时钟和实时时钟的操作
CAP_SYS_TTY_CONFIG	允许配置终端并执行 <code>vhangup()</code> 系统调用

权能的主要优点是, 任何时候每个进程只需要有限种权能。因此, 即使有恶意的用户发现一种利用有潜在错误的程序的方法, 他也只能非法地执行有限个操作类型。

例如, 假定一个有潜在错误的程序只有 CAP_SYS_TIME 权能。在这种情况下, 利用其错误的恶意用户只能在非法地改变实时时钟和系统时钟方面获得成功。她并不能执行任何其他特权的操作。

不管是 VFS 还是 Ext2 文件系统目前都不支持权能模型, 所以, 当进程执行一个可执行文件时, 无法把这个文件与本该强加的一组权能联系起来。然而, 进程可以分别用 `capget()` 和 `capset()` 系统调用显式地获得和降低它的权能。例如, 完全可以通过修改 `login` 程序只保留其权能的一个子集而删除其他权能。

事实上，Linux 内核已经考虑权能。例如，让我们考虑一下 `nice()` 系统调用，它允许用户改变进程的静态优先级。在传统的模型中，只有超级用户才能提升一个优先级，内核因此应该检查调用进程描述符的 `uid` 字段是否为 0。然而，Linux 内核定义了一个名为 `CAP_SYS_NICE` 的权能，就正好对应着这种操作。内核通过调用 `capable()` 函数并把 `CAP_SYS_NICE` 值传给这个函数来检查这个标志的值。

正是由于一些“兼容性小巧程序”已被加入到内核代码中，这种方法才起作用。每当一个进程把 `uid` 和 `fsuid` 字段设置为 0 时（或者通过调用表 20-2 中的一个系统调用，或者通过执行超级用户所拥有的 `setuid` 程序），内核就设置进程的所有权能，以便使所有的检查成功。类似地，当进程把 `uid` 和 `fsuid` 字段重新置为进程拥有者的实际 UID 时，内核检查进程描述符中的 `keep_capabilities` 标志，并在该标志设置时删除进程的所有权能。进程可以调用 Linux 专有的 `prctl()` 系统调用来设置和重新设置 `keep_capabilities` 标志。

Linux 安全模块框架

在 Linux 2.6 中，权能是与 Linux 安全模块 (LSM) 框架紧密结合在一起的。简单地说，LSM 框架允许开发人员定义几种可以选择的内核安全模型。

每个安全模型是由一组安全钩 (*security hook*) 实现的。安全钩是由内核调用的一个函数，用于执行与安全有关的重要操作。钩函数决定一个操作是否可以执行。

钩函数存放在 `security_operations` 类型的表中。当前使用的安全模型钩表地址存放在 `security_ops` 变量中。内核默认使用 `dummy_security_ops` 表实现最小安全模型。表中的每个钩函数实际上去检查相应的权能(如果有)是否允许，否则无条件返回 0 (允许操作)。

例如，`stime()` 和 `settimeofday()` 函数的服务例程在改变系统日期时间之前调用 `settime` 安全钩。`dummy_security_ops` 表指向相应的函数，而该函数约束自己去检查当前进程是否有 `CAP_SYS_TIME` 的权能，并相应地返回 0 或者 `-EPERM`。

Linux 内核更复杂的安全模型已经开发出来。一个广为人知的例子是由美国国家安全局开发的 `seclty_Enhanced Linux (SELinux)`。

命令行参数和 shell 环境

当用户键入一个命令时，为满足这个请求而装入的程序可以从 shell 接收一些命令行参数 (*command-line argument*)。例如，当用户键入命令：

```
$ ls -l /usr/bin
```

以获得在 `/usr/bin` 目录下的全部文件列表时，shell 进程创建一个新进程执行这个命令。这个新进程装入 `/bin/ls` 可执行文件。在这样做的过程中，从 shell 继承的大多数执行上下文被丢弃，但三个单独的参数 `ls`、`-l` 和 `/usr/bin` 依然保持。一般情况下，新进程可以接收任意多个参数。

传递命令行参数的约定依赖于所用的高级语言。在 C 语言中，程序的 `main()` 函数把传递给程序的参数个数和指向字符串指针数组的地址作为参数。下列原型形式化地表示了这种标准格式：

```
int main(int argc, char *argv[])
```

再回到前面的例子，当 `/bin/ls` 程序被调用时，`argc` 的值为 3，`argv[0]` 指向 `ls` 字符串，`argv[1]` 指向 `-l` 字符串，而 `argv[2]` 指向 `/usr/bin` 字符串。`argv` 数组的末尾处总以空指针来标记，因此，`argv[3]` 为 `NUL`。

在 C 语言中，传递给 `main()` 函数的第三个可选参数是包含环境变量的参数。环境变量用来定制进程的执行上下文，由此为用户或其他进程提供通用的信息，或者允许进程在执行 `execve()` 系统调用的过程中保持一些信息。

为了使用环境变量，`main()` 可以声明如下：

```
int main(int argc, char *argv[], char *envp[])
```

`envp` 参数指向环境串的指针数组，形式如下：

```
VAR_NAME=something
```

这里，`VAR_NAME` 表示一个环境变量的名字，而“=”后面的子串表示赋给变量的实际值。`envp` 数组的结尾用一个空指针标记，就像 `argv` 数组。`envp` 数组的地址存放在 C 库的 `environ` 全局变量中。

命令行参数和环境串都存放在用户态堆栈中，正好位于返回地址之前（参见第十章的“参数传递”一节）。图 20-1 显示了用户态堆栈的底部单元。注意，环境变量位于栈底附近正好在一个 0 长整数之后。

库

每个高级语言的源码文件都是经过几个步骤才转化为目标文件的，目标文件中包含的是汇编语言指令的机器代码，它们和相应的高级语言指令对应。目标文件并不能被执行，因为它不包含源代码文件所用的全局外部符号名的线性地址（例如库函数或同一程序中的其他源代码文件）。这些地址的分配或解析是由链接程序完成的，链接程序把程序所

有的目标文件收集起来并构造可执行文件。链接程序还分析程序所用的库函数，并以本章后面所描述的方式把它们粘合成可执行文件。

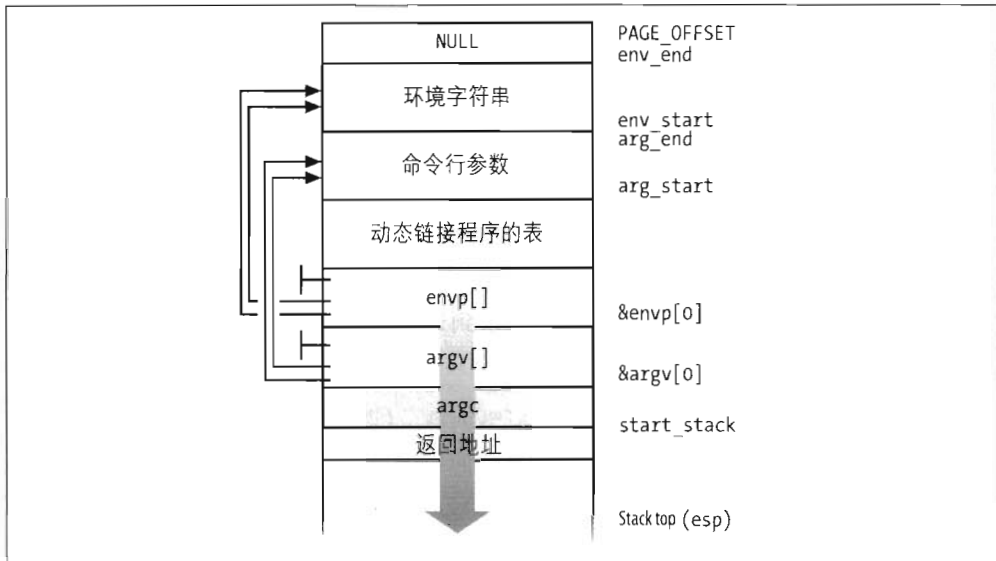


图 20-1：用户态堆栈的底部单元

大多数程序，甚至是最小的程序都会利用 C 库。例如，请看下面只有一行的 C 程序：

```
void main(void) { }
```

尽管这个程序没有做任何事情，但还是需要做很多工作来建立执行环境（参见本章后面的“exec 函数”一节），并在程序终止时杀死这个进程（参见第三章的“撤消进程”一节）。尤其是当 main() 函数终止时，C 编译程序把 `exit_group()` 函数插入到目标代码中。

从第十章我们知道，程序通常通过 C 库中的封装例程调用系统调用。C 编译器亦如此。任何可执行文件除了包括对程序的语句进行编译所直接产生的代码外，还包括一些“粘合”代码来处理用户态进程与内核之间的交互。这样的粘合代码有一部分存放在 C 库中。

除了 C 库，Unix 系统中还包含很多其他的函数库。一般的 Linux 系统通常就有几百个不同的库。这里仅仅列举其中的两个：数学库 `libm` 包含浮点操作的基本函数，而 `X11 库 libX11` 收集了所有 X11 窗口系统图形接口的基本底层函数。

传统 Unix 系统中的所有可执行文件都是基于静态库 (*static library*) 的。这就意味着链接程序所产生的可执行文件不仅包括原程序的代码，还包括程序所引用的库函数的代码。

静态库的一大缺点是：它们占用大量的磁盘空间。的确，每个静态链接的可执行文件都复制库代码的某些部分。

现代 Unix 系统利用共享库 (*shared library*)。可执行文件不用再包含库的目标代码，而仅仅指向库名。当程序被装入内存执行时，一个名为动态链接器 (*dynamic linker*, 也叫 `ld.so`) 的程序就专注于分析可执行文件中的库名，确定所需库在系统目录树中的位置，并使执行进程可以使用所请求的代码。进程也可以使用 `dlopen()` 库函数在运行时装入额外的共享库。

共享库对提供文件内存映射的系统尤为方便，因为它们减少了执行一个程序所需的主内存量。当动态链接程序必须把某一共享库链接到进程时，并不拷贝目标代码，而是仅仅执行一个内存映射，把库文件的相关部分映射到进程的地址空间中。这就允许共享库机器代码所在的页框被使用同一代码的所有进程共享。显然，如果程序是静态链接的，那么共享是不可能的。

共享库也有一些缺点。动态链接的程序启动时间通常比静态链接的程序长。此外，动态链接的程序的移植性也不如静态链接的好，因为当系统中所包含的库版本发生变化时，动态链接的程序运行时就可能出现问题。

用户可以始终请求一个程序被静态地链接。例如，GCC 编译器提供 `-static` 选项，即告诉链接程序使用静态库而不是共享库。

程序段和进程的线性区

从逻辑上说，Unix 程序的线性地址空间传统上被划分为几个叫做段 (*segment*) (注 4) 的区间：

正文段

包含程序的可执行代码。

已初始化数据段

包含已初始化的数据，也就是初值存放在可执行文件中的所有静态变量和全局变量（因为程序在启动时必须知道它们的值）。

未初始化数据段 (*bss* 段)

包含未初始化的数据，也就是初值没有存放在可执行文件中的所有全局变量（因为程序在引用它们之前才赋值）；历史上把这个段叫做 *bss* 段。

注 4：“段”这个术语有其历史根源，因为第一个 Unix 系统用不同的段寄存器实现每一个线性地址区间。不过，Linux 并不利用 80x86 微处理器的段机制实现程序分段。

堆栈段

包含程序的堆栈，堆栈中有返回地址、参数和被执行函数的局部变量。

每个 `mm_struct` 内存描述符（参见第九章中的“内存描述符”一节）都包含一些字段来标识相应进程特定线性区的作用：

`start_code, end_code`

程序的源代码所在线性区的起始和终止线性地址，即可执行文件中的代码。

`start_data, end_data`

程序的初始化数据所在线性区的起始和终止线性地址，正如在可执行文件中所指定的那样。这两个字段指定的线性区大体上与数据段对应。

`start_brk, brk`

存放线性区的起始和终止线性地址，该线性区包含动态分配给进程的内存区（参看第九章的“堆的管理”一节）。有时把这部分线性区叫做堆（*heap*）。

`start_stack`

正好在 `main()` 的返回地址之上的地址。如图 20-1 所示，更高的地址被保留（回想一下，栈是向低地址增长）。

`arg_start, arg_end`

命令行参数所在的堆栈部分的起始地址和终止地址。

`env_start, env_end`

环境串所在的堆栈部分的起始地址和终止地址。

注意，共享库和文件的内存映射使得基于程序段的进程地址空间分类有点过时，因为每个共享库被映射到与前面所讨论的线性区不同的线性区。

灵活线性区布局

灵活线性区布局（*flexible memory region layout*）在内核版本 2.6.9 中引入：实际上，每个进程是按照用户态堆栈预期的增长量来进行内存布局的。但是仍然可以使用老的经典布局（主要用于：当内核无法限制进程用户态堆栈的大小时）。表 20-4 以 80x86 结构的默认用户态地址空间为例描述了这两种布局，地址空间最大可以到 3GB。

正如你所看到的，布局之间只在文件内存映射与匿名映射时线性区的位置上有区别。在经典布局下，这些区域从整个用户态地址空间的 1/3 开始，通常在地址 `0x40000000`。新的区域往更高线性地址追加，因此，这些区域往用户态堆栈方向扩展。

表 20-4: 80x86结构的线性区布局

线性区种类	经典布局	灵活布局
正文段 (ELF)		起自: 0x08048000
数据与 bss 段		起自: 紧接正文段之后
堆		起自: 紧接数据与 bss 段之后
文件内存映射与匿名线性区	起自: 0x40000000 (该地址对应于整个用户态地址空间的 1/3), 库连续往高地址追加	起自: 紧接用户态堆栈尾 (最小地址), 库连续往低地址追加
用户态堆栈		起自: 0xc0000000 并向低地址增长

而相反的是,在灵活布局中,文件内存映射与匿名映射的线性区是紧接用户态堆栈尾的。新的区域往更低线性地址追加,因此,这些区域往堆的方向扩展。记住,堆栈也是连续往低地址追加的。

当内核能通过 `RLIMIT_STACK` 资源限制来限定用户态堆栈的大小时,通常使用灵活布局(参见第三章“进程资源限制”一节)。这个限制确定了为堆栈保留的线性地址空间大小。但是这个空间大小不能小于 128MB 或大于 2.5GB。

另外,如果 `RLIMIT_STACK` 资源限制设为无限 (infinity),或者系统管理员将 `sysctl_legacy_va_layout` 变量设为 1 (通过修改 `/proc/sys/vm/legacy_va_layout` 文件或调用相应的 `sysctl()` 系统调用实现),内核无法确定用户态堆栈的上限,就仍然使用经典线性区布局。

为什么引入灵活布局?其主要优点是可以允许进程更好地使用用户态线性地址空间。在经典布局中,堆的限制是小于 1GB,而其他线性区可以使用到约 2GB (要减去堆栈大小)。在灵活布局中,这些限制没有了,堆和其他线性区可以自由扩展,可以使用除了用户态堆栈和程序用固定大小的段以外的所有线性地址空间。

现在,一个实用的小试验很有启发意义。让我们录入和编译下面的 C 程序:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
    char cmd[32];
    brk((void *)0x8051000);
    sprintf(cmd, "cat /proc/self/maps");
    system(cmd);
}
```

```
    return 0;
}
```

实际上，程序将它的进程堆变大（参见第九章“堆的管理”一节），然后在 */proc* 特殊文件系统下读入 *maps* 文件，该文件产生进程自身的线性区清单。

让我们对堆栈大小不加任何限制并运行程序：

```
# ulimit -s unlimited; /tmp/memorylayout
08048000-08049000 r-xp 00000000 03:03 5042408 /tmp/memorylayout
08049000-0804a000 rwxp 00000000 03:03 5042408 /tmp/memorylayout
0804a000-08051000 rwxp 0804a000 00:00 0
40000000-40014000 r-xp 00000000 03:03 620801 /lib/ld-2.3.2.so
40014000-40015000 rwxp 00013000 03:03 620801 /lib/ld-2.3.2.so
40015000-40016000 rwxp 40015000 00:00 0
4002f000-40157000 r-xp 00000000 03:03 620804 /lib/libc-2.3.2.so
40157000-4015b000 rwxp 00128000 03:03 620804 /lib/libc-2.3.2.so
4015b000-4015e000 rwxp 4015b000 00:00 0
bffff000-c0000000 rwxp bffff000 00:00 0
ffffe000-ffffff00 ---p 00000000 00:00 0
```

（由于C编译器的版本不同与程序链接方式不同，见到的结果可能略有不同。）前两个十六进制数表示线性区的范围，后面是权限标志。最后面是线性区映射的文件的有关信息，如果有信息就是：文件内的开始偏移量、块设备号、索引节点号和文件名。

请注意，列出的所有区域是由私有内存映射实现的（权限列的 *p* 字母）。这并不奇怪，因为这些线性区是只为进程提供数据而存在的。当执行指令时，进程可以修改这些线性区的内容，但是与它们相关的磁盘文件会保持不变。私有内存映射就具有如此作用。

从 *0x08048000* 开始的线性区是与 */tmp/memorylayout* 文件的 *0~4095* 字节部分对应的内存映射。而相应的权限表示是可执行的（它包含了目标代码）、只读的（因为指令在执行期间是不改变的，因此不可写）和私有的。这很正确，这是程序正文段的映射区域。

从 *0x08049000* 开始的线性区也是与 */tmp/memorylayout* 文件的 *0~4095* 字节部分对应的另一个内存映射。这个程序太小，以至于程序的正文、数据和 *bss* 段都在同一个文件页里。因此，包含数据段和 *bss* 段的线性区与上一个线性区在线性地址空间是重叠的。

第三个线性区包含进程的堆。注意，它在线性地址 *0x08051000* 处终止，传递给 *brk()* 系统调用的就是该地址。

接下来从 *0x40000000* 和 *0x40014000* 开始的两个线性区，分别对应这个系统 ELF 共享库 (*/lib/ld-2.3.2.so*) 动态链接程序的正文段和数据、*bss* 段。动态链接程序决不单独执行，它总是以内存映射的方式映射到执行其他程序的进程地址空间内。从 *0x40015000* 开始的匿名线性区已由动态链接程序分配。

在这个系统上，C库正好存放在文件 `/lib/libc-2.3.2.so` 中。C库的正文段和数据、bss段被映射到从 `0x4002f000` 地址开始的两个线性区。还记得私有区域所在的页框，只要没被修改，就可以通过写时复制机制在几个进程间共享。因此，因为正文段是只读的，所示包含C库执行代码的页框几乎在所有当前运行进程间共享（除了静态链接程序）。从 `0x4015b000` 开始的匿名线性区已由C库分配。

从 `0xbffeb000` 到 `0xc0000000` 的匿名内存区对应于用户态堆栈。我们在第九章“缺页异常处理程序”一节已讨论过堆栈是如何在必要时自动地向低地址方向扩展的。

最后，从 `0xffffe000` 开始的单页匿名线性区包含进程的 `vsyscall` 页，当发出系统调用和从信号处理程序返回时会访问该区域（参见第十章“通过 `sysenter` 指令发出系统调用”一节和第十一章“捕获信号”一节）。

现在我们对用户态堆栈大小施加限制后再运行该程序：

```
# ulimit -s 100; /tmp/memorylayout
08048000-08049000 r-xp 00000000 03:03 5042408 /tmp/memorylayout
08049000-0804a000 rwxp 00000000 03:03 5042408 /tmp/memorylayout
0804a000-08051000 rwxp 0804a000 00:00 0
b7ea3000-b7fcb000 r-xp 00000000 03:03 620804 /lib/libc-2.3.2.so
b7fcb000-b7fcf000 rwxp 00128000 03:03 620804 /lib/libc-2.3.2.so
b7fcf000-b7fd2000 rwxp b7fcf000 00:00 0
b7feb000-b7fec000 rwxp b7feb000 00:00 0
b7fec000-b8000000 r-xp 00000000 03:03 620801 /lib/ld-2.3.2.so
b8000000-b8001000 rwxp 00013000 03:03 620801 /lib/ld-2.3.2.so
bffeb000-c0000000 rwxp bffeb000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

我们注意到布局发生了变化，即在最高堆栈地址之上为动态链接程序映射了一个约128MB的区域。而且，因为C库的线性区在稍后创建，所以就得到一个较低的线性地址。

执行跟踪

执行跟踪 (*execution tracing*) 是一个程序监视另一个程序执行的一种技术。被跟踪的程序一步一步地执行，直到接收到一个信号或调用一个系统调用。执行跟踪由调试程序 (debugger) 广泛使用，当然还使用其他技术 (包括在被调试程序中插入断点及运行时访问它的变量)。与往常一样，我们将集中讨论内核怎样支持执行跟踪，而不讨论调试程序怎样工作。

在Linux中，通过 `ptrace()` 系统调用进行执行跟踪，这个系统调用能处理如表 20-5 所示的命令。设置了 `CAP_SYS_PTRACE` 权能的进程可以跟踪系统中的任何进程（除了 `init`）。相反，没有 `CAP_SYS_PTRACE` 权能的进程 `P` 只能跟踪与 `P` 有相同属主的进程。此外，两个进程不能同时跟踪一个进程。

表 20-5: 80x86结构的 ptrace 命令

命令	说明
PTRACE_ATTACH	对另一个进程开始执行跟踪
PTRACE_CONT	重新恢复执行
PTRACE_DETACH	终止执行跟踪
PTRACE_GET_THREAD_AREA	代表被跟踪进程获得线程局部存储区 (TLS)
PTRACE_GETEVENTMSG	从被跟踪进程获得附加数据 (如, 新创建进程的 PID)
PTRACE_GETFPREGS	读浮点寄存器
PTRACE_GETFPXREGS	读 MMX 和 XMM 寄存器
PTRACE_GETREGS	读有特权的 CPU 的寄存器
PTRACE_GETSIGINFO	获得传给被跟踪进程最后一条信号的信息
PTRACE_KILL	删除被跟踪的进程
PTRACE_OLDSETOPTIONS	依赖于结构的命令, 等价于 PTRACE_SETOPTIONS
PTRACE_PEEKDATA	从数据段读一个 32 位值
PTRACE_PEEKTEXT	从文本段读一个 32 位值
PTRACE_PEEKUSR	读 CPU 的普通和调试寄存器
PTRACE_POKEDATA	把一个 32 位值写入数据段
PTRACE_POKETEXT	把一个 32 位值写入正文段
PTRACE_POKEUSR	写 CPU 的普通和调试寄存器
PTRACE_SET_THREAD_AREA	代表被跟踪进程设置线程局部存储区 (TLS)
PTRACE_SETFPREGS	写浮点寄存器
PTRACE_SETFPXREGS	写 MMX 和 XMM 寄存器
PTRACE_SETOPTIONS	修改 ptrace() 的行为
PTRACE_SETREGS	写有特权的 CPU 寄存器
PTRACE_SETSIGINFO	建立传给被跟踪进程最后一条信号的信息
PTRACE_SINGLESTEP	恢复单条汇编指令的执行
PTRACE_SYSCALL	恢复执行直到下一个系统调用的边界
PTRACE_TRACEME	对当前进程开始执行跟踪

ptrace() 系统调用修改被跟踪进程描述符的 parent 字段以使它指向跟踪进程, 因此, 跟踪进程变为被跟踪进程的有效父进程。当执行跟踪终止时, 也就是当以 PTRACE_DETACH 命令调用 ptrace() 时, 这个系统调用把 p_pptr 设置为 real_parent 的值, 恢复被跟踪进程原来的父进程 (参见第三章的“进程之间的关系”一节)。

与被跟踪程序相关的几个监控事件为：

- 一条单独汇编指令执行的结束
- 进入系统调用
- 退出系统调用
- 接收到一个信号

当一个监控的事件发生时，被跟踪的程序停止，并且将 SIGCHID 信号发送给它的父进程。当父进程希望恢复子进程的执行时，就使用 PTRACE_CONT、PTRACE_SINGLESTEP 和 PTRACE_SYSCALL 命令中的一条命令，这取决于父进程要监控哪种事件。

PTRACE_CONT 命令只继续执行，子进程将一直执行到收到另一个信号。这种跟踪是通过进程描述符的 ptrace 字段中的 PF_PTRACED 标志实现的，而这个标志的检查是由 do_signal() 函数进行的（参看第十一章中的“传递信号”一节）。

PTRACE_SINGLESTEP 命令强迫子进程执行下一条汇编语言指令，然后又停止它。这种跟踪是基于 80x86 机器的 eflags 寄存器的 TF 陷阱标志而实现的。当这个标志为 1 时，在任一条汇编语言指令之后正好产生一个“Debug”异常。相应的异常处理程序只是清除这个标志，强迫当前进程停止，并发送 SIGCHLD 信号给父进程。注意，设置 TF 标志并不是特权操作，因此用户态进程即使在没有 ptrace() 系统调用的情况下，也能强迫单步执行。内核检查进程描述符中的 PT_DTRACE 标志，以跟踪子进程是否通过 ptrace() 进行单步执行。

PTRACE_SYSCALL 命令使被跟踪的进程重新恢复执行，直到一个系统调用被调用。进程停止两次，第一次是在系统调用开始时，第二次是在系统调用终止时。这种跟踪是利用进程描述符中的 TIF_SYSCALL_TRACE 标志实现的。这个标志是在进程 thread_info 结构的 flags 字段中，并在 system_call() 汇编语言的函数中被检查（参见第十章“通过 int \$0 × 80 指令发出系统调用”一节）。

也可以利用 Intel Pentium 处理器的一些调试特点来跟踪进程。例如，父进程使用 PTRACE_POKEUSR 命令为子进程设置 dr0, …, dr7 调试寄存器的值。当由某调试寄存器监控的事情发生时，CPU 产生“Debug”异常，异常处理程序然后挂起被调试的进程并给父进程发送 SIGCHLD 信号。

可执行格式

Linux 标准的可执行格式是 ELF(*Executable and Linking Format*)，它由 Unix 系统实验

室开发并在 Unix 世界相当流行。几个著名的 Unix 操作系统（如 System V Release 4 和 Sun 的 Solaris 2）都把 ELF 作为它们的主要可执行格式。

Linux 的旧版支持另一种名叫 Assembler OUTput Format (a.out) 的格式，实际上，在 Unix 世界有好几种版本使用这种格式。因为现在 ELF 非常实用，因此已经很少用 a.out 格式。

Linux 支持很多其他不同格式的可执行文件。在这种方式下，Linux 能运行为其他操作系统所编译的程序，如 MS-DOS 的 EXE 程序，或 BSD Unix 的 COFF 可执行格式。有几种可执行格式，如 Java 或 bash 脚本，是与平台无关的。

由类型为 linux_binfmt 的对象所描述的可执行格式实质上提供以下三种方法：

load_binary

通过读存放在可执行文件中的信息为当前进程建立一个新的执行环境。

load_shlib

用于动态地把一个共享库捆绑到一个已经在运行的进程，这是由 uselib() 系统调用激活的。

core_dump

在名为 core 的文件中存放当前进程的执行上下文。这个文件通常是在进程接收到一个缺省操作为“dump”的信号时被创建的，其格式取决于被执行程序的可执行类型（参见第十一章的“传递信号之前所执行的操作”一节）。

所有的 linux_binfmt 对象都处于一个单向链表中，第一个元素的地址存放在 formats 变量中。可以通过调用 register_binfmt() 和 unregister_binfmt() 函数在链表中插入和删除元素。在系统启动期间，为每个编译进内核的可执行格式都执行 register_binfmt() 函数。当实现了一个新的可执行格式的模块正被装载时，也执行这个函数，当模块被卸载时，执行 unregister_binfmt() 函数。

在 formats 链表中的最后一个元素总是对解释脚本 (*interpreted script*) 的可执行格式进行描述的一个对象。这种格式只定义了 load_binary 方法。其相应的 load_script() 函数检查这种可执行文件是否以两个#! 字符开始。如果是，这个函数就把第一行的其余部分解释为另一个可执行文件的路径名，并把脚本文件名作为参数传递以执行它（注 5）。

注 5： 只要以用户 shell 能识别的语言把文件写入脚本文件，即使不以#! 字符开始，也可能执行这个脚本文件。但是，在这种情况下，用 shell（用户在 shell）或缺省的 Bourne shell sh 来对这种脚本进行解释，因此并不直接涉及内核。

Linux 允许用户注册自己定义的可执行格式。对这种格式的识别或者通过存放在文件前 128 字节的魔数，或者通过表示文件类型的扩展名。例如，MS-DOS 的扩展名由“.”把三个字符从文件名中分离出来：`.exe` 扩展名标识可执行文件，而 `.bat` 扩展名标识 shell 脚本。

当内核确定可执行文件是自定义格式时，它就启动相应的解释程序(*interpreter program*)。解释程序运行在用户态，读入可执行文件的路径名作为参数，并执行计算。例如，包含 Java 程序的可执行文件就由 Java 虚拟机（如 `/usr/lib/java/bin/java`）来解释。

这种机制与脚本格式类似，但功能更加强大，这是因为它对自定义格式不加任何限制。要注册一个新格式，就必须在 `binfmt_misc` 特殊文件系统（通常在 `/proc/sys/fs/binfmt_misc`）的注册文件内写入一个字符串，其格式如下：

```
:name:type:offset:string:mask:interpreter:flags
```

这里，每个字段的含义如下：

name

新格式的标识符。

type

识别类型（M 表示魔数，E 表示扩展）。

offset

魔数在文件中的起始偏移量。

string

以魔数或者以扩展名匹配的字节序列。

mask

用来屏蔽掉 string 中的一些位的字符串。

interpreter

解释程序的完整路径名。

flags

可选标志，控制必须怎样调用解释程序。

例如，超级用户执行的下列命令将使内核识别出 Microsoft Windows 的可执行格式：

```
$ echo ':DOSWin:M:0:MZ:0xff:/usr/bin/wine:'
> /proc/sys/fs/binfmt_misc/register
```

Windows 可执行文件的前两个字节是魔数 MZ，由解释程序 `/usr/bin/wine` 执行这个可执行文件。

执行域

在第一章已提到，Linux 的一个巧妙的特点就是能执行其他操作系统所编译的程序。当然，只有内核运行的平台与可执行文件包含的机器代码对应的平台相同时这才是可能的。对这些“外来”程序提供两种支持：

- 模拟执行 (emulated execution)：程序中包含的系统调用与 POSIX 不兼容时才有必要执行这种程序。
- 原样执行 (native execution)：只有程序中所包含的系统调用完全与 POSIX 兼容时才有效。

Microsoft MS-DOS 和 Windows 程序是被模拟执行的，因为它们包含的 API 不能被 Linux 所认识，因此不能原样执行。像 DOSEmu 或 Wine 这样的模拟程序（出现在上一节末尾的例子中）被调用来把每个 API 调用转换为一个模拟的封装函数调用，而封装函数调用又使用现有的 Linux 系统调用。因为模拟程序主要是作为用户态的应用程序来执行，因此我们在此不做进一步的讨论。

另一方面，不用太费力就可以执行为其他操作系统编译的与 POSIX 兼容的程序，因为与 POSIX 兼容的操作系统都提供了类似的 API（尽管实际上并不总是这种情况，但 API 应该相同）。内核必须消除的细微差别通常涉及如何调用系统调用或如何给各种信号编号。这种信息存放在类型为 `exec_domain` 的执行域描述符 (*execution domain descriptor*) 中。

进程可以指定它的执行域，这是通过设置进程描述符的 `personality` 字段，以及把相应 `exec_domain` 数据结构的地址存放到 `thread_info` 结构的 `exec_domain` 字段来实现的。进程可以通过发布一个叫做 `personality()` 的系统调用来改变它的个性 (*personality*)；表 20-6 列出了这个系统调用的参数所接收的典型值。程序员通常不希望直接改变其程序的个性；相反，应该通过建立进程的执行上下文的“粘合”代码来发出 `Personality()` 系统调用（参见下一节）。

表 20-6：Linux 内核所支持的主要个性

个性	操作系统
<code>PER_LINUX</code>	标准执行域
<code>PER_LINUX_32BIT</code>	Linux，64 位结构中 32 位物理地址
<code>PER_LINUX_FDPIC</code>	Linux 程序，格式为 ELF FDPIC
<code>PER_SVR4</code>	System V Release 4
<code>PER_SVR3</code>	System V Release 3
<code>PER_SCOSVR3</code>	SCO Unix Version 3.2

表 20-6: Linux 内核所支持的主要个性 (续)

个性	操作系统
PER_OSR5	SCO OpenServer Release 5
PER_WYSEV386	Unix System V/386 Release 3.2.1
PER_ISCR4	交互式 Unix
PER_BSD	BSD Unix
PER_SUNOS	SunOS
PER_XENIX	Xenix
PER_LINUX32	64 位结构中 32 位 Linux 程序模拟 (使用 4GB 用户态地址空间)
PER_LINUX32_3GB	64 位结构中 32 位 Linux 程序模拟 (使用 3GB 用户态地址空间)
PER_IRIX32	SGI Irix-5 32 位
PER_IRIXN32	SGI Irix-6 32 位
PER_IRIX64	SGI Irix-6 64 位
PER_RISCOS	RISC OS
PER_SOLARIS	Sun 的 Solaris
PER_UW7	SCO(正式为 Caldera)的 UnixWare 7
PER_OSF4	Digital UNIX (Compaq Tru64 UNIX)
PER_HPUX	HP 的 HP-UX

exec 函数

Unix 系统提供了一系列函数, 这些函数能用可执行文件所描述的新上下文代替进程的上下文。这样的函数名以前缀 `exec` 开始, 后跟一个或两个字母, 因此, 家族中的一个普通函数被当作 `exec` 函数来引用。

表 20-7 中列出了 `exec` 函数, 它们之间的差别在于如何解释参数。

表 20-7: `exec` 函数

函数名	路径搜索	命令行参数	环境数组
<code>execl()</code>	否	列表	否
<code>execlp()</code>	是	列表	否
<code>execle()</code>	否	列表	是
<code>execv()</code>	否	数组	否

表 20-7: exec 函数 (续)

函数名	路径搜索	命令行参数	环境数组
execvp()	是	数组	否
execve()	否	数组	是

每个函数的第一个参数表示被执行文件的路径名。路径名可以是绝对路径或是当前进程目录的相对路径。此外，如果路径名中不包含“/”字符，`execlp()`和`execvp()`函数就在 `PATH` 环境变量指定的所有目录中搜索这个可执行文件。

除了第一个参数，`execl()`、`execlp()`和`execle()`函数包含的其他参数个数都是可变的。每个参数指向一个字符串，这个字符串是对新程序命令行参数的描述，正如函数名中“l”字符所隐含的一样，这些参数组织成一个列表（最后一个值为 `NULL`）。通常情况下，第一个命令行参数复制可执行文件名。相反，`execv()`、`execvp()`和`execve()`函数指定单个参数的命令行参数，正如函数名中的“v”字符所隐含的一样，这单个参数是指向命令行参数串的指针向量地址。数组的最后一个元素必须存放 `NULL` 值。

`execle()`和`execve()`函数的最后一个参数是指向环境串的指针数组的地址；数组的最后一个元素照样必须为 `NULL`。其他函数对新程序环境参数的访问是通过 C 库定义的外部全局变量 `environ` 进行的。

所有的 `exec` 函数（除 `execve()` 外）都是 C 库定义的封装例程，并利用了 `execve()` 系统调用，这是 Linux 所提供的处理程序执行的唯一系统调用。

`sys_execve()` 服务例程接收下列参数：

- 可执行文件路径名的地址（在用户态地址空间）。
- 以 `NULL` 结束的字符串指针数组的地址（在用户态地址空间）。每个字符串表示一个命令行参数。
- 以 `NULL` 结束的字符串指针数组的地址（也在用户态地址空间）。每个字符串以 `NAME=value` 形式表示一个环境变量。

`sys_execve()` 把可执行文件路径名拷贝到一个新分配的页框。然后调用 `do_execve()` 函数，传递给它的参数为指向这个页框的指针、指针数组的指针及把用户态寄存器内容保存到内核态堆栈的位置。`do_execve()` 依次执行下列操作：

1. 动态地分配一个 `linux_binprm` 数据结构，并用新的可执行文件的数据填充这个结构。

2. 调用 `path_lookup()`、`dentry_open()` 和 `path_release()`，以获得与可执行文件相关的目录项对象、文件对象和索引节点对象。如果失败，则返回相应的错误码。
3. 检查是否可以由当前进程执行该文件，再检查索引节点的 `i_writecount` 字段，以确定可执行文件没被写入；把 `-1` 存放在这个字段以禁止进一步的写访问。
4. 在多处理器系统中，调用 `sched_exec()` 函数来确定最小负载 CPU 以执行新程序，并把当前进程转移过去（参见第七章）。
5. 调用 `init_new_context()` 检查当前进程是否使用自定义局部描述符表，参见第二章“Linux LDT”一节）。如果是，函数为新程序分配和准备一个新的 LDT。
6. 调用 `prepare_binprm()` 函数填充 `linux_binprm` 数据结构，这个函数又依次执行下列操作：
 - a. 再一次检查文件是否可执行（至少设置一个执行访问权限）。如果不可执行，则返回错误码（因为带有 `CAP_DAC_OVERRIDE` 权能的进程总能通过检查，所以第 3 步中的检查还不够。参见本章前面“进程的信任状和权能”一节）。
 - b. 初始化 `linux_binprm` 结构的 `e_uid` 和 `e_gid` 字段，考虑可执行文件的 `setuid` 和 `setgid` 标志的值。这些字段分别表示有效的用户 ID 和组 ID。也要检查进程的权能（在本章前面的“进程的信任状和权能”一节中介绍了兼容性技巧）。
 - c. 用可执行文件的前 128 字节填充 `linux_binprm` 结构的 `buf` 字段。这些字节包含的是适合于识别可执行文件格式的一个魔数和其他信息。
7. 把文件路径名、命令行参数及环境串拷贝到一个或多个新分配的页框中（最终，它们会被分配给用户态地址空间）。
8. 调用 `search_binary_handler()` 函数对 `formats` 链表进行扫描，并尽力应用每个元素的 `load_binary` 方法，把 `linux_binprm` 数据结构传递给这个函数。只要 `load_binary` 方法成功应答了文件的可执行格式，对 `formats` 的扫描就终止。
9. 如果可执行文件格式不在 `formats` 链表中，就释放所分配的所有页框并返回错误码 `-ENOEXEC`，表示 Linux 不认识这个可执行文件格式。
10. 否则，函数释放 `linux_binprm` 数据结构，返回从这个文件可执行格式的 `load_binary` 方法中所获得的代码。

可执行文件格式对应的 `load_binary` 方法执行下列操作（我们假定这个可执行文件所在的文件系统允许文件进行内存映射并需要一个或多个共享库）：

1. 检查存放在文件前 128 字节中的一些魔数以确认可执行格式。如果魔数不匹配，则返回错误码 `-ENOEXEC`。

2. 读可执行文件的首部。这个首部描述程序的段和所需的共享库。
3. 从可执行文件获得动态链接程序的路径名,并用它来确定共享库的位置并把它们映射到内存。
4. 获得动态链接程序的目录项对象(也就获得了索引节点对象和文件对象)。
5. 检查动态链接程序的执行许可权。
6. 把动态链接程序的前 128 字节拷贝到缓冲区。
7. 对动态链接程序类型执行一些一致性检查。
8. 调用 `flush_old_exec()` 函数释放前一个计算所占用的几乎所有资源。这个函数又依次执行下列操作:
 - a. 如果信号处理程序的表为其他进程所共享,那么就分配一个新表并把旧表的引用计数器减 1,而且它将进程从旧的线程组脱离(参见第三章“标识一个进程”一节)。这是通过调用 `de_thread()` 函数完成的。
 - b. 如果与其他进程共享,就调用 `unshare_files()` 函数拷贝一份包含进程已打开文件的 `files_struct` 结构。
 - c. 调用 `exec_mmap()` 函数释放分配给进程的内存描述符、所有线性区及所有页框,并清除进程的页表。
 - d. 将可执行文件路径名赋给进程描述符的 `comm` 字段。
 - e. 调用 `flush_thread()` 函数清除浮点寄存器的值和 `TSS` 段保存的调试寄存器的值。
 - f. 调用 `flush_signal_handlers()` 函数,用于将每个信号恢复为默认操作,从而更新信号处理程序的表。
 - g. 调用 `flush_old_files()` 函数关闭所有打开的文件,这些打开的文件在进程描述符的 `files->close_on_exec` 字段设置了相应的标志(参见第十二章中的“与进程相关的文件”一节)(注 6)。

现在,我们已经不能返回了:如果真出了差错,这个函数再不能恢复前一个计算。

9. 清除进程描述符的 `PF_FORKNOEXEC` 标志。这个标志用于在进程创建时设置进程记账,在执行一个新程序时清除进程记账。
10. 设立进程新的个性,即设置进程描述符的 `personality` 字段。

注 6: 可以通过 `fcntl()` 系统调用来读取和修改这些标志。

11. 调用 `arch_pick_mmap_layout()`，以选择进程线性区的布局（参见本章前面“程序段与进程的线性”一节）。
12. 调用 `setup_arg_pages()` 函数为进程的用户态堆栈分配一个新的线性区描述符，并把那个线性区插入到进程的地址空间。`setup_arg_pages()` 还把命令行参数和环境变量串所在的页框分配给新的线性区。
13. 调用 `do_mmap()` 函数创建一个新线性区来对可执行文件正文段（即代码）进行映射。这个线性区的起始线性地址依赖于可执行文件的格式，因为程序的可执行代码通常是不可重定位的。因此，这个函数假定从某一特定逻辑地址的偏移量开始（因此就从某一特定的线性地址开始）装入正文段。ELF 程序被装入的起始线性地址为 `0x08048000`。
14. 调用 `do_mmap()` 函数创建一个新线性区来对可执行文件的数据段进行映射。这个线性区的起始线性地址也依赖于可执行文件的格式，因为可执行代码希望在特定的偏移量（即特定的线性地址）处找到它自己的变量。在 ELF 程序中，数据段正好被装在正文段之后。
15. 为可执行文件的其他专用段分配另外的线性区，通常是无。
16. 调用一个装入动态链接程序的函数。如果动态链接程序是 ELF 可执行的，这个函数就叫做 `load_elf_interp()`。一般情况下，这个函数执行第 12~14 步的操作，不过要用动态链接程序代替被执行的文件。动态链接程序的正文段和数据段在线性区的起始线性地址是由动态链接程序本身指定的，但它们处于高地址区（通常高于 `0x40000000`），这是为了避免与被执行文件的正文段和数据段所映射的线性区发生冲突（参见前面的“程序段和进程的线性区”一节）。
17. 把可执行格式的 `linux_binfmt` 对象的地址存放在进程描述符的 `binfmt` 字段中。
18. 确定进程的新权能。
19. 创建特定的动态链接程序表并把它们存放在用户态堆栈，如图 20-1 所示，这些表处于命令行参数和指向环境串的指针数组之间。
20. 设置进程的内存描述符的 `start_code`、`end_code`、`start_data`、`end_data`、`start_brk`、`brk` 及 `start_stack` 字段。
21. 调用 `do_brk()` 函数创建一个新的匿名线性区来映射程序的 `bss` 段（当进程写入一个变量时，就触发请求调页，进而分配一个页框）。这个线性区的大小是在可执行程序被链接时就计算出来的。因为程序的可执行代码通常是不可重新定位的，因此，必须指定这个线性区的起始线性地址。在 ELF 程序中，`bss` 段正好装在数据段之后。

22. 调用`start_thread()`宏修改保存在内核态堆栈但属于用户态寄存器的`eip`和`esp`的值，以使它们分别指向动态链接程序的入口点和新的用户态堆栈的栈顶。
23. 如果进程正被跟踪，就通知调试程序`execve()`系统调用已完成。
24. 返回0值（成功）。

当`execve()`系统调用终止且调用进程重新恢复它在用户态的执行时，执行上下文被大幅度改变，调用系统调用的代码不复存在。从这个意义上看，我们可以说`execve()`从未成功返回。取而代之的是，要执行的新程序已被映射到进程的地址空间。

但是，新程序还不能执行，因为动态链接程序还必须考虑共享库的装载（注7）。

尽管动态链接程序运行在用户态，但我们还要在这里简要概述一下它是如何运作的。它的第一个工作就是从内核保存在用户态堆栈的信息（处于环境串指针数组和`arg_start`之间）开始，为自己建立一个基本的执行上下文。然后，动态链接程序必须检查被执行的程序，以识别哪个共享库必须装入及在每个共享库中哪个函数被有效地请求。接下来，解释器发出几个`mmap()`系统调用来创建线性区，以对将存放程序实际使用的库函数（正文和数据）的页进行映射。然后，解释器根据库的线性区的线性地址更新对共享库符号的所有引用。最后，动态链接程序通过跳转到被执行程序的主入口点而终止它的执行。从现在开始，进程将执行可执行文件的代码和共享库的代码。

你可能已注意到，执行程序是一个相当复杂的活动，它涉及内核设计的很多方面，如进程抽象、内存管理、系统调用及文件系统。这会使你认识到：Linux真是一个杰作！

注7： 如果可执行文件是静态链接的，即不需要共享库，事情就简单多了。`Load_binary`方法只需将程序的正文段、数据段、`bss`段和堆栈段映射到进程线性区，然后把用户态`eip`寄存器的内容设置为新程序的入口点即可。

系统启动

本附录介绍当用户打开计算机电源之后所发生的事情，也就是说，Linux 内核映像是如何被拷贝到内存的，又是如何被执行的。简而言之，我们讨论内核，继而是整个系统，是如何启动的。

“启动 (*bootstrap*)”这个术语的原意是一个人要穿上靴子站起来。在操作系统中，这个术语专门表示把一部分操作系统装载到主存中并让处理器执行它，也表示内核数据结构的初始化、一些用户进程的创建以及把控制权转移到其中某个进程。

计算机启动是一个冗长乏味的任务，因为最开始时几乎每个硬件设备（包括RAM）都处于一种随机的、不可预知的状态。此外，启动过程在很大程度上都依赖于计算机的体系结构；和以前一样，我们在本附录中特指 80x86 体系结构。

史前时代：BIOS

计算机在加电的那一刻几乎是毫无用处的，因为RAM芯片中包含的是随机数据，此时还没有操作系统在运行。在开始启动时，有一个特殊的硬件电路在CPU的一个引脚上产生一个RESET逻辑值。在RESET产生以后，就把处理器的一些寄存器（包括cs和eip）设置成固定的值，并执行在物理地址 0xffffffff0 处找到的代码。硬件把这个地址映射到某个只读、持久的存储芯片中，该芯片通常称为ROM（Read-Only Memory，只读内存）。ROM中所存放的程序集在80x86体系中通常叫作基本输入/输出系统（*Basic Input/Output System, BIOS*），因为它包括几个中断驱动的低级过程。所有操作系统在启动时，都要通过这些过程对计算机硬件设备初始化。一些操作系统，如微软的MS-DOS，依赖于BIOS实现大部分系统调用。

Linux 一旦进入保护模式（参见第二章“硬件中的分段”一节），就不再使用 BIOS，而是为计算机上的每个硬件设备提供各自的设备驱动程序。实际上，因为 BIOS 过程必须在实模式下运行，所以即使有益，两者之间也不能共享函数。

BIOS 使用实模式的地址，因为在计算机加电启动时只有这些可以使用。一个实模式的地址由一个 *seg* 段和一个 *off* 偏移量组成。相应的物理地址可以这样计算： $seg * 16 + off$ 。所以 CPU 寻址电路根本就不需要全局描述符表、局部描述符表或者页表把逻辑地址转换成物理地址。显然，对 GDT、LDT 和页表进行初始化的代码必须在实模式下运行。

Linux 在启动阶段必须使用 BIOS，此时 Linux 必须要从磁盘或者其他外部设备中获取内核映像。BIOS 启动过程实际上执行以下 4 个操作：

1. 对计算机硬件执行一系列的测试，用来检测现在都有什么设备以及这些设备是否正常工作。这个阶段通常称为 *POST* (*Power-On Self-Test*，上电自检)。在这个阶段中，会显示一些信息，例如 BIOS 版本号。

如今的 80x86、AMD64 和 Itanium 计算机使用高级配置与开机界面 (*Advanced Configuration and Power Interface*, *ACPI*) 标准。在 ACPI 兼容的 BIOS 中，启动代码会建立几个表来描述当前系统中的硬件设备。这些表的格式独立于设备生产商，而且可由操作系统读取以获得如何调用这些设备的信息。

2. 初始化硬件设备。这个阶段在现代基于 PCI 的体系结构中相当重要，因为它可以保证所有的硬件设备操作不会引起 IRQ 线与 I/O 端口的冲突。在本阶段的最后，会显示系统中所安装的所有 PCI 设备的一个列表。
3. 搜索一个操作系统来启动。实际上，根据 BIOS 的设置，这个过程可能要试图访问（按照用户预定义的次序）系统中软盘、硬盘和 CD-ROM 的第一个扇区（引导扇区）。
4. 只要找到一个有效的设备，就把第一个扇区的内容拷贝到 RAM 中从物理地址 0x00007c00 开始的位置，然后跳转到这个地址处，开始执行刚才装载进来的代码。

本附录其余的部分会带你体验从最原始的开始状态到运行 Linux 系统的整个历程。

远古时代：引导装入程序

引导装入程序 (*boot loader*) 是由 BIOS 用来把操作系统的内核映像装载到 RAM 中所调用的一个程序。让我们简要地描绘一下引导装入程序在 IBM 的 PC 体系结构中是如何工作的。

为了从软盘上启动，必须把第一个扇区中所存放的指令装载到 RAM 中并执行，这些指令再把包含内核映像的其他所有扇区都拷贝到 RAM 中。

从硬盘启动的实现有点不同。硬盘的第一个扇区称为主引导记录 (*Master Boot Record, MBR*)，该扇区中包括分区表 (注1) 和一个小程序，这个小程序用来装载被启动的操作系统所在分区的第一个扇区。诸如 Microsoft Windows 98 之类的操作系统使用分区表中所包含的一个活动 (*active*) 标志来标识这个分区 (注2)。按照这种方法，只有那些内核映像存放在活动分区中的操作系统才可以被启动。正如我们将在后面看到的一样，Linux 的处理方式更加灵活，因为 Linux 使用一个巧妙的引导装入程序取代这个 MBR 中不完善的程序，它允许用户来选择要启动的操作系统。

Linux 早期版本 (一直到 2.4 系列) 的内核映像，在第一个 512 字节有一个最小的引导装入程序，因此在第一扇区拷贝一个内核映像就可以使软盘可启动。但是在 Linux 2.6 中就不再有这样的引导装入程序，所以要从软盘启动，就必须在第一个磁盘扇区存放一个合适的引导装入程序。而现在从软盘启动与从硬盘或 CD-ROM 启动是十分相似的。

从磁盘启动 Linux

从磁盘启动 Linux 内核需要一个两步的引导装入程序。在 80x86 体系中，众所周知的 Linux 引导装入程序叫作 LInux LOader (LILO)。确实还有一些 80x86 体系的引导装入程序，如广泛使用的 GRand Unified Bootloader (GRUB)。GRUB 比 LILO 更为先进，因为它可识别多个基于磁盘的文件系统，而且可以从文件中读入部分引导程序。当然，对于 Linux 支持的所有体系结构都有各自专门的引导装入程序。

LILO 或许被装在 MBR 上 (代替那个装载活动引导扇区的小程序)，或许被装在每个磁盘分区的引导扇区上。在这两种情况下，最终的结果是相同的：装入程序在启动过程中被执行时，用户都可以选择装入哪个操作系统。

实际上，LILO 引导装入程序被分为两部分，因为不划分的话，它就太大而无法装进单个扇区。MBR 或者分区引导扇区包括一个小的引导装入程序，由 BIOS 把这个小程序装入从地址 0x00007c00 开始的 RAM 中。这个小程序又把自己移到地址 0x00096a00，建立实模式栈 (0x00098000~0x000969ff)，并把 LILO 的第二部分装入到从地址 0x00096c00 开始的 RAM 中。

第二部分又依次从磁盘读取可用操作系统的映射表，并提供给用户一个提示符，因此用户就可以从中选择一个操作系统。最后，用户选择了被装入的内核后 (或经过一个延迟

注 1: 每个分区表项通常包含分区的起止扇区和处理它的操作系统类型。

注 2: 活动标志可由程序 *fdisk* 设置。

时间以使 LILO 选择一个缺省值)，引导装入程序就可以把相应分区的引导扇区拷贝到 RAM 中并执行它，或直接把内核映像拷贝到 RAM 中。

假定 Linux 内核映像必须被导入，LILO 引导装入程序依赖于 BIOS 例程，主要执行如下步骤：

1. 调用一个 BIOS 过程显示“Loading”信息。
2. 调用一个 BIOS 过程从磁盘装入内核映像的初始部分，即将内核映像的第一个 512 字节从地址 0x00090000 开始存入 RAM 中，而将 setup() 函数的代码（参见下面）从地址 0x00090200 开始存入 RAM 中。
3. 调用一个 BIOS 过程从磁盘中装载其余的内核映像，并把内核映像放入从低地址 0x00010000（适用于使用 make zImage 编译的小内核映像）或者从高地址 0x00100000（适用于使用 make bzImage 编译的大内核映像）开始的 RAM 中。在以下的讨论中，我们将分别称内核映像是“低装载”到 RAM 中或者“高装载”到 RAM 中。大内核映像的支持虽然本质上与其他启动模式相同，但是它却把数据放在不同的物理内存地址，以避免在第二章“物理内存布局”一节所介绍的 ISA 黑洞问题。
4. 跳转到 setup() 代码。

中世纪：setup() 函数

setup() 汇编语言函数的代码由链接程序放在内核映像文件的偏移量 0x200 处。引导装入程序因此就可以很容易地确定 setup() 代码的位置，并把它拷贝到从物理地址 0x00090200 开始的 RAM 中。

setup() 函数必须初始化计算机中的硬件设备，并为内核程序的执行建立环境。虽然 BIOS 已经初始化了大部分硬件设备，但是 Linux 并不依赖于 BIOS，而是以自己的方式重新初始化设备以增强可移植性和健壮性。setup() 本质上执行以下操作：

1. 在 ACPI 兼容的系统中，它调用一个 BIOS 例程，以在 RAM 中建立系统物理内存布局表（通过检索“IOS-e820”标签，就可在引导内核信息中看到该表）。在早期系统中，它调用 BIOS 例程，返回系统可用内存。
2. 设置键盘重复延时和速率（当用户一直按下一个键超过一定的时间，键盘设备就反复地向 CPU 发送相应的键盘码）。
3. 初始化视频卡。
4. 重新初始化磁盘控制器并检测硬盘参数。

5. 检查 IBM 微通道总线 (MCA)。
6. 检查 PS/2 指针设备 (总线鼠标)。
7. 检查对高级电源管理 (APM) BIOS 的支持。
8. 如果 BIOS 支持增强磁盘驱动服务 (*Enhanced Disk Drive Service, EDD*)，它就调用相应的 BIOS 过程在 RAM 中建立系统可用硬盘表 (表中的信息可以通过 *sysfs* 特殊文件系统的 *firmware/edd* 目录查看)。
9. 如果内核映像被低装载到 RAM 中 (在物理地址 0x00010000 处)，就把它移动到物理地址 0x00001000 处。反之，如果内核映像被高装载到 RAM 中，就不用移动。这个步骤是必需的，因为为了能在软盘上存储内核映像并节省启动的时间，存放在磁盘上的内核映像都是压缩的，解压程序需要一些空闲空间作为临时缓冲区 (在 RAM 中紧挨内核映像的地方)。
10. 置位 8042 键盘控制器的 A20 引脚。A20 引脚是在 80286 系统中引入的，为的是与古老的 8088 微处理器物理地址兼容。不幸的是，在切换到保护模式之前必须将 A20 引脚正确置位，否则，每个物理地址的第 21 位都会被 CPU 看作 0。置位 A20 引脚是件讨厌的事情。
11. 建立一个临时中断描述符表 (IDT) 和一个临时全局描述符表 (GDT)。
12. 如果需要，重置浮点单元 (FPU)。
13. 重新编写可编程中断控制器 (Programmable Interrupt Controller, PIC)，以屏蔽所有中断，但保留 IRQ2，它是两个 PIC 之间的级联中断。
14. 通过设置 *cr0* 状态寄存器中的 PE 位，把 CPU 从实模式切换到保护模式。*cr0* 状态寄存器中的 PG 位被清 0，因此分页还没有启用。
15. 跳转到 *startup_32()* 汇编语言函数。

文艺复兴时期：startup_32()函数

有两个不同的 *startup_32()* 函数，我们此处所指的是在 *arch/i386/boot/compressed/head.S* 文件中实现的那一个。在 *setup()* 结束之后，*startup_32()* 就已经被移动到物理地址 0x00100000 处或者 0x00001000 处，这取决于内核映像是被高装载到 RAM 中还是低装载到 RAM 中。

该函数执行以下操作：

1. 初始化段寄存器和一个临时堆栈。

2. 清零 `eflags` 寄存器的所有位。
3. 用 0 填充由 `_edata` 和 `_end` 符号标识的内核未初始化数据区（参见第二章的“物理内存布局”一节）。
4. 调用 `decompress_kernel()` 函数来解压内核映像。首先显示“Uncompressing Linux ...”信息。完成内核映像的解压之后，显示“OK, booting the kernel.”信息。如果内核映像是低装载的，那么解压后的内核就被放在物理地址 `0x00100000` 处。否则，如果内核映像是高装载的，那么解压后的内核就被放在位于这个压缩映像之后的临时缓冲区中。然后，解压后的映像就被移动到从物理地址 `0x00100000` 开始的最终位置。
5. 跳转到物理地址 `0x00100000` 处。

解压的内核映像以包含在 `arch/i386/kernel/head.S` 中的另一个 `startup_32()` 函数开始。这两个函数使用相同的名字不会产生任何问题（除了使读者容易混淆外），因为这两个函数会跳转到自己的起始物理地址去执行。

第二个 `startup_32()` 函数为第一个 Linux 进程（进程 0）建立执行环境。该函数执行以下操作：

1. 把段寄存器初始化为最终值。
2. 把内核的 `bss` 段填充为 0（参见第二十章“程序段和进程的内存区域”一节）。
3. 初始化包含在 `swapper_pg_dir` 的临时内核页表，并初始化 `pg0`，以使线性地址一致地映射同一物理地址，这在第二章“内核页表”一节已经作了说明。
4. 把页全局目录的地址存放在 `cr3` 寄存器中，并通过设置 `cr0` 寄存器的 `PG` 位启用分页。
5. 为进程 0 建立内核态堆栈（参见第三章的“内核线程”一节）。
6. 该函数再一次清零 `eflags` 寄存器的所有位。
7. 调用 `setup_idt()` 用空的中断处理程序填充 IDT（参见第四章的“IDT 的初步初始化”一节）。
8. 把从 BIOS 中获得的系统参数和传递给操作系统的参数放入第一个页框中（参见第二章的“物理内存布局”一节）。
9. 识别处理器的型号。
10. 用 GDT 和 IDT 表的地址来填充 `gdtr` 和 `idtr` 寄存器。
11. 跳转到 `start_kernel()` 函数。

现代：start_kernel()函数

start_kernel()函数完成Linux内核的初始化工作。几乎每天内核部件都是由这个函数进行初始化的，我们只提及其中的少部分：

- 调用 sched_init()函数来初始化调度程序（参见第七章）。
- 调用 build_all_zonelists()函数来初始化内存管理区（参见第八章“内存管理区”一节）。
- 调用 page_alloc_init()函数来初始化伙伴系统分配程序（参见第八章“伙伴系统算法”一节）。
- 调用 trap_init()函数（参见第四章“异常处理”一节）和 init_IRQ()函数（参见第四章“IRQ数据结构”一节）以完成 IDT 初始化。
- 调用 softirq_init()函数初始化 TASKLET_SOFTIRQ 和 HI_SOFTIRQ（参见第四章“软中断”一节）。
- 调用 time_init()函数来初始化系统日期和时间（参见第六章“Linux 计时体系结构”一节）。
- 调用 kmem_cache_init()函数来初始化 slab 分配器（参见第八章的“普通和专用高速缓存”一节）。
- 调用 calibrate_delay()函数以确定 CPU 时钟的速度（参见第六章“延迟函数”一节）。
- 调用 kernel_thread()函数为进程 1 创建内核线程。正如我们在第三章的“内核线程”一节中已经描述的一样，这个内核线程又会创建其他的内核线程并执行 /sbin/init 程序。

在 start_kernel()开始执行之后会显示“Linux version 2.6.11...”信息，除此之外，在 init 程序和内核线程执行的最后阶段还会显示很多其他信息。最后，就会在控制台上出现熟悉的登录提示符（如果在启动时所启动的是 X Window 系统，那么登录提示符就会出现在一个图形窗口中），通知用户 Linux 内核已经启动，现在正在运行。

模块

正如我们在第一章中所介绍的那样，模块 (*module*) 是 Linux 用来高效地利用微内核的理论优点而不会降低系统性能的一种方法。

是否使用模块？

当系统程序员希望给 Linux 内核增加新功能时，就面临一个进退两难的问题：他们应该编写新代码从而将其作为一个模块进行编译，还是应该将这些代码静态地链接到内核中？

通常，系统程序员都倾向于把新代码作为一个模块来实现。因为模块可以根据需要进行链接，这样内核就不会因为装载那些数以百计的很少使用的程序而变得非常庞大，这一点我们后面就会看到。几乎 Linux 内核的每个高层组件——文件系统、设备驱动程序、可执行格式、网络层等等——都可以作为模块进行编译。Linux 的发布版，充分使用模块方式全面地支持多种硬件设备。例如，发布版中会将几十种声卡驱动程序模块放在某个目录下，但是在某个计算机上只会有效加载其中一个声卡驱动程序。

然而，有些 Linux 代码必须被静态链接，也就是说相应组件或者被包含在内核中，或者根本不被编译。典型情况下，这发生在组件需要对内核中静态链接的某个数据结构或函数进行修改时。

例如，假设某个组件必须在进程描述符中引入新字段。链接一个模块并不能修改诸如 `task_struct` 之类已经定义的数据结构，因为即使这个模块使用其数据结构的修改版，所有静态链接的代码看到的仍是原来的版本，这样就很容易发生数据崩溃。对此问题的一种局部解决方法就是“静态地”把新字段加到进程描述符，从而让这个内核组件可以

使用这些字段，而不用考虑组件究竟是如何被链接的。然而，如果该内核组件从未被使用，那么，在每个进程描述符中都复制这些额外的字段就是对内存的浪费。如果新内核组件对进程描述符的大小有很大的增加，那么，只有新内核组件被静态地链接到内核，才可能通过在这个数据结构中增加需要的字段获得较好的系统性能。

再例如，考虑一个内核组件，它要替换静态链接的代码。显然，这样的组件不能作为一个模块来编译，因为在链接模块时内核不能修改已经在RAM中的机器码。例如，系统不可能链接一个改变页框分配方法的模块，因为伙伴系统函数总是被静态地链接到内核（注1）。

内核有两个主要的任务来进行模块的管理。第一个任务是确保内核的其他部分可以访问该模块的全局符号，例如指向模块主函数的入口。模块还必须知道这些符号在内核及其他模块中的地址。因此，在链接模块时，一定要解决模块间的引用关系。第二个任务是记录模块的使用情况，以便在其他模块或者内核的其他部分正在使用这个模块时，不能卸载这个模块。系统使用了一个简单的引用计数器来记录每个模块的引用次数。

模块许可证

Linux 内核许可证（GPL, 版本 2）不限制用户与企业使用其源代码，但是它严格禁止在非 GPL 许可证下发行相关的源代码，而这些代码起源于或大部分起源于 Linux 代码。也就是说，内核开发者要确保他们的代码及其衍生代码可由所有用户自由使用。

但是，模块对这一机制造成了威胁。可能有人只发行一个用于 Linux 内核的二进制格式模块；例如，厂商可能只以二进制格式模块发行一个硬件驱动程序。现在，这种情况较为少见。理论上说，Linux 内核的特性和功能可被只有二进制格式的模块极大地改变，而把基于 Linux 的内核转变成商业产品。

因此，Linux 内核开发者社团不太接受只有二进制格式的模块。Linux 模块的实现就反映出这一点。一般地，使用 `MODULE_LICENSE` 宏，每个模块开发者应当在模块源代码中标出许可证类型。如果许可证是非 GPL 兼容（或根本没有标出），模块就不能使用内核的许多核心函数和数据结构。而且，使用非 GPL 许可证的模块会“玷污”内核，也就是说内核开发者不再考虑内核中可能的缺陷。

注 1： 你可能疑惑为什么不把你所钟爱的内核组件模块化。实际上，总的原因是软件许可证的问题，而不是技术原因。内核开发者想确保核心组件永远不会被仅发布二进制“黑盒”模块的私有代码所代替。

模块的实现

模块是作为 ELF 对象文件存放在文件系统上的，并通过执行 `insmod` 程序链接到内核中（参见后面的“模块的链接和取消”一节）。对于每个模块，系统都分配一个包含以下数据的内存区：

- 一个 `module` 对象
- 表示模块名的一个以 `null` 结束的字符串（所有的模块都必须有唯一的名字）
- 实现模块功能的代码

`module` 对象描述一个模块，其字段如表 B-1 所示。一个双向循环列表存放所有 `module` 对象。链表头部存放在 `modules` 变量中，而指向相邻单元的指针存放在每个 `module` 对象的 `list` 字段中。

表 B-1: `module` 对象

类型	字段名	说明
<code>enum module_state</code>	<code>state</code>	模块的内部状态
<code>struct list_head</code>	<code>list</code>	模块链表的指针
<code>char [60]</code>	<code>name</code>	模块名
<code>struct module_kobject</code>	<code>mkobj</code>	包含一个 <code>kobject</code> 数据结构和指向这个模块对象的指针
<code>struct module_param_attrs *</code>	<code>param_attrs</code>	指向模块参数描述符数组的指针
<code>const struct kernel_symbol *</code>	<code>syms</code>	指向导出符号数组的指针
<code>unsigned int</code>	<code>num_syms</code>	导出符号数
<code>const unsigned long *</code>	<code>crcls</code>	指向导出符号 CRC 值数组的指针
<code>const struct kernel_symbol *</code>	<code>gpl_syms</code>	指向 GPL 格式导出符号数组的指针
<code>unsigned int</code>	<code>num_gpl_syms</code>	GPL 格式导出符号数
<code>const unsigned long *</code>	<code>gpl_crcls</code>	指向 GPL 格式导出符号 CRC 值数组的指针
<code>unsigned int</code>	<code>num_exentries</code>	模块异常表项数
<code>const struct exception_table_entry *</code>	<code>exstable</code>	指向模块异常表的指针

表 B-1: module 对象 (续)

类型	字段名	说明
int (*)(void)	init	模块初始化方法
void *	module_init	用于模块初始化的动态内存区指针
void *	module_core	用于模块核心函数与数据结构的动态内存区指针
unsigned long	init_size	用于模块初始化的动态内存区大小
unsigned long	core_size	用于模块核心函数与数据结构的动态内存区大小
unsigned long	init_text_size	模块初始化的可执行代码大小, 只当链接模块时使用
unsigned long	core_text_size	模块核心可执行代码大小, 只当链接模块时使用
struct mod_arch_specific	arch	依赖于体系结构的字段 (80 × 86 结构中 没有)
int	unsafe	如果模块不能安全卸载, 则将该标志置位
int	license_gpl	如果模块许可证是 GPL 兼容的, 则将该标志置位
struct module_ref [NR_CPUS]	ref	每 CPU 使用计数器
struct list_head	modules_which_use_me	依赖于该模块的模块链表
struct task_struct *	waiter	正卸载模块的进程
void (*)(void)	exit	模块退出方法
Elf_Sym *	symtab	<i>/proc/kallsyms</i> 文件中所列模块 ELF 符号数组指针
unsigned long	num_symtab	<i>/proc/kallsyms</i> 文件中所列模块 ELF 符号数
char *	strtab	<i>/proc/kallsyms</i> 文件中所列模块 ELF 符号的字符串表
struct module_sect_attrs *	sect_attrs	模块分节属性描述符数组指针 (在 <i>sysfs</i> 文件系统中显示)
void *	percpu	特定于 CPU 的内存区指针
char *	args	链接模块时使用的命令行参数

state 字段记录模块内部状态，它可以是：MODULE_STATE_LIVE（模块为活动的）、MODULE_STATE_COMING（模块正在初始化）和MODULE_STATE_GOING（模块正在卸载）。

正如我们在第十章的“动态地址检查：修正代码”一节中已经介绍的那样，每个模块都有自己的异常表。该表包括（如果有）模块的修正代码的地址。在链接模块时，该表被拷贝到 RAM 中，其开始地址保存在 module 对象的 extable 字段中。

模块使用计数器

每个模块都有一组使用计数器，每个 CPU 一个，存放在相应 module 对象的 ref 字段中。在模块功能所涉及的操作开始执行时递增这个计数器，在操作结束时递减这个计数器。只有所有使用计数器的和为 0 时，模块才可以被取消链接。

例如，假设 MS-DOS 文件系统层作为模块被编译，而且这个模块已经在运行时被链接。最开始时，该模块的引用计数器是 0。如果用户装载一张 MS-DOS 软盘，那么模块引用计数器其中的一个就被递增 1。反之，当用户卸载这张软盘时，计数器其中之一就被减 1（甚至不是刚才递增的那一个）。模块的总的引用计数器就是所有 CPU 计数器的总和。

导出符号

当链接一个模块时，必须用合适的地址替换在模块对象代码中引用的所有全局内核符号（变量和函数）。这个操作与在用户态编译程序时链接程序所执行的操作非常类似（参见第二十章的“库”一节），这是委托给 *insmod* 外部程序完成的（将在后面的“模块的链接和取消”一节进行介绍）。

内核使用一些专门的内核符号表 (*kernel symbol table*)，用于保存模块访问的符号和相应的地址。它们在内核代码段中分三个节：_kstrtab 节（保存符号名）、_ksymtab 节（所有模块可使用的符号地址）和 _ksymtab_gpl 节（GPL 兼容许可证下发布的模块可以使用的符号地址）。当用于静态链接内核代码内时，EXPORT_SYMBOL 与 EXPORT_SYMBOL_GPL 宏让 C 编译器分别往 _ksymtab 和 _ksymtab_gpl 部分相应地加入一个专用符号。

只有某一现有的模块实际使用的内核符号才会保存在这个表中。如果系统程序员在某些模块中需要访问一个尚未导出的内核符号，那么他只要在 Linux 源代码中增加相应的 EXPORT_SYMBOL_GPL 宏就可以了。当然，如果许可证不是 GPL 兼容的，他就不能为模块合法导出一个新符号。

已链接的模块也可以导出自己的符号，这样其他模块就可以访问这些符号。模块符号部分表 (*module symbol table*) 保存在模块代码段的 _ksymtab、_ksymtab_gpl 和 _kstrtab

部分中。要从模块中导出符号的一个子集，程序员可以使用上面描述的 `EXPORT_SYMBOL` 和 `EXPORT_SYMBOL_GPL` 宏。当模块链接时，模块的导出符号被拷贝到两个内存数组中，而其地址保存在 `module` 对象的 `syms` 和 `gpl_syms` 字段中。

模块依赖

一个模块 (B) 可以引用由另一个模块 (A) 所导出的符号；在这种情况下，我们就说 B 装载在 A 的上面，或者说 A 被 B 使用。为了链接模块 B，必须首先链接模块 A，否则，对于模块 A 所导出的那些符号的引用就不能适当地链接到 B 中。简而言之，在这两个模块之间存在着依赖 (*dependency*)。

A 模块对象的 `modules_which_use_me` 字段是一个依赖链表的头部，该链表保存 A 使用的所有模块。链表中的每个元素是一个小型 `module_use` 描述符，该描述符保存指向链表中相邻元素的指针及一个指向相应模块对象的指针。在本例中，指向 B 模块对象的 `module_use` 描述符将出现在 A 的 `modules_which_use_me` 链表中。只要有模块装载在 A 上，`modules_which_use_me` 链表就必须动态更新。如果 A 的依赖链表非空，模块 A 就不能卸载。

当然，除了 A 和 B 之外，还会有其他模块 (C) 装载到 B 上，依此类推。模块的堆叠是对内核源代码进行模块化的一种有效方法，目的是为了加速内核的开发。

模块的链接和取消

用户可以通过执行 `insmod` 外部程序把一个模块链接到正在运行的内核中。该程序执行以下操作：

1. 从命令行中读取要链接的模块名。
2. 确定模块对象代码所在的文件在系统目录树中的位置。对应的文件通常都是在 `/lib/modules` 的某个子目录中。
3. 从磁盘读入存有模块目标代码的文件。
4. 调用 `init_module()` 系统调用，传入参数：存有模块目标代码的用户态缓冲区地址、目标代码长度和存有 `insmod` 程序所需参数的用户态内存区。
5. 结束。

`sys_init_module()` 服务例程是实际执行者，主要操作步骤如下：

1. 检查是否允许用户链接模块（当前进程必须具有 `CAP_SYS_MODULE` 权能）。只要给内核增加功能，而它可以访问系统中的所有数据和进程，安全就是至关重要的。

2. 为模块目标代码分配一个临时内存区,然后拷入作为系统调用第一个参数的用户态缓冲区数据。
3. 验证内存区中的数据是否有效表示模块的 ELF 对象,如果不能,则返回错误码。
4. 为传给 *insmod* 程序的参数分配一个内存区,并存入用户态缓冲区的数据,该缓冲区地址是系统调用传入的第三个参数。
5. 查找 *modules* 链表,以验证模块未被链接。通过比较模块名 (module 对象的 *name* 字段) 进行这一检查。
6. 为模块核心可执行代码分配一个内存区,并存入模块相应节的内容。
7. 为模块初始化代码分配一个内存区,并存入模块相应节的内容。
8. 为新模块确定模块对象地址,对象映像保存在模块 ELF 文件的正文段 *gnu.linkonce.this_module* 一节,而模块对象保存在第 6 步中的内存区。
9. 将第 6 和 7 步中分配的内存区地址存入模块对象的 *module_code* 和 *module_init* 字段。
10. 初始化模块对象的 *modules_which_use_me* 链表。当前执行 CPU 的计数器设为 1,而其余所有的模块引用计数器设为 0。
11. 根据模块对象许可证类型设定模块对象的 *license_gpl* 标志。
12. 使用内核符号表与模块符号表,重置模块目标码。这意味着用相应的逻辑地址偏移量替换所有外部与全局符号的实例值。
13. 初始化模块对象的 *syms* 和 *gpl_syms* 字段,使其指向模块导出的内存中符号表。
14. 模块异常表 (参见第十章“异常表”一节) 保存在模块 ELF 文件的 *__ex_table* 一节,因此它在第 6 步中已拷入内存区,将其地址存入模块对象的 *extable* 字段。
15. 解析 *insmod* 程序的参数,并相应地设定模块变量的值。
16. 注册模块对象 *mkobj* 字段中的 *kobject* 对象,这样在 *sysfs* 特殊文件系统的 *module* 目录中就有一个新的子目录 (参见第十三章“*kobject*”一节)。
17. 释放第 2 步中分配的临时内存区。
18. 将模块对象追加到 *modules* 链表。
19. 将模块状态设为 *MODULE_STATE_COMING*。
20. 如果模块对象的 *init* 方法已定义,则执行它。
21. 将模块状态设为 *MODULE_STATE_LIVE*。
22. 结束并返回 0 (成功)。

为了取消模块的链接,用户需要调用 *rmmod* 外部程序,该程序执行以下操作:

1. 从命令行中读取要取消的模块的名字。
2. 打开 `/proc/modules` 文件，其中列出了所有链接到内核的模块，检查待取消模块是否有效链接。
3. 调用 `delete_module()` 系统调用，向其传递要卸载的模块名。
4. 结束。

相应的 `sys_delete_module()` 服务例程执行以下操作：

1. 检查是否允许用户取消模块链接（当前进程必须具有 `CAP_SYS_MODULE` 权能）。
2. 将模块名存入内核缓冲区。
3. 从 `modules` 链表查找模块的 `module` 对象。
4. 检查模块的 `modules_which_use_me` 依赖链表，如果非空就返回一个错误码。
5. 检查模块状态，如果不是 `MODULE_STATE_LIVE`，就返回错误码。
6. 如果模块有自定义 `init` 方法，函数就要检查是否有自定义 `exit` 方法。如果没有自定义 `exit` 方法，模块就不能卸载，那么返回一个退出码。
7. 为了避免竞争条件，除了运行 `sys_delete_module()` 服务例程的 CPU 外，暂停系统中所有 CPU 的运行。
8. 把模块状态设为 `MODULE_STATE_GOING`。
9. 如果所有模块引用计数器的累加值大于 0，就返回错误码。
10. 如果已定义模块的 `exit` 方法，则执行它。
11. 从 `modules` 链表删除模块对象，并且从 `sysfs` 特殊文件系统注销该模块。
12. 从刚才使用的模块依赖链表中删除模块对象。
13. 释放相应内存区，其中存有模块可执行代码、`module` 对象及有关符号和异常表。
14. 返回 0（成功）。

根据需要链接模块

模块可以在系统需要其所提供的功能时自动进行链接，之后也可以自动删除。

例如，假设 MS-DOS 文件系统既没有被静态链接，也没有被动态链接。如果用户试图装载 MS-DOS 文件系统，那么 `mount()` 系统调用通常就会失败，返回一个错误码，因为 MS-DOS 没有被包含在已注册文件系统的 `file_systems` 链表中。然而，如果内核已配

置为支持模块的动态链接，那么Linux就试图链接MS-DOS模块，然后再扫描已经注册过的文件系统的列表。如果该模块成功地被链接，那么mount()系统调用就可以继续执行，就好像MS-DOS文件系统从一开始就存在一样。

modprobe 程序

为了自动链接模块，内核要创建一个内核线程来执行modprobe外部程序（注2），该程序要考虑由于模块依赖所引起的所有可能因素。模块依赖在前面已介绍过：一个模块可能需要一个或者多个其他模块，这些模块又可能需要其他模块。例如，MS-DOS模块需要另外一个名为fat的模块，该模块包含基于文件分配表（File Allocation Table, FAT）的所有文件系统所通用的一些代码。因此，如果fat模块还不在于系统中，那么在系统请求MS-DOS模块时，fat模块也必须被动态链接到运行的内核中。对模块依赖进行解析以及对模块进行查找的操作最好都在用户态中实现，因为这需要查找和访问文件系统中的模块对象文件。

modprobe外部程序和insmod类似，因为它链接在命令行中指定的一个模块。然而，modprobe还可以递归地链接命令行中模块所使用的所有模块。例如，如果用户调用modprobe来链接MS-DOS模块，那么在需要的时候，modprobe就会在MS-DOS模块之后链接fat模块。实际上，modprobe只是检查模块依赖关系，每个模块的实际的链接工作是通过创建一个进程并执行insmod命令来实现的。

modprobe又是如何知道模块间的依赖关系的呢？另外一个称为depmod的外部命令在系统启动时被执行。该程序查找为正在运行的内核而编译的所有模块，这些模块通常存放在/lib/modules目录下。然后它就把所有的模块间依赖关系写入一个名为modules.dep的文件。这样，modprobe就可以对该文件中存放的信息和/proc/modules文件产生的链接模块链表进行比较。

request_module()函数

在某些情况下，内核可以调用request_module()函数来试图自动链接一个模块。

再次考虑用户试图装载MS-DOS文件系统的情况。如果get_fs_type()函数发现这个文件系统还没有注册，就调用request_module()函数，希望MS-DOS已经被编译为一个模块。

注2： 这是内核依赖于外部程序的少数例子之一。

如果 `request_module()` 函数成功地链接所请求的模块，`get_fs_type()` 就可以继续执行，仿佛这个模块一直都存在一样。当然，并非所有的情况都是如此；在我们的例子中，MS-DOS 模块可能根本就没有被编译。在这种情况下，`get_fs_type()` 返回一个错误码。

`request_module()` 函数接收要链接的模块名作为参数。该函数调用 `kernel_thread()` 来创建一个新的内核线程并等待，直到这个内核线程结束为止。

而此内核线程又接收待链接的模块名作为参数，并调用 `execve()` 系统调用以执行 `modprobe` 外部程序（注 3），向其传递模块名。然后，`modprobe` 程序真正地链接所请求的模块以及这个模块所依赖的任何模块。

注 3：由 `exec_modprobe()` 执行的程序名和路径名可以通过向 `/proc/sys/kernel/modprobe` 文件写入而自定义。