

深入Linux

设备驱动程序内核机制

Internals of Linux Device Driver

◎ 陈学松 著



深入Linux

设备驱动程序内核机制

Internals of Linux Device Driver

© 陈学松 著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

这是一本系统阐述 Linux 设备驱动程序技术内幕的专业书籍，它的侧重点不是讨论如何在 Linux 系统下编写设备驱动程序，而是要告诉读者隐藏在这些设备驱动程序背后的那些内核机制及原理。作者通过对 Linux 内核源码抽丝剥茧般的解读，再辅之以精心设计的大量图片，使读者在阅读完本书后对驱动程序前台所展现出来的那些行为特点变得豁然开朗。

本书涵盖了编写设备驱动程序所需要的几乎所有的内核设施，比如内核模块、中断处理、互斥与同步、内存分配、延迟操作、时间管理，以及新设备驱动模型等内容。为了避免读者迷失在某一技术细节的讨论当中，本书在一个比较高的层面上进行展开，以一种先框架再细节的结构安排极大地简化了读者的阅读与学习。

本书不仅适合那些在 Linux 系统下从事设备驱动程序开发的专业技术人员阅读，也同样适合有志于从事 Linux 设备驱动程序开发或对 Linux 设备驱动程序及 Linux 内核感兴趣的在校学生等阅读。对于没有任何 Linux 设备驱动程序开发经验的初学者，建议先阅读那些讨论“如何”在 Linux 系统下编写设备驱动程序的入门书籍，然后再阅读本书来理解“为什么”要以这样或者那样的方式来编写设备驱动程序。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

深入 Linux 设备驱动程序内核机制 / 陈学松著. —北京：电子工业出版社，2012.1
ISBN 978-7-121-15052-4

I. ①深… II. ①陈… III. ①Linux 操作系统—程序设计 IV. ①TP316.89

中国版本图书馆 CIP 数据核字（2011）第 231765 号

策划编辑：张春雨

责任编辑：白 涛

印 刷：

装 订：三河市鑫金马印装有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：33.75 字数：856 千字

印 次：2012 年 1 月第 1 次印刷

印 数：3000 册 定价：98.00 元（含 DVD 光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

推 荐 序

这不是一本单纯的关于 Linux 设备驱动程序入门的书。它是给有一定的 Linux 设备驱动程序编写经验并且对众多 Linux 底层设备驱动内幕机制感兴趣的读者量身定制的。与市面上已经出版的 Linux 相关方面的图书的不同之处在于，本书并不着重于全面描述 Linux 内核，也不只是简单地告诉你如何去写一个 Linux 下的设备驱动程序。它是从设备驱动程序的视角出发，深入到 Linux 内核去剖析那些和驱动程序实现机制密切相关的技术内幕。比如让你理解为什么在这个地方驱动程序应该使用 `work queue` 而不是 `tasklet`，为什么在中断处理例程里应该使用 `spin_lock` 而不是 `mutex_lock`……因为只有当你对驱动程序中使用的各种内核实现有了清晰的认识，你才能在日常的工作当中随心所欲地驾驭它们，写出更高性能更安全的代码。知其然，更知其所以然，对于沉迷于技术领域的人而言，这种不断探索的好奇心是对技术工作能长期保持热情的一个基本特质。相对于市面上已经出版的相关书籍而言，本书具有以下两个鲜明的特色：

细节揭秘

目前市场上已经出版的 Linux 内核和驱动程序方面的书籍，大体上可分为两种。一种是侧重于内核本身，鉴于目前 Linux 的内核源码已经十分庞大，这些讲解内核的书有些本身非常全面，作者的写作态度也非常严谨，比如 *Deep Understanding Linux Kernel*，还有新近出版的 *Professional Linux Kernel Architecture*，后者几乎涵盖了新版 Linux 内核中绝大部分重要的构件，但也正因如此，这样的书籍就不可能在与驱动程序相关的机制上留下太多笔墨。另外还有一种是专门讲解 Linux 驱动方面的书籍，典型的有 *Linux Device Driver* 和 *Essential Linux Device Driver*。这些书着重于介绍 Linux 驱动的基本概念和架构，但是对于想了解更多幕后的技术细节的读者来说，《深入 Linux 设备驱动程序内核机制》一书可提供更详细的资源和帮助。通常当你想深入理解一些一般书籍没有描述的机制时，你可能会采用在线搜索或查看源码的方式，但有时这不仅费时也未必能得到满意的答案。本书提供了另一途径让你更系统、有效地理解这些内核机制。我相信对于广大忙于在校学习、职场深造或课题攻关的读者来说，本书可提供很多有益的帮助。

图片说理

这本书另外一个很大的特点是，作者大量使用其精心设计的图片来帮助你清晰地理解一些复杂的概念、流程和架构。这在中文版原创的图书中是很难能可贵的，相对而言外文书在这方面做得就要好很多。形象直观的图片胜过大量的文字，也能节省读者大量的时间。可以看到，本书的作者在这一方面做了很大的努力去加以完善，在我看来，这是一个非常好的尝试。本书作者当前正在 AMD 上海研发中心从事 Linux 显卡驱动等系统软件方面的研发工作，能在繁忙的工作之余，通过对自己学习和实践经验的总结写下这样一本书，对增进国内读者的 Linux 系统开发能力将起到很大的作用。我相信，如果作者有足够的时间与精力的话，这本书还可以进一步完善，包括在某些技术方面可以有更精细的描述。

AMD 图形软件架构师 PMTS 俞辉

2011 年 8 月 24 日于加拿大

前言

在 Linux 庞大的源码树中，设备驱动程序部分的代码已经占了相当大的比例。现实的工作中，大量的采用 Linux 系统的平台需要设备驱动程序才能把 Linux 的内核真正运行起来，同时通过编写 Linux 设备驱动程序，使得我们经由亲手编写具有特权等级的代码来一探 Linux 内核幕后的秘密成为可能。所以，无论是从日常工作的需要还是只为单纯满足对 Linux 内核机制好奇心的角度来说，学习并掌握 Linux 设备驱动程序的编写都是非常必要的，同时也是一件非常有趣且有意义的事情。

初衷与定位

这本书并不仅仅是单纯地讨论如何在 Linux 系统下编写一个设备驱动程序，因为关于这方面的内容，市面上已经有大量类似的图书可供参考。本书的总体思想是从内核的角度来看设备驱动程序，从设备驱动程序的角度深入到内核中，比如通过对 `spin_lock` 以及 `spin_lock_irq` 等内核源码的分析，来告诉你在什么场合下应该使用 `spin_lock`，什么场合下又应该选择 `spin_lock_irq`。还有，比如我们几乎每天都会在设备驱动程序所代表的内核模块中使用 `MODULE_LICENSE("GPL")` 这样的声明，这个声明是如此地平凡，以至于我们常常忽略它存在的价值。但是在某个夜深人静的夜晚，感觉长夜漫漫无心睡眠时，在你内心深处的某个地方是否会想过，这个声明对一个内核模块而言，它到底意味着什么，如果没有它，加载这样一个模块对系统又会造成什么样的影响，如此等等，读者都可以在阅读本书的过程中找到答案。

很显然，只有当你清楚地理解了一个东西的内在机制，你才能更好地去使用它们，如果不幸在使用过程中出现问题，也才可以快速将其定位并最终予以解决。台湾著名技术作家侯捷曾引林雨堂先生在《朱门》中的一句话，“只用一样东西，不明白它的道理，实在不高明”，来描述他当时写作时的心境，其实这句话也同样适合我用来阐明写作本书的初衷之一。

但是这并不意味着只有 Linux 系统下设备驱动程序的编写老手才适合阅读本书，因为我在本书写作过程中，一般会先给出一个总体的框架，然后在此基础上对 Linux 提供给设备驱动程序使用的每一个常见而重要的内核设施进行细致地分析，同时辅之以验证性质的代码来使得这种略嫌抽象的讨论具体化，以激发读者对技术探索的兴趣。所以即便是入门级的读者，也可以通过阅读本书来加深对 Linux 下编写设备驱动程序的理解。

另外要说的是，读者不应该寄希望于阅读两三本书就可以掌握 Linux 下设备驱动程序编写的精髓，所有的书籍只能在大体上给你一个参考借鉴的作用，真正的理解还要靠读者自己去努力，诚所谓“纸上得来终觉浅，绝知此事要躬行”。

编排与范围

在本书的结构编排上，我努力使各章节独立起来，但是少量的向前或者向后引用还是必不可少的。但是总体上，我将最基本的篇章尽量放到前面，一些加强型或者高级点的话题尽量放到后边。在描述驱动程序内核机制方面，为了避免单纯的代码解释所带来的抽象感，我会使用具体的例子来将所能看到的驱动程序的前台行为和它的幕后机制串联起来，以帮助读者建立起全面立体的设备驱动程序架构蓝图。不过 Linux 对某些特性的支持因为考虑到各种平台和性能等诸多因素，其实现很可能会有多种不同的方法，比如从内核态驱动程序向用户空间导出信息的文件系统方面，就至少有 `proc` 和 `sysfs` 两种形式。因此本书在描述具体的例子时，一定是遵循其中的某种实现，在诸多实现机制的选择上，本书会从实用性和实时性角度出发，采用内核中最新引入或者是最有发展前景的实现，对于某些即将过时的实现机制（因为兼容或者代码维护工作量的关系，一些老的机制可能依然残留在新版的内核代码中），除非出于技术细节的对比或者从增加知识面的角度考虑会有所涉及，否则将不会作为本书的主线。

在代码的引用上，为了突出功能主线部分和削减本书的篇幅，我会删除代码中用来增加调试信息、性能增强及防御性代码这些部分。对于系统体系架构相关的代码，我主要以 x86 与 ARM 平台为主，因为这两者是当前最流行的两种处理器架构。关于本书所参考的 Linux 内核源码的版本，在本书刚开始写作时参考的是 2.6.35 的版本，在写作的中后期，已经将内核版本更新到了 2.6.39，在本书的修订阶段，我已经努力将之前完成的内容更新到了 2.6.39。当然，因为作者时间精力所限，加之 Linux 内核本身就博大精深，内核版本也一直在不断更新变化中，所以书中肯定还会有这样那样潜在的错误，希望读者朋友们能不吝批评指正，以使我们得以共同提高。

创作历程

我有幸自参加工作以来，在 Linux 下从事设备驱动程序相关的开发工作已经有 9 年多的时间，这期间在 Linux 上所接触的平台既有 x86，也有 ARM，甚至包括少量的 PowerPC。在我看来，学习某一操作系统下的设备驱动程序的编写，主要包含两个方面：一个是该操作系统本身对设备驱动程序框架的支持，也可以称之为设备驱动模型，另一个则是对要驱动的硬件的理解。对于后者，设备驱动程序开发者将要面对各种各样的硬件设备，了解它们的最好也最直接的方法当然是这样硬件的 `datasheet`。前者则主要和操作系统息息相关，比如在 Linux 系统下开发设备驱动程序，必然要熟练掌握 Linux 为设备驱动的编写所提供的

各种内核实施及相关的各种数据结构，本书的内容主要就是探讨 Linux 内核为设备驱动程序编写所提供的所有这些设施的幕后技术。

本书最早的写作酝酿大约在 2010 年 10 月份前后，在此之前，或者是出于自己对以往积累的技术总结的需要，或者是出于将自己的一些技术心得与同行分享的目的，总之，我陆陆续续在一些论坛上发表了若干剖析 Linux 设备驱动程序内核机制的帖子，这些帖子最终使我萌发了用一本书来总结自己以往的 Linux 设备驱动程序开发经验的想法。我把最初的大约一章半的稿子发给了电子工业出版社，很快就得到了策划编辑张春雨先生的肯定，接下来也很顺利地通过了选题的论证，这之后就是一段极其漫长且非常辛苦的写作过程。时间是最大的挑战，由于白天需要工作，写作的时间只能是留给夜晚或者周末，在写作最紧张的时刻，经常要写到凌晨 2 点多。除了时间上的困难之外，如何将一个技术点用最透彻最简洁的语言描述清楚，如何对 Linux 内核中纷繁复杂的内容进行取舍，这些也都是非常耗费精力的事情。技术本身的理解也许并不困难，难在如何去把你心中掌握的东西清晰准确地以文字的方式表达出来，这不同于论坛的发帖，可以非常自由甚至随心所欲，写书的话，必须考虑它的完整性、逻辑性以及可读性，同时还要考虑将来潜在的读者群。尤其是如果你想认真真写一本书的话，有时候甚至需要反复推敲一个技术点的表达方式。在写作灵感枯竭的时候，看着时间飞快掠过，而眼前的文档却没有留下几行字，那种强烈的挫折感与沮丧感真得会让人动摇自己的信念：自己是否还能坚持下去？！所以当这本书即将出版时，我还很有些恍惚，不敢相信自己居然磕磕绊绊地最终完成了这些书稿。

意见反馈

读者如果在阅读本书的过程中有任何意见或者建议，欢迎通过下面的 E-mail 与我取得联系：
ricard_chen@yahoo.com。

关于本书使用到的源代码，读者可在 www.embexperts.com 网站上下载。另外，关于本书后续的一些勘误、某些技术细节方面的讨论也会在该网站相应的版面上进行。

致谢

首先，我要感谢我的家人，如前所述，写书占去了我大量的业余时间，我的父母和怀孕的妻子在此期间承担了几几乎所有的家务劳动，替我捣腾出不少的写作时间，感谢她们！我的宝贝女儿在今年 8 月 15 日健康出生，成为我的家庭中新一员，这本书也正好可以作为父亲的见面礼送给她——可爱的萌萌同学。

其次是电子工业出版社的张春雨与白涛编辑，从选题的论证到文字编辑，他们都付出了极其辛苦的劳动并且提出了很多有益的建议，那些逝去的不堪回首岁月里满眼尘封的 E-mail 见证了这一点！当然，还要感谢我现在所效力的 AMD 公司，因为它使得我不必为生活所

迫去写一本书，对技术的热情与兴趣才是我最终得以坚持下来的最大因素。

最后，在本书的审核方面，AMD 的 PMTS 及显卡驱动软件架构师俞辉在百忙中为本书作序并审核了部分章节，AMD 上海研发中心 Linux Graphic Base Driver 团队的 Lisa Wu 及研发经理刘刚也为本书的写作提供了支持，诺基亚与西门子的研发经理胡兵全审核了本书第 1 章及第 12 章，EMC 的 PE Thomas 审核了本书第 3 章及第 4 章。Marvell 的资深软件工程师 James Lai 亦审核了本书部分章节并有宝贵意见，在此一并表示感谢！

陈学松

2011 年 8 月 29 日于上海



目 录

第 1 章	内核模块	1
1.1	内核模块的文件格式	2
1.2	EXPORT_SYMBOL 的内核实现	5
1.3	模块的加载过程	8
1.3.1	sys_init_module (第一部分)	9
1.3.2	struct module	9
1.3.3	load_module	13
1.3.4	sys_init_module (第二部分)	49
1.3.5	模块的卸载	54
1.4	本章小结	55
第 2 章	字符设备驱动程序	57
2.1	应用程序与设备驱动程序互动实例	58
2.2	struct file_operations	62
2.3	字符设备的内核抽象	63
2.4	设备号的构成与分配	65
2.4.1	设备号的构成	65
2.4.2	设备号的分配与管理	66
2.5	字符设备的注册	71
2.6	设备文件节点的生成	74
2.7	字符设备文件的打开操作	77
2.8	本章小结	85
第 3 章	分配内存	87
3.1	物理内存的管理	87
3.1.1	内存节点 node	87
3.1.2	内存区域 zone	88
3.1.3	内存页	89
3.2	页面分配器 (page allocator)	90

3.2.1	gfp_mask	91
3.2.2	alloc_pages	95
3.2.3	__get_free_pages	96
3.2.4	get_zeroed_page	97
3.2.5	__get_dma_pages	97
3.3	slab 分配器 (slab allocator)	98
3.3.1	管理 slab 的数据结构	99
3.3.2	kmalloc 与 kzalloc	105
3.3.3	kmem_cache_create 与 kmem_cache_alloc	108
3.4	内存池 (mempool)	110
3.5	虚拟内存的管理	111
3.5.1	内核虚拟地址空间构成	111
3.5.2	vmalloc 与 vfree	112
3.5.3	ioremap	115
3.6	per-CPU 变量	115
3.6.1	静态 per-CPU 变量的声明与定义	116
3.6.2	静态 per-CPU 变量的链接脚本	117
3.6.3	setup_per_cpu_areas 函数	118
3.6.4	使用 per-CPU 变量	121
3.7	本章小结	125
第 4 章	互斥与同步	127
4.1	并发的来源	127
4.2	local_irq_enable 与 local_irq_disable	128
4.3	自旋锁	129
4.3.1	spin_lock	130
4.3.2	spin_lock 的变体	133
4.3.3	单处理器上的 spin_lock 函数	136
4.3.4	读取者与写入者自旋锁 rwlock	137
4.4	信号量 (semaphore)	141
4.4.1	信号量的定义与初始化	141
4.4.2	DOWN 操作	142
4.4.3	UP 操作	145
4.4.4	读取者与写入者信号量 rwsem	146
4.5	互斥锁 mutex	148
4.5.1	互斥锁的定义与初始化	148
4.5.2	互斥锁的 DOWN 操作	149

4.5.3	互斥锁的 UP 操作	150
4.6	顺序锁 seqlock	152
4.7	RCU	155
4.7.1	读取者的 RCU 临界区	156
4.7.2	写入者的 RCU 操作	156
4.7.3	RCU 使用的特点	157
4.8	原子变量与位操作	159
4.9	等待队列	162
4.9.1	等待队列头 wait_queue_head_t	162
4.9.2	等待队列的节点	163
4.9.3	等待队列的应用	164
4.10	完成接口 completion	164
4.11	本章小结	168
第 5 章	中断处理	169
5.1	中断的硬件框架	169
5.2	PIC 与软件中断号	170
5.3	通用的中断处理函数	171
5.4	do_IRQ 函数	172
5.5	struct irq_chip	178
5.6	struct irqaction	179
5.7	irq_set_handler	180
5.8	handle_irq_event	184
5.9	request_irq	186
5.10	中断处理的 irq_thread 机制	190
5.11	free_irq	191
5.12	SOFTIRQ	192
5.13	irq 的自动探测	196
5.14	中断处理例程	200
5.15	中断共享	201
5.16	本章小结	202
第 6 章	延迟操作	203
6.1	tasklet	203
6.1.1	tasklet 机制初始化	204
6.1.2	提交一个 tasklet	205
6.1.3	tasklet_action	209

6.1.4	tasklet 的其他操作	212
6.2	工作队列 work queue	214
6.2.1	数据结构	214
6.2.2	create_singlethread_workqueue 和 create_workqueue	216
6.2.3	工人线程 worker_thread	219
6.2.4	destroy_workqueue	221
6.2.5	提交工作节点 queue_work	224
6.2.6	内核创建的工作队列	229
6.3	本章小结	230
第 7 章	设备文件的高级操作	231
7.1	ioctl 文件操作	231
7.1.1	ioctl 的系统调用	231
7.1.2	ioctl 的命令编码	235
7.1.3	copy_from_user 和 copy_to_user	238
7.2	字符设备的 I/O 模型	243
7.3	同步阻塞型 I/O	244
7.3.1	wait_event_interruptible	244
7.3.2	wake_up_interruptible	246
7.4	同步非阻塞型 I/O	250
7.5	异步阻塞型 I/O	251
7.6	异步非阻塞型 I/O	258
7.7	驱动程序的 fsync 例程	259
7.8	fasync 例程	260
7.9	llseek 例程	269
7.10	访问权能	272
7.11	本章小结	273
第 8 章	时间管理	274
8.1	jiffies	274
8.1.1	时间比较	277
8.1.2	时间转换	278
8.2	延时操作	279
8.2.1	长延时	280
8.2.2	短延时	285
8.3	内核定时器	286
8.3.1	init_timer	289

8.3.2	add_timer	289
8.3.3	del_timer 和 del_timer_sync	293
8.4	本章小结	293
第 9 章	Linux 设备驱动模型	295
9.1	sysfs 文件系统	295
9.2	kobject 和 kset	298
9.2.1	kobject	298
9.2.2	kobject 的类型属性	305
9.2.3	kset	308
9.2.4	热插拔中的 uevent 和 call_usermodehelper	311
9.2.5	实例源码	320
9.3	总线、设备与驱动	328
9.3.1	总线及其注册	328
9.3.2	总线的属性	335
9.3.3	设备与驱动的绑定	338
9.3.4	设备	339
9.3.5	驱动	348
9.4	class	351
9.5	本章小结	355
第 10 章	内存映射与 DMA	356
10.1	设备缓存与设备内存	356
10.2	mmap	356
10.2.1	struct vm_area_struct	357
10.2.2	用户空间虚拟地址布局	358
10.2.3	mmap 系统调用过程	362
10.2.4	驱动程序中 mmap 方法的实现	368
10.2.5	mmap 使用范例	373
10.2.6	munmap	383
10.3	DMA	384
10.3.1	内核中的 DMA 层	384
10.3.2	物理地址与总线地址	386
10.3.3	dma_set_mask	387
10.3.4	DMA 映射	388
10.3.5	回弹缓冲区 (bounce buffer)	401
10.3.6	DMA 池	401

10.4	本章小结	405
第 11 章	块设备驱动程序	407
11.1	块子系统初始化	408
11.2	ramdisk 源码实例	410
11.2.1	make_request 版本的 RAM DISK 源码	411
11.2.2	request 版本的 RAM DISK 源码	416
11.2.3	ramdisk 的使用	420
11.3	块设备号的注册与管理	422
11.4	block_device	424
11.5	struct gendisk	425
11.6	struct hd_struct	428
11.7	用 alloc_disk 分配 gendisk 对象	428
11.8	向系统添加一个块设备 add_disk	430
11.9	block_device_operations	439
11.10	块设备文件的打开	440
11.11	blk_init_queue	448
11.12	blk_queue_make_request	459
11.13	向队列提交请求	460
11.14	块设备的请求处理函数	466
11.15	bio 结构	467
11.16	本章小结	472
第 12 章	网络设备驱动程序	473
12.1	net_device	475
12.2	网络设备的注册	488
12.3	设备方法	492
12.3.1	设备初始化	494
12.3.2	设备接口的打开与停止	495
12.3.3	数据包的发送	495
12.3.4	网络数据包发送过程中的流控机制	500
12.3.5	传输超时 (watchdog timeout)	503
12.3.6	数据包的接收	506
12.4	套接字缓冲区	510
12.5	中断处理	518
12.6	NAPI	520
12.7	本章小结	522

第 1 章

内核模块

模块最大的好处是可以动态扩展应用程序的功能而无须重新编译链接生成一个新的应用程序映像，这种广义上的模块概念其实并非 Linux 系统所特有，在微软的 Windows 系统上动态链接库 DLL（Dynamic Link Library）便是模块概念的一个典型应用场景，对应到 Linux 系统上这种模块以所谓的共享库 so（shared object）文件的形式存在¹。

本章要讨论的主题——Linux 内核模块，在概念及原理方面与上面提到的 DLL 和 so 模块类似，但又有其独特的一面，内核模块可以在系统运行期间动态扩展系统功能而无须重新启动系统²，更无须为这些新增的功能重新编译一个新的系统内核映像。内核模块的这个特性为内核开发者开发验证新的功能提供了极大的便利，因为像 Linux 这么庞大的系统，编译一个新内核并重新启动将浪费开发者大量的时间。

虽然设备驱动程序并不一定要以内核模块的形式存在，并且内核模块也不一定就代表着一个设备驱动程序，但是内核模块的这种特性似乎注定是为设备驱动程序而生。Linux 系统下的设备驱动程序在开发一个新的设备驱动的过程中，使用的最多的工具之一是 insmod，这就是一个简单的向系统动态加载内核模块的命令。很难想象，如果没有 insmod 这样的机制，在 Linux 底下调试一个设备驱动会是怎样的一件让人痛苦抓狂的事情！笔者相信，任何一个在 Linux 上面有过实际的驱动程序开发经历的人都会有类似的感受。

Linux 系统虽然为内核模块机制提供了完善的支持，使得其下的内核模块是如此强大，然而现实中事情往往并非如预想的那样一帆风顺，如果对其幕后的机制不甚了解，在实际的开发过程之中，除了驱动程序自身要实现的功能可能会遇到麻烦以外，在利用 Linux 中的内核模块机制时，也会遇到各种各样的问题，比如在用 insmod 命令加载一个模块时，就很可能碰到类似下面的错误信息：

```
root@AMDLinuxFGL:~# insmod demodev.ko
```

¹ 在软件工程中，模块这一术语在不同的上下文环境中有不同的语义。本书中提到的模块特指某种动态或者静态链接库。因为静态库在原理上和动态库有很大的区别，所以本章提到的模块，背后都暗含着动态链接的思想，更具体地，就是以 .ko 形式存在的所谓内核模块。

² 如果因为动态加载的模块自身的原因导致系统崩溃，则是另一回事了。


```
insmod: error inserting 'demodev.ko': -1 Invalid module format
```

如果 `dmesg` 一下，就会看到内核针对上述错误打印出的出错信息如下所示：

```
demodev: version magic '2.6.39 SMP mod_unload 586' should be '2.6.39 SMP mod_unload
modversions 586'
```

直觉上，这应该不是在驱动程序自身要实现的功能上出现了问题，问题应该出在驱动程序所在的模块在加载时与系统中内核模块框架互动的环节中。很明显，Linux 内核设计中为模块这种机制提供了完善的支持，以内核模块形式存在的设备驱动程序也必然要遵循这种框架下的规则才能正常工作，也许绝大多数情况下模块都会工作得很好，然而诸如上面提到的这类模块相关的错误也绝非罕见。

既然规则不由我们定义，那么了解并遵守规则就成了避免或者解决这类问题的唯一途径。一个成熟的 Linux 设备驱动程序开发者应该能很快确定这些错误的原因并给出相应的解决方案，而新手在这类错误面前更多的感觉则可能是迷惘和不知所措。因此，无论是出于现实工作的需要，还是为了满足自己的好奇心，Linux 下的设备驱动程序员都有必要花上足够多的时间来了解隐藏在内核模块背后的技术细节，而这也正是本章要深入探讨内核模块机制的目的。本章将重点关注并讨论如下的问题：

- 模块的加载过程。
- 模块如何引用内核或者其他模块中的函数与变量。
- 模块本身导出的函数与变量如何被别的内核模块所使用。
- 模块的参数传递机制。
- 模块之间的依赖关系。
- 模块中的版本控制机制。

1.1 内核模块的文件格式

以内核模块形式存在的驱动程序，比如 `demodev.ko`，其在文件的数据组织形式上是 ELF（Executable and Linkable Format）格式，更具体地，内核模块是一种普通的可重定位目标文件。用 `file` 命令查看 `demodev.ko` 文件，可以得到类似如下的输出：

```
dennis@AMDLinuxFGL:/$ file demodev.ko
demodev.ko: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

ELF 是 Linux 下非常重要的一种文件格式，常见的可执行程序都是以 ELF 的形式存在。本

书不会详细讨论 ELF 格式的技术细节，但是为了让读者能更好地理解后续的模块加载、导出符号和模块参数等相关主题，在这里我们结合 Linux 源代码中定义的 ELF 相关数据结构（基于 32 位体系架构），给出 ELF 格式的一个比较详细的结构图，如图 1-1 所示（这张图在后续的小节中会被多次引用）：

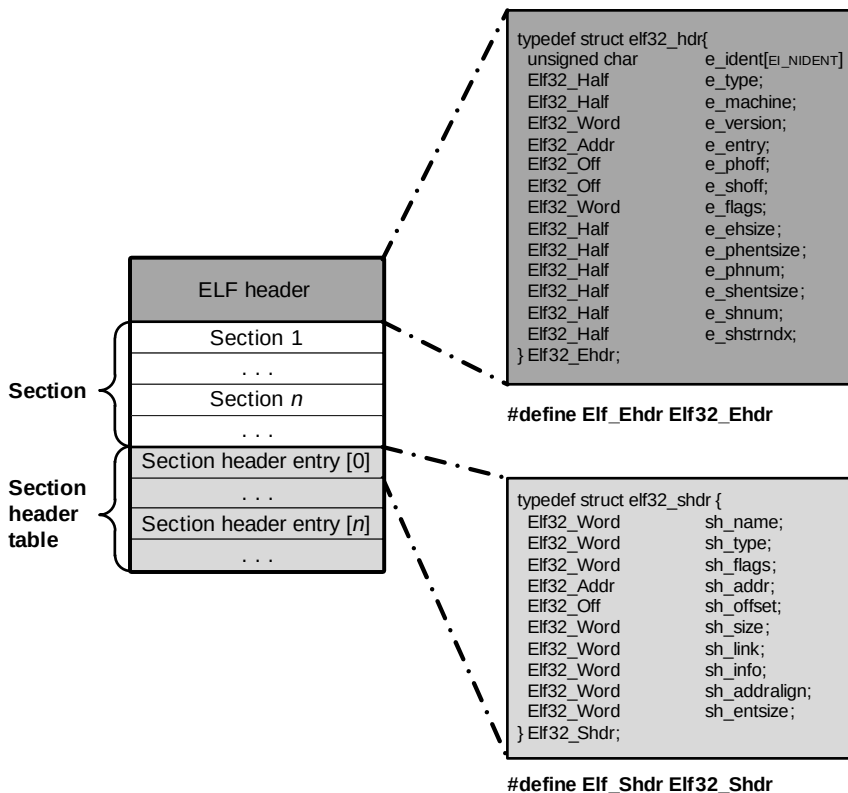


图 1-1 ELF 文件视图格式

图 1-1 中忽略了驱动程序模块 ELF 文件中不会用到的 Program header table。从图 1-1 可以看到，静态的 ELF 文件视图³总体上可分为三大部分：头部的 ELF header，中间的 Section 和尾部的 Section header table。

○ ELF header 部分

大小是 52 字节，位于文件头部。对于驱动模块文件而言，其中一些比较重要的数据成员如下：

e_type

³ 在说 ELF 文件视图时，它总是静态的。加上“静态的”是为了强调和“动态的”执行期 ELF 内存视图的对比。当然，ELF 文件被加载时总是会先被放到某段内存区域中，此时称为“静态的”内存视图；而在后续的加载处理过程中，其在内存中搬移的视图则称为“动态的”内存视图，简称内存视图。

表明文件类型，对于驱动模块，这个值是 1，也就是说驱动模块是一个可定位的 ELF 文件（relocatable file）。

`e_shoff`

表明 Section header table 部分在文件中的偏移量。

`e_shentsize`

表明 Section header table 部分中每一个 entry⁴的大小（以字节计）。

`e_shnum`

表明 Section header table 中有多少个 entry。因此，Section header table 的大小便为 `e_shentsize×e_shnum` 个字节。

`e_shstrndx`

与 Section header entry 中的 `sh_name` 一起用来指明对应的 section 的 name。

○ Section 部分

ELF 文件的主体，位于文件视图中间部分的一个连续区域中。但是当模块被内核加载时，会根据各自属性被重新分配到新的内存区域（有些 section 也可能只是起辅助作用，因而在运行时并不占用实际的内存空间）。

○ Section header table 部分

该部分位于文件视图的末尾，由若干个（具体个数由 ELF header 中的 `e_shnum` 变量指定）Section header entry 组成，每个 entry 具有同样的数据结构类型。对于设备驱动模块而言，一些比较重要的数据成员如下：

`sh_addr`

这个值用来表示该 entry 所对应的 section 在内存中的实际地址。在静态的文件视图中，这个值为 0，当模块被内核加载时，加载器会用该 section 在内存中的实际地址来改写 `sh_addr`（如果 section 不占用内存空间，该值为 0）。

`sh_offset`

表明对应的 section 在文件视图中的偏移量。

⁴ 在本章中，entry 具有特别的含义，特指 Section header table 中的一个 entry。在本章接下来的代码表示中会多次用到这一用法：entry[i]表示 Section header table 中索引值为 i 的 entry。

sh_size

表明对应的 section 在文件视图中的大小（以字节计）。类型为 SHT_NOBITS 的 section 例外，这种 section 在文件视图中不占有空间。

sh_entsize

主要用于由固定数量 entry 组成的表所构成的 section，如符号表，此种情况下用来表示表中 entry 的大小。

以上简单介绍了内核模块所属 ELF 文件的一些主要数据成员，显然设备驱动程序并不会使用到这些数据，它们是给内核模块加载器在加载模块时使用的，这里只是为了给后续的模块加载过程的讨论做一个简单的技术铺垫（如果读者对 ELF 文件的技术细节感兴趣，这里推荐一个非常实用的在 Linux 环境下读取 ELF 文件信息的工具——readelf）。接下来在进行模块加载这个沉重的话题讨论前，先来看一个有趣的东西：模块是如何向外界导出符号信息的。

1.2 EXPORT_SYMBOL 的内核实现

看过 Linux 内核源码的读者应该知道，源码中充斥着像 EXPORT_SYMBOL 这样的宏，在我们自己的设备驱动程序中也经常会发现它的身影。大部分时间里，我们只知道它用来向外界导出一个符号，仅此而已。我们对这些宏是如此习惯，以至于常常忽略其存在的意义，更不用说去仔细探究其背后的实现原理了。然而这些不起眼的宏却有着大用场，如果没有它们，我们的驱动程序甚至连 printk 这样常见的内核函数都不能使用。

本节描述 EXPORT_SYMBOL、EXPORT_SYMBOL_GPL 和 EXPORT_SYMBOL_GPL_FUTURE 宏定义导出符号的内核机制。之所以把导出符号作为独立的一节，是因为在模块加载的过程中会使用本节描述到的机制，而且导出符号这一特性在 Linux 系统中对模块的存在具有重要意义。读者应该结合本节和模块加载部分的相关内容来理解如何导出符号和使用导出的符号这一重要的内核机制。

如果没有独立存在的内核模块，作为单一的 Linux 内核映像，导出符号就失去了意义。对于静态编译链接而成的内核映像而言，所有的符号引用都将在静态链接阶段完成。然而，内核模块的出现让事情发生了变化：内核模块不可避免地要使用到内核提供的基础设施（以调用内核函数的形式发生），作为独立编译链接的内核模块，必须要解决这种静态链接无法完成的符号引用问题（在内核模块所在的 ELF 文件中，这种引用被称为“未解决的引用”）。处理“未解决引用”问题的本质是在模块加载期间找到当前“未解决的引用”符号在内存中的实际目标地址。

内核和内核模块通过符号表的形式向外部世界导出符号的相关信息，这种导出符号的方式在代码层面以 `EXPORT_SYMBOL` 宏定义的形式存在。从全局来看，`EXPORT_SYMBOL` 这类宏功能的完整实现需要经过三个部分来达成：`EXPORT_SYMBOL` 宏定义部分，链接脚本链接器部分和使用导出符号部分。本节讲述前两个部分，第三部分的描述将延后到模块加载的相关段落。

下面通过这些宏定义来仔细考量代码背后的技术细节。

```
<include/linux/module.h>
-----
#define __EXPORT_SYMBOL(sym, sec) \
    extern typeof(sym) sym; \
    __CRC_SYMBOL(sym, sec) \
    static const char __kstrtab_##sym[] \
        __attribute__((section("__ksymtab_strings"), aligned(1))) \
    = MODULE_SYMBOL_PREFIX #sym; \
    static const struct kernel_symbol __ksymtab_##sym \
        __used \
        __attribute__((section("__ksymtab" sec), unused)) \
    = { (unsigned long)&sym, __kstrtab_##sym }

#define EXPORT_SYMBOL(sym) \
    __EXPORT_SYMBOL(sym, "")

#define EXPORT_SYMBOL_GPL(sym) \
    __EXPORT_SYMBOL(sym, "_gpl")

#define EXPORT_SYMBOL_GPL_FUTURE(sym) \
    __EXPORT_SYMBOL(sym, "_gpl_future")
```

以上为来自 Linux 源码树中的 `EXPORT_SYMBOL` 等相关宏的定义细节。其中的 `__CRC_SYMBOL` 用来作为版本控制信息使用，在本章后续的“模块的版本控制”一节中将予以讨论。在接下来的分析中，为了使读者更清楚其中的实现细节，笔者会对内核中的源码稍作改写，这种改写并不会改变原来代码的本质，而只是为了让读者看起来更加方便。此外，为叙述简单起见，将用 `EXPORT_SYMBOL(my_exp_function)` 作为具体的例子，即向外部导出一个名为 `my_exp_function` 的函数，这个导出函数的例子同样也用在 `EXPORT_SYMBOL_GPL` 和 `EXPORT_SYMBOL_GPL_FUTURE` 中。

从源代码可以看出，每个 `EXPORT_SYMBOL` 宏实际上定义了两个变量：

```
static const char *__kstrtab_my_exp_function = "my_exp_function";
static const struct kernel_symbol __ksymtab_my_exp_function =
{ (unsigned long)& my_exp_function, __kstrtab_my_exp_function };
```

第一个变量是个简单的 `char` 型指针，用来表示导出的符号名称；第二个变量类型是 `struct`

kernel_symbol 数据结构，用来表示一个内核符号的实例，struct kernel_symbol 的定义为：

```
<include/linux/module.h>
-----
struct kernel_symbol
{
    unsigned long value;
    const char *name;
};
```

其中，value 是该符号在内存中的地址，name 是符号名。所以，单由该数据结构可以知道，用 EXPORT_SYMBOL(my_exp_function)来导出符号“my_exp_function”，实际上是要通过 struct kernel_symbol 的一个对象告诉外部世界关于这个符号的两点信息：符号名称和地址⁵。

可见，由 EXPORT_SYMBOL 等宏导出的符号，与一般的变量定义并没有实质性的差异，唯一的不同点在于它们被放在了特定的 section 中。

上面的__kstrtab_my_exp_function 会被放置在一个名为“__ksymtab_strings”的 section 中，__ksymtab_my_exp_function 会放置在一个名为“__ksymtab”的 section 中（对于 EXPORT_SYMBOL_GPL 和 EXPORT_SYMBOL_GPL_FUTURE 而言，其 struct kernel_symbol 实例所在的 section 名称则分别为“__ksymtab_gpl”和“__ksymtab_gpl_future”）。

对这些 section 的使用需要经过一个中间环节，即链接脚本与链接器部分。链接脚本告诉链接器把所有目标文件中的名为“__ksymtab”的 section 放置在最终内核（或者是内核模块）映像文件的名为“__ksymtab”的 section 中（对于目标文件中的名为“__ksymtab_gpl”、“__ksymtab_gpl_future”、“__kcrctab”、“__kcrctab_gpl”和“__kcrctab_gpl_future”的 section 都同样处理），看看下面的这个具体的链接脚本的例子就很清楚了。

```
<arch/x86/kernel/vmlinux.lds>
-----
__ksymtab : AT(ADDR(__ksymtab) - 0xC0000000)
{ __start__ksymtab = .; *(__ksymtab) __stop__ksymtab = .; }

__ksymtab_gpl : AT(ADDR(__ksymtab_gpl) - 0xC0000000)
{ __start__ksymtab_gpl = .; *(__ksymtab_gpl) __stop__ksymtab_gpl = .; }

__ksymtab_gpl_future : AT(ADDR(__ksymtab_gpl_future) - 0xC0000000)
{
    __start__ksymtab_gpl_future = .;
    *(__ksymtab_gpl_future) __stop__ksymtab_gpl_future = .;
}
```

⁵ 对于由内核模块导出的符号而言，由于在静态链接时无法确定该符号在内存中的最终地址，因此这个地址信息要一直等到模块被成功加载进系统后才有效。在模块加载的过程中，由内核模块加载器负责修改该成员以反映出符号在内存中的最终目标地址，这也就是所谓的“重定位”过程。

```

__kcrctab : AT(ADDR(__kcrctab) - 0xC0000000)
{ __start__kcrctab = .; *(__kcrctab) __stop__kcrctab = .; }

__kcrctab_gpl : AT(ADDR(__kcrctab_gpl) - 0xC0000000)
{ __start__kcrctab_gpl = .; *(__kcrctab_gpl) __stop__kcrctab_gpl = .; }

__kcrctab_gpl_future : AT(ADDR(__kcrctab_gpl_future) - 0xC0000000)
{ __start__kcrctab_gpl_future = .; *(__kcrctab_gpl_future) __stop__kcrctab_gpl_future = .; }

__ksymtab_strings : AT(ADDR(__ksymtab_strings) - 0xC0000000)
{ *(__ksymtab_strings) }

```

这里之所以要把所有向外界导出的符号统一放到一个特殊的 section 里面, 是为了在加载其他模块时用来处理那些“未解决的引用”符号, 在稍后的“模块的加载过程”一节中可看到这种用途。注意这里由链接脚本定义的几个变量__start__ksymtab、__stop__ksymtab、__start__ksymtab_gpl、__stop__ksymtab_gpl、__start__ksymtab_gpl_future、__stop__ksymtab_gpl_future, 它们会在对内核或者是某一内核模块的导出符号表进行查找时用到。

内核源码中为使用这些链接器产生的变量作了如下的声明:

```

<kernel/module.c>
-----
extern const struct kernel_symbol __start__ksymtab[];
extern const struct kernel_symbol __stop__ksymtab[];
extern const struct kernel_symbol __start__ksymtab_gpl[];
extern const struct kernel_symbol __stop__ksymtab_gpl[];
extern const struct kernel_symbol __start__ksymtab_gpl_future[];
extern const struct kernel_symbol __stop__ksymtab_gpl_future[];
extern const unsigned long __start__kcrctab[];
extern const unsigned long __start__kcrctab_gpl[];
extern const unsigned long __start__kcrctab_gpl_future[];

```

如此, 内核代码便可以直接使用这些变量而不会引起编译错误。

内核模块的加载器在处理模块中“未解决的引用”的符号时, 会使用到这里定义的这些变量。

1.3 模块的加载过程

在用户空间, 用 insmod 这样的命令来向内核空间安装一个内核模块, 本节将详细讨论模块加载时的内核行为。当调用“insmod demodev.ko”来安装 demodev.ko 这样的内核模块时, insmod 会首先利用文件系统的接口将其数据读取到用户空间的一段内存中, 然后通过系统调用 sys_init_module 让内核去处理模块加载的整个过程。

1.3.1 sys_init_module (第一部分)

sys_init_module 的函数原型为：

```
long sys_init_module(void __user *umod, unsigned long len, const char __user *uargs);
```

其中，第一参数 umod 是指向用户空间 demodev.ko 文件映像数据的内存地址，第二参数 len 是该文件的数据大小，第三参数 uargs 是传给模块的参数在用户空间下的内存地址。

在 sys_init_module 函数中，加载模块的任务主要是通过调用 load_module 函数来完成的，该函数的定义为：

```
<kernel/module.c>
-----
static struct module *load_module(void __user *umod,
                                   unsigned long len,
                                   const char __user *uargs)
```

所有参数同 sys_init_module 函数中的完全一样，实际上在 sys_init_module 函数的一开始便会调用该函数，调用时传入的实参完全来自于 sys_init_module 函数，没有经过任何的处理或者修改。

为了更清楚地解释模块加载时的内核行为，我们把 sys_init_module 分为两个部分：第一部分是调用 load_module，完成模块加载最核心的任务；第二部分是在模块被成功加载到系统之后的后续处理。我们将在讨论完 load_module 部分之后再继续讨论 sys_init_module 的第二部分。不过，在继续 load_module 话题之前，先要看一个内核中非常重要的数据结构——struct module。

1.3.2 struct module

load_module 函数的返回值是一个 struct module 类型的指针，struct module 是内核用来管理系统中加载的模块时使用的一个非常重要的数据结构，一个 struct module 对象代表着现实中一个内核模块在 Linux 系统中的抽象，该结构的定义如下（删除了一些 trace 和 unused symbol 相关的部分）：

```
<include/linux/module.h>
-----
struct module
{
    enum module_state state;

    /* Member of list of modules */
    struct list_head list;

    /* Unique handle for this module */
```



```
char name[MODULE_NAME_LEN];

/* Sysfs stuff. */
struct module_kobject mkobj;
struct module_attribute *modinfo_attrs;
const char *version;
const char *srcversion;
struct kobject *holders_dir;

/* Exported symbols */
const struct kernel_symbol *syms;
const unsigned long *crcs;
unsigned int num_syms;

/* Kernel parameters. */
struct kernel_param *kp;
unsigned int num_kp;

/* GPL-only exported symbols. */
unsigned int num_gpl_syms;
const struct kernel_symbol *gpl_syms;
const unsigned long *gpl_crcs;

/* symbols that will be GPL-only in the near future. */
const struct kernel_symbol *gpl_future_syms;
const unsigned long *gpl_future_crcs;
unsigned int num_gpl_future_syms;

/* Exception table */
unsigned int num_exentries;
struct exception_table_entry *extable;

/* Startup function. */
int (*init)(void);

/* If this is non-NULL, vfree after init() returns */
void *module_init;

/* Here is the actual code + data, vfree'd on unload. */
void *module_core;

/* Here are the sizes of the init and core sections */
unsigned int init_size, core_size;

/* The size of the executable code in each section. */
unsigned int init_text_size, core_text_size;
```

```

/* Size of RO sections of the module (text+rodata) */
unsigned int init_ro_size, core_ro_size;

/* Arch-specific module values */
struct mod_arch_specific arch;

unsigned int taints;      /* same bits as kernel:tainted */

#ifdef CONFIG_KALLSYMS
/*
 * We keep the symbol and string tables for kallsyms.
 * The core_* fields below are temporary, loader-only (they
 * could really be discarded after module init).
 */
Elf_Sym *symtab, *core_symtab;
unsigned int num_symtab, core_num_syms;
char *strtab, *core_strtab;

/* Section attributes */
struct module_sect_attrs *sect_attrs;

/* Notes attributes */
struct module_notes_attrs *notes_attrs;
#endif

#ifdef CONFIG_SMP
/* Per-cpu data. */
void __percpu *percpu;
unsigned int percpu_size;
#endif

/* The command line arguments (may be mangled).  People like
   keeping pointers to this stuff */
char *args;

#ifdef CONFIG_MODULE_UNLOAD
/* What modules depend on me? */
struct list_head source_list;
/* What modules do I depend on? */
struct list_head target_list;

/* Who is waiting for us to be unloaded */
struct task_struct *waiter;

/* Destruction function. */

```

```
void (*exit)(void);

struct module_ref {
    unsigned int incs;
    unsigned int decs;
} __percpu *refptr;
#endif

#ifdef CONFIG_CONSTRUCTORS
    /* Constructor functions. */
    ctor_fn_t *ctors;
    unsigned int num_ctors;
#endif
};
```

我们很快就会在后续模块加载部分看到使用这些成员的具体代码，现在先把一些重要的成员变量简单描述如下：

enum module_state state

用于记录模块加载过程中不同阶段的状态，**module_state** 的定义如下：

```
enum module_state
{
    //模块被成功加载进系统时的状态
    MODULE_STATE_LIVE,
    //模块正在加载中
    MODULE_STATE_COMING,
    //模块正在卸载中
    MODULE_STATE_GOING,
};
```

struct list_head list

用来将模块链接到系统维护的内核模块链表中，内核用一个链表来管理系统中所有被成功加载的模块。

char name[MODULE_NAME_LEN]

模块名称。

const struct kernel_symbol *syms

内核模块导出的符号所在起始地址。

const unsigned long *crcs

内核模块导出符号的校验码所在起始地址。

```
struct kernel_param *kp
```

内核模块参数所在的起始地址。

```
int (*init)(void)
```

指向内核模块初始化函数的指针，在内核模块源码中由 `module_init` 宏指定。

```
struct list_head source_list
```

```
struct list_head target_list
```

用来在内核模块间建立依赖关系。

1.3.3 load_module

作为内核模块加载器中最核心的函数，`load_module` 负责最艰苦的模块加载全过程。我们将仔细讨论该函数，因为除了可以了解内核模块加载的幕后机制之外，还能了解到一些非常有趣的特性，诸如内核模块如何调用内核代码导出的函数，被加载的模块如何向系统中其他的模块导出自己的符号，以及模块如何接收外部的参数等。在介绍这部分内容时，如果完全按照内核代码的顺序依序进行的话，逻辑上可能会显得比较凌乱。所以此处文字组织的基本思路是：将 `load_module` 函数按照各主要功能分成若干部分，各部分在下文中的出现顺序尽可能维持在代码中的出现顺序；如果某些功能之间存在着某种依赖关系，比如有 A 和 B 两个功能，A 功能的叙述需要用到 B 功能中提供的机制，则先介绍 B 功能；独立于功能模块之外的一些基础设施，比如某些功能性函数，则尽量往前放。

○ 模块 ELF 静态的内存视图

如图 1-2 所示，用户空间程序 `insmod` 首先通过文件系统接口读取内核模块 `demodev.ko` 的文件数据，将其放在一块用户空间的存储区域中（图中 `void *umod` 所示）。然后通过系统调用 `sys_init_module` 进入到内核态，同时将 `umod` 指针作为参数传递过去（同时传入的还有 `umod` 所指向的空间大小 `len` 和存放有模块参数的地址空间指针 `uargs`）。

`sys_init_module` 调用 `load_module`，后者将在内核空间利用 `vmalloc` 分配一块大小同样为 `len` 的地址空间，如图 1-2 中 `Elf_Ehdr *hdr` 所示。然后通过 `copy_from_user` 函数的调用将用户空间的文件数据复制到内核空间中，从而在内核空间构造出 `demodev.ko` 的一个 ELF 静态的内存视图。接下来的操作都将以此视图为基础，为使叙述简单起见，我们称该视图为 HDR 视图（图 1-2 下方点画线椭圆部分）。HDR 视图所占用的内存空间在 `load_module` 结束时通过 `vfree` 予以释放。

○ 字符串表（String Table）

字符串表是 ELF 文件中的一个 `section`，用来保存 ELF 文件中各个 `section` 的名称或符号名，

这些名称以字符串的形式存在。图 1-3 给出了一个具体的字符串表实例：

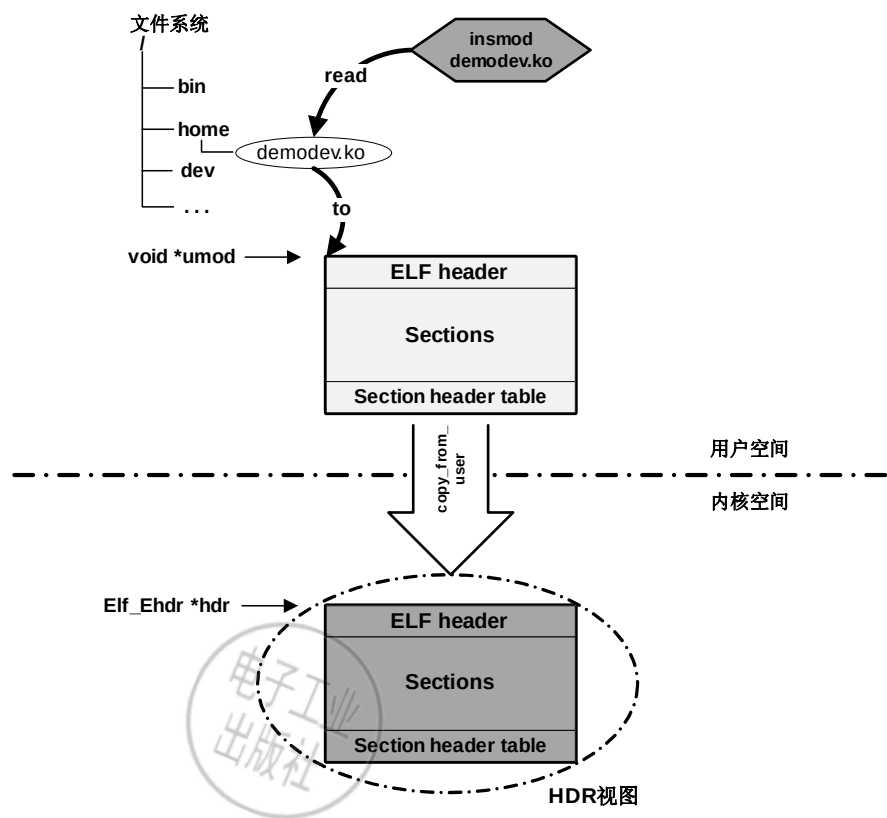


图 1-2 insmod 构造的 ELF 静态内存视图

index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

index	string
0	null string
1	name.
7	Variable
11	able
16	able
24	null string

图 1-3 字符串表

由图 1-3 可见，字符串表中各个字符串的构成和 C 语言中的字符串完全一样，都以'\0'作为一个字符串的结束标记。由 index 指向的字符串是从字符串表第 index 个字符开始，直到遇到一个'\0'标记，如果 index 处恰好是'\0'，那么 index 指向的就是个空串（null string）。

在驱动模块所在的 ELF 文件中，一般有两个这样的字符串表 section，一个用来保存各 section

名称的字符串，另一个用来保存符号表中每个符号名称的字符串。虽然同样都是字符串表 section，但是得到这两个 section 的基地址的方法并不一样。

section 名称字符串表的基地址为 `char *secstrings = (char *)hdr + entry[hdr->e_shstrndx].sh_offset`。而获得符号名称字符串表的基地址则有点绕：首先要遍历 Section header table 中所有的 entry，去找一个 `entry[i].sh_type = SHT_SYMTAB` 的 entry，`SHT_SYMTAB` 表明这个 entry 所对应的 section 是一符号表。这种情况下，`entry[i].sh_link` 是符号名称字符串表 section 在 Section header table 中的索引值，换句话说，符号名称字符串表所在 section 的基地址为 `char *strtab = (char *)hdr + entry[entry[i].sh_link].sh_offset`。

如此，若想获得某一 section 的名称（假设该 section 在 Section header table 中的索引值是 i），那么用 `secstrings + entry[i].sh_name` 即可。

至此，`load_module` 函数通过以上计算获得了 section 名称字符串表的基地址 `secstrings` 和符号名称字符串表的基地址 `strtab`，留作将来使用。

○ HDR 视图的第一次改写

在获得了 section 名称字符串表的基地址 `secstrings` 和符号名称字符串表的基地址 `strtab` 之后，函数开始第一次遍历 Section header table 中的所有 entry，将每个 entry 中的 `sh_addr` 改写为 `entry[i].sh_addr = (size_t)hdr + entry[i].offset`，这样 `entry[i].sh_addr` 将指向该 entry 所对应的 section 在 HDR 视图中的实际存储地址。

在遍历过程中，如果发现 `CONFIG_MODULE_UNLOAD` 宏没有定义⁶，表明系统不支持动态卸载一个模块，这样，对于名称为“`.exit`”的 section，将来就没有必要把它加载到内存中，内核代码于是清除对应 entry 中 `sh_flags` 里面的 `SHF_ALLOC` 标志位。

相对于刚复制到内核空间的 HDR 视图，HDR 视图的第一次改写只是在自身基础上修改了 Section header table 中的某些字段，其他方面没有任何变化。接下来在“HDR 视图的第二次改写”一节中将会看到改写后的 HDR 视图会再次被改写，在那里，HDR 视图中的绝大部分 section 会被搬移到一个新的内存空间中，那也是它们最终的内存位置。

○ find_sec 函数

内核用 `find_sec` 来寻找某一 section 在 Section header table 中的索引值，其函数原型为：

```
static unsigned int find_sec(Elf_Ehdr *hdr,
                             Elf_Shdr *sechdrs,
                             const char *secstrings,
```

⁶ `CONFIG_MODULE_UNLOAD` 宏用来表明 Linux 内核是否支持 module 的 unload 特性，即是否支持使用 `rmmod` 工具从系统中卸载一个模块。

```
const char *name);
```

函数返回该 section 的索引值，如果没有找到对应的 section，则返回 0。该函数的前两个参数分别是 ELF 文件的 ELF header 和 section header。因为函数要查找的是某一 section 的 name，所以第三个参数就是前面提到的 secstrings，第四个参数则是要查找的 section 的 name。函数的具体实现过程非常简单：遍历 Section header table 中所有的 entry（忽略没有 SHF_ALLOC 标志的 section，因为这样的 section 最终不占有实际内存地址），对每一个 entry，先找到其所对应的 section name，然后和第四个参数进行比较，如果相等，就找到对应的 section，返回该 section 在 Section header table 中的索引值。

在对 HDR 视图进行第一次改写之后，内核通过调用 find_sec，分别查找以下名称的 section：“.gnu.linkonce.this_module”，“__versions”和“.modinfo”。查找的索引值分别保存在变量 modindex、versindex 和 infoindex 中，以备将来使用。

○ struct module 类型变量 mod 初始化

1.3.2 节中提到了 struct module 是内核用来表示一个模块的非常重要的数据结构。在 load_module 函数中定义有一个 struct module 类型的变量 mod，该变量的初始化是通过模块 ELF 文件中一个名为“.gnu.linkonce.this_module”的 section 来完成的。

ELF 文件中出现的这个 section 其实是模块的编译工具链完成的，与设备驱动程序员无关。如果我们仔细看一下编译后的模块所在的目录，一定会发现一个扩展名为“.mod.c”的文件，打开该文件，会发现如下定义：

```
struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) = {
    .name = KBUILD_MODNAME,
    .init = init_module,
#ifdef CONFIG_MODULE_UNLOAD
    .exit = cleanup_module,
#endif
    .arch = MODULE_ARCH_INIT,
};
```

其中的__attribute__((section(".gnu.linkonce.this_module")))部分很清楚地揭示了内核模块 ELF 文件中“.gnu.linkonce.this_module” section 出现的根源。

这段定义还有一个比较有趣的地方在于对 struct module 结构体中的 init 和 exit 成员变量的初始化：

```
.init = init_module
.exit = cleanup_module
```

直觉告诉我们，这里的 init 和 exit 应该指向我们的驱动程序源码中定义的模块初始化和退

出函数，然而经过和实际的设备驱动程序源码对比，有些读者也许会很失望，在驱动程序的源码中定义的初始化和退出函数并不是 `init_module` 和 `cleanup_module`。这其实是拜 `module_init` 和 `module_exit` 宏所赐，它们利用了 gcc 提供的别名技术（`__attribute__((alias))`）。

```
#define module_init(initfn) \
    static inline initcall_t __inittest(void) \
    { return initfn; } \
    int init_module(void) __attribute__((alias(#initfn)));
```

该宏定义的核心是最后一句，它将 `init_module` 函数的别名设定为 `initfn`，而后者正是我们在设备驱动程序中定义的模块初始化函数。总之，模块的构造工具链为我们安插了一个“.gnu.linkonce.this_module” section，并初始化了其中的一些成员。在模块加载过程中，`load_module` 函数将利用这个 section 中的数据来初始化 `mod` 变量。

模块被加载到内存中之后，内核通过 `find_sec` 函数查找到“.gnu.linkonce.this_module” section 在 Section header table 中所对应的索引值 `modindex`，这样通过下面这行简单的代码就得到了“.gnu.linkonce.this_module” section 在内存中的实际地址。

```
mod = (void *)sechdrs[modindex].sh_addr;
```

于是，在第一次改写的 HDR 视图的基础上，`mod` 指针指向了实际的 struct module 所在的内存地址。接下来我们会看到，在 HDR 视图第二次被改写后，`mod` 指针将会重新指向“.gnu.linkonce.this_module” section 在内存中的最终地址。

○ HDR 视图的第二次改写

在这次改写中，HDR 视图中绝大多数的 section 会被搬移到新的内存空间中，之后会根据这些 section 新的内存地址再次改写图 1-2 中的 HDR 视图，使其中 Section header table 中各 entry 的 `sh_addr` 指向新的也是最终的内存地址。

在为那些需要移动的 section 分配新的内存空间地址之前，内核需要决定出 HDR 视图中哪些 section 需要移动，如果移动的话要移动到什么位置。内核代码中 `layout_sections` 函数用来做这件事，在 `layout_sections` 函数中，内核会遍历 HDR 视图中的每一个 section，对每一个标记有 `SHF_ALLOC`⁷的 section，将其划分到两大类 section 当中：CORE 和 INIT。

为了完成这种分类，`layout_sections` 函数首先为标记了 `SHF_ALLOC` 的 section 定义了四种类型：code、read-only data、read-write data 和 small data。任何一个标记了 `SHF_ALLOC` 的 section 必定属于这四类中的一类。之后，对应每一个分类，函数都会遍历 Section header table

⁷ `SHF_ALLOC` 标记表示该 section 在模块运行过程中，需要占用内存空间。根据 ELF spec (Portable Formats Specification, Version 1.1): `SHF_ALLOC`—The section occupies memory during process execution. Some control sections do not reside in the memory image of an object file, this attribute is off for those sections.

中的所有项，将 section name 不是以".init"开始的 section 划归为 CORE section，并且修改 HDR 视图中 Section header table 中对应 entry 的 sh_entsize，用以记录当前 section 在 CORE section 中的偏移量。

```
entry[i].sh_entsize = mod->core_size;
```

同时用 struct module 结构中的成员变量 core_size 记录下当前正在操作的 section 为止 CORE section 的空间大小。

```
mod->core_size += entry[i].sh_size;
```

对于 CORE section 中的 code section，内核用 struct module 结构中的 core_text_size 来记录。

对于 INIT section 的分类，和 CORE section 的划分基本一样，不同的地方在于属于 INIT section 的 section，其 name 必须以".init"开始，内核用 struct module 结构中的成员变量 init_size 来记录当前 INIT section 空间的大小。

```
mod->init_size += entry[i].sh_size;
```

对于 INIT section 中的 code section，内核用 struct module 结构中的 init_text_size 来记录。

在对 section 进行搬移之前，接下来会有个对符号表的处理，内核代码中通过调用 layout_symtab 函数来完成。Linux 的内核源码中根据是否启用了内核配置选项 CONFIG_KALLSYMS 给出了 layout_symtab 函数的两种不同的定义。

如果没有启用 CONFIG_KALLSYMS，那么 layout_symtab 函数就是个空函数，不任何事情。CONFIG_KALLSYMS 是一个决定内核映像中是否保留所有符号的配置选项，在内核配置文件 Kconfig 中，可以看到如下说明：

Say Y here to let the kernel print out symbolic crash information and symbolic stack backtraces. This increases the size of the kernel somewhat, as all symbols have to be loaded into the kernel image.

简言之，这是个为了方便系统调试而增加的选项，启用它的代价就是导致最终内核映像文件变大（当然占用的系统内存也会相应增加）。

在启用了 CONFIG_KALLSYMS 选项的 Linux 源码树基础上编译内核模块，会导致内核模块也会保留模块中的所有符号，这些符号都放在 ELF 符号表 section 中。由于在内核模块的 ELF 文件中，符号表所在的 section 没有 SHF_ALLOC 标志，所以上面提到的 layout_sections 函数不会把符号表 section 划到 CORE section 或者是 INIT section 中，这也是为什么要通过另外一个函数 layout_symtab 来把符号表搬移到 CORE section 内存区中的原因。

在对内核模块 ELF 文件中的 section 进行了 CORE 和 INIT 的划分之后，内核调用 vmalloc 相关的函数为 CORE section 和 INIT section 分配对应的内存空间，基地址分别记录在 mod->module_core 和 mod->module_init 中，然后把对应的 section 数据搬移到其在 CORE

section 和 INIT section 内存空间的最终位置上。显然，在把各 section 搬移到其新的内存地址之后，内核需要改写 HDR 视图中的 Section header table 中对应 entry 的 sh_addr，以使其指向新的地址。

注意，由于此时 “.gnu.linkonce.this_module” section 是一个带有 SHF_ALLOC 标志的可写数据 section，也会被搬移到 CORE section 内存空间中，所以必须更新 mod 变量使之指向新的内存地址。

```
mod = (void *)entry[modindex].sh_addr;
```

这里之所以要对 HDR 视图中的某些 section 做这样的搬移，是因为在模块加载过程结束时，系统会释放掉 HDR 视图所在的内存区域，不仅如此，在模块初始化工作完成后，INIT section 所在的内存区域也会被释放掉。由此可见，当一个模块被成功加载进系统，初始化工作完成之后，最终留下的仅仅是 CORE section 中的内容，因此 CORE section 中的数据应是模块在系统中整个存活期会使用到的数据。

如此处理之后，我们在图 1-2 的基础上得到了图 1-4：

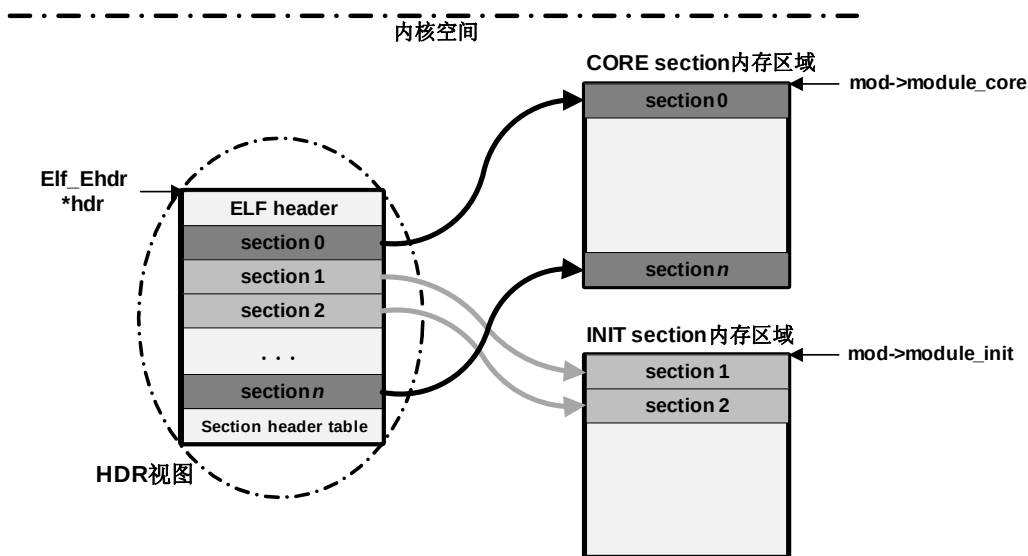


图 1-4 模块加载时的 section 搬移

○ 模块导出的符号

我们知道模块不仅可以使使用内核或者其他模块导出的符号，而且可以向外部导出自己的符号，模块导出符号使用的宏和内核导出符号所使用的完全一样：EXPORT_SYMBOL、EXPORT_SYMBOL_GPL 和 EXPORT_SYMBOL_FUTURE。由 1.2 节对这些宏的代码分析可知，内核模块会把导出的符号分别放到 “__ksymtab”、“__ksymtab_gpl” 和 “__ksymtab_gpl_future” section 中。

如果一个内核模块向外界导出了自己的符号，那么将由模块的编译工具链负责生成这些导出符号 section，而且这些 section 都带有 SHF_ALLOC 标志，所以在模块加载过程中会被搬到 CORE section 区域中。如果模块没有向外界导出任何符号，那么在模块的 ELF 文件中，将不会产生这些 section。

显然，内核需要对模块导出的符号进行管理，以便在处理其他模块中那些“未解决的引用”符号时能够找到这些符号。内核对模块导出的符号的管理使用到了 struct module 结构中如下的成员变量：

```
struct module
{
    ...
    /* Exported symbols */
    const struct kernel_symbol *syms;
    const unsigned long *crcs;
    unsigned int num_syms;

    /* GPL-only exported symbols. */
    unsigned int num_gpl_syms;
    const struct kernel_symbol *gpl_syms;
    const unsigned long *gpl_crcs;

    /* symbols that will be GPL-only in the near future. */
    const struct kernel_symbol *gpl_future_syms;
    const unsigned long *gpl_future_crcs;
    unsigned int num_gpl_future_syms;
    ...
}
```

在把 HDR 视图中的 section 搬到最终的 CORE section 和 INIT section 之后，内核通过对 HDR 视图中 Section header table 的查找，获得“__ksymtab”、“__ksymtab_gpl”和“__ksymtab_gpl_future” section 在 CORE section 中的地址，将其记录在 mod->syms、mod->gpl_syms 和 mod->gpl_future_syms 中，代码片段如下：

```
i = find_sec(..., "__ksymtab",...);
mod->syms = (struct kernel_symbol *)entry[i].sh_addr;
j = find_sec(..., "__ksymtab_gpl",...);
mod->syms_gpl = (struct kernel_symbol *)entry[j].sh_addr;
k = find_sec(..., "__ksymtab_gpl_future",...);
mod->gpl_future_syms = (struct kernel_symbol *)entry[k].sh_addr;
```

如此，内核通过这些变量将可得到模块导出的符号的所有信息，如图 1-5 所示。读者将在接下来的“find_symbol 函数”部分中看到这些变量的具体用途。

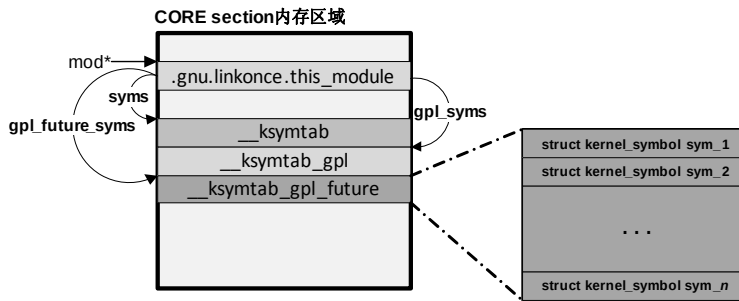


图 1-5 内核模块导出的符号

○ find_symbol 函数

在模块加载过程中，`find_symbol` 是个非常重要的函数，顾名思义，它用来查找一个符号。该函数的原型如下：

```
const struct kernel_symbol *find_symbol(const char *name,
                                       struct module **owner,
                                       const unsigned long **crc,
                                       bool gplok,
                                       bool warn);
```

其中，第一个参数表示要查找的符号名，第二个参数用以表明符号可能所在的模块。

在深入到这个函数内部之前，有必要先介绍几个数据结构，这几个数据结构将在 `find_symbol` 函数中用到。

```
struct symsearch {
    const struct kernel_symbol *start, *stop;
    const unsigned long *crcs;
    enum {
        NOT_GPL_ONLY,
        GPL_ONLY,
        WILL_BE_GPL_ONLY,
    } licence;
    bool unused;
};
```

`struct symsearch` 用来对应要查找的每一个符号表 section，换句话说，对要查找的每个符号表 section，内核代码都要为之产生一个 `struct symsearch` 类型的实例。结构体中的成员变量 `start` 和 `stop` 分别指向对应 section 的开始和结束地址，`bool` 型的 `unused` 成员用来表示内核是否配置了 `CONFIG_UNUSED_SYMBOLS` 选项，不过这个选项是“非主流”的，长远看这个选项最终会消失，因此本书只在这里提一下，在后续的章节中将忽略所有该选项被启用时才起作用的代码。另一个比较重要的成员是 `enum` 型的 `licence`，`GPL_ONLY` 表示符号只提供满足 GPL 协议的模块使用，`NOT_GPL_ONLY` 表示不一定要只给满足 GPL 协议

的模块使用，WILL_BE_GPL_ONLY 表示将来只提供给满足 GPL 协议的模块使用。再提醒一下，NOT_GPL_ONLY 符号由 EXPORT_SYMBOL 负责导出，GPL_ONLY 符号由 EXPORT_SYMBOL_GPL 负责导出，WILL_BE_GPL_ONLY 符号由 EXPORT_SYMBOL_GPL_FUTURE 负责导出。

```
struct find_symbol_arg {
    /* Input */
    const char *name;
    bool gplok;
    bool warn;

    /* Output */
    struct module *owner;
    const unsigned long *crc;
    const struct kernel_symbol *sym;
};
```

find_symbol_arg 用做查找符号的标识参数，可以看到其大部分数据成员与 find_symbol 函数原型中的参数完全一致，其中的 kernel_symbol 是一个用以表示内核符号构成的数据结构，在前面的“EXPORT_SYMBOL 的内核实现”一节中介绍过。

以下仔细分析 find_symbol 的功能，其源代码如下：

```
<kernel/module.c>
-----
const struct kernel_symbol *find_symbol(const char *name,
                                       struct module **owner,
                                       const unsigned long **crc,
                                       bool gplok,
                                       bool warn)
{
    struct find_symbol_arg fsa;

    fsa.name = name;
    fsa.gplok = gplok;
    fsa.warn = warn;

    if (each_symbol(find_symbol_in_section, &fsa)) {
        if (owner)
            *owner = fsa.owner;
        if (crc)
            *crc = fsa.crc;
        return fsa.sym;
    }

    DEBUGP("Failed to find symbol %s\n", name);
```

```

    return NULL;
}

```

函数首先构造被查找模块的标识参数 `fsa`，然后通过 `each_symbol` 来查找符号。`each_symbol` 是用来进行符号查找的主要函数，为节约篇幅起见，这里不再摘录其源代码，而是直接讲述其主要功能框架。

总体上，`each_symbol` 函数可以分成两个部分：第一部分是在内核导出的符号表中查找对应的符号，如果找到，就通过 `fsa` 返回该符号的信息，否则，再进行第二部分的查找；第二部分是在系统中已加载的模块（系统中所有已成功加载的模块都以链表的形式保存在一个全局变量 `modules` 中）的导出符号表中查找对应的符号，如果找到就通过 `fsa` 返回该符号的信息，否则函数返回 `false`。图 1-6 展示了 `find_symbol` 在查找一个符号时的搜索路径：

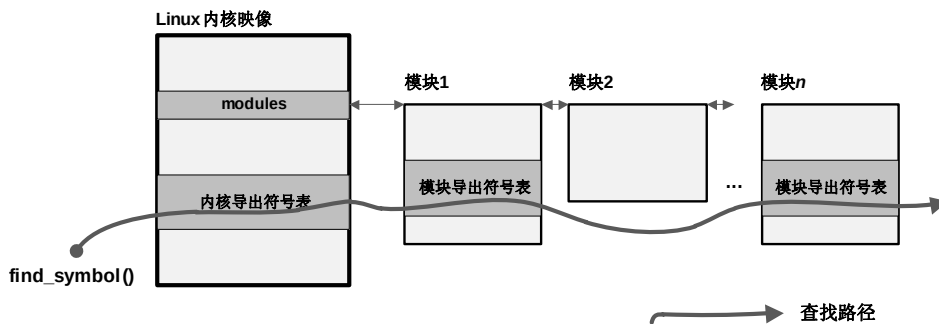


图 1-6 `find_symbol` 查找符号时的搜索路径

第一部分在对内核符号表进行查找时，首先构造一个 `struct symsearch` 类型的数组 `arr`。

```

static const struct symsearch arr[] = {
    { __start__ksymtab, __stop__ksymtab, __start__kcrctab,
      NOT_GPL_ONLY, false },
    { __start__ksymtab_gpl, __stop__ksymtab_gpl,
      __start__kcrctab_gpl,
      GPL_ONLY, false },
    { __start__ksymtab_gpl_future, __stop__ksymtab_gpl_future,
      __start__kcrctab_gpl_future,
      WILL_BE_GPL_ONLY, false },
};

```

注意这里的 `__start__ksymtab`、`__start__kcrctab` 和 `__stop__ksymtab` 等变量已经在前面的“`EXPORT_SYMBOL` 的内核实现”一节中交代过，它们在内核的链接脚本中定义，由链接器负责产生，由内核源码负责声明，现在到了使用它们的时候了。

接下来函数通过调用 `each_symbol_in_section` 查询内核的导出符号表，`each_symbol_in_section` 的核心代码如下（经过适当改写）：

```
<kernel/module.c>
```

```
static bool each_symbol_in_section(const struct symsearch *arr, struct module *owner, void *fsa)
{
    unsigned int i, j;

    for (j = 0; j < ARRAY_SIZE(arr); j++) {
        for (i = 0; i < arr[j].stop - arr[j].start; i++)
            if (find_symbol_in_section(&arr[j], owner, i, fsa))
                return true;
    }

    return false;
}
```

为了在内核的导出符号表中查找某一指定的符号名，`each_symbol_in_section` 函数使用了两层 `for` 循环：外层 `j` 引导的 `for` 循环用来遍历符号可能所在的内核导出符号表中的各 `section`；内层 `i` 引导的 `for` 循环用来遍历外层 `for` 循环所指定的 `section` 中的每个 `struct kernel_symbol` 类型的元素。对于每个 `kernel_symbol`，都会调用 `find_symbol_in_section` 函数。

为了清楚地理解内核加载模块时如何处理“未解决的引用”符号，有必要仔细分析一下 `find_symbol_in_section` 函数的主要功能。因为对 Linux 下的设备驱动程序员而言，几乎每天都在和这个功能打交道，清楚地理解其内核机制，将来一旦在加载模块时出现相关问题，也可以将其快速定位并最终解决。另外，对于带有“_GPL”后缀的符号名，在写驱动程序的 `内核模块` 时常常会遇到，然而其背后到底蕴涵着怎样的设计理念呢？通过分析 `find_symbol_in_section` 函数，就可以得到所需的答案。

`find_symbol_in_section` 函数的完整源代码如下：

```
<kernel/module.c>
```

```
static bool find_symbol_in_section(const struct symsearch *syms,
                                   struct module *owner,
                                   unsigned int symnum, void *data)
{
    struct find_symbol_arg *fsa = data;

    if (strcmp(syms->start[symnum].name, fsa->name) != 0)
        return false;

    if (!fsa->gplok) {
        if (syms->licence == GPL_ONLY)
            return false;
        if (syms->licence == WILL_BE_GPL_ONLY && fsa->warn) {
            printk(KERN_WARNING "Symbol %s is being used "
                       "by a non-GPL module, which will not ")
        }
    }
}
```

```

        "be allowed in the future\n", fsa->name);
    printk(KERN_WARNING "Please see the file "
           "Documentation/feature-removal-schedule.txt "
           "in the kernel source tree for more details.\n");
    }
}

fsa->owner = owner;
fsa->crc = symversion(syms->crs, symnum);
fsa->sym = &syms->start[symnum];
return true;
}

```

函数首先用 `strcmp` 函数来比较 `kernel_symbol` 结构体中的 `name` 与 `fsa` 中的 `name`（正在查找的符号名，即要加载的内核模块中出现的“未解决的引用”的符号）是否匹配，如果不匹配，那么函数直接返回 `false`。

`fsa->gplok` 和 `fsa->warn` 的设定最早是在 `find_symbol` 函数中，是通过后者的函数参数传入的。`fsa->warn` 主要用来控制警告信息的输出。`fsa->gplok` 用来表示当前的模块是不是满足 GPL 协议（GPL module 或 non-GPL module），`fsa->gplok = true` 表明这是个 GPL module，否则就是 non-GPL module。内核判断一个模块是否 GPL 兼容，要使用到本章后面的“模块的信息”部分中的内容。

对于一个 non-GPL module 而言，它不能使用内核导出的属于 `GPL_ONLY` 的那些符号，所以即使要查找的符号匹配上一个属于 `GPL_ONLY` 的符号，也不能认为查找成功。但是如果要查找的符号匹配上一个属于 `WILL_BE_GPL_ONLY` 的符号，因为这个导出的符号“将成为 `GPL_ONLY`”，所以即使现在还不是 `GPL_ONLY`，查找姑且算是成功的，不过即便如此，内核对模块将来对该符号的成功使用没有保障，所以应该给出一个警告信息。对于一个 GPL module 而言，一切好说，可以使用内核导出的所有符号。

函数如果成功查找到符号，利用传进来的 `data` 指针将符号相关信息传给上层调用的函数。

至此，`find_symbol` 的第一部分，即在内核导出的符号表中查找指定的符号已经结束。如果指定的符号没有出现在内核导出的符号表中，那么将进入 `find_symbol` 函数的第二部分。

下面开始介绍 `find_symbol` 的第二部分，在系统已经加载的模块导出的符号表中查找符号。内核为达成此目的，需要在加载一个内核模块时完成下面两件事。

第一，模块成功加载进系统之后，需要将表示该模块的 `struct module` 类型变量 `mod` 加入到 `modules` 中，后者是一个全局的链表变量，用来记录系统中所有已加载的模块。

```

<kernel/module.c>
-----
static LIST_HEAD(modules);
list_add_rcu(&mod->list, &modules);

```


第二，模块导出的符号信息记录在 `mod` 的相关成员变量中，这个过程的详细描述参见本章前面的“模块导出的符号”部分。

`each_symbol` 用来在系统所有已加载的模块导出的符号中查找某一指定符号，其核心代码片段如下：

```
if (each_symbol_in_section(arr, ARRAY_SIZE(arr), NULL, fn, data))
    return true;
list_for_each_entry_rcu(mod, &modules, list) {
    struct symsearch arr[] = {
        { mod->syms, mod->syms + mod->num_syms, mod->crcls,
          NOT_GPL_ONLY, false },
        { mod->gpl_syms, mod->gpl_syms + mod->num_gpl_syms,
          mod->gpl_crcls,
          GPL_ONLY, false },
        { mod->gpl_future_syms,
          mod->gpl_future_syms + mod->num_gpl_future_syms,
          mod->gpl_future_crcls,
          WILL_BE_GPL_ONLY, false },
    };

    if (each_symbol_in_section(arr, ARRAY_SIZE(arr), mod, fn, data))
        return true;
}
```

相对于 `find_symbol` 的第一部分（在内核导出的符号表中查找某一符号），第二部分唯一的区别在于构造的 `arr` 数组。函数在全局链表 `modules` 中遍历所有已加载的内核模块，对其中的每一模块都构造一个新的 `arr` 数组，然后在其中查找特定的符号。

○ 对“未解决的引用”符号（unresolved symbol）的处理

前文中已多次提到内核模块 ELF 文件中的“未解决的引用”符号，所谓的“未解决的引用”符号，就是模块的编译工具链在对模块进行链接生成最终的 `.ko` 文件时，对于模块中调用的一些函数，最简单的比如 `printk` 函数，链接工具无法在该模块的所有目标文件中找到这个函数的具体指令码（因为这个函数是在 Linux 的内核源代码中实现的，其指令码存在于编译内核生成的目标文件中，模块的链接工具显然不会也不应该去查找内核的目标文件），所以就会将这个符号标记为“未解决的引用”，对它的处理将一直延续到内核模块被加载时（处理的核心是在内核或者是其他内核模块导出的符号中找到这个“未解决的引用”符号⁸，继而找到该符号所在的内存地址，从而最终形成正确的函数调用）。

⁸ 读者可在 Linux 环境下通过 `nm` 命令来查看一个模块中出现的所有“未解决的引用”符号，比如：

```
dennis@AMDLinuxFGL:~$ nm demodrv.ko
```

该命令的输出中，所有前面带一“U”标志的符号均为“未解决的引用”符号。

Linux 内核中，一个名为 `simplify_symbols` 的函数用来实现这一功能，这是个很有意思的函数，我们不妨仔细看一下它的代码。

<kernel/module.c>

```

/* Change all symbols so that st_value encodes the pointer directly. */
static int simplify_symbols(struct module *mod, const struct load_info *info)
{
    Elf_Shdr *symsec = &info->sechdrs[info->index.sym];
    Elf_Sym *sym = (void *)symsec->sh_addr;
    unsigned long secbase;
    unsigned int i;
    int ret = 0;
    const struct kernel_symbol *ksym;

    for (i = 1; i < symsec->sh_size / sizeof(Elf_Sym); i++) {
        const char *name = info->strtab + sym[i].st_name;

        switch (sym[i].st_shndx) {
        case SHN_COMMON:
            /* We compiled with -fno-common. These are not
               supposed to happen. */
            DEBUGP("Common symbol: %s\n", name);
            printk("%s: please compile with -fno-common\n",
                   mod->name);
            ret = -ENOEXEC;
            break;

        case SHN_ABS:
            /* Don't need to do anything */
            DEBUGP("Absolute symbol: 0x%08lx\n",
                   (long)sym[i].st_value);
            break;

        case SHN_UNDEF:
            ksym = resolve_symbol_wait(mod, info, name);
            /* Ok if resolved. */
            if (ksym && !IS_ERR(ksym)) {
                sym[i].st_value = ksym->value;
                break;
            }

            /* Ok if weak. */
            if (!ksym && ELF_ST_BIND(sym[i].st_info) == STB_WEAK)
                break;

            printk(KERN_WARNING "%s: Unknown symbol %s (err %li)\n",

```

```

        mod->name, name, PTR_ERR(ksym));
    ret = PTR_ERR(ksym) ?: -ENOENT;
    break;

default:
    /* Divert to percpu allocation if a percpu var. */
    if (sym[i].st_shndx == info->index.pcpu)
        secbase = (unsigned long)mod_percpu(mod);
    else
        secbase = info->sechdrs[sym[i].st_shndx].sh_addr;
    sym[i].st_value += secbase;
    break;
}
}

return ret;
}

```

简言之，在加载模块的过程中，`simplify_symbols` 函数用来为当前正在加载的模块中所有“未解决的引用”符号产生正确的目标地址。对这段代码的透彻理解需要读者熟悉 ELF 文件格式规范的相关概念，我们不可能在本书中全面介绍 ELF 文件格式，但是为了让读者能理解上面的代码，还是从代码的角度出发，将其中所涉及的一些有关 ELF 文件的概念予以简单介绍。

代码中的 `Elf_Sym` 定义的是符号表中的元素，具体定义如下：

```

struct Elf_Sym {
    Elf32_Word    st_name;
    Elf32_Addr    st_value;
    Elf32_Word    st_size;
    unsigned char st_info;
    unsigned char st_other;
    Elf32_Half    st_shndx;
};

```

其中，`st_name` 是符号名在符号名称字符串表中的索引值，详见本章前面的“字符串表(String Table)”部分。`st_value` 是符号所在的内存地址。`simplify_symbols` 函数的唯一功能就是在加载模块时重新生成正确的 `st_value` 值。`st_shndx` 是该符号所在的 section 在 Section header table 中的索引值。但是该值还有一些特殊的定义。对于符号表，它是 ELF 文件中的一个 section，这个 section 就是由一系列 `struct Elf_Sym` 型元素所构成的一个数组，每个元素代码一个符号。

在对符号表的概念有了基本了解之后，回过头来看看 `simplify_symbols` 函数的代码实现。函数首先通过一个 `for` 循环遍历符号表中的所有符号，对于每一个符号都会根据该符号的

`st_shndx` 值分情况进行处理。前面刚刚提到 `st_shndx`，通常情况下，该值表示符号所在 section 的索引值，为方便叙述，我们称这种符号为一般符号。对于一般符号来说，它的 `st_value` 在 ELF 文件中的值是从其所在 section 起始处算起的一个偏移量，代码中在 switch 的 default 分支下进行处理：先得到符号所在 section 的最终内存地址，然后加上它在 section 中的偏移量，这样就得到了符号的最终内存地址。

除了一般符号，还有些符号的 `st_shndx` 具有特殊的含义，典型的如 `SHN_ABS` 和 `SHN_UNDEF`，前者表明该符号具有绝对地址，因此 `simplify_symbols` 函数无须对这种情况予以任何处理，后者表明该符号是一“undefined symbol”，其实就是我们一直说的“未解决的引用”符号。这种情况下 `simplify_symbols` 函数会调用 `resolve_symbol` 函数来处理该未定义符号，后者会调用 `find_symbol` 函数去查找该符号（详细的查找过程见本章前面的“`find_symbol` 函数”部分），如果找到了，就把它在内存中的实际地址赋值给 `st_value`。

如此，经过 `simplify_symbols` 函数的调用之后，内核模块符号表中的所有符号就都有了正确的 `st_value` 值，也即都有了正确的内存地址。

到目前为止，一切关于符号相关的处理貌似都很完美，然而情况真是如此吗？如果当前正在加载的模块中一个“未解决的引用”符号是由别的内核模块导出的，情况会怎样呢？如果读者的探索精神足够强烈，想想那些由内核模块导出的符号吧⁹。由前面的内容可知，“`__ksymtab`”、“`__ksymtab_gpl`”和“`__ksymtab_gpl_future`” section 都被搬移到了最终的内存地址处，而且这些地址也被表示模块的 `mod` 变量记录在案，但是这些 section 中的内容呢？

回头看看图 1-5 “内核模块导出的符号”，每个“`__ksymtab`”、“`__ksymtab_gpl`”和“`__ksymtab_gpl_future`” section 都是由 `struct kernel_symbol` 类型的元素所构成的数组。到目前为止，如果仔细考察每个元素的话，会发现其中的 `value` 成员依然是内核模块在静态编译时产生的地址。换句话说，根本不是这些符号在模块被加载进系统之后在内存中的实际地址。这显然不是我们想要的效果：想想本节前半部分提到的对模块中“未解决的引用”符号的处理，如果在别的模块中找到的符号其内存地址只是当初该模块在静态链接时填入的地址，那么对该符号的引用必然导致错误的内存访问。这是个很严重的问题。而 Linux 内核对这一问题的处理便引出了下一部分的内容——重定位。

○ 重定位 (relocation)

重定位主要用来解决静态链接时的符号引用与动态加载时实际符号地址不一致的问题，上节结束部分提到的模块导出的符号地址，就是一个典型的需要重定位的例子。仔细讨论重定位的内容不是件简单的事情，因为重定位的任务包含很多方面的内容，尤其是跟体系架

⁹ 为什么只关注那些由内核模块导出的符号呢？因为对于内核导出的符号而言，所有符号的实际链接地址都会被解决，除非内核被设计成可重定位的。而内核模块则不同，它本身就是可重定位的。

构相关的一些微妙晦涩的技术细节。考虑到本书的主题定位，也许在这里详细讲述重定位的技术细节并不是件很有价值的事情：篇幅把握得不够理想，很可能就冲淡了本章关于内核模块加载这一主线。消耗大量的篇幅和读者大量的时间，所涉及的主题在现实中却难有用武之地。但是重定位毕竟是内核加载过程中一个很重要的步骤，仔细权衡之下，笔者决定采取一个相对折中的方案，在讨论重定位时就事论事。本节就以上节末尾提出的问题的来展开重定位的话题。

如果模块有用 `EXPORT_SYMBOL` 导出的符号，那么模块的编译工具链会为此模块的 ELF 文件生成一个独立的特殊 section：“`.rel__ksymtab`”，它专门用于对“`__ksymtab`” section 的重定位，称为 relocation section。这个 section 是由下面的数据结构元素形成的一个数组。

```
typedef struct elf32_rel {
    Elf32_Addr    r_offset;
    Elf32_Word    r_info;
} Elf32_Rel;
```

先来看看 Linux 源码中用于内核模块加载时重定位的代码：

<kernel/module.c>

```
static int apply_relocations(struct module *mod, const struct load_info *info)
{
    unsigned int i;
    int err = 0;

    /* Now do relocations. */
    for (i = 1; i < info->hdr->e_shnum; i++) {
        unsigned int infosec = info->sechdrs[i].sh_info;

        /* Not a valid relocation section? */
        if (infosec >= info->hdr->e_shnum)
            continue;

        /* Don't bother with non-allocated sections */
        if (!(info->sechdrs[infosec].sh_flags & SHF_ALLOC))
            continue;

        if (info->sechdrs[i].sh_type == SHT_REL)
            err = apply_relocate(info->sechdrs, info->strtab,
                                info->index.sym, i, mod);
        else if (info->sechdrs[i].sh_type == SHT_RELA)
            err = apply_relocate_add(info->sechdrs, info->strtab,
                                    info->index.sym, i, mod);

        if (err < 0)
            break;
    }
}
```

```

    return err;
}

```

代码用一个 for 循环来遍历 HDR 视图中 Section header table 中所有的 entry。对于一个重定位的 section，其 entry 中的 sh_type 的值为 SHT_REL 或者 SHT_RELA，分别对应两种不同的重定位方式，我们拿第一种类型 SHT_REL 来说事。对于 sh_type = SHT_REL 的 section 而言，其 Section header 中的 sh_info 成员指明了被重定位的 section 在 Section header table 中的索引值，代码中用 info 变量来表示。

在遍历的过程中，如果发现了一个 sh_type = SHT_REL 的 section，系统就调用 apply_relocate 函数来执行重定位，后者是个体系统结构相关的函数。总体上，该函数对模块导出符号的重定位原理是，根据重定位元素中的 r_offset 以及 relocation section header entry 中的 sh_info 得到需要修改的导出符号 struct kernel_symbol 中 value 所在的内存地址：

```

Elf32_Rel *rel = (void *)entry[i].sh_addr; //entry[i]对应当前正在处理的 relocation section
int ksymtabidx = entry[i].sh_info;
Elf32_Shdr *ksymtabsec = &entry[ksymtabidx];
unsigned long location = ksymtabsec->sh_addr + rel->r_offset;

```

然后根据重定位元素中的 r_info 获得需要定位的符号在符号表中的偏移量：

```
offset = ELF32_R_SYM(rel->r_info);
```

因为符号表 section 的基地址很容易获得，于是就可以获得需要重定位的符号在符号表中对应的 Elf32_Sym 型元素：

```
sym = ((Elf32_Sym *)symsec->sh_addr) + offset; //symsec->sh_addr 为符号表 section 基地址
```

所以，最终导出符号的地址被修改。

这一过程简单地说，就是根据导出符号所在 section 的 relocation section，结合导出符号表 section，修改导出符号的地址为在内存中最终的地址值。如此，内核模块导出符号的地址在系统执行完重定位之后被更新为正确的值。

○ 模块参数

内核模块在用 insmod 命令加载时，可以通过诸如以下的命令向模块传递一些参数：

```
insmod demodev.ko dolphin=10 bobcat=5
```

其中 dolphin=10 和 bobcat=5 就是向模块传递的参数，dolphin 和 bobcat 是参数名，10 和 5 是具体的参数值。当然为了能正确接收外部的参数，内核模块本身在源代码中必须用 module_param 宏声明模块可以接收的参数。在上面的例子中，模块应该使用诸如 module_param(dolphin, int, 0)来声明一个模块参数，例如下面的代码片段：

```
<demodev.c>
```

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

int dolphin;
int bobcat;
module_param(dolphin, int, 0);
module_param(bobcat, int, 0);

static int demodev_init(void)
{
    printk("dolphin=%d,bobcat=%d\n", dolphin, bobcat);
    return 0;
}
static void demodev_exit(void)
{
    printk("+demodev_exit!\n");
}
module_init(demodev_init);
module_exit(demodev_exit);
```

从本章稍后的讨论中可以得知，内核模块加载器对模块参数的构造（初始化）过程发生在对模块初始化函数 `demodev_init` 的调用之前，所以在 `demodev_init` 函数被调用时，已经可以得到从命令行传过来的实际参数。

Linux 源码中 `module_param` 宏相关的完整定义如下：

```
<include/linux/moduleparam.h>
```

```
#define __module_param_call(prefix, name, ops, arg, isbool, perm)\
    /* Default value instead of permissions? */ \
    static int __param_perm_check_##name __attribute__((unused)) = \
        BUILD_BUG_ON_ZERO((perm) < 0 || (perm) > 0777 || ((perm) & 2)) \
        + BUILD_BUG_ON_ZERO(sizeof("")prefix > MAX_PARAM_PREFIX_LEN); \
    static const char __param_str_##name[] = prefix #name; \
    static struct kernel_param __moduleparam_const __param_##name \
        __used \
        __attribute__((unused, __section("__param"), aligned(sizeof(void *)))) \
    = { __param_str_##name, ops, perm, isbool ? KPARAM_ISBOOL : 0, \
        { arg } }

#define module_param_cb(name, ops, arg, perm) \
    __module_param_call(MODULE_PARAM_PREFIX, \
        name, ops, arg, __same_type((arg), bool *), perm)

#define __MODULE_PARM_TYPE(name, _type) \
    __MODULE_INFO(paramtype, name##type, #name ":", _type)
```

```

#define module_param_named(name, value, type, perm) \
    param_check_##type(name, &(value)); \
    module_param_cb(name, &param_ops_##type, &value, perm); \
    __MODULE_PARM_TYPE(name, #type)

#define module_param(name, type, perm) \
    module_param_named(name, name, type, perm)

```

基本上，上述的宏系列会在一个名为“__param”的 section¹⁰中定义一些变量。这段宏的定义细究起来可能稍嫌晦涩。这里不妨以本节开始的例子来展开该宏（除去一些跟调试相关的微末细节），以便使读者对 module_param 宏有一具体的印象，module_param(dolphin, int, 0)展开后如下：

```

param_check_int(dolphin, &(dolphin));
static int __param_perm_check_dolphin __attribute__((unused)) = \
static const char __param_str_dolphin[] = "dolphin "; \
static struct kernel_param __moduleparam_const __param_dolphin \
__used \
__attribute__((unused, __section__("__param"), aligned(sizeof(void *)))) \
= { __param_str_dolphin, &param_ops_int, 0, 0, \
    { &dolphin } }

```

可见 module_param(dolphin, int, 0)在“__param”section 中定义了一个类型为 struct kernel_param 的静态常量。struct kernel_param 的定义如下：

```

<include/linux/moduleparam.h>
-----
struct kernel_param {
    const char *name;
    const struct kernel_param_ops *ops;
    u16 perm;
    u16 flags;
    union {
        void *arg;
        const struct kparam_string *str;
        const struct kparam_array *arr;
    };
};

```

其中 name 为参数名，perm 为对 sysfs 文件系统中模块参数的访问许可，定义在结构体 struct kernel_param_ops 对象 ops 中的成员函数（set 和 get）用来在模块 mod 的 args 成员和模块

¹⁰ 如同模块导出的符号所在的 section 一样，模块参数所在的 section 也是一个 optional 的 section：如果模块没有用 module_param 相关宏声明任何参数变量，那么最终的 ELF 文件将不会生成对应的 section。

的参数 section 间拷贝数据，最后的 union 为指向参数的指针。

__used 和 unused 主要用来避免编译器产生警告信息，因为此处声明的 __param_dolphin 变量在模块源码的其他部分并不会被使用。

param_check_int 宏用来检测变量 dolphin 在 module_param 宏之前是否定义，因为 struct kernel_param 中的 union 联合体只是用来放置模块使用的参数所在地址，如果之前该参数没有定义，就不可能生成 &dolphin 的值。所以我们的内核模块示例源代码 demodev.c 在用 module_param 声明模块参数之前，要首先定义出这些参数。

```
int dolphin; //首先定义
module_param(dolphin, int, 0); //然后再声明模块参数
```

如果在设备驱动程序中忘了先定义参数变量 dolphin，只用 module_param(dolphin, int, 0) 来声明模块参数，将会得到如下编译错误：

```
root@AMDLinuxFGL:/home/dennis/Linux/book/chap01# make
make -C /lib/modules/2.6.39/build M=/home/dennis/Linux/book/chap01 modules
make[1]: Entering directory `/home/dennis/Linux/kernel/linux-2.6.39'
CC [M] /home/dennis/Linux/book/chap01/demodev.o
/home/dennis/Linux/book/chap01/demodev.c: In function '__check_dolphin':
/home/dennis/Linux/book/chap01/demodev.c:5: error: 'dolphin' undeclared (first use in this function)
/home/dennis/Linux/book/chap01/demodev.c:5: error: (Each undeclared identifier is reported only
once
/home/dennis/Linux/book/chap01/demodev.c:5: error: for each function it appears in.)
/home/dennis/Linux/book/chap01/demodev.c: At top level:
/home/dennis/Linux/book/chap01/demodev.c:5: error: 'dolphin' undeclared here (not in a function)
/home/dennis/Linux/book/chap01/demodev.c:5: warning: type defaults to 'int' in declaration of 'type
name'
make[2]: *** [/home/dennis/Linux/book/chap01/demodev.o] Error 1
make[1]: *** [_module_/home/dennis/Linux/book/chap01] Error 2
make[1]: Leaving directory `/home/dennis/Linux/kernel/linux-2.6.39'
make: *** [default] Error 2
```

在上面的宏展开的实例中，可以看到指向参数的指针值被设定为 “{ &dolphin }”，在模块静态链接期间，&dolphin 指令不可能生成其最终的运行期地址，因此模块参数所在的 “__param” section 需要有一个对应的 relocation section “.rel__param”，用来完成对参数指针的重定位，这样才能把命令行中的参数值正确复制到模块的 “__param” section 中。

除了 module_param 之外，Linux 系统下还有另外两个宏 module_param_array 和

`module_param_string`，分别用来设定数组型和字符串型参数，本书不再赘述。

下面讨论“`insmod demodev.ko dolphin=10 bobcat=5`”中携带的参数值如何为模块所用，不看代码也应该可以猜想出命令行中的参数值应该会被复制到模块的参数中，这样模块在开始使用参数 `dolphin` 之前，其值已经被 `insmod` 命令行中的实际值所改写。图 1-7 展示了命令行参数传递到模块的“`__param`” section 的全过程：

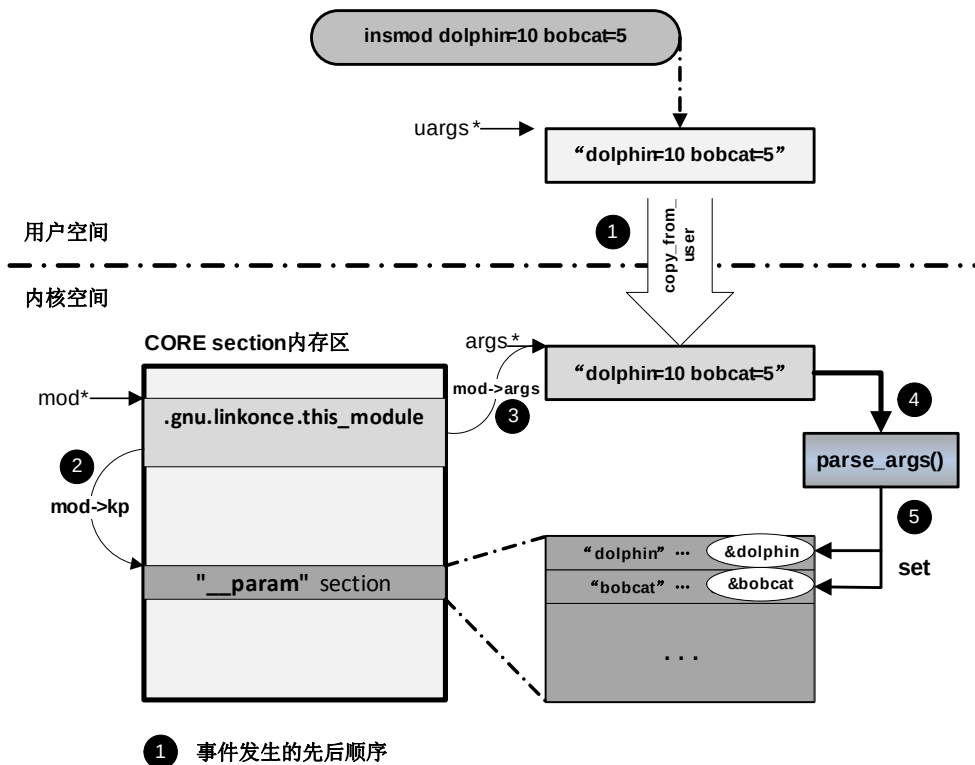


图 1-7 内核模块参数传递过程示意图

在实际的内核源代码中，`sys_init_module` 函数的最后一个参数 `const char __user *uargs` 清楚地表明这是由用户空间传递过来的放置模块参数的内存地址，在 `insmod` 一个模块时所携带的参数将以字符串的形式向内核空间传递。然后在 `load_module` 函数中，通过 `strndup_user` 的调用将用户空间的模块参数复制到内核空间。

```
args = strndup_user(uargs, ~0UL >> 1);
```

`strndup_user` 函数内部会调用 `kmalloc` 为在内核空间保存模块参数字符串分配一段内存区域，然后通过 `copy_from_user` 将模块参数从用户空间复制到内核空间。

接着，在 HDR 视图的 section 被搬移到 CORE 和 INIT section 之后，`load_module` 通过下面的 `section_objs` 函数调用取得“`__param`” section 在内存空间的最终地址，并记录在 `struct module` 的 `struct kernel_param *kp` 成员变量中。

```
mod->kp = section_objs(hdr, sechdrs, secstrings, "__param", sizeof(*mod->kp), &mod->num_kp);
```

mod->num_kp 为 “__param” section 中 struct kernel_param 对象的个数。此后 mod->args 参数也将指向内核空间中保留参数字符串的内存区域。

最后调用 parse_args 函数将 mod->args 中的参数值复制到 “__param” section 中对应的参数。

```
parse_args(mod->name, mod->args, mod->kp, mod->num_kp, NULL);
```

parse_args 函数的主要流程是，针对命令行中出现的每一个参数，用其参数名与 “__param” section 中出现的每一个 struct kernel_param 对象的 name 成员进行匹配，如果匹配成功即认为找到了对应的参数，然后通过 struct kernel_param 中的 set 函数指针将参数值复制到 struct kernel_param 中 arg、str 或者 arr 所指向的地址空间。对于本节开始的例子，set 指针指向 param_set_int 函数，后者在 Linux 内核源码中由 STANDARD_PARAM_DEF 宏 (kernel/params.c) 负责定义，展开后的 param_set_int 如下：

```
int param_set_int(const char *val, struct kernel_param *kp)
{
    long l;
    int ret;

    if (!val) return -EINVAL;
    ret = strict_strtol (val, 0, &l);
    if (ret == -EINVAL || ((int)l != l))
        return -EINVAL;
    *((int *)kp->arg) = l; //将命令行中的参数值复制到 “__param” section 的 kp 中
    return 0;
}
```

param_set_int 函数调用之后，模块 “__param” section 中 “dolphin” 和 “bobcat” 所对应的 struct kernel_param 对象中的 arg 成员将被赋值为 10 和 5，也就是在 insmod 命令行中传入的模块实际参数值。这之后，内核模块将可以通过自身的 dolphin 和 bobcat 变量引用到这些传入的参数值。

如果将这一过程简单总结一下的话，应该是在把命令行的参数值复制到模块的参数这个过程中，module_param 宏所定义的 “__param” section 起了桥梁的作用，通过 “__param” section，内核可以找到模块中定义的参数所在的内存地址，继而可以用命令行中的参数值改写之。因为内核模块加载器在解析命令行参数时，对命令行中参数的构成格式有严格的要求，在参数名与参数值之间只能用 “=”，且不能有空格，如果把前面的命令行写成如下形式 (“dolphin=” 和 “10” 之间有个空格)：

```
insmod demodev.ko dolphin= 10 bobcat=5
```

那么将会得到如下信息：

```
insmod: error inserting 'demodev.ko': -1 Unknown symbol in module
```

dmesg 中针对这个错误的输出如下：

```
[262282.558333] demodev: Unknown parameter `dolphin'
```

至此，我们已经理解了内核模块的参数机制，包括模块源码中如何声明参数，内核模块加载时如何把命令行中的参数复制到模块中等。

○ 模块间的依赖关系

实际运行的系统中并不是只加载一个模块，模块可以随时添加进系统，也可以随时被卸载。这些内核模块之间并不是完全相对独立的，比如当一个模块引用到另一个模块中导出的符号时，这两个模块间就建立了依赖关系。因此依赖关系只存在于模块与模块之间，模块与内核之间不构成依赖关系，因为在模块生存期间我们不可能去卸载内核，当然内核也不可能引用到模块导出的符号。内核必须能跟踪模块间的这种依赖关系，只有这样，如果由于存在依赖关系卸载一个模块有可能影响到系统的稳定性，内核才可能采取必要的措施防止这种情况发生。

内核用 `struct module` 数据结构中定义的如下成员变量来跟踪模块间的这种依赖关系：

```
<include/linux/module.h>
-----
#ifdef CONFIG_MODULE_UNLOAD
    /* What modules depend on me? */
    struct list_head source_list;
    /* What modules do I depend on? */
    struct list_head target_list;

#endif
```

其中的 `struct list_head source_list` 和 `struct list_head target_list` 用来构建有依赖关系模块的链表，对该链表的使用要结合数据结构 `struct module_use`：

```
<include/linux/module.h>
-----
struct module_use {
    struct list_head source_list;
    struct list_head target_list;
    struct module *source, *target;
};
```

显然，模块间的这种依赖关系只有当模块能够被卸载时才有可能出现问题，对一个没有启用 `CONFIG_MODULE_UNLOAD` 宏的系统而言，因为模块被禁止卸载，因此内核无须对依赖关系做出处理。

模块的依赖关系的建立最早发生在当前模块对象 `mod` 被加载时，模块加载函数调用

resolve_symbol 函数来解决其中一些“未解决的引用”符号。如果成功地在其他模块导出的符号中找到了指定的符号，那么 resolve_symbol 函数会将导出这一“未解决的引用”符号的模块记录在一个变量 struct module *owner 中，然后调用 ref_module(mod, owner) 在模块 mod 和 owner 之间建立依赖关系。ref_module 函数在做一些必要的安全性检查之后调用 add_module_usage(mod, owner) 在 mod 和 owner 模块间建立依赖关系，add_module_usage 函数的定义如下：

<kernel/module.c>

```
static int add_module_usage(struct module *a, struct module *b)
{
    struct module_use *use;

    DEBUGP("Allocating new usage for %s.\n", a->name);
    use = kmalloc(sizeof(*use), GFP_ATOMIC);
    if (!use) {
        printk(KERN_WARNING "%s: out of memory loading\n", a->name);
        return -ENOMEM;
    }

    use->source = a;
    use->target = b;
    list_add(&use->source_list, &b->source_list);
    list_add(&use->target_list, &a->target_list);
    return 0;
}
```

函数首先调用 kmalloc 分配一 struct module_use 型内存空间 use，然后将 use 中的 source 指向 mod 模块，target 指向 owner 模块，同时将 use 的 target_list 加入 mod 中的 target_list 指向的双向链表，将 use 的 source_list 加入 owner 中的 source_list 指向的双向链表。图 1-8 展示了通过 struct module_use 对象在三个模块间建立依赖关系的技术细节：

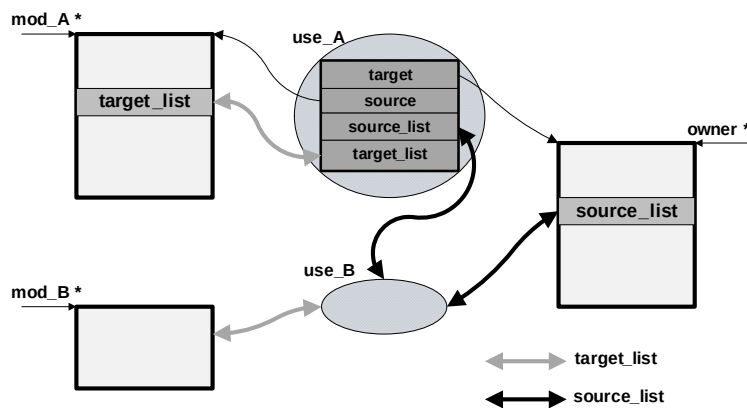


图 1-8 mod_A 和 mod_B 模块依赖于 owner 模块

图 1-8 中，模块 `mod_A` 和 `mod_B` 均依赖于模块 `owner`，`mod_A` 和 `mod_B` 之间则没有依赖关系。`owner` 模块先加入系统，它导出一个函数为模块 `mod_A` 和 `mod_B` 所使用。图中显示 `mod_A` 先于 `mod_B` 加入系统，在 `mod_A` 加入系统时，模块加载器创建了 `use_A` 对象在 `mod_A` 和 `owner` 间建立依赖关系。随后 `mod_B` 加入了系统，因为它和 `owner` 模块间存在依赖关系，加载器同样创建了 `use_B` 对象在 `mod_B` 和 `owner` 间建立关联。从图中可以看到，`use_A` 对象中的 `source_list` 成员已不再指向 `owner->source_list`，而是指向了 `use_B->source_list`，后者则和 `owner->source_list` 建立了直接的链接关系。

如此，`mod_A` 和 `mod_B` 模块可以通过遍历其 `target_list` 成员知道所依赖的所有模块，而 `owner` 模块则可以通过遍历其 `source_list` 成员知道所有依赖于自己的模块。

当从系统中卸载一个模块时，系统必须确保没有其他模块依赖于该模块，根据上面的讨论，只要模块结构中的 `source_list` 是一空链表，就表明没有其他模块依赖于它。相关代码在模块卸载部分讨论。

○ 模块的版本控制

版本控制主要用来解决内核模块和内核之间的接口一致性问题。所谓内核模块和内核之间的接口，简单地说是指出由内核导出并被内核模块调用的那些符号。产生这种问题的根源在于内核模块和内核作为独立实体各自分开编译。想象一下，一个在 Linux 2.6.18 内核源码树基础上编译出来的内核模块能否成功加载到内核版本号为 2.6.35 的 Linux 系统中？如果模块使用的一个接口在 2.6.35 版本中已被改变或者废弃，那么前者将会因为无法找到一个未经定义的符号而导致加载失败，后者虽然有可能加载成功，但因为使用到了一个内核已经废弃的接口而需承担相应的风险。

因此，内核和模块之间必须协商出一种机制确保上述问题不会出现。Linux 系统对此的解决方案是使用接口的校验和，也叫接口 CRC 校验码。这种方法的基本思想非常简单，根据函数的参数生成一个大小为 4 字节的 CRC 校验码，当双方校验码相等时视为相同接口，否则为不同接口。

为了确保这种机制能够正常工作，内核必须首先启用 `CONFIG_MODVERSIONS` 这个宏，在此基础上内核模块在编译时也必须启用 `CONFIG_MODVERSIONS`，否则模块将会因为出现无法解决的未定义符号错误导致加载失败。显然这是个需要双方共同协作才能解决的问题。如果内核编译时没有启用 `CONFIG_MODVERSIONS`，那么系统将不会启用本节所说的方案，即使被加载的模块是在另一个启用 `CONFIG_MODVERSIONS` 的内核源码树的基础上编译出来的。

下面首先从内核的角度来看看 `CONFIG_MODVERSIONS` 宏对内核导出符号产生的影响。在前面“`EXPORT_SYMBOL` 的内核实现”一节中我们看到了内核用于导出符号的 `EXPORT_SYMBOL` 相关宏的定义，其中出现了一个 `__CRC_SYMBOL` 宏，如果在内核编

译时启用了 `CONFIG_MODVERSIONS`，即对内核启用了版本控制特性，那么 `__CRC_SYMBOL` 的定义为：

```
#define __CRC_SYMBOL(sym, sec) \
    extern void *__crc_##sym __attribute__((weak)); \
    static const unsigned long __kcrctab_##sym \
    __used \
    __attribute__((section("__kcrctab" sec), unused)) \
    = (unsigned long) &__crc_##sym;
```

如此，对于 `EXPORT_SYMBOL(my_exp_function)` 的例子，将会在原来的基础上新增如下的定义：

```
extern void *__crc_my_exp_function;
static const unsigned long __kcrctab_my_exp_function = (unsigned long) &__crc_my_exp_function;
```

`__kcrctab_my_exp_function` 用来保存 `__crc_my_exp_function` 变量地址并将其放在一个名为“`__kcrctab`”的 section 中（对于 `EXPORT_SYMBOL_GPL` 和 `EXPORT_SYMBOL_GPL_FUTURE`，其所在的 section 的名称分别为“`__kcrctab_gpl`”和“`__kcrctab_gpl_future`”）。

由此可见，如果内核编译时启用了 `CONFIG_MODVERSIONS` 宏，那么对于每一个导出的符号，都会生成一个对应的 CRC 校验码。反之，如果内核编译时没有启用 `CONFIG_MODVERSIONS` 宏，那么系统将不会为导出的符号产生 CRC 校验码。“`__kcrctab`”等 section 存在的意义在于当符号查找时，可以通过该 section 找到对应符号的校验码，以此来判断模块所使用的接口是否和当前正运行的内核提供的接口相匹配。

`CONFIG_MODVERSIONS` 宏对内核的另一个影响存在于加载内核模块的过程中。前面在“对‘未解决的引用’符号（unresolved symbol）的处理”部分中提到，对于模块中出现的未定义的符号，内核会调用 `resolve_symbol` 函数予以解决，在 `resolve_symbol` 函数的内部会调用 `find_symbol` 来查找该符号。如果成功查找到，则函数接下来会调用 `check_version` 对这种接口进行校验码的验证。在没有启用 `CONFIG_MODVERSIONS` 的系统中，这个函数直接返回 1，因此没有启用 `CONFIG_MODVERSIONS` 的内核不会进行接口一致性的检验。

接下来看一下 `CONFIG_MODVERSIONS` 对内核模块的影响。

首先，前面提到的 `CONFIG_MODVERSIONS` 宏对内核导出符号的影响同样适用于模块导出的符号。

其次，启用 `CONFIG_MODVERSIONS` 会导致模块的编译工具链在模块最终的 ELF 文件中产生一个名为“`__versions`”的 section¹¹，打开模块源码所在目录下的 `.mod.c` 文件，会发现

¹¹ “`__versions`” section 是由 `scripts/mod/modpost.c` 生成的，为叙述简单起见，本书将其统称为工具链。

类似如下代码：

```
static const struct modversion_info ____versions[]
__used
__attribute__((section("__versions"))) = {
    { 0x58334a4a, "module_layout"12 },
    { 0x6980fe91, "param_get_int" },
    { 0xff964b25, "param_set_int" },
    { 0xb72397d5, "printk" },
    { 0xb4390f9a, "mcount" },
};
```

代码定义了一个类型为 `struct modversion_info` 的数组 `____versions`，放在“`__versions`”section 中。`struct modversion_info` 的定义如下：

```
<include/linux/module.h>
struct modversion_info
{
    unsigned long crc;
    char name[MODULE_NAME_LEN];
};
```

可见当编译内核模块时，若对应的内核配置文件中启用了 `CONFIG_MODVERSIONS`，则模块最终的 ELF 文件中会生成一个“`__versions`”section，该 section 会将模块中的所有“未解决的引用”符号名和对应的校验码放入其间。在前面的那个 `.mod.c` 的示例中，`printk` 作为模块的一个“未解决的引用”符号被放在了“`__versions`”section 中，工具链为其产生的 CRC 校验码为 `0xb72397d5`。

在分析完启用 `CONFIG_MODVERSIONS` 宏对内核和内核模块双方的影响后，再回过头来看看当模块加载时处理“未解决的引用”符号时是如何对接接口一致性进行验证的。这种验证是通过在 `resolve_symbol` 函数里调用 `check_version` 函数完成的，`check_version` 函数的完整定义如下：

```
<kernel/module.c>
-----
#ifdef CONFIG_MODVERSIONS
static int check_version(Elf_Shdr *sechdrs,
                        unsigned int versindex,
                        const char *symname,
                        struct module *mod,
                        const unsigned long *crc,
                        const struct module *crc_owner)
```

¹² 2.6.35 及以后版本的内核会为启用 `CONFIG_MODVERSIONS` 的模块生成一个名为“`module_layout`”的未定义符号，启用了 `CONFIG_MODVERSIONS` 的 2.6.35 版本内核在加载时会检查“`module_layout`”符号和对应的 CRC 校验码。


```
{
    unsigned int i, num_versions;
    struct modversion_info *versions;

    /* Exporting module didn't supply crcs? OK, we're already tainted. */
    if (!crc)
        return 1;

    /* No versions at all? modprobe --force does this. */
    if (versindex == 0)
        return try_to_force_load(mod, symname) == 0;

    versions = (void *) sechdrs[versindex].sh_addr;
    num_versions = sechdrs[versindex].sh_size
        / sizeof(struct modversion_info);

    for (i = 0; i < num_versions; i++) {
        if (strcmp(versions[i].name, symname) != 0)
            continue;

        if (versions[i].crc == maybe_relocated(*crc, crc_owner))
            return 1;
        DEBUGP("Found checksum %lX vs module %lX\n",
            maybe_relocated(*crc, crc_owner), versions[i].crc);
        goto bad_version;
    }

    printk(KERN_WARNING "%s: no symbol version for %s\n",
        mod->name, symname);
    return 0;

bad_version:
    printk("%s: disagrees about version of symbol %s\n",
        mod->name, symname);
    return 0;
}

#else
static inline int check_version(Elf_Shdr *sechdrs,
                                unsigned int versindex,
                                const char *symname,
                                struct module *mod,
                                const unsigned long *crc,
                                const struct module *crc_owner)
{
    return 1;
}
```

```
#endif
```

在定义了 CONFIG_MODVERSIONS 的前提下, check_version 用一个 for 循环在“__versions” section 中进行遍历, 对每一个 struct modversion_info 元素和找到的符号名 symname 进行匹配, 如果匹配成功, 再进行接口的校验码比较, 如果校验码相等, 说明模块所使用的接口和内核导出的接口是一致的, 否则产生版本不匹配的错误。

关于校验码的计算¹³, 内核源码树中提供了一个产生 CRC 校验码的工具 genksyms, 它位于 Linux 内核源码的 scripts/genksyms 目录下。内核模块编译工具链通过它来生成导出符号的 CRC 校验码。比如下面的 demodev.h 文件, 其中导出了一个函数符号 my_exp_function:

```
<demodev.h>
-----
int my_exp_function(int a, int b);
EXPORT_SYMBOL(my_exp_function);
```

那么使用如下命令就可以看到 genksyms 为 my_exp_function 生成的 CRC 校验和:

```
root@AMDLinuxFGL:/home/dennis/book# gcc -E demodev.h -D__GENKSYMS__ -D__KERNEL__
| ./genksyms > demodev.crc
```

打开 demodev.crc 文件, 可以看到类似下面的内容:

```
__crc_my_exp_function = 0x48a0010f;
```

genksyms 只为由 EXPORT_SYMBOL 系列的宏导出符号生成 CRC 校验码。对于内核和内核模块而言, 它们在各自编译链接阶段均独立使用 genksyms 来为导出的符号和那些“未解决的引用”符号生成校验和。

图 1-9 展示了接口校验码在验证接口有效性时的一个具体例子, 图中的模块 A 使用到了内核导出的一个函数 demofunc, 该模块当前的.ko 文件最初是在内核源码树的 2.6.18 版本上编译而成的, 现在要在运行内核 B (版本号为 2.6.35) 的 Linux 系统上加载该模块。现在的问题是, 内核导出的函数 demofunc 在从版本 2.6.18 到 2.6.35 的演变过程中发生了改变, 新版本内核源码中该函数原型较旧版本多出了一个参数。如果内核在加载模块时没有版本控制机制, 那么在运行新版本的 Linux 系统中加载该模块时会发生什么呢? 答案是结果不确定, 内核也许会正常运行, 也许会崩溃, 但无论如何这是个潜在的危险行为。

如果内核 A 和 B 在当初配置时都启用了版本控制机制, 那么它们除了导出 demofunc 这个符号外, 还会分别为这个接口生成对应的 CRC 校验码。当图中的模块 A 在内核 A 源码树的基础上进行编译时, 模块的构造工具链也会负责为模块中这个“未解决的引用”符号 demofunc 生成 CRC 校验码, 因为模块编链工具链和内核工具链使用的 CRC 校验码生成工

¹³ 关于校验码生成算法, 本书不作详细讨论, 读者可以简单认为: $V_{CRC}=f(\text{导出函数名, 函数的参数表})$ 。

具是完全一样的，所以它们都会获得针对 demofunc 函数接口的一个 CRC 校验码，图中的值为 0xa206cef4。

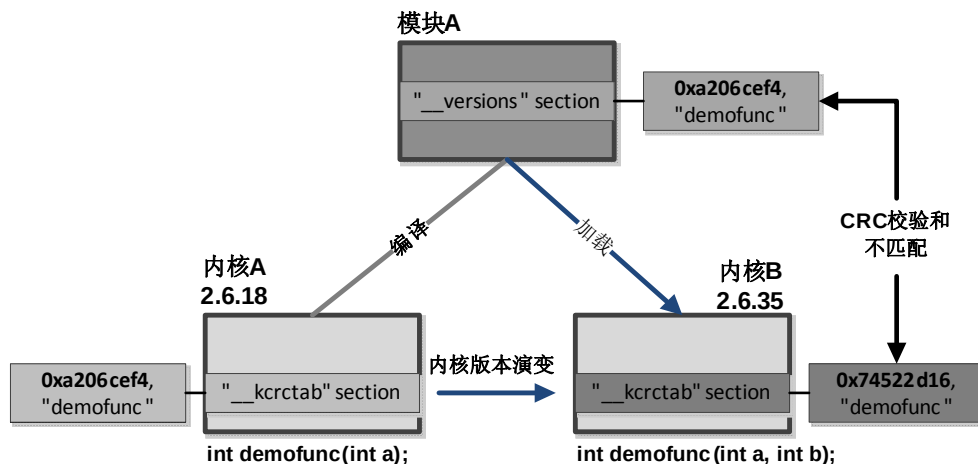


图 1-9 新旧内核导出的接口 CRC 不匹配

而对于图中的内核 B 而言，虽然使用了同样的 CRC 校验码生成工具，但因为函数多出了一个参数，所以内核 B 得到的 demofunc 函数接口校验码为 0x74522d16。这样，当将图中的模块 A 向内核 B 加载时，会因为两者的 CRC 校验码不匹配而导致无法加载该模块，从而避免可能造成内核不稳定的危险行为。

基于上面的讨论，我们建议在对内核进行配置时启用 CONFIG_MODVERSIONS 选项，这样编译出的内核在模块加载时就会启动版本控制机制。如果模块在一个没有启用 CONFIG_MODVERSIONS 宏的内核源码树基础上进行编译链接，模块的构造工具将不会为模块中导出的符号和那些“未解决的引用”符号生成 CRC 校验码，如果将这样的模块安装到一个启用了 CONFIG_MODVERSIONS 的 Linux 系统中，加载该模块就会在版本控制环节出现问题，这种情况下即使强行加载也会导致内核的污染。

最后要强调的是，对于 CONFIG_MODVERSIONS 机制，模块的编译工具链只对导出的符号产生 CRC 校验码，最明显的，内核在编译过程中所有导出的符号都会被记录到一个名为“Module.symvers”的文件中，下面是该文件的一部分：

```

root@AMDLinuxFGL:/home/dennis/Linux/kernel/linux-2.6.39# cat Module.symvers | grep printk
0xc60f75ec __ftrace_vprintk vmlinux EXPORT_SYMBOL_GPL
0x5ebefe4b v4l_printk_ioctl drivers/media/video/videodev EXPORT_SYMBOL
0xbdd295f0 trace_vprintk vmlinux EXPORT_SYMBOL_GPL
0x50eedeb8 printk vmlinux EXPORT_SYMBOL
  
```

其中每行第二列就是导出的符号，第一列是该导出符号的 CRC 校验码，第三列是导出该符

号的模块，第四列是导出符号的类型。

如果一个独立编译的内核模块，比如 `demodev.ko`，引用到了内核导出的符号，比如 `printk`，那么在 `demodev.ko` 的编译链接过程中，工具链会到内核源码所在的目录下查找 `Module.symvers` 文件，将得到的 `printk` 的 CRC 校验码记录到 `demodev.ko` 的 “__versions” section 中，换句话说，工具链不会在使用导出符号的地方重新为之生成一个 CRC 校验码。在 `demodev` 模块成功编译后，读者可以在其源码所在的目录下发现一个名为 “`demodev.mod.c`” 的文件，在该文件中可以发现类似下面的内容：

```
<demodev.mod.c>
-----
static const struct modversion_info __versions[]
__used
__attribute__((section("__versions"))) = {
    { 0xe1c343b8, "module_layout" },
    { 0x50eedeb8, "printk" },
    { 0xb4390f9a, "mcount" },
};
```

从中可以看到 `printk` 函数的校验码和内核源码树中 `Module.symvers` 中的完全一样。

从这里还可以引申出一个有趣的问题，如果有两个模块 A 和 B，A 导出的一个符号 “`a_sym`” 为 B 所用，因为 A 和 B 都是各自独立编译链接，意味着 A 导出的符号及其 CRC 校验码将放在 A 所在目录的 `Module.symvers` 文件中，这样在编译链接 B 模块时将会产生一个 WARNING，大意是 “`a_sym` [/home/dennis/Linux/B.ko] undefined!”，即便在 B 模块源码中加入 `extern ... a_sym` 这样的声明也无法消除这个 WARNING，产生该 WARNING 的原因是 B 模块只能找到 Linux 内核导出符号所在的 `Module.symvers` 文件，而无法找到 A 模块产生的 `Module.symvers` 文件，所以在模块编译阶段无法确定最终在模块加载时是否能找到这个 “`a_sym`” 符号，因此只是简单地给了个 WARNING。从表象上看，这个 WARNING 尚不是致命的，因为还有模块加载器这最后一道防线，正如前面所讨论的，它会到内核及所有加载进系统的内核模块所导出的符号表中查找 “`a_sym`”，假如在 B 模块加载前 A 模块已经被加载进系统，那么有理由相信模块加载器是可以帮 B 模块找到 “`a_sym`” 符号的，但是遗憾的是模块 B 通常都不可能成功被加载，`dmesg` 对此给出的提示信息是 “B: no symbol version for `a_sym`” 云云。所以此时再去查看 B 模块的 `.mod.c` 文件，在它的 `versions[]` 里一定不会有 “`a_sym`” 的身影，因为工具链根本得不到它的 CRC 校验码。知道了原因问题就好解决了，最简单的，把 A 模块 `Module.symvers` 文件的内容添加到 Linux 内核源码树中的 `Module.symvers` 文件中，这样就可以消除 WARNING 而且 B 模块也可以成功加载。如果此时再看 B 模块的 `.mod.c` 文件，就会发现 “`a_sym`” 已经出现在 `versions[]` 中，并且 `modinfo` 显示 B 模块有了个依赖关系 `depends`：

```
root@AMDLinuxFGL:/home/dennis/Linux/book/chap09/framework/device#modinfo B.ko
```

```
filename: B.ko
description: A simple kernel module
author: dennis chen @ AMDLinuxFGL
license: GPL
srcversion: 7093EDF7B908CDD5212F5F1
depends: A
vermagic: 2.6.39 SMP mod_unload modversions 586
```

○ 模块的信息 (modinfo)

模块的最终 ELF 文件中都会有一个名为 “.modinfo” 的 section，这个 section 以文本的形式保留着模块的一些相关信息。在 Linux 环境下可以用 modinfo 工具来查看一个模块存储的信息，比如下面 modinfo 输出的 demodev.ko 模块的信息：

```
root@AMDLinuxFGL:/home/dennis/Linux/book/chap01# modinfo demodev.ko
filename: demodev.ko
srcversion: D2FF50581DA6C0AF3F6B3EC
depends:
vermagic: 2.6.39 SMP mod_unload modversions 686
parm:dolphin:int
parm:bobcat:int
```

模块的源码中用 MODULE_INFO 宏来向该 section 添加模块信息，MODULE_INFO 宏的相关定义如下：

```
<include/linux/module.h>
-----
#ifdef MODULE
#define __module_cat(a,b) __mod_## a ## b
#define __module_cat(a,b) __module_cat(a,b)
#define __MODULE_INFO(tag, name, info) \
static const char __module_cat(name,__LINE__)[ \
    __used \
    __attribute__((section(".modinfo"),unused)) = __stringify(tag) "=" info
#else /* !MODULE */
#define __MODULE_INFO(tag, name, info)
#endif
/* Generic info of form tag = "info" */
#define MODULE_INFO(tag, info) __MODULE_INFO(tag, tag, info)
```

在 MODULE 没有定义的情况下，MODULE_INFO 是个空定义。而对于内核模块而言，MODULE_INFO 在 “.modinfo” section 中定义了一个类似“tag=info”的字符串，内核中通过调用 get_modinfo 函数来获得 tag 所在字符串的值 info。

模块加载过程中，需要获得 “.modinfo” section 中的相关信息以便进一步处理，这些信息包括：

- 模块的 license

模块的 license 在模块源码中以 MODULE_LICENSE 宏引出，该宏的主体就是 MODULE_INFO：

```
#define MODULE_LICENSE(_license) MODULE_INFO(license, _license)
```

内核模块加载过程中，会调用 license_is_gpl_compatible 来确定模块的 license 是否与 GPL 兼容：

```
<include/linux/license.h>
-----
static inline int license_is_gpl_compatible(const char *license)
{
    return (strcmp(license, "GPL") == 0
        || strcmp(license, "GPL v2") == 0
        || strcmp(license, "GPL and additional rights") == 0
        || strcmp(license, "Dual BSD/GPL") == 0
        || strcmp(license, "Dual MIT/GPL") == 0
        || strcmp(license, "Dual MPL/GPL") == 0);
}
```

非以上形式的 license 被认为与 GPL 不兼容，这样的模块在加载进系统后会导致内核被污染，内核用 mod 对象的 unsigned int taints 成员记录一个模块是否会污染内核：

```
mod->taints |= (1U << 0);
```

这样，以后就可以通过 mod->taints 来判断要加载的模块是否 GPL 兼容。而对于运行中的系统是否被污染，内核用一个 unsigned long 型全局变量 tainted_mask 来表示，在系统因故障挂起时，tainted_mask 会影响系统调试信息的输出，以告之内核是否已被污染。

另外，non-GPL 的模块无法使用内核或其他内核模块用 EXPORT_SYMBOL_GPL 导出的符号，在加载这样的模块时将出现 "Unknown symbol in module" 类似的错误信息。

- 模块的 vermagic

内核和内核模块的 vermagic 都是通过 MODULE_INFO 定义的一个 VERMAGIC_STRING 字符串，后者实际上是一个生成字符串的宏，该宏会根据不同的内核配置信息生成不同的字符串。模块加载过程中会检查模块中的 vermagic 是否和当前运行的内核定义的 vermagic 一致，如果不一致加载将失败，dmesg 命令会发现类似下面的错误信息：

```
demodev: version magic '2.6.39 SMP mod_unload 586' should be '2.6.39 SMP mod_unload
modversions 586'
```

上面的信息输出对应着 `load_module` 函数如下的代码片段：

```
<kernel/module.c>
-----
static ninline struct module *load_module(void __user *umod,
                                           unsigned long len,
                                           const char __user *uargs)
{
    ...
    modmagic = get_modinfo(sechdrs, infoindex, "vermagic");
    /* This is allowed: modprobe --force will invalidate it. */
    if (!modmagic) {
        err = try_to_force_load(mod, "bad vermagic");
        if (err)
            goto free_hdr;
    } else if (!same_magic(modmagic, vermagic, versindex)) {
        printk(KERN_ERR "%s: version magic '%s' should be '%s'\n",
               mod->name, modmagic, vermagic);
        err = -ENOEXEC;
        goto free_hdr;
    }
    ...
}
```

该代码片段首先在当前模块的“`.modinfo`”section 中查找 `vermagic` 对应的字符串 `modmagic`，如果找到则和当前内核的 `vermagic` 字符串进行比较，以确定两者是否匹配，不匹配的话模块将加载失败。读者可以通过如下命令查看一个模块的“`.modinfo`”section 的内容：

```
root@AMDLinuxFGL:/home/dennis/Linux/book/chap01# readelf -p .modinfo demodev.ko
String dump of section '.modinfo':
[ 0] parmtype=bobcat:int
[ 14] parmtype=dolphin:int
[ 40] srcversion=D2FF50581DA6C0AF3F6B3EC
[ 63] depends=
[ 80] vermagic=2.6.39 SMP mod_unload modversions 686
```

所以对于 `demodev.ko` 这个模块而言，“`.modinfo`”section 中 `vermagic` 对应的字符串就为“2.6.39 SMP mod_unload modversions 686”，打开模块源码目录下的 `demodev.mod.c` 文件，可以看到下列信息：

```
MODULE_INFO(vermagic, VERMAGIC_STRING);
```

内核模块中用来产生 `vermagic` 的 `MODULE_INFO` 是通过 `scripts/mod/modpost.c` 文件自动生成的，内核模块开发者无须在源码中显式添加这一信息。

前面已经提到，VERMAGIC_STRING 实际上是由内核源码树的相关配置信息所组合而成的一个字符串，如下所示：

```
<include/linux/vermagic.h>
-----
#define VERMAGIC_STRING \
    UTS_RELEASE " " \
    MODULE_VERMAGIC_SMP MODULE_VERMAGIC_PREEMPT \
    MODULE_VERMAGIC_MODULE_UNLOAD MODULE_VERMAGIC_MODVERSIONS \
    MODULE_ARCH_VERMAGIC
```

读者很容易将 VERMAGIC_STRING 所组合的信息与"2.6.39 SMP mod_unload modversions 686"这样的具体字符串对应起来。因为不同的内核源码树的配置信息不一定相同，所以从这个角度而言，vermagic 也可以看做是模块版本控制的一部分。

1.3.4 sys_init_module (第二部分)

load_module 函数完成模块加载几乎所有的艰苦工作之后，重新返回到 sys_init_module,后者在 load_module 的基础上所做的事情就很简单了，主要有：

○ 调用模块的初始化函数

“struct module 类型变量 mod 初始化”部分中已经提到了 mod 中 init 函数指针是如何指向模块源码中的初始化函数，现在由 sys_init_module 负责调用它，相关代码如下：

```
<kernel/module.c>
-----
sys_init_module( void __user * umod, unsigned long len, const char __user * uargs)
{
    ...
    if (mod->init != NULL)
        ret = do_one_initcall(mod->init);
    if (ret < 0) {
        /* Init routine failed: abort. Try to protect us from
           buggy refcounters. */
        mod->state = MODULE_STATE_GOING;
        synchronize_sched();
        module_put(mod);
        blocking_notifier_call_chain(&module_notify_list,
                                    MODULE_STATE_GOING, mod);
        free_module(mod);
        wake_up(&module_wq);
        return ret;
    }
    ...
}
```


从以上代码可以看出，内核模块可以不提供模块初始化函数。如果模块提供了初始化函数，那么它将在 `do_one_initcall` 函数内部被调用。如果模块初始化函数被成功调用，那么模块就算是被加载进了系统，因此需要更新模块的状态为 `MODULE_STATE_LIVE`：

```
mod->state = MODULE_STATE_LIVE;
```

○ 释放 INIT section 所占用的空间

模块一旦被成功加载，HDR 视图和 INIT section 所占的内存区域将不再会被用到，因此需要释放它们。在 `sys_init_module` 函数中，释放 INIT section 所在内存区域由函数 `module_free` 完成，后者调用 `vfree` 来释放 INIT section 区域 (`mod->module_init`)：

```
module_free(mod, mod->module_init);
```

HDR 视图所占空间的释放实际上发生在 `load_module` 函数的最后部分：

```
<kernel/module.c>
-----
static noinline struct module *load_module(void __user *umod,
                                           unsigned long len,
                                           const char __user *uargs)
{
    ...
    vfree(hdr);
    ...
}
```

模块成功加载进系统之后，正在运行的内核和系统中所有加载的模块关系如图 1-10 所示：

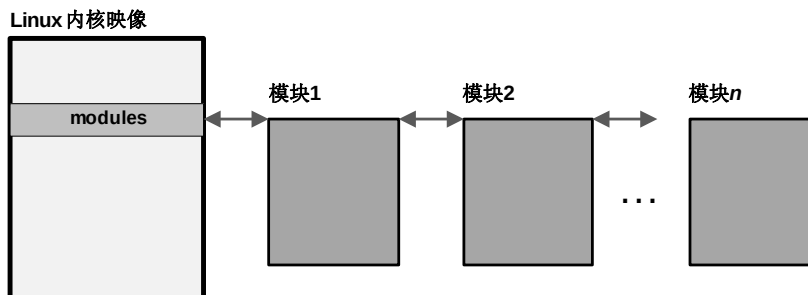


图 1-10 内核与内核模块

内核用一全局变量 `modules` 链表记录系统中所有已加载的模块。

○ 呼叫模块通知链

Linux 内核提供了一个很有趣的特性，也就是所谓的通知链（notifier call chain），这个特性不只是在模块的加载过程中会使用到，在内核系统的其他组件中也常常会使用到。通过通知链，模块或者其他的内核组件可以对向其感兴趣的一些内核事件进行注册，当该事件发

生时，这些模块或者组件当初注册的回调函数将会被调用。内核模块机制中实现的模块通知链 `module_notify_list` 就是内核中众多通知链中的一条。通知链背后的实现机制其实很简单，通过链表的形式，内核将那些注册进来的关注同类事件的节点构成一个链表，当某一特定的内核事件发生时，事件所属的内核组件负责遍历该通知链上的所有节点，调用节点上的回调函数。所有的通知链头部都是一个 `struct blocking_notifier_head` 类型的变量，该类型的定义为：

```
<include/linux/notifier.h>
-----
struct blocking_notifier_head {
    struct rw_semaphore rwsem;
    struct notifier_block *head;
};
```

这里以内核模块的通知链 `module_notify_list` 为例，来讨论一下通知链的实现原理。`module_notify_list` 为一全局变量，用来管理所有对内核模块事件感兴趣的通知节点，其定义为：

```
<kernel/module.c>
-----
static BLOCKING_NOTIFIER_HEAD(module_notify_list);
```

上述定义其实是定义并初始化了一个类型为 `struct blocking_notifier_head` 的对象 `module_notify_list`。如果一个内核模块想了解当前系统中所有与模块相关的事件，可以调用 `register_module_notifier` 向内核注册一个节点对象，该节点对象中包含有一个回调函数。当 `register_module_notifier` 函数成功向系统注册了一个回调节点之后，系统中所有那些模块相关的事件发生时都会调用到这个回调函数。为了具体了解幕后的机制，下面先来看看 `register_module_notifier` 函数在内核源码中的实现：

```
<kernel/module.c>
-----
int register_module_notifier(struct notifier_block * nb)
{
    return blocking_notifier_chain_register(&module_notify_list, nb);
}
```

函数的参数是个 `struct notifier_block` 型指针，代表一个通知节点对象。`struct notifier_block` 的定义为：

```
<include/linux/notifier.h>
-----
struct notifier_block {
    int (*notifier_call)(struct notifier_block *, unsigned long, void *);
    struct notifier_block *next;
    int priority;
};
```

其中，`notifier_call` 就是所谓的通知节点中的回调函数，`next` 用来构成通知链，`priority` 代表

一个通知节点优先级，用来决定通知节点在通知链中的先后顺序，数值越大代表优先级越高。

`blocking_notifier_chain_register` 最终通过调用 `notifier_chain_register` 函数将一个通知节点加入 `module_notify_list` 管理的链表。在向一个通知链中加入新节点时，系统会把各节点的 `priority` 作为一个排序关键字进行简单排序，其结果是越高优先级的节点越靠近头节点，当有事件发生时，最先被通知。图 1-11 展示了多个模块在同一条通知链上调用了 `register_module_notifier`，由此形成的该调用链结构示意图：

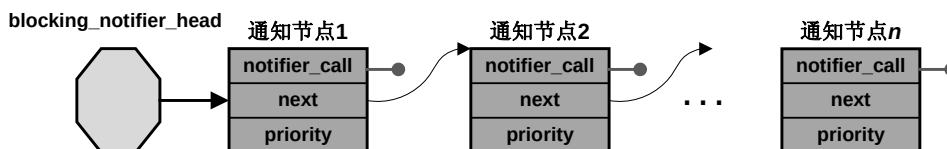


图 1-11 内核中的一条通知链构成示意图

与 `register_module_notifier` 相反，如果要从一条通知链中注销一个通知节点，那么应该使用 `unregister_module_notifier` 函数，该函数的原型为：

```
int unregister_module_notifier(struct notifier_block * nb);
```

以上讨论了如何向/从系统中的一条通知链注册/注销一个通知节点，接下来看看当某个特定的内核事件发生时，通知链上各节点的回调函数如何被触发。以内核模块加载过程为例，`sys_init_module` 函数在调用完 `load_module` 之后，会通过 `blocking_notifier_call_chain` 函数来通知调用链 `module_notify_list` 上的各节点，例如，如果 `load_module` 函数成功返回，表明模块加载的大部分工作已经完成，此时 `sys_init_module` 会通过调用 `blocking_notifier_call_chain` 函数来通知 `module_notify_list` 上的节点，一个模块正在被加入系统 (`MODULE_STATE_COMING`)：

<kernel/module.c>

```

sys_init_module( void __user * umod, unsigned long len, const char __user * uargs)
{
    ...
    mod = load_module(umod, len, uargs);
    if (IS_ERR(mod))
        return PTR_ERR(mod);

    blocking_notifier_call_chain(&module_notify_list, MODULE_STATE_COMING, mod);
    ...
}

```

上面代码段中的 `blocking_notifier_call_chain` 函数将使通知链 `module_notify_list` 上的各节点的回调函数均被调用，其实现原理可以简单概括为遍历 `module_notify_list` 上的各节点，依

次调用各节点上的 `notifier_call` 函数。读者从源码中也可以发现，对于模块加载的其他阶段（`MODULE_STATE_LIVE` 和 `MODULE_STATE_GOING`），内核模块加载器也都会调用 `blocking_notifier_call_chain` 函数来通知 `module_notify_list` 上的各个通知节点。

最后用一个实际的例子来加深读者对模块通知链的理解，下面所列内核模块 `modnoti.ko` 的源码展示了如何利用模块通知链来监听系统中与模块相关的事件：

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

static struct notifier_block *pnb = NULL;

static char *mstate[] = {"LIVE", "COMING", "GOING"};

//通知节点对象 pnb 上的回调函数
int get_notify(struct notifier_block *p, unsigned long v, void *m)
{
    printk("module <%s> is %s, p->priority=%d\n", ((struct module *)m)->name, mstate[v],
           p->priority);
    return 0;
}

static int hello_init(void)
{
    //分配一个 struct notifier_block 通知节点对象
    pnb = kzalloc(sizeof(struct notifier_block), GFP_KERNEL);
    if(!pnb)
        return -1;
    //通知节点上的回调函数
    pnb->notifier_call = get_notify;
    pnb->priority = 10;
    register_module_notifier(pnb);
    printk("A listening module is coming...\n");
    return 0;
}

static void hello_exit(void)
{
    unregister_module_notifier(pnb);
    kfree(pnb);
    printk("A listening module is going\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

当通过“insmod modnoti.ko”把该内核模块加入系统后，在 dmesg 的输出中可以发现如下的输出信息：

```
root@AMDLinuxFGL:/home/dennis/book/gene-module# dmesg
[230330.738545] A listening module is coming...
[230330.738555] module <modnoti> is LIVE, p->priority=10
```

因为 modnoti.ko 中向通知链 module_notify_list 注册一个节点的动作发生在模块的初始化函数中，所以虽然 sys_init_module 函数在模块初始化函数前调用了 blocking_notifier_call_chain(&module_notify_list, MODULE_STATE_COMING, mod)，但是此时 modnoti 模块的通知节点还没有出现在 module_notify_list 通知链中，因而 modnoti 只能接收到发生在模块初始化函数之后的 blocking_notifier_call_chain(&module_notify_list, MODULE_STATE_LIVE, mod)的通知消息。

在成功加载完 modnoti.ko 模块后，再通过“insmod demodev.ko”来加载另一个内核模块，此时 dmesg 的输出又多出了如下两行：

```
[230357.414452] module <demodev> is COMING, p->priority=10
[230357.414467] module <demodev> is LIVE, p->priority=10
```

对比 sys_init_module 函数源码，读者应该很容易理解为何出现上述两行输出。

1.3.5 模块的卸载

相对于模块的加载，从系统中卸载一个模块的任务则要轻松得多。将一个模块从系统中卸载，使用 rmmod 命令，比如“rmmod demodev”。rmmod 通过系统调用 sys_delete_module 来完成卸载工作，该函数原型如下：

```
long sys_delete_module(const char __user * name_user, unsigned int flags);
```

○ find_module 函数

sys_delete_module 函数首先将来自用户空间的欲卸载模块名用 strncpy_from_user 函数复制到内核空间：

```
if (strncpy_from_user(name, name_user, MODULE_NAME_LEN-1) < 0)
    return -EFAULT;
```

然后调用 find_module 函数在内核维护的模块链表 modules 中利用 name 来查找要卸载的模块。find_module 函数的定义如下：

```
<kernel/module.c>
-----
/* Search for module by name: must hold module_mutex. */
struct module *find_module(const char *name)
```

```

{
    struct module *mod;

    list_for_each_entry(mod, &modules, list) {
        if (strcmp(mod->name, name) == 0)
            return mod;
    }
    return NULL;
}

```

函数通过 `list_for_each_entry` 在 `modules` 链表中遍历每一个模块 `mod`，通过前面的讨论我们知道，全局变量 `modules` 用来管理系统中所有已加载的模块形成的链表。如果查找到指定的模块，则函数返回该模块的 `mod` 结构，否则返回 `NULL`。

○ 检查模块依赖关系

如果 `sys_delete_module` 函数成功查找到了要卸载的模块，那么接下来就要检查是否有别的模块依赖于当前要卸载的模块，为了系统的稳定，一个有依赖关系的模块不应该从系统中卸载掉。在前面“模块间的依赖关系”部分中已经讨论了内核如何跟踪系统中各模块之间的依赖关系，这里只需检查要卸载模块的 `source_list` 链表是否为空，即可判断这种依赖关系，相关代码段为：

```

if (!list_empty(&mod->source_list)) {
    /* Other modules depend on us: get rid of them first. */
    ret = -EWOULDBLOCK;
    goto out;
}

```

○ `free_module` 函数

如果一切正常，`sys_delete_module` 函数最后会调用 `free_module` 函数来做模块卸载末期的一些清理工作，包括更新模块的状态为 `MODULE_STATE_GOING`，将卸载的模块从 `modules` 链表中移除，将模块占用的 `CORE section` 空间释放，释放模块从用户空间接收的参数所占的空间等，函数的实现相对比较直白，这里就不再仔细讨论了。

1.4 本章小结

本章详细讨论了作为设备驱动程序的重要存在形式——内核模块幕后的技术细节，内核模块可以在系统运行期间动态扩展系统的功能，这是其最大的优势。在用户空间中，加载和卸载模块使用的是一组称为 `mod utils` 的工具包，其中包括最基本的 `insmod` 和 `rmmod` 工具。

内核模块在文件格式上是一种可重定位的 `ELF` 文件，由 `Linux` 系统中的内核模块加载器负

责加载和卸载。模块可以调用内核源码或者其他模块实现的函数，这需要模块的构造工具链和内核模块加载器共同协作才可以完成。为此，内核和内核模块通过 `EXPORT_SYMBOL` 宏向外导出符号，这些导出的符号可以被其他模块所使用，它们被放在一个特殊的 `section` 中，内核和内核模块拥有各自的 `section` 用来保存导出符号的信息。系统中所有成功加载的模块都以链表的形式存放在内核的一个全局变量 `modules` 中。

模块在编译时需要指定一个内核源码树，这种指定不是出于链接的需要，而是模块需要内核源码头文件中的一些定义，包括以头文件形式出现的内核配置信息。因此，一个 `.ko` 文件总是基于某一特定内核源码树所构成，如果要在一个运行不同内核版本的系统中加载该 `.ko` 文件，有可能会引起一些潜在问题。例如，随着内核版本的演进，老的 `.ko` 文件所调用的一些内核函数在新版本的内核中可能消失或者改变了。为了防止这一问题的发生，内核引入了版本控制机制。如果内核模块以开放源码的形式向外发布，则版本不一致并不会成为一个问题，用户可以在新版本的内核上重新编译构造新的 `.ko` 文件。



第 2 章

字符设备驱动程序

现实世界中存在着大量的设备，这些设备在电气特性和 I/O 方式上都各不相同。为了简化设备驱动程序员的工作，Linux 系统从这些各异的设备中提取出了共性的特征，将其划分为三大类：字符设备、块设备和网络设备。内核针对每一类设备都提供了对应的驱动模型框架，包括基本的内核设施和文件系统接口。这样设备驱动程序员在写某类设备驱动程序时，就有一套完整的驱动模型框架可以使用，从而可将大量的精力放在设备本身的操作上。图 2-1 展示了一个粗略的 Linux 设备驱动程序结构图：

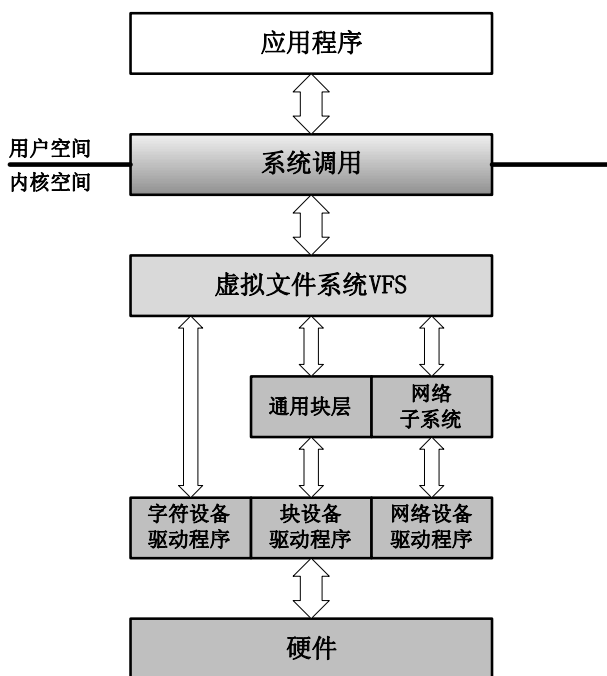


图 2-1 Linux 设备驱动程序结构图

其中，字符设备驱动程序是这三类设备驱动程序中最常见，也是相对比较容易理解的一种，现实中的大部分硬件都可由字符设备驱动程序来控制。这类硬件的特征是，在 I/O 传输过程中以字符为单位，这种字符流的传输速率通常都比较缓慢（因其内核设施中不提供缓存机制），常见的如键盘、鼠标及打印机等设备。

本章将详细讨论构成字符设备驱动程序的内核设施的幕后机制，此外还将讨论应用程序如何与字符设备驱动程序进行交互，也即应用程序如何使用字符设备驱动程序提供的服务，这将涉及字符设备的文件系统接口等相关内容。

字符设备驱动程序所提供的功能是以设备文件¹的形式提供给用户空间程序使用，本章将首先讨论应用程序与设备文件，然后再深入探讨字符设备驱动程序的内核机制。

2.1 应用程序与设备驱动程序互动实例

在深入讨论字符设备驱动程序之前，我们先用一个实际的例子来展示应用程序如何与字符设备驱动程序进行交互。这个例子中，我们首先给出一个简单的字符设备驱动程序的内核模块，接着通过 `insmod` 工具将这个内核模块加入到系统中，之后通过 `mknod` 来创建一个设备文件节点（在这个例子中我们将手动创建设备文件节点，不过后面会讨论设备节点的自动生成机制），最后再编写一个小的应用程序，用该应用程序来调用前面设备驱动程序所提供的服务。因为这里只是展示应用程序与设备驱动程序相互交互的环节，所以无论是设备驱动程序还是应用程序，都尽量保持简单且与具体硬件设备无关。在本章后续的小节中将仔细讨论这个例子中所有关键环节的幕后技术细节。

○ 字符设备驱动程序源码

`<demo_chr_dev.c>`

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>
```

```
static struct cdev chr_dev; //定义一个字符设备对象
static dev_t ndev; //字符设备节点的设备号
```

```
static int chr_open(struct inode *nd, struct file *filp)
{
    int major = MAJOR(nd->i_rdev);
    int minor = MINOR(nd->i_rdev);
    printk("chr_open, major=%d, minor=%d\n", major, minor);
    return 0;
}
```

```
static ssize_t chr_read(struct file *f, char __user *u, size_t sz, loff_t *off)
{

```

¹ 本书中“设备文件”、“设备文件节点”和“设备节点”表述的是同一个意思。

```

        printk("In the chr_read() function!\n");
        return 0;
    }

//字符设备驱动程序中非常关键的一个数据结构 struct file_operations
struct file_operations chr_ops =
{
    .owner = THIS_MODULE,
    .open = chr_open,
    .read = chr_read,
};

//模块的初始化函数
static int demo_init(void)
{
    int ret;
    cdev_init(&chr_dev, &chr_ops); //初始化字符设备对象
    ret = alloc_chrdev_region(&ndev, 0, 1, "chr_dev"); //分配设备号
    if(ret < 0)
        return ret;
    printk("demo_init():major=%d, minor=%d\n", MAJOR(ndev), MINOR(ndev));
    ret = cdev_add(&chr_dev, ndev, 1); //将字符设备对象 chr_dev 注册进系统
    if(ret < 0)
        return ret;

    return 0;
}

static void demo_exit(void)
{
    printk("Removing chr_dev module...\n");
    cdev_del(&chr_dev); //将字符设备对象 chr_dev 从系统中注销掉
    unregister_chrdev_region(ndev, 1); //释放分配的设备号
}

module_init(demo_init);
module_exit(demo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dennis @AMDLinuxFGL");
MODULE_DESCRIPTION("A char device driver as an example");

```

以上就是一个字符设备驱动程序的源码，虽然极其简单，以至于没有做任何有实质意义的事情，但是它展示了字符设备驱动程序的典型框架结构，字符设备驱动程序中绝大多数的关键元素都出现在了上面这个示例程序中，它们将成为本章后续讨论的核心。

读者可以参照下面这个简单的 Makefile 文件来编译上述的模块：

```
obj-m := demo_chr_dev.o
KERNELDIR := /lib/modules/$(shell uname -r) /build2
PWD := $(shell pwd)

default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:
    rm -f *.o *.ko *.mod.c
```

如果一切顺利，将得到一个名为 demo_chr_dev.ko 的内核模块。

○ 应用程序源码

```
<main.c>
-----
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define CHR_DEV_NAME "/dev/chr_dev"

int main()
{
    int ret;
    char buf[32];
    int fd = open(CHR_DEV_NAME, O_RDONLY|O_NDELAY);
    if(fd < 0)
    {
        printf("open file %s failed!\n", CHR_DEV_NAME);
        return -1;
    }
    read(fd, buf, 32);
    close(fd);

    return 0;
}
```

读者可以用 gcc 来生成该应用程序的可执行文件 main：

```
root@AMDLinuxFGL:/home/dennis/book/chap2/app# gcc main.c -o main
```

应用程序主要是用 open 打开一个设备文件节点，然后在打开的设备文件描述符 fd 上调用 read 函数。调用 read 函数时，除了 fd 必须使用外，其他两个参数完全是为了满足 read 函

² 读者可能需要根据自己系统中实际的内核源码路径来修改这里的 KERNELDIR 值，以消除可能出现的编译错误。

数调用的需要，设备驱动程序中的 `chr_read` 不会用到这些参数。

这个简单的例子将展示应用程序如何通过文件系统调用，穿越到内核空间，呼叫到设备驱动程序实现的各种接口函数。本章稍后将和读者一道去探讨这个示例程序背后所包含的技术细节，等到对字符设备驱动程序的各种内核设施及文件系统的接口有了深入的理解，相信在实际的工作中一定可以自由地驾驭它们，即便遇到问题也可以快速定位和解决。

○ 示例操作步骤

现在用 `insmod` 把 `demo_chr_dev.ko` 加入到系统：

```
root@AMDLinuxFGL:/home/dennis/book/chap2/gene-module# insmod demo_chr_dev.ko
```

`dmesg` 针对这个 `insmod` 的输出信息为：

```
[19611.946440] demo_init():major=248, minor=0
```

通过上面 `dmesg` 的输出信息，我们知道 `alloc_chrdev_region` 函数给内核模块 `demo_chr_dev.ko` 分配的主设备号为 248，次设备号为 0。根据这个设备号信息用 `mknod` 命令在系统的 `/dev` 目录下为该模块生成一个新的设备文件节点：

```
root@AMDLinuxFGL:/home/dennis/book/chap2/app# mknod /dev/chr_dev c 248 0
```

如果一切正常，那么在 `/dev` 目录下就会产生一个新的设备文件节点 “`/dev/chr_dev`”，可以用 `ls` 命令来仔细观察一下它：

```
root@AMDLinuxFGL:/home/dennis/book/chap2/app# ls -l /dev/chr_dev
crw-r--r-- 1 root root 248, 0 2011-05-11 21:41 /dev/chr_dev
```

上面 `ls` 命令的输出反映出了设备节点 “`/dev/chr_dev`” 的如下一些关键信息：

“`crw-r--r--`” 中的字符 “`c`” 表明这是个字符设备文件，248 是该设备节点的主设备号，次设备号则是 0，这跟我们的预期是完全一致的。

有了对应的设备文件之后，现在可以运行我们的应用程序了：

```
root@AMDLinuxFGL:/home/dennis/book/chap2/app# ./main
```

查看 `dmesg` 对此的输出信息：

```
root@AMDLinuxFGL:/home/dennis/book/chap2/app# dmesg -c
[20340.589750] chr_open, major=248, minor=0
[20340.589760] In the chr_read() function!
```

对比前面内核模块 `demo_chr_dev.ko` 的源码，读者应该知道上述两行的输出分别来自内核

模块中的 `chr_open` 和 `chr_read` 函数，虽然在这个示例程序中它们几乎没做任何事情，但是我们见证了应用程序成功调用到了设备驱动程序实现的函数，这正是我们所预期的目标。

2.2 struct file_operations

在开始讨论字符设备驱动程序内核机制前，有必要先交代一下 `struct file_operations` 数据结构，其定义如下：

```
<include/linux/fs.h>
-----
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long,
                                      unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **);
    long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
};
```

可以看到，`struct file_operations` 的成员变量几乎全是函数指针，因为本书的后续章节会陆续讨论到这个结构体中绝大多数成员的实现，所以这里不再解释其各自的用途。读者也许很快会发现，现实中字符设备驱动程序的编写，其实基本上是围绕着如何实现 `struct file_operations` 中的那些函数指针成员而展开的。通过内核文件系统组件在其间的穿针引线，应用程序中对文件类函数的调用，比如 `read()` 等，将最终被转接到 `struct file_operations`

中对应函数指针的具体实现上。

该结构中唯一非函数指针类成员 `owner`，表示当前 `struct file_operations` 对象所属的内核模块，几乎所有的设备驱动程序都会用 `THIS_MODULE` 宏给 `owner` 赋值，该宏的定义为：

```
<include/linux/module.h>
-----
#define THIS_MODULE (&__this_module)
```

`__this_module` 是内核模块的编译工具链为当前模块产生的 `struct module` 类型对象，所以 `THIS_MODULE` 实际上是当前内核模块对象的指针，`file_operations` 中的 `owner` 成员可以避免当 `file_operations` 中的函数正在被调用时，其所属的模块被从系统中卸载掉。如果一个设备驱动程序不是以模块的形式存在，而是被编译进内核，那么 `THIS_MODULE` 将被赋值为空指针，没有任何作用。

2.3 字符设备的内核抽象

顾名思义，字符设备驱动程序管理的核心对象是字符设备。从字符设备驱动程序的设计框架角度出发，内核为字符设备抽象出了一个具体的数据结构 `struct cdev`，其定义如下：

```
<include/linux/cdev.h>
-----
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```

在本章后续的内容中将陆续看到它们的实际用法，这里只把这些成员的作用简单描述如下：

`struct kobject kobj`

内嵌的内核对象，其用途将在“Linux 设备驱动模型”一章中讨论。

`struct module *owner`

字符设备驱动程序所在的内核模块对象指针。

`const struct file_operations *ops`

字符设备驱动程序中一个极其关键的数据结构，在应用程序通过文件系统接口呼叫到设备驱动程序中实现的文件操作类函数的过程中，`ops` 指针起着桥梁纽带的作用。

`struct list_head list`

用来将系统中的字符设备形成链表。

`dev_t dev`

字符设备的设备号，由主设备号和次设备号构成。

`unsigned int count`

隶属于同一主设备号的次设备号的个数，用于表示由当前设备驱动程序控制的实际同类设备的数量。

设备驱动程序中可以用两种方式来产生 `struct cdev` 对象。一是静态定义的方式，比如在前面的那个示例程序中，通过下列代码静态定义了一个 `struct cdev` 对象：

```
static struct cdev chr_dev;
```

另一种是在程序的执行期通过动态分配的方式产生，比如：

```
static struct cdev *p = kmalloc(sizeof(struct cdev), GFP_KERNEL);
```

其实 Linux 内核源码中提供了一个函数 `cdev_alloc`，专门用于动态分配 `struct cdev` 对象。`cdev_alloc` 不仅会为 `struct cdev` 对象分配内存空间，还会对该对象进行必要的初始化：

```
<fs/char_dev.c>
```

```
struct cdev *cdev_alloc(void)
```

```
{
    struct cdev *p = kzalloc(sizeof(struct cdev), GFP_KERNEL);
    if (p) {
        INIT_LIST_HEAD(&p->list);
        kobject_init(&p->kobj, &ktype_cdev_dynamic);
    }
    return p;
}
```

需要注意的是，内核引入 `struct cdev` 数据结构作为字符设备的抽象，仅仅是为了满足系统对字符设备驱动程序框架结构设计需要，现实中一个具体的字符硬件设备的数据结构的抽象往往要复杂得多，在这种情况下 `struct cdev` 常常作为一种内嵌的成员变量出现在实际设备的数据机构中，比如：

```
struct my_keypad_dev{
    //硬件相关的成员变量
    int a;
    int b;
    int c;
    ...
}
```

```
//内嵌的 struct cdev 数据结构
struct cdev cdev;

};
```

在这样的情况下，如果要动态分配一个 `struct real_char_dev` 对象，`cdev_alloc` 函数显然就无能为力了，此时只能使用下面的方法：

```
static struct real_char_dev *p = kzalloc(sizeof(struct real_char_dev), GFP_KERNEL);
```

前面讨论了如何分配一个 `struct cdev` 对象，接下来的一个话题是如何初始化一个 `cdev` 对象，内核为此提供的函数是 `cdev_init`：

```
<fs/char_dev.c>
-----
void cdev_init(struct cdev *cdev, const struct file_operations *fops)
{
    memset(cdev, 0, sizeof *cdev);
    INIT_LIST_HEAD(&cdev->list);
    kobject_init(&cdev->kobj, &ktype_cdev_default);
    cdev->ops = fops;
}
```

函数的代码非常直白，不再赘述。一个 `struct cdev` 对象在被最终加入系统前，都应该被初始化，无论是直接通过 `cdev_init` 或者是其他途径。理由很简单，这是 Linux 系统中字符设备驱动程序框架设计的需要。

照理在谈完 `cdev` 对象的分配和初始化之后，下面应该讨论如何将一个 `cdev` 对象加入到系统了，但是由于这个过程需要用到设备号相关的技术点，所以暂且先来探讨设备号的问题。

2.4 设备号的构成与分配

本节开始讨论设备号相关的问题，不过设备号对于设备驱动程序而言究竟意味着什么，换句话说，它在内核中起着怎样的作用，本节暂不讨论，这里只关心它在内核中是如何分配和管理的。

2.4.1 设备号的构成

Linux 系统中一个设备号由主设备号和次设备号构成，Linux 内核用主设备号来定位对应的设备驱动程序，而次设备号则由驱动程序使用，用来标识它所管理的若干同类设备。因此，从这个角度而言，设备号作为一种系统资源，必须仔细加以管理，以防止因设备号与驱动程序错误的对应关系所带来的混乱。

Linux 用 `dev_t` 类型变量来标识一个设备号，这是个 32 位的无符号整数：


```
<include/linux/types.h>
typedef __u32 __kernel_dev_t;
typedef __kernel_dev_t dev_t;
```

图 2-2 显示了 2.6.39 版本内核中设备号的构成：



图 2-2 Linux 的设备号的构成

在这一内核版本中，`dev_t` 的低 20 位用来表示次设备号，高 12 位用来表示主设备号。随着内核版本的演变，上述的主次设备号的构成也许会发生改变，所以设备驱动程序开发者应该避免直接使用主次设备号所占有的位宽来获得对应的主设备号或次设备号。为了保证在主次设备号位宽发生改变时，现有的程序依然可以正常工作，内核提供了如下几个宏供设备驱动程序操作设备号时使用：

```
<include/linux/kdev_t.h>
#define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev) ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))
```

`MAJOR` 宏用来从一个 `dev_t` 类型的设备号中提取出主设备号，`MINOR` 宏则用来提取设备号中的次设备号。`MKDEV` 则是将主设备号 `ma` 和次设备号 `mi` 合成一个 `dev_t` 类型的设备号。在上述宏定义中，`MINORBITS` 宏在 2.6.39 版本中定义的值是 20，如果之后的内核对主次设备号所占用的位宽重新进行调整，例如将 `MINORBITS` 改成 12，只要设备驱动程序坚持使用 `MAJOR`、`MINOR` 和 `MKDEV` 来操作设备号，那么这部分代码应该无须修改就可以在新内核中运行。

2.4.2 设备号的分配与管理

在内核源码中，涉及设备号分配与管理的函数主要有以下两个：

○ `register_chrdev_region` 函数

该函数的代码实现如下：

```
<fs/char_dev.c>
int register_chrdev_region(dev_t from, unsigned count, const char *name)
{
    struct char_device_struct *cd;
    dev_t to = from + count;
    dev_t n, next;
```

```

    for (n = from; n < to; n = next) {
        next = MKDEV(MAJOR(n)+1, 0);
        if (next > to)
            next = to;
        cd = __register_chrdev_region(MAJOR(n), MINOR(n),
                                     next - n, name);
        if (IS_ERR(cd))
            goto fail;
    }
    return 0;
fail:
    to = n;
    for (n = from; n < to; n = next) {
        next = MKDEV(MAJOR(n)+1, 0);
        kfree(__unregister_chrdev_region(MAJOR(n), MINOR(n), next - n));
    }
    return PTR_ERR(cd);
}

```

该函数的第一参数 `from` 表示的是一个设备号，第二参数 `count` 是连续设备编号的个数，代表当前驱动程序所管理的同类设备的个数，第三参数 `name` 表示设备或者驱动的名称。`register_chrdev_region` 的核心功能体现在内部调用的 `__register_chrdev_region` 函数中，在讨论这个函数之前，先要看一个全局性的指针数组 `chrdevs`，它是内核用于设备号分配与管理的核心元素，其定义如下：

```

<fs/char_dev.c>
-----
static struct char_device_struct3 {
    struct char_device_struct *next;
    unsigned int major;
    unsigned int baseminor;
    int minorct;
    char name[64];
    struct cdev *cdev;    /* will die */
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE4];

```

这个数组中的每一项都是一个指向 `struct char_device_struct` 类型的指针。系统刚开始运行时，该数组的初始状态如图 2-3 所示：

现在回过头来看看 `register_chrdev_region` 函数，这个函数要完成的主要功能是将当前设备驱动程序要使用的设备号记录到 `chrdevs` 数组中，有了这种对设备号使用情况的跟踪，系统

³ 这个结构体中的成员变量 `cdev` 在设备号管理模块中没有任何用处，至少在目前看来是这样。如果不是保留用于将来的某种扩展，那么可以预见，在不久的将来这个成员最终会被清除掉，正如源码注释中所说的那样，“will die”。

⁴ 在 2.6.39 版本的内核源码中，`CHRDEV_MAJOR_HASH_SIZE` 定义的值为 255。

就可以避免不同的设备驱动程序使用同一个设备号的情形出现。这意味着当设备驱动程序调用这个函数时，事先已经明确知道它所要使用的设备号，之所以调用这个函数，是要将所使用的设备号纳入到内核的设备号管理体系中，防止别的驱动程序错误使用到。当然如果它试图使用的设备号已经被之前某个驱动程序使用了，调用将不会成功，`register_chrdev_region` 函数将会返回一个负的错误码告知调用者，如果调用成功，函数返回 0。

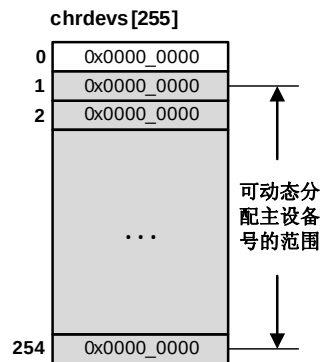


图 2-3 初始状态的 `chrdevs` 数组结构

上述这些设备号功能的实现其实最终发生在 `register_chrdev_region` 函数内部所调用的 `__register_chrdev_region` 函数中，它会首先分配一个 `struct char_device_struct` 类型的对象 `cd`，然后对其进行一些初始化：

```
<fs/char_dev.c>
-----
static struct char_device_struct *
__register_chrdev_region(unsigned int major, unsigned int baseminor,
                        int minorct, const char *name)
{
    cd = kzalloc(sizeof(struct char_device_struct), GFP_KERNEL);
    ...
    cd->major = major;
    cd->baseminor = baseminor;
    cd->minorct = minorct;
    strlcpy(cd->name, name, sizeof(cd->name));
}
```

这个过程完成之后，它开始搜索 `chrdevs` 数组，搜索是以哈希表的形式进行的，为此必须首先获取一个散列关键值，正如读者所预料的那样，它用主设备号来生成这个关键值：

```
i = major_to_index(major);
```

这是个非常简单的获得散列关键值的方法， $i = \text{major} \% 255$ 。此后函数将对 `chrdevs[i]` 元素管理的链表进行扫描，如果 `chrdevs[i]` 上已经有了链表节点，表明之前有别的设备驱动程序使用的主设备号散列到了 `chrdevs[i]` 上，为此函数需要相应的逻辑确保当前正在操作的设备号不会与这些已经在使用的设备号发生冲突，如果有冲突，函数将返回错误码，表明本次

调用没有成功。如果本次调用使用的设备号与 `chrdevs[i]` 上已有的设备号没有发生冲突，先前分配的 `struct char_device_struct` 对象 `cd` 将加入到 `chrdevs[i]` 领衔的链表中成为一个新的节点。没有必要再仔细分析 `_register_chrdev_region` 函数中的相关代码了，接下来以一个具体的例子来了解这一过程。

在 `chrdevs` 数组尚处于初始状态的情形下，假设现在有一个设备驱动程序要使用的主设备号是 257，次设备号分别是 0、1、2 和 3（意味着该驱动程序将管理四个同类型的设备）。它对 `register_chrdev_region` 函数的调用如下：

```
int ret = register_chrdev_region(MKDEV(257, 0), 4, "demodev");
```

上述对 `register_chrdev_region` 函数的调用完毕后，`chrdevs` 数组的状态将变成图 2-4 所示（图中假设新分配的 `struct char_device_struct` 节点的基地址为 `0xC8000004`，这些节点基地址数值只是用来使读者有个直观的概念，并非代表系统中实际分配的地址值）：

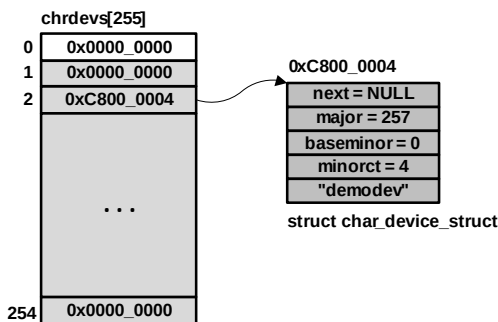


图 2-4 主设备号 257 注册后的 `chrdevs` 数组状态

现在假设有另一个设备驱动程序使用的主设备号为 2，次设备号为 0，当它调用 `register_chrdev_region(MKDEV(2, 0), 1, "augdev")` 来向系统注册设备号时，因为 $2 \% 255 = 2$ ，所以也将索引到 `chrdevs` 数组的第 2 项。虽然数组的第 2 项中已经有 "demodev" 设备在使用，但是因为这次注册的设备号是 `MKDEV(2, 0)`，与设备 "demodev" 的设备号 `MKDEV(257, 0)` 并不冲突，所以注册总会成功。因为 Linux 在将设备 "augdev" 对应的 `struct char_device_struct` 对象节点加入到哈希表中时，采用了插入排序，这导致同一哈希列表将按照 `major` 的大小递增排列，因此此时的 `chrdevs` 数组状态如图 2-5 所示：

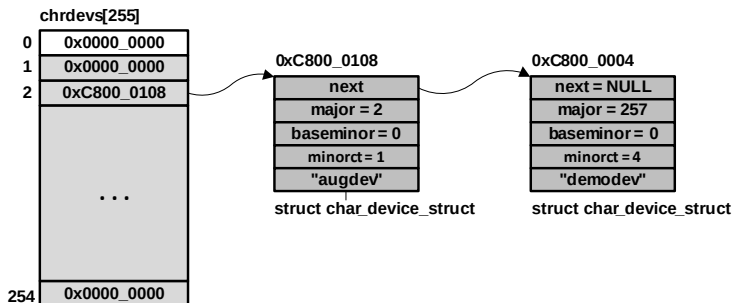


图 2-5 主设备号 2 加入后的 `chrdevs` 数组状态

一个有趣的事实是，在图 2-5 的基础上，假设有另一个设备驱动程序调用 `register_chrdev_region` 函数向系统注册，主设备号也为 257，那么只要其次设备号所在的范围`[baseminor, baseminor + minorct]`不与设备"demodev"的次设备号范围发生重叠，系统依然会生成一个新的 `struct char_device_struct` 节点并加入到对应的哈希链表中。在主设备号相同的情况下，如果次设备号的范围有重叠，则意味着有设备号的冲突，这将导致对 `register_chrdev_region` 函数的调用失败。对主设备号相同的若干 `struct char_device_struct` 对象，当系统将其加入链表时，将根据其 `baseminor` 成员的大小进行递增排序。

○ `alloc_chrdev_region` 函数

该函数由系统协助分配设备号，分配的主设备号范围将在 1~254 之间，其定义如下：

```
<fs/char_dev.c>
-----
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,
                        const char *name)
{
    struct char_device_struct *cd;
    cd = __register_chrdev_region(0, baseminor, count, name);
    if (IS_ERR(cd))
        return PTR_ERR(cd);
    *dev = MKDEV(cd->major, cd->baseminor);
    return 0;
}
```

这个函数的核心调用也是 `__register_chrdev_region`，相对于 `register_chrdev_region`，`alloc_chrdev_region` 在调用 `__register_chrdev_region` 时，第一个参数为 0，这将导致 `__register_chrdev_region` 执行下面的逻辑：

```
<fs/char_dev.c>
-----
static struct char_device_struct *
__register_chrdev_region(unsigned int major, unsigned int baseminor,
                        int minorct, const char *name)
{
    ...
    if (major == 0) {
        for (i = ARRAY_SIZE(chrdevs)-1; i > 0; i--) {
            if (chrdevs[i] == NULL)
                break;
        }
        if (i == 0) {
            ret = -EBUSY;
            goto out;
        }
        major = i;
        ret = major;
    }
```

```

    }
    ...
}

```

上述代码片段的实现原理非常简单，它在 for 循环中从 `chrdevs` 数组的最后一项（也就是第 254 项）依次向前扫描，如果发现该数组中的某项，比如第 `i` 项，对应的数值为 `NULL`，那么就把该项对应的索引值 `i` 作为分配的主设备号返回给驱动程序，同时生成一个 `struct char_device_struct` 节点，并将其加入到 `chrdevs[i]` 对应的哈希链表中。如果从第 254 项一直到第 1 项，这其中所有的项对应的指针都不为 `NULL`，那么函数失败并返回一非 0 值，表明动态分配设备号失败。如果分配成功，所分配的主设备号将记录在 `struct char_device_struct` 对象 `cd` 中，并将该对象返回给 `alloc_chrdev_region` 函数，后者通过下面的代码将新分配的设备号返回给函数的调用者：

```
*dev = MKDEV(cd->major, cd->baseminor);
```

设备号作为一种系统资源，当所对应的设备驱动程序被卸载时，很显然要把其所占用的设备号归还给系统，以便分配给其他内核模块使用。不管是用 `register_chrdev_region` 还是 `alloc_chrdev_region` 注册或者分配的设备号，在 Linux 中都由下面的函数负责释放：

```
<fs/char_dev.c>
```

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

函数在 `chrdevs` 数组中查找参数 `from` 和 `count` 所对应的 `struct char_device_struct` 对象节点，找到以后将其从链表中删除并释放该节点所占用的内存，从而将对应的设备号释放以供其他设备驱动模块使用。

以上讨论了内核中用于设备号分配与管理的技术细节，焦点是 `register_chrdev_region` 和 `alloc_chrdev_region` 两个函数，除了 `alloc_chrdev_region` 还具有让系统协助分配一个主设备号的功能外，它们最主要的作用其实都是通过 `chrdevs` 数组来跟踪系统中设备号的使用情况，以防止实际使用中出现设备号冲突的情况。这是内核提供给设备驱动程序使用的一种预防性措施，并没有必然的理由说设备驱动程序一定要使用这两个函数，如果可以确定设备驱动程序将要使用的设备号不会与系统中已有的设备号发生冲突，完全可以绕开它们。但很明显这是一种非常糟糕的习惯，如果某些设备驱动程序没有使用系统提供的 `register_chrdev_region` 或者 `alloc_chrdev_region` 函数，那么系统将失去一个对设备号使用情况进行跟踪的措施。既然内核在设备驱动程序的框架设计中定义了这种规则，作为设备驱动程序的实际开发者，没有理由不去遵循这些规则。

2.5 字符设备的注册

前面已经讨论了字符设备对象的分配、初始化及设备号等概念，在一个字符设备初始化阶

段完成之后，就可以把它加入到系统中，这样别的模块才可以使用它。把一个字符设备加入到系统中所需调用的函数为 `cdev_add`，它在 Linux 源码中的实现如下：

`<fs/char_dev.c>`

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count)
{
    p->dev = dev;
    p->count = count;
    return kobj_map(cdev_map, dev, count, NULL, exact_match, exact_lock, p);
}
```

其中，参数 `p` 为要加入系统的字符设备对象的指针，`dev` 为该设备的设备号，`count` 表示从次设备号开始连续的设备数量。

`cdev_add` 的核心功能通过 `kobj_map` 函数来实现，后者通过操作一个全局变量 `cdev_map` 来把设备（*p）加入到其中的哈希链表中。`cdev_map` 的定义如下：

`<fs/char_dev.c>`

```
static struct kobj_map *cdev_map;
```

这是一个 `struct kobj_map` 指针类型的全局变量，在 Linux 系统启动期间由 `chrdev_init` 函数负责初始化。`struct kobj_map` 的定义如下：

`<drivers/base/map.c>`

```
struct kobj_map {
    struct probe {
        struct probe *next;
        dev_t dev;
        unsigned long range;
        struct module *owner;
        kobj_probe_t *get;
        int (*lock)(dev_t, void *);
        void *data;
    } *probes[255];
    struct mutex *lock;
};
```

`kobj_map` 函数中哈希表的实现原理和前面注册分配设备号中的几乎完全一样，通过要加入系统的设备的主设备号 `major(major=MAJOR(dev))` 来获得 `probes` 数组的索引值 `i` ($i = \text{major} \% 255$)，然后把一个类型为 `struct probe` 的节点对象加入到 `probes[i]` 所管理的链表中，如图 2-6 所示。其中 `struct probe` 所在的矩形块中的深色部分是我们重点关注的内容，记录了当前正在加入系统的字符设备对象的有关信息。其中，`dev` 是它的设备号，`range` 是从次设备号开始连续的设备数量，`data` 是一 `void *` 变量，指向当前正要加入系统的设备对象指针 `p`。图 2-6 展示了两个满足主设备号 $\text{major} \% 255 = 2$ 的字符设备通过调用 `cdev_add` 之后，

cdev_map 所展现出来的数据结构状态。

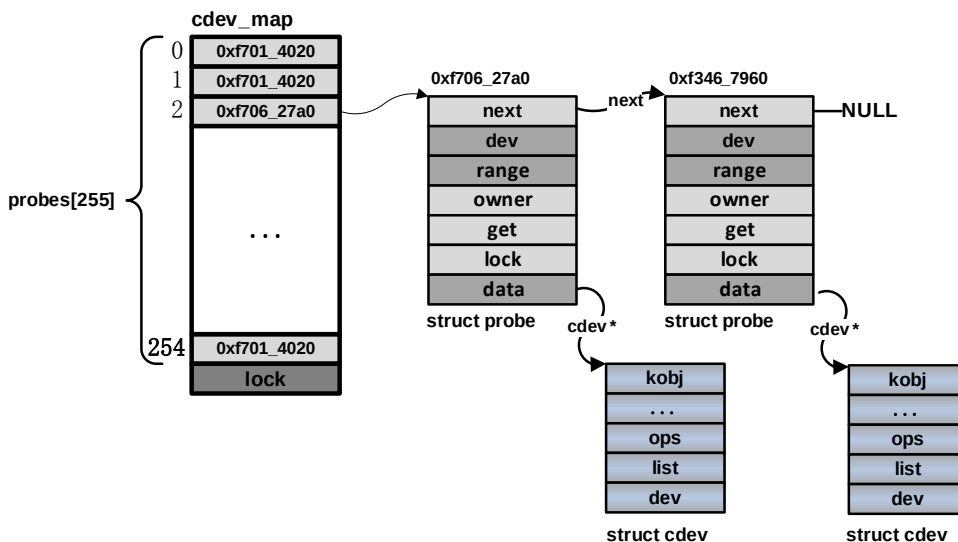


图 2-6 通过 cdev_add 向系统中加入设备

所以，简单地说，设备驱动程序通过调用 cdev_add 把它所管理的设备对象的指针嵌入到一个类型为 struct probe 的节点之中，然后再把该节点加入到 cdev_map 所实现的哈希链表中。

对系统而言，当设备驱动程序成功调用了 cdev_add 之后，就意味着一个字符设备对象已经加入到了系统，在需要的时候，系统就可以找到它。对用户态的程序而言，cdev_add 调用之后，就已经可以通过文件系统的接口呼叫到我们的驱动程序，本章稍后将会详细描述这一过程。

不过在开始文件系统如何通过 cdev_map 来使用驱动程序提供的服务这个话题之前，我们要来看看与 cdev_add 相对应的另一个函数 cdev_del。其实光通过这个函数名，读者想必也想到这个函数的作用了：在 cdev_add 中我们动态分配了 struct probe 类型的节点，那么当对应的设备从系统中移除时，显然需要将它们从链表中删除并释放节点所占用的内存空间。在 cdev_map 所管理的链表中查找对应的设备节点时使用了设备号。cdev_del 函数的实现如下：

```
<fs/char_dev.c>
void cdev_del(struct cdev *p)
{
    cdev_unmap(p->dev, p->count);
    kobject_put(&p->kobj);
}
```

对于以内核模块形式存在的驱动程序，作为通用的规则，模块的卸载函数应负责调用这个函数来将所管理的设备对象从系统中移除。

2.6 设备文件节点的生成

在 Linux 系统下，设备文件是种特殊的文件类型，其存在的主要意义是沟通用户空间程序和内核空间驱动程序。换句话说，用户空间的应用程序要想使用驱动程序提供的服务，需要经过设备文件来达成。当然，如果你的驱动程序只是为内核中的其他模块提供服务，则没有必要生成对应的设备文件。

按照通用的规则，Linux 系统所有的设备文件都位于 `/dev` 目录下。`/dev` 目录在 Linux 系统中算是一个比较特殊的目录，在 Linux 系统早期还不支持动态生成设备节点时，`/dev` 目录就是挂载的根文件系统下的 `/dev`，对这个目录下所有文件的操作使用的是根文件系统提供的接口。比如，如果 Linux 系统挂载的根文件系统是 `ext3`，那么对 `/dev` 目录下所有目录/文件的操作都将使用 `ext3` 文件系统的接口。随着后来 Linux 内核的演进，开始支持动态设备节点的生成⁵，使得系统在启动过程中会自动生成各个设备节点，这就使得 `/dev` 目录不必要作为一个非易失的文件系统的形式存在。因此，当前的 Linux 内核在挂载完根文件系统之后，会在这个根文件系统的 `/dev` 目录上重新挂载一个新的文件系统 `devtmpfs`，后者是个基于系统 RAM 的文件系统实现。当然，对动态设备节点生成的支持并不意味着一定要将根文件系统中的 `/dev` 目录重新挂载到一个新的文件系统上，事实上动态生成设备节点技术的重点并不在文件系统上面。

动态设备节点的特性需要其他相关技术的支持，在后续的章节中会详细描述这些特性。目前先假定设备节点是通过 Linux 系统下的 `mknod` 命令静态创建。为方便叙述，下面用一个具体的例子来描述设备文件产生过程中的一些关键要素，这个例子的任务很简单：在一个 `ext3` 类型的根文件系统中的 `/dev` 目录下用 `mknod` 命令来创建一个新的设备文件节点 `demodev`，对应的驱动程序使用的设备主设备号为 2，次设备号是 0，命令形式为：

```
root@LinuxDev:/home/dennis# mknod /dev/demodev c 2 0
```

上述命令成功执行后，将会在 `/dev` 目录下生成一个名为 `demodev` 的字符设备节点。如果用 `strace` 工具来跟踪一下上面的命令，会发现如下输出（删去了若干不相关部分）：

```
root@LinuxDev:/home/dennis# strace mknod /dev/demodev c 2 0
execve("/bin/mknod", ["mknod", "/dev/demodev", "c", "30", "0"], [/* 36 vars */]) = 0
...
mknod("/dev/demodev", S_IFCHR|0666, makedev(30,0)) = 0
```

⁵ 这里动态生成设备节点的说法是相对于使用 `mknod` 命令生成设备节点而言的，前者直接通过文件系统接口来生成对应的设备节点。

...

可见 Linux 下的 `mknod` 命令最终是通过调用 `mknod` 函数来实现的，调用时的重要参数有两个，一是设备文件名（`"/dev/demodev"`），二是设备号（`makedev(30,0)`）。设备文件名主要在用户空间使用（比如用户空间程序调用 `open` 函数时），而内核空间则使用 `inode` 来表示相应的文件。本书只关注内核空间的操作，对于前面的 `mknod` 命令，它将通过系统调用 `sys_mknod` 进入内核空间，这个系统调用的原型是：

```
<include/linux/syscalls.h>
-----
long sys_mknod(const char __user *filename, int mode, unsigned dev);
```

注意 `sys_mknod` 的最后一个参数 `dev`，它是由用户空间的 `mknod` 命令构造出的设备号。`sys_mknod` 系统调用将通过 `/dev` 目录上挂载的文件系统接口来为 `/dev/demodev` 生成一个新的 `inode`⁶，设备号将被记录到这个新的 `inode` 对象上。

图 2-7 展示了通过 `ext3` 文件系统在 `/dev` 目录下生成一个新的设备节点 `/dev/demodev` 的主要流程。

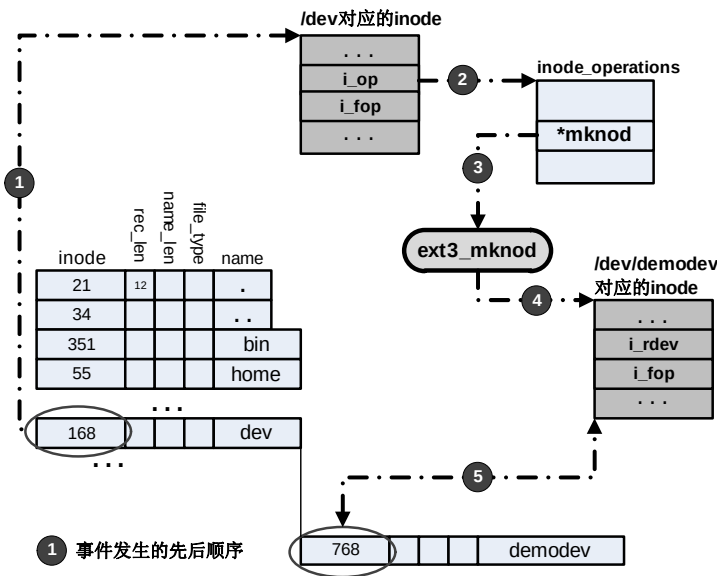


图 2-7 ext3 文件系统 mknod 的主要流程

完整了解设备节点产生的整个过程需要知晓 VFS 和特定文件系统的技术细节。然而从驱动程序员的角度来说，没有必要知道文件系统相关的所有细节，只需关注文件系统和驱动程序间是如何建立上关联的就足够了。

⁶ 对于实际的文件系统，比如 `ext3` 文件系统，产生一个 `node` 的过程因为同时要涉及底层存储设备的操作，因而会变得很复杂。

`sys_mknod` 首先在根文件系统 `ext3` 的根目录 “/” 下寻找 `dev` 目录所对应的 `inode`，图中对应的 `inode` 编号为 168，`ext3` 文件系统的实现会通过某种映射机制，通过 `inode` 编号最终得到该 `inode` 在内存中的实际地址（图中由标号 1 的线段表示）。接下来会通过 `dev` 的 `inode` 结构中的 `i_op` 成员指针所指向的 `ext3_dir_inode_operations`（这是个 `struct inode_operations` 类型的指针），来调用该对象中的 `mknod` 方法，这将导致 `ext3_mknod` 函数被调用。

`ext3_mknod` 函数的主要作用是生成一个新的 `inode`（用来在内核空间表示 `demodev` 设备文件节点，`demodev` 设备节点文件与新生成的 `inode` 之间的关联在图 2-7 中由标号 5 的线段表示）。在 `ext3_mknod` 中会调用一个和设备驱动程序关系密切的 `init_special_inode` 函数，其定义如下：

<fs/inode.c>

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {
        inode->i_fop = &def_chr_fops;
        inode->i_rdev = rdev;
    } else if (S_ISBLK(mode)) {
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = rdev;
    } else if (S_ISFIFO(mode)) {
        inode->i_fop = &def_fifo_fops;
    } else if (S_ISSOCK(mode)) {
        inode->i_fop = &bad_sock_fops;
    } else {
        printk(KERN_DEBUG "init_special_inode: bogus i_mode (%o) for"
               " inode %s:%lu\n", mode, inode->i_sb->s_id,
               inode->i_ino);
    }
}
```

这个函数最主要的功能便是为新生成的 `inode` 初始化其中的 `i_fop` 和 `i_rdev` 成员。设备文件节点 `inode` 中的 `i_rdev` 成员用来表示该 `inode` 所对应设备的设备号，通过参数 `rdev` 为其赋值。设备号在由 `sys_mknod` 发起的整个内核调用链中进行传递，最早来自于用户空间的 `mknod` 命令行参数。

`i_fop` 成员的初始化根据是字符设备还是块设备而有不同的赋值。对于字符设备，`fop` 指向 `def_chr_fops`，后者主要定义了一个 `open` 操作：

<fs/char_dev.c>

```
const struct file_operations def_chr_fops = {
    .open = chrdev_open,
    ...
};
```

相对于字符设备，块设备的 `def_blk_fops` 的定义则要有点复杂：

```
<fs/block_dev.c>
-----
const struct file_operations def_blk_fops = {
    .open      = blkdev_open,
    .release   = blkdev_close,
    .llseek    = block_llseek,
    .read      = do_sync_read,
    .write     = do_sync_write,
    .aio_read  = generic_file_aio_read,
    .aio_write = blkdev_aio_write,
    .mmap      = generic_file_mmap,
    .fsync     = blkdev_fsync,
    .unlocked_ioctl = block_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl = compat_blkdev_ioctl,
#endif
    .splice_read = generic_file_splice_read,
    .splice_write = generic_file_splice_write,
};
```

关于块设备，将在本书第 11 章“块设备驱动程序”中详细讨论，这里依然把考察的重点放在字符设备上。字符设备 `inode` 中的 `i_fop` 指向 `def_chr_fops`。至此，设备节点的所有相关铺垫工作都已经结束，接下来可以看看打开一个设备文件到底意味着什么。

2.7 字符设备文件的打开操作

作为例子，这里假定前面对应于 `/dev/demodev` 设备节点的驱动程序在自己的代码里实现了如下的 `struct file_operations` 对象 `fops`：

```
static struct file_operations fops = {
    .open = demoopen,
    .read = demoread,
    .write = demowrite,
    .ioctl = demoioctl,
};
```

用户空间 `open` 函数的原型为：

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

这个函数如果成功，将返回一个文件描述符，否则返回-1。函数的第一个参数 `filename` 表示要打开的文件名，第二个参数 `flags` 用于指定文件的打开或者创建模式，本书在后续“字符设备的高级操作”一章中会讨论其中一些常见取值对驱动程序的影响，最后一个参数

`mode` 只在创建一个新文件时才使用，用于指定新建文件的访问权限，比如可读、可写及可执行等权限。

位于内核空间的驱动程序中 `open` 函数的原型为：

```
<include/linux/fs.h>
-----
struct file_operations {
    ...
    int (*open) (struct inode *, struct file *);
    ...
};
```

两者相比差异很大。接下来我们将描述从用户态的 `open` 是如何一步一步调用到驱动程序提供的 `open` 函数（在我们的例子中，它的具体实现是 `demoopen`）的。如同设备文件节点的生成一样，透彻了解这里的每一个步骤也需要掌握全面的 Linux 下文件系统的技术细节。从设备驱动程序员的角度，我们依然将重点放在两者如何建立联系的关键点上。

用户程序调用 `open` 函数返回的文件描述符，本文用 `fd` 表示，这是个 `int` 型的变量，会被用户程序后续的 `read`、`write` 和 `ioctl` 等函数所使用。同时可以看到，在驱动程序中的 `demodev_read`、`demodev_write` 和 `demodev_ioctl` 等函数其第一个参数都是 `struct file *filp`。显然内核需要在打开设备文件时为 `fd` 与 `filp` 建立某种联系，其次是为 `filp` 与驱动程序中的 `fops` 建立关联。

用户空间程序调用 `open` 函数，将发起一个系统调用，通过 `sys_open` 函数进入内核空间，其中一系列关键的函数调用关系如图 2-8 所示：

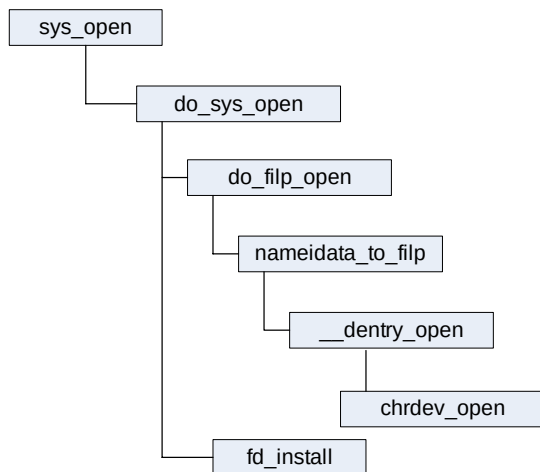


图 2-8 `sys_open` 到 `chrdev_open` 调用流程

`do_sys_open` 函数首先通过 `get_unused_fd_flags` 为本次的 `open` 操作分配一个未使用过的文

件描述符 fd⁷:

```
<fs/open.c>
-----
long do_sys_open(int dfd, const char __user *filename, int flags, int mode)
{
    ...
    fd = get_unused_fd_flags(flags);
    ...
}
```

get_unused_fd_flags 实际上是封装了 alloc_fd 的一个宏，真正分配 fd 的操作发生在 alloc_fd 函数中，后者会涉及大量文件系统方面的细节，这不是本书的主题。读者这里只需知道 alloc_fd 将会为本次的 open 操作分配一个新的 fd。

do_sys_open 随后调用 do_filp_open 函数，后者会首先查找"/dev/demodev"设备文件所对应的 inode。在 Linux 文件系统中，每个文件都有一个 inode 与之对应。从文件名查找对应的 inode 这一过程，同样会涉及大量文件系统方面的细节。

do_filp_open 在成功查找到"/dev/demodev"设备文件对应的 inode 之后，接着会调用函数 get_empty_filp，后者会为每个打开的文件分配一个新的 struct file 类型的内存空间（本书将把指向该结构体对象的内存指针简写为 filp）：

```
<fs/namei.c>
-----
struct file *do_filp_open(int dfd, const char *pathname,
                          const struct open_flags *op, int flags)
{
    struct nameidata nd;
    struct file *filp;

    filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
    ...
    return filp;
}
```

内核用 struct file 对象来描述进程打开的每一个文件的视图，即使是打开同一文件，内核也会为之生成一个新的 struct file 对象，用来表示当前操作的文件的相关信息，其定义为：

```
<include/linux/fs.h>
-----
struct file {
```

⁷ 为了跟踪对文件的读写等操作，内核对于每一次打开的文件都会分配一个文件描述符 fd 和一个 struct file 类型的实例 filp，这个二元组(fd, filp)会被后续的 read、write 等操作使用以向内核记录本次读写操作的信息。从这个角度而言，fd 实际上相当于一个文件可能出现的多种视图的一个索引。作为一种系统资源，一个进程可以分配多少个 fd 决定了一个进程可以 open 多少个文件。

```

union {
    struct list_head  fu_list;
    struct rcu_head   fu_rcuhead;
} f_u;
struct path          f_path;
#define f_dentry      f_path.dentry
#define f_vfsmnt      f_path.mnt
const struct file_operations  *f_op;
spinlock_t                f_lock;
atomic_long_t              f_count;
unsigned int               f_flags;
fmode_t                    f_mode;
loff_t                     f_pos;
struct fown_struct f_owner;
const struct cred  *f_cred;
struct file_ra_state  f_ra;

u64                f_version;
#ifdef CONFIG_SECURITY
void                *f_security;
#endif
/* needed for tty driver, and maybe others */
void                *private_data;

#ifdef CONFIG_EPOLL
/* Used by fs/eventpoll.c to link all the hooks to this file */
struct list_head  f_ep_links;
#endif /* #ifdef CONFIG_EPOLL */
struct address_space  *f_mapping;
};

```

这个结构中与服务驱动程序关系最密切的是 `f_op`、`f_flags`、`f_count` 和 `private_data` 成员。`f_op` 指针的类型是 `struct file_operations`，恰好我们的字符设备驱动程序中也需要实现一个该类型的对象，马上我们将看到这两者之间是如何建立联系的。`f_flags` 用于记录当前文件被 `open` 时所指定的打开模式，这个成员将会影响后续的 `read/write` 等函数的行为模式。成员 `f_count` 用于对 `struct file` 对象的使用计数，当 `close` 一个文件时，只有 `struct file` 对象中 `f_count` 成员为 0 才真正执行关闭操作。`private_data` 常被用来记录设备驱动程序自身定义的数据，因为 `filp` 指针会在驱动程序实现的 `file_operations` 对象其他成员函数之间传递，所以可以通过 `filp` 中的 `private_data` 成员在某一个特定文件视图的基础上共享数据。

进程为文件操作维护一个文件描述符表（`current->files->fdt`），正如在本节开始部分看到的那样，对设备文件的打开，最终会得到一个文件描述符 `fd`，然后用该描述符 `fd` 作为进程维护的文件描述符表（指向 `struct file *` 类型数组）的索引值，将之前新分配的 `struct file` 空间地址赋值给它：

```
current->files->fdt->pfd[fd] = filp;
```

这样,用户空间程序在后续的 read、write、ioctl 等函数调用中利用 fd 就可以找到对应的 filp,如图 2-9 所示:

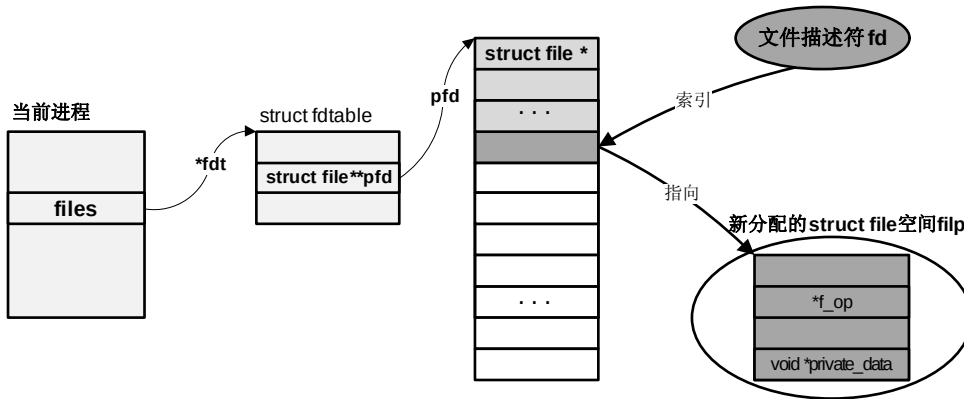


图 2-9 fd 与 filp 的关联

在 do_sys_open 的后半部分,会调用__dentry_open 函数将"/dev/demodev"对应节点的 inode 中的 i_fop 赋值给 filp->f_op, 然后调用 i_fop 中的 open 函数:

<fs/open.c>

```
static struct file *__dentry_open(struct dentry *dentry, struct vfsmount *mnt,
                                struct file *f,
                                int (*open)(struct inode *, struct file *),
                                const struct cred *cred)
{
    struct inode *inode;
    ...
    f->f_op = fops_get(inode->i_fop);
    ...
    if (!open && f->f_op)
        open = f->f_op->open;
    if (open) {
        error = open(inode, f);
        ...
    }
    ...
}
```

__dentry_open 函数当初在 nameidata_to_filp 中被调用时,第四个实参是 NULL,所以在 __dentry_open 中, open = f->f_op->open。在上节设备文件节点的生成中,我们知道 inode->i_fop = &def_chr_fops, 这样 filp->f_op = &def_chr_fops。接下来会利用 filp 中的这个新的 f_op 作调用: filp->f_op->open(inode, filp), 于是 chrdev_open 函数将被调用到。该函数非常重要,为了突出其主线,下面先将它改写成以下简单几行:


```
<fs/char_dev.c>
```

```
static int chrdev_open(struct inode *inode, struct file *filp)
{
    int ret = 0, idx;

    struct kobject *kobj = kobj_lookup(cdev_map, inode->i_rdev, &idx);
    struct cdev *new = container_of(kobj, struct cdev, kobj);
    inode->i_cdev = new;
    list_add(&inode->i_devices, &new->list);
    filp->f_op = new->ops;
    if (filp->f_op->open) {
        ret = filp->f_op->open(inode, filp);
    }
    return ret;
}
```

函数首先通过 `kobj_lookup` 在 `cdev_map` 中用 `inode->i_rdev` 来查找设备号所对应的设备 `new`，这里展示了设备号的作用。成功查找到设备后，通过 `filp->f_op = new->ops` 这行代码将设备对象 `new` 中的 `ops` 指针（前面曾讨论过，驱动程序通过调用 `cdev_init` 将其实现的 `file_operations` 对象的指针赋值给设备对象 `cdev` 的 `ops` 成员）赋值给 `filp` 对象中的 `f_op` 成员，此处展示了如何将驱动程序中实现的 `struct file_operations` 与 `filp` 关联起来，从此图 2-9 中的 `filp->f_op` 将指向驱动程序中实现的 `struct file_operations` 对象。

接下来函数会检查驱动程序中是否实现了 `open` 函数（`if (filp->f_op->open)`），如果实现了，就调用设备驱动程序中实现的 `open` 函数。打开一个字符设备节点的大体流程如图 2-10 所示：

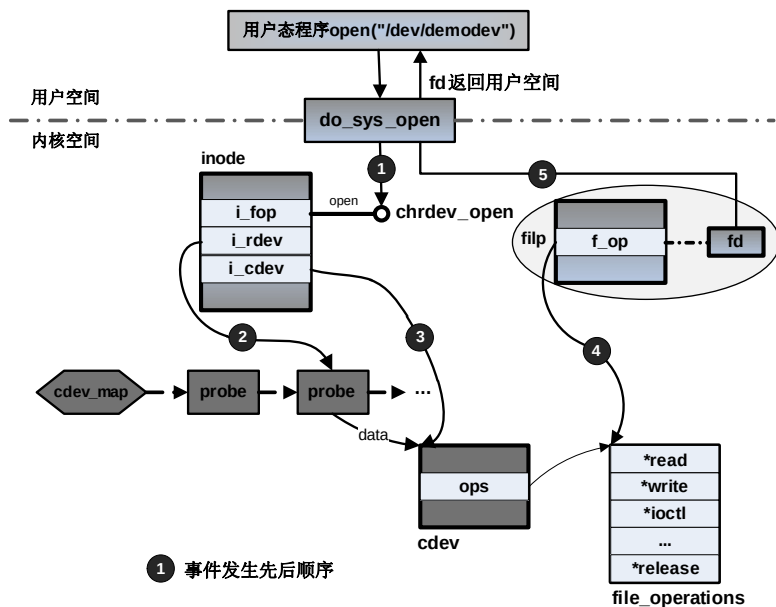


图 2-10 开一个字符设备节点的功能流程

图中，当应用程序打开一个设备文件时，将通过系统调用 `sys_open` 进入内核空间。在内核空间将主要由 `do_sys_open` 函数负责发起整个设备文件打开操作，它首先要获得该设备文件所对应的 `inode`，然后调用其中的 `i_fop` 函数，对字符设备节点的 `inode` 而言，`i_fop` 函数就是 `chrdev_open`（图中标号 1 的线段），后者通过 `inode` 中的 `i_rdev` 成员在 `cdev_map` 中查找该设备文件所对应的设备对象 `cdev`（图中标号 2 的线段），在成功找到了该设备对象之后，将 `inode` 的 `i_cdev` 成员指向该字符设备对象（图中标号 3 的线段），这样下次再对该设备文件节点进行打开操作时，就可以直接通过 `i_cdev` 成员得到设备节点所对应的字符设备对象，而无需再通过 `cdev_map` 进行查找。内核在每次打开一个设备文件时，都会产生一个整型的文件描述符 `fd` 和一个新的 `struct file` 对象 `filp` 来跟踪对该文件的这一次操作，在打开设备文件时，内核会将 `filp` 和 `fd` 关联起来，同时会将 `cdev` 中的 `ops` 赋值给 `filp->f_op`（图中标号 4 的线段）。最后，`sys_open` 系统调用将设备文件描述符 `fd` 返回到用户空间，如此在用户空间对后续的文件操作 `read`、`write` 和 `ioctl` 等函数的调用，将会通过该 `fd` 获得文件所对应的 `filp`，根据 `filp` 中的 `f_op` 就可以调用到该文件所对应的设备驱动上实现的函数。

通过以上过程，我们看到了设备号在其中的重要作用。当设备驱动程序通过 `cdev_add` 把一个字符设备对象加入到系统时，需要一个设备号来标记该对象在 `cdev_map` 中的位置信息。当我们在用户空间通过 `mknod` 来生成一个设备文件节点时，也需要在命令行中提供设备号的信息，内核会将该设备号信息记录到设备文件节点所对应 `inode` 的 `i_rdev` 成员中。当我们的应用程序打开一个设备文件时，系统将会根据设备文件对应的 `inode->i_rdev` 信息在 `cdev_map` 中寻找设备。所以在这个过程中务必要保证设备文件节点的 `inode->i_rdev` 数据和设备驱动程序使用的设备号完全一致，否则就会发生严重问题。对应到现实世界的操作，那就是在用 `mknod` 生成设备节点时所提供的设备号信息一定要与设备驱动程序中分配使用的设备号一致。

在上述 `open` 一个设备文件的基础上，接下来不妨看看它的相反操作 `close`。有了前面对 `open` 操作技术细节讨论所打下的良好基础，现在理解起 `close` 并不困难，在此读者也正好可以看看用户空间 `open` 函数返回的文件描述符 `fd` 如何被 `close` 等函数使用。

用户空间 `close` 函数的原型为：

```
int close(unsigned int fd);
```

针对 `close` 的系统调用函数为 `sys_close`，这里将其核心代码重新整理如下：

```
<fs/open.c>
```

```
int sys_close(unsigned int fd)
{
    struct file * filp;
    struct files_struct *files = current->files;
    struct fdtable *fdt;
    int retval;
```

```

...
fdt = files_fdttable(files);
...
filp = fdt->fd[fd];
...
retval = filp_close(filp, files);
...
return retval;
}

```

从 fd 得到 filp 这段代码，请读者参考本章 2-9。接下来调用 filp_close 函数，close 函数的大部分秘密都隐藏在其中，有必要看看其主要代码片段：

<fs/open.c>

```

int filp_close(struct file *filp, fil_owner_t id)
{
    int retval = 0;

    if (!file_count(filp)) {
        printk(KERN_ERR "VFS: Close: file count is 0\n");
        return 0;
    }

    if (filp->f_op && filp->f_op->flush)
        retval = filp->f_op->flush(filp, id);
    ...
    fput(filp);
    return retval;
}

```

if (!file_count(filp)) 用来判断 filp 中的 f_count 成员是否为 0，如果针对同一个设备文件 close 的次数多于 open 次数，就会出现这种情况，此时函数直接返回 0，因为实质性的工作都被前面的 close 做完了。接下来的情况有点意思，如果设备驱动程序定义了 flush 函数，那么在 release 函数被调用前，会首先调用 flush，这是为了确保在把文件关闭前缓存在系统中的数据被真正写回到硬件中。字符设备很少会出现这种情况，因为这种设备的慢速 I/O 特性决定了它无须使用这种缓冲机制来提升系统性能，但是块设备就不一样了，比如 SCSI 硬盘会和系统进行大量数据的传输，为此内核为块设备驱动程序设计了高速缓存机制，这种情况下为了保证文件数据的完整性，必须在文件关闭前将高速缓存中的数据写回到磁盘中。不过这是后话了，块设备驱动程序的这种机制将在“块设备驱动程序”一章中讨论。

函数的最后调用 fput，貌似很简单的一个函数，其实内涵却很丰富：

<fs/file_table.c>

```

void fput(struct file *file)
{

```

```

        if (atomic_long_dec_and_test(&file->f_count))
            __fput(file);
    }

```

函数中的那个 `atomic_long_dec_and_test` 是个体体系架构相关的原子测试操作，就是说，如果 `file->f_count` 的值为 1，那么它将返回 `true`，这意味着可以真正关闭当前的文件了，所以 `__fput` 将被调用，并最终完成文件关闭的任务，它的一些关键调用节点如下所示：

```

<fs/file_table.c>
-----
static void __fput(struct file *file)
{
    ...
    if (unlikely(file->f_flags & FASYNC)) {
        if (file->f_op && file->f_op->fasync)
            file->f_op->fasync(-1, file, 0);
    }
    if (file->f_op && file->f_op->release)
        file->f_op->release(inode, file);
    ...
    fops_put(file->f_op);
    file_free(file);
}

```

注意上面的 `FASYNC` 标志位，在本书后面的章节会讨论到 `file_operations` 中的一些常用的函数实现。然后函数调用到了设备驱动程序中提供的 `release` 函数，接下来是一些系统资源的释放。可见，对于应用程序的一个 `close` 调用，并非必然对应着 `release` 函数的调用，只有在当前文件的所有副本都关闭之后，`release` 函数才会被调用。

2.8 本章小结

本章描述了字符设备驱动程序内核框架的技术细节。基本上可以看到，字符设备驱动内核框架的展开是按照两条线进行的：一条是设备与系统的关系，一个字符设备对象 `cdev` 通过 `cdev_add` 加入到系统中（由 `cdev_map` 所管理的哈希链表），此时设备号作为哈希索引值；另一条是设备与文件系统的关系，设备通过设备号以设备文件的形式向用户空间宣示其存在。这两条线间的联系通过文件系统接口去打开一个字符设备文件而建立：

- `mknod` 命令将为字符设备创建一个设备节点，`mknod` 的系统调用将会为此设备节点产生一个 `inode`，`mknod` 命令行中给出的设备号将被记录到 `inode->i_rdev` 中，同时 `inode` 的 `i_fop` 会将 `open` 成员指向 `chrdev_open` 函数。
- 当用户空间 `open` 一个设备文件时，`open` 函数通过系统进入内核空间。在内核空间，首先找到该设备节点所对应的 `inode`，然后调用 `inode->i_fop->open()`，我们知道这将导致

`chrdev_open` 函数被调用。同时, `open` 的系统调用还将产生一个(`fd`, `filp`)二元组来标识本次的文件打开操作, 这个二元组是一一对应的关系。

- `chrdev_open` 通过 `inode->i_rdev` 在 `cdev_map` 中查找 `inode` 对应的字符设备, `cdev_map` 中记录着所有通过 `cdev_add` 加入系统的字符设备。
- 当在 `cdev_map` 中成功查找到该字符设备时, `chrdev_open` 将 `inode->i_cdev` 指向找到的字符设备对象, 同时将 `cdev->ops` 赋值给 `filp->f_op`。
- 字符设备驱动程序负责实现 `struct file_operations` 对象, 在字符设备对象初始化时 `cdev_init` 函数负责将字符设备对象 `cdev->ops` 指向该 `file_operations` 对象。
- 用户空间对字符设备的后续操作, 比如 `read`、`write` 和 `ioctl` 等, 将通过 `open` 函数返回的 `fd` 找到对应的 `filp`, 然后调用 `filp->f_op` 中实现的各类字符设备操作函数。

以上就是内核为字符设备驱动程序设计的大体框架, 从中可以看到设备号在沟通用户空间的设备文件与内核中的设备对象之间所起的重要作用。

另外, 对于字符设备驱动程序本身而言, 核心的工作是实现 `struct file_operations` 对象中的各类函数, `file_operations` 结构中虽然定义了众多的函数指针, 但是现实中设备驱动程序并不需要为它的每一个函数指针都提供相应的实现。本书后面的“字符设备的高级操作”一章会详细讨论其中一些重要函数的作用和实现原理。

第 3 章

分配内存

在所有驱动程序所使用的内核设施中，分配并使用内存是最基本也是最重要的一个环节。本章将详细讨论驱动程序所使用的内核分配函数的实现机制，以透彻了解这些内存分配函数的幕后细节，进而开发可以更有效更安全地使用系统内存资源的驱动程序。内核中的这些内存分配函数都依赖于内核中一个复杂而重要的构件：内存管理。作为一本 Linux 驱动程序方面的书籍，不可能在这一章中详细讨论所有的内存管理细节，但是讲解内核分配函数的实现，又不可避免地要与内核中的内存管理模块打交道。为使读者能够理解分配函数的代码机制，笔者会概括性地介绍与代码相关的内存管理的实现原理，在这一过程中将尽量不涉及太多体系架构相关的细节，而引用内核源码树中这部分代码时的原则则是，尽可能只保留与驱动程序所使用到的内存分配函数代码实现相关的部分。

Linux 下对内存的管理总体上可以分为两大类：一是对物理内存的管理；二是对虚拟内存的管理。前者用于特定的平台构架上实际物理内存空间的管理，后者用于特定的处理器体系架构上虚拟地址空间的管理。

3.1 物理内存的管理

Linux 系统为了用统一的代码获得最大程度的兼容性，在对物理内存的定义方面，引入了内存节点（node）、内存区域（zone）和内存页（page）的概念。其对物理内存的管理总体上又可以分成两大部分：最底层实现的是页面级内存管理，然后是基于页面级管理之上的 slab 内存管理。

3.1.1 内存节点 node

内存节点的引入，是因为 Linux 系统为了最大程度的兼容性，将 UMA 系统和 NUMA 系统统一起来，对于 UMA 而言是只有一个内存节点的系统。

在计算机世界中，有两种物理内存管理模型被广泛使用，它们分别是：

(1) UMA（一致内存访问，Uniform Memory Access）模型，该模型的内存空间在物理上也

许是不连续的（比如空洞的存在），但所有的内存空间对系统中的处理器而言具有相同的访问特性，也即系统中所有处理器对这些内存的访问具有相同的速度。

(2) NUMA（非一致内存访问，Non-Uniform Memory Access）模型，使用这种模型的总是多处理器系统，系统中的各个处理器都有本地内存，处理器与处理器之间通过总线连接起来以支持对其他处理器本地内存的访问。与 UMA 模型不同的是，处理器访问本地内存的速度要快于对其他处理器本地内存的访问。

Linux 源码中以 struct pglist_data 数据结构来表示单个内存节点。对于 NUMA 模型，多个内存节点通过链表串联起来；UMA 模型因为只有一个内存节点，因而不存在这样的链表。

图 3-1 展示了两种内存模型的区别：

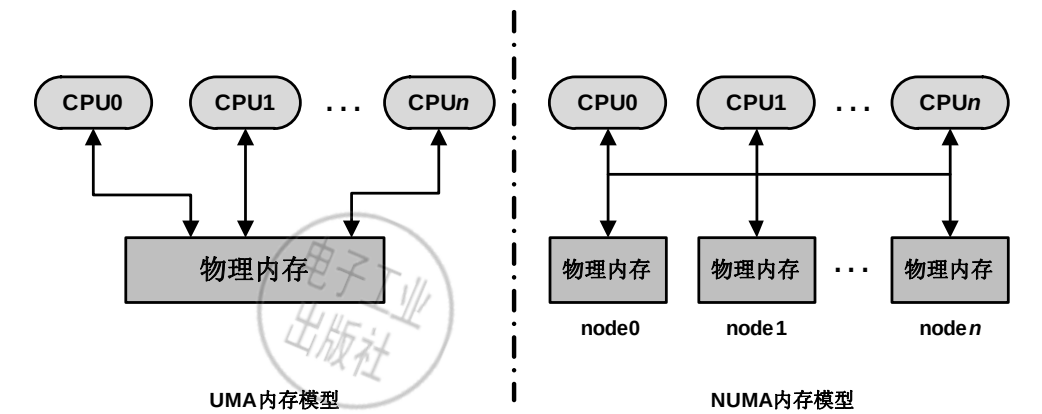


图 3-1 两种内存模型示意图

3.1.2 内存区域 zone

内存区域属于单个内存节点中的概念，考虑到系统的各个模块对分配的物理内存有不同的要求，比如 32 位 x86 体系架构下的 DMA 只能访问 16 MB 以下的物理内存空间，因此 Linux 又将每个内存节点管理的物理内存划分为不同的内存区域。在 Linux 源码中，以 struct zone 数据结构表示每一个内存区域，内存区域的类型用 zone_type 表示，是一枚枚举型变量：

```
<include/linux/mmzone.h>
enum zone_type {
#ifdef CONFIG_ZONE_DMA
    /*
     * 当有些设备不能使用所有的 ZONE_NORMAL 区域中的内存空间作
     * DMA 访问时，就可以使用 ZONE_DMA 所表示的内存区域。于是我
     * 们把这部分空间划分出来专门用做 DMA 访问的内存空间。
     * 该区域的空间访问是处理器体系架构相关的
     */

```

```

    *一些例子:
    *
    *体系架构          限制
    * -----
    * parisc, ia64, sparc  <4GB
    * s390                  <2GB
    * arm                   Various
    * alpha                 Unlimited or 0-16MB
    * i386, x86_64 and multiple other arches
    *                      <16MB
    */
    ZONE_DMA,
#endif
#ifdef CONFIG_ZONE_DMA32
    /*
     *x86_64 架构因为除了支持只能使用低于 16MB 空间的 DMA 设备外,
     *还支持可以访问 4GB 以下空间的 32 位 DMA 设备, 所以需要两个
     *ZONE_DMA 内存区域。
     */
    ZONE_DMA32,
#endif
    /*
     *常规内存访问区域由 ZONE_NORMAL 标识。如果 DMA 设备可以
     *在此区域作内存访问, 也可以使用本区域。
     */
    ZONE_NORMAL,
#ifdef CONFIG_HIGHMEM
    /*
     *高端内存区域用 ZONE_HIGHMEM 标识, 该区域无法从内核虚拟地址
     *空间直接作线性映射, 所以为访问该区域必须经内核作特殊的页映射。
     *比如在 i386 体系上, 内核空间 1GB, 除去其他一些开销, 能对物理
     *地址进行线性映射的空间大约只有 896MB。此时高于 896MB 以上的物理
     *地址空间就叫 ZONE_HIGHMEM 区域。
     */
    ZONE_HIGHMEM,
#endif
    ZONE_MOVABLE,
    __MAX_NR_ZONES
};

```

3.1.3 内存页

内存页是物理内存管理中的最小单位, 有时也叫页帧 (page frame)。Linux 会为系统物理内存的每个页都创建一个 struct page 对象, 系统用一全局变量 struct page *mem_map 来存放所有物理页 page 对象的指针。页的大小取决于系统中的内存管理单元 MMU (Memory

Management Unit), 后者用来将虚拟空间的地址转化为物理空间地址。鉴于 4 KB 大小的物理页对大多数体系架构而言都能很好地工作, 所以本书以 4 KB 页面为主要的讨论对象。

3.2 页面分配器 (page allocator)

本节开始讨论物理内存的分配, Linux 系统中对物理内存进行分配的核心建立在页面级的伙伴系统之上。在系统初始化期间, 伙伴系统负责对物理内存页面进行跟踪, 记录哪些是已经被内核使用的页面, 哪些是空闲页面。

有了伙伴系统就可以让系统分配单个物理页面或者连续的几个物理页面。驱动程序在内存分配时如果需要分配比较大的地址空间, 可以在这一层面利用页面分配器提供的接口函数。这些函数 (或者是宏) 只能分配 2 的整数次幂个连续的物理页, 返回值的形式各有不同, 对驱动程序而言, 理解这些函数的返回值其实更重要, 本章后面会详细讨论如何使用这些返回值。接下来将讨论页面分配级的函数, 为了使读者能更好地理解所讨论的内容, 图 3-2 给出了 mem_map、物理内存页面及系统虚拟地址之间关系的一个概略的示意图¹:

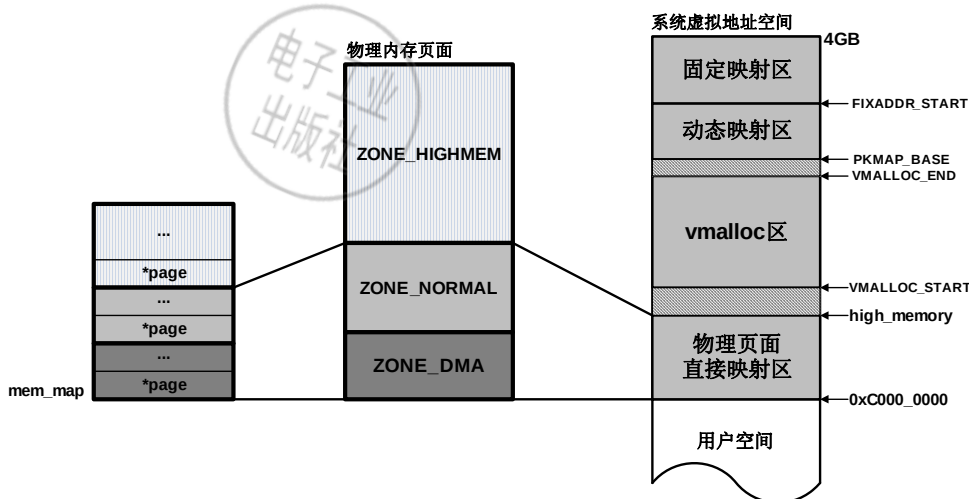


图 3-2 mem_map、物理页面及虚地址空间关系

图中, 每个物理页面都有一个 struct page 对象与之对应。根据内存使用及内核虚拟地址空间限制等因素, 内核将物理内存分为三个区²: ZONE_DMA、ZONE_NORMAL 和 ZONE_HIGHMEM。因为 mem_map 中每个 struct page 对象与物理页面之间严格的一一对应关系, 这使得在 mem_map 所引导的 struct page 实例中, 事实上也形成了三个区。

¹ 此图基于 32 位系统的 3GB/1GB 的经典布局, 如果是 64 位系统, 内存的布局与 32 位系统会有很大不同。

² 鉴于没有配置 CONFIG_HIGHMEM 的系统并不常见, 所以本书不考虑这种情况。

Linux 系统初始化期间，会将虚拟地址空间的物理页面直接映射区作线性地址映射到 ZONE_DMA 和 ZONE_NORMAL，这意味着如果页面分配器所分配的页面落在这两个 zone 中，那么对应的内核虚拟地址到物理地址的映射的页目录表项已经建立，而且是所谓的线性映射，也就是虚拟地址和物理地址之间只有一个差值（PAGE_OFFSET，也即图中的 0xC0000000）。

而如果页面分配器所分配的页面落在 ZONE_HIGHMEM 中，那么内核此时尚没有对该页面进行地址映射，因此，页面分配器的调用者（比如设备驱动程序等内核模块）在这种情况下需要做的事是，在内核虚拟地址空间的动态映射区或者固定映射区分配一个虚拟地址，然后映射到该物理页面上。当然内核提供了实现这些步骤的接口函数，内核模块只要调用相应的函数就可以了。

以上是页面分配器的大致工作原理，接下来开始具体讨论页面分配器所提供的接口函数，无论是对 UMA 还是 NUMA 系统而言，这些函数的接口是完全一致的。页面分配器函数的核心成员其实只有两个，分别是 alloc_pages 和 __get_free_pages，其他的一些函数则是在这二者的基础上通过调整某些参数而来。而 alloc_pages 和 __get_free_pages 最终都会调用到 alloc_pages_node，所以两者背后的实现原理完全一样，只是 __get_free_pages 不能在高端内存区分配页面，此外两者返回值的形式也有所区别。

3.2.1 gfp_mask

gfp_mask 并不是页面分配器函数，而只是这些页面分配函数中一个重要的参数，是个用于控制分配行为的掩码，并可以告诉内核应该到哪个 zone 中分配物理内存页面。这里将一些常见的 gfp_mask 掩码含义说明如下，然后重点讨论内核模块中使用最多的 GFP_KERNEL 和 GFP_ATOMIC：

```
<include/linux/gfp.h>
```

```
-----
#define __GFP_DMA                ((__force gfp_t)0x01u)
#define __GFP_HIGHMEM            ((__force gfp_t)0x02u)
#define __GFP_DMA32              ((__force gfp_t)0x04u)
#define __GFP_MOVABLE            ((__force gfp_t)0x08u)

#define __GFP_WAIT                ((__force gfp_t)0x10u)
#define __GFP_HIGH                ((__force gfp_t)0x20u)
#define __GFP_IO                  ((__force gfp_t)0x40u)
#define __GFP_FS                  ((__force gfp_t)0x80u)
#define __GFP_COLD                ((__force gfp_t)0x100u)
#define __GFP_NOWARN              ((__force gfp_t)0x200u)
#define __GFP_REPEAT              ((__force gfp_t)0x400u)
#define __GFP_NOFAIL              ((__force gfp_t)0x800u)
#define __GFP_NORETRY             ((__force gfp_t)0x1000u)
```

```
#define __GFP_COMP                ((__force gfp_t)0x4000u)
#define __GFP_ZERO                ((__force gfp_t)0x8000u)
#define __GFP_NOMEMALLOC         ((__force gfp_t)0x10000u)
#define __GFP_HARDWALL           ((__force gfp_t)0x20000u)
```

`__GFP_DMA`

在 `ZONE_DMA` 标识的内存区域中查找空闲页。

`__GFP_HIGHMEM`

在 `ZONE_HIGHMEM` 标识的内存区域中查找空闲页。

`__GFP_DMA32`

在 `ZONE_DMA32` 标识的内存区域中查找空闲页。

`__GFP_MOVABLE`

内核将分配的物理页标记为可移动的。

`__GFP_WAIT`

当前正在向内核申请页分配的进程可以被阻塞，意味着调度器可以在此请求期间调度另外一个进程执行。

`__GFP_HIGH`

内核允许使用紧急分配链表中的保留内存页。该请求必须以原子方式完成，意味着请求过程不允许被中断。

`__GFP_IO`

内核在查找空闲页的过程中可以进行 I/O 操作，如此内核可以将换出的页写到硬盘。

`__GFP_FS`

查找空闲页的过程中允许执行文件系统相关操作。

`__GFP_COLD`

从非缓存的“冷页”中分配。

`__GFP_NOWARN`

禁止分配失败时的告警。

`__GFP_REPEAT`

如果分配行为失败，可以自动尝试再次分配。尝试若干次后会终止。

`__GFP_NOFAIL`

分配失败后一直重试，直到分配成功为止，分配函数的调用者无法处理分配失败的情形。根据 2.6.39 版本内核中的源码注释（`__GFP_NOFAIL` is not to be used in new code.），以后新代码将不再使用该掩码。

`__GFP_NORETRY`

如果分配失败，不会进行重试操作。

`__GFP_COMP`

增加复合页元数据。

`__GFP_ZERO`

用 0 填充成功分配出来的物理页。

`__GFP_NOMEMALLOC`

不要使用仅限紧急分配使用的保留分配链表。

`__GFP_HARDWALL`

只能在当前进程允许运行的各个 CPU 所关联的节点分配内存。该标志只有在 NUMA 系统上才有意义。

通常意义上（并非严格规定），这些以“__”打头的 GFP 掩码只限于在内存管理组件内部的代码使用，对于提供给外部的接口，比如驱动程序中所使用的页面分配函数，`gfp_mask` 掩码以“GFP_”的形式出现，而这些掩码基本上就是上面提到的掩码的组合，例如内核为外部模块提供的最常使用的几个掩码如下：

```
<include/linux/gfp.h>
```

```
-----
#define GFP_ATOMIC      (__GFP_HIGH)
#define GFP_NOIO        (__GFP_WAIT)
#define GFP_NOFS        (__GFP_WAIT | __GFP_IO)
#define GFP_KERNEL      (__GFP_WAIT | __GFP_IO | __GFP_FS)
#define GFP_USER        (__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HARDWALL)
#define GFP_HIGHUSER    (__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HARDWALL | \
                        __GFP_HIGHMEM)
#define GFP_DMA         __GFP_DMA
```

`GFP_ATOMIC`

内核模块中最常使用的掩码之一，用于原子分配，也是上面几个掩码中唯一不带 `__GFP_WAIT` 的。此掩码告诉页面分配器，在分配内存页时，绝对不能中断当前进程或者把当前进程移出调度器。必要的情况下可以使用仅限紧急情况使用的保留内存页。在驱动

程序中，一般在中断处理例程或者非进程上下文的代码中使用 `GFP_ATOMIC` 掩码进行内存分配，因为这两种情况下分配都必须保证当前进程不能睡眠。

`GFP_KERNEL`

内核模块中最常使用的掩码之一，带有该掩码的内存分配可能导致当前进程进入睡眠状态。

`GFP_USER`

用于为用户空间分配内存页，可能引起进程的休眠。

`GFP_NOIO`

`GFP_NOFS`

都带有 `__GFP_WAIT`，因此可以被中断。前者在分配过程中禁止 I/O 操作，后者则是禁止文件系统相关的函数调用。

`GFP_HIGHUSER`

对 `GFP_USER` 的一个扩展，可以使用非线性映射的高端内存。

`GFP_DMA`

限制页面分配器只能在 `ZONE_DMA` 域中分配空闲物理页面，用于分配适用于 DMA 缓冲区的内存。

对于以上掩码，内核模块开发人员其实更关心的是页面分配器将到哪个域中分配物理页面，在页面分配过程中这实际上是由 `gfp_zone` 函数根据上述掩码来指定，如果没有在 `gfp_mask` 中明确指定 `__GFP_DMA` 或者是 `__GFP_HIGHMEM`，那么默认是在 `ZONE_NORMAL` 中分配物理页，如果 `ZONE_NORMAL` 中现有空闲页不足以满足当前的分配，那么页分配器会到 `ZONE_DMA` 域中查找空闲页，而不会到 `ZONE_HIGHMEM` 中查找。小结一下，这里的分配域优先次序是：

`__GFP_HIGHMEM`。先在 `ZONE_HIGHMEM` 域中查找空闲页，如果无法满足当前分配，页分配器将回退到 `ZONE_NORMAL` 域中继续查找，如果依然无法满足当前分配，分配器将回退到 `ZONE_DMA` 域，或者成功或者失败。

没有 `__GFP_NORMAL` 这样的掩码，但是前面已经提到，如果 `gfp_mask` 中没有明确指定 `__GFP_HIGHMEM` 或者是 `__GFP_DMA`，默认就相当于 `__GFP_NORMAL`，优先在 `ZONE_NORMAL` 域中分配，其次是 `ZONE_DMA` 域。

`__GFP_DMA`。只能在 `ZONE_DMA` 中分配物理页面，如果无法满足，则分配失败。

设备驱动程序中最常使用的是 `GFP_KERNEL` 与 `GFP_ATOMIC`，两者中都没有明确指定内

存域的标识符，这意味着使用它们的页分配器只能在 ZONE_NORMAL 和 ZONE_DMA 中分配物理页面。

3.2.2 alloc_pages

在源码中，alloc_pages 以宏的形式出现，其定义为：

```
<include/linux/gfp.h>
-----
#define alloc_pages(gfp_mask, order) \
    alloc_pages_node(numa_node_id(), gfp_mask, order)

static inline struct page *alloc_pages_node(int nid, gfp_t gfp_mask,
                                             unsigned int order)
{
    /* Unknown node is current node */
    if (nid < 0)
        nid = numa_node_id();

    return __alloc_pages(gfp_mask, order, node_zonelist(nid, gfp_mask));
}
```

__alloc_pages 函数负责分配 2^{order} 个连续的物理页面并返回起始页的 struct page 实例。在调用这个函数时，如果 gfp_mask 中没有明确指定 __GFP_HIGHMEM，那么分配的物理页面必然来自 ZONE_NORMAL 或者 ZONE_DMA，由于这两个域中内核已经在初始化阶段就为之建立了映射关系，所以内核模块可以使用 page_address 来获得对应页面的内核虚拟地址 KVA (Kernel Virtual Address)。因为是线性映射，所以此时获得 KVA 很简单，这里用伪代码将 page_address 的原理大致表述如下：

```
unsigned long pfn = (unsigned long)(page - mem_map); //获得页帧号
unsigned long pg_pa = pfn << PAGE_SHIFT; //获得页面的物理地址
return (void*)__va(pg_pa); //返回物理页面对应的 KVA, KVA=PAGE_OFFSET+pg_pa
```

如果在调用 alloc_pages 时在 gfp_mask 中指定了 __GFP_HIGHMEM，那么页分配器将优先在 ZONE_HIGHMEM 域中分配物理页，但也不排除因为 ZONE_HIGHMEM 没有足够的空闲页导致页面来自 ZONE_NORMAL 与 ZONE_DMA 域的可能性。对于新分配出的高端物理页面，由于内核尚未在页表中为之建立映射关系，所以此时需要：1.在内核的动态映射区分配一个 KVA；2.通过操作页表，将 1 中的 KVA 映射到该物理页面上。内核为此提供了一个函数 kmap：

```
<arch/x86/mm/highmem_32.c>
-----
void *kmap(struct page *page)
{
    might_sleep();
    if (!PageHighMem(page))
        return page_address(page);
}
```

```

    return kmap_high(page);
}

```

首先，该函数在执行过程中可能睡眠，所以不能用在中断处理等上下文中。其次，可以看到它用 `PageHighMem(page)` 来判断页面是否真的来自高端内存，如果不是，则用 `page_address` 来返回页面所对应的 KVA，否则将调用 `kmap_high` 在内核虚拟地址空间的动态映射区或者固定映射区分配一个新的 KVA 并将其映射到物理页面上，之后将该 KVA 返回给调用者。因为涉及页表的操作，所以从高端内存分配物理页对系统的开销是比较大的。

与 `kmap` 行为相反的函数是 `kunmap`，在 x86 平台上的定义如下：

```

<arch/x86/mm/highmem_32.c>
-----
void kunmap(struct page *page)
{
    if (in_interrupt())
        BUG();
    if (!PageHighMem(page))
        return;
    kunmap_high(page);
}

```

函数将在页表项中拆除对 `page` 的映射，同时将来自动态映射区中的 KVA 释放出去，这样该 KVA 可以被再次映射到别的物理页面。

内核针对 `kmap` 函数可能睡眠的情形提供了另一个备选的功能 `kmap_atomic`，该函数的执行是原子的，而且比 `kmap` 要快，此处不再详细讨论。

另一个页面分配函数是 `alloc_page`，只用于分配一个物理页面。`alloc_page(gfp_mask)` 是 `order=0` 时 `alloc_pages` 的简化形式，只分配单个页面。

如果系统中没有足够的空闲页面来满足 `alloc_pages` 的分配，函数将返回 `NULL`，内核模块需要仔细检查 `alloc_pages` 函数的返回值，以作出适当的应对。

3.2.3 `__get_free_pages`

`__get_free_pages` 函数在内核源码中的定义为：

```

<mm/page_alloc.c>
-----
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)
{
    struct page *page;

    /*
     * __get_free_pages() returns a 32-bit address, which cannot represent
     * a highmem page
     */
}

```

```

VM_BUG_ON((gfp_mask & __GFP_HIGHMEM) != 0);

page = alloc_pages(gfp_mask, order);
if (!page)
    return 0;
return (unsigned long) page_address(page);
}

```

函数负责分配 2^{order} 个连续的物理页面，返回起始页面所在内核线性地址。函数内部调用 `alloc_pages` 负责实际的页面分配工作。从函数源码中可以看到，`__get_free_pages` 不能从高端内存中分配物理页，`VM_BUG_ON` 宏在 `CONFIG_DEBUG_VM` 定义的情形下可以捕捉到这一错误，如果 `CONFIG_DEBUG_VM` 没有定义，且调用者在 `gfp_mask` 中设置了 `__GFP_HIGHMEM` 掩码，那么 `__get_free_pages` 返回 0。在正常情况下，`__get_free_pages` 从低端内存区中分配 2^{order} 个连续物理页面，并通过 `page_address` 来返回这些页面中起始页面的内核线性地址。

如果内核模块只想分配单个物理页面，那么可以使用 `__get_free_page(gfp_mask)`，它是 `order=0` 时 `__get_free_pages` 的简化形式。

3.2.4 get_zeroed_page

`get_zeroed_page` 用于分配一个物理页同时将页面对应的内容填充为 0，函数返回页面所在的内核线性地址。其定义为：

```

<mm/page_alloc.c>
-----
unsigned long get_zeroed_page(gfp_t gfp_mask)
{
    return __get_free_pages(gfp_mask | __GFP_ZERO, 0);
}

```

3.2.5 __get_dma_pages

`__get_dma_pages` 用于从 `ZONE_DMA` 区域中分配物理页，返回页面所在线性地址。其定义为：

```

<include/linux/gfp.h>
-----
#define __get_dma_pages(gfp_mask, order) \
    __get_free_pages((gfp_mask) | GFP_DMA, (order))

```

前面讨论了页面分配器提供的最常用的接口函数，现在看看如果要释放这些被分配的页面应该怎么做。针对 `alloc_pages` 和 `__get_free_pages`，内核提供的释放函数分别是 `__free_pages` 和 `free_pages`，其背后的实现原理其实都一样（`free_pages` 内部最终调用 `__free_pages` 来完成页面的释放工作），只不过在函数的原型定义方面有所区分。`__free_pages` 的定义是：

<mm/page_alloc.c>

```
void __free_pages(struct page *page, unsigned int order)
{
    if (put_page_testzero(page)) {
        if (order == 0)
            free_hot_cold_page(page, 0);
        else
            __free_pages_ok(page, order);
    }
}
```

调用__free_pages 时，参数 page 应该是由 alloc_pages 返回的 page 对象指针，order 是分配阶，alloc_pages 和__free_pages 应该一致。

而 free_pages 的定义则是：

<mm/page_alloc.c>

```
void free_pages(unsigned long addr, unsigned int order)
{
    if (addr != 0) {
        VM_BUG_ON(!virt_addr_valid((void *)addr));
        __free_pages(virt_to_page((void *)addr), order);
    }
}
```

调用 free_pages 时，参数 addr 应该是__get_free_pages 返回的内核线性虚拟地址。

3.3 slab 分配器 (slab allocator)

上节提到的页面分配器用于连续物理页面的分配，驱动程序中可以使用这些函数来分配一大块连续的内存空间。然而只是有页面级的内存分配函数还不够，因为很多情况下我们需要分配比 4 KB 要小很多的物理地址空间，比如只有几十或者几百个字节，如果对这样的地址空间需求也分配一个完整的物理页，显然会对物理内存的使用造成巨大浪费。基于这一需求，Linux 系统在物理页分配的基础上实现了对更小内存空间进行管理的 slab、slob 和 slub 分配器。slab 是 Linux 内核最早推出的小内存分配方案，slob 和 slub 分配器则是 Linux 2.6 内核开发期间新增的 slab 分配器的替代品，主要针对大型系统和嵌入式系统。本书并不会详细讨论这些分配器的具体实现细节，所以下文中将 slab、slob 和 slub 统称 slab 分配器。

slab 分配器的原理是很简单的，但是具体到代码层面，由于牵涉到多方面的考虑，包括最大兼容性、优化以及调试等，这部分代码相当烦冗晦涩。本书因为是从驱动程序的角度出发，所以会侧重于驱动程序使用的内存分配的接口函数方面，对这部分内容只需要在相对高点的层面了解 slab 分配器的实现思想就足够了。

slab 分配器的基本思想是，先利用页面分配器分配出单个或者一组连续的物理页面，然后在此基础上将整块页面分割成多个相等的小内存单元，以满足小内存空间分配的需要。当然，为了有效地管理这些小的内存单元并保证极高的内存使用速度和效率，内核代码的复杂度要远远超出其基本思想所展现的面貌。但是这并不影响我们以框架的形式来揭示这些代码背后最实质性的东西：抛开烦冗的细节，以粗线条的形式勾勒出 slab 分配器的主干部分。

3.3.1 管理 slab 的数据结构

为了对 slab 进行管理，内核必须定义相关的数据结构，其中最重要的两个数据结构是 `struct kmem_cache` 和 `struct slab`，这两个数据结构是 slab 分配器的基石，要了解清楚 slab 分配器的架构原理，必须要了解这两个数据结构的具体用途。

在 Linux 内核源码中，这两个数据结构定义如下（为了突显 slab 分配器的框架结构，删除了一些调试相关的成员）：

○ `struct kmem_cache`

```
<include/linux/slab_def.h>
-----
struct kmem_cache {
    /* 1) per-cpu data, touched during every alloc/free */
    struct array_cache *array[NR_CPUS];

    /* 2) Cache tunables. Protected by cache_chain_mutex */
    unsigned int batchcount;
    unsigned int limit;
    unsigned int shared;

    unsigned int buffer_size;
    u32 reciprocal_buffer_size;

    /* 3) touched by every alloc & free from the backend */

    unsigned int flags;    /* constant flags */
    unsigned int num;      /* # of objs per slab */

    /* 4) cache_grow/shrink */
    /* order of pgs per slab (2^n) */
    unsigned int gfporder;

    /* force GFP flags, e.g. GFP_DMA */
    gfp_t gfpflags;

    size_t colour;        /* cache colouring range */
    unsigned int colour_off; /* colour offset */
    struct kmem_cache *slabp_cache;
```

```

    unsigned int slab_size;
    unsigned int dflags;          /* dynamic flags */

    /* constructor func */
    void (*ctor)(void *obj);

    /* 5) cache creation/removal */
    const char *name;
    struct list_head next;

    /*
     * We put nodelists[] at the end of kmem_cache, because we want to size
     * this array to nr_node_ids slots instead of MAX_NUMNODES
     * (see kmem_cache_init())
     * We still use [MAX_NUMNODES] and not [1] or [0] because cache_cache
     * is statically defined, so we reserve the max number of nodes.
     */
    struct kmem_list3 *nodelists[MAX_NUMNODES];
    /*
     * Do not add fields after nodelists[]
     */
};

```

该结构中的最后一个成员 `struct kmem_list3` 的定义为：

<mm/slab.c>

```

struct kmem_list3 {
    struct list_head slabs_partial; /* partial list first, better asm code */
    struct list_head slabs_full;
    struct list_head slabs_free;
    unsigned long free_objects;
    unsigned int free_limit;
    unsigned int colour_next;      /* Per-node cache coloring */
    spinlock_t list_lock;
    struct array_cache *shared;     /* shared per node */
    struct array_cache **alien;     /* on other nodes */
    unsigned long next_reap;       /* updated without locking */
    int free_touched;              /* updated without locking */
};

```

○ struct slab

<mm/slab.c>

```

struct slab {
    struct list_head list;
    unsigned long colouroff;
    void *s_mem;                  /* including colour offset */
};

```

```

    unsigned int inuse;      /* num of objs active in slab */
    kmem_bufctl_t free;
    unsigned short nodeid;
};

```

上述结构中一些常见的成员变量说明如下：

`unsigned int gfporder`

指明该 `kmem_cache` 中每个 `slab` 占用的页面数量，为 2^{gfporder} 个页。

`gfp_t gfpflags`

影响通过伙伴系统寻找空闲页时的行为，见本章前面“页面分配器”一节。

`const char *name`

`kmem_cache` 的名字，会导出到 `/proc/slabinfo` 中。

`struct list_head next`

将该 `kmem_cache` 加入到 `cache_chain` 链表中。

`void (*ctor)(void *obj)`

构造函数。当在 `kmem_cache` 中分配一个新的 `slab` 时，用来初始化 `slab` 中的所有内存对象。

`struct list_head slabs_partial`

将 `kmem_cache` 中所有的半空闲的 `slab` 加入到该链表中。

`struct list_head slabs_full`

将 `kmem_cache` 中所有已经满员的 `slab` 加入到该链表中。

`struct list_head slabs_free`

将 `kmem_cache` 中所有完全空闲的 `slab` 加入到该链表中

为了让读者更好地理解后面要讲述的东西，先来看一张描述 `slab` 分配器实现原理的框架图（图 3-3）。

`struct kmem_cache` 和 `struct slab` 在一个 `slab` 分配器中形成分级管理，图中的 `struct kmem_cache` 用于管理其下所有的 `struct slab`，它通过三个链表成员 `struct list_head slabs_full`、`struct list_head slabs_partial` 和 `struct list_head slabs_free`，将其下所有 `struct slab` 实例加入链表。其中，`slabs_full` 表示链表中每一个 `slab` 所在的物理内存页都已经分配完；`slabs_partial`

表示链表中每一个 slab 所在的物理内存页还有部分空闲空间可继续用于分配；slabs_free 表示链表中每一个 slab 所在的物理内存页完全空闲，没有分配任何内存对象。

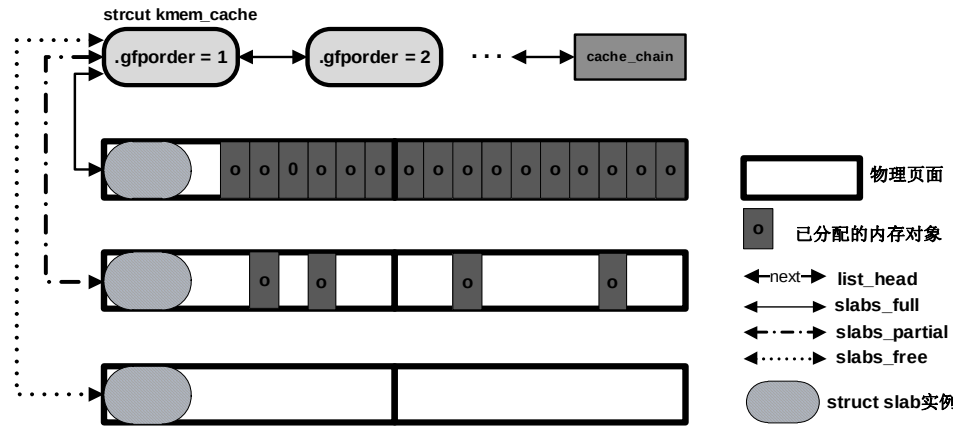


图 3-3 slab 分配器框架图

struct slab 结构用于管理一块连续的物理页面中内存对象的分配。在实际的代码实现中，struct slab 结构的实例存放位置有两种：一是像图 3-3 那样，将 struct slab 的实例放在物理页面首页的开始处；二是放在物理页面的外部(通过下面要讨论的 kmalloc 函数来分配 struct slab 的实例)。内核将从性能优化的角度出发来决定 slab 实例的存放位置，源码中的 CFLGS_OFF_SLAB 宏用于表示 slab 对象存放于外部。

系统中的 slab 分配器并不是孤立的，内核通过一个全局的双向链表 cache_chain 将每一个 slab 分配器链接起来(通过 slab 分配器 kmem_cache 中的 next 成员，后者是个 struct list_head 型变量)。

○ cache_cache

从图 3-3 可以看出，对于每一个 slab 分配器，都需要一个 struct kmem_cache 实例，那么，在 slab 系统尚未完全建立起来时，kmem_cache 实例所在的空间从哪里分配呢？答案是系统在初始化期间提供了一个特殊的 slab 分配器 cache_cache，专门用来分配 struct kmem_cache 空间。

因为 cache_cache 在 slab 系统还未完备时就被创造了出来，所以这个 struct kmem_cache 结构采用了静态内存分配的方法。在 Linux 源码中，cache_cache 定义如下：

```
<mm/slab.c>
-----
static struct kmem_cache cache_cache = {
    .batchcount = 1,
    .limit = BOOT_CPUCACHE_ENTRIES,
    .shared = 1,
    .buffer_size = sizeof(struct kmem_cache),
}
```

```

        .name = "kmem_cache",
    };

```

这个最早的 `kmem_cache` 有个不错的名字“`kmem_cache`”，告诉我们它所领衔的 slab 分配器专门用来分配 `struct kmem_cache` 这样的内存对象，`.buffer_size = sizeof(struct kmem_cache)` 则为这个论断提供了进一步的佐证³。

因为系统在初始化 `cache_cache` 时伙伴系统已经完备，所以如果采用把 `struct slab` 放在页面内部的方式，这个 slab 分配器就可以工作了。

○ size_cache

很多书中把 `cache_sizes` 叫做通用 cache，Linux 源码中称之为 `general cache` 或者 `default cache`，想到前面的 `cache_cache`，也许把它叫做 `size_cache` 更确切点。这个 `size_cache` 是下面要讨论的 `kmalloc` 函数实现的基础。

首先看几个数据结构的定义（这些定义中省略了一些元素，笔者认为略去这些元素不会影响读者对 `size_cache` 机制的理解）：

```

<include/linux/slab_def.h>
-----
struct cache_sizes {
    size_t          cs_size;
    struct kmem_cache *cs_cachep;
}
<mm/slab.c>
struct cache_sizes malloc_sizes[] = {
    { .cs_size = 32 },
    { .cs_size = 64 },
    { .cs_size = 128 },
    { .cs_size = 256 },
    { .cs_size = 512 },
    { .cs_size = 1024 },
    { .cs_size = 2048 },
    { .cs_size = 4096 },
    { .cs_size = 8192 },
    { .cs_size = 16384 },
    { .cs_size = 32768 },
    { .cs_size = 65536 },
    { .cs_size = 131072 },
    ...
    { .cs_size = ~0UL },

```

³ 当然在实际的代码当中，因为考虑到性能等方面的因素，`buffer_size` 等一些成员会在系统初始化期间通过调用 `kmem_cache_init` 函数被重新初始化一遍。

```
};
```

在系统初始化期间，内核委托 `kmem_cache_init` 函数遍历 `malloc_sizes` 数组，对应每个元素，都调用 `kmem_cache_create` 函数在 `cache_cache` 中分配一个 `struct kmem_cache` 实例，并将实例所在的地址存放在元素的 `cs_cachep` 变量中，核心代码如下：

```
<mm/slab.c>
```

```
void __init kmem_cache_init(void)
{
    struct cache_sizes *sizes = malloc_sizes;
    struct cache_names *names = cache_names;
    ...
    while (sizes->cs_size != ULONG_MAX) {
        /*
         * For performance, all the general caches are L1 aligned.
         * This should be particularly beneficial on SMP boxes, as it
         * eliminates "false sharing".
         * Note for systems short on memory removing the alignment will
         * allow tighter packing of the smaller caches.
         */
        if (!sizes->cs_cachep) {
            sizes->cs_cachep = kmem_cache_create(names->name,
                sizes->cs_size,
                ARCH_KMALLOC_MINALIGN,
                ARCH_KMALLOC_FLAGS|SLAB_PANIC,
                NULL);
        }
#ifdef CONFIG_ZONE_DMA
        sizes->cs_dmacachep = kmem_cache_create(
            names->name_dma,
            sizes->cs_size,
            ARCH_KMALLOC_MINALIGN,
            ARCH_KMALLOC_FLAGS|SLAB_CACHE_DMA|
                SLAB_PANIC,
            NULL);
#endif
        sizes++;
        names++;
    }
    ...
}
```

这段代码实现的功能非常明显：在 `while` 循环中遍历 `malloc_sizes` 数组，对每一元素调用 `kmem_cache_create` 函数创建 `kmem_cache` 对象。

如此，系统中的 `size_cache` 在初始化完成后的形态将如图 3-4 所示：

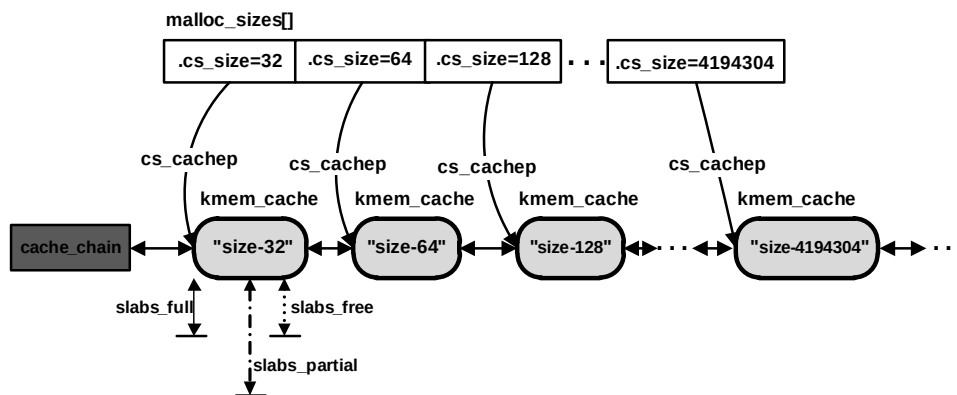


图 3-4 size_cache 的初始化

图中对应 `malloc_sizes` 数组中的每个元素,都产生了一个 slab 分配器用以分配大小为 `cs_size` 的内存空间。可以看到,初始化完成后,因为还没有在其上进行内存分配,所以还没有 slab 对象产生, `kmem_cache` 中的 `slabs_full`、`slabs_partial` 和 `slabs_free` 三个链表指针都为空。此情况将一直延续到有内核模块调用 `kmalloc` 函数。

3.3.2 kmalloc 与 kcalloc

`kmalloc` 函数是驱动程序中使用最多的一个内存分配函数,它分配出来的内存空间在物理上是连续的,函数不负责把分配出的内存空间中的内容清零,换言之,分配出来的内存空间保留有原来的数据。`kmalloc` 函数的原型为

```
void *kmalloc(size_t size, gfp_t flags)
```

参数 `size` 用来表示想要分配的内存空间的大小, `flags` 是分配掩码,同前面讨论的页面分配器中的 GFP 掩码完全一样,掩码会影响到伙伴系统对空闲内存页的查找行为。

这个函数建立在 slab 分配器基础之上,它的实现主要围绕 `size_cache` 展开。虽然在实际的 Linux 源码中,函数的实现比较复杂,但是我们可以用下面这段简明的代码来揭示 `kmalloc` 函数的实现原理,改写后的代码突显了 `kmalloc` 函数的主体脉络:

```
void * kmalloc(size_t size, int flags)
{
    struct cache_sizes *csizp = malloc_sizes;
    struct kmem_cache *cachep;

    while (size > csizp->cs_size)
        csizp++;

    cachep = csizp->cs_cachep;
    return kmem_cache_alloc(cachep, flags);
}
```



```
}
```

函数利用参数 `size` 在 `malloc_sizes` 数组中查找，目的是找到在所有大于等于它的数列中的最小值。简单来说，假设 `size=20`，那么 `cs_size=32` 的元素满足需求；如果 `size=238`，那么 `cs_size=256` 满足需求。这点很容易理解：分配出的内存对象必须够大才能满足分配要求，但又不能太大，否则会造成内存资源的浪费。

找到了这样一个数组元素之后，也就获得了该元素所对应的 slab 分配器的 `kmem_cache` 对象 `cachep`（这些对象早在系统初始化期间就已经分配好，详见上一节）。

最后函数调用 `kmem_cache_alloc` 函数在 `cachep` 领衔的 slab 分配器中进行内存的分配。对于 `kmem_cache_alloc` 函数而言，大部分情况下，它都会返回 `cachep` 所对应的 slab 分配器中一个空闲的内存对象。但是，万一分配器中已经没有这样的空闲内存对象可用，则必须新建一个 slab，这意味着 slab 分配器需要利用下层的页面分配器来分配一段新的物理页面，此时发生的调用链是：`__cache_alloc()`→`__do_cache_alloc()`→`cache_alloc_refill()`→`cache_grow()`→`kmem_getpages()`→`alloc_pages_exact_node()`→`__alloc_pages()`。可以看到在这种情况下，slab 分配器最终会调用 `__alloc_pages` 去分配 2^{order} 个连续的物理页面。

在设备驱动程序等内核模块中调用 `kmalloc` 函数时，最后一个参数最常见的就是 `GFP_KERNEL` 和 `GFP_ATOMIC`，前面讨论过，这两个标志都会使得页面分配器在低端内存域中分配物理页面。读者也许会好奇，如果调用 `kmalloc` 时最后一个参数是 `__GFP_HIGHMEM`，`__alloc_pages()` 会到高端内存区去分配页面吗？答案是否定的。对于 slab 分配器而言，它只能在低端内存区分配物理页面。对应的代码来自上述调用链中的 `cache_grow()` 函数：

```
<mm/slab.c>
```

```
static int cache_grow(struct kmem_cache *cachep,
                     gfp_t flags, int nodeid, void *objp)
{
    gfp_t local_flags;
    ...
    BUG_ON(flags & GFP_SLAB_BUG_MASK);
    local_flags = flags & (GFP_CONSTRAINT_MASK|GFP_RECLAIM_MASK);
    ...
    if (!objp)
        objp = kmem_getpages(cachep, local_flags, nodeid);
    ...
}
```

函数中的宏 `GFP_SLAB_BUG_MASK` 定义如下：

```
<include/linux/gfp.h>
```

```
/* Do not use these with a slab allocator */
```

```
#define GFP_SLAB_BUG_MASK (__GFP_DMA32|__GFP_HIGHMEM|~__GFP_BITS_MASK)
```

所以,如果在调用 `kmalloc` 时传入 `__GFP_HIGHMEM` 或者 `__GFP_DMA32` 或者两者的组合,将触发代码中的 `BUG_ON`,后者在当前 Linux 源码中基本等同空操作,虽然 `BUG_ON` 可能不会触发 `kmalloc` 函数在执行时发生异常,但是这里体现了 slab 分配器的一个基本设计原则:底层的页面分配来自低端物理内存区域。更进一步地,后续的 `local_flags = flags & (GFP_CONSTRAINT_MASK|GFP_RECLAIM_MASK)` 将会清除掉 `__GFP_HIGHMEM` 或者 `__GFP_DMA32` 标志,所以即便内核模块使用 `kmalloc(64, __GFP_HIGHMEM)` 这样的调用形式来分配内存,函数依然会返回低端物理内存页面所对应的线性内核虚拟地址,而不是 `vmalloc` 区或者其他动态映射区的虚拟地址。

如果系统中没有足够多的内存,分配连续的物理页面会失败,对于内核空间而言,这种情况非常少见,但并非不可能。如果因内存不足而导致最终的分配失败, `kmalloc` 函数将返回 `NULL` 指针。所以函数的调用者需要仔细考量 `kmalloc` 函数的返回值以做出正确的应对。

`kzalloc` 函数是 `kmalloc` 在设置了 `__GFP_ZERO` 情况下的简化版本, `kzalloc(size, flags)` 就等于 `kmalloc(size, flags | __GFP_ZERO)`, 所以 `kzalloc` 函数会用 0 来填充分配出来的内存空间。

○ kfree 函数

`kfree` 函数用来释放 `kmalloc` 分配的内存,其原型为

```
void kfree(const void *objp)
```

如同讨论 `kmalloc` 函数那样, `kfree` 函数的实现代码可以简化为

```
void kfree(const void *objp)
{
    struct kmem_cache *c;

    struct page *page = virt_to_page(objp);
    c = (struct kmem_cache *)page->lru.next;

    __cache_free(c, (void *)objp);
}
```

函数首先根据要释放内存的指针 `objp` 调用 `virt_to_page` 函数来获得 `objp` 所在的页面对象指针 `*page` (如果 `objp` 所在的是由一组连续物理页组成的页块,那么 `virt_to_page` 返回页块的第一个物理页的对象指针;如果 `objp` 所在的是单个页面,那么就返回该页对象指针)。

`virt_to_page` 函数的原理是:先根据 `objp` 获得物理页帧号 `__pa(objp) >> PAGE_SHIFT`,接着取得该页帧号所对应的页对象指针 `*page = mem_map4 + (__pa(objp) >> PAGE_SHIFT)`。

⁴ `mem_map` 是系统中所有物理内存页对象所构成数组的首地址: `struct page *mem_map;`

页对象所在 slab 分配器的 `kmem_cache` 指针 `*c` 保留在 `page->lru.next`。

由上面通过 `objp` 来获得 `page` 的过程可以看出, `kfree` 释放的内存只能来自于 `kmalloc`, 后者实际上只使用 `ZONE_NORMAL` 和 `ZONE_DMA` 中的物理页。

函数最后调用 `__cache_free` 函数在 `c` 所对应的 slab 分配器中释放内存对象。

3.3.3 `kmem_cache_create` 与 `kmem_cache_alloc`

提供小内存分配并不是 slab 分配器的唯一用途, 在某些情况下, 有些内核模块可能需要频繁地分配和释放相同的内核对象。slab 分配器在这种情况下可以作为一种内核对象的缓存: 对象在 slab 中被分配, 当释放对象时, slab 分配器并不会将对象占用的空间返回给伙伴系统, 如此, 当再次分配该对象时, 可以从 slab 中直接得到对象的内存。另外, 由于 slab 分配器代码经过精心而严密的设计, 充分利用了 CPU 硬件的高速缓存, 可以想象, 在这些内核对象被频繁分配和释放的应用场景中, 利用 slab 分配器的这种优势, 可以大大提升系统的性能。在 Linux 自身的内核源码中, 就大量地使用了 `kmem_cache_create` 来创建内核对象的缓存。读者可以通过 `/proc/slabinfo` 查看当前系统中有多少活动的 `kmem_cache`。

相对于系统已经定义好的 `cache_cache` 与 `size_cache`, 这里讨论的内容实际上是内核模块开发者如何定制满足自己特定要求的 `kmem_cache`。生成 `kmem_cache` 的函数是 `kmem_cache_create`, 这个函数在前面讲述 `size_cache` 时已经提到过, 其原型如下:

```
struct kmem_cache *
kmem_cache_create(const char *name, size_t size, size_t align,
                  unsigned long flags, void (*ctor)(void *))
```

参数 `name` 是一指向字符型的指针, 用来表示生成的 `kmem_cache` 的名称, 该名称会导出到 `/proc/slabinfo` 文件中。需要注意的是, 创建出的 `kmem_cache` 对象会用一个指针指向该 `name`, 因此函数的调用者必须确保传入的 `name` 指针在 `kmem_cache` 的整个生存期内都有效, 否则可能会导致无效的引用。

参数 `size` 用来指定在缓存中分配对象的大小。

参数 `align` 用于指定数据对齐时的偏移量。内核代码的调用中这个参数几乎全为 0, 也即它的默认值。

参数 `flags` 是用于创建 `kmem_cache` 时的标志位掩码, 0 表示默认值。驱动程序中常用的标志位有:

`SLAB_HWCACHE_ALIGN`

该标志位要求 slab 分配器中的所有内存对象跟处理器的高速缓存行 (cache line) 对齐, 如果能将一些会被频繁访问的对象放入到高速缓存行中, 将会大幅提升内存访问性能。但

是对齐的要求会在对象与对象之间造成无用的填充，从而造成内存的浪费。

SLAB_CACHE_DMA

在 slab 分配器调用伙伴系统获得内存页时，告诉伙伴系统在 DMA_ZONE 区域获取内存页。该标志位在配置有 DMA_ZONE 的系统上，会设置 GFP_DMA。

SLAB_PANIC

在 kmem_cache 分配失败时将导致系统 panic。

最后一个参数 ctor 是个函数指针，称为 kmem_cache 的构造函数。如果函数的调用者提供了该函数，那么当 slab 分配器分配一块新的页面时，会对该页面中的每个内存对象调用此处设定的构造函数。所以此处的构造函数并不是在每次分配一个对象时都会被调用。

函数的核心是通过 cache_cache 来分配 kmem_cache 对象。函数如果成功执行，将返回指向 kmem_cache 的指针 *cachep，否则返回 NULL。新分配的 kmem_cache 对象最终会被加入到 cache_chain 所表示的链表中。

成功创建一个 kmem_cache 对象之后，就可以通过 kmem_cache_alloc 在 kmem_cache 中分配对象了。kmem_cache_alloc 的函数原型为

```
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
```

参数 cachep 就是 kmem_cache_create 函数返回的 kmem_cache 对象的指针。

参数 flags 是页面分配器中使用的掩码，前面在讨论页分配器时已经提到过。只有 kmem_cache_alloc 在内部需要与页分配器交互时，才会使用到这个参数。

这个函数的大体实现原理在 kmalloc 函数一节中已经讨论过，此处不再赘述。

○ kmem_cache_destroy 与 kmem_cache_free

kmem_cache_destroy 和 kmem_cache_free 是与 kmem_cache_create 和 kmem_cache_alloc 相对应的行为相反的函数。

kmem_cache_destroy 负责把 kmem_cache_create 创建的 kmem_cache 对象销毁，而 kmem_cache_free 则负责把 kmem_cache_alloc 分配的对象释放掉。

kmem_cache_destroy 函数原型为

```
void kmem_cache_destroy(struct kmem_cache *cachep)
```

参数 cachep 是要销毁的 kmem_cache 对象的指针。

函数首先从 cache_chain 链表中摘下要销毁的 kmem_cache 对象，在此之后调用

`__cache_shrink` 函数，以确保 `cachep` 中已经没有尚未被释放的内存对象。如有未释放的内存对象，函数将不会销毁 `kmem_cache` 对象，而会把已经从链表中摘下的 `kmem_cache` 对象重新加入到链表中，并在给出一段错误信息之后返回到调用者。`__cache_shrink` 函数的核心代码如下：

```
<mm/slab.c>
-----
static int __cache_shrink(struct kmem_cache *cachep)
{
    int ret = 0;
    struct kmem_list3 *l3
    ...
    ret += !list_empty(&l3->slabs_full) ||
           !list_empty(&l3->slabs_partial);
    ...
    return (ret ? 1 : 0);
}
```

如果 `cachep` 中所有的对象都已经被释放，函数最终通过 `kmem_cache_free(&cache_cache, cachep)` 从 `cache_cache` 中释放掉 `cachep` 所指向的 `kmem_cache` 对象。对于驱动程序所在的内核模块而言，如果代码中使用了 `kmem_cache_create` 创建的缓存来分配内存，那么模块的卸载函数将是调用 `kmem_cache_destroy` 的最佳场合。

`kmem_cache_free` 函数原型为

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp)
```

参数 `cachep` 是 `kmem_cache` 对象的指针。`objp` 是要释放的内存对象的指针。

函数会根据 per-CPU 缓存状态来尽量做最优化处理（内核总是挖空心思去干这种事情，还常常乐此不疲），但是我们可以想象 `free` 一个在 `kmem_cache` 中的内存对象，必然会引起一连串的连锁反应：如果要释放的对象恰恰是该 slab 上唯一被分配的对象，那么由于它的释放，将导致整个 slab 空闲，这种情况下内核有可能将该 slab 所对应的页面释放给伙伴系统，同时把该 slab 从 `kmem_cache` 对象的 `slabs_partial` 链表中摘除，更新对应的管理数据。

对这一连串事件的处理，内核委托给了 `free_block` 函数，本书不再讨论该函数。

相对于 `kmalloc` 函数，`kmem_cache_create` 相关函数提供给了开发者一个能更有效利用内存的方法。

3.4 内存池（mempool）

设备驱动程序对内存池的使用机会已经非常渺茫，在 Linux 2.6.39 版本的源码中，仅有屈

指可数的几个驱动模块还在使用内存池，读者大可跳过本节（不会对手头的工作有任何影响）。本书之所以还要在这里提一下内存池的概念，只是出于让读者增加点信息量的考虑，假使某天有面试官提出诸如茴香豆的“茴”字有几种写法之类的问题（实际的面试中这类问题居然不在少数），你不至于太沮丧。

内存池的总体思想是：预先为将来要使用的数据对象（比如 `a`）分配几个内存空间，把这些空间地址存放在内存池对象中。当代码真正需要为 `a` 分配空间时，正常调用前面几节提到的分配函数，如果分配失败，那么此时便可从内存池中取得预先分配好的 `a` 的地址空间。

所以内存池的概念实际上没有任何新鲜的东西，其分配函数的核心依然是前面介绍过的那些函数，其对实际内存分配失败时的补救措施也只限于预先分配的那些空间。

3.5 虚拟内存的管理

主流的 32 位处理器（比如 IA32、ARM 等）能寻址 2^{32} B 也即 4 GB 大小的地址空间，这部分空间称为虚拟地址空间。从虚拟地址到物理地址的转换通过处理器中的一个部件内存管理单元 MMU（Memory Management Unit）完成，为完成这种转变，系统软件比如操作系统必须建立适当的页表。

Linux 内核将 4 GB 的虚拟地址空间划分为两大块：顶部的 1 GB 空间给内核使用，称为内核空间；底部的 3 GB 给用户空间使用，称为用户空间⁵。内核代码中用 `PAGE_OFFSET` 宏来标示虚拟地址空间中内核部分的起始地址。本书只讨论跟驱动程序关系密切的 1 GB 的内核空间。

为了讲述下面的 `vmalloc` 相关内存分配函数，有必要先讨论一下内核是如何使用 1 GB 的内核空间的。

3.5.1 内核虚拟地址空间构成

内核会将 1 GB 的内核空间大体上分为三个部分：第一部分位于 1 GB 空间的开头，用于对系统物理内存的直接映射（本书也称之为线性映射），内核用全局变量 `high_memory` 来表示这段空间的上界；第二部分位于中间，主要用于 `vmalloc` 函数，本书称之为“VM 区”或者“`vmalloc` 区”；第三部分位于 1 GB 空间的结尾部分，用于特殊映射。整个 1 GB 空间的划分构成如图 3-5 所示：

⁵ 这种虚拟地址空间的划分通过内核的配置选项可以改变，不过这不是本书要讨论的内容。

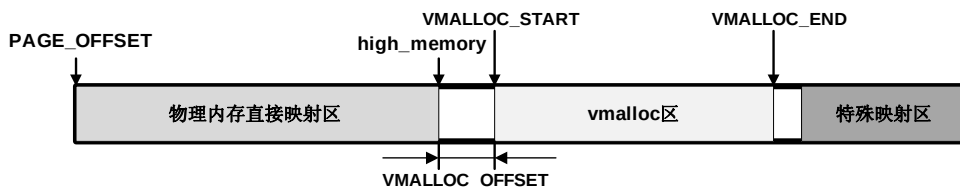


图 3-5 内核虚拟地址空间结构图

图中的白色区域为 1 GB 虚拟地址空间中的“空洞”，空洞部分不作任何地址映射，主要用做安全保护，防止不正确的越界内存访问（越界如果进入到空洞地带，因为此处没有进行任何形式的映射，对应的页表项将会使得处理器产生一个异常）。

3.5.2 vmalloc 与 vfree

`vmalloc` 函数也是内核模块会使用到的一个内存分配函数，它的特点是分配的虚拟地址空间是连续的，但是这段虚拟地址空间所映射的物理地址可能是不连续的。`vmalloc` 函数主要对图 3-5 中的 `vmalloc` 区进行操作，它返回的地址就来自于该区域。

在驱动程序中并不鼓励使用 `vmalloc` 函数，这主要是出于以下几个方面的考虑：首先，`vmalloc` 的实现机制决定了它的使用效率没有 `kmalloc` 这样的函数高；其次，在某些体系结构比如 x86 上，因为物理内存通常都比较大，这使得 `vmalloc` 区域相对变得很小，对 `vmalloc` 的调用失败的可能性增大。当然在嵌入式领域，内存通常都比较小，这个问题并不是很明显；最后，`vmalloc` 分配出的地址空间在物理上并不能保证是连续的，这对那些要求物理地址空间连续的设备比如 DMA 造成了麻烦。

然而，在某些情况下，如果获得连续物理内存的可能性不是很大，那么可以通过 `vmalloc` 来用不连续的物理内存组装出一块连续的内存区域（在虚拟地址空间）。在“内核模块”一章中看到的模块加载过程，就使用了 `vmalloc` 来为模块的 ELF 文件数据分配空间，这主要是因为模块可以随时被加载进系统，如果系统运行了很长的时间而且模块的 ELF 文件又比较大，就很有可能无法分配出连续的物理空间来容纳 ELF 文件中的数据，所以内核选择用 `vmalloc` 来为模块分配空间。下面简单讨论 `vmalloc` 函数的实现原理。

`vmalloc` 函数原型为

```
void *vmalloc(unsigned long size)
```

`vmalloc` 函数的实现原理可简单概括为三大步骤：

- (1) 在 `vmalloc` 区分配出一段连续的虚拟内存区域。
- (2) 通过伙伴系统获得物理页。
- (3) 通过对页表的操作将步骤 1 中分配的虚拟内存映射到步骤 2 中获得的物理页上。

在内核具体的代码实现上，步骤 1 利用红黑树来解决 `vmalloc` 区中动态虚拟内存块的分配和释放。对于 `vmalloc` 区中每一个分配出来的虚拟内存块，内核用 `struct vm_struct` 对象来表示。`struct vm_struct` 定义如下：

```
<include/linux/vmalloc.h>
-----
struct vm_struct {
    struct vm_struct *next;
    void *addr;
    unsigned long size;
    unsigned long flags;
    struct page **pages;
    unsigned int nr_pages;
    unsigned long phys_addr;
    void *caller;
};
```

其中，`next` 用来把 `vmalloc` 区中所有已分配的 `struct vm_struct` 对象构成链表，该链表的表头为一全局变量 `struct vm_struct *vmlist`。`addr` 为对应虚拟内存块的起始地址，应该是页对齐。`size` 为虚拟内存块的大小，总是页面大小的整数倍。`flags` 为表示当前虚拟内存块映射特性的标志，本章只介绍 `VM_ALLOC` 和 `VM_IOREMAP`，余下的标志推迟到“内存映射与 DMA”一章再讨论：`VM_ALLOC` 标志表示当前虚拟内存块是给 `vmalloc` 函数使用，映射的是实际物理内存（RAM）；`VM_IOREMAP` 标志表示当前虚拟内存块是给 `ioremap` 相关函数使用，映射的是 I/O 空间地址，也就是设备内存。`pages` 是被映射的物理内存页面所形成的数组首地址。`nr_pages` 表示映射的物理页的数量。`phys_addr` 多在 `ioremap` 函数中使用，表示映射的 I/O 空间起始地址，页对齐。

这一步骤中需要注意的是，内核总是会把 `vmalloc` 函数的参数 `size` 调整到页对齐，同时会在调整后的数值上再加一个页面的大小：

```
size = PAGE_ALIGN(size);
...
size += PAGE_SIZE;
```

内核之所以在把 `size` 对齐到页面大小之后再加上一个页面的大小，是为了防止可能出现的越界访问。因为在步骤 3 的页表操作中并不会向这个附加在末尾的虚拟地址上提交实际物理页面，所以当有访问进入到这个区间时，处理器将会产生异常。此处的原理同图 3-5 中的“空洞”完全一样。

步骤 2 中内核在调用伙伴系统获取物理内存页时，使用了 `GFP_KERNEL | __GFP_HIGHMEM` 标志，`GFP_KERNEL` 意味着 `vmalloc` 函数在执行过程中可能睡眠，因而不可在中断等非进程上下文中调用，`__GFP_HIGHMEM` 标志则告诉伙伴系统在 `ZONE_HIGHMEM` 区中查找空闲页，这是因为 `ZONE_NORMAL` 区中的物理内存资源非常

宝贵，主要留给 `kmalloc` 这类函数使用来获得连续的物理内存页面，因此对于 `vmalloc` 函数应该尽量使用高端的物理内存页。此外，内核在分配物理页时使用 `alloc_page` 或者是 `order=0` 情形下的 `alloc_pages_node` 函数，这意味着内核在此处是以每次只分配单个页面的形式来完成物理页的分配，这与 `vmalloc` 的设计初衷是完全吻合的：用来分配大块内存但无须保证在物理内存空间上的连续性。

步骤 3 没有特别需要注意的地方，唯一的一点是不对步骤 1 中内存区域的末尾 4 KB 大小部分作映射（步骤 2 中当然也不会为这段虚拟空间分配物理页）：

<mm/vmalloc.c>

```
int map_vm_area(struct vm_struct *area, pgprot_t prot, struct page ***pages)
{
    ...
    unsigned long end = addr + area->size - PAGE_SIZE; //去掉末尾的页面不映射
    ...
}
```

图 3-6 展示了用 `vmalloc` 函数分配内存的情形。图中在 `vmalloc` 区中分配出来的虚拟内存块通过内核页表的配置之后，被映射到了高端内存区中两个离散的物理页面 205 和 273，虚拟内存块的最后一个页面没有映射到实际的物理页上，旨在对可能出现的越界访问起保护作用。

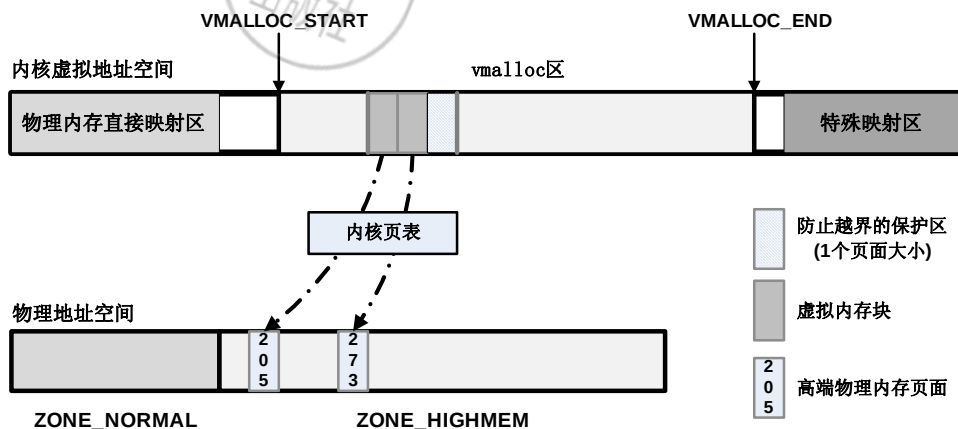


图 3-6 vmalloc 的页面映射

函数 `vfree` 用来释放 `vmalloc` 获得的虚拟地址块，它执行的是 `vmalloc` 的反操作：红黑树算法释放 `vmalloc` 生成的节点，清除内核页表中对应表项，调用伙伴系统一页一页地释放由 `vmalloc` 映射的物理页，`kfree` 掉管理数据所占用的内存。

`vfree` 函数原型为

```
void vfree(const void *addr)
```

3.5.3 ioremap

ioremap 函数（宏）是体系架构相关的，其函数原型基本上等同于：

```
void __iomem * ioremap(unsigned long phys_addr, size_t size)
```

此处的__iomem 的作用只是提醒调用者返回的是一 io 类型的地址，如同__user、__percpu 一样，某些工具软件有可能会利用这些定义符作一些诸如代码质量等方面的检查。

ioremap 函数及其变种用来将 vmalloc 区的某段虚拟内存块映射到 I/O 空间，其实现原理与 vmalloc 函数基本上完全一样，都是通过在 vmalloc 区分配虚拟地址块，然后修改内核页表的方式将其映射到设备的内存区，也就是设备的 I/O 地址空间。与 vmalloc 函数不同的是，ioremap 并不需要通过伙伴系统去分配物理页，因为 ioremap 要映射的目标地址是 I/O 空间，不是物理内存。

因为 I/O 空间在不同的体系架构上有不同的解释，比如 IA32 架构上有独立于内存访问指令之外的 I/O 指令，ARM 的架构上则没有，所以在函数返回地址的使用上，有些要注意的地方。假设返回地址是 pVaddr，对于有专门 I/O 指令的体系，比如 IA32，不能直接用内存访问的方式来使用该地址，*pVaddr = 0x1234 是错误的，应该使用 readw(pVaddr)，后者实际上使用了 inw 指令，这是 IA32 架构上专门的 I/O 指令；而在 ARM 处理器上，*pVaddr = 0x1234 则是完全正确的。因此，为了简化不同的架构平台代码移植工作，对于 ioremap 返回的地址，应该统一使用 readb/writeb、readw/writew 这样的宏，这些宏在不同的平台上会展开成架构相关的代码。

实际代码中 ioremap 还有一些相关的变体，包括 ioremap_nocache、ioremap_cached 等，这些变体的主要功能是通过加入一些映射标志位来影响相关内核页表项的设置，比如设备驱动程序中最常用的 ioremap_nocache，就是通过清除页表项中的 C(ache)标志⁶，使得处理器在访问这段地址时不会被 cache，这对外设空间的地址是非常重要的。

如果被映射的 I/O 空间不再使用，应该使用 iounmap 函数来做相关的清除工作，iounmap 函数要完成的工作包括将 vmalloc 区中分配的虚拟内存块返还给 vmalloc 区，清除对应的页表目录项等。

3.6 per-CPU 变量

本来 per-CPU 变量可以在“互斥与同步”一章中介绍，但鉴于其实现的核心部分在于对这

⁶ 不同处理器的页表/目录项有不同的配置形式，具体可参考特定处理器的开发手册。

些变量空间的分配和使用上（这其实是 Linux 内核中另一种内存分配的形式，源码中称之为 percpu memory allocator），因此，本章前面提供的上下文环境是最适合讨论 per-CPU 变量实现机制的地方，故而本节将用一定的篇幅讨论一些其内部的实现机制。当然，对 per-CPU 变量的使用上，也会有具体的案例给出，并在此基础上探讨 per-CPU 变量与互斥问题之间的关联。

per-CPU 变量是 Linux 内核中一个非常有趣的特性，它为系统中的每个处理器都分配了该变量的一个副本。这样做的好处是，在多处理器系统中，当处理器操作属于它的变量副本时，不需要考虑与其他处理器竞争的问题，同时该副本还可以充分利用处理器本地的硬件缓存以提高访问速度。然而读者不应该认为只要使用的是 per-CPU 变量，在并发访问方面就一定是安全的，本节结束的地方会有些这方面的思考。

基于 per-CPU 变量的以上特性，其最典型的应用场合是在统计计数方面（为此内核源码中专门提供了基于 per-CPU 的一个计数器实现，感兴趣的读者可参考 lib/percpu_counter.c）。例如在网络系统中，内核需要跟踪已接收到的各类数据包的数量，而这些数量在系统中更新的频率极快，每秒可能成千上万次。此时就可以使用 per-CPU 变量，让系统中每个处理器都使用独属于自己的该变量的副本，这样在变量更新时就无须考虑多处理器的锁定问题，可以提高性能。如果需要统计出系统接收数据包的总量，只要将各处理器副本中的值相加即可。

下面通过 Linux 实际代码来探究 per-CPU 变量的实现机制（基于 SMP 系统讨论）。per-CPU 变量按照存储变量的空间来源可以分为静态 per-CPU 变量和动态 per-CPU 变量：前者的存储空间是在代码编译时静态分配的；后者的存储空间则是在代码的执行期间动态分配的。先讨论静态 per-CPU 变量。总体上说，要使一个静态 per-CPU 变量能够工作，除了特别的 per-CPU 变量声明，还必须有链接脚本和相关内核源码的配合。

3.6.1 静态 per-CPU 变量的声明与定义

在 Linux 系统中声明一个 per-CPU 变量的方法是使用 DECLARE_PER_CPU 宏，相关定义如下：

```
<include/linux/percpu_defs.h>
-----
#define DECLARE_PER_CPU(type, name) \
    DECLARE_PER_CPU_SECTION(type, name, "")

#define DECLARE_PER_CPU_SECTION(type, name, sec) \
    extern __PCPU_ATTRS(sec) __typeof__(type) name

#define __PCPU_ATTRS(sec) \
    __percpu __attribute__((section(PER_CPU_BASE_SECTION sec))) \
    PER_CPU_ATTRIBUTES
```

```
<include/asm-generic/percpu.h>
-----
#define PER_CPU_BASE_SECTION ".data..percpu"
```

上面的定义看起来不是很直白，这里不妨用一个具体的例子来看看上面的宏到底做了什么，比如 `DECLARE_PER_CPU(int, dolphin);`。根据上面的定义，该宏在多处理器系统中展开之后如下：

```
extern __percpu __attribute__((section(".data..percpu"))) int dolphin;
```

可见该宏在源码中声明了一个变量 `int dolphin`，该变量放在一个名为 `".data..percpu"` 的 section 中。以上只是变量的声明部分，定义部分要用宏 `DEFINE_PER_CPU`：

```
<include/linux/percpu_defs.h>
-----
#define DEFINE_PER_CPU (type, name) \
    DEFINE_PER_CPU_SECTION (type, name, "")

#define DEFINE_PER_CPU_SECTION(type, name, sec) \
    __PCPU_ATTRS(sec) PER_CPU_DEF_ATTRIBUTES \
    __typeof__(type) name
```

相比于 `DECLARE_PER_CPU`，`DEFINE_PER_CPU` 只是去掉了变量声明前的 `extern`，所以 `DEFINE_PER_CPU(int, dolphin)` 将会在源码中定义一个变量：

```
__percpu __attribute__((section(".data..percpu"))) int dolphin;
```

看了以上 per-CPU 变量的声明和定义，似乎除了把变量放到 `".data..percpu"` section 里，和其他普通变量的声明与定义相比也没有什么特别之处。但既然把 per-CPU 变量放到了 `".data..percpu"` section，还是看看都有哪些地方用到了这个 section。

3.6.2 静态 per-CPU 变量的链接脚本

考察内核的链接脚本，对于 per-CPU 变量，发现有如下相关定义：

```
<kernel/vmlinux.lds>
-----
. = ALIGN((1 << 12));
.data..percpu : AT(ADDR(.data..percpu) - 0xC0000000)
{
    __per_cpu_load = .;
    __per_cpu_start = .;
    *(.data..percpu..first)
    *(.data..percpu..page_aligned)
    *(.data..percpu)
    *(.data..percpu..shared_aligned)
    __per_cpu_end = .;
}
. = ALIGN((1 << 12));
```

可见，内核在编译链接时会把所有静态定义的 per-CPU 变量统一放到".data..percpu" section 中，链接器生成__per_cpu_start 和__per_cpu_end 两个变量来表示该 section 的起始和结束地址。紧接着为了配合链接器的行为，Linux 内核源码中针对以上的链接脚本声明了如下的外部变量：

```
<include/asm-generic/sections.h>
```

```
extern char __per_cpu_load[], __per_cpu_start[], __per_cpu_end[];
```

现在，似乎已万事具备，就看内核如何为系统中的每个 CPU 产生一份变量的副本了。

3.6.3 setup_per_cpu_areas 函数

前面提到用 DEFINE_PER_CPU 定义的变量，系统中的每个 CPU 都拥有该变量的一个副本。但到目前为止，我们看到的 int dolphin 变量也只在 ".data..percpu" section 中才有一份，内核如何让系统中每个 CPU 都拥有该变量的一个副本呢？

答案在于系统初始化期间调用的 setup_per_cpu_areas 函数⁷，这个函数不但会完成变量副本的生成，而且会对 per-CPU 变量的动态分配机制进行初始化。

○ 静态 per-CPU 变量副本的产生

图 3-7 大体上勾画了这一过程，以下讨论的内容都可参考该图：

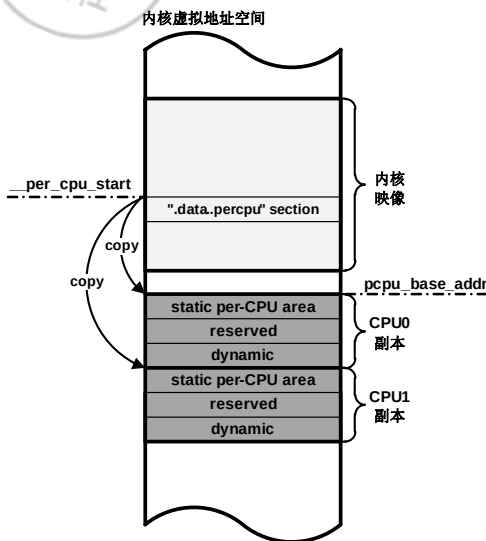


图 3-7 静态 per-CPU 变量副本的产生

⁷ 严格地说，此处是指 setup_per_cpu_areas 函数的整个调用链，setup_per_cpu_areas 函数本身只是这个调用链的发起者。

为方便叙述，笔者只简单说是 setup_per_cpu_areas 函数。

```
<mm/percpu.c>
```

```
void __init setup_per_cpu_areas(void)
```

setup_per_cpu_areas 函数首先计算出 ".data..percpu" section 的空间大小 (static_size = __per_cpu_end - __per_cpu_start)，此处利用的正是上节链接脚本中的内容。static_size 是内核源码中所有用 DEFINE_PER_CPU 及其变体所定义出的静态 per-CPU 变量所占空间的大小。此外内核还为模块使用的 per-CPU 变量以及动态分配的 per-CPU 变量预留了空间（对应图 3-7 中的 reserved 和 dynamic 部分），大小分别记为 reserved_size 和 dyn_size。

然后 setup_per_cpu_areas 函数调用 alloc_bootmem_nopanic 来分配一段内存，用来保存 per-CPU 变量副本。此时因为系统的内存管理系统还没有建立起来，所以使用的是 Linux 引导期内存分配器。这块内存的大小要依赖于系统中 CPU 的数量，因为要为每个 CPU 创建变量的副本。内核代码称每个 CPU 变量副本所在内存空间为一个 unit，所以代码中的 nr_units 变量实际上表示了系统中 CPU 的数量，每个 unit 的大小记为 unit_size，unit_size = PFN_ALIGN(static_size + reserved_size + dyn_size)。如此，变量副本所在空间的大小就是 nr_units * unit_size。指针变量 pcpu_base_addr 指向副本空间的起始地址。

容纳副本的空间有了，还要做一件事，把内核映像 ".data..percpu" section 中的变量数据复制到 pcpu_base_addr 空间，细节如下面的代码段所示：

```
for (i = 0; i < nr_units; i++, pcpu_base_addr += unit_size) {
    memcpy(pcpu_base_addr, __per_cpu_load, static_size);
}
```

很好很强大，再次用到了链接脚本中的变量 __per_cpu_load，这段代码的功能在图 3-7 中为两个带 copy 标志的箭头线条所示。

至此，系统针对 DEFINE_PER_CPU 定义的变量已经为每个 CPU 产生了一个副本，接下来的问题是如何使用这些变量的副本。不过在进入这一话题之前，先稍微扩展这里讨论的内容范围，探讨 per-CPU 变量的动态分配机制。

○ 动态 per-CPU 变量副本的产生

相对于静态 per-CPU 变量的定义，动态分配一个 per-CPU 变量可以用下面两个函数：

- (1) alloc_percpu
- (2) void __percpu * __alloc_percpu(size_t size, size_t align)

前者是一个宏，定义为

```
<include/linux/percpu.h>
```

```
#define alloc_percpu(type) \
    (typeof(type) __percpu *) __alloc_percpu(sizeof(type), __alignof__(type))
```

`alloc_percpu` 底层还是调用了 `__alloc_percpu`，使用了默认的对齐参数 `__alignof__(type)`。如果有对对齐方式的特别要求，那么应该直接使用 `__alloc_percpu` 函数。

对应的释放函数为

```
<mm/percpu.c>
-----
void free_percpu(void __percpu *ptr)
```

关于动态 per-CPU 变量的分配机制，内核中的相关代码比较烦琐，但是核心思想同静态 per-CPU 是一样的，大体可分为两部分：第一部分，为系统中的每个 CPU 分配副本的空间；第二部分，通过某种机制实现对 CPU 特定的副本空间的访问。

这里的“某种机制”很快就会在下一节中谈到。关于第一部分，前面已经看到静态 per-CPU 变量是如何达成这一目标的，因为静态定义的变量所在空间大小是预先确定的，所以内核能很轻易完成副本空间的分配和变量数据的复制。但是对于动态分配的 per-CPU 变量，则没有这么幸运，变量可以随时被申请，也可以随时被释放⁸。为此，内核使用一种基于 chunk 的手法来实现，chunk 作为一种存放管理数据的容器而存在，根据其上空闲空间的大小而在一个 `pcpu_slot` 数组所表示的链表中进行迁移，数组的索引 `i` 指明了其链表中 chunk 空闲空间的大小，当需要动态分配一个 per-CPU 变量时，内核在 `pcpu_slot` 数组中查找有无 chunk 的空闲空间满足需要，如果有，就在此 chunk 的空闲空间中为系统中的每个 CPU 生成变量的副本空间（`pcpu_populate_chunk`），如果没有，就重新创建一个新的 chunk，对于新分配的 chunk，会在内核虚拟地址空间的 `vmalloc` 区为它分配副本空间（这是一个虚拟地址连续的，包含了系统中所有 CPU 的副本的存储空间），空间的起始地址保存在 chunk 的 `base_addr` 成员中。chunk 用一个整型数组 `map` 来跟踪副本空间的分配情况，当要分配一个动态 per-CPU 变量时，就在副本空间查找空闲区域，找到之后为每个 CPU 都分配出存储该 per-CPU 变量的存储小块。注意此时存储 per-CPU 变量的空间还是在 `vmalloc` 区，如果之前该存储小块上还没有映射物理页面的话，需要为新分配变量映射新的页面，物理页通过页分配器从伙伴系统获得，chunk 通过成员变量 `populated` 来跟踪物理页面的提交。读者仔细看看源码就会发现，chunk 其实是整个 percpu memory allocator 的基础，即使是静态 per-CPU 变量，最终也都被统一到了 chunk 的体系当中（从前面静态 per-CPU 变量实现机制的讨论，静态 per-CPU 变量其实并不需要 chunk），内核创建的第一个 chunk 就是用来管理内核中定义的静态 per-CPU 变量（`pcpu_setup_first_chunk`），这个 chunk 中的 `base_addr` 就是图 3-7 中的 `pcpu_base_addr`，不过这第一个 chunk 有点特殊，它的副本空间是通过 Linux 引导期内存分配器获得的。

总之，执行期生成的 chunk 在 `vmalloc` 区分配副本空间，通过 `map` 成员跟踪空间分配信息，通过 `populated` 成员跟踪物理页面的提交信息。对此过程的详细描述读者可参考

⁸ 这和 `vmalloc` 函数要完成的功能非常相似，`vmalloc` 也要对 `malloc` 区域中的分配与释放进行管理。

www.embexperts.com。

概括下来, `alloc_percpu` 返回的是 `chunk->base_addr + offset - delta` (此处的 `offset` 是刚分配的 per-CPU 变量在 CPU0 所属的副本空间的偏移量, `delta` 将在下面的“使用 per-CPU 变量”一节中讲述), 在访问该变量 CPU 特定的副本时, 需要在该地址上使用“某种机制”获得 CPU 特定的变量所在地址, 然后才可以进行访问。

至此, 所有关于静态和动态 per-CPU 变量副本空间的问题都已经解决, 处理器拥有了自己独立的变量空间, 该是使用它们的时候了。访问 per-CPU 变量的要点是根据不同的处理器获得对应的变量副本。

3.6.4 使用 per-CPU 变量

已经有了 per-CPU 变量副本所在空间的首地址, 现在我们需要“某种机制”来访问它。在内核源码中, “某种机制”的现实表现形式是内核定义的一组宏, 为了正确访问 per-CPU 变量, 应该在代码中使用这些宏。为了了解访问 per-CPU 变量的“某种机制”, 下面以 `get_cpu_var` 宏⁹为例来说明对静态 per-CPU 变量的访问。

```
<include/asm-generic/percpu.h>
-----
#define my_cpu_offset __per_cpu_offset [raw_smp_processor_id()]

<include/linux/compiler.h>
-----
# define RELOC_HIDE(ptr, off)
    ({ unsigned long __ptr;
        __ptr = (unsigned long) (ptr);
        (typeof(ptr)) (__ptr + (off)); })

<include/asm-generic/percpu.h>
-----
#define SHIFT_PERCPU_PTR(__p, __offset) ({
    __verify_pcpu_ptr((__p));
    RELOC_HIDE((typeof(*(__p)) __kernel __force *)(__p), (__offset)); \
})

<include/asm-generic/percpu.h>
-----
#define __get_cpu_var(var) \
    (*SHIFT_PERCPU_PTR(&(var), my_cpu_offset))

<include/linux/percpu.h>
-----
#define get_cpu_var(var) ({
    preempt_disable();
    &__get_cpu_var(var); })
```

⁹ 在 `get_cpu_var` 宏和 `put_cpu_var` 宏的定义中利用了 `&` 运算符对左值类型的变量检查的小技巧, 这使得代码初看起来有点晦涩。

所以对于本节开始的例子 `int val = get_cpu_var(dolphin)`，其实是展开成如下等价的代码：

```
int *p;
preempt_disable();
p = (int *)&dolphin + __per_cpu_offset[raw_smp_processor_id()];
val = *p;
```

代码中的 `__per_cpu_offset` 是用来实现处理器副本访问的基础，每个处理器副本所在空间的偏移地址都由 `__per_cpu_offset` 引出，这是个全局性的数组变量：

```
<mm/percpu.c>
-----
unsigned long __per_cpu_offset[NR_CPUS] __read_mostly;
```

它的初始化出现在 `setup_per_cpu_areas` 函数中，内核在启动阶段调用这个函数来初始化 per-CPU 变量机制：

```
<mm/percpu.c>
-----
void __init setup_per_cpu_areas(void)
{
    unsigned long delta;
    unsigned int cpu;
    ...
    delta = (unsigned long)pcpu_base_addr - (unsigned long)__per_cpu_start;
    for_each_possible_cpu(cpu)
        __per_cpu_offset[cpu] = delta + pcpu_unit_offsets[cpu];
}
```

函数首先计算出副本空间首地址（`pcpu_base_addr`）与 `".data..percpu"section` 首地址（`__per_cpu_start`）之间的偏移量 `delta`。代码中的 `pcpu_unit_offsets` 是个数组，`pcpu_unit_offsets[cpu]` 保存对应 `cpu` 所在副本空间相对于 `pcpu_base_addr` 的偏移量，这样就得到 per-CPU 变量副本的偏移值，放在 `__per_cpu_offset` 数组中。

如此，对于 CPU0 中的变量 `var`，它的地址应为 `&val + __per_cpu_offset[0]`；对于 CPU1 而言，变量 `var` 的地址则为 `&val + __per_cpu_offset[1]`。

而这正是 `get_cpu_var` 宏所完成的功能。

其中 `preempt_disable()` 用来关闭内核可抢占性，这是因为对于可抢占的内核而言，即使是在单处理器上，依然会有竞争的情况出现。关闭内核可抢占性可确保在对 per-CPU 变量操作的临界区中，当前进程不会被换出处理器。由于这个因素的存在，需要一个和 `get_cpu_var` 配对使用的宏 `put_cpu_var`，用来恢复内核调度器的可抢占性：

```
<include/linux/percpu.h>
-----
#define put_cpu_var(var) do {                \
    (void)&(var);                             \
}
```

```
preempt_enable();          \
} while (0)
```

如果需要读取其他处理器中的副本，可以使用 `per_cpu(var, cpu)`：

```
<include/asm-generic/percpu.h>
-----
#define per_cpu(var, cpu) \
    (*SHIFT_PERCPU_PTR(&(var), per_cpu_offset(cpu)))
```

为了使读者对 per-CPU 变量的使用有个直观的印象，这里给出一个具体例子：

```
//定义一个静态 per-CPU 变量 my_birthday，变量类型为 struct birth_day
struct birth_day {
    int day;
    int month;
    int year;
};

static DEFINE_PER_CPU(struct birth_day, my_birthday) = {12, 12, 1860};

//通过 get_cpu_var 访问该变量
get_cpu_var(my_birthday). year ++;
put_cpu_var(my_birthday);
```

上面的例子中首先定义了一个数据结构 `struct birth_day` 作为 per-CPU 变量的类型，然后用 `DEFINE_PER_CPU` 定义了一个 per-CPU 变量 `my_birthday`，最后用 `get_cpu_var` 来使用该变量。在多处理器系统中，当 CPU 0 执行到 `get_cpu_var` 时，因为宏展开后的 `raw_smp_processor_id()`，所以它将获得属于它的 `my_birthday` 变量的副本。

对于动态分配的 per-CPU 变量，在前面的“动态 per-CPU 变量副本的产生”部分中已经介绍了内核如何为动态分配的 per-CPU 变量分配存储空间。下面看看系统是如何访问动态 per-CPU 变量的。访问动态 per-CPU 变量通过宏 `per_cpu_ptr`：

```
<include/linux/percpu.h>
-----
#define per_cpu_ptr(ptr, cpu) SHIFT_PERCPU_PTR((ptr), per_cpu_offset((cpu)))
```

前面已经提到过 `SHIFT_PERCPU_PTR`，和静态 per-CPU 变量一样，用来产生 CPU 特定的变量存储空间的地址。到此，为了完整动态 per-CPU 变量的实现机制，我们把前面的内容与这里的讨论串联一下，首先我们通过 `alloc_percpu` 获得了变量的一个地址，该地址的值 `ptr = chunk->base_addr + offset - delta`，此处 `offset` 是该变量在 CPU0 副本空间的偏移量，然后假设 CPU1 要访问属于它自己的变量，那么它应该使用 `per_cpu_ptr(ptr, 1)`，这相当于：`chunk->base_addr + offset - delta + delta + percpu_unit_offsets[1]`，由于 `percpu_unit_offsets[1] = unit_size`，也就是每个 CPU 副本空间的大小，所以 `per_cpu_ptr(ptr, 1) = chunk->base_addr + offset + unit_size`，这样就得到了 CPU1 中该变量的实际虚拟地址（在 `vmalloc` 区）。

理论上的原理大致如此，下面再结合一个实际的案例具体化本节的讨论。

以下为 Linux 2.6.35 版本内核树中 drivers/dma/dmaengine.c 使用的动态分配 per-CPU 变量的具体代码：

```
struct dma_chan_tbl_ent {
    struct dma_chan *chan;
};
static struct dma_chan_tbl_ent __percpu *channel_table[DMA_TX_TYPE_END];

//动态分配一类型为 struct dma_chan_tbl_ent 的 per-CPU 变量，channel_table[cap]为指向变
//量所在空间的地址
channel_table[cap] = alloc_percpu(struct dma_chan_tbl_ent);

//通过 per_cpu_ptr 得到特定 cpu 上的变量指针
for_each_possible_cpu(cpu)
    per_cpu_ptr(channel_table[cap], cpu)->chan = NULL;
...
//程序不再需要使用该变量，释放其所在空间
free_percpu(channel_table[cap]);
```

如果考虑到内核的抢占性可能造成的问题，那么在使用 per_cpu_ptr 的时候需要用 get_cpu 和 put_cpu 来关闭和开启内核的可抢占性，如下面的示例代码所示：

```
int cpu = get_cpu();
struct dma_chan_tbl_ent *ptr = per_cpu_ptr(channel_table[cap], cpu);
//开始使用 ptr
L0:
...
L1:
//结束 ptr 的使用
put_cpu();
```

get_cpu 与 put_cpu 定义如下：

```
<include/linux/smp.h>
-----
#define get_cpu()    ({ preempt_disable(); smp_processor_id(); })
#define put_cpu()    preempt_enable()
```

这里关于可抢占性的问题在于，假设内核启动了调度器的可抢占特性，如果在 L0 与 L1 之间发生中断的话，当前进程可能被切换出处理器 CPU0，那么等到下次该进程被调度执行时，调度器在极端情况下可能把该进程提交到另一个处理器 CPU1 上运行，CPU1 拥有一个指向 CPU0 本地变量副本的指针，此时当初 per-CPU 变量被设计出来的初衷就被破坏了。更深层地探讨本例中出现的问题，貌似是把 ptr 的获得和使用分散开造成的，然而，若不能保证对 per_cpu_ptr 使用的原子性，这个问题总是存在的（虽然其出现的概率异常渺茫）。

所以为安全起见，`per_cpu_ptr` 结合 `get_cpu` 和 `put_cpu` 的配对使用，总是没错的。

3.7 本章小结

本章介绍了设备驱动程序中常用的内存分配函数及其实现机制。

首先是基于伙伴系统的页面分配器，这是内核中整个内存分配模块的核心，用于分配单个或者是连续的物理内存页。页面分配器提供的接口函数总体上可以分为两类。

一类是以 `alloc_pages(gfp_mask, gfporder)` 函数领衔。该类函数可以工作在三个内存域：`ZONE_HIGHMEM`、`ZONE_NORMAL` 和 `ZONE_DMA`，具体在哪个区域分配物理页由 `gfp_mask` 参数指定，如果所指定的区域没有足够的空闲页面满足要求，函数会自动到下一级区域重新分配，但是不会进入上级区域。比如在调用 `alloc_pages` 函数时指定 `gfp_mask` 为 `__GFP_HIGHMEM` 的话，函数将首先到高端内存域中试图分配物理页面，如果该区域没有足够空闲物理页面满足分配要求，则函数自动到 `ZONE_NORMAL` 中查找连续空闲页，如果该区也没有足够物理页面，则函数将下行到 `DMA` 区继续进行分配。但是如果指定 `gfp_mask` 为 `__GFP_DMA`，函数将到 `DMA` 区中分配物理页，即使 `DMA` 域中没有足够内存，函数也不会上溯到 `ZONE_NORMAL` 区，更不会到 `ZONE_HIGHMEM` 区中。`NORMAL` 和 `DMA` 这两个区域常统称为低端内存区域，它们在内核的虚拟地址空间中是被一一线性映射的，这意味着物理内存地址和虚拟地址之间只存在一个常量差值（`PAGE_OFFSET`），所以页面分配器在内部的实现上无须去更改这段映射区域所在的内核页目录表项（对常规内存的映射在系统初始化期间就完成了，在系统的整个运行期，这部分空间对应的页表项不会改变）。所以如果 `alloc_pages` 返回的页面位于低端物理内存区，那么通过 `page_address` 函数可以将返回的 `struct page` 对象指针转变成内核线性地址；如果 `alloc_pages` 返回的页面位于高端物理内存区，由于这部分页面尚未被映射，调用者就需要用 `kmap` 等函数对返回的 `struct page` 对象建立映射关系，如果 `kmap` 成功完成映射关系的建立，就将返回一个内核虚拟地址，该地址位于内核虚拟地址空间中的动态映射区。`gfp_mask` 没有为 `ZONE_NORMAL` 区定义专门的分配掩码，比如 `__GFP_NORMAL`，但是 `alloc_pages` 系列函数内建的默认行为是：如果没有明确指定 `__GFP_HIGHMEM` 或者是 `__GFP_DMA`，那么就视同分配时指定的分配掩码为 “`__GFP_NORMAL`”。设备驱动程序等内核模块最常用的分配掩码是 `GFP_KERNEL` 和 `GFP_ATOMIC`，由于二者均未明确指定 `__GFP_HIGHMEM` 或者是 `__GFP_DMA`，所以它们都会在低端内存区中分配物理页面。

另一组页面分配器以 `__get_free_pages(gfp_t gfp_mask, unsigned int order)` 函数领衔，它们最终也是通过 `alloc_pages` 来分配物理页面，所不同的是，`__get_free_pages` 函数只能在低端内存区域中分配物理页，函数返回的是内核线性地址，而不是 `struct page` 对象指针。

其次是基于 slab 分配器的 `kmalloc` 函数和 `kmem_cache_alloc` 函数。slab 分配器是工作在页

分配器基础之上的，但是它只能在低端物理内存区中分配页面。`kmalloc` 函数在内部的实现上基于 `size_cache`，这是一组 `kmem_cache` 所构成的缓存，每个缓存对应特定大小的内存对象。`kmem_cache_alloc` 函数实际上是对 `size_cache` 的一种扩展，它可以对 `kmem_cache` 中的内存对象大小进行定制，所以当代码需要频繁地分配和释放同一类型的数据结构对象时，使用 `kmem_cache_alloc` 函数可以更有效地使用系统的内存资源。释放它们所分配的空间使用 `kfree` 和 `kmem_cache_free` 函数。

接下来是 `vmalloc` 函数，这个函数工作在内核虚拟空间的 `VMALLOC_START` 和 `VMALLOC_END` 所表示的 `vmalloc` 区。该虚拟空间对物理内存的映射不是一一对应的，因此 `vmalloc` 函数分配的内存空间的特点是，在虚拟地址空间是连续的，但是所映射的物理地址不一定是连续的。它所映射的物理内存指定优先从高端内存区（`__GFP_HIGHMEM`）中分配。所以 `vmalloc` 的使用场景是，当需要分配一段大内存时，如果此时系统中能满足要求的连续物理页面不一定存在的话（此时用 `kmalloc` 函数就可能失败），可以使用 `vmalloc` 在内核虚拟地址空间 `vmalloc` 区分配一段连续的虚拟地址空间，然后通过映射到分散的物理内存页面来满足内存分配的需求。所以如果设备要求分配出的空间在物理上是连续的话，就不能使用 `vmalloc` 函数。另外，`vmalloc` 函数不能保证原子性，因此不能用在非进程上下文中，比如中断处理例程。而 `kmalloc` 函数可以通过 `GFP_ATOMIC` 标志来达成原子性。`vmalloc` 在分配过程中因为涉及对页表项的操作，这种操作常常要重建 TLB，会导致对 `vmalloc` 返回的地址进行访问时带来较大的系统开销，所以在使用效率上不及 `kmalloc` 函数，设备驱动程序应优先考虑 `kmalloc` 等函数。

最后一个内存分配的函数主要是用在多处理器系统中，即 per-CPU 内存分配器，虽然设备驱动程序中使用 per-CPU 变量的机会极少，但毕竟也是比较重要的内存分配机制，而且在本书后续的一些驱动程序的内核设施上也会看到它的身影。它的核心思想是，通过为系统中每个处理器都分配一个 CPU 特定的变量副本，来减少多处理器并发访问时的锁定操作，借此达到提高系统性能的目的。

至于系统启动阶段的 `bootmem` 内存分配机制，因为只存在于系统启动阶段内核内存管理模块框架建立起来之前使用，所以设备驱动程序使用这种内存分配机制的机会非常少。

第 4 章

互斥与同步

本章讨论 Linux 内核为设备驱动程序等内核模块提供的互斥与同步的内核设施。如果运行的系统中自始至终只有一个执行路径，那么无须考虑互斥与同步的问题，然而不幸的是，现代的 Linux 系统不只支持多进程而且支持多处理器，在这样的环境下，当多个执行路径并发执行时确保对共享资源的访问安全是驱动程序员不得不面对的问题。概括地说，互斥是指对资源的排他性访问，而同步则要对进程执行的先后顺序做出妥善的安排。

因为程序的并发执行而导致的竞态是 Linux 内核中一个非常复杂的方面。对于设备的驱动程序开发者而言，熟悉 Linux 内核提供的并发互斥的处理机制相当重要。所谓竞态，简而言之，就是多个执行路径有可能对同一资源进行操作时可能导致的资源数据紊乱的行为。我们把对共享的资源进行访问的代码片段称为临界区（critical section），而把导致出现多个执行路径的因素称为并发源。

本章将首先考察 Linux 系统中并发执行的来源，然后逐一讨论内核为保证对资源的互斥访问所提供的内核设施的幕后机制以及各自的应用场景。最后讨论 Linux 内核为保证各个执行路径之间的先后顺序所提供的同步机制。大部分的篇幅都将用来讨论与互斥相关的东西，因为对并发的互斥管理是非常令人头疼的。在面对实际的项目时，很可能不确定哪些地方需要用到内核的互斥机制，或者即使能够想到要对资源进行保护，也有可能在面对内核提供的众多互斥机制时无法做出正确抉择。更让人沮丧的是，互斥导致的问题常常极其隐蔽，绝大部分的时间里程序都运转良好，然而有时候却出现莫名的崩溃，这种崩溃因为难以复现，会给后期的调试工作带来很大的困难。驱动程序开发者对内核提供的互斥机制的切实理解是写出高安全性代码的关键。鉴于这些内核设施的代码严重依赖特定的处理器体系架构，所以在对具体的代码进行分析时，主要以 ARM 平台为主。代码分析的意义，旨在让读者对抽象的概念有具体的印象。

4.1 并发的来源

当我们说并发时，是指可能导致对共享资源的访问出现竞争状态的若干执行路径，不一定是严格的时间意义上的并发执行。Linux 系统下并发的来源主要有：

○ 中断处理路径

当系统正在执行当前进程时，发生了中断，中断处理函数和被中断的进程之间形成的并发，在单处理器中，虽然中断处理函数的执行路径与被中断的进程之间不是真正严格意义上的并发，然而中断处理函数和被中断进程之间却可能形成竞态。软中断的执行也可归结到这种类型的并发中。

○ 调度器的可抢占性

在单处理器上，因为调度器的可抢占特性，导致的进程与进程之间的并发。这种行为非常类似多处理器系统上进程间的并发。

○ 多处理器的并发执行

多处理器系统上进程与进程之间是严格意义上的并发，每个处理器都可独自调度运行一个进程，在同一时刻有多个进程在同时运行。

4.2 local_irq_enable 与 local_irq_disable

在单处理器不可抢占系统中，使用 `local_irq_enable` 与 `local_irq_disable` 是消除异步并发源的有效方式，虽然驱动程序中应该避免使用这两个宏（理由将在本章稍后的内容中给出），但是在 `spinlock` 等互斥机制中常常用到这两个宏，所以在此用一节的篇幅来对它们进行介绍。`local_irq_enable` 宏用来打开本地处理器的中断，而 `local_irq_disable` 则正好相反，用来关闭处理器的中断。这两个宏的定义如下：

```
<include/linux/irqflags.h>
-----
#define local_irq_enable() \
    do { trace_hardirqs_on(); raw_local_irq_enable(); } while (0)
#define local_irq_disable() \
    do { raw_local_irq_disable(); trace_hardirqs_off(); } while (0)
```

其中 `trace_hardirqs_on()` 和 `trace_hardirqs_off()` 用做调试，这里重点关注 `raw_local_irq_enable()` 和 `raw_local_irq_disable()`，这两个宏的具体实现都依赖于处理器体系架构，不同处理器有不同的指令来启用或者关闭处理器响应外部中断的能力，比如在 x86 平台上，会最终利用 `sti` 和 `cli` 指令来分别设置和清除 x86 处理器中的 `FLAGS`¹ 寄存器的 `IF` 标志，这样处理器就可以响应或者不响应外部的中断。ARM 平台则使用 `CPSIE` 指令。

在单处理器不可抢占系统中，如果某段代码要访问某共享资源，那么在进入临界区前使用

¹ 虽然不同的处理器上 `FLAGS` 寄存器的名称各有不同，但是所实现的功能大体类似。因此本书统一称其为 `FLAGS` 寄存器。

`local_irq_disable` 来关闭中断，这样在临界区中可保证系统不会出现异步并发源，访问完共享数据在出临界区时，再调用 `local_irq_enable` 来启用中断。

`local_irq_enable` 与 `local_irq_disable` 还有一种变体，是 `local_irq_save` 与 `local_irq_restore` 宏，定义如下：

```
<include/linux/irqflags.h>
-----
#define local_irq_save(flags) \
do { \
    typecheck(unsigned long, flags); \
    raw_local_irq_save(flags); \
    trace_hardirqs_off(); \
} while (0)

#define local_irq_restore(flags) \
do { \
    typecheck(unsigned long, flags); \
    if (raw_irqs_disabled_flags(flags)) { \
        raw_local_irq_restore(flags); \
        trace_hardirqs_off(); \
    } else { \
        trace_hardirqs_on(); \
        raw_local_irq_restore(flags); \
    } \
} while (0)
```

这两个宏相对于 `local_irq_enable` 与 `local_irq_disable` 最大的不同在于，`local_irq_save` 会在关闭中断前，将处理器当前的标志位保存在一个 `unsigned long flags` 中，在调用 `local_irq_restore` 的时候，再将保存的 `flags` 恢复到处理器的 `FLAGS` 寄存器中。这样做的目的是，防止在一个中断关闭的环境中因为调用 `local_irq_disable` 与 `local_irq_enable` 将之前的中断响应状态破坏掉。

在单处理器不可抢占系统中，使用 `local_irq_enable` 与 `local_irq_disable` 及其变体来对共享数据保护是种简单而有效的方法。但在使用时应该注意，因为 `local_irq_enable` 与 `local_irq_disable` 是通过关中断的方式进行互斥保护，所以必须确保处于两者之间的代码执行时间不能太长，否则将影响到系统的性能。

4.3 自旋锁

设计自旋锁的最初目的是在多处理器系统中提供对共享数据的保护，其背后的核心思想是：设置一个在多处理器之间共享的全局变量锁 `V`，并定义当 `V=1` 时为上锁状态，`V=0` 为解锁

状态。如果处理器 A 上的代码要进入临界区，它要先读取 V 的值，判断其是否为 0，如果 $V \neq 0$ 表明有其他处理器上的代码正在对共享数据进行访问，此时处理器 A 进入忙等待即自旋状态，如果 $V=0$ 表明当前没有其他处理器上的代码进入临界区，此时处理器 A 可以访问该资源，它先把 V 置 1（自旋锁的上锁状态），然后进入临界区，访问完毕离开临界区时将 V 置 0（自旋锁的解锁状态）。

上述自旋锁的设计思想在用具体代码实现时的关键之处在于，必须确保处理器 A “读取 V，判断 V 的值与更新 V” 这一操作序列是个原子操作（atomic operation）。所谓原子操作，简单地说就是执行这个操作的指令序列在处理器上执行时等同于单条指令，也即该指令序列在执行时是不可分割的²。

4.3.1 spin_lock

不同的处理器上有不同的指令用以实现上述的原子操作，所以 spin_lock 的相关代码在不同体系架构上有不同的实现，为了帮助读者对 spin_lock 这一机制建立具体的印象，下面以 ARM 处理器上的实现为例，仔细考察 spin_lock 的幕后行为。下面的讨论先以多处理器为主，然后再讨论 spin_lock 及其变体在单处理器上的演进。

在给出实际源码细节之前，先做个简短的说明，为了让读者更清楚地理解这里的代码，下面会对代码进行轻微调整，使之外在的表现形式更加紧凑而又不影响其内涵，同时也不会关注一些调试相关的数据成员，所以在摘录的代码中已将其移除。

下面是 Linux 源码中提供给设备驱动程序等内核模块使用的 spin_lock 接口函数的定义：

```
<include/linux/spinlock.h>
-----
static inline void spin_lock(spinlock_t *lock)
{
    raw_spin_lock(&lock->rlock);
}
```

代码中的数据结构 spinlock_t，就是前面提到的在多处理器之间共享的自旋锁在现实源码中的具体表现，透过层层定义，会发现实际上它就是个 volatile unsigned int 型变量：

```
<include/linux/spinlock_types.h>
-----
typedef struct raw_spinlock {
    volatile unsigned int raw_lock;
} raw_spinlock_t;

typedef struct spinlock {
```

² 此处是从所谓原子指令的使用效果的角度而言，像 ARM 中的 LDREX 和 STREX，当代码正在这两条指令间执行时，可以被换出处理器，但是因为指令本身的设计原理，它们可以在指令执行的效果上实现原子的操作。

```

union {
    struct raw_spinlock rlock;
};
} spinlock_t;

```

`spin_lock` 函数中调用的 `raw_spin_lock` 是个宏，其实现是处理器相关的，对于 ARM 处理器而言，最终展开为

```

static inline void raw_spin_lock (raw_spinlock_t *lock)
{
    preempt_disable();
    do_raw_spin_lock(lock)
}

```

函数首先调用 `preempt_disable` 宏，后者在定义了 `CONFIG_PREEMPT`，也即在支持内核可抢占的调度系统中时，将关闭调度器的可抢占特性。在没有定义 `CONFIG_PREEMPT` 时，`preempt_disable` 是个空定义，什么也不做。

真正的上锁操作发生在后面的 `do_raw_spin_lock` 函数中，不过在讨论该函数的实现细节前，先来看看为什么 `raw_spin_lock` 要先调用 `preempt_disable` 来关闭系统的可抢占性。在一个打开了 `CONFIG_PREEMPT` 特性的 Linux 系统中，一个在内核态执行的路径也有可能被切换出处理器，典型地，比如当前进程正在内核态执行某一系统调用时，发生了一个外部中断，当中断处理函数返回时，因为内核的可抢占性，此时将会出现一个调度点，如果 CPU 的运行队列中出现了一个比当前被中断进程优先级更高的进程，那么被中断的进程将会被换出处理器，即便此时它正运行在内核态。单处理器上的这种因为内核的可抢占性所导致的两个不同进程并发执行的情形，非常类似于 SMP 系统上运行在不同处理器上的进程之间的并发，因此为了保护共享的资源不会受到破坏，必须在进入临界区前关闭内核的可抢占性。因为 Linux 内核源码试图统一自旋锁的接口代码，即不论是单处理器还是多处理器，不论内核是否配置了可抢占特性，提供给外部模块使用的相关自旋锁代码都只有一份，所以可以看到在上述的 `raw_spin_lock` 函数中加入了内核可抢占性相关的代码，即便是在没有配置内核可抢占的系统上，外部模块也都统一使用相同的 `spin_lock` 和 `spin_unlock` 接口函数。

函数接着调用 `do_raw_spin_lock` 开始真正的上锁操作（为了便于后面的叙述，展开的嵌入汇编代码前加了行号标志 L，下同）：

```

static inline void do_raw_spin_lock (raw_spinlock_t *lock)
{
    unsigned long tmp;

    __asm__ __volatile__(
L1  "1:  ldrex %0, [%1]\n"
L2  "teq  %0, #0\n"
L3  "strexeq  %0, %2, [%1]\n"

```

```

L4  "teqeq    %0, #0\n"
L5  "bne  1b"
    : "=&r" (tmp)
    : "r" (&lock->raw_lock), "r" (1)
    : "cc");
    smp_mb();
}

```

do_raw_spin_lock 函数中嵌入的汇编代码段是 ARM 处理器上实现自旋锁的核心代码，它通过使用 ARM 处理器上专门用以实现互斥访问的指令 ldrex 和 strex 来达到原子操作的目的：

- "ldrex%0, [%1]"相当于"tmp = lock->raw_lock"，即读取自旋锁 V 的初始状态，放在临时变量 tmp 中。
- "teq %0, #0"判断 V 是否为 0，如果不为 0，表明此时自旋锁处于上锁状态，代码执行"bne 1b"指令，开始进入忙等待：不停地到标号 1 处读取自旋锁的状态，并判断是否为 0。
- "strexeq %0, %2, [%1]"这条指令是说，如果 V=0（自旋锁处于解锁的状态），说明可以进入临界区，那么就用常量 1 来更新 V 的值，并把更新操作执行的结果放到变量 tmp 中。
- "teqeq %0, #0"用来判断上一条指令对 V 的更新操作其结果 tmp 是否为 0，如果是 0 则表明更新 V 的操作成功，此时 V=1，代码可以进入临界区。如果 tmp≠0，则表明更新 V 的操作没有成功，代码执行"bne 1b"指令进入忙等待。

这里之所以要执行"teqeq %0, #0"，正是要利用 ldrex 和 strex 指令来达成原子操作的目的。

假设系统中有两个处理器 A 和 B，其上运行的代码现在都通过调用 spin_lock 试图进入临界区。开始的时候，自旋锁 V=0 处于解锁状态，注意这里是真正地并发执行。当处理器 A 执行完 L1 处的指令，尚未开始执行 L2 时，处理器 B 开始执行 L1，等到处理器 A 执行完 L2 准备执行 L3 时，处理器 B 执行完 L1。这样会发生什么情况呢？此时在处理器 A 和 B 看来，V 都是 0（因为处理器 B 执行完 L1 时，处理器 A 还没有执行 L3，因此 V 还没有被更新），这意味着它们都将以为自己可以成功获得锁而进入临界区，所以接下来它们都将试图去更新 V 为 1。谁先更新 V 并不重要，重要的是如果没有 L4 处的指令，处理器 A 和 B 都将跳过 L5 处的指令而进入临界区，而这意味着 spin_lock 函数对并发访问时的互斥管理是失败的，将可能在系统中引起非常严重的后果。

但是因为 L4 处代码的出现情况发生了变化，L4 处的代码在这种危急关头所起的作用得益于 strex 和 ldrex 指令，相对于 ARM 中普通的 str 与 ldr 指令，strex 和 ldrex 加入了对共享内存互斥访问的支持³。针对本例，在处理器 A 和 B 都使用 L1 处的 ldrex 来访问自旋锁 V

³ 关于这方面的具体技术细节，感兴趣的读者可查阅 ARM V6 手册，LDREX 和 STREX 是在 ARM V6 开始引入的对互斥访问提供支持的指令。

之后，在执行到 L4 时将导致只有其中一个处理器可以成功执行 L4，也即成功更新 V 为 1，tmp=0。另一个处理器将不会完成对 V 的更新动作，对它而言 tmp=1，意味着更新动作失败，这样它将不得不执行 L5 进入自旋状态。如此就可以保证对自旋锁 V 的“读取—检测—更新”操作序列的原子性。

与 spin_lock 相对的是 spin_unlock 函数，这是一个应该在离开临界区时调用的函数，用来释放此前获得的自旋锁。其外部接口定义如下：

```
<include/linux/spinlock.h>
-----
static inline void spin_unlock(spinlock_t *lock)
{
    raw_spin_unlock(&lock->rlock);
}

static inline void raw_spin_unlock (raw_spinlock_t * lock)
{
    do_raw_spin_unlock(lock);
    preempt_enable();
}
```

函数先调用 do_raw_spin_unlock 做实际的解锁操作，然后调用 preempt_enable() 函数打开内核可抢占性，对于没有定义 CONFIG_PREEMPT 的系统，该宏是个空定义。do_raw_spin_unlock 函数在 ARM 处理器上的代码如下：

```
static inline void do_raw_spin_unlock(raw_spinlock_t * lock)
{
    smp_mb();

    __asm__ __volatile__(
        "str  %1, [%0]\n"
        :
        : "r" (&lock->lock), "r" (0)
        : "cc");
}
```

解锁操作比获得锁的操作要相对简单，只需更新锁变量为 0 即可，在 ARM 平台上利用单条指令 str 就可以完成该任务，所以代码非常简单，直接用 str 指令将自旋锁的状态更新为 0，即解锁状态。针对 spin_lock 应该调用 spin_unlock 而不是其他形式的释放锁函数，驱动程序必须确保这种获得锁和释放锁函数调用的一致性。

4.3.2 spin_lock 的变体

在前面讨论 spin_lock 函数时，spin_lock 对多处理器系统中这种进程间真正的并发执行引起的竞态问题解决得很好，但是考虑图 4-1 所示这样一个场景：

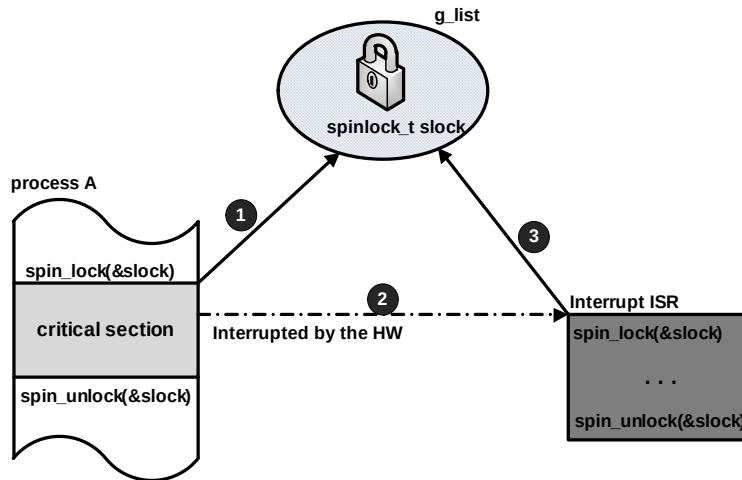


图 4-1 spin_lock 在中断并发情形下所引入的问题

处理器上的当前进程 A 因为要对某一全局性的链表 `g_list` 进行操作，所以在操作前通过调用 `spin_lock` 来进入临界区（图中标号 1 所示），当它正处于临界区中时，进程 A 所在的处理器上发生了一个外部硬件中断，此时系统必须暂停当前进程 A 的执行转而去处理该中断（图中标号 2 所示），假设该中断的处理例程中恰好也要操作 `g_list`，因为这是一个共享的全局变量，所以在操作之前也要调用 `spin_lock` 函数来对该共享变量进行保护（图中标号 3 所示），当中断处理例程中的 `spin_lock` 试图去获得自旋锁 `slock` 时，因为被它中断的进程 A 之前已经获得该锁，于是将导致中断处理例程进入自旋状态。在中断处理例程中出现一个自旋状态是非常致命的，因为中断处理例程必须在尽可能短的时间内返回，而此时它却必须自旋。同时被它中断的进程 A 因中断处理函数不能返回而无法恢复执行，也就不可能释放锁，所以将导致中断处理例程中的 `spin_lock` 一直自旋下去，导致死锁。出现这种特定情况的本质原因在于对锁的竞争发生在不能真正并发执行的两条路径上，如果可以并发执行，那么在上面的案例中，被中断的进程依然可以继续执行继而释放锁。对这种问题的解决导致了 `spin_lock` 函数其他变体的出现。

因处理外部的中断而引发 `spin_lock` 缺陷的例子，使得必须在这种情况下对 `spin_lock` 予以修正，于是出现了 `spin_lock_irq` 和 `spin_lock_irqsave` 函数。`spin_lock_irq` 函数接口定义如下：

```

<include/linux/spinlock.h>
-----
static inline void spin_lock_irq(spinlock_t *lock)
{
    raw_spin_lock_irq(&lock->rlock);
}

static inline void raw_spin_lock_irq(raw_spinlock_t *lock)
{
    local_irq_disable();

```

```

    preempt_disable();
    do_raw_spin_lock(lock);
}

```

其中的 `raw_spin_lock_irq` 函数的实现，相对于 `raw_spin_lock` 只是在调用 `preempt_disable` 之前又调用了 `local_irq_disable()`，后者在本章前面部分已经讨论过，用来关闭本地处理器响应外部中断的能力，这样在获取一个锁时就可以确保不会发生中断，从而避免上面提到的死锁问题。`local_irq_disable` 只能用来关闭本地处理器的中断，当一个通过调用 `spin_lock_irq` 拥有自旋锁 `V` 的进程在处理器 `A` 上执行时，虽然在处理器 `A` 上中断被关闭了，但是外部中断依然有机会发送处理器 `B` 上，如果处理器 `B` 上的中断处理函数也试图去获得锁 `V`，情况会怎样呢？因为此时处理器 `A` 上的进程可以继续执行，在它离开临界区时将释放锁，这样处理器 `B` 上的中断处理函数就可以结束此前的自旋状态。这从一个侧面说明通过自旋锁进入的临界区代码必须在尽可能短的时间内执行完毕，因为它执行的时间越长，别的处理器就越需要自旋以等待更长的时间（尤其是这种自旋发生在中断处理函数中），最糟糕的情况是进程在临界区中因为某种原因被换出处理器。所以作为使用自旋锁时一条确定的规则，任何拥有自旋锁的代码都必须是原子的，不能休眠。在实际的使用中，这条规则实践起来还是相当具有挑战性，远不像规则描述的那样直白，调用者需要仔细审视在拥有锁时的每个函数调用，因为睡眠有可能发生在这些函数的内部，比如以 `GFP_KERNEL` 作为分配掩码通过 `kmalloc` 函数来分配一块内存时，系统中空闲的内存不足以满足本次分配的情形虽然非常少见，但是毕竟存在这种可能性，一旦这种可能性被确定，`kmalloc` 会阻塞从而会被切换出处理器，如果 `kmalloc` 的调用者在此之前拥有某个自旋锁，那么这种情形下将对系统的稳定性造成极大的威胁。

如此，当知道一个自旋锁在中断处理的上下文中有可能会被使用到时，应该使用 `spin_lock_irq` 函数，而不是 `spin_lock`，后者只有在能确定中断上下文中不会使用到自旋锁的情形下才能使用。`spin_lock_irq` 对应的释放锁函数为 `spin_unlock_irq`，其接口定义为

```

<include/linux/spinlock.h>
-----
static inline void spin_unlock_irq(spinlock_t *lock)
{
    raw_spin_unlock_irq(&lock->rlock);
}

static inline void raw_spin_unlock_irq(raw_spinlock_t *lock)
{
    do_raw_spin_unlock(lock);
    local_irq_enable();
    preempt_enable();
}

```

可见，在 `raw_spin_unlock_irq` 函数中除了调用 `do_raw_spin_unlock` 做实际的解锁操作外，还会打开本地处理器上的中断，以及开启内核的可抢占性。

与 `spin_lock_irq` 类似的还有一个 `spin_lock_irqsave` 宏，它与 `spin_lock_irq` 函数最大的区别是，在关闭中断前会将处理器当前的 `FLAGS` 寄存器的值保存在一个变量中，当调用对应的 `spin_unlock_irqrestore` 来释放锁时，会将 `spin_lock_irqsave` 中保存的 `FLAGS` 值重新写回到寄存器中。对于 `spin_lock_irqsave` 和 `spin_unlock_irqrestore` 的使用场合，可参考前面关于 `local_irq_save` 和 `local_irq_restore` 的讨论。

下面是一个使用 `spin_lock_irqsave` 和 `spin_unlock_irqrestore` 函数的具体例子：

```
//定义一个全局性的 spinlock_t 变量 my_lock
spinlock_t my_lock;

//使用到 my_lock 锁的函数
void demo_write(){
    unsigned long flags;

    //进入到临界区之前调用 spin_lock_irqsave 来获得锁
    spin_lock_irqsave(&my_lock, flags);
    //进入临界区
    ...
    //完成临界区中的操作，准备离开临界区，调用 spin_unlock_irqrestore 来释放锁
    spin_unlock_irqrestore(&my_lock, flags);
}
```

另一个与中断处理相关的 `spinlock` 版本是 `spin_lock_bh` 函数，该函数用来处理进程与延迟处理导致的并发中的互斥问题。相对于 `spin_lock_irq` 函数，`spin_lock_bh` 用来关闭 `softirq` 的能力，关于 `softirq` 将在“中断处理”一章中讲解，此处只要知道 `spin_lock_bh` 的功能就可以了。该函数的上锁和解锁操作分别是

```
void spin_lock_bh(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

最后，自旋锁还设计了一组对应的非阻塞的版本，分别是

```
static inline int spin_trylock(spinlock_t *lock);
static inline int spin_trylock_irq(spinlock_t *lock);
spin_trylock_irqsave(lock, flags);
int spin_trylock_bh(spinlock_t *lock);
```

这些非阻塞版本的自旋锁函数在试图获得一个锁时，如果发现该锁处于上锁状态，会直接返回 0 而不是自旋（spin），如果成功获得锁则返回 1。

4.3.3 单处理器上的 spin_lock 函数

现在讨论单处理器上的 `spin_lock` 的问题，单处理器系统可分为内核不可抢占及可抢占两种。对于第一种系统而言，并发主要来源于外部中断等异步事件，所以在这种系统中，在

进入临界区时只需要关闭处理器的中断（调用 `local_irq_disable/local_irq_save`）即可，在离开临界区时只需要打开/恢复处理器的中断（调用 `local_irq_enable/local_irq_restore`）。对于第二种系统，并发来源除了中断与异常等异步事件外，还包括因为可抢占性导致的进程间的并发，所以在这种系统中，在进入临界区时除了要关闭处理器的中断，还需要关闭内核调度器的可抢占性。

Linux 内核为了统一单处理器和多处理器上这种竞态处理的代码，将 `spin_lock` 函数及其变体从多处理器系统延伸到了单处理器上。对于单处理器而言，如果是非抢占式系统，那么 `spin_lock/spin_unlock` 将等同于空操作；而对于内核可抢占的系统，`spin_lock/spin_unlock` 则分别用来关闭和打开可抢占性，此时它们等同于 `preempt_disable/preempt_enable`。而 `spin_lock_irq/spin_lock_irqsave` 和 `spin_unlock_irq/spin_unlock_restore` 在单处理器上则等同于 `local_irq_disable/local_irq_save` 和 `local_irq_enable/local_irq_restore`。如果是可抢占式系统，那么需要在上述的中断控制函数后再加上对内核可抢占性的 `preempt` 操作。

为了方便读者阅读，笔者将此处具体的对应关系简单地做了张表格，如表 4-1 所示。

表 4-1 多处理器与单处理器上的 `spin_lock`

锁 操 作		多处理器		单处理器	
加锁	解锁	加锁	解锁	加锁	解锁
<code>spin_lock</code>	<code>spin_unlock</code>	<code>preempt_disable</code> <code>do_raw_spin_lock</code>	<code>do_raw_spin_unlock</code> <code>preempt_enable</code>	<code>preempt_disable</code>	<code>preempt_enable</code>
<code>spin_lock_irq</code>	<code>spin_unlock_irq</code>	<code>local_irq_disable</code> <code>preempt_disable</code> <code>do_raw_spin_lock</code>	<code>do_raw_spin_unlock</code> <code>local_irq_enable</code> <code>preempt_enable</code>	<code>local_irq_disable</code> <code>preempt_disable</code>	<code>local_irq_enable</code> <code>preempt_enable</code>
<code>spin_lock_irqsave</code>	<code>spin_unlock_irqrestore</code>	<code>local_irq_save</code> <code>preempt_disable</code> <code>do_raw_spin_lock</code>	<code>do_raw_spin_unlock</code> <code>local_irq_restore</code> <code>preempt_enable</code>	<code>local_irq_save</code> <code>preempt_disable</code>	<code>local_irq_restore</code> <code>preempt_enable</code>
<code>spin_lock_bh</code>	<code>spin_unlock_bh</code>	<code>local_bh_disable</code> <code>preempt_disable</code> <code>do_raw_spin_lock</code>	<code>do_raw_spin_unlock</code> <code>preempt_enable</code> <code>local_bh_enable</code>	<code>local_bh_disable</code> <code>preempt_disable</code>	<code>local_bh_enable</code> <code>preempt_enable</code>

因此从代码移植性的角度考虑，即使在单处理器上只需要调用 `local_irq_disable/local_irq_enable` 来对共享资源进行保护时，也应该使用 `spin_lock_irq/spin_unlock_irq` 函数，因为若将来代码移植到多处理器上，则 `local_irq_disable/local_irq_enable` 将不足以保护共享资源，届时需要额外修改相应的代码。

4.3.4 读取者与写入者自旋锁 `rwlock`

`spin_lock` 类的函数在进入临界区时，对临界区中的操作行为不作细分，也就是说 `spin_lock` 不会考虑临界区中代码对共享资源访问的具体类型，只要是访问共享资源，就执行加锁操作。

但是有些时候，比如某些临界区的代码段只是去读这些共享的数据，并不会改写，如果采用 `spin_lock` 函数，意味着任一时刻只能有一个进程可以读取这些共享数据，如果系统中有大量对这些共享资源的读操作，很明显用 `spin_lock` 将会降低系统的性能。在对共享资源访问类型（读或者写）进行细分的基础上，提出了所谓读取者与写入者自旋锁的概念 `rwlock`。

与之前的 `spin_lock` 类比起来，这种锁比较有意思的地方在于：它允许任意数量的读取者同时进入临界区，但写入者必须进行互斥访问。一个进程想去读的话，必须检查是否有进程正在写，有的话必须自旋，否则可以获得锁。一个进程想去写的话，必须先检查是否有进程正在读或者写，有的话必须自旋。

相比较 `spinlock`，`rwlock` 在锁的定义以及 `irq` 与 `preempt` 操作方面没有任何不同，唯一不同的是，`rwlock` 针对读和写都设计了各自的锁操作函数，这些核心的上锁/解锁操作都是平台相关的，下面以 ARM 处理器为例，看看 `rwlock` 的实现机制。

先看写入者的上锁操作：

```
static inline int do_raw_write_lock (raw_rwlock_t * rw)
{
    unsigned long tmp;

    __asm__ __volatile__(
L1:  "1: ldrex %0, [%1]\n"
L2:  "    teq      %0, #0\n"
L3:  "    strexeq  %0, %2, [%1]\n"
L4:  "    teq      %0, #0\n"
L5:  "    bne      1b"
        : "=&r" (tmp)
        : "r" (&rw->lock), "r" (0x80000000)
        : "cc");

    smp_mb();
}
```

代码先在 L1 处把 `lock` 的值读进来，然后在 L2 处测试它是否为 0，如果是 0（表明没有人在使用锁，由此可见写入者要想成功获得锁，必须保证此前没有进程正在该锁上进行读或者写，因为一个进程不管因为读或者写而获得锁，都会改变锁的值使之不为 0），那么在 L3 处用 `0x80000000` 去更新 `lock` 的值。如果 `lock` 的值不为 0，表明之前该锁已被别的进程所使用（读或者写进程），那么该进程将执行 L5 进入自旋状态（`"bne 1b"`指令的意思是跳转到后面的标号 1 处执行）。L4 用来测试更新 `lock` 值为 `0x80000000` 的操作是否成功，关于这个测试的用途，在 `spinlock` 的代码中已详细讨论过。

写入者的解锁操作：

```
static inline int do_raw_write_unlock (raw_rwlock_t * rw)
```

```
{
    smp_mb();

    __asm__ __volatile__(
        "str    %1, [%0]\n"
        :
        : "r" (&rw->lock), "r" (0)
        : "cc");
}
```

代码很简单，将 lock 的值设为 0。

再看读取者的上锁操作：

```
static inline void do_raw_read_lock (raw_rwlock_t * rw)
{
    unsigned long tmp, tmp2;

    __asm__ __volatile__(
        L1  "1:  ldrex %0, [%2]\n"
        L2  "    adds    %0, %0, #1\n"
        L3  "    strexpl    %1, %0, [%2]\n"
        L4  "    rsbpl %0, %1, #0\n"
        L5  "    bmi      1b"
        : "=&r" (tmp), "=&r" (tmp2)
        : "r" (&rw->lock)
        : "cc");

    smp_mb();
}
```

代码先在 L1 处读入 lock 的值，然后在 L2 处将 lock 值加 1（因为 add 指令的标志后缀形式 s，所以会更新 flag），L3 是说如果 adds 更新 flag 标志导致其中的 N=0，那么就执行 strex，否则跳过该条指令。那么什么情况下 N=0 呢？假设 result 是前面 adds 指令的运算结果，那么 $N = (result \& (1 \ll 31)) ? 1 : 0$ 。通过前面写入者的上锁操作，显然在有写入者占有锁的情况下（lock=0x80000000），N=1，这种情况下直接到 L5 处执行，进程进入自旋状态。如果没有写进程占有锁，则基本上会得到该锁（说基本上，是因为还存在另一读进程与当前读进程竞争该锁的可能性，这种情况下的处理依靠的是 L4 处的指令，L4 的主要目的是防止多个读取者对 lock 值更新可能引起的混乱。一旦一个读取者进入临界区，与之竞争的读取者随后也可以成功进入）。

读取者解锁操作：

```
static inline void arch_read_unlock(raw_rwlock_t * rw)
{
    unsigned long tmp, tmp2;

    smp_mb();

    __asm__ __volatile__(
"1: ldrex %0, [%2]\n"
"   sub      %0, %0, #1\n"
"   strex    %1, %0, [%2]\n"
"   teq      %1, #0\n"
"   bne      1b"
: "=&r" (tmp), "=&r" (tmp2)
: "r" (&rw->lock)
: "cc");
}
```

读取者的解锁操作主要是将 lock 值减 1，因为上锁时读取者的操作是加 1。但是因为临界区可能有多个读取者，所以此处应该注意确保多个读取者对 lock 值的减 1 不会出现混乱。

相对于 spinlock 的多个版本，rwlock 同样有多个版本。对于读取者：

```
void read_lock4(rwlock_t *lock);
void read_lock_irq (rwlock_t *lock);
void read_lock_irqsave (rwlock_t *lock, unsigned long flags);
void read_unlock (rwlock_t *lock);
void read_unlock_irq (rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
```

对于写入者：

```
void write_lock (rwlock_t *lock);
void write_lock_irq (rwlock_t *lock);
void write_lock_irqsave (rwlock_t *lock, unsigned long flags);
void write_unlock (rwlock_t *lock);
void write_unlock_irq (rwlock_t *lock);
void write_unlock_irqsave(rwlock_t *lock, unsigned long flags);
```

try 版本：

```
int read_lock (rwlock_t *lock);
int write_lock (rwlock_t *lock);
```

从以上对读取者/写入者代码的实际分析可以看出（假设针对由同一读/写锁保护的共享资源）：

⁴ 实际的代码中是个宏定义，此处是宏的展开形式。下同。

(1) 如果当前有进程正在写，那么其他进程就不能读，当然也不能写。

(2) 如果当前有进程正在读，那么其他进程可以读，但是不能写。

如此，当一个进程试图写，只要有其他进程正在读或者正在写，它都必须自旋。

如此，当一个进程试图读，只要没有其他进程正在写，它都可以获得锁。

因此从概率上讲，当一个进程试图写时，成功获得锁的概率要低于一个进程试图读。在一个读/写相互依赖的生产者与消费者系统，这种设计思想会在一定程度上导致读取者饥饿（没有数据可读）。所以，在一个存在大量读取操作而数据的更新较少发生的系统中，使用读/写锁对共享资源进行保护，相对普通形式的自旋锁，无疑会大大提升系统性能。

4.4 信号量（semaphore）

相对于自旋锁，信号量的最大特点是允许调用它的线程进入睡眠状态。这意味着试图获得某一信号量的进程会导致对处理器拥有权的丧失，也即出现进程的切换。

4.4.1 信号量的定义与初始化

信号量的定义如下：

```
<include/linux/semaphore.h>
-----
struct semaphore {
    spinlock_t    lock;
    unsigned int  count;
    struct list_head wait_list;
};
```

其中，lock 是个自旋锁变量，用于实现对信号量的另一个成员 count 的原子操作。

无符号整型变量 count 用于表示通过该信号量允许进入临界区的执行路径的个数。

wait_list 用于管理所有在该信号量上睡眠的进程，无法获得该信号量的进程将进入睡眠状态。

如果驱动程序中定义了一个 struct semaphore 型的信号量变量，需要注意的是不要直接对该变量的成员进行赋值，而应该使用 sema_init 函数来初始化该信号量。sema_init 函数定义如下：

```
<include/linux/semaphore.h>
-----
static inline void sema_init(struct semaphore *sem, int val)
{
    static struct lock_class_key __key;
    *sem = (struct semaphore) __SEMAPHORE_INITIALIZER(*sem, val);
```

```
lockdep_init_map(&sem->lock.dep_map, "semaphore->lock", &__key, 0);
}
```

初始化主要通过__SEMAPHORE_INITIALIZER 宏完成：

```
#define __SEMAPHORE_INITIALIZER(name, n) \
{ \
    .lock      = __SPIN_LOCK_UNLOCKED((name).lock), \
    .count     = n, \
    .wait_list = LIST_HEAD_INIT((name).wait_list), \
}
```

所以 sema_init(struct semaphore *sem, int val)调用会把信号量 sem 的 lock 值设定为解锁状态，count 值设定为函数的调用参数 val，同时初始化 wait_list 链表头。

4.4.2 DOWN 操作

信号量上的主要操作是 DOWN 和 UP，在 Linux 内核中对信号量的 DOWN 操作有：

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_killable(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
int down_timeout(struct semaphore *sem, long jiffies);
```

上面这些函数中，驱动程序使用最频繁的是 down_interruptible 函数，本节将重点讨论该函数，之后再对其他 DOWN 操作的功能作一概述性的描述。

down_interruptible 函数定义如下：

<kernel/semaphore.c>

```
int down_interruptible(struct semaphore *sem)
{
    unsigned long flags;
    int result = 0;

    spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        result = __down_interruptible(sem);
    spin_unlock_irqrestore(&sem->lock, flags);

    return result;
}
```

函数首先通过对 spin_lock_irqsave 的调用来保证对 sem->count 操作的原子性，防止多个进

程对 `sem->count` 同时操作可能引起的混乱。如果代码成功进入临界区，则判断 `sem->count` 是否大于 0：如果 `count` 大于 0，表明当前进程可以获得信号量，就将 `count` 值减 1，然后退出；如果 `count` 不大于 0，表明当前进程无法获得该信号量，此时调用 `__down_interruptible`，由后者完成一个进程无法获得信号量时的操作，在内部调用 `__down_common(struct semaphore *sem, long state, long timeout)`，调用时的参数 `state = TASK_INTERRUPTIBLE`，`timeout = LONG_MAX`。所以当进程无法获得信号量时，最终调用的函数为 `__down_common`：

<kernel/semaphore.c>

```
static inline int __sched __down_common(struct semaphore *sem, long state,
                                       long timeout)
{
    struct task_struct *task = current;
    struct semaphore_waiter waiter;

    list_add_tail(&waiter.list, &sem->wait_list);
    waiter.task = task;
    waiter.up = 0;

    for (;;) {
        if (signal_pending_state(state, task))
            goto interrupted;
        if (timeout <= 0)
            goto timed_out;
        __set_task_state(task, state);
        spin_unlock_irq(&sem->lock);
        timeout = schedule_timeout(timeout);
        spin_lock_irq(&sem->lock);
        if (waiter.up)
            return 0;
    }

timed_out:
    list_del(&waiter.list);
    return -ETIME;

interrupted:
    list_del(&waiter.list);
    return -EINTR;
}
```

函数的功能是，首先通过对一个 `struct semaphore_waiter` 变量 `waiter` 的使用，把当前进程放到信号量 `sem` 的成员变量 `wait_list` 所管理的队列中，接着在一个 `for` 循环中把当前进程的状态设置为 `TASK_INTERRUPTIBLE`，再调用 `schedule_timeout` 使当前进程进入睡眠状态，

函数将停留在 `schedule_timeout` 调用上，直到再次被调度执行。当该进程再一次被调度执行时，`schedule_timeout` 开始返回，接下来根据进程被再次调度的原因进行处理：如果 `waiter.up` 不为 0，说明进程在信号量 `sem` 的 `wait_list` 队列中被该信号量的 UP 操作所唤醒，进程可以获得信号量，返回 0。如果进程是因为被用户空间发送的信号所中断或者是超时引起的唤醒，则返回相应的错误代码。因此对 `down_interruptible` 的调用总是应该坚持检查其返回值，以确定函数是已经获得了信号量还是因为操作被中断因而需要特别处理，通常驱动程序对返回的非 0 值要做的工作是返回-`ERESTARTSYS`⁵，比如下面的代码段：

```
//定义一个信号量
struct semaphore demosem;
sema_init(&demosem, 2);
if (down_interruptible (&demosem))
    return -ERESTARTSYS;
```

然而对 `down_interruptible` 的调用最常见的可能还是返回 0 表明调用者获得了信号量。为了让讨论具体化，下面以一个例子来说明，假设一个信号量 `sem` 的 `count=2`，说明允许有两个进程进入临界区，假设有进程 A、B、C、D 和 E 先后调用 `down_interruptible` 来获得信号量，那么进程 A 和 B 将得到信号量进入临界区，C、D 和 E 将睡眠在 `sem` 的 `wait_list` 中，此时的情形如图 4-2 所示：

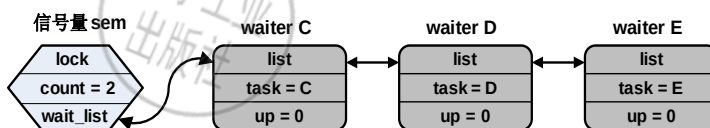


图 4-2 信号量上的睡眠进程

在接下来的 UP 操作中还会用到这里的例子，来讨论进程 A 和 B 结束临界区中的操作返回时执行 UP 操作对 `wait_list` 中进程 C、D 和 E 的影响。

在讨论完驱动程序最常使用的 `down_interruptible` 函数之后，再回过头来看看其他几种 DOWN 操作：

```
void down(struct semaphore *sem)
```

与 `down_interruptible` 相比，`down` 函数是不可中断的，这意味着调用它的进程如果无法获得信号量，将一直处于睡眠状态直到有别的进程释放了该信号量。从用户空间的角度，如果应用程序阻塞在了驱动程序的 `down` 函数中，将无法通过一些强制措施比如按 `Ctrl+C` 组合键等来结束该进程。因此，除非必要，否则驱动程序中应该避免使用 `down` 函数。

⁵ 返回-`ERESTARTSYS` 只是若干措施中最常见的一种，还有其他返回值可用，比如-`EAGAIN` 和-`EINTR`。驱动程序应该根据实际情况作出选择。

```
int down_killable(struct semaphore *sem)
```

睡眠的进程可以因收到一些致命性信号（fatal signal）被唤醒而导致获取信号量的操作被中断，在驱动程序中极少使用。

```
int down_trylock(struct semaphore *sem)
```

进程试图获得信号量，但若无法获得信号量则直接返回 1 而不进入睡眠状态，返回 0 意味着函数的调用者已经获得了信号量。

```
int down_timeout(struct semaphore *sem, long jiffies)
```

函数在无法获得信号量的情况下将进入睡眠状态，但是处于这种睡眠状态有时间限制，如果在 jiffies 指明的时间到期时函数依然无法获得信号量，则将返回一错误码-ETIME，在到期前进程的睡眠状态为 TASK_UNINTERRUPTIBLE。成功获得信号量的函数返回 0。

4.4.3 UP 操作

相对众多版本的 DOWN 操作，Linux 下只有一个 UP 函数：

```
<kernel/semaphore.c>
```

```
void up(struct semaphore *sem)
{
    unsigned long flags;

    spin_lock_irqsave(&sem->lock, flags);
    if (likely(list_empty(&sem->wait_list)))
        sem->count++;
    else
        __up(sem);
    spin_unlock_irqrestore(&sem->lock, flags);
}
```

如果信号量 sem 的 wait_list 队列为空，则表明没有其他进程正在等待该信号量，那么只要把 sem 的 count 加 1 即可。如果 wait_list 队列不为空，则说明有其他进程正睡眠在 wait_list 上等待该信号量，此时调用__up(sem)来唤醒进程：

```
<kernel/semaphore.c>
```

```
static ninline void __sched __up(struct semaphore *sem)
{
    struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list,
                                                         struct semaphore_waiter, list);
    list_del(&waiter->list);
    waiter->up = 1;
    wake_up_process(waiter->task);
}
```



```
}
```

下面在图 4-2 的基础上讨论此处的操作。__up 函数首先用 list_first_entry 取得 sem->wait_list 链表上的第一个 waiter 节点 C，然后将其从 sem->wait_list 链表中删除，waiter->up = 1，最后调用 wake_up_process 来唤醒 waiter C 上的进程 C。这样进程 C 将从之前 down_interruptible 调用中的 timeout = schedule_timeout(timeout)处醒来，waiter->up = 1，down_interruptible 返回 0，进程 C 获得信号量，进程 D 和 E 继续等待直到有进程释放信号量或者被用户空间中断掉。

即使不是信号量的拥有者，也可以调用 up 函数来释放一个信号量，这点与下节介绍的 mutex 是不同的。

在 Linux 系统中，信号量的一个常见的用途是实现互斥机制，这种情况下信号量的 count 值为 1，也就是任意时刻只允许一个进程进入临界区。为此 Linux 内核源码提供了一个宏 DECLARE_MUTEX，专门用于这种用途的信号量定义和初始化：

```
<include/linux/semaphore.h>
-----
#define DECLARE_MUTEX(name)\
    struct semaphore  name = __SEMAPHORE_INITIALIZER(name, 1)
```

该宏定义了一个 count=1 的信号量变量 name，并初始化了相关成员。所以接下来就可以使用信号量的 DOWN 和 UP 操作来实现互斥，比如下面的这个用 DECLARE_MUTEX 定义的信号量来实现互斥的代码段：

```
//先用 DECLARE_MUTEX 定义一个全局性的信号量 demo_sem
DECLARE_MUTEX(demo_sem);

//函数 demo_write 里使用 demo_sem 作互斥用
int demo_write()
{
    //打算进入临界区，调用 down_interruptible 获得信号量
    if (down_interruptible (&demo_sem))
        return -ERESTARTSYS;

    //成功获得信号量进入临界区
    ...
    //离开临界区，调用 up 释放信号量
    up(&demo_sem);
}
```

4.4.4 读取者与写入者信号量 rwsem

如同 spinlock 一样，如果对操作共享资源的访问类型进行细分，在普通信号量的基础上可以实现读取者与写入者信号量。这里的概念完全等同于读取者与写入者自旋锁，所以下面将不再仔细讨论读取者与写入者信号量的实现机制。

读取者与写入者信号量的定义如下：

```
<include/linux/rwsem-spinlock.h>
-----
struct rw_semaphore {
    __s32          activity;
    spinlock_t     wait_lock;
    struct list_head wait_list;
};
```

其中 activity 的确切含义是

- activity=0，表明当前在该信号量上没有任何活动的读取者或者是写入者。
- activity=-1，表明当前在该信号量上有一个活动的写入者。
- activity 为正值 n，表明当前信号量上有 n 个活动的读取者。

静态定义一个 rwsem 变量同时用 DECLARE_RWSEM 宏进行初始化：

```
<include/linux/rwsem-spinlock.h>
-----
#define __RWSEM_INITIALIZER(name) \
{ 0, __SPIN_LOCK_UNLOCKED(name.wait_lock), LIST_HEAD_INIT((name).wait_list) \
  __RWSEM_DEP_MAP_INIT(name) }

#define DECLARE_RWSEM(name) \
    struct rw_semaphore name = __RWSEM_INITIALIZER(name)
```

对一个 rwsem 变量动态初始化使用 init_rwsem 宏，其展开形式为

```
void __init_rwsem(struct rw_semaphore *sem)
{
    sem->activity = 0;
    spin_lock_init(&sem->wait_lock);
    INIT_LIST_HEAD(&sem->wait_list);
}
```

rwsem 的初始状态是没有任何活动的读取者与写入者。

读取者的 DOWN 操作：

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

读取者的 UP 操作：

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

写入者的 DOWN 操作：

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

写入者的 UP 操作:

```
void up_write(struct rw_semaphore *sem)
```

4.5 互斥锁 mutex

用 `count=1` 的信号量实现的互斥方法还不是 Linux 下经典的用法，Linux 内核针对 `count=1` 的信号量重新定义了一个新的数据结构 `struct mutex`，一般都称其为互斥锁或者互斥体。同时内核根据使用场景的不同，把用于信号量的 DOWN 和 UP 操作在 `struct mutex` 上作了优化与扩展，专门用于这种新的数据类型。

4.5.1 互斥锁的定义与初始化

互斥锁 `mutex` 的概念本来就来自 `semaphore`，如果去除掉那些跟调试相关的成员，`struct mutex` 和 `struct semaphore` 并没有本质的不同：

```
<include/linux/mutex.h>
-----
struct mutex {
    /* 1: unlocked, 0: locked, negative: locked, possible waiters */
    atomic_t          count;
    spinlock_t        wait_lock;
    struct list_head   wait_list;
    #if defined(CONFIG_DEBUG_MUTEXES) || defined(CONFIG_SMP)
        struct thread_info *owner;
    #endif
};
```

如同 `struct semaphore` 一样，对 `struct mutex` 的初始化不能直接通过操作其成员变量的方式进行，而应该利用内核提供的宏或者函数。

定义一个静态的 `struct mutex` 变量同时初始化的方法是利用内核的 `DEFINE_MUTEX`：

```
<include/linux/mutex.h>
-----
#define __MUTEX_INITIALIZER(lockname) \
    { .count = ATOMIC_INIT(1), \
      .wait_lock = __SPIN_LOCK_UNLOCKED(lockname.wait_lock), \
      .wait_list = LIST_HEAD_INIT(lockname.wait_list) \
    }

#define DEFINE_MUTEX(mutexname) \
    struct mutex mutexname = __MUTEX_INITIALIZER(mutexname)
```

如果在程序执行期间要初始化一个 `mutex` 变量，则可以使用 `mutex_init` 宏。去除掉那些与调试相关的操作之后，`mutex_init` 宏可以展开成如下的函数定义形式：

```
void mutex_init(struct mutex *lock)
{
    atomic_set(&lock->count, 1);
    spin_lock_init(&lock->wait_lock);
    INIT_LIST_HEAD(&lock->wait_list);
}
```

4.5.2 互斥锁的 DOWN 操作

互斥锁 `mutex` 上的 DOWN 操作在 Linux 内核中为 `mutex_lock` 函数，定义如下：

```
<kernel/mutex.c>
-----
void __sched mutex_lock(struct mutex *lock)
{
    might_sleep();
    /*
     * The locking fastpath is the 1->0 transition from
     * 'unlocked' into 'locked' state.
     */
    __mutex_fastpath_lock(&lock->count, __mutex_lock_slowpath);
    mutex_set_owner(lock);
}
```

函数的设计思想体现在 `__mutex_fastpath_lock` 和 `__mutex_lock_slowpath` 两条主线上，`__mutex_fastpath_lock` 用来快速判断当前可否获得互斥锁，如果成功获得锁，则函数直接返回，否则进入到 `__mutex_lock_slowpath` 函数中。这种设计是基于这样一个事实：想要获得某一互斥锁的代码绝大部分时候都可以成功获得。由此延伸开来在代码层面就是，`mutex_lock` 函数进入 `__mutex_lock_slowpath` 的概率很低。

`__mutex_fastpath_lock` 是一平台相关函数，下面以 ARM 处理器为例，分析其代码实现：

```
<arch/arm/include/asm/mutex.h>
-----
static inline void __mutex_fastpath_lock(atomic_t *count, void (*fail_fn)(atomic_t *))
{
    int __ex_flag, __res;

    __asm__ (
        L1 "ldrex%0, [%2] \n\t"
        L2 "sub      %0, %0, #1 \n\t"
        L3 "strex %1, %0, [%2]    "
        : "=&r" (__res), "=&r" (__ex_flag)
        : "r" (&(count)->counter)
```

```

        : "cc", "memory" );

    __res |= __ex_flag;
    if (unlikely(__res != 0))
        fail_fn(count);
}

```

函数在 L1 处通过 `ldrex` 完成 `__res = count->counter`，L2 处完成 `__res = __res - 1`，L3 处试图用 `__res` 的当前值来更新 `count->counter`。这里说“试图”是因为这个更新的操作未必会成功，主要是考虑到可能有别的进程也在操作 `count->counter`，为不使这种可能的竞争引起对 `count->counter` 值更新的混乱，这里用了 ARM 指令中用于实现互斥访问的指令 `ldrex` 和 `strex`（前面在 `spinlock` 的代码分析时已经提过）。`ldrex` 和 `strex` 保证了对 `count->counter` 的“读取—更新—写回”操作序列的原子性。如果 L3 处的更新操作成功，那么 `__ex_flag` 将为 0。

接下来在 `__res |= __ex_flag` 执行完之后，通过 `if` 语句判断 `__res` 是否为 0，有两种情况会导致 `__res` 不为 0：一是在调用这个函数前 `count->counter=0`，表明互斥锁已经被别的进程获得，这样 L2 处的 `__res = -1`；二是在 L3 处的更新操作不成功，表明当前有另外一个进程也在对 `count->counter` 进行同样的操作。这两种情况都将导致 `__mutex_fastpath_lock` 不能直接返回，而是进入 `fail_fn`，也就是调用 `__mutex_lock_slowpath`。

此处 `if` 语句中的 `unlikely` 是利用 GCC 编译优化扩展的一个宏，这里的意思是条件语句 `__res != 0` 为真的可能性很小，编译器借此可以调整一些编译后代码的顺序达到某种程度的优化。与之对应的是 `likely`。

如果 `__mutex_fastpath_lock` 函数不能在第一时间获得互斥锁返回，那么将进入 `__mutex_lock_slowpath`，正如其名字所预示的那样，代码将进入一段艰难坎坷的旅途。

在 Linux 源码中，`__mutex_lock_slowpath` 函数与信号量 DOWN 操作中的 `down` 函数非常相似，不过 `__mutex_lock_slowpath` 在把当前进程放入 `mutex` 的 `wait_list` 之前会试图多次询问 `mutex` 中的 `count` 是否为 1，也就是说当前进程在进入 `wait_list` 之前会多次考察别的进程是否已经释放了这个互斥锁。这主要基于这样一个事实：拥有互斥锁的进程总是会在尽可能短的时间里释放。如果别的进程已经释放了该互斥锁，那么当前进程将可以获得该互斥锁而没有必要再去睡眠。

4.5.3 互斥锁的 UP 操作

互斥锁的 UP 操作为 `mutex_unlock`，函数定义如下：

```

<kernel/mutex.c>
-----
void __sched mutex_unlock(struct mutex *lock)
{
    /*

```

```

    * The unlocking fastpath is the 0->1 transition from 'locked'
    * into 'unlocked' state:
    */
#ifdef CONFIG_DEBUG_MUTEXES
    /*
     * When debugging is enabled we must not clear the owner before time,
     * the slow path will always be taken, and that clears the owner field
     * after verifying that it was indeed current.
     */
    mutex_clear_owner(lock);
#endif
    __mutex_fastpath_unlock(&lock->count, __mutex_unlock_slowpath);
}

```

和 `mutex_lock` 函数一样，`mutex_unlock` 函数也有两条主线：`__mutex_fastpath_unlock` 和 `__mutex_unlock_slowpath`，分别用于对互斥锁的快速和慢速解锁操作。

`__mutex_fastpath_unlock` 定义如下：

```

<arch/arm/include/asm/mutex.h>
-----
static inline void
__mutex_fastpath_unlock(atomic_t *count, void (*fail_fn)(atomic_t *))
{
    int __ex_flag, __res, __orig;

    __asm__ (
        "ldrex%0, [%3] \n\t"
        "add      %1, %0, #1 \n\t"
        "strex%2, %1, [%3] \n\t"
        : "=&r" (__orig), "=&r" (__res), "=&r" (__ex_flag)
        : "r" (&(count)->counter)
        : "cc", "memory");

    __orig |= __ex_flag;
    if (unlikely(__orig != 0))
        fail_fn(count);
}

```

这里除了是将 `count->counter` 的值加 1 以外，代码和 `__mutex_fastpath_lock` 中的几乎完全一样。在最后的 if 语句中，导致代码中 `__orig` 不为 0 也有两种情况：一是在调用这个函数前 `count->counter` 不为 0，表明在当前进程占有互斥锁期间有别的进程竞争该互斥锁；二是对 `count->counter` 的更新操作不成功，表明当前有另外一个进程也在对 `count->counter` 进行操作，这种情况主要是针对别的进程此时调用 `mutex_lock` 函数导致的竞争，因为互斥的原因别的进程此时不可能调用 `mutex_unlock`。这种情况的处理是非常重要的，不只是关系到 `count->counter` 正确更新的问题，还涉及能否防止一个唤醒操作的丢失。

在没有别的进程竞争该互斥锁的情况下，`__mutex_fastpath_unlock` 函数要完成的工作最简单，把 `count->counter` 的值加 1 然后返回。如果有别的进程在竞争该互斥锁，那么函数进入 `__mutex_unlock_slowpath`，这个函数主要用来唤醒在当前 `mutex` 的 `wait_list` 中休眠的进程，如同 `up` 函数一样。

4.6 顺序锁 seqlock

顺序锁的设计思想是，对某一共享数据读取时不加锁，写的时候加锁。为了保证读取的过程中不会因为写入者的出现导致该共享数据的更新，需要在读取者和写入者之间引入一整形变量，称为顺序值 `sequence`。读取者在开始读取前读取该 `sequence`，在读取后再重读该值，如果与之前读取到的值不一致，则说明本次读取操作过程中发生了数据更新，读取操作无效。因此要求写入者在开始写入的时候要更新 `sequence` 的值。

Linux 内核中 `seqlock` 定义如下：

```
<include/linux/seqlock.h>
```

```
typedef struct {
    unsigned sequence;
    spinlock_t lock;
} seqlock_t;
```

无符号型整数 `sequence` 用来协调读取者与写入者的操作，`spinlock` 变量 `lock` 在多个写入者之间做互斥使用。

程序中如果想静态定义一个 `seqlock` 并同时初始化，可以使用 `DEFINE_SEQLOCK` 宏，该宏会定义一个 `seqlock_t` 型变量并初始化其 `sequence` 为 0，`lock` 为 0⁶：

```
<include/linux/seqlock.h>
```

```
#define DEFINE_SEQLOCK(x) \
    seqlock_t x = __SEQLOCK_UNLOCKED(x)

#define __SEQLOCK_UNLOCKED(lockname) \
    { 0, __SPIN_LOCK_UNLOCKED(lockname) }
```

如果要动态初始化一个 `seqlock` 变量，可以使用 `seqlock_init`：

```
<include/linux/seqlock.h>
```

```
#define seqlock_init(x) \
    do { \
        (x)->sequence = 0; \
    }
```

⁶ 自旋锁在解锁与上锁状态时的数值其实依赖于具体的实现，大部分情况下解锁状态时的值为 0，上锁状态为 1。

```
spin_lock_init(&(x)->lock);    \
} while (0)
```

下面看看写入者在 seqlock 上的上锁操作 write_seqlock:

```
<include/linux/seqlock.h>
-----
static inline void write_seqlock(seqlock_t *sl)
{
    spin_lock(&sl->lock);
    ++sl->sequence;
    smp_wmb();
}
```

写入者在对写之前需要先获得 seqlock 上的自旋锁 lock, 这说明在写入者之间必须保证互斥操作, 如果某一写入者成功获得 lock, 那么需要更新 sequence 的值以便让其他写入者知道共享数据发生了更新。写入者与写入者之间并不需要 sequence。

写入者在 seqlock 上的解锁操作 write_sequnlock:

```
<include/linux/seqlock.h>
-----
static inline void write_sequnlock(seqlock_t *sl)
{
    smp_wmb();
    sl->sequence++;
    spin_unlock(&sl->lock);
}
```

主要的工作是释放自旋锁 lock, 至于写入者对 sequence 的更新, 主要是用来告诉读取者有数据更新发生, 所以必须确保 sequence 的值在写入的前后发生变化。在此基础上 sequence 提供的另外一个信息是写入过程有没有结束, 这是用 sequence 的最低位来完成的, 如果 sequence & 0 为 0 表明写入过程已经结束, 否则表明写入过程正在进行。接下来会在读取者的 seqlock 操作函数中看到 sequence 的这两种用途。

某一写入者可以使用 write_tryseqlock 来保证在无法获得 lock 时不让自己进入自旋状态(当然也就无法更新数据)而直接返回 0, 成功获得锁则返回 1:

```
<include/linux/seqlock.h>
-----
static inline int write_tryseqlock(seqlock_t *sl)
{
    int ret = spin_trylock(&sl->lock);

    if (ret) {
        ++sl->sequence;
        smp_wmb();
    }
    return ret;
}
```



```
}
```

读取者在读取开始前需要先调用 `read_seqbegin` 函数，该函数主要用来返回读取开始之前的 `sequence` 值：

```
<include/linux/seqlock.h>
-----
static __always_inline unsigned read_seqbegin(const seqlock_t *sl)
{
    unsigned ret;

repeat:
    ret = sl->sequence;
    smp_rmb();
    if (unlikely(ret & 1)) {
        cpu_relax();
        goto repeat;
    }

    return ret;
}
```

从函数的实现也可以看出，如果当前正好有写入者在进行写操作，那么该函数将不停循环直到写过程结束，前面曾提到 `sequence` 最低位的用途，这里正好是其实际使用的地方。另一方面，从读取者对写入过程结束的循环等待可以看出，写入者的实际写入操作占用的时间不应太长。

内核还给读取者提供了一个 `read_seqretry` 函数，与 `read_seqbegin` 的返回值一起使用，来判定本次的读取操作是否有效：

```
<include/linux/seqlock.h>
-----
static __always_inline int read_seqretry(const seqlock_t *sl, unsigned start)
{
    smp_rmb();

    return (sl->sequence != start);
}
```

函数的参数 `start` 是读取者在读取操作之前调用 `read_seqbegin` 获得的初始值。如果本次读取无效（读取过程中发生了数据更新），那么 `read_seqretry` 返回 1，否则返回 0。

下面分别给出写入者和读取者利用上面介绍的 `seqlock` 函数进行数据读/写协调的例子：

```
//定义一个全局的 seqlock 变量 demo_seqlock
DEFINE_SEQLOCK(demo_seqlock);

//对于写入者的代码...
```

```

//实际写之前调用 write_seqlock 获取自旋锁，同时更新 sequence 的值
write_seqlock(&demo_seqlock);
//获得自旋锁之后，调用 do_write 进行实际的写入操作
do_write();
//写入结束，调用 write_sequnlock 释放锁
write_sequnlock(&demo_seqlock);

//对于读取者的代码...
unsigned start;
do{
    //读操作前先得到 sequence 的值 start，用以在读操作结束时判断是否发生数据更新
    //注意读操作无须获得锁
    start = read_seqbegin(&demo_seqlock);
    //调用 do_read 进行实际的读操作
    do_read();
}while(read_seqretry(&demo_seqlock, start));//如果有数据更新，再重新读取

```

如果考虑到中断安全的问题，可以使用读取者与写入者的对应版本：

```

<include/linux/seqlock.h>
-----
write_seqlock_irq(lock)
write_seqlock_irqsave(lock, flags)
write_seqlock_bh(lock)

write_sequnlock_irq(lock)
write_sequnlock_irqrestore(lock, flags)
write_sequnlock_bh(lock)

read_seqbegin_irqsave(lock, flags)
read_seqretry_irqrestore(lock, iv, flags)

```

前面曾讨论过读取者与写入者自旋锁 `rwlock`，对比这里的 `seqlock`，会发现两者非常相似。不同之处在于 `seqlock` 在写的时候只与其他写入者互斥，而 `rwlock` 在写的时候与读取者和写入者都互斥。因此当要保护的资源很小很简单，会很频繁被访问并且写入操作很少发生且必须快速时，就可以使用 `seqlock`。

4.7 RCU

RCU 的全称是 Read-Copy-Update，意即读/写—复制—更新，在 Linux 提供的所有内核互斥设施当中属于一种免锁机制。同前面讨论过的读取者与写入者自旋锁 `rwlock`、读取者与写入者信号量 `rwsem` 以及顺序锁 `seqlock` 一样，RCU 的适用模型也是读取者与写入者共存的系统。与 `rwlock`、`rwsem` 和 `seqlock` 不同的是，RCU 中的读取和写入操作无须考虑两者之

间的互斥问题。通过前面的讨论我们知道，加锁与解锁都要涉及内存操作，同时还伴有内存屏障方法的引入，这些都使得锁操作的系统开销变得很大。在此基础上，Linux 内核加入了对 RCU 这种免锁的互斥访问机制的支持。虽然在设备驱动程序中使用 RCU 的机会很少，但是通过对 RCU 的讨论以及与其他加锁机制的对比，可以更深入理解 Linux 内核为设备驱动程序提供的这些内核设施各自的利弊。

RCU 并不是很新的概念，但是 Linux 内核直到 2.5 版本才开始引入这种机制。其核心思想讲起来也许并不复杂，但是从 Linux 内核中相关源码的设计看来，还是很晦涩难懂的。限于篇幅的原因，本书并不打算在源码的层面上详细分析其实现过程。

RCU 的原理简单地说，是将读取者和写入者要访问的共享数据放在一个指针 `p` 中，读取者通过 `p` 来访问其中的数据，而写入者则通过修改 `p` 来更新数据。在具体的实现上，读取者一方并没有太多的事要做，大量的工作集中在写入者一方。免锁的实现必定要通过双方恪守一定的规则才可达成。

4.7.1 读取者的 RCU 临界区

对于读取者来说，如果要访问共享数据，所要做的工作首先是调用 `rcu_read_lock` 和 `rcu_read_unlock` 函数构建自己所谓的读取者侧的临界区（read-side critical section），然后在临界区中获得指向共享数据区的指针，实际的读取操作就是对该指针的引用。这里对于读取者的一个明确的规则是，对指针的引用必须在临界区中完成，离开临界区之后不应该出现任何形式的对该指针的引用。在临界区中，关闭内核的可抢占性意味着在临界区中不会因为中断的发生导致进程的切换，而且作为确定的规则，临界区中的代码不能发生睡眠。简言之，临界区中的代码不应该导致任何形式的进程切换。

虽然函数的名称中含有 `lock` 字样，但是 `rcu_read_lock` 和 `rcu_read_unlock` 实际要做的工作仅仅是分别关闭和打开内核的可抢占性而已。

4.7.2 写入者的 RCU 操作

RCU 操作中写入者要完成的工作是重新分配一个被保护的共享数据区，（视具体情况决定是否）将老数据区的数据复制到新数据区，然后再根据需要修改新数据区，最后用新数据区指针替换掉老的指针，替换指针的操作是一个原子操作，不需要与读取者进行互斥操作。在写入者做完这些工作之后，后续的所有 RCU 的读取操作都将访问到这个新的共享数据区。但是写入者在用新指针替换掉老指针之后还不能马上释放老指针指向的数据区所占用的内存空间，这是因为系统中还可能存在对老指针的引用。这主要发生在如下两种情况：一是在单处理器的范围看，假设读取者在进入 RCU 临界区后，刚获得共享区的指针之后发生了一个中断（因为 `rcu_read_lock` 只是关闭了内核可抢占性，并没有关闭本地的中断），如果写入者恰好是中断处理函数中的行为，那么当中断返回后，被中断进程在 RCU 临界区

中继续执行时，将会继续引用老指针；另一个可能是在多处理器系统，当处理器 A 上的一个读取者进入 RCU 临界区并获得共享数据区中的指针后，在其还没来得及引用该指针时，处理器 B 上的一个写入者更新了指向共享数据区的指针，这样处理器 A 上的读取者也将引用到老指针。

因此，写入者在替换掉共享区的指针后，老指针所指向的共享数据区所在的空间还不能马上释放。写入者需要和内核共同协作，在确定所有对老指针的引用都结束后才可以释放老指针指向的内存空间。为此，写入者在用新指针替换掉老指针之后需要做的操作是，调用 `call_rcu` 函数向内核注册一个回调函数，内核在确定所有对老指针的引用都结束时调用该回调函数，回调函数的功能则主要是释放老指针指向的内存空间。下面是 `call_rcu` 的原型：

```
void call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *rcu));
```

RCU 的写入者负责在替换掉老指针之后调用 `call_rcu` 向内核注册一回调函数，回调函数负责实现释放老指针指向的内存空间，`call_rcu` 中的参数 `func` 就是指向该回调函数的指针。函数中的 `head` 是内核在调用 `func` 时传递到 `func` 中的参数。实际的使用中，会把 `struct rcu_head` 内嵌到共享数据所在的结构体中，这样在回调函数中可以通过传进来的 `struct rcu_head` 指针，使用 `container_of` 宏获得指向旧的共享数据区的指针，然后调用 `kfree` 释放旧的数据区。

关于回调函数被调用的时机，内核必须确保没有对老指针的引用时才能调用回调函数释放老指针。内核确保没有读取者对老指针的引用是基于以下规则：所有可能的对共享数据区指针的不一致引用一定是发生在读取者的 RCU 临界区（RCU 的一条明确的规则是，离开临界区之后不应该出现任何形式的对该指针的引用），因为临界区由 `rcu_read_lock` 和 `rcu_read_unlock` 界定，所以就单处理器范围而言，在临界区中一定不会发生进程的切换（`rcu_read_lock` 将会关闭内核的可抢占性，这也是读取者在其临界区中的代码一定不会出现进程切换的原因），所以如果在某一 CPU 上发生了一次进程切换，那么所有对老指针的引用都会结束，之后的读取者再进入 RCU 临界区都将看到新指针。因此，内核确定没有对老指针的引用的条件是：系统中所有处理器上都至少发生了一次进程切换。

4.7.3 RCU 使用的特点

通过前面对 RCU 读取者与写入者操作的讨论，可以看到 RCU 实质上是对读取者与写入者自旋锁 `rwlock` 的一种优化：RCU 的读取者在读取数据时除了关闭内核可抢占性外，与普通数据的读取操作没有任何区别，读取者也不关心当前有没有写入者正在对共享数据区进行操作；而对于 `rwlock`，在读取者打算工作时，必须确保没有写入者正在工作，否则读取进程将进入自旋状态，所以 RCU 可以让多个读取者与写入者同时工作。相对于读取者，RCU 写入者的开销比较大，它需要申请新的内存空间，正常的数据更新操作，向内核注册回调函数，同时也要考虑与其他写入者之间的互斥问题，但是与 `rwlock` 不一样的是，写入者不

需要考虑与读取者的互斥问题。

可见，RCU 读取者性能的提升是在增加写入者负担的前提下完成的。因此在一个读取者与写入者共存的系统中，按照设计者的说法，如果写入者的操作比例在 10% 以上，那么就应该考虑其他的互斥方法，反之采用 RCU 的实现可以获得更高的性能。另外，RCU 的设计思想决定了必须要以指针的方式来访问被保护资源。

为了在代码中使用 RCU，所有 RCU 相关的操作都应该使用内核提供的 RCU API 函数，以确保 RCU 机制的正确使用，这些 API 主要集中在指针和链表的操作。

下面是一个 RCU 的典型用法范例：

```
//假设 struct shared_data 是一个在读取者和写入者之间共享的受保护数据
struct shared_data{
    int a;
    int b;
    struct rcu_head rcu;
};

//
//读取者侧的代码。读取者调用 rcu_read_lock 和 rcu_read_unlock 构建它的读取临界区，所
//有对指向被保护资源指针的引用都应该只在临界区中出现，而且临界区中的代码不能睡眠
//
static void demo_reader(struct shared_data *ptr)
{
    struct shared_data *p = NULL;
    rcu_read_lock();
    //调用 rcu_dereference 获得 ptr 的指针
    p = rcu_dereference(ptr);
    if(p)
        do_something_withp(p);
    rcu_read_unlock();
}

//
//写入者侧的代码
//
//写入者提供的回调函数，用于释放老指针
static void demo_del_oldptr(struct rcu_head *rh)
{
    struct shared_data * p = container_of(rh, struct shared_data, rcu);
    kfree(p);
}

static void demo_writer(struct shared_data *ptr)
{

```

```

struct shared_data *new_ptr = kmalloc(...);
...
new_ptr->a = 10;
new_ptr->b = 20;
//用新指针更新老指针
rcu_assign_pointer(ptr,new_ptr);
//调用 call_rcu 让内核在确保所有对老指针 ptr 的引用都结束后回调 demo_del_oldptr 释
//放老指针
call_rcu(ptr->rcu, demo_del_oldptr);
}

```

上面的例子中，写入者在调用 `rcu_assign_pointer` 更新了老指针之后，为了在所有对老指针的引用都消失后释放老指针指向的空间，使用 `call_rcu` 向系统注册了一个回调函数 `demo_del_oldptr`，系统将在确定没有对老指针的引用之后调用该函数。另一个类似的函数是 `synchronize_rcu`，这个函数可能会阻塞，因为它要等待所有对老指针的引用都结束时才返回，函数返回意味着系统中所有对老指针的引用都消失了，此时再释放老指针的空间是安全的。如果在中断上下文中执行写入者的操作，那么就不能使用 `synchronize_rcu`，而应该使用 `call_rcu`。

4.8 原子变量与位操作

有时候需要保护的共享资源可能只是个简单的整型变量，即便如此对它的操作依然需要保证原子性，否则就会造成不可预料的结果，看一看下面这个例子：

```

//一个在 Task A 与 B 之间共享的全局变量
int g_flag = 0;

//Task A
void taska_addflag()
{
    g_flag++;
}

//Task B
void taskb_addflag()
{
    g_flag++;
}

```

系统中的 Task A 和 B 运行之后，`g_flag` 会是多少，2 吗？答案是有可能！这是一个典型的对变量的非原子操作可能导致错误结果的例子。原因在于即便是简单如 `g_flag++` 这样的操作，在汇编指令级，也很可能产生如下代码：

```
//将 g_flag 的值从内存中读到 EAX 寄存器
L1  "movl $g_flag, %eax"
//将 EAX 寄存器中的值加 1
L2  "incl %eax"
//将 EAX 寄存器中的值写回到 g_flag 中
L3  "movl %eax, $g_flag"
```

如果 Task A 先被调度运行，在其执行完 L1 尚未执行 L2 时，Task B 开始被调度执行，在其执行完整代码后 `g_flag=1`，然后系统又调度 Task A 从 L2 处继续执行，因为此前已执行了 L1，导致 `EAX=0`，经 L2 后，`EAX=1`，于是在 L3 执行完后，`g_flag=1`。

在这个例子中当然可用 `spinlock` 来保证 `g_flag++` 操作的原子性，但是加锁操作导致的开销较大，用在这里总是有点浪费。此时可以考虑利用特定架构上的汇编指令来完成原子操作，比如上面的 `g_flag++`，可以用类似“`incl $g_flag`”这样的汇编指令实现。显然这种原子操作在纯粹的 C 语言层面难以达成，必须借助汇编语言或者是嵌入到 C 中的汇编指令来实现。

针对这种特殊的原子操作，Linux 源码中定义了一个类型为 `atomic_t` 的原子变量。`atomic_t` 的具体定义为

```
<include/linux/types.h>
-----
typedef struct {
    int counter;
} atomic_t;
```

为此 Linux 系统中定义了一大堆以“`atomic_`”打头的原子操作函数，这些函数的实现都依赖于特定的硬件平台。为了给读者一个具体的感受，下面挑出能解决上述 `g_flag++` 问题的 `atomic_inc` 函数来分析它在 x86 和 ARM 上的实现。

x86 上的 `atomic_inc` 函数：

```
<arch/x86/include/asm/atomic.h>
-----
static inline void atomic_inc(atomic_t *v)
{
    asm volatile("lock incl %0"
                  : "+m" (v->counter));
}
```

x86 上用一条带有“`lock`”前缀的 `inc` 指令来保证原子变量 `v` 加 1 操作的原子性，“`lock`”前缀在 x86 上的作用是在执行 `inc` 指令时独占系统总线，这样即便系统总线上还有其他的 master，在 `inc` 执行期间也无法修改 `v->counter` 的值。

ARM 上的 `atomic_inc` 函数：

```
<arch/arm/include/asm/atomic.h>
-----
#define atomic_inc(v)      atomic_add(1, v)
```

```

static inline void atomic_add(int i, atomic_t *v)
{
    unsigned long tmp;
    int result;

    __asm__ __volatile__("@ atomic_add\n"
        "1: ldrex %0, [%3]\n"
        "    add      %0, %0, %4\n"
        "    strex    %1, %0, [%3]\n"
        "    teq      %1, #0\n"
        "    bne     1b"
        : "=&r" (result), "=&r" (tmp), "+Qo" (v->counter)
        : "r" (&v->counter), "Ir" (i)
        : "cc");
}

```

ARM 使用 `ldrex` 和 `strex` 来保证 `add` 指令的原子性,这在前面分析 `spinlock` 的代码时已经讲过。

这样,再回到刚开始的 `g_flag++` 的例子,使用原子变量就可以轻松解决问题:

```

//一个在 Task A 与 B 之间共享的原子变量,并用 ATOMIC_INIT 将其中的 counter 初始化为 0
atomic_t g_flag = ATOMIC_INIT(0);

//Task A
void taska_addflag()
{
    atomic_inc(&g_flag);
}

//Task B
void taskb_addflag()
{
    atomic_inc(&g_flag);
}

```

这样 Task A 和 B 执行之后, `g_flag.counter` 的结果一定是 2。

前面已经看到了 `atomic_t` 的定义,它是个 `struct` 类型,所以在需要整型变量的地方不能直接用 `atomic_t` 变量,否则会产生编译错误。另外在实际使用时应当注意, `atomic_t` 型变量只能保证自身操作的原子性,对一个由多个整型变量组成的共享数据,即便把这些变量全部声明为原子型,对它们的使用也都是用 `atomic` 类的函数,也不能保证对该共享数据操作的原子性,此时需要用到前面介绍的其他互斥方法。

与单个原子变量相对的是位操作的原子性,其实现原理和原子变量完全一样,依赖于特定的处理器指令实现对变量上的位进行原子性的操作和测试。

4.9 等待队列

等待队列并不是一种互斥机制，之所以把等待队列放在这里作为独立的一节，是因为本书在讨论接下来的一些内核设施的实现机制时，会经常用到等待队列的概念。等待队列是内核定义的一种数据结构，用来实现其他的内核机制，比如下面要提到的完成接口 `completion` 以及工作队列等。

等待队列本质上是一双向链表，由等待队列头和队列节点构成，当运行的进程要获得某一资源而暂不可得时，进程有时候需要等待，此时它可以进入睡眠状态，内核为此生成一个新的等待队列节点将睡眠的进程挂载到等待队列中。

4.9.1 等待队列头 `wait_queue_head_t`

内核为等待队列头节点定义的数据结构为

```
<include/linux/wait.h>
-----
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

其中：

`spinlock_t lock`

等待队列的自旋锁，用做等待队列被并发访问时的互斥机制。

`struct list_head task_list`

双向链表结构体，用来将等待队列构成链表。

如果程序需要定义一个等待队列，有两种方法。一是通过 `DECLARE_WAIT_QUEUE_HEAD` 宏来完成等待队列头对象的静态定义与初始化：

```
<include/linux/wait.h>
-----
#define __WAIT_QUEUE_HEAD_INITIALIZER(name) {
    .lock      = __SPIN_LOCK_UNLOCKED(name.lock),
    .task_list = { &(name).task_list, &(name).task_list } }

#define DECLARE_WAIT_QUEUE_HEAD(name) \
    wait_queue_head_t name = __WAIT_QUEUE_HEAD_INITIALIZER(name)
```

二是通过 `init_waitqueue_head` 宏在程序运行期间初始化一个头节点对象：

```
<include/linux/wait.h>
-----
#define init_waitqueue_head(q) \
do { \
    static struct lock_class_key __key; \
    __init_waitqueue_head((q), &__key); \
} while (0)

<kernel/wait.c>
-----
void __init_waitqueue_head(wait_queue_head_t *q, struct lock_class_key *key)
{
    spin_lock_init(&q->lock);
    lockdep_set_class(&q->lock, key);
    INIT_LIST_HEAD(&q->task_list);
}
```

4.9.2 等待队列的节点

等待队列节点的数据结构为

```
<include/linux/wait.h>
-----
typedef struct __wait_queue wait_queue_t;
typedef int (*wait_queue_func_t)(wait_queue_t *wait, unsigned mode, int flags, void *key);

struct __wait_queue {
    unsigned int flags;
    void *private;
    wait_queue_func_t func;
    struct list_head task_list;
};
```

其中：

`unsigned int flags`

唤醒等待队列上的进程时，该标志会影响唤醒操作的行为模式。内核为此定义了 `WQ_FLAG_EXCLUSIVE`，如果一个等待节点设置了该标志位，表明睡眠在其上的进程在被唤醒时具有排他性，关于这方面的细节将留到等待队列实际应用的讨论中。

`void *private`

等待队列的私有数据，实际使用中用来指向睡眠在该节点上的进程的 `task_struct` 结构。

`wait_queue_func_t func`

当该节点上的睡眠进程需要被唤醒时执行的唤醒函数。

```
struct list_head task_list
```

用来将各独立的等待队列节点链接起来形成链表。

程序可以通过 DECLARE_WAITQUEUE 来定义并初始化一个等待队列的节点：

```
<include/linux/wait.h>
-----
#define __WAITQUEUE_INITIALIZER(name, tsk) {
    .private    = tsk,
    .func       = default_wake_function,
    .task_list  = { NULL, NULL } }

#define DECLARE_WAITQUEUE(name, tsk)
    wait_queue_t name = __WAITQUEUE_INITIALIZER(name, tsk)
```

如果要在程序运行期初始化一个等待队列节点对象，可以使用 init_waitqueue_entry 函数：

```
<include/linux/wait.h>
-----
static inline void init_waitqueue_entry(wait_queue_t *q, struct task_struct *p)
{
    q->flags = 0;
    q->private = p;
    q->func = default_wake_function;
}
```

4.9.3 等待队列的应用

等待队列常用的模式便是实现进程的睡眠等待，当某一进程在运行过程中需要的资源暂时无法获得时，进程将进入睡眠状态以让出处理器资源给其他进程。进程进入睡眠状态，意味着进程将从调度器的运行队列中移除，此时进程将被挂载到某一等待队列的节点中。为了实现进程的睡眠机制，系统会产生一个新的等待队列节点，然后将进程的 task_struct 对象放到等待队列节点对象的 private 成员中。

内核中对等待队列的核心操作是等待（wait）与唤醒（wake up），这里打算把相关内容的讨论推迟到具体使用到这些操作的时候，比如下面即将介绍的完成接口。

4.10 完成接口 completion

本章的最后来讨论一个被称为“完成接口 completion”的同步机制，该机制被用来在多个执行路径间作同步使用，也即协调多个执行路径的执行顺序。完成接口在内核中用一个数据结构 struct completion 表示，定义如下：

```
<include/linux/completion.h>
```

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

其中，done 表示当前 completion 的状态。wait 是一等待队列，用来管理当前等待在该 completion 上的所有进程。

如果要静态定义一个 struct completion 变量并初始化，可以使用 DECLARE_COMPLETION 宏：

```
<include/linux/completion.h>
```

```
#define DECLARE_COMPLETION(work) \
    struct completion work = COMPLETION_INITIALIZER(work)
```

如果要重新初始化一个已使用过的 struct completion 变量，可以使用 INIT_COMPLETION 宏：

```
<include/linux/completion.h>
```

```
#define INIT_COMPLETION(x) ((x).done = 0)
```

如果要动态初始化一个 struct completion 变量，则应该调用 init_completion 函数：

```
<include/linux/completion.h>
```

```
static inline void init_completion(struct completion *x)
{
    x->done = 0;
    init_waitqueue_head(&x->wait);
}
```

完成接口 completion 对执行路径间的同步可以通过等待者与完成者模型来表述。对于等待者的行为，内核定义的一个典型的函数是 wait_for_completion：

```
<kernel/sched.c>
```

```
void __sched wait_for_completion(struct completion *x)
{
    wait_for_common(x, MAX_SCHEDULE_TIMEOUT, TASK_UNINTERRUPTIBLE);
}
```

wait_for_completion 内调用 wait_for_common 来使当前进程以 TASK_UNINTERRUPTIBLE 睡眠在 completion x 上的 wait 队列中。wait_for_common 内部调用了 do_wait_for_common 来做这件事：

```
<kernel/sched.c>
```

```
static inline long __sched
do_wait_for_common(struct completion *x, long timeout, int state)
{
    if (!x->done) {
```

```

DECLARE_WAITQUEUE(wait, current);

__add_wait_queue_tail_exclusive(&x->wait, &wait);
do {
    if (signal_pending_state(state, current)) {
        timeout = -ERESTARTSYS;
        break;
    }
    __set_current_state(state);
    spin_unlock_irq(&x->wait.lock);
    timeout = schedule_timeout(timeout);
    spin_lock_irq(&x->wait.lock);
} while (!x->done && timeout);
__remove_wait_queue(&x->wait, &wait);
if (!x->done)
    return timeout;
}
x->done--;
return timeout ? 1;
}

```

等待者首先检查 completion 中的 done 成员，它表示当前在 completion 上的完成者数量，如果没有完成者，那么等待者将进入睡眠队列进行等待，这种睡眠是不可中断的。DECLARE_WAITQUEUE 定义并初始化了一个等待节点 wait，代表当前进程的 current 变量将会记录到 wait 的 private 变量，wait 中的 func 函数指针指向 default_wake_function，当 wait 上的进程被唤醒时将调用该函数。进程需要睡眠时，通过 __add_wait_queue_tail_exclusive 把 wait 节点加入到 completion 管理的等待队列的尾部，wait->flags |= WQ_FLAG_EXCLUSIVE，等待节点 wait 中的这个 flags 标记将在完成者的唤醒操作中使用到。

若干时间之后，进程因某种原因被唤醒，表现为从 schedule_timeout 函数返回，它将检查 done 成员和 timeout 变量以决定后续的行为。timeout>0 表示进程还没有超时，x->done=0 表示 completion 上还没有完成者，此时当前进程如果没有信号需要处理，将继续睡眠。

如果进程睡眠超时，将返回 timeout 的值。如果没有超时且有完成者在 completion 上出现（这是绝大多数会出现的情形），那么进程将离开睡眠队列，在将完成者数量减 1 之后，等待者结束等待状态返回。

如果考虑到进程进入睡眠队列的状态及睡眠超时时间的设定，内核提供了 wait_for_completion 的另外一些版本供使用：

```
int wait_for_completion_interruptible(struct completion *x);
```

可中断的等待状态。

```
int wait_for_completion_killable(struct completion *x);
```

可杀死的等待状态。等该的进程可以被一个 kill signal 唤醒并中止等待状态。

```
unsigned long wait_for_completion_timeout(struct completion *x,unsigned long timeout);
```

不可中断的等待状态，但在 timeout 指定的时间到期之后，进程将中止等待状态。

```
unsigned long wait_for_completion_interruptible_timeout(
    struct completion *x, unsigned long timeout);
```

可中断的等待状态，但在 timeout 指定的时间到期之后，进程将中止等待状态。

```
unsigned long wait_for_completion_killable_timeout(
    struct completion *x, unsigned long timeout);
```

可杀死的等待状态，但在 timeout 指定的时间到期之后，进程将中止等待状态。

对于完成者的行为，内核为其定义的函数是 complete 和 complete_all，前者只唤醒一个等待者，后者将唤醒所有的等待者。

<kernel/sched.c>

```
void complete(struct completion *x)
{
    unsigned long flags;

    spin_lock_irqsave(&x->wait.lock, flags);
    x->done++;
    __wake_up_common(&x->wait, TASK_NORMAL, 1, 0, NULL);
    spin_unlock_irqrestore(&x->wait.lock, flags);
}
```

函数先将完成者的数量加 1，然后调用 __wake_up_common 函数执行唤醒等待者的操作，注意这里的第三和第四个参数，分别表示排他性唤醒的个数和唤醒标志。

<kernel/sched.c>

```
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
    int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;

    list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
        unsigned flags = curr->flags;

        if (curr->func(curr, mode, wake_flags, key) &&
            (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}
```

函数遍历当前 completion 所管理的等待队列的每一个节点, 此时 nr_exclusive=1, flags 中含有 WQ_FLAG_EXCLUSIVE 标志, 意味着本次唤醒只会唤醒一个等待者。func 指向 default_wake_function, 用来做实际的唤醒工作。

相对于 complete 一次只唤醒一个等待者, complete_all 用来唤醒 completion 等待队列上的所有等待者进程:

```
<kernel/sched.c>
-----
void complete_all(struct completion *x)
{
    unsigned long flags;

    spin_lock_irqsave(&x->wait.lock, flags);
    x->done += UINIT_MAX/2;
    __wake_up_common(&x->wait, TASK_NORMAL, 0, 0, NULL);
    spin_unlock_irqrestore(&x->wait.lock, flags);
}
```

注意 complete_all 在这里假设完成者的最大数量是(UINIT_MAX)/2, 这是个很大的值, 现实系统中很少有等待者进程的数量会达到该值, 因此在 complete_all 之后 completion 中的 done 值将失去其本来的意义, 如果后面要继续该 completion, 应该调用前面提过的 INIT_COMPLETION 宏。

4.11 本章小结

本章详细介绍了 Linux 内核提供给驱动程序使用的各种互斥和同步设施, 其中最常用的是自旋锁 spinlock 和互斥锁 mutex。

自旋锁不会进入睡眠, 因而最适合在不允许睡眠的上下文环境中执行, 比如中断处理函数。因为一个进程试图获得锁而不可得时, 实际上处于忙等待状态, 因此要求获得锁的进程在尽可能短的时间内完成对共享资源的访问, 然后释放锁。自旋锁根据不同的应用场景有不同的变体, 读者应该深入理解这些自旋锁的幕后实现机制, 以便在实际使用时能够选定正确的自旋锁来实现对共享资源的保护。

而互斥锁的实现来源于信号量, 所以如果一个进程在进入临界区前试图调用互斥锁时, 有可能会进入休眠状态, 所以在中断上下文中严格禁止使用互斥锁和信号量。

虽然自旋锁是一种基于忙等待的互斥机制, 但是现实中被自旋锁保护的临界区代码往往可以很快执行完毕释放掉锁, 这种情况下如果用互斥锁的话, 可能引起的进程切换的开销往往要比忙等待大得多, 因此不能以为进程睡眠一定会比忙等待更有利于系统性能。

第 5 章

中断处理

外部设备与中央处理器交互一般有两种手段：轮询和中断。对于轮询，要求处理器不停地查询外设的状态，在此期间处理器不能做别的事情。而中断不要求处理器不停地查询自己的状态，而是在自己的状态满足处理器的要求时主动发送一个硬件信号给处理器，后者在接收到这一信号时，会挂起当前正在执行的任务转而去处理外设的中断信号。

现代设备绝大多数采用中断的方式与处理器进行沟通，因此设备驱动程序必须能够支持设备的中断特性。处理器在中断到达时会根据不同的中断号找到对应的处理函数对该信号进行处理，这些处理函数称为中断处理例程 **ISR**（**I**nterrupt **S**ervice **R**outine）¹，设备驱动程序负责为管理的设备提供中断处理例程并向系统注册。从设备发出中断信号，到处理器最终调用中断处理例程进行处理，期间会经过很多步骤，这个过程构成了中断处理框架。不同的操作系统对中断处理框架的设计不尽相同，但是要达到的目的是一样的，那就是最终调用设备的中断处理例程。

本章将先描述 **Linux** 系统下的中断处理框架设计，然后在此基础上讨论设备驱动程序如何利用内核提供的接口函数向系统挂载中断处理例程，最后讨论中断上下文的相关内容，包括软中断等。

5.1 中断的硬件框架

处理器一般只有两根左右的中断引脚，而管理的外设却很多。为了解决这个问题，现代设备的中断信号线并不是与处理器直接相连，而是与一个称为中断控制器的设备相连接，后者才跟处理器的中断引脚直接连接。中断控制器一般可以通过处理器进行编程配置，所以常称为可编程中断控制器 **PIC**（**P**rogrammable **I**nterrupt **C**ontroller）。图 5-1 是一个典型的中断硬件连接的系统框架图：

¹ 关于 **ISR** 的定义，有的书可能将处理器接收到中断信号后的跳转地址指向的“函数”称为 **ISR**，本书将这部分称为“通用中断处理函数”，而将驱动程序提供的中断处理函数称为 **ISR**。总之，这只是个称谓问题。

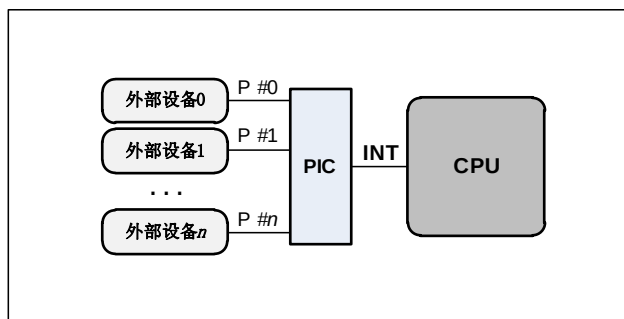


图 5-1 中断连接框图

图中，PIC 的输出中断信号线连接到处理器的 INT 引脚上，这是处理器专门用来接收中断信号的 pin 脚。外部设备的中断线连接到 PIC 的 pin 引脚上，这是 PIC 用来接收外设中断的 pin 脚，比如外部设备 1 的中断线通过 P0 连到 PIC 上。在实际的硬件平台上，PIC 有的在 CPU 外部，比如 x86 平台上的 8259 控制器；有的被封装到了 CPU 的内部，这广泛见于嵌入式领域，一颗 SoC 芯片内部集成了处理器和各种外部设备的控制器，其中包括 PIC。中断方面的内容常常涉及硬件平台的差别，但是这里不会纠结于某个具体的硬件设计，而是希望相关的内容可以很快被读者吸纳到自己手边的平台上。为了让讨论更加方便，下面把图中中断的连接逻辑作为通用的硬件平台。

5.2 PIC 与软件中断号

实际使用中，在处理器能处理外部设备的中断前，常常需要对 PIC 进行配置，配置工作常常作为操作系统初始化任务的一部分。当然中断处理框架也需要提供适当的 PIC 配置接口函数供设备驱动程序调用，因为设备驱动所管理的设备也许并不是一开始就连接到 PIC 的某一中断引脚上的。如果在系统运行起来之后，某一外设才被用户接入系统，那么它的驱动程序应该负责配置 PIC 的对应引脚，使该外设能正常中断处理器。

对 PIC 的配置工作主要包括：

- (1) 设定外部设备中断触发电信号的类型，常见的触发类型有水平触发和边沿触发。
- (2) 将外设的中断引脚编号映射到处理器可见的软件中断号 irq。
- (3) 屏蔽掉某些外部设备的中断触发²。

为了让处理器可以配置自己，PIC 常常需要提供一系列的控制寄存器。这些控制寄存器可

² 屏蔽一个中断有两层含义：一是指在处理器内部的中断屏蔽，这种情况下处理器不会响应外部中断信号；二是指在 PIC 层面的屏蔽，此时屏蔽只针对某个 PIC 中断引脚信号。

以完成上述所有配置工作，并且配置粒度可以细分到 PIC 的每一个中断输入引脚 P。此处一个需要明确定义的概念是软件中断号 `irq`，它是发生设备中断时处理器从 PIC 中读到的中断号码，在操作系统建立的中断处理框架内，会使用这个 `irq` 号来标识一个外设的中断并调用对应的中断处理例程。作为描述的示例，考虑图 5-1 中外部设备 0 触发的一个中断电信号被处理的大体流程。PIC 将首先接收到该信号，如果它没有被屏蔽，那么 PIC 应该在 INT 引脚上产生一个中断信号告诉处理器。后者在接收到该信号后会从 PIC 那里得到一个特定的标识号码，该号码告诉中断处理框架，是设备 0 发生了中断。于是中断处理框架会调用设备 0 的中断处理例程，此处的这个特定的标识设备 0 的中断号码就称为软件中断号 `irq` 或者中断号 `irq`。

此处还有一个概念需要提一下，那就是中断向量表（`vector table`）。这其实是处理器内部的一个概念，因为处理器除了会被外部设备中断，其内部也可能产生异常等事件。当这些事发生时，CPU 必须暂停当前的工作，转而去处理中断或者异常，因此处理器需要知道到哪里去获得这些中断或异常的处理函数的目标地址。中断向量表就用来解决这个问题，其每一项都是一个中断或异常处理函数的入口地址。外部设备的中断常常对应向量表中的某一项，这是个通用的外部中断处理函数的入口，因此在进入通用的中断处理函数之后，系统必须要知道正在处理的中断是哪一个设备产生的，而这正是由前面提到的软件中断号 `irq` 决定的。中断向量表中的内容由操作系统在初始化阶段来填写，对于外部中断，操作系统负责实现一个通用的外部中断处理函数，然后把这个函数的入口地址放到中断向量表中的对应位置。

5.3 通用的中断处理函数

当有外部中断发生时，预先设计好的处理器硬件逻辑往往会做一些特定的动作，为从软件层面发起的中断处理做准备工作。不同的处理器有不同的逻辑设计，但这些动作常常包括把当前任务的上下文寄存器保存在一个特定的中断栈中，屏蔽掉处理器响应外部中断的能力等。在这些动作的结束部分，硬件逻辑根据中断向量表中的外部中断对应的入口地址，开始调用由操作系统提供的通用中断处理函数。

不同的架构平台上通用中断处理函数的实现也不尽相同，但在开始部分，都会设法从 PIC 中得到导致本次中断发生的外部设备对应的软件中断号 `irq`，这部分代码通常都是用汇编语言实现，在 Linux 源码树中散落在各个特定架构对应的目录中。然后通用处理函数开始调用一个 C 函数，大部分平台上这个 C 函数的名字是 `do_IRQ`，但也有例外，比如 ARM 平台上是 `asm_do_IRQ`，本书采用 `do_IRQ` 来指代该 C 函数的名称。

中断处理的绝大部分流程都浓缩在了这个 C 函数当中，当这个函数返回时，通用中断处理函数余下部分的代码将完成中断现场恢复的工作，这也标志着整个中断处理流程的结束：

被中断的任务开始继续执行，仿佛中断根本没有发生过一样³。

通常，处理器在接收到外部的中断信号时，硬件逻辑会自动屏蔽处理器响应外部中断的能力，因此如果操作系统实现的中断处理框架不主动打开中断的话，整个中断处理的流程是在中断关闭的情况下进行的。因为各个设备的中断处理函数一般是由驱动程序实现的，内核无法保证这些中断处理函数执行时间的长短，如果某一中断处理函数执行时间过长，则将会导致系统可能很长时间无法接收中断，这可能会使某些外部设备丢失数据或者操作系统响应时间变长等。为了解决这一问题，Linux 内核为驱动程序提供的中断处理机制分成了两个部分：HARDIRQ 和 SOFTIRQ。前者是在中断关闭的情况下执行⁴，用来完成中断发生后最关键的操作，它的执行时间应该尽可能短。后者是在中断开启的情况下进行，此时外部设备仍可以继续中断处理器，驱动程序因此可以将一些比较耗时的工作延迟到这部分执行。在 `do_IRQ` 函数中，对 `irq_enter` 的调用可以认为是 HARDIRQ 部分的开始，而 SOFTIRQ 则在 `irq_exit` 中完成。

5.4 do_IRQ 函数

上节提到，`do_IRQ` 函数从通用中断处理函数中发起，负责整个中断处理流程中实质性的中断处理任务。虽然该函数在各个平台上的实现代码不尽相同，但是原理基本上大同小异，一个典型的实现如下：

```
void __irq_entry do_IRQ(unsigned int irq, struct pt_regs *regs)
{
    struct pt_regs *old_regs = set_irq_regs(regs);
    irq_enter();
    check_stack_overflow();
    generic_handle_irq(irq);
    irq_exit();
    set_irq_regs(old_regs);
}
```

先看该函数的两个参数，`irq` 是该函数的调用者——通用中断处理函数从 PIC 中得到的软件中断号，`regs` 是保存下来的被中断任务的执行现场，不同的处理器有不同的执行现场，也就是有不同的寄存器。

³ 对于内核可抢占的系统来说，如果之前被中断的路径运行在内核态，那么在中断返回时会启动调度器以确定是否进行进程切换。对于没有启用可抢占性的内核，被中断的内核路径继续执行，中断的返回不会导致调度器的介入。

⁴ 内核设计者为设备驱动程序的中断处理框架设定的理想状况，然而真正的设备中断处理函数由各个设备驱动程序自己提供，它们具备足够的特权等级来打开处理器响应中断的能力。因此，内核设计者不得不额外提供一些防护措施来防止设备驱动程序员一些非常规的操作，即便这种防护措施是非常有限的，我们接下来会谈到这个问题。

函数首先调用 `set_irq_regs` 将一个 per-CPU 型的指针变量 `__irq_regs` 保存到 `old_regs` 中，然后将 `__irq_regs` 赋予了一个新值 `regs`，这样中断处理过程中，系统中的每一个 CPU 都可以通过 `__irq_regs` 来访问系统保存的中断现场。在函数的结束，调用 `set_irq_regs(old_regs)` 来恢复 `__irq_regs`。`__irq_regs` 一般用来在调试或者诊断时打印当前栈的信息，也可以通过这些保存的中断现场寄存器判断出被中断的进程当时运行在用户态还是内核态。

接下来 `irq_enter` 会更新系统中的一些统计量，同时会把当前栈中的 `preempt_count` 变量加上 `HARDIRQ_OFFSET` 来标识一个 `HARDIRQ` 中断上下文：`preempt_count() += HARDIRQ_OFFSET`，`HARDIRQ` 是 Linux 下对中断处理上半部分的称谓，与之对应的是中断处理的下半部分 `SOFTIRQ`，此处 `irq_enter` 告诉系统现在进入了中断处理的上半部分。与 `irq_enter` 行为配对的是 `irq_exit`，在当前中断处理完成准备退出时调用，除了更新一些系统统计量和清除中断上下文的标识外，它还有一个重要的功能是处理软中断，也就是中断处理的下半部分，本书将在“延迟操作”一章详细讨论软中断的实现机制。

`check_stack_overflow()` 函数用来检查当前中断是否会导致栈的溢出，因为每次中断发生时系统都会做保护现场的动作，从代码的层面，就是将系统的寄存器压入中断栈中。理想情况下，一个中断处理结束时将恢复现场，也就是将之前在栈中保存的寄存器弹出堆栈，因此不会发生栈溢出的情况。但是如果中断处理函数中打开了处理器响应外部中断的能力，那就有可能在当前中断正在被处理时，处理器又接收到了新的中断，也就是所谓的中断嵌套，这将导致系统重复地进行中断现场保护的动作，甚至发生大量的中断嵌套行为，使得栈不断增长，从而出现堆栈的溢出，影响到系统的稳定性。为此，系统使用 `check_stack_overflow` 函数来对栈是否溢出进行检查，如果发现本次中断有可能导致栈的溢出，通常会打印出当前栈的信息 (`dump_stack`)，对于某些启用了 `watchdog` 的系统，也可能会强制系统进行 `reset` 动作。

`do_IRQ` 的核心是调用 `generic_handle_irq` 函数，后者在其函数调度链中负责对当前发生的中断进行实际的处理：

```
<include/linux/irq.h>
-----
static inline void generic_handle_irq(unsigned int irq)
{
    struct irq_desc *desc = &irq_desc[irq];
    desc->handle_irq(irq, desc);
}
```

函数通过软件中断号 `irq` 来索引数组 `irq_desc`，得到一个 `struct irq_desc` 类型的指针变量 `desc`，然后调用其成员函数 `handle_irq` 对当前中断进行实际的处理。`irq_desc` 是个 `struct irq_desc` 类型的数组，在 Linux 的整个中断处理框架中非常重要，起着沟通从通用的中断处理函数到设备特定的中断处理例程之间的桥梁作用，图 5-2 展示了该数组的组成结构：

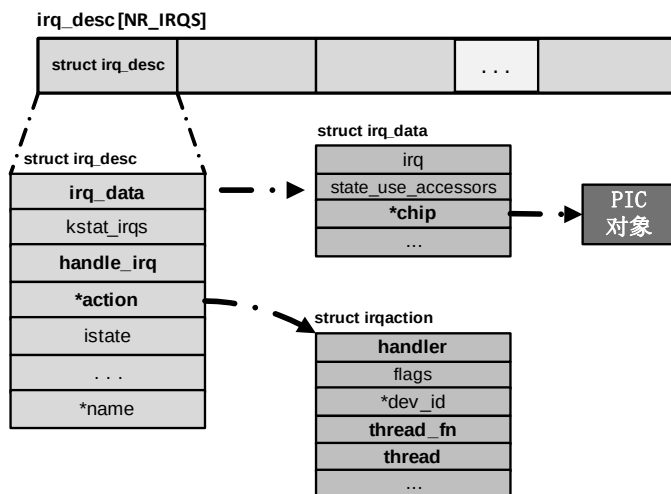


图 5-2 irq_desc 数组的构成形式

通常，其定义和默认值如下：

<kernel/irq/irqdesc.c>

```

struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {
    [0 ... NR_IRQS-1] = {
        .handle_irq = handle_bad_irq,
        .depth      = 1,
        .lock       = __RAW_SPIN_LOCK_UNLOCKED(irq_desc->lock),
    }
};
  
```

NR_IRQS 是个平台相关的常量，用来表示特定的平台上可以处理的外部中断的数量。Linux 操作系统初始化期间通过调用 early_irq_init 函数来对这个数组初始化：

<kernel/irq/irqdesc.c>

```

nt __init early_irq_init(void)
{
    int count, i, node = first_online_node;
    struct irq_desc *desc;

    init_irq_default_affinity();
    printk(KERN_INFO "NR_IRQS:%d\n", NR_IRQS);
    desc = irq_desc;
    count = ARRAY_SIZE(irq_desc);

    for (i = 0; i < count; i++) {
        desc[i].irq_data.irq = i;
        desc[i].irq_data.chip = &no_irq_chip;
        desc[i].kstat_irqs = alloc_percpu(unsigned int);
    }
}
  
```

```

        irq_settings_clr_and_set(desc, ~0, _IRQ_DEFAULT_INIT_FLAGS);
        alloc_masks(desc + i, GFP_KERNEL, node);
        desc_smp_init(desc + i, node);
        lockdep_set_class(&desc[i].lock, &irq_desc_lock_class);
    }
    return arch_early_irq_init();
}

```

数组的类型 `struct irq_desc` 是个非常重要的数据结构, 在下面的讨论中会经常用到。定义如下:

```

<include/linux/irqdesc.h>
-----
struct irq_desc {
    struct irq_data      irq_data;    struct timer_rand_state *timer_rand_state;
    unsigned int __percpu *kstat_irqs; irq_flow_handler_t      handle_irq;
#ifdef CONFIG_IRQ_PREFLOW_FASTEOI
    irq_preflow_handler_t preflow_handler;
#endif
    struct irqaction      *action;    /* IRQ action list */
    unsigned int          status_use_accessors;
    unsigned int          istate; unsigned int          depth;          /* nested irq disables */
    unsigned int          wake_depth; /* nested wake enables */
    unsigned int          irq_count; /* For detecting broken IRQs */
    unsigned long         last_unhandled; /* Aging timer for unhandled count */
    unsigned int          irqs_unhandled;
    raw_spinlock_t        lock;
#ifdef CONFIG_SMP
    const struct cpumask   *affinity_hint;
    struct irq_affinity_notify *affinity_notify;
#endif
#ifdef CONFIG_GENERIC_PENDING_IRQ
    cpumask_var_t          pending_mask;
#endif
#ifdef CONFIG_IRQ_THREADS
    unsigned long          threads_oneshot;
    atomic_t               threads_active;
    wait_queue_head_t      wait_for_threads;
#endif
#ifdef CONFIG_PROC_FS
    struct proc_dir_entry  *dir;
#endif
    const char             *name;
} ____cacheline_internodealigned_in_smp;

```

其中一些重要的成员有:

`struct irq_data irq_data`

主要用来保存软件中断号 `irq` 和 `chip` 相关的数据:

```
<include/linux/irqdesc.h>
```

```
struct irq_data {
    unsigned int    irq;
    unsigned int    node;
    unsigned int    state_use_accessors;
    struct irq_chip *chip;
    void            *handler_data;
    void            *chip_data;
    struct msi_desc *msi_desc;
#ifdef CONFIG_SMP
    cpumask_var_t   affinity;
#endif
};
```

成员 `irq` 代表软件中断号。`chip` 成员则代表着当前中断来自的 PIC，`chip` 所在的数据结构是在软件层面对 PIC 的一个抽象。Linux 通过封装在 `struct irq_data` 中的 `chip` 来屏蔽各种不同硬件平台上 PIC 的差异，给上层的软件提供统一的对 PIC 操作的接口。利用 PIC 中封装的函数，可以屏蔽或启用当前中断，设定外部设备中断触发电信号的类型等。

```
unsigned int __percpu    *kstat_irqs
```

一个 per-CPU 型成员，用于系统的中断统计计数。

```
irq_flow_handler_t  handle_irq
```

这是个函数指针，一般用来指向一个跟当前设备中断触发电信号类型相关的函数。比如，如果外部设备的中断电信号是边沿触发，那么此处 `handle_irq` 将指向一个边沿触发类的处理函数；如果是水平触发，那么将指向一个水平触发类的处理函数。如果在某一平台上边沿触发的中断和水平触发的中断处理起来完全相同，那么就没有必要如此细分，提供一个常规的处理函数就可以了。在 `handle_irq` 指向的函数内部，才会调用设备特定的中断服务例程。特定平台的 Linux 系统在初始化阶段会提供 `handle_irq` 的具体实现，这是内核设计者或者嵌入式平台 BSP 模块⁵所承担的任务，设备驱动程序员在这一层面通常没有什么工作要做。

```
struct irqaction *action
```

`action` 是针对某一具体设备的中断处理的抽象。设备驱动程序会通过 `request_irq` 来向其中挂载设备特定的中断处理函数，相对于前面提到的通用中断处理函数，本书称 `action` 中的 `handler` 为设备中断服务例程 ISR，在下一节中将描述其具体用法。通过前面讨论的

⁵ Linux 嵌入式平台上的 BSP（Board Support Package）部分通常包含了内核代码初始化、配置以及设备驱动程序，本书提到 BSP，一般是指平台的初始化及配置部分，而将设备驱动程序从传统意义上的 BSP 概念中独立出来。

`handle_irq` 和 `action`，可以看到，从通用中断处理函数发起的对某一中断处理，实际上又被划分成了两个层次，第一层是 `handle_irq` 函数，它与软件中断号 `irq` 一一对应，代表了对 IRQ line 上的处理动作，而 `action` 则代表着与具体设备相关的中断处理，也是设备驱动程序要直接与之打交道的对象，通过 `action` 成员，可以在一条 IRQ line 上挂载多个设备，换句话说多个设备可以通过同一条 IRQ line 来共享同一个软件中断号 `irq`，形成所谓的中断链，所以可以推想到 `action` 中必然有构成链表的成员。图 5-3 展示了 `handle_irq` 与 `action` 的层次关系：

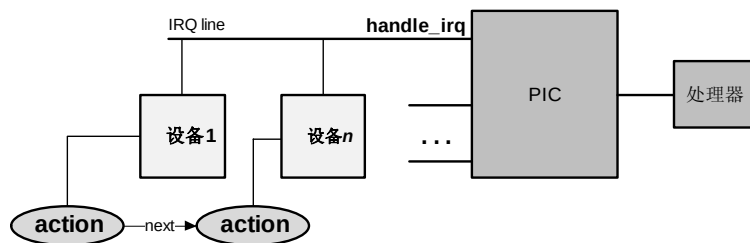


图 5-3 `handle_irq` 与 `action` 之间的层次关系

`unsigned int` `istate`

实际的内核源码中使用的是 `istate` 的宏定义 `core_internal_state__do_not_mess_with_it`，该成员取代了较早版本中的 `status` 成员，主要用于当前中断线 IRQ line 上的状态管理，由一组状态位掩码构成，比如 `IRQS_ONESHOT`、`IRQS_WAITING` 和 `IRQS_PENDING` 等。

`raw_spinlock_t` `lock`

操作 `irq_desc` 数组时用做互斥保护的成员，因为 `irq_desc` 在多个处理器之间共享，即便是单处理器系统，也有并发操作该数组的可能。

`const char` `*name`

`handle_irq` 所对应的名称，最终显示在 `/proc/interrupts` 文件中。

通过上面的讨论，为使读者对 Linux 下的中断处理流程有个全局性的直观印象，这里给出图 5-4。

从图中可以看到，Linux 内核将中断的处理分成了两大部分，分别是 `HARDIRQ` 和 `SOFTIRQ`，前者一般是在处理器屏蔽外部中断的情况下工作，而后者在工作前会启用处理器响应外部中断的能力。通用中断处理函数是外部设备的中断到达处理器后，处理器首先进入的函数，在完成必要的工作后，调用 `do_IRQ` 来对中断进行实际的处理。后者通过引发本次中断的软件中断号来索引 `irq_desc` 数组，找到对应的处理函数并调用，而设备驱动程序等内核模块则通过修改 `irq_desc` 数组中对应项的 `action` 成员来达到安装或卸载设备中断处理服务例程 `ISR` 的目的。设备的中断处理函数调用结束后，中断流程进入 `SOFTIRQ` 部分，在这里

如果有等待的 softirq 需要处理，则处理之，否则返回到通用中断处理函数。

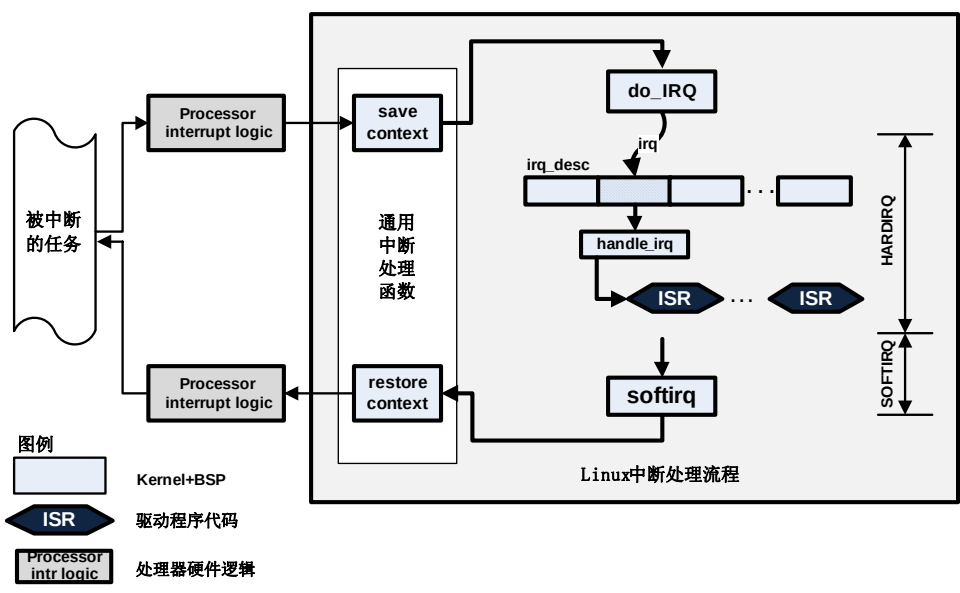


图 5-4 Linux 中断处理流程

5.5 struct irq_chip

数据结构 struct irq_data 中的 struct irq_chip *chip 成员用来表示一个 PIC 的对象，如果系统中只有一个 PIC，那么 irq_desc 数组的每一项中的 chip 都应该指向该 PIC 的对象。平台的初始化函数负责实现该平台使用的 PIC 的对象并将其安装到 irq_desc 数组中。PIC 对象用来实现对 PIC 的配置，配置工作主要包括设定外部设备的中断触发信号的类型，屏蔽或者启用某一设备的中断信号，向发出中断请求的设备发送中断响应信号等。struct irq_chip 定义如下：

```
<include/linux/irq.h>
-----
struct irq_chip {
    const char *name;
    unsigned int(*irq_startup)(struct irq_data *data);
    void (*irq_shutdown)(struct irq_data *data);
    void (*irq_enable)(struct irq_data *data);
    void (*irq_disable)(struct irq_data *data);

    void (*irq_ack)(struct irq_data *data);
    void (*irq_mask)(struct irq_data *data);
    void (*irq_mask_ack)(struct irq_data *data);
    void (*irq_unmask)(struct irq_data *data);
    void (*irq_eoi)(struct irq_data *data);
};
```

```

int      (*irq_set_affinity)(struct irq_data *data, const struct cpumask *dest, bool force);
int      (*irq_retrigger)(struct irq_data *data);
int      (*irq_set_type)(struct irq_data *data, unsigned int flow_type);
int      (*irq_set_wake)(struct irq_data *data, unsigned int on);

void      (*irq_bus_lock)(struct irq_data *data);
void      (*irq_bus_sync_unlock)(struct irq_data *data);

void      (*irq_cpu_online)(struct irq_data *data);
void      (*irq_cpu_offline)(struct irq_data *data);

void      (*irq_print_chip)(struct irq_data *data, struct seq_file *p);

unsigned long    flags;

/* Currently used only by UML, might disappear one day.*/
#ifdef CONFIG_IRQ_RELEASE_METHOD
void      (*release)(unsigned int irq, void *dev_id);
#endif
};

```

其成员绝大多数是函数指针，用来指向具体平台实现的 PIC 控制函数。

5.6 struct irqaction

在继续下面的讨论前，有必要了解 struct irqaction 这个重要的数据结构。在 struct irq_desc 结构中，成员变量 action 是一指向 struct irqaction 类型的指针，设备驱动程序通过这个结构将其中断处理函数挂载在 action 上。以下是该数据结构的定义：

```

<include/linux/interrupt.h>
-----
struct irqaction {
    irq_handler_t handler;
    unsigned long flags;
    void *dev_id;
    struct irqaction *next;
    int irq;
    irq_handler_t thread_fn;
    struct task_struct *thread;
    unsigned long thread_flags;
    unsigned long thread_mask;
    const char *name;
    struct proc_dir_entry *dir;
} ____cacheline_internodealigned_in_smp;

```

其中：

`irq_handler_t handler`

指向设备特定的中断服务例程函数的指针，`irq_handler_t` 的声明如下：

```
<include/linux/interrupt.h>
-----
typedef irqreturn_t (*irq_handler_t)(int, void *);
```

设备驱动程序负责实现该函数，然后调用 `request_irq` 函数，后者会把驱动程序实现的中断服务例程赋值给 `handler`。

`void *dev_id`

调用 `handler` 时传给它的参数，在多个设备共享一个 `irq` 的情况下特别重要，这种链式的 `action` 中，设备驱动程序通过 `dev_id` 来标识自己。

`struct irqaction *next`

指向下一个 `action` 对象，用于多个设备共享同一个 `irq` 的情形，此时 `action` 通过 `next` 构成一个链表。

`struct proc_dir_entry *dir`

中断处理函数中用来创建在 `proc` 文件系统中的目录项。

`irq_handler_t thread_fn`、`struct task_struct *thread` 和 `unsigned long thread_flags`

当驱动程序调用 `request_threaded_irq` 函数来安装中断处理例程时，用来实现 `irq_thread` 机制。

5.7 irq_set_handler

现在把讨论的焦点集中到 `irq_desc` 数组中被软件中断号 `irq` 索引的某一项 `irq_desc[irq]`，对于一个特定的 `irq_desc[irq]`，其上的中断处理分为两级，第一级是调用 `irq_desc[irq].handle_irq`，第二级是设备特定的中断处理例程 `ISR`，在 `handle_irq` 的内部通过 `irq_desc[irq].action->handler` 调用。第一级函数在平台初始化期间被安装到 `irq_desc` 数组中，第二级函数的注册发生在设备驱动程序调用 `request_irq` 安装对应设备的中断处理例程时。第一级函数主要面向 `PIC` 的某一中断线 `IRQ line`，第二级函数则面向该中断线上连接的具体设备，正如我们在前面图 5-3 中看到的那样。内核通过这种两级操作的方式除了可以增加设计的灵活性外，也可以获得某些额外的好处，比如后面将看到的设备软件中断号的探测机制等。

从上一节的讨论可知，`irq_desc[irq].handle_irq` 会被 `do_IRQ` 调用到，在 Linux 源码中 `handle_irq` 的类型声明如下：

```
<include/linux/irq.h>
```

```
typedef void (*irq_flow_handler_t)(unsigned int irq, struct irq_desc *desc);
```

为了让平台的初始化代码能够通过 `handle_irq` 注册第一级中断处理函数，内核提供了两个接口函数：`irq_set_handler` 和 `irq_set_chained_handler`。

```
<include/linux/irq.h>
```

```
static inline void irq_set_handler(unsigned int irq, irq_flow_handler_t handle)
```

```
{
    __irq_set_handler(irq, handle, 0, NULL);
}
```

```
static inline void irq_set_chained_handler(unsigned int irq, irq_flow_handler_t handle)
```

```
{
    __irq_set_handler(irq, handle, 1, NULL);
}
```

参数 `handle` 就是要安装在 `irq_desc[irq].handle_irq` 上的第一级处理函数，最终的安装任务通过 `__irq_set_handler` 来完成。其原型如下：

```
<kernel/irq/chip.c>
```

```
void __irq_set_handler(unsigned int irq, irq_flow_handler_t handle, int is_chained,
                      const char *name)
```

`__irq_set_handler` 在对传递进来的参数作一些必要的检查后，将 `handle` 安装到 `irq_desc[irq]` 上：

```
irq_desc[irq].handle_irq = handle;
irq_desc[irq].name = name;
```

参数 `is_chained` 用来表示 `irq_desc[irq]` 对应的项是否支持中断共享，如果是则将 `irq_desc[irq].status_use_accessors` 作如下设置：

```
desc->status_use_accessors |= _IRQ_NOPROBE | _IRQ_NOREQUEST;
```

`_IRQ_NOREQUEST` 意味着对于 `irq_desc[irq]` 而言，无法通过 `request_irq` 来安装中断处理例程。`_IRQ_NOPROBE` 意味着无法对 `irq_desc[irq]` 执行中断号的探测机制。因此若 `irq_desc[irq]` 对应的项支持中断的共享，那么它将不能支持自动探测中断号，这是由自动探测机制的设计原理所决定的，后面会看到这一点。

作为 `handle_irq` 的一个具体实现例子，下面来看一个用来处理边沿中断触发信号的函数 `handle_edge_irq` 的主要实现代码：

<kernel/irq/chip.c>

```
void handle_edge_irq(unsigned int irq, struct irq_desc *desc)
{
    raw_spin_lock(&desc->lock);
    desc->istate &= ~(IRQS_REPLAY | IRQS_WAITING);
    /*
     * If we're currently running this IRQ, or its disabled,
     * we shouldn't process the IRQ. Mark it pending, handle
     * the necessary masking and go out
     */
    if (unlikely(!irqd_irq_disabled(&desc->irq_data) ||
                irqd_irq_inprogress(&desc->irq_data) || !desc->action)) {
        if (!irq_check_poll(desc)) {
            desc->istate |= IRQS_PENDING;
            mask_ack_irq(desc);
            goto out_unlock;
        }
    }
    kstat_incr_irqs_this_cpu(irq, desc);

    /* Start handling the irq */
    desc->irq_data.chip->irq_ack(&desc->irq_data);

    do {
        if (unlikely(!desc->action)) {
            mask_irq(desc);
            goto out_unlock;
        }

        /*
         * When another irq arrived while we were handling
         * one, we could have masked the irq.
         * Renable it, if it was not disabled in meantime.
         */
        if (unlikely(desc->istate & IRQS_PENDING)) {
            if (!irqd_irq_disabled(&desc->irq_data) &&
                irqd_irq_masked(&desc->irq_data))
                unmask_irq(desc);
        }

        handle_irq_event(desc);

    } while ((desc->istate & IRQS_PENDING) &&
            !irqd_irq_disabled(&desc->irq_data));

    out_unlock;
```

```

raw_spin_unlock(&desc->lock);
}

```

desc->istate &= ~(IRQS_REPLAY IRQS_WAITING)清除掉 IRQ_REPLAY 和 IRQ_WAITING 位, 用来实现设备软件中断号的自动探测机制, 稍后有专门的小节讨论如何自动探测中断号。

函数首先检查 desc->irq_data 中的 state_use_accessors 成员, 确定其 IRQD_IRQ_DISABLED 或 IRQD_IRQ_INPROGRESS 位有没有被置 1, 这两位中的任一位被置 1 或者 desc->action 为空, handle_edge_irq 函数都需要做进一步的特殊处理。IRQD_IRQ_DISABLED 表示当前的 desc 指向一个被禁止的中断线 IRQ line, IRQD_IRQ_INPROGRESS 表示当前的中断线正在处理中, 同一中断 irq 的嵌套或者共享会出现该情况。desc->action 为空表示当前中断线上尚没有被安装特定的设备的中断 ISR。从设备驱动程序员的角度来看, 这三种情况出现的概率较小, if 条件中的 unlikely 也可说明这一点。这里的特殊处理是:

如果当前的中断线不处在正被轮询的阶段 (IRQS_POLL_INPROGRESS), handle_edge_irq 需要将 desc->istate 的 IRQS_PENDING 位置 1, 同时调用 mask_ack_irq(desc)利用 PIC 对象的 irq_mask 例程将该条中断线在 PIC 中屏蔽掉, 然后将 IRQD_IRQ_MASKED 位置 1。这样的处理其实很好理解: 对一个正在被处理 (因此没有必要作进一步处理), 或者被 disabled (当前的触发信号是非预期的, 很可能是一种人为或者硬件线路的故障导致的“假”中断信号), 或者压根儿没有安装设备中断处理例程 ISR (没有设备在使用这根中断线), 对于这样的中断线来说, 这条正在触发中断信号的 IRQ line 都应该被屏蔽掉, 当然为了后续的跟踪处理, IRQS_PENDING 和 IRQD_IRQ_MASKED 位需要置 1。如果当前中断线正在被轮询, 那么需要根据轮询的结果决定下一步的处理。

kstat_incr_irqs_this_cpu 用来更新与中断相关的一些统计量, 比如统计某一 CPU 上中断发生的次数。

经过上述这些步骤之后, 可以正式进入下一阶段对该中断信号进行处理。handle_edge_irq 首先调用 desc->irq_data.chip->irq_ack(&desc->irq_data)函数, 利用 PIC 对象的 irq_ack 例程向设备发出一个中断响应信号, 从硬件逻辑角度, 这一步通常使得当前发出中断信号的设备中产生一个信号电平的转换, 防止设备在它的中断已经在设备驱动程序中处理时依然不停地发出同一中断信号。

do while 循环是 handle_edge_irq 函数的核心部分, 通过调用 handle_irq_event 来对本次中断进行实际的处理操作。do while 中首先对当前 irq 对应的 desc->action 指针进行判断, 如果 action 是个空指针表明到目前为止还没有设备驱动程序在这条中断线上安装中断处理例程 ISR, 对于这种情况的处理是调用 mask_irq 函数通过 PIC 对象的 irq_mask 例程来屏蔽掉当前中断线在 PIC 中对应的中断位, 同时将 desc->irq_data.state_use_accessors 的 IRQD_IRQ_MASKED 位置 1, 这样做是合理的, 对于一个没有安装中断处理函数的外部中断, 应该屏蔽掉它直到它的处理函数被安装上, 否则该设备将不停地中断处理器。之后再

次对 `desc->istate` 进行检查，如果发现有等待的中断信号出现而且是被屏蔽掉的，同时其所对应的中断线又没有被 `disable` 掉，则通过 PIC 的 `unmask` 函数取消对应设备的屏蔽位，这主要是针对中断处理例程在执行过程中又产生了新的中断这种情况，对于第二次出现的中断信号，`handle_edge_irq` 做的处理是将 `desc->istate` 上的 `IRQS_PENDING` 位和 `desc->irq_data.state_use_accessors` 上的 `IRQD_IRQ_MASKED` 位置 1，同时在 PIC 中将对应的中断线屏蔽掉。这样，当前的中断处理例程结束后 `while` 循环条件满足，重新执行 `do while`，在接下来的新循环中，这个处于 `IRQS_PENDING` 状态的中断线在 PIC 中的屏蔽将被解除，`IRQD_IRQ_MASKED` 位也被清除掉。

5.8 handle_irq_event

函数 `handle_irq_event` 的工作比较简单，它为调用设备驱动程序安装的中断处理例程做最后的准备工作，比较容易急躁的读者此时也许还要耐心一点，不过我们很快就会看到与具体的设备中断处理例程相关的调用。实现如下：

<kernel/irq/handle.c>

```
irqreturn_t handle_irq_event(struct irq_desc *desc)
{
    struct irqaction *action = desc->action;
    irqreturn_t ret;

    desc->istate &= ~IRQS_PENDING;
    irqd_set(&desc->irq_data, IRQD_IRQ_INPROGRESS);
    raw_spin_unlock(&desc->lock);

    ret = handle_irq_event_percpu(desc, action);

    raw_spin_lock(&desc->lock);
    irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);
    return ret;
}
```

在进入正式的设备中断处理例程之前，通过 `desc->istate &= ~IRQS_PENDING` 语句清除掉 `IRQS_PENDING` 位，因为紧接下来就会调用设备的中断处理例程 `ISR`，所以 `IRQS_PENDING` 不应再置 1，同时需要将当前的中断线设置 `IRQD_IRQ_INPROGRESS` 状态，表明该中断线上一个中断正在被处理。真正的设备驱动程序实现的中断处理函数例程的调用发生在 `handle_irq_event_percpu` 函数中，后者在源码中的实现为：

<kernel/irq/handle.c>

```
irqreturn_t
handle_irq_event_percpu(struct irq_desc *desc, struct irqaction *action)
```

```

{
    irqreturn_t retval = IRQ_NONE;
    unsigned int random = 0, irq = desc->irq_data.irq;

    do {
        irqreturn_t res;

        trace_irq_handler_entry(irq, action);
        res = action->handler(irq, action->dev_id);
        trace_irq_handler_exit(irq, action, res);

        if (WARN_ONCE(!irqs_disabled(), "irq %u handler %pF enabled interrupts\n",
                    irq, action->handler))
            local_irq_disable();

        switch (res) {
        case IRQ_WAKE_THREAD:
            /*
             * Set result to handled so the spurious check
             * does not trigger.
             */
            res = IRQ_HANDLED;

            /*
             * Catch drivers which return WAKE_THREAD but
             * did not set up a thread function
             */
            if (unlikely(!action->thread_fn)) {
                warn_no_thread(irq, action);
                break;
            }

            irq_wake_thread(desc, action);

            /* Fall through to add to randomness */
        case IRQ_HANDLED:
            random |= action->flags;
            break;

        default:
            break;
        }

        retval |= res;
        action = action->next;
    } while (action);
}

```



```

    if (random & IRQF_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);

    if (!noirqdebug)
        note_interrupt(irq, desc, retval);
    return retval;
}

```

函数的主体是一 do while 循环，用于遍历 action 可能形成的链表结构，当然大部分情况下，一个中断线只安装了一个设备中断处理例程，此时 action 对象并不构成链表，但是从代码中可以清楚地看到内核对同一中断线上多个设备共享中断的支持。循环的一开始就通过 action->handler 来调用具体设备的中断处理例程（action 对象中的 handler 由设备驱动程序通过 request_irq 函数进行安装，Linux 下的设备驱动程序员对此应该不会陌生）。函数接下来对 action->handler 调用的返回值进行处理，驱动程序中实现的中断处理例程函数绝大部分返回值 IRQ_HANDLED，返回 IRQ_WAKE_THREAD 的情形相对比较少，如果返回的是 IRQ_WAKE_THREAD，那么函数将调用 irq_wake_thread 来唤醒 action->thread 表示的一个内核线程，关于这种情形将在后续的中断安装部分予以讨论。在结束一个具体设备的中断处理例程之后，函数通过 action = action->next 来获得 action 的下一个节点（如果节点存在的话）。

do while 的循环条件是 action->next 不为空，这种情况表明正在处理一个共享的中断。对共享中断形成的链式结构的处理是遍历 action 链表，对每一个节点调用其上的 handler 函数。

5.9 request_irq

前面讲了 Linux 下处理一个外部中断的整个流程，其中大部分的工作都是由内核来完成，这里之所以用一定量的篇幅对其进行讨论，目的是希望读者对设备驱动程序提供的中断处理例程被调用时的上下文背景有个清晰的认识，这样我们才能知道如何去实现一个无安全隐患的中断处理例程。现在开始讨论驱动程序如何与 Linux 的中断处理框架进行交互，向 irq_desc 数组中安装设备的中断处理例程。驱动程序中安装一个设备中断服务例程是通过调用 request_irq 函数完成的，其定义如下：

```

<include/linux/interrupt.h>
-----
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char *name, void *dev)
{
    return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}

```

函数的第一个参数 `irq` 是当前要安装的中断处理例程所对应的软中断号；`handler` 就是已经多次提及的中断处理例程 `ISR`，由设备驱动程序负责实现；`flags` 是标志变量，可影响内核在安装 `ISR` 时的一些行为模式；`name` 是当前安装中断 `ISR` 的设备名称，内核会在 `proc` 文件系统中生成 `name` 的一个入口点；`dev` 是个传递到中断处理例程的指针，在中断共享的情形下，将在 `free_irq` 时被用到，以区分当前的 `free_irq` 要释放的是哪一个 `struct irqaction` 对象，因此必须确保 `dev` 参数在内核整个中断处理框架中的唯一性，由于内核在用 `request_irq` 安装一个中断处理例程时并不对 `dev` 的唯一性进行检查，因此设备驱动程序应该努力做到这一点，通常的做法是将设备驱动程序所管理的与设备相关的某一数据结构对象的指针作为 `dev` 的实参。另外，由于内核中断处理框架在调用设备驱动程序的 `ISR` 时，会将该 `dev` 参数一并传入，因此也可以借助它在被中断的进程与中断处理例程中传递数据之用。

`request_irq` 函数的核心是通过调用 `request_threaded_irq` 完成中断处理函数的实际安装工作，可以看到 `request_irq` 在调用 `request_threaded_irq` 函数时传入的第三个参数是 `NULL`，这个参数跟内核中一个用于中断处理的线程 `irq_thread` 有关，如果设备驱动程序通过 `request_irq` 来安装一个中断处理例程，因为对 `thread_fn` 传入的实参是 `NULL`，所以不会涉及 `irq_thread` 部分，但是设备驱动程序也可以直接通过调用 `request_threaded_irq` 来安装中断，此时就有机会使用到 `irq_thread` 机制。

`request_threaded_irq` 函数的实现源码为：

<kernel/irq/manage.c>

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                        irq_handler_t thread_fn, unsigned long irqflags,
                        const char *devname, void *dev_id)
{
    struct irqaction *action;
    struct irq_desc *desc;
    int retval;

    if ((irqflags & IRQF_SHARED) && !dev_id)
        return -EINVAL;

    desc = irq_to_desc(irq);
    if (!desc)
        return -EINVAL;

    if (!irq_settings_can_request(desc))
        return -EINVAL;

    if (!handler) {
        if (!thread_fn)
            return -EINVAL;
        handler = irq_default_primary_handler;
    }
```

```

    }

    action = kzalloc(sizeof(struct irqaction), GFP_KERNEL);
    if (!action)
        return -ENOMEM;

    action->handler = handler;
    action->thread_fn = thread_fn;
    action->flags = irqflags;
    action->name = devname;
    action->dev_id = dev_id;

    chip_bus_lock(irq, desc);
    retval = __setup_irq(irq, desc, action);
    chip_bus_sync_unlock(irq, desc);

    if (retval)
        kfree(action);

    return retval;
}

```

函数一开始进行了一系列的检查。比如，如果 `irqflags` 中的 `IRQF_SHARED` 位被置 1，表明正在安装一个共享的中断，这种情况下驱动程序必须提供 `dev_id`，如果 `dev_id` 为空则是非法情况，因为在 `free_irq` 中将无法确定到底卸载哪一个 `action`。如果 `desc->status_use_accessors` 上的 `_IRQ_NOREQUEST` 位被置 1，表明 `irq_desc` 数组中的这一项禁止通过 `request_threaded_irq` 来安装中断处理函数，也是非法情况。

这些检查通过之后，函数调用 `kzalloc` 分配一块类型为 `struct irqaction` 的地址空间 `action`，然后根据函数传入的参数初始化 `action`，并调用 `__setup_irq` 来安装中断处理函数。`__setup_irq` 的声明如下：

```

static int
__setup_irq(unsigned int irq, struct irq_desc *desc, struct irqaction *new);

```

因为中断安装时有诸多的细节需要内核仔细处理，所以 `__setup_irq` 函数的源码实现看起来比较长，但其实质性的工作其实就是将 `desc` 中的 `action` 成员指针指向要安装的中断处理例程。

下面按照 `request_irq` 调用时 `desc->action` 是否为空分别进行讨论（先暂不考虑 `irq_thread` 机制）。

○ `desc->action` 为空

这种情况比较简单，因为此时 `desc->action` 为空，意味着当前尚无设备驱动程序正在使用这条中断线，所以只需先获得指向 `desc->action` 的指针 `old_ptr`：`struct irqaction **old_ptr = &desc->action`，然后将 `request_threaded_irq` 中新分配的 `action` 指针赋值给 `old_ptr` 即可：

`*old_ptr = new;`。

从设备驱动程序的角度而言，有几个需要注意的地方，如果设备驱动调用 `request_irq` 时，参数 `flags` 中设定了 `IRQF_TRIGGER_MASK` 标志位，表明驱动程序需要利用 `request_irq` 对 `irq` 的触发类型进行配置，因为 `desc->irq_data` 中的 `chip` 是 PIC 的抽象，所以此时只需调用 `chip` 中的 `irq_set_type` 成员函数就可配置 PIC。系统定义的中断信号触发类型标志有：

```
<include/linux/interrupt.h>
-----
//上升沿触发
#define IRQF_TRIGGER_RISING      0x00000001
//下降沿触发
#define IRQF_TRIGGER_FALLING    0x00000002
//高电平触发
#define IRQF_TRIGGER_HIGH       0x00000004
//低电平触发
#define IRQF_TRIGGER_LOW        0x00000008
//中断触发信号掩码
#define IRQF_TRIGGER_MASK      (IRQF_TRIGGER_HIGH | IRQF_TRIGGER_LOW | \
                                IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING)
```

设定中断触发信号类型的函数为 `__irq_set_trigger`，其主要功能是通过 PIC 对象的 `irq_set_type` 成员函数设定当前中断线上有效的中断触发信号类型，同时将设定的类型记录到 `desc->irq_data.state_use_accessors` 中：

```
<kernel/irq/manage.c>
-----
int __irq_set_trigger(struct irq_desc *desc, unsigned int irq,
                    unsigned long flags)
{
    struct irq_chip *chip = desc->irq_data.chip;
    int ret, unmask = 0;
    ...
    ret = chip->irq_set_type(&desc->irq_data, flags);
    ...
    desc->irq_data.state_use_accessors &= IRQD_TRIGGER_MASK;
    desc->irq_data.state_use_accessors |= flags;
    ...
}
```

因为不是共享中断的情形，所以当前的 `request_irq` 调用将独占 `irq` 所对应的中断线的所有权，可以根据设备自身需要随意设置其中断触发信号类型，这在存在中断共享的情形下是不可能的。

所以如果驱动程序需要将 `irq` 的触发信号配置成下降沿触发，可以作如下调用：

```
request_irq(irq, demo_handler, IRQF_TRIGGER_FALLING, NULL, NULL);
```

○ desc->action 不为空

这种情形表明当前 irq 所对应的中断线此前已经被安装了中断处理函数，换言之，这意味着正在安装一个共享该 irq 的中断处理例程。在中断共享的情况下，事情变得有些复杂，因为在此之前至少有一个设备驱动程序在当前的中断线上安装了中断处理例程，此时内核再安装一个新的中断处理例程就有了相当的限制，一个大体的原则是，新的安装不能破坏之前已有的中断工作模式。从代码的角度，新的 irqaction 对象的 flags 成员必须与 action 链上已有的节点的 flags 成员作检查比较，如果有不一致的情形出现，安装将不会成功，函数返回一个错误码 -EBUSY。被检查的 flags 标志有 IRQF_SHARED、IRQF_TRIGGER_MASK、IRQF_ONESHOT 及 IRQF_PERCPU，这些都是设备驱动程序在调用 request_irq 时通过参数 flags 传入的标志位。

在共享中断的情形下，如果新的 request_irq 调用去设定当前的触发信号的类型，__setup_irq 函数并不会去真正调用 PIC 对象的 irq_set_type 函数，而只是检查当前要设定的中断触发信号类型是否与这条线上已经设定的类型相符，如果不符合，__setup_irq 会给出一个警告信息，当前的安装虽然可以成功，但是未必能如预期的那样正常工作。

如果这些检查都成功通过，那么 request_irq 此时要做的是，将新分配的 action 加到 action 链的末尾。

在 __setup_irq 函数的结束部分，如果 desc->dir 还是空，那么调用 register_irq_proc 在 /proc/irq 目录下创建类似 /proc/irq/125 这样的新目录项。最后调用的 register_handler_proc 在 action->name 不为空的情况下，会为此新 action 在 proc 文件系统中创建类似 /proc/irq/125/action_name 这样的目录。内核通过这些 proc 文件系统的操作，可以方便开发者在用户空间查看系统中设备驱动程序的中断安装情况，例如 x86 平台上对应 irq=45 的 proc 文件系统节点的下列输出：

```
root@AMDLinuxFGL:/proc/irq/45# ll
total 0
dr-xr-xr-x  3  root root 0   Aug  6   17:48.
dr-xr-xr-x 27  root root 0   Aug  6   17:44..
-r-----   1  root root 0   Aug  6   17:48affinity_hint
dr-xr-xr-x  2  root root 0   Aug  6   17:48hda_intel
-r--r--r--   1  root root 0   Aug  6   17:48node
-rw-----   1  root root 0   Aug  6   17:48smp_affinity
-r--r--r--   1  root root 0   Aug  6   17:48spurious
```

5.10 中断处理的 irq_thread 机制

下面再简单讨论一下内核为中断处理提供的另一种机制，这种机制在设备驱动程序通过调

用 `request_threaded_irq` 函数来安装一个中断时，需要在 `struct irqaction` 对象中实现它的 `thread_fn` 成员。`request_threaded_irq` 函数内部会生成一个名为 `irq_thread` 的独立线程：

```
<kernel/irq/manage.c>
-----
static int
__setup_irq(unsigned int irq, struct irq_desc *desc, struct irqaction *new)
{
    ...
    if (new->thread_fn && !nested) {
        struct task_struct *t;

        t = kthread_create(irq_thread, new, "irq/%d-%s", irq,
                           new->name);
        if (IS_ERR(t))
            return PTR_ERR(t);
        /*
         * We keep the reference to the task struct even if
         * the thread dies to avoid that the interrupt code
         * references an already freed task_struct.
         */
        get_task_struct(t);
        new->thread = t;
    }
    ...
}
```

`irq_thread` 线程被创建出来时将以 `TASK_INTERRUPTIBLE` 的状态睡眠等待中断的发生，当中断发生时 `action->handler` 只负责唤醒睡眠的 `irq_thread`，后者将调用 `action->thread_fn` 进行实际的中断处理工作。因为 `irq_thread` 本质上是系统中的一个独立进程，所以采用这种机制将使实质的中断处理工作发生在进程空间，而不是中断的上下文中。

5.11 free_irq

通过 `request_irq` 安装的中断处理函数，如果不再需要的话应该调用 `free_irq` 予以释放。`free_irq` 完成的任务和 `request_irq` 正好相反，其声明如下：

```
<include/linux/interrupt.h>
-----
extern void free_irq(unsigned int irq, void * dev_id);
```

根据第一个参数 `irq`，函数在 `irq_desc` 数组中查找对应的 `action`，遍历该 `action` 所在的链表，如果有 `action->dev_id == dev_id`，那么就找到了要释放的 `action`。找到后调用 `kfree` 释放 `action` 所占的空间。如果释放的 `action` 是 `irq_desc[irq]` 中唯一的一个 `action` 节点，那么释放后还需要把 `desc->irq_data.state_use_accessors` 的 `IRQD_DISABLED` 位置 1，同时调用 `irq_desc[irq].chip` 的 `irq_shutdown` 或者 `irq_disable/irq_mask` 函数在 PIC 中屏蔽掉 `irq` 所对应

的外部设备中断线。request_irq 中建立的 proc 文件系统节点也将被删除。

5.12 SOFTIRQ

前面讨论了 HARDIRQ 的执行流程，下面再来看看 Linux 内核如何实现 SOFTIRQ。SOFTIRQ 的处理是在 do_IRQ 函数的 irq_exit 中实现的，irq_exit 函数中实现 SOFTIRQ 调用的代码为：

```
<kernel/softirq.c>
-----
void irq_exit(void)
{
    ...
    sub_preempt_count(IRQ_EXIT_OFFSET);
    if (!in_interrupt() && local_softirq_pending())
        invoke_softirq();
    ...
}
```

函数首先把当前栈中的 preempt_count 变量减去 IRQ_EXIT_OFFSET 来标识一个 HARDIRQ 中断上下文的结束：preempt_count() -= IRQ_EXIT_OFFSET，这步动作对应 do_IRQ 中的 irq_enter。

在没有配置内核可抢占的系统中，IRQ_EXIT_OFFSET=HARDIRQ_OFFSET；如果配置了可抢占，那么 IRQ_EXIT_OFFSET=(HARDIRQ_OFFSET-1)，意味着在 HARDIRQ 部分结束之后，内核已经启动可抢占性。

invoke_softirq 是真正处理 SOFTIRQ 部分的函数，不过这个函数的调用有个前提，就是 if 中的两个条件：in_interrupt 和 local_softirq_pending。

in_interrupt 是个宏，展开为：

```
<include/linux/hardirq.h>
-----
#define in_interrupt() (preempt_count() & (HARDIRQ_MASK | SOFTIRQ_MASK | NMI_MASK))
```

其主要用意是根据当前栈中的 preempt_count 变量，来判断当前是否在一个中断上下文中执行。根据 in_interrupt 的定义来看，Linux 内核认为 HARDIRQ、SOFTIRQ 和 NMI 都属于 interrupt 范畴。对于 HARDIRQ，前面讨论 do_IRQ 时可以看到在 irq_enter 和 irq_exit 之间，内核在 preempt_count()上标示了 HARDIRQ_OFFSET，表示这是个 HARDIRQ 的上下文。Linux 内核对 preempt_count 的使用如图 5-5 所示：



图 5-5 preempt_count 结构

由图可见, preempt_count 的低 8 位与 PREEMPT 相关, 8~15 位留给 SOFTIRQ 使用, 16~25 位给 HARDIRQ 使用, NMI 占据 1 位。

local_softirq_pending 也是一个宏, 展开为:

```
<include/linux/irq_cpustat.h>
-----
#define local_softirq_pending() (irq_stat[smp_processor_id()].__softirq_pending)
```

irq_stat 是个数组, 其具体定义取决于 __ARCH_IRQ_STAT 宏, 在大部分体系架构中这是个 per-CPU 变量, 比如对于 x86 平台:

```
<arch/x86/include/asm/hardirq.h>
-----
DECLARE_PER_CPU_SHARED_ALIGNED(irq_cpustat_t, irq_stat);
```

如果没有定义 __ARCH_IRQ_STAT, 那么 irq_stat 定义如下:

```
<kernel/softirq.c>
-----
#ifndef __ARCH_IRQ_STAT
irq_cpustat_t irq_stat[NR_CPUS] ____cacheline_aligned;
EXPORT_SYMBOL(irq_stat);
#endif
```

基本上可以认为这是个 per-CPU 变量, 系统中的每个 CPU 都拥有各自的副本。其类型 irq_cpustat_t 定义如下:

```
<include/asm-generic/hardirq.h>
-----
typedef struct {
    unsigned int __softirq_pending;
} ____cacheline_aligned irq_cpustat_t;
```

内核用一个无符号整型 __softirq_pending 来表示当前正在等待被处理的 softirq, 每一种 softirq 在 __softirq_pending 中占据一位, 每个 CPU 都拥有自己的 __softirq_pending 变量。

回到 irq_exit, 现在知道 invoke_softirq 被调用的前提是: 当前不在 interrupt 上下文中而且 __softirq_pending 中有等待的 softirq。当前不在 interrupt 上下文中保证了如果代码正在 SOFTIRQ 部分执行时 (此时处理器可以处理外部中断), 如果发生了一个外部中断, 那么在中断处理函数结束 HARDIRQ 部分时, 将不会处理 softirq, 而是直接返回, 这样此前被中断的 SOFTIRQ 部分将继续执行。

现在开始讨论 softirq 的处理部分, invoke_softirq 是一个宏, 定义如下:

```
<kernel/softirq.c>
-----
#ifdef __ARCH_IRQ_EXIT_IRQS_DISABLED
# define invoke_softirq() __do_softirq()
#else
# define invoke_softirq() do_softirq()
```



```
#endif
```

__ARCH_IRQ_EXIT_IRQS_DISABLED 是个体体系架构相关的宏，用来决定在 HARDIRQ 部分结束时，有没有关闭处理器响应外部中断的能力。如果定义了 __ARCH_IRQ_EXIT_IRQS_DISABLED，就意味着在处理 SOFTIRQ 部分时，可以保证外部中断已经关闭，此时可以直接调用 __do_softirq，否则调用 do_softirq，后者最终会调用到 __do_softirq，不过之前要做一些中断屏蔽的事情，保证 __do_softirq 开始执行时中断是关闭的：

```
<kernel/softirq.c>
```

```
asmlinkage void do_softirq(void)
{
    __u32 pending;
    unsigned long flags;

    if (in_interrupt())
        return;
    local_irq_save(flags);
    pending = local_softirq_pending();
    if (pending)
        __do_softirq();
    local_irq_restore(flags);
}
```

__do_softirq 的核心代码如下：

```
<kernel/softirq.c>
```

```
#define MAX_SOFTIRQ_RESTART 10
asmlinkage void __do_softirq(void)
{
    struct softirq_action *h;
    __u32 pending;
    int max_restart = MAX_SOFTIRQ_RESTART;
    int cpu;

    pending = local_softirq_pending();
    __local_bh_disable((unsigned long)__builtin_return_address(0));
    cpu = smp_processor_id();
restart:
    set_softirq_pending(0);
    local_irq_enable();
    h = softirq_vec;

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

```

        h++;
        pending >>= 1;
    } while (pending);

    local_irq_disable();
    pending = local_softirq_pending();
    if (pending && --max_restart)
        goto restart;

    if (pending)
        wakeup_softirqd();

    _local_bh_enable();
}

```

在具体讨论这个函数之前，先看看系统定义的几个 softirq 的类型：

```

enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    BLOCK_IOPOLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ,
    RCU_SOFTIRQ,
    NR_SOFTIRQS
};

```

每个 softirq 对应 __softirq_pending 中的一个位。其中，HI_SOFTIRQ 和 TASKLET_SOFTIRQ 用来实现 tasklet，TIMER_SOFTIRQ 和 HRTIMER_SOFTIRQ 用于定时器，NET_TX_SOFTIRQ 和 NET_RX_SOFTIRQ 用于网络设备的发送和接收操作，BLOCK_SOFTIRQ 和 BLOCK_IOPOLL_SOFTIRQ 用于块设备的操作，SCHED_SOFTIRQ 用于调度器。

内核在此基础上定义了一个 struct softirq_action 类型的数组 softirq_vec，用来放置 softirq 对应的处理函数：

```

<include/linux/interrupt.h>
-----
struct softirq_action
{
    void (*action)(struct softirq_action *);
};

```

```
<kernel/softirq.c>
```

```
static struct softirq_action softirq_vec[NR_SOFTIRQS] __cacheline_aligned_in_smp;
```

所以__do_softirq 的核心思想是：从 CPU 本地的__softirq_pending 的最低位开始，依次往高位扫描，如果发现某位为 1，说明对应该位有个等待中的 softirq 需要处理，那么就调用 softirq_vec 数组中对应项的 action 函数。这个过程会一直持续下去，直到__softirq_pending 为 0。

具体的函数实现上，有以下几点需要注意：

(1) __local_bh_disable 和 __local_bh_enable 用来在 preempt_count() 上标示 SOFTIRQ 的上下文，考虑到 SOFTIRQ 执行过程可能会被外部中断的情况，这可以防止 SOFTIRQ 部分的重入，因为只有在非 interrupt 的上下文中才可以进入到 SOFTIRQ 部分。

(2) 在执行 softirq 的前后分别调用了 local_irq_enable 和 local_irq_disable，这说明 SOFTIRQ 部分在执行时处理器可以响应并处理外部的中断。

(3) softirq 执行的先后顺序由__softirq_pending 中的位决定，低位的 softirq 要先于高位的 softirq 执行。

(4) 在 do while 循环之后，会再次检测__softirq_pending 是否为 0，这主要是因为 SOFTIRQ 在执行过程中可能被外部设备中断，其设备驱动程序在实现该中断处理函数时可能使用了一个 softirq，因此在 do while 循环之后，需要检测有没有新加入的 softirq 需要处理。

(5) 如果上面第 3 步的动作执行超过一定的次数，则需要唤醒 ksoftirqd 来处理。因为如果在 SOFTIRQ 部分耗费太多的时间，会导致一个中断处理流程迟迟无法结束，这意味着此前被中断的任务无法继续运行。为了避免这种情况，Linux 系统在初始化期间生成了一个新的进程 ksoftirqd，该进程运行时要完成的主要任务就是调用 do_softirq 来执行等待中的 softirq，如果没有 softirq 需要处理，该进程将进入睡眠状态。

因此，为了避免在一个中断的 SOFTIRQ 部分耗费太多时间处理 softirq 导致该中断流程迟迟无法结束，__do_softirq 通过 wakeup_softirqd 唤醒 ksoftirqd，让调度器来平衡当前中断在 SOFTIRQ 部分的工作负荷。

关于 softirq 更进一步的论述，请读者参考本书“延迟操作”一节。

5.13 irq 的自动探测

如果一个设备的驱动程序无法确定它所管理的设备的软件中断号 irq，此时设备驱动程序可以使用 irq 的自动探测机制来获得其正在使用的 irq。内核为此提供了几个接口函数供驱动程序使用，需要注意的是，中断探测机制的实现需要内核和驱动程序共同努力才能完成，并且这种探测只限于非共享中断的情况，因此只有当一个设备能确定其 irq 不会与别的设备

共享时，才可以使用这里的探测。探测前的情形是，该设备关联到了某个 irq，但是因为设备驱动程序还不清楚是哪个 irq，因此不可能调用 request_irq 来向该 irq 安装中断处理例程，所以对应该 irq 的 action 为空。探测要完成的任务是找到该设备所关联的 irq。

探测的原理是，调用 probe_irq_on 函数遍历整个 irq_desc 数组，对于每个 action 为空的元素且在该项允许自动探测的情形下，将其 istate 上的 IRQS_WAITING 位置 1，然后让设备产生一次中断，irq_desc 数组中与该设备 irq 关联的那一项的第一级中断函数 handle_irq 会被调用，后者将会清除 IRQS_WAITING 位，然后调用 probe_irq_off 再遍历一遍 irq_desc 数组，对于每个 action 为空的元素，查看其 istate 上的 IRQS_WAITING 位是否被清 0，如果是，那么该元素对应的 irq 就是正与目前设备关联的。

以下是一个设备驱动程序用来实现自动探测的代码序列的示例：

```
unsigned long irq;
//清除设备内部的中断
...
irqs = probe_irq_on();
/*等待 5ms */
msleep(5);

/*让设备产生一次中断 */
...
/*等待 5ms */
msleep(5);

/* 得到探测到的中断号 */
irq = probe_irq_off(irqs);
```

这段代码中用到了 probe_irq_on 和 probe_irq_off 两个函数，它们都是内核为驱动程序实现的自动探测接口函数，下面将看到这两个函数的实现原理。

probe_irq_on 函数的核心代码是：

```
<kernel/irq/autoprobe.c>
-----
unsigned long probe_irq_on(void)
{
    struct irq_desc *desc;
    unsigned long mask = 0;
    int i;

    /*
     * quiesce the kernel, or at least the asynchronous portion
     */
    async_synchronize_full();
    mutex_lock(&probing_active);
```

```
/*
 * something may have generated an irq long ago and we want to
 * flush such a longstanding irq before considering it as spurious.
 */
for_each_irq_desc_reverse(i, desc) {
    raw_spin_lock_irq(&desc->lock);
    if (!desc->action && irq_settings_can_probe(desc)) {
        /*
         * Some chips need to know about probing in
         * progress:
         */
        if (desc->irq_data.chip->irq_set_type)
            desc->irq_data.chip->irq_set_type(&desc->irq_data,
                                              IRQ_TYPE_PROBE);

        irq_startup(desc);
    }
    raw_spin_unlock_irq(&desc->lock);
}

/* Wait for longstanding interrupts to trigger. */
}
raw_spin_unlock_irq(&desc->lock);
}

msleep(20);

/*
 * enable any unassigned irqs
 * (we must startup again here because if a longstanding irq
 * happened in the previous stage, it may have masked itself)
 */
for_each_irq_desc_reverse(i, desc) {
    raw_spin_lock_irq(&desc->lock);
    if (!desc->action && irq_settings_can_probe(desc)) {
        desc->istate |= IRQS_AUTODETECT | IRQS_WAITING;
        if (irq_startup(desc))
            desc->istate |= IRQS_PENDING;
    }
    raw_spin_unlock_irq(&desc->lock);
}

}

raw_spin_unlock_irq(&desc->lock);
}

/*
```

```

    * Wait for spurious interrupts to trigger
    */
    msleep(100);

    /*
    * Now filter out any obviously spurious interrupts
    */
    for_each_irq_desc(i, desc) {
        raw_spin_lock_irq(&desc->lock);

        if (desc->istate & IRQS_AUTODETECT) {
            /* It triggered already - consider it spurious. */
            if (!(desc->istate & IRQS_WAITING)) {
                desc->istate &= ~IRQS_AUTODETECT;
                irq_shutdown(desc);
            } else
                if (i < 32)
                    mask |= 1 << i;
        }
        raw_spin_unlock_irq(&desc->lock);
    }
    return mask;
}

```

函数的主体是三个 `for_each_irq_desc` 所引导的循环。第一个 `for_each_irq_desc` 循环从后向前遍历 `irq_desc` 数组，遍历过程中对于每一个 `desc`，只要能满足 `desc->action` 为空并且 `desc->status_use_accessors` 没有设置 `IRQ_NOPROBE` 位，那么就通过 PIC 中的 `irq_startup` 函数把对应的中断启用起来。`desc->action` 为空说明该 `irq` 上还没有安装中断处理例程，`desc->status_use_accessors` 没有设置 `IRQ_NOPROBE` 位说明该 `desc` 允许被探测，设备所关联的 `irq` 只可能存在满足这两个条件的 `desc` 中。

第二个 `for_each_irq_desc` 循环依旧从后向前遍历 `irq_desc` 数组，对于满足 `desc->action` 为空并且 `desc->status_use_accessors` 没有设置 `IRQ_NOPROBE` 位的 `desc`，将其 `istate` 重新设置为：

```
desc->istate |= IRQS_AUTODETECT | IRQS_WAITING;
```

第三个 `for_each_irq_desc` 循环从前向后遍历 `irq_desc` 数组，对于满足 `(desc->istate & IRQS_AUTODETECT) != 0` 的每一个 `desc`，说明它正是我们在探测的元素，此时检查 `desc->istate` 上的 `IRQS_WAITING` 位有没有被置 1。因为根据探测的流程，在调用 `probe_irq_on` 时，驱动程序还没有让设备产生中断，因此 `IRQS_WAITING` 位不可能被清 0，如果它被清 0，说明该 `desc` 上的第一级函数被调用了，这意味着这个 `irq` 所对应的中断线上正在产生无意义的触发信号（不可能是由安装了 ISR 的正常设备所产生，因为 `request_irq` 在安装 ISR 时会清除掉 `IRQS_AUTODETECT` 位），对此的处理是通过 PIC 屏蔽该中断，然

后继续查找下一个 `irq_desc` 元素。

这个函数的返回值如同 `probe_irq_off` 中的参数一样并无实际的用途。

当 `probe_irq_off` 被调用时，驱动程序已经让设备产生了一次中断，所以 `probe_irq_off` 需要使用 `for_each_irq_desc` 循环从前向后遍历 `irq_desc` 数组，试图找到这样一个 `desc`：
(`desc->istate & IRQS_AUTODETECT`) != 0 并且 `desc->istate` 的 `IRQS_WAITING` 位被清除，这正是 `probe_irq_off` 的主要流程。如果找到了设备所关联的 `irq` 就返回之，否则函数返回 0。

5.14 中断处理例程

如果设备需要通过中断这种方式与处理器进行沟通，那么它的驱动程序就有必要实现一个中断处理例程并负责把它安装到系统中，这样当设备的中断信号来临时，处理器才可能调用到它的处理例程。虽然中断处理例程不过是种普通的函数，但是内核作为这种游戏规则的制定者，为了确保一切尽在它的掌握之中，对于中断处理例程的实现有着特定的要求。

首先，从中断处理例程的原型看，它必须与 `struct irqaction` 中 `handler` 函数指针的原型保持一致。这很正常，因为中断处理例程的安装，本质上是让这个指针指向中断处理例程。`handler` 的原型前面提过，这里再重复一遍：

```
typedef irqreturn_t (*irq_handler_t)(int, void *);
```

因此，一个实际的中断处理例程应该这样声明自己：

```
irqreturn_t demo_isr(int irq, void * dev_id);
```

函数的返回值是个 `irqreturn_t` 类型，该类型在内核源码中的定义如下：

```
<include/linux/irqreturn.h>
-----
enum irqreturn {
    IRQ_NONE,
    IRQ_HANDLED,
    IRQ_WAKE_THREAD,
};
typedef enum irqreturn irqreturn_t;
```

因此，中断处理例程只能有三种返回值，分别是：

`IRQ_NONE`

中断例程发现正在处理一个不是自己的设备触发的中断，此时它唯一要做的就是返回该值。

`IRQ_HANDLED`

中断处理例程成功地处理了自己设备的中断，返回该值。

IRQ_WAKE_THREAD

中断处理例程被用来作唤醒一个等待在它的 `irq` 上的一个进程使用，此时它返回该值。

其次，中断处理例程是在中断上下文中执行，这对它的实现提出了某些限制。因为中断上下文不隶属于某个进程，在这里 `current`⁶ 指针不再有意义，它们游离在 Linux 进程世界的边缘，因此在这种环境下绝对禁止任何形式的进程切换。实际的代码实现中，确保中断处理上下文中不出现进程的切换并不是件容易的事，需要仔细审查中断处理例程中的每行代码，包括调用的每一个函数，确保它们不会进入睡眠状态而使调度器介入其中。一个典型的例子是在中断处理例程中使用内存分配函数 `kmalloc`，如果在调用这个函数时使用了 `GFP_KERNEL` 标志而不是 `GFP_ATOMIC`，那么很小的概率下会因为内存难以满足需求而进入睡眠，虽然大部分时间都不会遇到麻烦，但是偶尔出现的睡眠将会带来真正的大麻烦。

最后，中断处理例程作为系统中的一种并发源头，可能会去访问一些共享的资源，如果不幸恰好有别的进程（最典型的是被它中断的进程）也在使用同样的共享资源，竞态将不可避免，因此需要考虑互斥的机制来保护。本书第 4 章讨论了内核所提供的用于互斥的各种机制，在中断处理例程中使用这些机制需要格外小心，防止出现睡眠的可能性，因此信号量和互斥锁首先就会被排除掉，绝大多数的情况你需要使用自旋锁 `spin lock` 及其变体。

5.15 中断共享

即便 PIC 已经提供足够多的中断引脚供外部设备使用，但也有不够用的时候，此时中断共享机制可能就会派上用场。所谓中断共享是指多个设备共享一根中断线，使用同一个 `irq`。对驱动程序来说，需要注意的地方在调用 `request_irq` 和中断处理例程的实现上。对于一个共享的中断，驱动程序在调用 `request_irq` 时应该使用 `IRQF_SHARED` 标志，同时提供 `dev_id`，提供的 `dev_id` 在中断处理例程中并没有什么特别的用处，之所以要求在中断共享时提供这个参数，主要是为了在 `free_irq` 时能在 `action` 链中找到它，因此这个 `dev_id` 在中断共享的 `action` 链中应该具有唯一性，实际使用中可以像下面这样：

```
//pDev 是一个指向设备相关的结构体的指针
struct demo_dev *pDev = ...

//request_irq
int retval = request_irq(irq, demo_isr, IRQF_SHARED, "demo device", pDev);
```

⁶ Linux 内核实现了一个宏，该宏通过对当前堆栈指针 `sp` 的某种操作，使之指向当前进程的 `task_struct` 结构。具体的实现原理属于描述内核的书籍范畴了。中断上下文不是一个 `task_struct` 结构的对象。

接下来是中断共享时的中断处理例程的实现，因为当 `irq` 上的中断发生时，内核会调用 `irq` 上的每个 `action` 中的 `handler`，因此即便不是你的设备产生的中断，你的中断处理例程 `ISR` 也会被调用到，因此共享中断时的 `ISR` 需要能判断是否是自己的设备产生的中断，这主要靠读取自己设备的中断状态寄存器来完成。如果发现你的设备没有产生中断，那么 `ISR` 只需要返回一个 `IRQ_NONE` 就好了，下面是一个共享中断下的 `ISR` 的实现：

```
irqreturn_t demo_isr(int irq, void * dev_id)
{
    //读取设备中断状态寄存器
    status = read_intr_reg(...);
    //判断自己的设备有没有产生中断，没有的话直接返回 IRQ_NONE
    if(status & ...){
        return IRQ_NONE;
    }else{
        //中断处理
    }

    return IRQ_HANDLED
}
```

5.16 本章小结

本章讨论了 Linux 下一个外部中断发生后的整个处理流程。通过内核精心设计的中断处理框架，如果一个设备驱动程序为其所管理的设备通过 `request_irq` 注册了中断处理例程，那么该设备产生的中断在中断处理框架中经过多层的调用，最终会进入到该中断处理例程中来。可以看到，这其中大部分的任务来自于内核的代码（嵌入式系统还需要 `BSP` 代码）。驱动程序对中断的支持相对简单，只需要实现中断处理例程函数并调用 `request_irq` 向系统注册即可，在某些特定的情形下，设备驱动程序也可以使用 `request_threaded_irq` 函数来向系统注册中断处理例程，比如需要使用 `irq_thread` 机制。了解整个中断处理流程，有助于了解中断处理例程的执行环境，避免因为不安全的中断例程实现给系统造成负面影响。

Linux 内核将中断处理分成了 `HARDIRQ` 和 `SOFTIRQ` 两部分。`HARDIRQ` 在执行时中断是关闭的，因此这部分的代码应该完成中断处理中最关键的任务，执行时间也应尽可能短。如果需要执行时间很长的操作，可以将其延迟到 `SOFTIRQ` 部分执行，因为 `SOFTIRQ` 部分在执行时处理器的中断是打开的。

第 6 章

延迟操作

有些时候设备驱动程序可能需要延迟某些操作的进行，典型的情况是在处理设备中断的时候，正如我们在本书第 5 章中断处理中了解到的，Linux 内核将对一个外部设备中断的处理分成两大部分 `HARDIRQ` 和 `SOFTIRQ`，因为 `HARDIRQ` 部分在执行时处理器的中断是关闭的，所以驱动程序的中断处理例程在这部分只应该完成一些关键的中断操作，而将耗时的的工作延迟到 `SOFTIRQ` 部分执行。内核为此给驱动程序提供了一个基于 `SOFTIRQ` 的任务延迟的实现机制 `tasklet`。因为 `tasklet` 需要在中断上下文中执行，所以有些延迟的操作无法用 `tasklet` 来完成，为此内核又提供了一个基于进程的延迟操作实现机制——工作队列 `workqueue`。

本章将先描述 `tasklet` 和工作队列的内核实现机制，然后再分别讨论设备驱动程序如何使用它们来实现延迟的操作。当然，驱动程序中可以使用的延迟操作机制并非只有 `softirq/tasklet` 和工作队列 `workqueue` 这两种，比如定时器 `timer` 也可以用来实现延迟的操作。不过笔者打算把 `timer` 放到“时间管理”一章中讨论，因为定时器 `timer` 和时间管理这一话题在逻辑上的联系要更紧密一些。

6.1 tasklet

`tasklet` 是内核定义的几种 `softirq` 之一，设备驱动程序的中断处理例程常常利用 `tasklet` 来完成一些延后的处理。根据优先级的不同，内核将 `tasklet` 分成两种，在 `softirq` 中对应 `TASKLET_SOFTIRQ` 和 `HI_SOFTIRQ`，后者的执行顺序优于前者。Linux 内核定义的 `softirq` 有：

```
<include/linux/interrupt.h>
-----
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
```

```

        BLOCK_IOPOLL_SOFTIRQ,
        TASKLET_SOFTIRQ,
        SCHED_SOFTIRQ,
        HRTIMER_SOFTIRQ,
        RCU_SOFTIRQ, /* Preferable RCU should always be the last softirq */

        NR_SOFTIRQS
};

```

其中 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 就是本章要讨论的主题 tasklet。

6.1.1 tasklet 机制初始化

Linux 系统初始化期间通过调用 softirq_init 为 TASKLET_SOFTIRQ 和 HI_SOFTIRQ 安装了执行函数：

```

<kernel/softirq.c>
-----
void __init softirq_init(void)
{
    int cpu;

    for_each_possible_cpu(cpu) {
        int i;

        per_cpu(tasklet_vec, cpu).tail =
            &per_cpu(tasklet_vec, cpu).head;
        per_cpu(tasklet_hi_vec, cpu).tail =
            &per_cpu(tasklet_hi_vec, cpu).head;
        for (i = 0; i < NR_SOFTIRQS; i++)
            INIT_LIST_HEAD(&per_cpu(softirq_work_list[i], cpu));
    }

    register_hotcpu_notifier(&remote_softirq_cpu_notifier);

    open_softirq(TASKLET_SOFTIRQ, tasklet_action);
    open_softirq(HI_SOFTIRQ, tasklet_hi_action);
}

```

函数中的 for_each_possible_cpu 循环用来初始化管理 tasklet 链表的变量 tasklet_vec 和 tasklet_hi_vec, 稍后会谈到这两个变量的具体用途。open_softirq 用来给 TASKLET_SOFTIRQ 和 HI_SOFTIRQ 安装对应的执行函数：

```

softirq_vec[TASKLET_SOFTIRQ].action = tasklet_action;
softirq_vec[HI_SOFTIRQ].action = tasklet_hi_action;

```

上述代码中，softirq_vec 是一个 struct softirq_action 类型的数组，数组中的每一项都对应一

个软中断处理函数指针。该数组在源码中的定义如下：

```
<kernel/softirq.c>
```

```
static struct softirq_action softirq_vec[NR_SOFTIRQS] __cacheline_aligned_in_smp;
```

软中断处理函数原型则由 `struct softirq_action` 来定义：

```
<include/linux/interrupt.h>
```

```
struct softirq_action
```

```
{
    void (*action)(struct softirq_action *);
};
```

如此，在中断处理的 `SOFTIRQ` 部分，如果发现本地 CPU 的 `__softirq_pending` 上 `TASKLET_SOFTIRQ` 或者 `HI_SOFTIRQ` 位被置 1，就将调用 `tasklet_action` 或者 `tasklet_hi_action`。后面会看到 `__softirq_pending` 与 `softirq_vec` 数组间的对应关系。

6.1.2 提交一个 tasklet

本节将讨论设备驱动程序如何利用内核提供的 `tasklet` 机制来实现一个延迟的操作。内核为此定义了一个表示 `tasklet` 对象的数据结构 `struct tasklet_struct`：

```
<include/linux/interrupt.h>
```

```
struct tasklet_struct
```

```
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

其中：

```
struct tasklet_struct *next
```

用来将系统中的 `tasklet` 对象构建成链表。

```
unsigned long state
```

记录每个 `tasklet` 在系统中的状态，其值是枚举型变量 `TASKLET_STATE_SCHED` 和 `TASKLET_STATE_RUN` 两者之一。`TASKLET_STATE_SCHED` 表示当前 `tasklet` 已经被提交；`TASKLET_STATE_RUN` 只用在对称多处理器系统 SMP 中，表示当前 `tasklet` 正在执行。

```
atomic_t count
```

用来实现 tasklet 的 disable 和 enable 操作, count.counter=0 表示当前的 tasklet 是 enabled 的, 可以被调度执行, 否则便是个 disabled 的 tasklet, 不可以被执行。

```
void (*func)(unsigned long)
```

该 tasklet 上的执行函数或者延迟函数, 当该 tasklet 在 SOFTIRQ 部分被调度执行时, 该函数指针指向的函数被调用, 用来完成驱动程序中实际的延迟操作任务。

```
unsigned long data
```

func 所指向的函数被调用时, data 将作为参数传给 func 函数。驱动程序可以利用 data 向 tasklet 上的执行函数传递特定的参数。

驱动程序为了实现基于 tasklet 机制的延迟操作, 首先需要声明一个 tasklet 对象。驱动程序可以用 DECLARE_TASKLET 宏声明并初始化一个静态的 tasklet 对象:

```
<include/linux/interrupt.h>
-----
#define DECLARE_TASKLET(name, func, data) \
    struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }
```

相对于 DECLARE_TASKLET 宏, 另一个相似的宏 DECLARE_TASKLET_DISABLED 则用来声明一个处于 disabled 状态的 tasklet 对象:

```
#define DECLARE_TASKLET_DISABLED(name, func, data) \
    struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func, data }
```

以上宏定义中, name 是声明的 tasklet 对象的名称, func 是驱动程序中用来实现延迟操作的函数, data 是传递给 func 函数的参数。

如果驱动程序在运行过程中构建了一个 tasklet 对象, 这种情况下对 tasklet 对象的初始化可以通过调用函数 tasklet_init 来完成:

```
<kernel/softirq.c>
-----
void tasklet_init(struct tasklet_struct *t,
                  void (*func)(unsigned long), unsigned long data)
{
    t->next = NULL;
    t->state = 0;
    atomic_set(&t->count, 0);
    t->func = func;
    t->data = data;
}
```

声明了 tasklet 对象之后, 驱动程序需要调用 tasklet_schedule 来向系统提交这个 tasklet。这里所谓的提交, 实际上就是将一个 tasklet 对象加入到 tasklet_vec 管理的链表中。对于 HI_SOFTIRQ, 提交 tasklet 对象的函数为 tasklet_hi_schedule, 除了用来管理 tasklet 对象链

表的变量为 `tasklet_hi_vec` 外，其他方面完全一样。鉴于这种代码层面的一致性，所以接下来把讨论的主角设定为 `tasklet_schedule`。为了方便这里的讨论，这里对 `tasklet_schedule` 函数进行了轻微的改动。

```
<include/linux/interrupt.h>
-----
static inline void tasklet_schedule(struct tasklet_struct *t)
{
    unsigned long flags;

    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state)){
        local_irq_save(flags);
        t->next = NULL;
        *__get_cpu_var(tasklet_vec).tail = t;
        __get_cpu_var(tasklet_vec).tail = &(t->next);
        raise_softirq_irqoff(TASKLET_SOFTIRQ);
        local_irq_restore(flags);
    }
}
```

函数中用到的 `tasklet_vec` 是个 per-CPU 型的变量，用来将系统中所有通过 `tasklet_schedule` 函数提交的 `tasklet` 对象构建成链表，如果是多处理器系统，那么每个处理器都将用各自的 `tasklet_vec` 链表管理提交到其上的 `tasklet`。`tasklet_vec` 在 Linux 源码中具体的声明和类型如下：

```
<kernel/softirq.c>
-----
struct tasklet_head
{
    struct tasklet_struct *head;
    struct tasklet_struct **tail;
};
static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec);
```

其中：

```
struct tasklet_struct *head
```

`head` 总是指向 `tasklet` 对象链表的第一个节点。

```
struct tasklet_struct **tail
```

这是个指向 `tasklet` 对象指针的指针。实际使用中，`tail` 总是保存 `tasklet` 链表最后一个节点所在 `tasklet` 对象中 `next` 成员的地址。

`tasklet_vec` 变量的初始化最早发生在 Linux 系统初始化阶段调用的 `softirq_init` 函数中。上一小节提到了该函数，在那里将 `tasklet_vec` 的成员 `head` 的地址赋给了 `tail`，在 `tasklet_schedule` 函数中正是通过操作 `tail` 的方式将 `tasklet` 对象依次加入到了链表中。

函数首先检查要提交的 tasklet 的 state 上的 TASKLET_STATE_SCHED 位有没有置 1，对一个尚未提交过的 tasklet 对象来说，其值应该是 0，所以 test_and_set_bit 函数会返回 0，同时把 tasklet 的 state 上的 TASKLET_STATE_SCHED 位置 1 表明这个 tasklet 已被提交。此后该 tasklet 对象的 TASKLET_STATE_SCHED 位一直为 1 直到被调度运行，因此一个 tasklet 对象在被成功提交进系统但尚未被调度执行时，处于 TASKLET_STATE_SCHED 状态。此时即便是在多处理器系统中，运行在其他处理器上的 tasklet_schedule 函数也无法再次提交一个处于 TASKLET_STATE_SCHED 状态的 tasklet 对象，因此一个 tasklet 对象在同一时间只可能在一个处理器上运行，而不会同时有多个实例在不同的 CPU 上运行。

如果 tasklet 可以被提交，那么接下来的工作就是把它加入到当前处理器 tasklet_vec 管理的链表中，然后再通过 raise_softirq_irqoff(TASKLET_SOFTIRQ)调用告诉 SOFTIRQ 部分当前处理器有个 TASKLET_SOFTIRQ 正等待处理。raise_softirq_irqoff 用一个整型变量的位来表示该位上是否有待决的 softirq 等待处理，1 表示有，0 则是没有。

关于 tasklet_vec 建立链表的操作，因为接下来的讨论中还会经常见到，读者不妨通过图 6-1 来建立一个初步的印象：

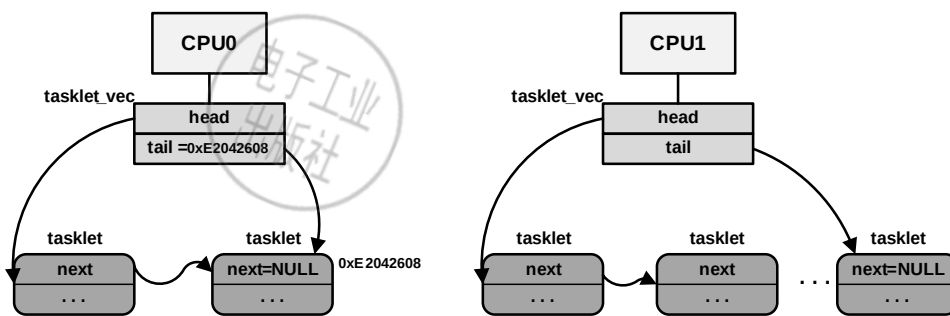


图 6-1 tasklet_vec 链表

图中，tasklet_vec 的 head 成员总是指向所管理链表的第一个节点，tail 总是保存链表最后一个节点 next 成员的地址。作为示例，图中的 0xE2042608 表示 next 所在的内存地址，这里笔者简单地用一根带箭头的线指向链表的最后一个节点。这样，如果要把一个新的节点 t 加入到链表尾部，只需如下操作即可：

```

t->next = NULL;
*__get_cpu_var(tasklet_vec).tail = t;
__get_cpu_var(tasklet_vec).tail = &(t->next);

```

其实像这种单向链表的操作很简单，似乎用不着通过 tasklet_vec 这种很绕的方式来实现，但是 Linux 内核这样做有它的道理，后面在讨论 tasklet_action 的时候会看到这点。

下面是一个设备驱动程序在其中断处理例程 demo_isr 中通过 tasklet_schedule 实现的一个延迟操作示例：

```

//设备相关的指针
static struct demo_dev * p = ...;

//延迟操作函数
void demo_delay_action(unsigned long data)
{
    //通过 data 获得设备相关指针
    static struct demo_dev * pdev = (static struct demo_dev *)data;
    //延迟操作
    ...
}

//用 DECLARE_TASKLET(name, func, data)定义一个 tasklet 对象 demo_tasklet
DECLARE_TASKLET(demo_tasklet, demo_delay_action, (unsigned long)p);

//中断处理例程
irqreturn_t demo_isr(int irq, void * dev_id)
{
    ...
    //通过 tasklet_schedule 实现延迟操作
    tasklet_schedule(&demo_tasklet);
}

```

示例中的 `demo_delay_action` 函数将延迟到中断处理的 `SOFTIRQ` 部分才会被执行到。

6.1.3 tasklet_action

上一小节讨论了设备驱动程序通过 `tasklet_schedule` 向系统提交一个 `tasklet` 对象执行延迟操作的实现机制，本节将会看到中断处理的 `SOFTIRQ` 部分如何去调用这些延迟的操作函数。

在“tasklet 机制初始化”小节，内核为 `TASKLET_SOFTIRQ` 和 `HI_SOFTIRQ` 分别安装了执行函数 `tasklet_action` 和 `tasklet_hi_action`，鉴于这两个执行函数的实现机制完全一样，在此只对 `tasklet_action` 的实现机制进行分析。下面是这个函数的实现：

```

<kernel/softirq.c>
-----
static void tasklet_action(struct softirq_action *a)
{
    struct tasklet_struct *list;

    local_irq_disable();
    list = __get_cpu_var(tasklet_vec).head;
    __get_cpu_var(tasklet_vec).head = NULL;
    __get_cpu_var(tasklet_vec).tail = &__get_cpu_var(tasklet_vec).head;
    local_irq_enable();
}

```



```

while (list) {
    struct tasklet_struct *t = list;
    list = list->next;
    if (tasklet_trylock(t)) {
        if (!atomic_read(&t->count)) {
            if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
                BUG();
            t->func(t->data);
            tasklet_unlock(t);
            continue;
        }
        tasklet_unlock(t);
    }

    local_irq_disable();
    t->next = NULL;
    *__get_cpu_var(tasklet_vec).tail = t;
    __get_cpu_var(tasklet_vec).tail = &(t->next);
    __raise_softirq_irqoff(TASKLET_SOFTIRQ);
    local_irq_enable();
}
}

```

函数的主体是个 while 循环，在进入 while 循环之前，需要得到 tasklet 链表的头指针，这需要访问 per-CPU 变量 tasklet_vec，因为该变量用来管理 tasklet 链表，tasklet_vec.head 指向 tasklet 链表的第一个节点。注意在访问 tasklet_vec 之前，函数用 local_irq_disable 关闭了处理器的中断，这是因为虽然 tasklet_vec 在系统的每个处理器中都有个副本，但是在单一 CPU 的范围里，依然存在 SOFTIRQ 在执行时被外部设备中断，在它的中断处理例程中使用到了 tasklet 的功能比如调用 tasklet_schedule 来提交一个 tasklet 对象，这样会导致两个执行路径都有操作 tasklet_vec 的可能性。所以此处用 local_irq_disable 和 local_irq_enable 来保护 tasklet_vec 不会在可能的并发访问中遭到破坏，其间的代码将 tasklet_vec 管理的链表的第一个节点存放在本地变量 list 中，然后将 tasklet_vec 设置成其最初的状态（空链表）。

在继续对 while 循环中代码的讨论之前，有两点需要注意。

首先，tasklet_action 作为一个 softirq 执行函数，在多处理器系统中可能同时在不同的 CPU 上运行。虽然一个处于 TASKLET_STATE_SCHED 状态的 tasklet 对象不能被多次提交，但是当个 tasklet 对象被调度运行时，TASKLET_STATE_SCHED 状态位会被清除，这样就可能导致该 tasklet 对象在别的处理器上被重新提交。考虑一下如下的情形：在一个有 A 和 B 两个处理器的系统中，某设备对处理器 A 产生了一次中断，在它的中断处理例程中会调用 tasklet_schedule 函数向系统提交一个 tasklet 对象，假设处理器 A 已经进入本次中断处理的 SOFTIRQ 部分并且正在运行该 tasklet，注意此时它的 TASKLET_STATE_SCHED 状态位已经被清除，此时该设备又产生了一次中断，这次的中断发送给了处理器 B，处理器 B 在

它的中断处理例程中同样会调用 `tasklet_schedule` 把同一个 `tasklet` 对象向处理器 B 上的 `tasklet_vec` 链表提交, 因为该 `tasklet` 的 `TASKLET_STATE_SCHED` 状态位已经被清除, 所以提交是可能成功的。如此就可能出现同一 `tasklet` 对象的执行函数在不同的处理器上同时运行的情形, 因此 `while` 循环需要某种机制来确保这种情况不会发生。我们不妨把这个问题称为 SMP 中 `tasklet` 运行冲突的问题, 等下会在 `while` 循环具体的代码实现中看到内核对此给出的解决方案。

其次, 在 `while` 循环中 `tasklet_action` 通过一个本地变量 `list` 来实现对 `tasklet` 链表的遍历。对于遍历过程中的每一个 `tasklet` 节点, 如果不满足执行的条件, 将通过操作 `tasklet_vec.tail` 指针将其重新加入 `tasklet_vec` 链表; 如果它被成功执行了, 那么该 `tasklet` 对象将不会再出现于 `tasklet_vec` 链表中。通过启用一个本地变量 `list`, 使得我们在调用 `tasklet` 上的执行函数时, 无须再考虑 `list` 链表的互斥访问问题, 因此读者可以看到 `tasklet` 上的执行函数在运行期间, 中断是打开的, 这也是 `SOFTIRQ` 当初的设计初衷。如果考虑到在某个 `tasklet` 运行期间发生了中断, 那么可能会有新的 `tasklet` 要被提交到当前处理器的 `tasklet_vec` 链表上, 不过这不会影响到 `list` 所在链表, 新的 `tasklet` 对象将会加入到 `tasklet_vec` 链表中。如此, 在 `tasklet_action` 执行前后, `tasklet_vec` 链表发生的变化是: 一些新的 `tasklet` 对象可能被提交进来, 只是因为还没有被运行过, 所以新节点将处于 `TASKLET_STATE_SCHED` 状态, 而被运行过的老节点其 `TASKLET_STATE_SCHED` 状态位将被清除, 而且也不会再出现在当前处理器的 `tasklet_vec` 链表中。

现在来看看 `while` 循环实际的代码, `tasklet_trylock` 在单处理器系统中直接返回 1, 在多处理器中, 其定义是:

```
<include/linux/interrupt.h>
-----
static inline int tasklet_trylock(struct tasklet_struct *t)
{
    return !test_and_set_bit(TASKLET_STATE_RUN, &(t)->state);
}
```

函数将 `tasklet` 中 `state` 的 `TASKLET_STATE_RUN` 位置 1, 同时返回 `TASKLET_STATE_RUN` 位原来的值。因此, `while` 循环中的 `if (tasklet_trylock(t))` 实际上就是用来解决前面提到的 SMP 中 `tasklet` 运行冲突的问题的。在 SMP 系统中一个运行中的 `tasklet` (其 `TASKLET_STATE_RUN` 位被置 1, `TASKLET_STATE_SCHED` 位被清 0) 有可能被重新提交到另一个处理器的 `tasklet_vec` 链表中, 为了防止该 `tasklet` 同时在不同的处理器上运行, 内核在 SMP 系统中为 `tasklet` 对象增加了一个额外的状态位 `TASKLET_STATE_RUN`, 这个状态位只对 SMP 系统有效, 单处理器系统不需要这个状态。内核用 `tasklet` 对象的 `TASKLET_STATE_RUN` 位来标记对应的 `tasklet` 当前是否正在运行, 如果没有, 那么 `tasklet_trylock(t)` 将返回真, 同时 `tasklet_trylock` 也会将 `state` 中的 `TASKLET_STATE_RUN` 位置 1, 这样别的 CPU 再运行 `tasklet_action` 时, 将不会处理该 `tasklet` 直到其运行完毕清除掉

`TASKLET_STATE_RUN`。内核通过这种方式实现了 `tasklet` 的串行化：任一时刻 `tasklet` 只能在一个 CPU 上运行。对于单处理器，不存在 `tasklet` 运行冲突的问题，所以 `tasklet_trylock` 直接返回 1。

接下来通过 `atomic_read` 对 `tasklet` 的 `count` 成员进行测试，这个成员主要用来实现 `enable` 或者 `disable` 一个 `tasklet`，如果某个 `tasklet` 对象的 `count` 为 0，说明它处在 `enabled` 的状态。对于一个 `enabled` 的 `tasklet`，需要再测试其 `state` 的 `TASKLET_STATE_SCHED` 位有没有被置 1，提交 `tasklet` 的函数会设置该位，如果该位没有被设置，说明 `tasklet_action` 函数正试图调度一个没有被提交的 `tasklet`，这是非正常状况。如果一切顺利，当前 `tasklet` 上的函数被调用，意味着延迟的操作开始进行。从代码中可以看到，如果一个 `tasklet` 被调度执行完之后，其 `state` 的 `TASKLET_STATE_SCHED` 位被清 0，这意味着除非被再次提交，否则下次的 `SOFTIRQ` 部分将不会再调度到它，这是一种 `one-shot` 特性：提交一次，调度运行一次，运行完后就从 CPU 的 `tasklet_vec` 链表中消失，除非有代码再次提交该 `tasklet` 对象。

通过上面对 `tasklet_action` 的分析可以看出，一个提交的 `tasklet` 在被 `SOFTIRQ` 调度执行完后，将从当前处理器的 `tasklet_vec` 链表中消失，因此除非再次提交，否则该 `tasklet` 对象将不会有会被再次运行。同时，内核对 `tasklet` 的实现机制确保了同一个 `tasklet` 对象不会同时在不同的处理器上运行，因此驱动程序在实现 `tasklet` 的延迟函数时，无须考虑多处理器间的并发问题。另外，`tasklet` 运行在中断上下文环境中，因此在中断上下文中的种种限制同样适用于 `tasklet` 的延迟函数。这些都是 `tasklet` 这种机制最典型的特质。

6.1.4 tasklet 的其他操作

前面已经讨论了 `tasklet` 的整个实现机制，下面在此基础上讲述 `tasklet` 一些其他的操作，包括如何 `disable` 和 `enable` 一个 `tasklet` 等。

○ `tasklet_disable` 和 `tasklet_disable_nosync`

这两个函数可以用来 `disable` 一个 `tasklet`，使之无法被 `SOFTIRQ` 调度运行。函数定义为：

```
<include/linux/interrupt.h>
-----
static inline void tasklet_disable_nosync(struct tasklet_struct *t)
{
    atomic_inc(&t->count);
    smp_mb__after_atomic_inc();
}

static inline void tasklet_disable(struct tasklet_struct *t)
{
    tasklet_disable_nosync(t);
    tasklet_unlock_wait(t);
    smp_mb();
}
```

```
}

```

disable 本身的行为很简单，将要操作的 tasklet 对象 t 上的 count 加 1 就可以了。相对于 tasklet_disable_nosync，tasklet_disable 是个“同步”版本，它在调用 tasklet_disable_nosync 函数之后，会再调用 tasklet_unlock_wait 函数实现所谓“同步”功能，这里的术语“同步”只限于 SMP 系统，单处理器系统中，tasklet_unlock_wait 什么也不做。多处理器系统中，如果要 disable 的 tasklet 正在运行，那么 tasklet_unlock_wait 要一直忙等待到 t 的 TASKLET_STATE_RUN 状态位被清除，就是说 tasklet_disable 要等到 t 运行完毕才会返回，这意味着 tasklet_disable 返回之后，可以确保该 tasklet 不会在系统的任何地方运行。

一个处于 disabled 状态的 tasklet 可以被提交到 tasklet_vec 中，但是不会被调度执行。

○ tasklet_enable

tasklet_enable 函数用来 enable 一个 tasklet，其定义如下：

```
<include/linux/interrupt.h>
-----
static inline void tasklet_enable(struct tasklet_struct *t)
{
    smp_mb__before_atomic_dec();
    atomic_dec(&t->count);
}
```

将指定的 tasklet 对象 t 上的 count 减 1。一个 tasklet 对象要能被执行，count 应该为 0。所以如果想 enable 一个先前被 disable 的 tasklet，使之能被调度执行，tasklet_enable 和 tasklet_disable 的调用次数要匹配。

○ tasklet_kill

```
<kernel/softirq.c>
-----
void tasklet_kill(struct tasklet_struct *t);
```

该函数通过清除一个 tasklet 对象的 TASKLET_STATE_SCHED 状态位，使 SOFTIRQ 不再能够调度运行它。如果当前 tasklet 对象正在运行，那么 tasklet_kill 将忙等待直到 tasklet 运行结束，这样可以确保 tasklet_kill 返回后系统中不再有运行中的该 tasklet 对象。如果一个 tasklet 对象被提交到了系统但还没有被调度执行，那么针对该 tasklet 对象调用 tasklet_kill，后者将会睡眠直到该 tasklet 被执行完从 tasklet_vec 链表中移除，所以 tasklet_kill 是个可能会被阻塞的函数。

一般在设备驱动程序所在的内核模块要被移除或者是设备要被关闭时，才调用该函数，因为这种情况下虽然你可以删除 tasklet 对象所在的空间，但这不会影响到 tasklet_vec 已有的链表元素构成，所以一个可能的情况是，在你的驱动模块已经移出系统，SOFTIRQ 还是调度运行了你的驱动程序提交的 tasklet 对象，这是一种危险情况，因为当你的模块已经从系

统中移除之后，被调度运行的 `tasklet` 函数也许会使用到模块中的资源，但是现在它们已经不存在了。内核模块调用 `tasklet_kill` 可以确保不会发生这种情况。

6.2 工作队列 work queue

工作队列是设备驱动程序可以使用的另一种延迟执行的方法。为了实现这种延迟执行的机制，内核或者驱动程序需要建立一套完整的基础设施，这里的设计思想与现实中的工厂加工非常相像：基础设施就是一条加工厂的成产流水线和在流水线上工作的工人，平时没事的时候，流水线上的工人就休息。如果某一客户想要加工一件工件，只需要把要加工的工件打个包（包里放有记载该工件应该如何加工的文档），扔到流水线上，然后客户可以继续做自己的事情。流水线上的工人发现有活要做，就结束休息，帮助客户加工工件。这个我们所熟悉的场景对应到 Linux 内核代码的世界，流水线变成了 `worklist`，工人变成了 `worker_thread`，打成包的工件就是 `struct work_struct` 对象，把包扔到流水线的工作变成了 `queue_work` 函数的调用等等，所有这些我们都将在接下来的内容中看清它们的内部运作流程。为了叙述上的方便，我们不妨就把这整个所谓的基础设施统称为工作队列。

内核本身提供了一套默认的工作队列，但是驱动程序自身也可以另起炉灶创建属于自己的基工作队列。本节将先从驱动程序创建自己的工作队列谈起，讨论整个延迟处理的工作流程，然后再把讨论的范围延伸到内核自己创建的基础设施上去，最后对比工作队列与 `tasklet` 机制的区别以及各自的适用场景。

6.2.1 数据结构

在具体讨论创建工作队列的内核机制之前，先交代几个核心的数据结构，它们在后面的讨论中会频频出现。

```
<include/linux/workqueue.h>
```

```
struct work_struct {  
    atomic_long_t data;  
    struct list_head entry;  
    work_func_t func;  
};
```

驱动程序要通过工作队列实现延迟操作时，需要生成一个 `struct work_struct` 对象，本书称之为工作节点，然后通过 `queue_work` 函数将其提交给工作队列。

`atomic_long_t data`

驱动程序可以利用 `data` 来将设备驱动程序使用的某些指针传递给延迟函数。

struct list_head entry

双向链表对象，用来将提交的等待处理的工作节点形成链表。

work_func_t func

工作节点的延迟函数，用来完成实际的延迟操作。其原型定义如下：

```
typedef void (*work_func_t)(struct work_struct *work);
```

<kernel/workqueue.c>

```
struct cpu_workqueue_struct {
    spinlock_t lock;
    struct list_head worklist;
    wait_queue_head_t more_work;
    struct work_struct *current_work;
    struct workqueue_struct *wq;
    struct task_struct *thread;
} ____cacheline_aligned;
```

实际的代码中，struct cpu_workqueue_struct 对象是个 per-CPU 型的变量，通过 alloc_percpu 函数动态创建，系统中的每个 CPU 都有一份，本书称 struct cpu_workqueue_struct 为 CPU 工作队列管理结构。

spinlock_t lock

对象的自旋锁，用于对可能的并发访问该对象时提供互斥保护机制。

struct list_head worklist

双向链表对象，用来将驱动程序提交的工作节点形成链表。驱动程序中的延迟操作以工作节点的形式存在。

wait_queue_head_t more_work

等待队列头节点，工作队列的工人线程（worker_thread）没有工作节点需要处理时将进入睡眠状态，此时它需要进入该等待队列。

struct work_struct *current_work

用于记录当前工人线程正在处理的工作节点。

struct workqueue_struct *wq

指向系统工作队列管理结构，接下来有它的具体定义。

struct task_struct *thread

指向工人线程所在的进程空间结构 `workqueue_struct`。

```
struct workqueue_struct {
    struct cpu_workqueue_struct *cpu_wq;
    struct list_head list;
    const char *name;
    int singlethread;
    int freezeable;
    int rt;
};
```

相对于上面的 CPU 工作队列管理结构, 本书称 `struct workqueue_struct` 为工作队列管理结构, 内核会为创建的每个工作队列生成一个工作队列管理结构对象。

```
struct cpu_workqueue_struct *cpu_wq
```

指向 CPU 工作队列管理结构的 per-CPU 类型的指针。根据该指针, 系统中的每个 CPU 都可以通过 `per_cpu_ptr` 来获得属于自己的 CPU 工作队列管理结构的对象。

```
struct list_head list
```

双向链表对象, 用于将工作队列管理结构加入到一个全局变量 `workqueues` 中, 只对非 `singlethread` 工作队列有效。

```
const char *name
```

工作队列的名称。

```
int singlethread
```

标识创建的工作队列中工人线程的数量。

```
int freezeable
```

表示进程可否处于冻结状态。

```
int rt
```

用来调整 `worker_thread` 线程所在进程的调度策略。

6.2.2 `create_singlethread_workqueue` 和 `create_workqueue`

设备驱动程序通过这两个函数创建属于自己的基础设施, 严格地说, 其实它们是宏, 不过这种文字上的小区别对理解整个流程的内核实现并没有什么特别的意义, 所以不妨先展开来看看它们各自的定义:

```
<include/linux/workqueue.h>
```

```
#define create_workqueue(name) __create_workqueue_key ((name), 0, 0, 0, NULL, NULL)
#define create_singlethread_workqueue(name) \
__create_workqueue_key ((name), 1, 0, 0, NULL, NULL)
```

所以最终调用的函数是__create_workqueue_key，这才是真正的核心函数，create_workqueue 和 create_singlethread_workqueue 的区别在于调用__create_workqueue_key 时的第二个参数，接下来讨论__create_workqueue_key 函数的实现时再来看这个参数对驱动程序而言意味着什么。

内核源码中__create_workqueue_key 的定义如下：

```
<kernel/workqueue.c>
```

```
struct workqueue_struct * __create_workqueue_key(const char *name,
                                                int singlethread,
                                                int freezeable,
                                                int rt,
                                                struct lock_class_key *key,
                                                const char *lock_name)
{
    struct workqueue_struct *wq;
    struct cpu_workqueue_struct *cwq;
    int err = 0, cpu;

    wq = kzalloc(sizeof(*wq), GFP_KERNEL);
    if (!wq)
        return NULL;

    wq->cpu_wq = alloc_percpu(struct cpu_workqueue_struct);
    if (!wq->cpu_wq) {
        kfree(wq);
        return NULL;
    }

    wq->name = name;
    wq->singlethread = singlethread;
    wq->freezeable = freezeable;
    wq->rt = rt;
    INIT_LIST_HEAD(&wq->list);

    if (singlethread) {
        cwq = init_cpu_workqueue(wq, singlethread_cpu);
        err = create_workqueue_thread(cwq, singlethread_cpu);
        start_workqueue_thread(cwq, -1);
    } else {
        cpu_maps_update_begin();
```



```

spin_lock(&workqueue_lock);
list_add(&wq->list, &workqueues);
spin_unlock(&workqueue_lock);
for_each_possible_cpu(cpu) {
    cwq = init_cpu_workqueue(wq, cpu);
    if (err || !cpu_online(cpu))
        continue;
    err = create_workqueue_thread(cwq, cpu);
    start_workqueue_thread(cwq, cpu);
}
cpu_maps_update_done();
}

if (err) {
    destroy_workqueue(wq);
    wq = NULL;
}
return wq;
}

```

函数一开始便调用 `kzalloc` 生成了一个工作队列管理结构的对象 `wq` 并初始化，同时利用 `alloc_percpu` 函数生成了 `per-CPU` 类型的 CPU 工作队列管理结构对象：

```
wq->cpu_wq = alloc_percpu(struct cpu_workqueue_struct);
```

接下来函数根据参数 `singlethread` 的值对单线程队列和多线程队列分别进行处理。`create_singlethread_workqueue` 函数生成的工作队列是单线程的，`singlethread=1`，对这种情况，函数需要做的是：

(1) 调用 `init_cpu_workqueue` 函数，在该函数中获得系统中第一个 CPU（代码中的称谓是 `singlethread_cpu`）对应的 CPU 工作队列管理结构的指针 `cwq`，同时初始化 `cwq` 中的等待队列和双向链表等成员变量。

(2) 调用 `create_workqueue_thread` 函数生成工人线程（`worker_thread`）。Linux 内核中所谓的内核线程其实是一个进程，拥有独立的 `task_struct` 结构，这里的工人线程也不例外。`create_workqueue_thread` 函数实际的操作是生成一个新的进程，将该进程 `task_struct` 中保存有进程执行现场寄存器的 `pc` 值指向 `worker_thread` 函数¹（本书称 `worker_thread` 函数为工人线程的线程函数），这样当该进程被调度运行时将执行 `worker_thread` 函数，传给函数的参数是系统中第一个 CPU 上的 `cwq` 指针。新进程的 `task_struct` 结构体指针 `p` 将保存在 CPU 工作队列管理结构的 `thread` 成员中：`cwq->thread = p`。

¹ 新进程的实际入口点被设置为 `kernel_thread_helper` 函数，这样当新进程第一次被调度运行时将调用 `kernel_thread_helper` 函数，在那里才会调用 `worker_thread` 函数，内核在调用 `worker_thread` 函数之前将返回地址设定为 `do_exit`，如此若 `worker_thread` 函数结束运行，进程将在 `do_exit` 中消亡，所以系统中的 `worker_thread` 函数不会轻易退出。

(3) 调用 `start_workqueue_thread` 函数，后者再通过 `wake_up_process` 函数将新进程投入到系统的运行队列中：`wake_up_process(p)`，如此之后新进程就具备了被调度器调度运行的条件。

如果 `singlethread` 不为 1，那么 `__create_workqueue_key` 将对系统中的每个 CPU 调用 `singlethread` 中的三大步骤，这样每个 CPU 都将拥有自己的 CPU 工作队列管理结构和工作在其上的工人线程。这种情况下，工作队列管理结构对象 `wq` 还将把自己加入到 `workqueues` 管理的链表中：

```
list_add(&wq->list, &workqueues);
```

`workqueues` 是一个全局型的双向链表对象，用来链接系统中所有非 `singlethread` 的工作队列：

```
<kernel/workqueue.c>
```

```
static LIST_HEAD(workqueues);
```

图 6-2 描述了工作队列的实现框架：

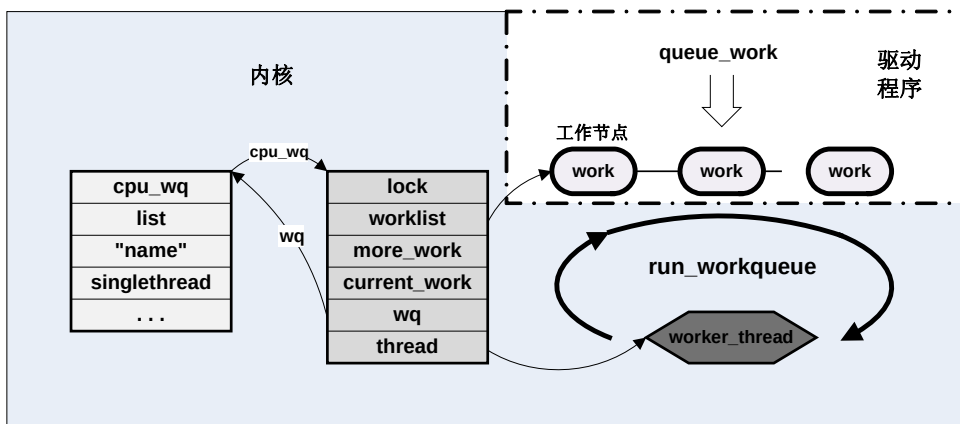


图 6-2 工作队列实现框架

图中，内核部分描述了通过 `create_singlethread_workqueue` 或者 `create_workqueue` 创建的工作队列及其上的工人线程 `worker_thread`，后者的任务是操作 `worklist` 链表上的工作节点，如果 `worklist` 上面没有工作节点，那么 `worker_thread` 所在的进程将进入睡眠状态并驻留在 `more_work` 维护的等待队列中。驱动程序部分将要延迟的操作打包进 `struct work_struct` 类型的工作节点中，然后通过 `queue_work` 向 `worklist` 上提交该工作节点，最后唤醒 `worker_thread` 线程。

图 6-2 描述的是 `singlethread` 工作队列，对于非 `singlethread` 工作队列，上面的工作原理依然适用，只是此时系统中的每个 CPU 都拥有自己的工作队列和工人线程 `worker_thread`。至于驱动程序提交节点时向哪个工作队列提交，在 `queue_work` 部分再讨论。

6.2.3 工人线程 `worker_thread`

工人线程 `worker_thread` 用来处理驱动程序提交到工作队列中的工作节点，如果工作队列中

没有节点需要处理，那么它将睡眠在 `cwq->more_work` 表示的等待队列中。`worker_thread` 运行在一个独立的新进程空间中。

<kernel/workqueue.c>

```
static int worker_thread(void *__cwq)
{
    struct cpu_workqueue_struct *cwq = __cwq;
    DEFINE_WAIT(wait);

    if (cwq->wq->freezable)
        set_freezable();

    for (;;) {
        prepare_to_wait(&cwq->more_work, &wait, TASK_INTERRUPTIBLE);
        if (!freezing(current) &&
            !kthread_should_stop() &&
            list_empty(&cwq->worklist))
            schedule();
        finish_wait(&cwq->more_work, &wait);

        try_to_freeze();

        if (kthread_should_stop())
            break;

        run_workqueue(cwq);
    }

    return 0;
}
```

`worker_thread` 的主体是一 `for(;;)` 循环，它首先用 `kthread_should_stop` 检测有没有别的函数对它调用了 `kthread_stop`，如果有的话，代表该线程的 `kthread` 对象的 `should_stop` 成员将被置 1，此时 `worker_thread` 将通过 `break` 跳出循环，线程函数所在的进程将会终结。如果 `worker_thread` 不需要 `stop` 而且 `cwq->worklist` 上也没有工作节点等待处理，工人线程将调用 `schedule` 以 `TASK_INTERRUPTIBLE` 状态睡眠在等待队列 `cwq->more_work` 中，直到驱动程序向 `cwq->worklist` 上提交了一个新的节点并唤醒 `worker_thread`，它醒来之后将调用 `run_workqueue` 来处理 `cwq->worklist` 上的工作节点：

<kernel/workqueue.c>

```
static void run_workqueue(struct cpu_workqueue_struct *cwq)
{
    spin_lock_irq(&cwq->lock);
    while (!list_empty(&cwq->worklist)) {
        struct work_struct *work = list_entry(cwq->worklist.next,
```

```

        struct work_struct, entry);

    work_func_t f = work->func;
    cwq->current_work = work;
    list_del_init(cwq->worklist.next);
    spin_unlock_irq(&cwq->lock);

    work_clear_pending(work);
    f(work);
    spin_lock_irq(&cwq->lock);
    cwq->current_work = NULL;
}
spin_unlock_irq(&cwq->lock);
}

```

函数在 `while` 循环中遍历 `cwq->worklist` 链表，对于其中的每个工作节点 `work`，先将其从 `cwq->worklist` 链表删除，然后调用工作节点上的延迟函数 `f(work)`，传递给函数的参数是延迟函数所在工作节点的指针 `work`。从 `run_workqueue` 的代码可以看出，一个工作节点被处理完之后，将不会再出现在工作队列的 `cwq->worklist` 链表中，除非被再次提交。

函数中的 `work_clear_pending` 用来清除 `work->data` 的 `WORK_STRUCT_PENDING` 位(位 0)，这里内核把 `work->data` 的低 2 位用于记录 `work` 的状态信息²，当驱动程序调用 `queue_work` 向工作队列提交节点 `work` 时，`queue_work` 会把 `work->data` 的 `WORK_STRUCT_PENDING` 位置 1，这是为了防止驱动程序将一个尚未被处理的工作节点再次向 `cwq->worklist` 上提交。

6.2.4 destroy_workqueue

`destroy_workqueue` 执行与 `create_singlethread_workqueue/create_workqueue` 相反的任务，当驱动程序不再需要使用后者创建的工作队列时（比如驱动程序所在的模块要从系统中移走或者关闭设备等），需要调用 `destroy_workqueue` 来做工作队列的清理善后工作，比如释放 `create_workqueue` 分配使用的一些系统资源如内存等，还有 `worker_thread` 线程也应该被安全地终结。

```

<kernel/workqueue.c>
-----
void destroy_workqueue(struct workqueue_struct *wq)
{
    const struct cpumask *cpu_map = wq_cpu_map(wq);
    int cpu;

    cpu_maps_update_begin();

```

² 此处内核利用了 `struct work_struct` 中 `data` 成员的低 2 位，所以驱动程序在利用 `data` 向延迟函数传递信息时，低 2 位应该是 0。这里暗含的意思是驱动程序只应该把 `data` 当做指针类型来使用。

```

spin_lock(&workqueue_lock);
list_del(&wq->list);
spin_unlock(&workqueue_lock);

for_each_cpu(cpu, cpu_map)
    cleanup_workqueue_thread(per_cpu_ptr(wq->cpu_wq, cpu));
cpu_maps_update_done();

free_percpu(wq->cpu_wq);
kfree(wq);
}

```

除了将 wq 从 workqueues 中移走及释放工作队列管理结构等对象所占用的内存外，主要的工作是调用 cleanup_workqueue_thread 来完全地终结 worker_thread，因为 destroy_workqueue 被调用的时候，worker_thread 很有可能正在处理 worklist 中余下的工作节点，因此函数要小心处理，避免发生不必要的麻烦。这里将 cleanup_workqueue_thread 稍作改写以突出其主线：

<kernel/workqueue.c>

```

static void cleanup_workqueue_thread(struct cpu_workqueue_struct *cwq)
{
    if (cwq->thread == NULL)
        return;

    flush_cpu_workqueue(cwq);

    kthread_stop(cwq->thread);
    cwq->thread = NULL;
}

```

函数的主要作用是通过调用 kthread_stop 函数来让 worker_thread 所在的进程终止，因为一旦进程的执行函数 worker_thread 结束，进程就将调用 do_exit 而终结，所以 kthread_stop 让 worker_thread 结束的原理就是设置 should_stop=1，前面在讨论 worker_thread 时已看到过 should_stop 的这一用法。

但是终止 worker_thread 所在进程的一个前提条件是要确保所有提交到 cwq->worklist 中的工作节点都已处理完毕，这是由 flush_cpu_workqueue 函数完成的：

<kernel/workqueue.c>

```

static int flush_cpu_workqueue(struct cpu_workqueue_struct *cwq)
{
    int active = 0;
    struct wq_barrier barr;

    WARN_ON(cwq->thread == current);
}

```

```

spin_lock_irq(&cwq->lock);
if (!list_empty(&cwq->worklist) || cwq->current_work != NULL) {
    insert_wq_barrier(cwq, &barr, &cwq->worklist);
    active = 1;
}
spin_unlock_irq(&cwq->lock);

if (active) {
    wait_for_completion(&barr.done);
    destroy_work_on_stack(&barr.work);
}

return active;
}

```

`flush_cpu_workqueue` 确保 `cwq->worklist` 上所有工作节点都已处理完毕的设计思想是利用完成接口 `completion`：如果 `cwq->worklist` 不为空或者 `cwq->current_work` 不为空，说明 `cwq_worklist` 上还有工作节点或者 `worker_thread` 正在处理一个工作节点，则向 `cwq->worklist` 上提交一个新的工作节点，这里不妨称之为中止节点。当中止节点上的延迟函数被执行时，它将调用 `complete` 函数通知 `flush_cpu_workqueue`，而后者在提交完中止节点之后将睡眠等待在 `wait_for_completion` 函数上，直到之前提交的中止节点上的延迟函数执行结束，如此可确保所有中止节点前的工作节点都会被处理完毕。

从函数的实现代码可以看到，虽然在 `insert_wq_barrier` 函数提交了中止节点之后，其他部分的代码依然可以向 `cwq->worklist` 提交新的工作节点，但是内核无法保证这些工作节点上的延迟函数有机会执行。函数中的 `WARN_ON(cwq->thread == current)` 意味着驱动程序不应该在提交的工作节点延迟函数中调用 `flush_cpu_workqueue`。

`flush_cpu_workqueue` 的操作范围只限于单个 CPU。对于非 `singlethread` 工作队列，因为每个 CPU 上都有一个工作队列和 `worker_thread`，要确保系统中所有 CPU 上的工作队列中的工作节点都被处理完，应该使用 `flush_workqueue` 函数：

<kernel/workqueue.c>

```

void flush_workqueue(struct workqueue_struct *wq)
{
    ...
    for_each_cpu(cpu, cpu_map)
        flush_cpu_workqueue(per_cpu_ptr(wq->cpu_wq, cpu));
}

```

不管是 `flush_cpu_workqueue` 还是 `flush_workqueue`，都是对工作队列 `worklist` 上所有的工作节点进行操作：函数返回后，可以确保函数调用前提交的所有工作节点都已处理完毕。与之类似的还有另外一个函数 `flush_work`，如果驱动程序想等待在某单个提交的工作节点上

直到该节点处理完毕函数才返回，就可以使用 `flush_work` 函数，其原型如下：

```
int flush_work(struct work_struct *work);
```

参数 `work` 就是调用者要等待在其上的工作节点，如果该函数调用时 `work` 已处理完毕，那么函数返回 0。

6.2.5 提交工作节点 `queue_work`

前面几节描述了工作队列在内核内部的实现机制，本节开始讨论驱动程序如何通过向工作队列提交节点的方式来执行延迟操作。

设备驱动程序将要延迟的操作打包进一个 `struct work_struct` 对象，也就是所谓的工作节点，然后通过 `queue_work` 函数来向工作队列提交该节点。

<kernel/workqueue.c>

```
int queue_work(struct workqueue_struct *wq, struct work_struct *work)
{
    int ret;

    ret = queue_work_on(get_cpu(), wq, work);
    put_cpu();

    return ret;
}
```

在之前的讨论中，驱动程序可以调用 `create_singlethread_workqueue` 和 `create_workqueue` 函数来让内核生成属于自己的工作队列，两者的区别是：`create_singlethread_workqueue` 只在系统中的第一个 CPU（`singlethread_cpu`）上创建工作队列和工人线程，而 `create_workqueue` 函数会在系统中的每个 CPU 上都创建工作队列和工人线程³。在用 `queue_work` 向工作队列提交工作节点时，如果工作队列是 `singlethread` 类型的，因为此时只有一个 `worklist`，所以 `queue_work` 没得选择，工作节点只能提交到这唯一的一个 `worklist` 上。反之，如果队列不是 `singlethread` 类型的，那么工作节点将会提交到当前运行 `queue_work` 的 CPU 所在的 `worklist` 中。

<kernel/workqueue.c>

```
int queue_work_on(int cpu, struct workqueue_struct *wq, struct work_struct *work)
{
    int ret = 0;

    if (!test_and_set_bit(WORK_STRUCT_PENDING, work_data_bits(work))) {
```

³ 如果是单处理器系统，两者就没有什么区别了。

```

        BUG_ON(!list_empty(&work->entry));
        __queue_work(wq_per_cpu(wq, cpu), work);
        ret = 1;
    }
    return ret;
}

```

函数首先检测 `work->data` 的 `WORK_STRUCT_PENDING` 位有没有被置 1，置 1 的话意味着此前该 `work` 已被提交还没有处理，内核禁止驱动程序在一个工作节点还没处理完就再次提交该节点。此处的检测也告诉驱动程序，在构造工作节点对象 `work` 时，应该确保 `work->data` 低 2 位为 0。如果 `work->data` 的 `WORK_STRUCT_PENDING` 位是 0，那么就把该位置 1 表明工作节点处于等待处理的状态，然后调用 `__queue_work` 来提交节点。`__queue_work` 的原型为：

```
static void __queue_work(struct cpu_workqueue_struct *cwq, struct work_struct *work);
```

第一个参数是 CPU 工作队列管理结构，第二个参数是待提交的节点指针。`queue_work_on` 在调用 `__queue_work` 时传递的第一个参数是 `wq_per_cpu(wq, cpu)`，为了搞清向哪个 `cwq` 提交节点，不妨看看 `wq_per_cpu` 的实现：

<kernel/workqueue.c>

```

static struct cpu_workqueue_struct *wq_per_cpu(struct workqueue_struct *wq, int cpu)
{
    if (unlikely(is_wq_single_threaded(wq)))
        cpu = singlethread_cpu;
    return per_cpu_ptr(wq->cpu_wq, cpu);
}

```

函数的实现很简单，如果是 `singlethread` 类型的工作队列，那么工作节点就提交到第一个 CPU 的 `cwq` 上，否则哪个 CPU 调用 `queue_work`，工作节点就提交到哪个 CPU 的 `cwq` 上。

下面继续看 `__queue_work`，其内部通过调用 `insert_work(cwq, work, &cwq->worklist)` 来完成节点的提交。`insert_work` 的定义如下：

<kernel/workqueue.c>

```

static void insert_work(struct cpu_workqueue_struct *cwq,
                       struct work_struct *work, struct list_head *head)
{
    set_wq_data(work, cwq);
    smp_wmb();
    list_add_tail(&work->entry, head);
    wake_up(&cwq->more_work);
}

```

函数的主体和我们的预期完全一样：将工作节点加到 `cwq->worklist` 链表的尾部，然后调用

wake_up 唤醒在等待队列 cwq->more_work 上睡眠的 worker_thread，如果 worker_thread 正在运行，那么 wake_up 就什么也不做。

在把驱动程序向工作队列提交节点的 queue_work 函数搞清楚之后，再回过头来看看实际的驱动程序代码中如何动态初始化一个工作队列节点 work_struct 的对象。内核为此提供了两个宏 PREPARE_WORK 和 INIT_WORK，展开后如下：

```
<include/linux/workqueue.h>
-----
#define PREPARE_WORK(_work, _func) \
do { \
    (_work)->func = (_func); \
} while (0)

#define INIT_WORK(_work, _func) \
do { \
    (_work)->data = (atomic_long_t) ATOMIC_LONG_INIT(0); \
    INIT_LIST_HEAD(&(_work)->entry); \
    PREPARE_WORK((_work), (_func)); \
} while (0)
```

INIT_WORK 初始化 struct work_struct 中的每个成员，而 PREPARE_WORK 只是重新设置 struct work_struct 中的 func 指针。在实际的驱动程序中，使用 INIT_WORK 的机会要比 PREPARE_WORK 大得多。

除了这种动态初始化，内核还提供了另外一个宏 DECLARE_WORK，可以让驱动程序静态定义一个 struct work_struct 对象同时初始化：

```
<include/linux/workqueue.h>
-----
#define DECLARE_WORK(n, f) struct work_struct n = { \
    .data = WORK_DATA_STATIC_INIT(), \
    .entry = { &(n).entry, &(n).entry }, \
    .func = (f), \
    __WORK_INIT_LOCKDEP_MAP(#n, &(n)) \
}
```

除 queue_work 之外，内核还提供了另外一个提交节点的函数 queue_delayed_work：

```
<kernel/workqueue.c>
-----
int queue_delayed_work(struct workqueue_struct *wq,
    struct delayed_work *dwork, unsigned long delay)
{
    if (delay == 0)
        return queue_work(wq, &dwork->work);

    return queue_delayed_work_on(-1, wq, dwork, delay);
}
```

```
}
```

相对于 `queue_work`, `queue_delayed_work` 多了个参数 `delay`, 不过在 `delay=0` 的情况下, 对 `queue_delayed_work` 就蜕变成了 `queue_work`, 上面的代码很明显地展示了这一点。如果 `delay` 不等于 0, 那么它代表一个延迟的时间, 换句话说调用 `queue_delayed_work` 之后, 工作节点 `work` 需要等到 `delay` 指定的时间过后才会被真正提交到队列 `wq` 上。这种延迟提交的工作在 `queue_delayed_work_on` 函数中完成:

```
<kernel/workqueue.c>
```

```
int queue_delayed_work_on(int cpu, struct workqueue_struct *wq,
                          struct delayed_work *dwork, unsigned long delay)
{
    int ret = 0;
    struct timer_list *timer = &dwork->timer;
    struct work_struct *work = &dwork->work;

    if (!test_and_set_bit(WORK_STRUCT_PENDING, work_data_bits(work))) {
        timer_stats_timer_set_start_info(&dwork->timer);
        set_wq_data(work, wq_per_cpu(wq, raw_smp_processor_id()));
        timer->expires = jiffies + delay;
        timer->data = (unsigned long)dwork;
        timer->function = delayed_work_timer_fn;

        if (unlikely(cpu >= 0))
            add_timer_on(timer, cpu);
        else
            add_timer(timer);
        ret = 1;
    }
    return ret;
}
```

函数的设计思想比较直白, 利用定时器 `timer` 来实现延迟提交的工作, `timer->expires = jiffies + delay`, 这样当 `delay` 时间到期后, `timer->function = delayed_work_timer_fn` 将被调用, `delayed_work_timer_fn` 会把 `queue_delayed_work_on` 要提交的节点提交到工作队列中。所以, 驱动程序如果要使用 `queue_delayed_work`, 要先生成一个 `struct delayed_work` 对象。`struct delayed_work` 定义为:

```
<include/linux/workqueue.h>
```

```
struct delayed_work {
    struct work_struct work;
    struct timer_list timer;
};
```

延迟函数所在的工作节点在 `struct delayed_work` 结构体的 `work` 成员中, 其另一个成员是个

timer 对象,用来实现时间上的延迟操作,queue_delayed_work 中的 delay 参数将用来给 timer 中的延时成员赋值。

至此,我们已经完整地讨论了工作队列整个框架的实现机制,包括内核部分如何建立队列和工人线程,以及驱动程序部分如何利用内核提供的接口向工作队列中提交工作节点实现延迟的操作。为了加深读者对这部分内容的理解,下面用一个使用了工作队列执行延迟操作的代码片段作为具体的范例:

```
//定义全局性的 struct workqueue_struct 指针 demo_dev_wq
static struct workqueue_struct * demo_dev_wq;

//设备特定的数据结构,实际使用中大部分 struct work_struct 结构都内嵌在这个数据结构中
struct demo_device{
    ...
    struct work_struct work;
    ...
}
static struct demo_device *demo_dev;

//定义延迟的操作函数
void demo_work_func(struct work_struct *work)
{
    ...
}

//驱动程序模块初始化代码调用 create_singlethread_workqueue 创建工作队列
static int __init demo_dev_init (void)
{
    ...
    demo_dev = kzalloc(sizeof * demo_dev, GFP_KERNEL);
    demo_dev_wq = create_singlethread_workqueue("demo_dev_workqueue");
    INIT_WORK(&demo_dev->work, demo_work_func);
    ...
}

//模块退出函数
static void demo_dev _exit(void)
{
    ...
    flush_workqueue(demo_dev_wq);
    destroy_workqueue(demo_dev_wq );
    ...
}

//中断处理函数
irqreturn_t demo_isr(int irq, void * dev_id)
{

```

```

...
queue_work(demo_dev_wq, &demo_dev-> work);
...
}

```

6.2.6 内核创建的工作队列

Linux 系统在初始化阶段的 `init_workqueues` 函数中通过调用 `create_workqueue` 创建了一个名为 `events` 的工作队列：

```

<kernel/workqueue.c>
-----
static struct workqueue_struct *keventd_wq __read_mostly;
void __init init_workqueues(void)
{
    ...
    keventd_wq = create_workqueue("events");
    ...
}

```

前面已经仔细讨论了 `create_workqueue` 函数，内核在初始化阶段创建的工作队列与驱动程序自己调用 `create_workqueue` 函数创建的在本质上没有任何不同。设备驱动程序就算不创建自己的工作队列，也可以利用内核创建的工作队列来实现延迟操作，在提交工作节点时只需要调用 `queue_work(keventd_wq, work)` 即可。不过内核已经用另一个函数 `schedule_work` 包装了 `queue_work(keventd_wq, work)` 调用：

```

<kernel/workqueue.c>
-----
int schedule_work(struct work_struct *work)
{
    return queue_work(keventd_wq, work);
}

```

可以看到，驱动程序如果使用内核创建的工作队列，在提交工作节点时只需调用 `schedule_work` 函数就可以了。

对应 `queue_delayed_work`，对于内核创建的工作队列而言，延迟提交函数就变成了 `schedule_delayed_work`。

使用内核提供的工作队列的好处是，驱动程序无须创建自己的工作队列就可以提交节点来实现延迟操作。但是不好的地方也很明显：我们正在与系统中其他模块共享一个工作队列以及该队列上的 `worker_thread`，所以队列上的工作节点的多少我们无法预期，意味着我们无法确定在提交一个工作节点之后，需要多长时间才有机会被调度执行。同时，我们也应该避免在提交的延迟函数中执行很耗时的任务影响到他人。所以对使用内核创建的工作队列总结起来就是：虽然少了创建队列销毁队列这些麻烦事，但是使用起来的灵活性就

下降了。这里没有一成不变的规则，设备驱动程序需要根据实际情况做出适合自己的选择。

6.3 本章小结

本章深入讨论了设备驱动程序中经常使用的两种延迟操作的实现机制，分别是 `tasklet` 和 `workqueue`。

`tasklet` 的实现基于 `softirq` 机制，内核在初始化期间就初始化了 `HI_SOFTIRQ` 和 `TASKLET_SOFTIRQ` 两个 `softirq` 所对应的 action 函数：`tasklet_hi_action` 和 `tasklet_action`。

驱动程序在可以使用 `tasklet` 机制实现延迟操作前，需要定义一个 `tasklet` 对象，将要延迟的函数封装到该对象中，之后需要调用 `tasklet_schedule` 函数向系统提交该对象。中断处理的 `SOFTIRQ` 部分发现有等待的 `softirq` 需要处理时，就去处理被驱动程序提交的 `tasklet` 对象，在那里，被打包进 `tasklet` 对象的驱动程序中的实际延迟函数将被调用。当一个 `tasklet` 对象在 `SOFTIRQ` 部分处理完之后，除非再次提交，否则将不再会被执行。由此可见，通过 `tasklet` 实现的延迟操作，是运行在中断处理的上下文环境中，因此它不应该引入睡眠。`tasklet` 是严格串行化的：在任一时刻，同一 `tasklet` 只能有一个实例在运行，即使是多处理器系统也是如此。`tasklet` 的另一个特性是：哪个处理器调用 `tasklet_schedule` 提交的 `tasklet`，只能在该处理器上运行。

相对于 `tasklet`，工作队列的延迟函数是在一个独立的进程环境下运行的。系统中可能有两种形式的工作队列：`singlethread` 的和非 `singlethread` 的。对于多处理器系统而言，前者只在系统中的第一个 CPU 上产生工作队列和工人线程，而后者则为系统中的每个处理器都产生一个工作队列和工人线程。内核在初始化阶段创建了一个非 `singlethread` 的工作队列，驱动程序可以使用该队列，也可以调用 `create_workqueue` 或者 `create_singlethread_workqueue` 创建属于自己的工作队列。

为了实现延迟操作，驱动程序需要生成一个类型为 `struct work_struct` 的工作队列对象，将要延迟执行的函数打包到该对象中，然后通过 `queue_work` 函数向工作队列提交该节点。同 `tasklet` 对象一样，当 `work` 对象被处理完毕后除非被再次提交，否则将不再有执行的机会。与 `tasklet` 不同的是，基于工作队列的延迟操作是运行在进程的上下文中，所以允许睡眠。

第 7 章

设备文件的高级操作

在“字符设备驱动程序”一章中，讨论了 Linux 系统下字符设备驱动程序框架的内核机制，包括设备号的分配、设备对象的注册，以及设备文件节点的生成和文件的打开操作等。从本章开始，将在此基础上讨论针对设备文件的一些高级文件操作，包括驱动程序最常用的 `ioctl`、阻塞型 I/O、`poll`，以及异步通知机制等，基本上将围绕设备驱动程序需要实现的 `struct file_operations` 对象中所定义的一些函数来展开，重点集中在 `ioctl` 和字符设备的 I/O 模型上，希望读者阅读完本章后能进一步认识和理解字符设备驱动程序所要完成的各种操作。

7.1 `ioctl` 文件操作

设备文件的 `ioctl` 操作常用来对设备的行为进行某种控制，典型的用法比如对一个串口设备，可以通过 `ioctl` 配置其波特率和流控方法等。所以 `ioctl` 一般用来在用户空间的应用程序和驱动程序模块之间传递控制参数，而很少用于大数据量的传递。

7.1.1 `ioctl` 的系统调用

本节主要讨论 `ioctl` 的系统调用流程，使读者了解用户空间程序的 `ioctl` 如何调用到设备驱动程序中实现的 `ioctl`。

在用户空间，`ioctl` 函数的原型为：

```
<sys/ioctl.h>
```

```
int ioctl(int fd, int request, ...);
```

而对于设备驱动程序而言，所要实现的 `ioctl` 函数的原型有两个：

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

```
struct file_operations{
```

```
...
```

```
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
```

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

```
...
```

```
};
```

驱动程序应该实现其中的 `unlocked_ioctl`，`ioctl` 在 Linux 内核中属于比较陈旧的代码，之所以还存在，是考虑到一些老的驱动只实现了 `ioctl`，调整这些代码的工作量非常庞大。在后续对 `ioctl` 系统调用的讨论中，会看到这两者之间的区别。本书为了叙述的方便，将驱动程序中实现的这些函数原型统称为 `ioctl` 函数。

当用户空间程序调用 `ioctl` 函数时，系统会经过 `sys_ioctl` 进入到内核空间。系统调用 `sys_ioctl` 的定义为：

```
<fs/ioctl.c>
```

```
SYSCALL_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned long, arg)
{
    struct file *filp;
    int error = -EBADF;
    int fput_needed;

    filp = fget_light(fd, &fput_needed);
    if (!filp)
        goto out;

    error = security_file_ioctl(filp, cmd, arg);
    if (error)
        goto out_fput;

    error = do_vfs_ioctl(filp, fd, cmd, arg);
out_fput:
    fput_light(filp, fput_needed);
out:
    return error;
}
```

从图 2-9 “fd 与 filp 的关联” 可以看到，当用户空间打开一个设备文件时，内核将为之分配一个文件描述符 `fd`，同时生成一个 `struct file` 对象 `filp`，`fd` 与 `filp` 通过当前进程的 `files` 管理的文件描述符表关联起来，这意味着当接下来在刚打开的文件上执行其他操作时，比如这里的 `ioctl`，系统会把打开文件时获得的文件描述符 `fd` 作为参数传递给 `ioctl` 函数。在 `ioctl` 的系统调用函数 `sys_ioctl` 中，第一个参数就是刚打开文件的描述符 `fd`，因此可以推测 `sys_ioctl` 将会用 `fd` 作为进程管理的文件描述符表的索引，继而得到 `fd` 所对应的 `struct file` 对象的指针 `filp`，这个 `filp` 对象在之前打开文件的操作中已被创建并初始化，其中最重要的初始化是把设备对象 `cdev` 中的 `ops` 指针赋给了 `filp->f_op`，因此通过 `filp->f_op` 将调用到驱动程序提供的文件操作函数。

有了这种总体的脉络把握，再看上面的 `sys_ioctl` 函数的具体实现来验证刚才的推测。没错，

函数中的第一个调用 `fget_light` 就是通过 `fd` 获得 `struct file` 对象的指针 `filp`。函数中另一处关键的调用来自 `do_vfs_ioctl`，此处需要仔细考察一下该函数的实现代码来获得更多关于 `ioctl` 的认识，包括用户空间 `ioctl` 函数中各种参数的使用方法等。`do_vfs_ioctl` 的定义是：

<fs/ioctl.c>

```
int do_vfs_ioctl(struct file *filp, unsigned int fd, unsigned int cmd, unsigned long arg)
{
    int error = 0;
    int __user *argp = (int __user *)arg;

    switch (cmd) {
    case FIOCLEX:
        set_close_on_exec(fd, 1);
        break;

    case FIONCLEX:
        set_close_on_exec(fd, 0);
        break;

    case FIONBIO:
        error = ioctl_fionbio(filp, argp);
        break;

    case FIOASYNC:
        error = ioctl_fioasync(fd, filp, argp);
        break;

    case FIOQSIZE:
        if (S_ISDIR(filp->f_path.dentry->d_inode->i_mode) ||
            S_ISREG(filp->f_path.dentry->d_inode->i_mode) ||
            S_ISLNK(filp->f_path.dentry->d_inode->i_mode)) {
            loff_t res =
                inode_get_bytes(filp->f_path.dentry->d_inode);
            error = copy_to_user((loff_t __user *)arg, &res,
                                sizeof(res)) ? -EFAULT : 0;
        } else
            error = -ENOTTY;
        break;

    case FIFREEZE:
        error = ioctl_fsfreeze(filp);
        break;

    case FITHAW:
        error = ioctl_fsthaw(filp);
        break;
```



```

case FS_IOC_FIEMAP:
    return ioctl_fiemap(filp, arg);

case FIGETBSZ:
{
    struct inode *inode = filp->f_path.dentry->d_inode;
    int __user *p = (int __user *)arg;
    return put_user(inode->i_sb->s_blocksize, p);
}

default:
    if (S_ISREG(filp->f_path.dentry->d_inode->i_mode))
        error = file_ioctl(filp, cmd, arg);
    else
        error = vfs_ioctl(filp, cmd, arg);
    break;
}
return error;
}

```

从上面的函数实现中，可以看到用户空间 `ioctl` 函数原型中第二和第三个参数的用法，它们分别对应 `do_vfs_ioctl` 中的 `unsigned int cmd` 和 `unsigned long arg`。这里只是想让读者知道，通过 `ioctl` 的不同 `cmd` 参数，应用程序可以做很多事情，并不打算讲解每个 `cmd` 的具体用法。还是回到本章的主题“字符设备的高级操作”，有过用户空间对字符设备文件进行 `ioctl` 操作经验的读者应该知道，这种情况下调用流程将落入 `do_vfs_ioctl` 中 `default` 分支下的 `vfs_ioctl` 函数。虽然已经可以猜出后续的调用就是 `filp->f_op->ioctl`，但是笔者还是打算把 `vfs_ioctl` 的代码实现摘录于下，使读者获得更加直观的印象：

<fs/ioctl.c>

```

static long vfs_ioctl(struct file *filp, unsigned int cmd,
                     unsigned long arg)
{
    int error = -ENOTTY;

    if (!filp->f_op)
        goto out;

    if (filp->f_op->unlocked_ioctl) {
        error = filp->f_op->unlocked_ioctl(filp, cmd, arg);
        if (error == -ENOIOCTLCMD)
            error = -EINVAL;
        goto out;
    } else if (filp->f_op->ioctl) {
        lock_kernel();
        error = filp->f_op->ioctl(filp->f_path.dentry->d_inode,
                                filp, cmd, arg);
        unlock_kernel();
    }
}

```

```
    }

    out:
        return error;
}
```

函数没有需要太多解释的地方，如果读者之前认真阅读了 2.7 节“字符设备文件的打开操作”，那么此处 `filp->f_op` 与设备驱动程序中实现的 `struct file_operations` 对象之间的关联就会非常清晰。所以在你的设备驱动程序模块中要么实现了 `file_operations` 中的 `unlocked_ioctl`，要么是实现了 `ioctl`，它们都将在 `vfs_ioctl` 函数中的 `if...else if...` 框架中被检验并被调用，来自用户空间的 `cmd` 和 `arg` 参数将原样不动地传递给它们，驱动程序模块中实现的 `ioctl` 显然需要通过某些手段来获得用户空间的参数，在接下来的讨论中将看到这一点。

关于这里的 `unlocked_ioctl` 和 `ioctl` 调用，前面已经提过 `ioctl` 属于老的代码，在 `vfs_ioctl` 函数的实现中可以看到，内核在调用 `ioctl` 时，使用了 `lock_kernel` 和 `unlock_kernel` 作为互斥的手段。`lock_kernel` 和 `unlock_kernel` 是一种粗粒度的所谓大内核锁 BKL (Big Kernel Lock)，这种全局范围内使用的锁相对于现代的细粒度的锁机制而言，很明显会降低系统的性能，所以虽然大内核锁依然存在于 2.6 版本的内核中，但应该避免使用。现代的设备驱动程序应该使用 `unlocked_ioctl`，它们已经脱离了大内核锁的保护，因此驱动程序在实现 `unlocked_ioctl` 函数时，应该使用自己的互斥锁机制。关于 `unlocked_ioctl` 和 `ioctl` 的更多讨论可以参考 <http://lwn.net/Articles/119652/>。

至此，我们已经明白了用户空间的 `ioctl` 函数的调用最终是如何传递到驱动程序模块实现的文件操作上的，下面开始讨论 `ioctl` 中跟参数相关的问题。

7.1.2 ioctl 的命令编码

前面已经提到，`ioctl` 主要用来在用户空间程序和设备驱动模块之间传递控制信息，这个控制信息以 `cmd` 和 `arg` 的形式存在，因此需要在用户空间和内核空间建立一套规则来构造和解析 `cmd` 参数的组成，这样做的目的是确保任何一个遵循这些规则编码出来的 `cmd` 在系统范围内具有唯一性：如果有人无意间使用了某一 `cmd` 编码来对设备进行 `ioctl` 操作，驱动程序应该能识别出这一无意识的错误并返回相应的信息。内核为此定义了一套完整的宏，设备驱动程序应该使用这些宏来定义和解析各自使用的 `cmd` 参数。

为构造 `ioctl` 的 `cmd` 参数，内核使用了一个 32 位无符号整数并将其分成四个部分，如图 7-1 所示：

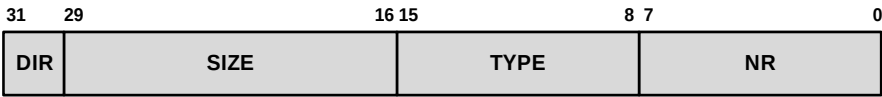


图 7-1 ioctl cmd 参数构成

NR

为功能号，长度为 8 位（_IOC_NRBITS）。

TYPE

为一 ASCII 字符，假定对每个驱动程序而言都是唯一的，长度是 8 位（_IOC_TYPEBITS）。实际的宏定义中因常常含有“MAGIC”字样，所以有时候也被称为魔数。

SIZE

表示 ioctl 调用中 arg 参数的大小，该字段的长度与体系架构相关，通常是 14 位（_IOC_SIZEBITS）。接下来会看到，其实内核在 ioctl 的调用中并没有用到该字段。

DIR

表示 cmd 的类型：read、write 和 read-write，长度是 2 位。这个字段用于表示在 ioctl 调用过程中用户空间和内核空间数据传输的方向，此处方向的定义是从用户空间的视角出发。内核为该字段定义的宏有：_IOC_NONE，表示在 ioctl 调用过程中，用户空间和内核空间没有需要传递的参数；_IOC_WRITE，表示在 ioctl 调用过程中，用户空间需要向内核空间写入数据；_IOC_READ，表示在 ioctl 调用过程中，用户空间需要从内核空间读取数据；_IOC_WRITE | _IOC_READ，表示在 ioctl 调用过程中，参数数据在用户空间和内核空间进行双向传递。

内核用宏 _IOC 将 NR、TYPE、SIZE 和 DIR 构造 cmd 参数：

```
<include/asm-generic/ioctl.h>
-----
#define _IOC(dir,type,nr,size) \
    (((dir) << _IOC_DIRSHIFT) | \
     ((type) << _IOC_TYPERSHIFT) | \
     ((nr) << _IOC_NRSHIFT) | \
     ((size) << _IOC_SIZESHIFT))
```

在此基础上，内核为方便代码的使用，定义了如下构造 cmd 参数的宏：

```
<include/asm-generic/ioctl.h>
-----
#define _IO(type,nr)      _IOC(_IOC_NONE,(type),(nr),0)
#define _IOR(type,nr,size) _IOC(_IOC_READ,(type),(nr),(_IOC_TYPECHECK(size)))
#define _IOW(type,nr,size) _IOC(_IOC_WRITE,(type),(nr),(_IOC_TYPECHECK(size)))
#define _IOWR(type,nr,size) _IOC(_IOC_READ|_IOC_WRITE,(type),(nr),(_IOC_TYPECHECK(size)))
```

其中 _IOC_TYPECHECK 用来对宏参数 size 进行检测，只在定义了 __KERNEL__ 的情况下有效，否则退化为 sizeof 运算符。

作为范例，下面给出一个用以上的宏定义的 ioctl 命令 DEMODEV_IOCINT，该命令从用户

空间向内核空间传递一个 int 型参数：

```
#define DEMODEV_IOC_MAGIC 'm'
#define DEMODEV_IOCINT      _IOW(DEMODEV_IOC_MAGIC, 0, int)
```

以上介绍的是内核为构造 ioctl 命令 cmd 所定义的宏。与此类似，内核为解析出这些 cmd 中的各个字段也定义了对应的宏：

```
<include/asm-generic/ioctl.h>
-----
#define _IOC_DIR(nr)      (((nr) >> _IOC_DIRSHIFT) & _IOC_DIRMASK)
#define _IOC_TYPE(nr)     (((nr) >> _IOC_TYPERSHIFT) & _IOC_TYPEMASK)
#define _IOC_NR(nr)       (((nr) >> _IOC_NRSHIFT) & _IOC_NRMASK)
#define _IOC_SIZE(nr)     (((nr) >> _IOC_SIZESHIFT) & _IOC_SIZEMASK)
```

这些宏的定义都比较直观易懂，这里不再多作介绍。

至此就介绍完了 ioctl 的命令构造和解析的方法。读者在实际使用这些宏的时候，有一点需要注意，前面在讨论 ioctl 系统调用的流程时，曾提到 sys_ioctl 最终会调用 do_vfs_ioctl，在那里函数会用 switch...case 语句对系统中几个预定义的 cmd 进行先行处理，所以设备驱动程序在定义自己的 ioctl cmd 时不应该和这些预定义的 cmd 发生冲突，否则内核将不会调用到驱动程序实现的 ioctl 函数，而且应用程序因为误用了系统预定义的 ioctl 命令而导致其行为的不可预期。一些常见的预定义有：

FIOCLEX

执行时关闭标志，即 File IOctl Close on EXec，通知内核在调用进程执行一个新程序时，比如 exec() 系统调用，自动关闭打开的文件。

FIONCLEX

清除执行时关闭标志，即 File IOctl Not CClose on EXec，与 FIOCLEX 标志相反，清除由 FIOCLEX 命令设置的标志。

FIONBIO

文件的 ioctl 为非阻塞型 I/O 操作，即 File IOctl Non-Blocking I/O，这个调用修改在 filp->f_flags 中的 O_NONBLOCK 标志。

FIOASYNC

设置或者复位文件的异步通知，这两个动作在内核中实际的执行者是 fcntl，所以内核代码并不使用该 cmd。

FIOQSIZE

获得一个文件或者目录的大小，用于设备文件时，将返回一个 ENOTTY 错误。

7.1.3 copy_from_user 和 copy_to_user

copy_from_user 和 copy_to_user 两个函数用于在用户空间和内核空间传递数据，设备驱动程序的 ioctl 函数实现中经常会用到这两个函数。先来看 copy_from_user 函数：

```
<include/asm-generic/uaccess.h>
-----
static inline long copy_from_user(void *to,
                                   const void __user * from, unsigned long n)
{
    might_sleep();
    if (access_ok(VERIFY_READ, from, n))
        return __copy_from_user(to, from, n);
    else
        return n;
}
```

先看函数的三个参数，*to 是内核空间的指针，*from 是用户空间指针，n 表示从用户空间向内核空间拷贝数据的字节数。关于函数的返回值，如果成功完成拷贝动作，返回 0，否则返回还没有完成拷贝的字节数。

might_sleep 在定义了 CONFIG_PREEMPT_VOLUNTARY 的情况下会形成一个显式的抢占调度点，换句话说 might_sleep 可能会自动放弃 CPU，关于这方面的细节不是本书要讨论的主题，总之，copy_from_user 有可能让当前进程进入睡眠状态。

access_ok 用来对用户空间的地址指针 from 作某种有效性检验，这个宏的定义是体系架构相关的，在 ARM 平台上为：

```
<arch/arm/include/asm/uaccess.h>
-----
#define access_ok(type,addr,size)  (__range_ok(addr,size) == 0)
```

可以看到，access_ok 中第一个参数 type 并没有被用到，__range_ok 的作用在于判断 addr + size 之后是否还在进程的用户空间范围之内。

接下来是核心函数 __copy_from_user，该函数的定义也是体系架构相关的，这里不打算一步一步详细展开这个函数。为了让读者理解在驱动程序中使用 copy_from_user 的必要性，下面给出 x86 平台上该函数的核心实现：

```
<arch/x86/lib/usercopy_32.c>
-----
#define __copy_user_zeroing(to, from, size) \
do { \
    int __d0, __d1, __d2; \
    __asm__ __volatile__( \
        "    cmp    $7,%0\n" \
        "    jbe    1f\n" \
        \
```

```

"    movl %1,%0\n"                \
"    negl %0\n"                    \
"    andl $7,%0\n"                \
"    subl %0,%3\n"                \
"4:  rep; movsb\n"                  \
"    movl %3,%0\n"                \
"    shrl $2,%0\n"                \
"    andl $3,%3\n"                \
"    .align 2,0x90\n"              \
"0:  rep; movsl\n"                  \
"    movl %3,%0\n"                \
"1:  rep; movsb\n"                  \
"2:\n"                             \
".section .fixup,\"ax\"\n"         \
"5:  addl %3,%0\n"                \
"    jmp 6f\n"                    \
"3:  lea 0(%3,%0,4),%0\n"          \
"6:  pushl %0\n"                   \
"    pushl %%eax\n"               \
"    xorl %%eax,%%eax\n"          \
"    rep; stosb\n"                 \
"    popl %%eax\n"                \
"    popl %0\n"                   \
"    jmp 2b\n"                    \
".previous\n"                      \
".section __ex_table,\"a\"\n"      \
"    .align 4\n"                   \
"    .long 4b,5b\n"               \
"    .long 0b,3b\n"               \
"    .long 1b,6b\n"               \
".previous\n"                      \
: "&c"(size), "&D" (__d0), "&S" (__d1), "=r"(__d2)    \
: "3"(size), "0"(size), "1"(to), "2"(from)          \
: "memory");
} while (0)

```

这段汇编代码在".section .fixup,\"ax\"\n"之前就是常规的内存拷贝操作（基本等同于 x86 上 memcpy 的实现），特殊的地方在于后半段定义的两个 section：".fixup"和"__ex_table"。

在继续这个话题之前，先来想一个问题：为什么要使用 copy_from_user 函数呢？

理论上说，内核空间可以直接使用用户空间传过来的指针，即使要做数据拷贝的动作，也可以使用 memcpy，事实上在没有 MMU 的体系架构上，copy_from_user 最终的实现就是利用了 memcpy。但是对于大多数有 MMU 的平台，情况就有了些变化：用户空间传过来的指针是在虚拟地址空间上的，它所指向的虚拟地址空间很可能还没有真正映射到实际

的物理页面上。但是这又能怎样呢？缺页导致的异常会很透明地被内核予以修复（为缺页的地址空间提交新的物理页面），访问到缺页的指令会继续运行仿佛什么都没有发生一样。但这只是用户空间缺页异常的行为，在内核空间这种缺页异常必须被显式地修复，这是由内核提供的缺页异常处理函数的设计模式决定的，其背后的思想是：在内核态，如果程序试图访问一个尚未被提交物理页面的用户空间地址，内核必须对此保持警惕而不能像用户空间那样毫无察觉。

现在回到 `__copy_user_zeroing` 函数，标号 4 处显然是个内存访问指令，如果搬移（`mov`）的源地址（在用户空间）位于一个尚未被提交物理页面的空间中，将产生缺页异常，内核会调用 `do_page_fault` 函数来处理这个异常，因为异常发生在内核空间，`do_page_fault` 将调用 `search_exception_tables` 在 `"__ex_table"` section 中查找产生异常指令所对应的修复（`fixup`）指令。在 `__copy_user_zeroing` 函数的后半段，在 `"__ex_table"` section 中定义了如下数据：

```
".long 4b,5b\n"
```

其中 4b 对应标号 4 处的指令，5b 对应标号 5 处的指令，是 4b 处指令的修复指令。这样，当标号 4 处发生缺页异常时，系统将调用 `do_page_fault` 提交物理页面，然后跳转到标号 5 的指令处继续运行。

如果在驱动程序中不使用 `copy_from_user` 而用 `memcpy` 来代替，对于上述的情形会产生什么结果呢？当标号 4 处发生缺页异常时，系统在 `"__ex_table"` section 中将找不到修复地址（因为 `memcpy` 没有像 `copy_from_user` 那样定义一个 `"__ex_table"` section），此时 `do_page_fault` 将通过 `no_context` 函数产生 oops，极有可能会看到类似如下信息：

```
BUG: unable to handle kernel paging request at 188be008
```

所以为了确保设备驱动程序的安全，应该使用 `copy_from_user` 函数而不是 `memcpy`。

这里用了 x86 平台作为例子，虽然其他平台上的实现可能会有不同，但是总体上它们的设计思想是一样的。

如果需要将内核空间的数据拷贝到用户空间，可以使用 `copy_to_user` 函数，其背后的设计理念和 `copy_from_user` 完全一样。该函数的定义为：

```
<include/asm-generic/uaccess.h>
-----
static inline long copy_to_user(void __user *to,
                                const void *from, unsigned long n)
{
    might_sleep();
    if (access_ok(VERIFY_WRITE, to, n))
        return __copy_to_user(to, from, n);
    else
        return n;
```

```
}

```

参数*to 是用户空间指针，*from 是内核空间指针，n 是要拷贝的字节数。如果拷贝成功函数返回 0，否则返回尚未被拷贝的字节数。

除了上述的 copy_from_user 和 copy_to_user 函数，在用户空间和内核空间交换数据时还有两个常用的函数：get_user 和 put_user。相对于 copy_from_user 和 copy_to_user，这两个函数主要用来完成一些简单类型变量（char、int、long 等）的拷贝任务，对于一些复合类型的变量，如数据结构或者数组类型，get_user 和 put_user 函数则无法胜任：函数内部将对 ptr 所指向的对象长度进行检查，大部分平台只支持长度为 1,2,4 的变量。

```
<include/asm-generic/uaccess.h>
-----
#define get_user(x, ptr) \
({ \
    might_sleep(); \
    access_ok(VERIFY_READ, ptr, sizeof(*ptr)) ? \
        __get_user(x, ptr) : \
        -EFAULT; \
})

```

get_user 将用户空间 ptr 指向的数据拷贝到内核空间的变量 x 中，其内部实现虽然多为单条汇编指令实现的内存操作（比如 x86 的 MOV 指令、ARM 的 LDR 指令等），但依然会有 ".fixup"和"__ex_table"section。函数如果成功则返回 0，否则返回-EFAULT。

```
<include/asm-generic/uaccess.h>
-----
#define put_user(x, ptr) \
({ \
    might_sleep(); \
    access_ok(VERIFY_WRITE, ptr, sizeof(*ptr)) ? \
        __put_user(x, ptr) : \
        -EFAULT; \
})

```

put_user 用来将内核空间的一个简单类型变量 x 拷贝到 ptr 所指向的用户空间中。函数能自动判断变量的类型，如果成功则返回 0，否则返回-EFAULT。

以下是 get_user 和 put_user 的用法示例：

```
//p 为用户空间指针
int __user *p = ...;
//val 为内核空间的变量
int val;
//将 p 对应的用户空间整数值拷贝到内核空间的 val 变量中
if (get_user(val, p))
    return -EFAULT;

```



```
//将内核空间的整型变量 val 拷贝到 p 对应的用户空间中
val = 10;
if (put_user(val, p))
    return -EFAULT;
```

在设备驱动程序中，其实现的 `ioctl` 函数主体往往都是一个 `switch...case` 结构，下面是一个实际的设备驱动程序实现的 `unlocked_ioctl` 代码片段：

```
static long zl30310_espi_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    ...
    if (_IOC_TYPE(cmd) != ZL30310_IOC_MAGIC)
        return -ENOTTY;

    if (_IOC_DIR(cmd) & _IOC_READ)
        err = !access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));
    if (err == 0 && _IOC_DIR(cmd) & _IOC_WRITE)
        err = !access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd));
    if (err)
        return -EFAULT;

    size = _IOC_SIZE(cmd);

    mutex_lock(&zl30310_espi->buf_lock);
    switch (cmd) {
    case ZL30310_SPI_IOC_RD_MODE:
        retval = __put_user(spi->mode & SPI_MODE_MASK,
                           (__u8 __user *)arg);

        break;

    case ZL30310_SPI_IOC_REG_READ:
        retval = __get_user(addr, (__u8 __user *)arg);
        zl30310_reg_read8(zl30310_espi, addr, &regval);
        if (retval == 0){
            __put_user(regval, (__u8 __user *)arg);
        }
        break;

    case ZL30310_SPI_IOC_REG_WRITE:
        if(__copy_from_user(&regop, (const void *)arg, size)){
            retval = -EFAULT;
            break;
        }
        zl30310_reg_write8(zl30310_espi, regop.addr, regop.regval);
        break;
    }
```

```
default:
    retval = -EFAULT;
    break;
}
...
mutex_unlock(&z130310_espi->buf_lock);
return retval;
}
```

以上是一个比较典型的 `ioctl` 的实现，代码首先用 `_IOC_TYPE` 对 `cmd` 进行初步的检验，之后调用 `access_ok` 来验证用户空间指针的有效性，因为 `access_ok` 在 `ioctl` 函数开始就被调用了，所以之后都是用的 `__put_user`、`__get_user` 和 `__copy_from_user` 这样的形式，否则应该使用 `put_user`、`get_user` 和 `copy_from_user`。因为是 `unlocked_ioctl` 函数，所以驱动程序需要实现自己的互斥机制，上面的例子中使用的是 `mutex_lock` 和 `mutex_unlock`。

`ioctl` 函数的返回值应该用来表示该函数的执行状态，有些书认为可以把它作为在用户空间和内核空间进行数据交换的一种方式，这不是一种好的编码习惯。

7.2 字符设备的 I/O 模型

一个字符设备的主要功能是用来实现 I/O 操作，反映到应用程序中就是进行 `read` 和 `write` 等相关的操作。在对一个设备进行读写操作时，鉴于设备在实际的操作中响应速度各不相同，因此数据并不总是在任何时候都可用：对于读操作来说，也许请求的数据还没有达到设备的缓冲区；对于写操作来说，应用层传递过来的数据也许不能够一下子全部放进设备狭小的缓冲区。此时对于这些读写操作来说，要么放弃等待直接返回一个错误码给上层，要么让发起读写操作的进程进入等待状态直到数据可用为止。

根据不同的需求和使用场景，Linux 内核支持几种不同的 I/O 操作模式，称为字符设备的 I/O 模型，这些模型根据同步与异步、阻塞与非阻塞可以划分为四大类。图 7-2 简单地描述了这种 I/O 模型的分类：

○ 同步阻塞 I/O

这是 I/O 模型中最常用的一种操作。对于这种同步阻塞型 I/O，应用程序执行一个系统调用对设备进行 `read/write` 操作，这种操作会阻塞应用程序直到设备完成 `read/write` 操作或者返回一个错误码。在应用程序阻塞的这段时间里，程序所代表的进程并不消耗 CPU 的时间，因此从这个角度而言，这种操作模式是非常高效的。为了支持设备的这种 I/O 操作模式，设备驱动程序需要实现 `file_operations` 对象中的 `read` 和 `write` 方法。

	阻塞	非阻塞
同步	read/write	read/write (O_NONBLOCK)
异步	poll/select/epoll	AIO

图 7-2 字符设备 I/O 模型

○ 同步非阻塞 I/O

在这种 I/O 操作模式下，设备文件以非阻塞的形式打开（O_NONBLOCK），如果设备不能立即完成用户程序所要求的 I/O 操作，应该返回一个错误码（EAGAIN 或者 EWOULDBLOCK，两者是同一个值）。

○ 异步阻塞 I/O

这种模式的 I/O 操作并不是阻塞在设备的读写操作本身，而是阻塞在某一组设备文件的描述符上，当其中的某些描述符上代表的设备对读写操作已经就绪时，阻塞状态将被解除，用户程序随后可以对这些描述符代表的设备进行读写操作。Linux 的字符设备驱动程序需要实现 file_operations 对象中的 poll 方法以支持这种 I/O 模式。

○ 异步非阻塞 I/O

在这种 I/O 操作模式下，读写操作会立即返回，用户程序的读写请求将被放入一个请求队列中由设备在后台异步完成，当设备完成了本次的读写操作时，将通过信号或者回调函数的方式通知用户程序。需要说明的是，Linux 系统的设备中，块设备和网络设备的 I/O 模型属于异步非阻塞型，对于字符设备而言，极少有驱动程序需要去实现这种模式的 I/O 操作。

字符设备驱动程序应该根据具体的需求实现上面四种 I/O 模型中的部分或全部，本节将讨论驱动程序如何实现这些 I/O 模型。下面先讨论同步模式下的阻塞和非阻塞操作，然后再讨论异步模式的阻塞与非阻塞操作。

7.3 同步阻塞型 I/O

本节讨论驱动程序对同步阻塞型 I/O 方法 read 和 write 实现机制的底层支持，介绍在实现阻塞型 I/O 时内核为设备驱动程序提供的相关设施。这些内核实施的核心设计建立在等待队列的基础之上，在 4.9 节“等待队列”中详细介绍过内核中等待队列的数据结构，下面将在此基础上展开讨论驱动程序实现阻塞型 I/O 时用到的核心函数 wait_event 系列和 wake_up 系列，在这部分讨论中首先介绍使用频率最广的函数，然后再延伸到其他的一些变体函数。

7.3.1 wait_event_interruptible

在 Linux 内核中，宏 wait_event_interruptible 用来将当前调用它的进程睡眠等待在一个 event 上直到进程被唤醒并且需要的 condition 条件为真。睡眠的进程其状态是 TASK_INTERRUPTIBLE 的，这意味着它可以被用户程序所中断而结束，但通常的情况是进程等待的 event 事件发生了，它被唤醒重新加入到调度器的运行队列中等待下一次调度执行。这个宏的定义是：

```
<include/linux/wait.h>
```

```
-----
#define wait_event_interruptible(wq, condition) \
({ \
    int __ret = 0; \
    if (!(condition)) \
        __wait_event_interruptible(wq, condition, __ret); \
    __ret; \
})
```

`wait_event_interrupt` 在 `condition` 条件不为真时将睡眠在一个等待队列 `wq` 上, 所以函数首先判断 `condition` 是否为真, 如果为真, 函数将直接返回, 否则调用它的进程将通过 `__wait_event_interruptible` 最终进入睡眠状态, 后者是操作进程睡眠与否的核心函数:

```
<include/linux/wait.h>
```

```
-----
#define __wait_event_interruptible(wq, condition, ret) \
do { \
    DEFINE_WAIT(__wait); \
    \
    for (;;) { \
        prepare_to_wait(&wq, &__wait, TASK_INTERRUPTIBLE);\
        if (condition) \
            break; \
        if (!signal_pending(current)) { \
            schedule(); \
            continue; \
        } \
        ret = -ERESTARTSYS; \
        break; \
    } \
    finish_wait(&wq, &__wait); \
} while (0)
DEFINE_WAIT(__wait)用来定义一个名为“__wait”的等待队列节点对象:
wait_queue_t __wait = { \
    .private    = current, \
    .func       = autoremove_wake_function, \
    .task_list  = LIST_HEAD_INIT((__wait).task_list), \
};
```

`__wait` 中的 `autoremove_wake_function` 函数在节点上的进程被唤醒时被调用, `private` 指向当前调用 `wait_event_interrupt` 的进程。

接下来是函数的核心结构 `for(;;)` 循环, 首先是调用 `prepare_to_wait` 来完成睡眠前的准备工作, 该函数要做的具体任务是: 1. 清除 `__wait` 节点 `flags` 中的 `WQ_FLAG_EXCLUSIVE` 标志, 该标志在唤醒函数中要用到: `__wait->flags &= ~WQ_FLAG_EXCLUSIVE`; 2. 将 `__wait` 节点加入到等待队列 `wq` 中: `__add_wait_queue(q, wait)`, 该函数将把 `__wait` 节点加入到等待队列中成为头节点后的第一个等待节点, 所以后进来的进程将最先被唤醒; 3. 将当前进程的

状态设置为 `TASK_INTERRUPTIBLE`。

`prepare_to_wait` 之后进程依然在调度器的运行队列中，之后如果 `condition` 条件依然为假并且当前进程也没有等待的信号需要处理，`schedule` 函数将被调用，在那里调度器将把当前进程从它的运行队列中移除¹，`wait_event_interruptible` 的表现形式是阻塞（block）在了 `schedule` 函数上直到进程下次被唤醒并被调度执行。

当进程被唤醒时，`schedule` 函数返回（此时进程状态为 `TASK_RUNNING`，所在的等待节点 `__wait` 已经从 `wq` 中删除），通过 `continue` 继续 `for` 循环直到 `condition` 为真时，才通过 `break` 进入到 `finish_wait`，后者基本是 `prepare_to_wait` 的一个反向操作：重新设置进程状态为 `TASK_RUNNING`，然后将 `__wait` 节点从 `wq` 中删除，这是对 `prepare_to_wait` 所做工作的清理。如果休眠的进程被某个信号所中断，那么该函数将返回 `-ERESTARTSYS`。

当程序需要等待在某一队列中直到某一条件满足时，除了使用 `wait_event_interruptible` 之外，内核还提供了一些变体函数：

`wait_event`

该函数使调用的进程进入等待队列，赋予睡眠进程的状态是 `TASK_UNINTERRUPTIBLE`。该函数与 `wait_event_interruptible` 的区别是，它使睡眠的进程不可被中断，而且当进程被唤醒时也不会检查是否有等待的信号需要处理。

`wait_event_timeout`

调用该函数的进程如果进入睡眠，其状态也是 `TASK_UNINTERRUPTIBLE`，意味着不可被中断，而且当进程被唤醒时也不检查是否有等待的信号需要处理。该函数与 `wait_event` 的区别是，会指定一个时间期限，在指定的时间到达时将返回 0。

`wait_event_interruptible_timeout`

在 `wait_event_interruptible` 函数的基础上加入了时间期限，在指定的时间到达时函数将返回 0。

7.3.2 `wake_up_interruptible`

宏 `wake_up_interruptible` 用来唤醒一个等待队列上的睡眠进程，其调用序列如下：

```
<include/linux/wait.h>
-----
#define wake_up_interruptible(x)  __wake_up(x, TASK_INTERRUPTIBLE, 1, NULL)
```

¹ `schedule` 函数中调用 `deactivate_task` 来将当前任务从运行队列中移除，在多处理器系统中，每个 CPU 都拥有属于自己的运行队列。

```
<kernel/sched.c>
```

```
void __wake_up(wait_queue_head_t *q, unsigned int mode,
               int nr_exclusive, void *key)
{
    unsigned long flags;

    spin_lock_irqsave(&q->lock, flags);
    __wake_up_common(q, mode, nr_exclusive, 0, key);
    spin_unlock_irqrestore(&q->lock, flags);
}
```

```
<kernel/sched.c>
```

```
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
                             int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;

    list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
        unsigned flags = curr->flags;
        if (curr->func(curr, mode, wake_flags, key) &&
            (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}
```

所以对于一个等待队列 `x`，`wake_up_interruptible(x)` 展开后的实际调用是 `__wake_up_common(x, TASK_INTERRUPTIBLE, 1, 0, NULL)`，后者通过 `list_for_each_entry_safe` 对等待队列 `x` 进行遍历，对于遍历过程中的每个等待节点，都会调用该节点上的函数 `func`，前面讨论 `wait_event_interruptible` 时看到 `func` 所指向的函数为 `autoremove_wake_function`，其主要功能是唤醒当前等待节点上的进程（把进程加入调度器的运行队列，进程状态变成 `TASK_RUNNING`）并将等待节点从等待队列中移除，通常情况下函数都会成功返回 1。从上面的代码中也看到，如果想让函数结束遍历，必须满足三个条件：1. 负责唤醒进程的函数 `func` 成功返回；2. 等待节点的 `flags` 成员设置了 `WQ_FLAG_EXCLUSIVE` 标志，这是个排他性的标志，如果设置有该标志，那么唤醒当前节点上的进程后将不会继续唤醒操作；3. `nr_exclusive` 等于 1，`nr_exclusive` 表示允许唤醒的排他性进程的数量。对于条件 1，通常都会满足，如果不满足，那么极大的可能性是因为唤醒时使用的进程状态的标志不对，这点马上会提到。所以在此可以将函数结束继续唤醒队列中的进程的条件简单归纳为：遇到一个排他性唤醒的节点并且当前允许排他性唤醒的进程数量为 1。

因为在 `wait_event_interruptible` 函数调用中，`WQ_FLAG_EXCLUSIVE` 标志是被清除的，这意味着 `wake_up_interruptible` 会试图唤醒等待队列 `x` 每个节点上的进程。

`wake_up_interruptible` 函数在内核中同样有自己的一些变体，它们之间的主要区别除了 `TASK_INTERRUPTIBLE` 和 `TASK_UNINTERRUPTIBLE` 之外，在于每次调用时试图唤醒进程的数量，因为唤醒一个进程不存在 `timeout` 的问题，所以没有类似 `wake_up_timeout` 这样的函数。关于 `wake_up` 系列函数中进程状态的使用，可能会给驱动程序员造成一定的困惑，有必要具体讨论一下。

可以看到 `wake_up_interruptible` 宏定义中用到了 `TASK_INTERRUPTIBLE` 参数，这个参数会在调用等待节点上的 `func`，也就是 `autoremove_wake_function` 函数时用到，实际的代码发生在 `autoremove_wake_function` 调用的 `try_to_wake_up` 函数里：

<kernel/sched.c>

```
static int try_to_wake_up(struct task_struct *p, unsigned int state,
                        int wake_flags)
{
    int success = 0;
    ...
    if (!(p->state & state))
        goto out;
    ...
    success = 1;
    ...
out:
    return success;
}
```

函数用 `p->state & state` 将 `wake_up` 系列函数中的进程状态与要唤醒的进程的状态进行检查，如果 `p->state & state=0` 的话那么唤醒操作返回 0，是一次不成功的操作。因此，`wake_up_interruptible` 只能唤醒通过 `wait_event_interruptible` 睡眠的进程。这里将 `wake_up` 一些常见的变体列在下面，读者通过这些宏定义应该可以明白它们实际要完成的功能：

<include/linux/wait.h>

```
#define wake_up(x)                __wake_up(x, TASK_NORMAL, 1, NULL)
#define wake_up_nr(x, nr)         __wake_up(x, TASK_NORMAL, nr, NULL)
#define wake_up_all(x)            __wake_up(x, TASK_NORMAL, 0, NULL)
#define wake_up_locked(x)         __wake_up_locked((x), TASK_NORMAL)

#define wake_up_interruptible(x)   __wake_up(x, TASK_INTERRUPTIBLE, 1, NULL)
#define wake_up_interruptible_nr(x, nr) __wake_up(x, TASK_INTERRUPTIBLE, nr, NULL)
#define wake_up_interruptible_all(x) __wake_up(x, TASK_INTERRUPTIBLE, 0, NULL)
#define wake_up_interruptible_sync(x) __wake_up_sync((x), TASK_INTERRUPTIBLE, 1)
```

因为 `TASK_NORMAL` 在内核中的定义为 `(TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)`，所以 `wake_up` 可以取代 `wake_up_interruptible`，也可以用来唤醒因 `wait_event` 而睡眠的进程。虽然使用的规则可简单归纳为：要将 `wait_event` 和

`wake_up`, `wait_event_interruptible` 和 `wake_up_interruptible` 分别配对使用, 但是读者如果能对此处提到的各个函数背后的行为机制有个清晰的认识, 相信在实际的代码编写中一定可以灵活运用。

`wake_up_nr` 和 `wake_up_all` 表示可以唤醒的排他性进程的数量, `wake_up_nr` 可以唤醒 `nr` 个这样的进程, `wake_up_all` 则可以唤醒队列中所有的排他性进程, `wake_up` 则只能唤醒一个, 当然对于非排他性节点上的进程, 这些函数都会试图去唤醒它们。对应的 `wake_up_interruptible` 系列函数除了只能唤醒 `TASK_INTERRUPTIBLE` 状态的进程外, 其他的功能和 `wake_up` 系列一样。

其中 `wake_up_locked` 和 `wake_up` 的唯一区别是, `wake_up` 函数内部会使用等待队列的自旋锁, `wake_up_locked` 则不会, 所以如果程序中要使用 `wake_up_locked`, 那么需要自己考虑加锁的问题。`wake_up_interruptible_sync` 则用来保证调用它的进程不会被唤醒的进程所抢占而调度出处理器²。

通过上面对 `wait_event_interruptible` 函数的讨论, 现在看看驱动程序中如何利用它实现阻塞的 I/O 操作:

(1) 驱动程序首先需要定义一个等待队列的头节点, 可以通过 `DECLARE_WAIT_QUEUE_HEAD` 宏来完成静态定义和初始化, 比如:

```
static DECLARE_WAIT_QUEUE_HEAD(demo_rd_wq);
```

如果要在程序运行期间初始化一个头节点, 可以使用 `init_waitqueue_head` 函数, 比如:

```
static wait_queue_head_t    demo_rd_wq;
...
init_waitqueue_head(&demo_rd_wq);
```

总之, 驱动程序需要先建立一个属于自己的等待队列。

(2) 在阻塞 I/O 函数的实现中调用 `wait_event_interruptible` 等待数据可用, 比如在 `read` 函数中:

```
static ssize_t    demo_read (struct file * filp, char __user * buf, size_t count, loff_t *f_pos){
    ...
    wait_event_interruptible(wq, test_bit(RD_DATA_READY, &demodev_buf->state));
    ...
}
```

(3) 实现一个 `demo_read` 等待的数据可用时唤醒操作, 一般多在驱动程序的中断处理例程里:

```
irqreturn_t    demo_isr(int irq, void * dev_id){
    ...
```

² 在 `try_to_wake_up` 函数中, 会调用 `check_preempt_curr` 来检查新被唤醒的进程是否可以抢占当前正运行的进程。


```

        set_bit(RD_DATA_READY, &demodev_buf->state);
        wake_up_interruptible(demo_rd_wq);
        ...
    }

```

以上讨论了内核提供的 `wait_event` 和 `wake_up` 函数，内核在这些函数中为等待队列提供了默认的操作模式，已可满足大多数设备驱动程序的要求。不过如果必要，程序员也可以按照 `wait_event` 和 `wake_up` 函数的实现原理来构建自己的睡眠和唤醒函数，比如一个典型的睡眠序列可能是下面这个样子：

```

//定义一个等待节点 wait
DECLARE_WAITQUEUE(wait, current);
//设置进程状态
set_current_state(TASK_UNINTERRUPTIBLE);
//将节点加入等待队列
add_wait_queue(&demo_rd_wq, &wait);
//调用 schedule 函数让进程进入睡眠状态
schedule();
//唤醒以后将等待节点从队列中移除
remove_wait_queue(&demo_rd_wq, &wait);

```

使用内核提供的函数还是构建自己的睡眠和唤醒序列，这里没有既定的规则可以遵循，总之根据实际情况灵活处理就好了。笔者的建议是，应该首先使用内核提供的函数，如果它们的确无法满足需要，那么再构造自己的睡眠和唤醒函数。

7.4 同步非阻塞型 I/O

前面看到，驱动程序在实现 `read/write` 函数时，如果所需要的数据暂时不可用，那么默认的行为是让执行这些操作的进程休眠（阻塞）。但是驱动程序员必须意识到，有时候从用户空间应用程序的角度来说，可能并不希望让 `read/write` 进入阻塞状态，不管用户这样做是出于什么原因，作为驱动程序，必须能支持用户的这一要求。用户的要求以 `O_NONBLOCK` 标志的形式传达到驱动程序中，如果用户希望这是一个不能阻塞的操作，他可能会在 `open` 这个文件时指定 `O_NONBLOCK` 或是在 `read/write` 操作之前在指定的文件描述符上通过 `fcntl` 设置 `O_NONBLOCK` 标志，无论如何这种情况下驱动程序可以通过传递到 `read/write` 的参数 `struct file *filp` 来获得这一信息：在用户指定了 `O_NONBLOCK` 的情形下，`filp->f_flags & O_NONBLOCK` 的结果为真。

所以，一个功能完整的 `read/write` 应该在它的执行流程中检测 `filp->f_flags` 上的 `O_NONBLOCK` 位有没有被设置，如果设置了的话操作可以简单地返回一个错误码 `-EAGAIN` 而宣告结束，否则应该按照默认的阻塞方式来进行。这里虽然只用 `read/write` 作为描述的例子，但事实上字符设备驱动程序可以根据需要在所有可以获得 `filp->f_flags` 的操

作函数中执行相应的策略。

7.5 异步阻塞型 I/O

本节讨论驱动程序中如何支持设备的异步阻塞型 I/O 操作模式，反映到 `file_operations` 对象上，就是讨论在设备驱动程序中如何实现 `poll` 方法。专注于内核空间的设备驱动程序员也许不太关注应用层的操作，但是至少要了解一下驱动程序提供的 `poll` 函数如何在用户空间里使用。相对于驱动程序，用户空间的程序在 `poll` 这一类的操作上也许有更广的选择，除了原生态的 `poll` 调用，还有 `select` 和 `epoll`。对驱动程序员来说，一个比较好的消息是这些调用最终到驱动程序中只由 `poll` 函数来实现。

在讨论驱动程序的 `poll` 函数实现之前，不妨先简单看一下应用程序使用 `poll`、`select` 和 `epoll` 要完成什么功能。在用户空间，`poll`、`select` 和 `epoll` 的原型声明分别是：

```
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
int select(int nfds, fd_set *readset, fd_set *writeset, fd_set *exceptset, struct timeval *timeout);

int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

在用户空间，应用程序将要操作的一组文件的描述符加入到一个集合中，然后在这个集合的基础上使用这些函数来监控其中的每个文件描述符：倘若集合中的每个文件目前都不可进行读取写入操作，进程也许会因此而被阻塞，直到该集合中的任一文件可读或者可写。`poll` 和 `select` 函数本质上是一样的，鉴于 `poll` 函数较之 `select` 函数更易于使用，所以本章会以 `poll` API 函数为讨论的重点。`poll` API 的第一个参数是类型为 `struct pollfd` 的指针，在用户空间，其定义为：

```
<sys/poll.h>
-----
struct pollfd{
    int fd; //文件描述符
    short events; //等待的事件，通常是一组宏常数的组合，每个常数表示一种事件
    short revent; //驱动程序中实际发生的事件
};
```

`poll` API 的第二个参数表示等待的文件描述符的个数（代码层面就是 `struct pollfd` 对象的个数），第三个参数是超时期限，单位是毫秒。

显然，集合中的这些文件描述符所对应的底层设备的驱动程序需要支持应用程序的这种机制，更具体地，驱动程序需要在它们的 `file_operations` 结构中实现自己的 `poll` 函数：

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

虽然设备驱动程序员也许不需要了解应用程序和驱动程序在 poll 机制上进行互动的所有细节，然而了解内核赋予的这种整体框架上的设计思想有助于驱动程序员在设计的工作中提供更完美的解决方案。上述三种应用空间的函数原型的声明和使用方法以及性能或许不同，但总体上说，其背后蕴藏的设计思想非常相似（epoll 可能要特殊一些，可以认为是对前两种方法在性能上的一种改进）。这里不妨以 poll 函数为例，看看从用户空间到驱动程序间的作业流程。直觉上看来，背后少不了等待队列的支持，问题是内核如何把驱动程序与应用程序巧妙地沟通起来。

应用程序的 poll 函数将通过系统调用 sys_poll 进入内核空间：

```
<fs/select.c>
```

```
int sys_poll(struct pollfd __user *ufds, unsigned int nfds, long timeout_msecs);
```

该系统调用的函数原型和用户空间的 poll 一样，函数内部主要调用 do_sys_poll 来实现其核心功能，后者的主体框架在内核源码中看起来如下：

```
<fs/select.c>
```

```
int do_sys_poll(struct pollfd __user *ufds, unsigned int nfds, struct timespec *end_time)
{
    ...
    poll_initwait(&table);
    fdcount = do_poll(nfds, head, &table, end_time);
    poll_freewait(&table);
    ...
}
```

其中 table 变量是个关键元素，因为这个变量中的成员 poll_table pt 将被传递给驱动程序，所以有必要看看 table 的类型定义和 poll_initwait 的实现。table 是个 struct poll_wqueues 类型的变量，后者的定义为：

```
<include/linux/poll.h>
```

```
struct poll_wqueues {
    poll_table pt;
    struct poll_table_page *table;
    struct task_struct *polling_task;
    int triggered;
    int error;
    int inline_index;
    struct poll_table_entry inline_entries[N_INLINE_POLL_ENTRIES];
};
```

struct poll_wqueues 对象的初始化发生在 poll_initwait 函数中：

```
<fs/select.c>
```

```
void poll_initwait(struct poll_wqueues *pwq)
```

```

{
    pwq->pt.qproc = __pollwait;
    pwq->pt.key    = ~0UL;
    pwq->polling_task = current;
    pwq->triggered = 0;
    pwq->error = 0;
    pwq->table = NULL;
    pwq->inline_index = 0;
}

```

`poll_initwait` 中除了初始化 `poll_wqueues` 中的 `poll_table` 成员 `pt`，另一个比较重要的步骤是把当前进程的 `task_struct` 对象指针 `current` 放入到 `pwq` 的 `polling_task` 中，唤醒操作将会用这个变量找到需要唤醒的进程。

`poll_freewait` 函数调用只做一些资源释放类的辅助工作。`do_sys_poll` 的核心实现是在 `do_poll` 中，该函数可能会被阻塞，当其返回时，其返回值 `fdcount` 将传递到用户空间用以指示本次操作的状态：

- `fdcount>0` 表明集合中有 `fdcount` 个文件描述符可以进行读或者写。
- `fdcount=0` 表明集合中所有文件描述符尚无状态变化时，`timeout` 指定的时间到，函数超时。
- `fdcount<0` 表明函数调用失败，错误原因将写入 `errno`。

`do_poll` 函数的框架结构可以看成如下形式：

```

static int do_poll(unsigned int nfd, struct poll_list *list,
                  struct poll_wqueues *wait, struct timespec *end_time)
{
    poll_table* pt = &wait->pt;

    if (end_time && !end_time->tv_sec && !end_time->tv_nsec) {
        pt = NULL;
        timed_out = 1;
    }

    for (;;) {
        struct poll_list *walk;
        for (walk = list; walk != NULL; walk = walk->next) {
            struct pollfd * pfd, * pfd_end;
            pfd = walk->entries;
            pfd_end = pfd + walk->len;
            for (; pfd != pfd_end; pfd++) {
                if (do_pollfd(pfd, pt)) {
                    count++;
                    pt = NULL;
                }
            }
        }
    }
}

```

```

    }
    }
}
pt = NULL;
...
if (count || timed_out)
    break;
if (!poll_schedule_timeout(wait, TASK_INTERRUPTIBLE, to, slack))
    timed_out = 1;
}
return count;
}

```

其核心是一 `for(;;)` 循环，在该循环中首先会对文件描述符集合中的每个描述符调用 `do_pollfd`，同时传入一个 `struct pollfd` 类型的指针，在内核空间 `struct pollfd` 的定义为：

```
<include/asm/poll.h>
```

```

struct pollfd {
    int fd;
    short events;
    short revents;
};

```

`do_pollfd` 的主要功能是根据当前的 `fd` 找到对应的 `struct file *filp` 对象，然后调用 `poll` 例程：

```

fd = pollfd->fd;
file = fget_light(fd, &fput_needed);
mask = file->f_op->poll(file, pwait);
pollfd->revents = mask;

```

这里需要注意 `mask` 的使用，它由设备驱动程序中的 `poll` 例程返回，用来记录驱动程序中发生的事件，然后 `do_pollfd` 将其记录在 `pollfd->revents` 中，并最终返回到用户空间。

根据 `do_poll` 的框架可知：首先每个 `fd` 所对应的设备驱动程序在自己实现的 `poll` 例程中不应该睡眠，应用程序调用的 `poll` 只会睡眠在 `poll_schedule_timeout` 这里；其次如果不考虑超时的因素，当前进程从 `poll_schedule_timeout` 中醒来应该是拜驱动程序中的 `poll` 例程所赐，因为只有驱动程序才知道自己管理的设备什么时候数据就绪，因此驱动程序少不了数据就绪后的唤醒环节。另一点要注意的是，调用 `select` 的用户进程以 `TASK_INTERRUPTIBLE` 的状态进入睡眠队列，这是一种被称为可中断的睡眠状态。

所以我们看到驱动程序对 `poll` 特性的支持实际上可分为两部分：第一部分是 `poll` 例程本身，那里它将某一等待节点对象加入到自己管理的等待队列中；第二部分是数据就绪后的唤醒操作。同时我们也看到核心的 `for(;;)` 循环有一条 `break` 语句，表明或者是用户程序调用 `poll` 函数时设定 `timeout` 参数为 0，或者是对当前设备执行 `poll` 操作时设备中恰好有就绪的数据，前一种表明用户不希望对 `poll` 的调用出现阻塞，用户只需要查询一下文件描述符集是否有

些 fd 已经就绪，后一种情况表明文件描述符集合中至少已经有一个设备文件当前是可以进行无阻塞操作的，所以这两种情况都不应该进入 poll_schedule_timeout 让进程睡眠。

现在正是讨论驱动程序的 poll 例程如何实现的好时机，在 do_poll 中，对于每个 fd 调用 do_pollfd(pfd, pt) 时都会传入一个 pt 参数，这个变量在 do_poll 一开始就给出了定义：

```
poll_table* pt = &wait->pt;
```

可见 pt 是个 poll_table 型的指针，而且对于每个 fd 的调用，都会将该指针传入。驱动程序在 poll 例程中的关键调用是 poll_wait：

```
<include/linux/poll.h>
```

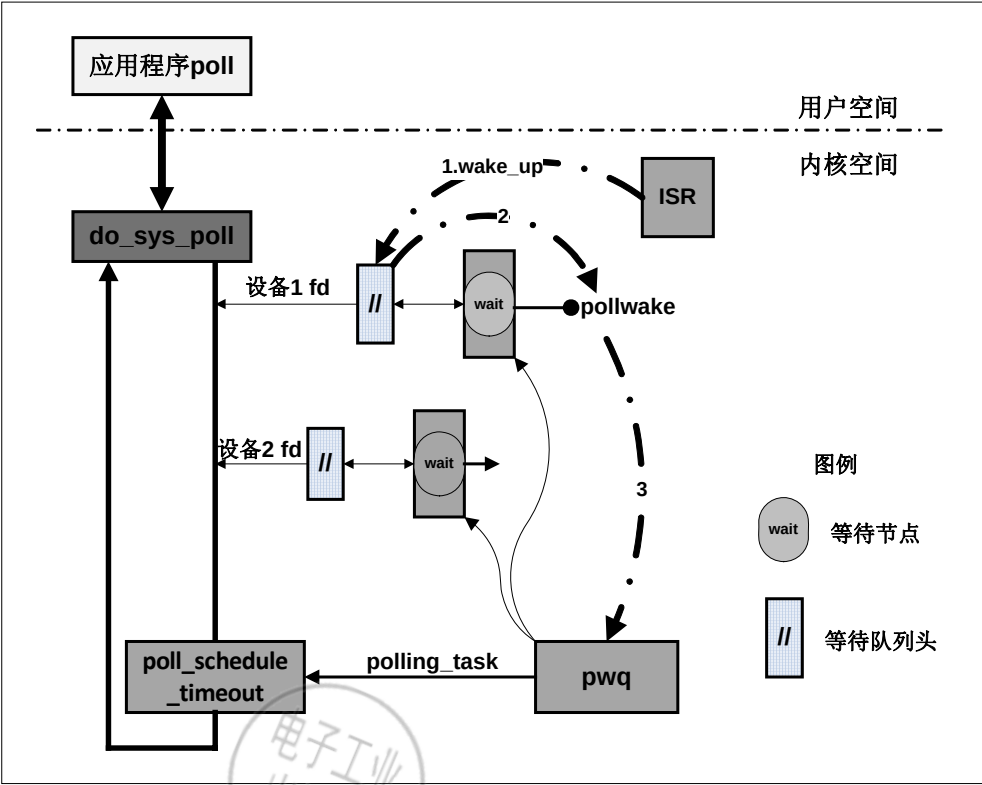
```
static inline void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
{
    if (p && wait_address)
        __pollwait (filp, wait_address, p); //源码中为 p->qproc(filp, wait_address, p);
}
```

```
</fs/select.c>
```

```
static void __pollwait(struct file *filp, wait_queue_head_t *wait_address,
                      poll_table *p)
{
    struct poll_wqueues *pwq = container_of(p, struct poll_wqueues, pt);
    struct poll_table_entry *entry = poll_get_entry(pwq);
    if (!entry)
        return;
    get_file(filp);
    entry->filp = filp;
    entry->wait_address = wait_address;
    entry->key = p->key;
    init_waitqueue_func_entry(&entry->wait, pollwake);
    entry->wait.private = pwq;
    add_wait_queue(wait_address, &entry->wait);
}
```

函数的大意是产生一个等待节点 entry->wait，该节点上的唤醒函数为 pollwake，然后将其加入等待队列 wait_address 中，后者是由各自的设备驱动程序维护的等待队列。这里对 poll_table 指针 p 的使用是，通过 p 获得 pwq 指针，然后从 pwq 管理的数据结构上获得等待节点所在的空间。注意，如果调用 poll_wait 时，传入的 poll_table 参数指针为 NULL，则该函数不会进行任何操作，当应用程序调用 poll 函数时，如果指明 timeout=0，则表明用户不希望在该函数进行任何等待，所以这种情况下驱动程序无须将任何等待节点加入自己的等待队列，do_poll 函数在一开始就针对这种情况赋予了 poll_table 参数一个空指针。

以上流程大体如图 7-3 所示：



对于一个设备驱动而言，为了实现自己的 poll 例程，需要构造自己的等待队列，然后通过调用 poll_wait 将一个等待节点加入到自己的等待队列中，poll_wait 函数内部负责从 pwq 对象中申请容纳等待节点的空间并对其初始化，其中的唤醒函数为 pollwake。等待节点对象来自于内核，因此内核可以在随后的 poll_freewait 函数中将这些等待节点清除掉。

图 7-3 中的虚线部分是唤醒操作的执行路径，唤醒源自于设备驱动程序发现自己设备的数据缓冲区已经可以使用。图中虽然以 ISR 来展示唤醒操作的发起者，然而在实际的代码中，唤醒函数的发起者可能来自任何使数据就绪的执行路径。不管怎样，驱动程序将在自己管理的等待队列上调用前面提到的 wake_up 系列函数，这将导致等待节点中的 pollwake 被调用：

```
</fs/select.c>
static int pollwake(wait_queue_t *wait, unsigned mode, int sync, void *key)
{
    struct poll_table_entry *entry;

    entry = container_of(wait, struct poll_table_entry, wait);
    if (key && !((unsigned long)key & entry->key))
        return 0;
    return __pollwake(wait, mode, sync, key);
}
```

```
}

```

唤醒进程的操作发生在__pollwake 中，在那里它将调用 try_to_wake_up 去唤醒进程，虽然目前的__pollwake 代码在实现唤醒时有点不够顺畅³，但框架应该比较清楚，只需要找到要唤醒进程的 task_struct 对象的指针即可，前面在 poll_initwait 函数中看到，pwq->polling_task 保存了该指针，所以就有了下面的实现方法：

```
struct poll_wqueues *pwq = wait->private;
task_struct *p = pwq->polling_task;
try_to_wake_up(p...);
```

最后一句 try_to_wake_up 将试图唤醒睡眠在 poll_schedule_timeout()上的进程，这将导致用户态程序从 select 等 API 函数的调用中返回，或者是因为指定的时间到期，或者是因为文件描述符集合中有某些描述符就绪了。

驱动程序的 poll 例程除了报告设备的数据是否就绪外，另一个作用是如果数据就绪，那么应该向用户态程序报告是哪种情况的就绪，读或者写或者其他。这些状态在内核中以掩码的形式存在，内核为此定义了一些状态位，其中一些常见的状态定义如下：

```
<include/asm-generic/poll.h>
```

```
-----
#define POLLIN      0x0001
#define POLLPRI     0x0002
#define POLLOUT     0x0004
#define POLLERR     0x0008
#define POLLHUP     0x0010
#define POLLRDNORM  0x0040
#define POLLWRNORM  0x0100
```

POLLIN

非高优先级的数据（即带外数据 out-of_band）可以被无阻塞地读取。

POLLPRI

高优先级的数据（即带外数据 out-of_band）可以被无阻塞地读取。

POLLOUT

数据可以无阻塞地写入。

POLLERR

设备发生了错误。

³ 通过一个 dummy_wait 节点来实现唤醒操作，具体的原因在__pollwake 函数的注释中讲得很明白。

POLLHUP

与设备的链接已经断开。

POLLRDNORM

正常数据可以无阻塞地读取。

POLLWRNORM

正常数据可以无阻塞地写入。

更多这方面的细节读者可参考：<http://www.opengroup.org/onlinepubs/009695399/functions/poll.html>。

至此就讨论完了 Linux 设备驱动程序的 poll 例程的实现机制，为了加深读者的印象，下面给出一个驱动程序实现的 poll 例程的示例：

```

//定义一个用于读取的等待队列 demo_inq
static DECLARE_WAIT_QUEUE_HEAD(demo_inq);

//驱动程序实现的 poll 例程
unsigned int demo_poll (struct file * filp, struct poll_table_struct * wait)
{
    struct demo_buf_list *list = filp->private_data;
    //初始化 mask 为 0, 表明目前关于设备的数据的状态没有发生任何变化
    unsigned int mask = 0;
    ...
    //调用 poll_wait 将来自内核中的一个等待节点加入 demo_inq 队列
    poll_wait(filp, &demo_inq, wait);
    //判断缓冲区是否可读
    if (list->head != list->tail)
        mask |= POLLIN | POLLRDNORM;
    return mask;
}

//设备驱动程序实现的中断处理例程
irqreturn_t demo_isr(int irq, void * dev_id)
{
    ...
    //如果缓冲区可读, 调用 wake_up 函数唤醒阻塞在 poll 上的进程
    wake_up_interruptible(&demo_inq);
    ...
}

```

7.6 异步非阻塞型 I/O

字符设备驱动程序如果需要支持设备的这种异步非阻塞型的 I/O 操作模式，需要实现

file_operations 对象中的 aio_read 和 aio_write 方法，这两个方法的原型为：

```
<include/linux/fs.h>
-----
ssize_t (*aio_read)(struct kiocb *, const struct iovec *, unsigned long, loff_t);
ssize_t (*aio_write)(struct kiocb *, const struct iovec *, unsigned long, loff_t);
```

函数的第一个参数是个 struct kiocb 类型的指针，用来封装一个读写请求的完整上下文，因为其结构体比较复杂，这里只摘录那些与异步 I/O 实现原理密切相关的成员：

```
<include/linux/aio.h>
-----
struct kiocb {
    ...
    struct file      *ki_filp;
    ...
    union {
        void __user   *user;
        struct task_struct *tsk;
    } ki_obj;
    ...
    unsigned short   ki_opcode;
    size_t            ki_nbytes; /* copy of iocb->aio_nbytes */
    char              __user *ki_buf; /* remaining iocb->aio_buf */
    size_t            ki_left; /* remaining bytes */
    ...
}
```

上面列出的数据结构 struct kiocb 中的成员大体上可以分成三部分：一个是文件相关的 ki_filp；一个是与发出异步 I/O 请求的进程相关的 ki_obj；还有一个是与当前 I/O 请求本身相关数据存储空间、字节数与偏移量等一些成员。

用户空间使用异步 I/O 有两种方式：一是使用异步 I/O 的 API 函数，比如 aio_read、aio_write 和 aio_error 等，这些函数是标准的 POSIX 库函数；二是使用 Linux 的系统调用，比如 io_setup、io_submit 和 io_destroy 等。

鉴于字符类设备驱动程序支持这种异步 I/O 的情况非常少见，而且我们将在本书块设备驱动程序一章中讨论到类似的异步 I/O 操作的细节，虽然那里的讨论与字符设备相比会有些差异，但是两者的背后的设计思想有不少相似之处，所以本书此处不再仔细分析 Linux 内核关于 AIO 的实现机制。

7.7 驱动程序的 fsync 例程

fsync 用来同步设备的写入操作，考虑将一块数据写入到 U 盘的操作，如果使用 write 函数，

函数返回后只能保证数据被写入到驱动程序或者内核管理的数据缓存中，而无法保证数据被真正写入到 U 盘的存储块里。但是 `fsync` 可以做到这一点，函数在数据没有真正写到 U 盘的存储块里时不会返回，若返回则意味着要么设备在写入过程中发生错误，要么数据已经写入到了设备的存储块中。

显然大量的工作在驱动程序这边，驱动程序需要针对不同的设备实现 `write` 和 `fsync` 例程，以满足上层应用程序调用两者时所期望的语义。对于字符设备而言，大部分驱动程序都没有实现这个例程，只是简单地将它们的 `struct file_operations` 对象的 `fsync` 指针赋予了一个 `NULL` 值；而对于块设备而言，总是使用通用的 `block_fsync` 函数作为 `fsync` 例程的实现。

7.8 fsync 例程

本节讨论设备驱动程序 `struct file_operations` 的另一个例程 `fsync`：

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

前面看到基于驱动程序 `poll` 例程之上的应用层面的三个函数 `poll`、`select` 和 `epoll`，它们在与设备驱动程序沟通数据是否就绪时，本质上是采用了轮询的方式：应用程序在一组由设备文件描述符的集合上调用 `poll`，由此获得设备可否进行无阻塞操作的信息。其实除了轮询的方式，应用程序与设备驱动程序的沟通还有一种类似中断的方式：当设备中的数据就绪时，作为通知的方式，设备驱动程序会给应用程序发送一个信号。驱动程序对这种模式的支持体现在其 `struct file_operations` 对象的 `fsync` 例程，从名字看，是种异步的操作模式。

先看看应用程序怎么操作，然后再考虑驱动程序如何配合。应用程序要做的有两件事：第一步，通过 `fcntl` 函数的 `F_SETOWN` 命令将进程的 ID 号告诉驱动程序，这样当驱动程序发现设备的数据就绪时才知道要通知哪个进程；第二步，通过 `fcntl` 函数的 `F_SETFL` 命令设置 `FASYNC` 标志让驱动程序启动异步通知机制。这两步都通过 `fcntl` 函数来完成，下面先简单探讨一下 `fcntl` 对这两个命令的内部处理流程，然后再过渡到驱动程序那里。

`fcntl` 通过系统调用 `sys_fcntl` 与内核空间交互，后者的核心调用是 `do_fcntl`，其实现框架如下：

```
static long do_fcntl(int fd, unsigned int cmd, unsigned long arg,
                    struct file *filp)
{
    long err = -EINVAL;

    switch (cmd) {
        ...
        case F_SETFL:
            err = setfl(fd, filp, arg);
            break;
```

```

    case F_SETOWN:
        err = f_setown(filp, arg, 1);
        break;
    ...
    default:
        break;
    }
    return err;
}

```

这里略去了很多其他的 `fcntl` 命令，只保留了跟目前讨论相关的 `F_SETFL` 和 `F_SETOWN` 命令。这两个命令对应的处理函数都比较直白，`f_setown` 函数将要通知进程的 ID 相关的信息记录在 `filp->f_owner` 中，驱动程序方面无须对此作出回应，`setfl` 则会在内部直接调用到驱动程序提供的 `fasync` 例程：

```

<fs/fcntl.c>
-----
static int setfl(int fd, struct file * filp, unsigned long arg)
{
    ...
    error = filp->f_op->fasync(fd, filp, (arg & FASYNC) != 0);
    ...
}

```

了解了应用程序的动作，下面开始讨论驱动程序如何在它的 `fasync` 例程中提供相应的支持。驱动程序在其 `fasync` 例程中需要 `fasync_helper` 和 `kill_fasync` 两个函数：前者主要将当前要通知的进程加入一个链表或者从链表中移除，这取决于应用程序调用 `fcntl` 时是否设置了 `FASYNC` 标志；而 `kill_fasync` 则在设备中的某一事件发生时负责通知链表中所有相关的进程。下面仔细考察这两个函数的内部实现机制。

```

<fs/fcntl.c>
-----
int fasync_helper(int fd, struct file * filp, int on, struct fasync_struct **fapp)
{
    if (!on)
        return fasync_remove_entry(filp, fapp);
    return fasync_add_entry(fd, filp, fapp);
}

```

关于函数的参数列表，需要关注一下 `int on` 和 `struct fasync_struct **fapp`。对于 `int on`，在前面刚提到的 `setfl` 函数中，传递给它的是一个条件表达式 `(arg & FASYNC) != 0`，这意味着如果应用程序在调用 `fcntl` 时，对于 `F_SETFL` 命令使用的参数 `arg` 设置了 `FASYNC`，那么 `(arg & FASYNC) != 0` 结果为 1，所以 `fasync_helper` 中的参数 `on` 将为 1，这表明应用程序正在启用驱动程序的异步通知机制；反之，若对 `fcntl` 函数使用 `F_SETFL` 命令时清除了 `FASYNC` 标志，将导致驱动程序的 `fasync` 例程关闭异步通知特性。

所以 `fasync_helper` 的主要功能是维护一个需要通知的进程链表 `fapp`，如果应用程序需要获得异步通知的能力，那么需要通过 `fcntl` 的 `F_SETFL` 命令设置 `FASYNC` 标志，如果设置了该标志，驱动程序的 `fasync` 例程在调用 `fasync_helper` 时将用 `fasync_add_entry` 将需要通知的进程加入到驱动程序维护的一个链表中，否则调用 `fasync_remove_entry` 将其从链表中移除。

如图 7-4 所示，驱动程序为实现 `fasync` 例程，需要维护一个 `struct fasync_struct` 类型的链表，链表中的每个节点对象代表着一个需要通知的进程，进程的 ID 信息存放在节点对象的 `fa_file->f_owner` 中。在链表中增加节点或者更新节点的操作在内核中由 `fasync_add_entry` 完成，而从链表中删除节点则由 `fasync_remove_entry` 函数完成。图中展示的情景是有两个用户进程需要得到设备驱动程序的异步通知。内核在实现 `fasync_helper` 函数时，对一个新加入的 `struct fasync_struct` 对象节点，`fasync_helper` 会将其放到链表的头部，这意味着后调用 `fcntl(fd, F_SETFL, FASYNC)` 的应用程序反而会先得到通知。

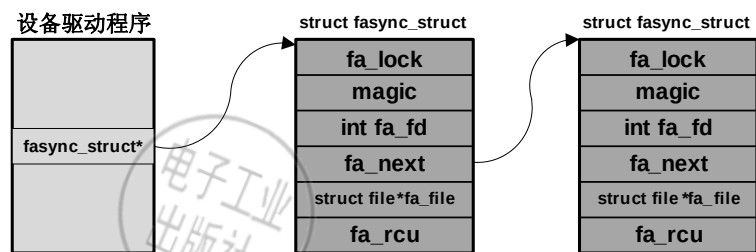


图 7-4 驱动程序用 `fasync_helper` 维护的 `fasync_struct` 链表

现在，驱动程序已经将需要通知的进程所在节点加入了 `fasync` 链表。当需要的条件满足时，比如进程所请求的数据已经就绪，驱动程序需要向 `fasync` 链表中的每个等待通知的进程发送通知信号。内核为此向驱动程序提供了一个 `kill_fasync` 函数，用来发送通知信号。`kill_fasync` 函数定义如下：

```
<fs/fcntl.c>
-----
void kill_fasync(struct fasync_struct **fp, int sig, int band)
{
    /* First a quick test without locking; usually
     * the list is empty.
     */
    if (*fp) {
        rcu_read_lock();
        kill_fasync_rcu(rcu_dereference(*fp), sig, band);
        rcu_read_unlock();
    }
}
```

其实质性的操作发生在 `kill_fasync_rcu` 函数中，后者的实现如下：

```

<fs/fcntl.c>
static void kill_fasync_rcu(struct fasync_struct *fa, int sig, int band)
{
    while (fa) {
        struct fown_struct *fown;
        unsigned long flags;

        if (fa->magic != FASYNC_MAGIC) {
            printk(KERN_ERR "kill_fasync: bad magic number in "
                    "fasync_struct!\n");
            return;
        }
        spin_lock_irqsave(&fa->fa_lock, flags);
        if (fa->fa_file) {
            fown = &fa->fa_file->f_owner;
            /* Don't send SIGURG to processes which have not set a
               queued signal: SIGURG has its own default signalling
               mechanism. */
            if (!(sig == SIGURG && fown->signal == 0))
                send_sigio(fown, fa->fa_fd, band);
        }
        spin_unlock_irqrestore(&fa->fa_lock, flags);
        fa = rcu_dereference(fa->fa_next);
    }
}

```

函数的思想很明确，通过 while 循环遍历 fasync 链表，对每个进程调用 send_sigio 来向其发送信号（SIGIO）以通知进程。

现在从驱动程序的角度出发，总结一下对 fasync 例程的支持。首先，驱动程序需要定义一个 struct fasync_struct 类型的指针，当用户态程序调用 fcntl 用 F_SETFL 命令来设置或者清除 FASYNC 标志时，驱动程序应该在其 fasync 例程中调用内核提供的 fasync_helper 函数在 struct fasync_struct 指针所指向的链表中增加或者删除一个节点，每个节点代表一个需要通知的进程。其次，当进程所需要的数据就绪或关注的某个事件发生时，驱动程序负责向其维护的 fasync_struct 链表中的每个进程发送通知信号，设备驱动程序通过调用内核提供的另一个函数 kill_fasync 来完成信号发送任务。

下面用一个具体的例子来展示设备驱动程序如何实现 fasync 方法，以及应用程序如何得到来自设备驱动程序的异步通知。这个例子同时也展示了 sysfs 文件系统在驱动程序中的用法，以及通过 Linux 设备模型来创建设备节点及其他一些特性（这个看起来很简单的内核模块其实体现了设备驱动程序中一些比较重要且典型的特征）。

首先是设备驱动程序的代码，在代码中，我们将 Linux 设备模型中的一些概念融入其中（本书第9章会详细讨论 Linux 的设备驱动模型，不过读者可以在这里先热身），这样我们可

以动态创建一个设备节点而无须再手动地使用 `mknod` 命令，同时代码中还创建了一个 `sysfs` 文件接口，这使得我们可以直接操控设备驱动程序中的一些数据而不必采用 `ioctl` 的方式，也许这就是设备驱动模型给我们带来的好处吧。

<fasync_demo.c>

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <asm/signal.h>
#include <asm/siginfo.h>

static struct cdev *pcdev;
static dev_t ndev;
static struct class *fa_cls;
static struct device *fadev;

static unsigned long flag = 0;
static struct fasync_struct *sigio_list;

static ssize_t read_flag(struct device *dev, struct device_attribute *attr, char *buf)
{
    size_t count = 0;
    count += sprintf(&buf[count], "%lu\n", flag);

    return count;
}

static ssize_t write_flag(struct device *dev, struct device_attribute *attr,
                        const char *buf, size_t count)
{
    flag = buf[0] - '0';
    //给所有以 FASYNC 标志调用 fcntl 的应用程序发送信号
    kill_fasync(&sigio_list, SIGIO, POLL_IN);
    return count;
}

static struct device_attribute flag_attr =
    __ATTR(flag, S_IRUGO | S_IWUSR, read_flag, write_flag);

static int fa_open(struct inode *inode, struct file *flp)
{
    return 0;
}
```

```
static int fa_async(int fd, struct file *filp, int onflag)
{
    //将需要通知的进程加入 sigio_list 链表或者从链表中移除
    return fasync_helper(fd, filp, onflag, &sigio_list);
}

static struct file_operations ops = {
    .owner = THIS_MODULE,
    .open = fa_open,
    .fasync = fa_async,
};

static int fa_init(void)
{
    int ret = 0;

    ret = alloc_chrdev_region(&ndev, 0, 1, "fa_dev");
    if(ret < 0)
        return ret;

    pcdev = cdev_alloc();
    cdev_init(pcdev, &ops);
    pcdev->owner = THIS_MODULE;
    cdev_add(pcdev, ndev, 1);

    fa_cls = class_create(THIS_MODULE, "fa_dev");
    if(IS_ERR(fa_cls))
        return PTR_ERR(fa_cls);
    fadev = device_create(fa_cls, NULL, ndev, NULL, "fa_dev");
    if(IS_ERR(fadev))
        return PTR_ERR(fadev);

    //在 sysfs 文件系统中创建一个名为 "flag" 的文件
    ret = device_create_file(fadev, &flag_attr);

    return ret;
}

static void fa_exit(void)
{
    device_remove_file(fadev, &flag_attr);
    device_destroy(fa_cls, ndev);
    class_destroy(fa_cls);
    cdev_del(pcdev);
    unregister_chrdev_region(ndev, 1);
}
```



```

module_init(fa_init);
module_exit(fa_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("dennis chen @ AMDLinuxFGL");
MODULE_DESCRIPTION("A simple character device driver to demo the implementation of fasync
method");

```

程序在内核版本 2.6.39 的 Linux 系统上编译通过。Makefile 如下：

```

obj-m := fasync_demo.o
KERNELDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
    rm -f *.o *.ko *.mod.*

```

在给出应用程序的代码前，不妨就此例看看单纯的字符设备驱动程序和加入了设备驱动模型特性之间的关系，虽然这是第九章要讨论的主题，但是先大体上了解一下总没有错，不感兴趣的读者也可以直接跳过，大家互不干涉。

可以看到，在模块初始化函数 `fa_init` 中，`cdev_add` 调用与之前的代码就是所谓的单纯的字符设备驱动（也许以后开源社区会把它称为传统的设备驱动），很明显它已经可以很好地工作了，为了让应用程序可以使用到它提供的服务，只需用 `mknod` 创建一个字符设备文件节点就可以了（这是本书第二章的内容）。后面从 `class_create` 开始，实际上是在原有的设备驱动基础上利用设备模型的概念增加了一些新的特性，它使得我们的设备驱动程序可以向用户空间提供更多的信息，自动生成设备节点，以及更便捷地在用户空间和内核空间传递数据（通过 `sysfs` 文件系统）。Linux 设备模型的内核架构开发者们的工作是辛苦的（你只是看看那些抽象晦涩的 `object`、`kset`、`device` 及 `class` 等等相关代码就会有所体验），所以我们应该理解它，驾驭并善用它，使之为我们服务，才不会辜负开源社区那些辛勤工作的开拓者们，而一个设计低劣的内核模块就可以轻易抹杀掉开拓者们杰出的工作成果。

回到正题，接下来是这个例子使用的应用程序代码：

```

<main.c>
-----
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>

```

```
#define DEVFILE      "/dev/fa_dev"
static unsigned long eflag = 1;

static void sigio_handler(int sigio)
{
    printf("Get the SIGIO signal, we exit the application!\n");
    eflag = 0;
}

static int block_sigio(void)
{
    sigset_t set, old;
    int ret;

    sigemptyset(&set);
    sigaddset(&set, SIGIO);
    sigprocmask(SIG_BLOCK, &set, &old);
    ret = sigismember(&old, SIGIO);
    return ret;
}

static void unblock_sigio(int blocked)
{
    sigset_t set;
    if(!blocked){
        sigemptyset(&set);
        sigaddset(&set, SIGIO);
        sigprocmask(SIG_UNBLOCK, &set, NULL);
    }
}

int main(void)
{
    int fd;
    struct sigaction sigact, oldact;
    int oflag;
    int blocked;

    blocked = block_sigio();

    sigemptyset(&sigact.sa_mask);
    sigaddset(&sigact.sa_mask, SIGIO);
    sigact.sa_flags = 0;
    sigact.sa_handler=sigio_handler;
    if(sigaction(SIGIO, &sigact, &oldact) < 0){
```

```

        printf("sigaction failed!\n");
        unblock_sigio(blocked);
        return -1;
    }

    unblock_sigio(blocked);

    fd = open(DEVFILE, O_RDWR);
    if(fd >= 0){
        fcntl(fd, F_SETOWN, getpid());
        oflag = fcntl(fd, F_GETFL);
        fcntl(fd, F_SETFL, oflag | FASYNC);
        printf("Do everything you want until we get a signal...\n");
        while(eflag);
        close(fd);
    }

    return 0;
}

```

现在用 `insmod` 把 `fasync_demo.ko` 模块加入系统，在模块成功加入后，可以发现在 `/dev` 目录下生成了一个字符设备文件 `fa_dev`：

```

root@AMDLinuxFGL:/home/dennis/Linux/book/chap07/fasync/driver# ls -l /dev/fa_dev
crw-rw---- 1 root root 249, 0 Jun 15 02:04 /dev/fa_dev

```

同时会在 `/sys/devices/virtual/fa_dev/fa_dev` 目录下发现一个名为 `flag` 的文件，它是设备驱动程序在模块初始化函数中通过调用 `device_create_file(fadev, &flag_attr)` 生成的：

```

root@AMDLinuxFGL:/sys/devices/virtual/fa_dev/fa_dev$ ls -l flag
-rw-r--r-- 1 root root 4096 Jun 15 02:13 flag

```

因为在代码中为 `flag` 文件实现了读写操作，所以可以通过该文件来改写设备驱动程序中的 `flag` 变量：

```

root@AMDLinuxFGL:/sys/devices/virtual/fa_dev/fa_dev$ cat flag
0

```

因为驱动程序将 `flag` 变量初始化为 0，所以此处显示 `flag` 的值为 0，现在可以通过下面的命令将 `flag` 的值改为 5：

```

root@AMDLinuxFGL:/sys/devices/virtual/fa_dev/fa_dev# echo '5' > flag
root@AMDLinuxFGL:/sys/devices/virtual/fa_dev/fa_dev$ cat flag
5

```

可以看到 `flag` 的值已经变成了 5。在代码中，除了改 `flag` 的值外，`write_flag` 函数还会调用 `kill_fasync` 函数给应用程序发送信号。

现在可以运行应用程序 `main` 了：

```
root@AMDLinuxFGL:/home/dennis/Linux/book/chap07/fasync/app# ./main
Do everything you want until we get a signal...
```

应用程序将停在此处（代码中的 `while` 循环那里），现在用 `echo '5' > flag` 的方式来模拟设备驱动程序的数据就绪或者其他应用程序感兴趣的事情，以此来通知应用程序：

```
root@AMDLinuxFGL:/sys/devices/virtual/fa_dev/fa_dev# echo '5' > flag
```

此时会发现 `main` 程序将退出：

```
root@AMDLinuxFGL:/home/dennis/Linux/book/chap07/fasync/app# ./main
Do everything you want until we get a signal...
Get the SIGIO signal, we exit the application!
root@AMDLinuxFGL:/home/dennis/Linux/book/chap07/fasync/app#
```

7.9 llseek 例程

驱动程序中的 `llseek` 例程的原型声明为：

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

如果了解从应用层到内核层关于 `llseek` 相关操作的整个流程，在驱动程序中实现 `llseek` 的例程本身并不困难。所以还是从系统调用开始看起，读者不要沮丧，这个过程看起来并不像想象中那么烦琐和无聊。`llseek` 的系统调用为：

<fs/read-write.c>

```
SYSCALL_DEFINE3(llseek, unsigned int, fd, loff_t, offset, unsigned int, origin)
{
    off_t retval;
    struct file * file;
    int fput_needed;

    retval = -EBADF;
    file = fget_light(fd, &fput_needed);
    if (!file)
        goto bad;

    retval = -EINVAL;
```

```

        if (origin <= SEEK_MAX) {
            loff_t res = vfs_llseek(file, offset, origin);
            retval = res;
            if (res != (loff_t)retval)
                retval = -EOVERFLOW; /* LFS: should only happen on 32 bit platforms */
        }
        fput_light(file, fput_needed);
bad:
        return retval;
    }

```

函数中需要关注的地方一个是 if (origin <= SEEK_MAX)处，因为应用程序在调用 lseek 时，对于 origin 参数只有三个选择，如下：

```

<include/linux/fs.h>
-----
#define SEEK_SET    0    /* seek relative to beginning of file */
#define SEEK_CUR    1    /* seek relative to current file position */
#define SEEK_END    2    /* seek relative to end of file */
#define SEEK_MAX    SEEK_END

```

所以这里的检查就是要确保 origin 参数的有效性。函数另外一个地方就是检查完 origin 参数之后开始调用 vfs_llseek：

```

<fs/read-write.c>
-----
loff_t vfs_llseek(struct file *file, loff_t offset, int origin)
{
    loff_t (*fn)(struct file *, loff_t, int);
    fn = no_llseek;
    if (file->f_mode & FMODE_LSEEK) {
        fn = default_llseek;
        if (file->f_op && file->f_op->llseek)
            fn = file->f_op->llseek;
    }
    return fn(file, offset, origin);
}

```

这个函数需要关注的是函数指针 fn 的赋值，fn 在函数中有三个可能的值：no_llseek、default_llseek 和驱动程序提供的 llseek 例程（file->f_op->llseek）。如果 file->f_mode 中的 FMODE_LSEEK 标志没有设置，那么应用程序调用的 lseek 最终调用的是 no_llseek，该函数直接返回一个错误码-ESPIPE。因为设备文件上的 open 操作默认是设置 FMODE_LSEEK 标志的，所以如果驱动程序提供了 llseek 例程，那么将调用它，否则将调用系统默认的 default_llseek 函数，该函数通过修改 filp->f_pos 来达到定位文件的目的。如果调用到设备驱动程序提供的 llseek 例程，那么在该例程中，驱动程序需要根据用户传入的偏移值 off 和调整的起始位置参数来决定如何定位文件，下面是一个实际的设备驱动程序实现的 llseek

文件定位例程：

```
static loff_t vol_cdev_llseek(struct file *file, loff_t offset, int origin)
{
    struct ubi_volume_desc *desc = file->private_data;
    struct ubi_volume *vol = desc->vol;
    loff_t new_offset;

    if (vol->updating) {
        return -EBUSY;
    }

    switch (origin) {
    case 0: /* SEEK_SET */
        new_offset = offset;
        break;
    case 1: /* SEEK_CUR */
        new_offset = file->f_pos + offset;
        break;
    case 2: /* SEEK_END */
        new_offset = vol->used_bytes + offset;
        break;
    default:
        return -EINVAL;
    }

    if (new_offset < 0 || new_offset > vol->used_bytes) {
        dbg_err("bad seek %lld", new_offset);
        return -EINVAL;
    }

    file->f_pos = new_offset;
    return new_offset;
}
```

在实际当中，有些设备的定位是没有意义的，比如所谓的流式设备，其中的数据如同水流一样。这类设备的驱动程序不应该提供 `llseek` 例程，同时也不希望使用内核提供的默认的 `default_llseek`，那么可以在打开这类设备时调用 `nonseekable_open` 来关闭 `FMODE_LSEEK` 标志。`nonseekable_open` 的定义为：

<fs/open.c>

```
int nonseekable_open(struct inode *inode, struct file *filp)
{
    filp->f_mode &= ~(FMODE_LSEEK | FMODE_PREAD | FMODE_PWRITE);
    return 0;
}
```

7.10 访问权能

有时候驱动程序需要检查一个正试图访问它的进程是否有权限做某些事情，这里不妨把进程是否有权限使用驱动程序提供的服务的能力称为权能，此时驱动程序可以用 `capable` 这个函数。这跟文件系统层面的权限检查不同，可以认为驱动程序内部进行的 `capable` 操作是一种粒度更细的权限检查，只有当试图访问它的进程有足够的权限操作设备文件后，驱动程序自身的这种权限检查才会介入，否则进程将直接被挡在文件系统的外围，连设备文件的边都摸不到，更遑论去操作它了。

<kernel/capability.c>

```
int capable(int cap)
{
    if (unlikely(!cap_valid(cap))) {
        printk(KERN_CRIT "capable() called with invalid cap=%u\n", cap);
        BUG();
    }

    if (security_capable(cap) == 0) {
        current->flags |= PF_SUPERPRIV;
        return 1;
    }
    return 0;
}
```

参数 `cap` 用来指定对当前进程进行检查的权能数值，内核在 `include/linux/capability.h` 中总共定义了 33 个权能数值。函数中的 `cap_valid` 用来检查 `cap` 所表示的权能值是否在内核事先定义的权限范围之内。

真正的权能检查发生在 `security_capable` 函数中，其定义为：

<include/linux/security.h>

```
static inline int security_capable(int cap)
{
    return cap_capable(current, current_cred(), cap, SECURITY_CAP_AUDIT);
}
```

其中的宏 `current_cred()` 用来获得当前进程的一个权能证书，展开就是 `current->cred`。如果将整个 `cap_capable` 函数的调用链全部展开，那么它看起来如下：

<security/commoncap.c>

```
int cap_capable(struct task_struct *tsk, const struct cred *cred, int cap,
                int audit)
```

```

{
    int tmp;
    tmp = ((cred->cap_effective).cap[CAP_TO_INDEX(cap)] & CAP_TO_MASK(cap));
    return tmp? 0 : -EPERM;
}

```

函数检查当前进程的权能证书中的 `cap_effective` 成员，以确定进程是否具有参数 `cap` 中指定的权能，如果进程具有指定的权能，它将返回 0，返回一直错误码-EPERM。所以当函数返回到 `capable` 那里，我们知道如果进程具有指定的权能，那么函数将在当前进程的 `flags` 上设置 `PF_SUPERPRIV` 标志着一个超级用户的身份：`current->flags |= PF_SUPERPRIV`，同时返回 1；如果进程不具有指定的权能，那么函数返回 0。当驱动程序发现当前进程不具有进一步操作的权限时，常常返回一个错误码-EPERM，如下所示：

```

if (!capable(CAP_NET_ADMIN))
    return -EPERM;

```

7.11 本章小结

本章讨论了 `struct file_operations` 操作中的大部分函数的实现，其中字符型设备驱动程序最常用的是 `ioctl` 例程。通过设备驱动程序的 `ioctl` 例程的实现，应用程序可以与驱动程序实现数据的交互，实际当中这些数据量多半比较少，常常用来作为控制设备操作模式的一种方法。

除此之外还重点讨论了字符设备的 `poll` 和 `fasync` 操作。前者可以让应用程序通过 `select` 等 API 睡眠等待在一组 `fd` 上，当前的字符设备驱动程序所在的设备节点就对应其中的一个 `fd`。进程醒来的条件是，或者指定时间到期，或者 `fd_set` 中的某一 `fd` 就绪。字符设备在实现自己的 `poll` 例程时，需要维护一个自己的等待队列，将来自内核的等待节点通过 `poll_wait` 加入到自己的等待队列上，当数据就绪时唤醒等待队列上的进程，这使得应用程序的 `select` 函数返回。

`fasync` 用来实现一个异步通知机制，用户程序通过 `fcntl` 函数来向设备驱动程序表明是否希望在某一事件出现时得到通知。设备驱动程序在实现 `fasync` 例程时主要依赖两个内核提供的函数：`fasync_helper` 和 `kill_fasync`，前者将需要通知的进程加入一个链表，后者在应用程序关注的事件发生时通过信号发送的方式来通知应用程序。

第 8 章

时间管理

设备驱动程序需要对时间进行操作，典型的可以分为两大类：延时与定时。前者是在两个连续的动作 A 与 B 之间插入一段时间空白，也即在动作 A 执行后需要等待若干时间才能执行动作 B，至于在这段时间空白内，当前处理器也许是进入忙等待状态，也许是切换到一个新进程上。后者是在一个指定的时间点到达后执行某些动作，轮询是其最典型的应用。

本章将讨论这两类时间上的操作的技术细节，设备驱动程序员在掌握了这些幕后的技术之后可以更好地理解设备驱动是如何对时间进行掌控的，当程序中需要对时间进行管理时选择最合适的解决方案。

8.1 jiffies

内核源代码中几乎到处充斥着 jiffies 这样的变量，作为设备驱动程序员对此想必也一定不会陌生，在某些书中它被形象地称为“时钟滴答”。内核源码中针对 32 位和 64 位系统分别定义了 jiffies 和 jiffies_64：

```
<include/linux/jiffies.h>
-----
#define __jiffy_data __attribute__((section(".data")))
extern u64 __jiffy_data jiffies_64;
extern unsigned long volatile __jiffy_data jiffies;
```

其中的__jiffy_data 表明这两个变量将出现在内核最终映像的".data"区中，另外在头文件中在一个变量的声明前使用了“extern”关键字，提示了这个变量可能定义在某个别的文件中，事实上它们出现在内核的链接脚本文件 vmlinux.lds 中。除了数据位宽不一样外，上述两个变量在原理上是一样的。为了叙述的方便，下面只提 jiffies，本节稍后会给出两者在操作上的一些细微的区别。

通常 jiffies 在 Linux 系统启动引导阶段被初始化为 0，当系统完成了对时钟中断¹的初始化

¹ 基于 x86 体系架构的 Linux 系统中产生时钟中断的硬件典型的有可编程中断计数器 PIT (Programmable Interrupt Timer) 8253 和高级可编程中断控制器 APIC (Advanced Programmable Interrupt Controller)，后者的分辨率及稳定性都要比前者好得多，用来实现高分辨率的时间源。

之后，在每个时钟中断（“时钟滴答”）处理例程中该值都会被加 1，如图 8-1 所示：

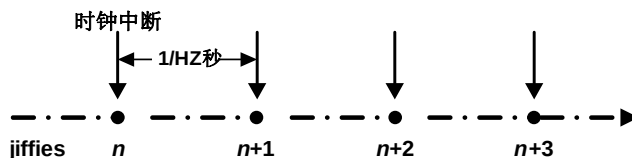


图 8-1 每隔 1/HZ 秒 jiffies 的值增 1

因此该值储存了系统自最近一次启动以来的时钟滴答数。在形式上，它跟我们日常所熟悉的时分秒这样的时间概念有很大的不同，不过对于设备驱动程序而言，出于时间管理的需要，使用 jiffies 就已经足够，因为它甚少用这种时间形式与应用程序进行沟通。除了时钟中断处理例程中对 jiffies 进行更新外，其他任何模块（驱动程序当然也不例外）都只是读取该值以获得当前时钟计数。

在实际使用 jiffies 时，还需要了解 Linux 内核中另一个与时钟中断息息相关的宏 HZ，它用来表示系统中时钟中断发生的频率：

```
<include/asm-generic/param.h>
-----
#ifdef __KERNEL__
# define HZ      CONFIG_HZ      /* Internal kernel timer frequency */
# define USER_HZ 100           /* some user interfaces are */
# define CLOCKS_PER_SEC (USER_HZ) /* in "ticks" like times() */
#endif
```

从上述的定义可以看出，内核提供了在配置阶段通过 CONFIG_HZ 修改 HZ 数值的可能性，但绝大多数情况下都没有必要修改它，使用内核默认的值 1000 就足够了。事实上 CONFIG_HZ 并未出现在内核的配置菜单选项中，而是就在内核源码根目录下的 config 文件中。HZ 值为 1000 意味着系统 1 秒内要发生 1000 次时钟中断，也就是说每隔 1 毫秒，jiffies 的值就会增加 1。所以，如果驱动程序使用 jiffies 来对时间进行度量的话，其精度只能局限在毫秒级别上，更高精度的时间管理单纯使用 jiffies 无法满足要求。

相对于 jiffies 而言，jiffies_64 是个 64 位的变量（即便是在 32 位的体系架构上也是一样，此时它是一个 unsigned long long 型的变量，通过组合两个 unsigned long 型变量得到），在 64 位平台上，它们其实是同一个变量，而在 32 位平台上，jiffies 和 jiffies_64 的低 32 位是重合的。之所以引入 jiffies_64，是考虑到了 32 位变量 jiffies 的溢出问题，在 HZ=1000 的情况下，大约 50 天就会导致 jiffies 溢出。对于驱动程序中的时间度量而言，这并不是个大问题（但是在作时间比较的时候仍然需要小心处理），不过现实中显然要考虑某些系统²有需要知道自系统最近一次运行以来真正的时钟滴答数的需求，因此 Linux 内核中同时引入

² 比如对某些特殊的服务器而言，它们启动一次运行的时间也许足够久，以至于久到发生了 jiffies 的溢出。

了 `jiffies_64` 来记录系统的时钟计数。为了保证 `jiffies` 和 `jiffies_64` 两个变量无论在 32 位还是 64 位平台上在记录时钟滴答数时的一致性，内核通过链接脚本做了些手脚，有兴趣的读者可以阅读一下链接内核时所使用的链接脚本 `vmlinux.lds`，可发现类似下面的内容：

```
#ifdef CONFIG_X86_32
OUTPUT_ARCH(i386)
jiffies = jiffies_64;
#else
OUTPUT_ARCH(i386:x86-64)
jiffies_64 = jiffies;
#endif
```

如果要在 32 位系统上读取 `jiffies_64` 的值，必须使用 `get_jiffies_64` 函数，因为在直接读取 `jiffies_64` 的高 32 位或者低 32 位时，对应的低 32 位或者高 32 位可能已经发生更新。`get_jiffies_64` 函数使用顺序锁的方式来保证对 `jiffies_64` 变量读取操作的原子性：

```
<kernel/time.c>
-----
#if (BITS_PER_LONG < 64)
u64 get_jiffies_64(void)
{
    unsigned long seq;
    u64 ret;

    do {
        seq = read_seqbegin(&xtime_lock);
        ret = jiffies_64;
    } while (read_seqretry(&xtime_lock, seq));
    return ret;
}
#endif
```

从设备驱动程序的角度出发，仅仅使用 `jiffies` 变量就已足够满足所有基于 `jiffies` 的时间度量任务，所以本章接下来的部分将主要以 `jiffies` 为讨论对象。

由于 `jiffies` 在内核源码中作为一个全局性变量被导出，所以如果驱动程序中需要读取当前的 `jiffies` 值，只需在源码中包含头文件 `linux/jiffies.h` 即可，比如下面的代码片段：

```
#include <linux/jiffies.h>
unsigned long j, timestamp_1, timestamp_2;

j = jiffies; //读取当前的时钟计数值
timestamp_1 = jiffies + 2 * HZ; //timestamp_1 为未来的 2 秒
timestamp_2 = jiffies + 3 * HZ / 1000; //timestamp_2 为未来的 3 毫秒
```

Linux 设备驱动程序中使用 `jiffies` 的几个常用的场景分别有时间比较、时间转换以及设置定时器（timer）时对未来时间的设定，本节先讨论前两个话题，后一个话题将在稍后的系统

定时器一节中予以讨论。

8.1.1 时间比较

设备驱动程序有时候需要对程序执行过程中的两个时间点进行比较，以确定时间点之间的先后次序。因为 `jiffies` 在每次的时钟中断处理例程中都会被更新，因此可以通过两次时间点所对应的 `jiffies` 值来进行判断。如果没有前面提到的 `jiffies` 值溢出的问题，那么这种判断的逻辑非常简单，但是因为溢出的可能性是存在的，所以程序应该谨慎处理。好在 Linux 内核为此提供了一组用以判断时间点先后顺序的宏，通过特定的技巧非常安全地处理了 `jiffies` 值溢出的情况，程序中可以放心使用。这组宏为：

`time_after(a, b)`

如果时间点 `a` 在时间点 `b` 之后，该宏返回 `true`。

`time_before(a, b)`

如果时间点 `a` 在时间点 `b` 之前，该宏返回 `true`。

`time_after_eq(a, b)`

该宏类似于 `time_after`，但是在 `a` 和 `b` 两个时间点相等时，该宏也返回 `true`。

`time_before_eq(a, b)`

该宏类似于 `time_before`，但是在 `a` 和 `b` 两个时间点相等时，该宏也返回 `true`。

`time_in_range(a, b, c)`

该宏用来检查时间点 `a` 是否包含在时间间隔 `[b, c]` 内，因为检查包含边界，所以当 `a` 等于 `b` 或者 `c` 时，该宏也会返回 `true`。

在使用以上宏时，参数 `a` 和 `b` 都应该是 `unsigned long` 型变量。如果是针对 `jiffies_64` 类型来作这种时间顺序的判断，那么除了 `time_in_range` 宏之外，只需在宏的后面加上后缀 `64` 即可，例如 `time_after64`，不过设备驱动程序中使用 `64` 位的情形极其罕见。

因此，设备驱动程序中不应该直接使用 `jiffies` 的值来作时间点先后顺序的比较，而应该使用内核提供的上述宏来完成。下面给出一个具体的例子，假设驱动程序的某个函数 `demo_function` 需要调用比如 `do_time_task` 函数来完成一个任务，但是对任务完成的时间有特定的要求（比如要在 2 毫秒之内完成），如果在规定的时间内没有完成，就需要调用 `task_timeout` 函数来处理，否则 `demo_function` 函数就算顺利完成。如果使用下面的代码将是不安全的：

```
int demo_function()
```

```

{
    unsigned long timeout = jiffies + 2 * HZ / 1000; //设定超时的时间为 2 毫秒
    do_time_task(); //调用 do_time_task 来完成某一任务
    if (timeout < jiffies) //根据当前最新的 jiffies 值来判断是否超时
        return task_timeout(); //do_time_task()完成时间超过了 2 毫秒，调用超时处理函数
    return 0;
}

```

因为存在 jiffies 溢出环绕的可能性，所以上述的 if 语句中 `timeout < jiffies` 条件在超时的情况下也可能返回 false (比如计算出的 timeout 值本身已经接近 jiffies 溢出环绕的临界点时)。正确的代码应该是：

```

int demo_function()
{
    unsigned long timeout = jiffies + 2 * HZ / 1000; //设定超时的时间为 2 毫秒
    do_time_task(); //调用 do_time_task 来完成某一任务
    if (time_after(jiffies, timeout)) //根据当前最新的 jiffies 值来判断是否超时
        return task_timeout(); //do_time_task()完成时间超过了 2 毫秒，调用超时处理函数
    return 0;
}

```

8.1.2 时间转换

有时候，设备驱动程序可能需要将用 jiffies 表达的时间间隔转化成毫秒 ms 或者是微秒 us 的形式，这种情况大多出现在需要将时钟滴答这种形式转化成人易于理解的 ms 或者是 us 这样的时间形式下，比如为了在驱动程序中打印出一次 DMA 传输所花费的时间，在 DMA 传输开始前记录 `start_jiffies = jiffies`，然后进行 DMA 传输，在 DMA 传输结束后记录下 `end_jiffies = jiffies`，这样本次的 DMA 传输所花费的时间将为 `time = jiffies_to_msecs(end_jiffies - start_jiffies)`，这里 time 的单位将会是 ms。

Linux 内核源码为此提供了一组相关的转换函数：

```

<include/linux/jiffies.h>
-----
unsigned int jiffies_to_msecs(const unsigned long j);
unsigned int jiffies_to_usecs(const unsigned long j);
unsigned long msecs_to_jiffies(const unsigned int m);
unsigned long usecs_to_jiffies(const unsigned int u);

```

从这些函数的命名上已经可以很清楚地知道其各自的功能，此处不再赘述。时间转换的另一种情形发生在用户态程序和设备驱动程序的交互上，应用程序员更多地使用秒以及毫秒等时间形式。此种情形下，内核定义了 `struct timeval` 和 `struct timespec` 两种数据结构：

```

<include/linux/time.h>
-----
struct timespec {

```

```

    __kernel_time_t tv_sec;           /* seconds */
    long tv_nsec;                     /* nanoseconds */
};

struct timeval {
    __kernel_time_t tv_sec;           /* seconds */
    __kernel_suseconds_t tv_usec;     /* microseconds */
};

```

可见 `timespec` 用秒和纳秒来描述时间，而 `timeval` 则采用秒和毫秒的形式。内核同样提供了 `jiffies` 变量和这两个数据结构的实例间相互转换的函数：

```

<include/linux/jiffies.h>
-----
unsigned long timespec_to_jiffies(const struct timespec *value);
void jiffies_to_timespec(const unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(const struct timeval *value);
void jiffies_to_timeval(const unsigned long jiffies, struct timeval *value);

```

这些函数的用法也是很明显的。

到目前为止，已经讨论了基于 `jiffies` 的时间度量的方法，鉴于 `jiffies` 自身精度的局限性，如果需要使用更高精度的时间度量方法，也许要借助于某些体系架构特定的计时寄存器，例如很有名的时间戳计数器 TSC（Time Stamp Counter），但是使用这些寄存器的程序代码将失去在不同平台之间的可移植性。因为用 `jiffies` 来衡量一个时间间隔在绝大多数的情况下已经足以满足需要，所以本书将不再讨论其他的时间度量方法。

8.2 延时操作

设备驱动程序中延时操作的常见应用场景是，当 CPU 通过外部设备的寄存器对设备发出指令时，外设执行相应的动作，在该动作完成之后通过更新比如状态寄存器来告之本次操作的执行结果，CPU 需要读取该状态寄存器的值来获得设备的执行结果。因为 CPU 的速度很快，而外部设备可能需要一定的时间才能完成本次操作，如果 CPU 在写完寄存器后直接读取设备的状态寄存器，那么很有可能得到错误的结果（设备尚未完成指定的操作，因而还没有更新状态寄存器），所以 CPU 在对外设发出操作指令后，需要延时一段时间以等待设备操作的完成。这种情形在设备驱动程序中一个简单而典型的代码执行序列应该是：

```

001    write_command_reg(...);    //CPU 写外设的寄存器以发起一个操作指令
002    delay(...); //延时操作，等待设备操作完成。它的实现机制是本节要讨论的主题
003    read_status_reg(...);    //CPU 读外设的寄存器以获得设备执行结果

```

上述序列的第 2 行执行的就是一个延时操作，因为它的存在使得第 3 行的指令向后推迟了

一段时间才被执行到，这使得外部设备有足够的时间³来完成当前的操作并将执行结果更新到状态寄存器中。下面讨论设备驱动程序如何实现 `delay` 函数以实现对应的时间延迟。

从实现延时精度的角度出发，可以将延迟函数分成两大类：一类是基于时钟滴答 `jiffies` 实现的延迟，因为这类延迟的时间粒度一般在毫秒 `ms` 级别，所以被称为“长延时”；另一类的延时精度已经超越了时钟滴答的边界，比如微秒 `us` 和纳秒 `ns` 级的延迟，显然单纯依靠 `jiffies` 已经无法满足要求，此时需要有另外的实现机制，也就是所谓的“短延时”函数。下面先从长延时开始讨论。

8.2.1 长延时

基于时钟滴答 `jiffies` 的长延时函数有几种实现方法，主要围绕在延迟实现过程中是否让出处理器来展开，在具体的实现上分为“忙等待”和“让出处理器”两大类。

○ 忙等待

用忙等待来实现延时是最简单的。虽然因为忙等待的关系其执行效率饱受诟病，但是不得不承认在设备驱动程序早期的开发调试阶段这是一种很简便的手法来判断设备是否能像预期的那样工作。

最简单的忙等待实现是用 `while` 或者其他的什么循环，比如：

```
unsigned long t = 0xFFFFF;  
while(t--);
```

这是种相当粗糙的实现延时操作的原始方法，现实当中的程序员也许迫切想获得某种延迟效果而又不想稍微用点脑力去寻求更好的解决方案时，上面的代码就会出现。因为它的缺点是如此明显，所以它显然不应该出现在最终发布出去的软件版本中。

其实完全可以利用 `jiffies` 来实现一种比较理想的忙等待延时策略，而且相对前面的那种忙等待，这种方法对延时的长短也有很好的控制，比如为了实现 1 s 的等待，可以使用下面的方法：

```
unsigned long j = jiffies + HZ;  
while(time_before(jiffies, j))  
    cpu_relax();
```

上面的这段延时代码看起来已经很像那么回事了，除了利用 `time_before` 和 `jiffies` 来实现 1 s 延时的控制，还在 `while` 循环体中加入了对 `cpu_relax` 函数的调用。显然 `cpu_relax` 的实现不

³ 设备驱动程序员也许要根据外部设备的 `data sheet` 等操作规范文档来评估一个操作需要花费多少时间，在获得这个数据后才可能精确地设定后续 `delay` 操作需要延迟的时间宽度。

会导致当前代码让出处理器，否则就不能称为忙等待了⁴。cpu_relax 是个平台相关的函数，在 x86 架构上，其核心指令是 NOP，也就是空指令：

```
<arch/x86/include/asm/processor.h>
-----
/* REP NOP (PAUSE) is a good thing to insert into busy-wait loops. */
static inline void rep_nop(void)
{
    asm volatile("rep; nop" ::: "memory");
}

static inline void cpu_relax(void)
{
    rep_nop();
}
```

其他平台上，比如 ARM，cpu_relax 的实现可能是一个内存屏障类的函数调用：

```
<arch/arm/include/asm/processor.h>
-----
#if __LINUX_ARM_ARCH__ == 6
#define cpu_relax()          smp_mb()
#else
#define cpu_relax()          barrier()
#endif
```

无论如何，这段代码实现背后的原理还是非常直白的，其缺陷也同样很直白：

(1) 基本上在这 1 s 的延迟时间段内进入忙等待的 CPU 做不了任何事情，对于当前的高速 CPU 而言，这种资源的浪费是很可观的。而且对于单 CPU 系统来说，如果不幸在进入这段忙等待的代码前关闭了 CPU 的中断，那么 jiffies 的值将不会被更新，while 循环的条件将一直满足，这种情况下除了重启系统似乎没有更好的解决方法。

(2) 对于一个可抢占式的系统而言，上面的忙等待代码有可能在某次中断的过程中被抢占，比如，进程 A 在等待外设的数据时进入了睡眠状态，此时调度器调度进程 B 到当前的 CPU 上运行，假设进程 B 恰好执行的就是上述的忙等待代码，如果在延迟时间段尚未结束时，进程 A 因外设数据的就绪被唤醒并且调度优先级高于进程 B，那么 A 将被重新调度到当前处理器上运行，如果 A 再次被调度后连续运行的时间超过了 1 s，那么当 B 被再次调度运行时，while 中的条件显然已经不再满足，此时延时的目的虽然是达到了，但是延迟的时间并不是当初设定的 1 s，而可能是比如 1.5 s 等。

⁴ 至于这段忙等待代码在执行过程中是否会让出当前处理器，要看内核是否配置启用了可抢占性。在 1 s 的时间里，这个 while 循环体在执行过程中会出现大量的时钟中断，如果内核不可抢占，那么运行在内核态的这个忙等待代码不会产生新的调度点，因此所占有的处理器不会被强制剥夺。但对于可抢占式内核而言，如果有更高优先级的任务就绪，则它可能会被调度出处理器。

上述的问题 2 对于驱动程序来说并不是什么大的问题，即便是在后面讨论的一些改进型的延迟实现中也同样存在。如果没有 CPU 资源的浪费，那么即便延迟函数造成了预设延迟时间段的延伸，对设备驱动的性能而言也不会有实质性的影响，而对这个问题的根本性解决也许要涉及对调度器的改进，这对于延时精度本来就要求不高的长延迟函数而言，没有充足的理由。而对问题 1 的改进则导致了“让出处理器”解决方案的出现。

○ 让出处理器

在忙等待的实现中，处于忙等待中的代码一直占用处理器会导致系统性能降低，于是一种改进的方案是：当代码进入到 while 循环时，不再调用 `cpu_relax()` 函数而是调用 `schedule()` 调度函数以让出处理器，这样就解决了忙等待一直浪费处理器的弊端。比如下面的代码：

```
unsigned long j = jiffies + HZ;
while(time_before(jiffies, j))
    schedule();
```

这种方法相对于前面的忙等待来说，解决了无端占用 CPU 的问题，但还不是最佳的解决方案，因为主动调用 `schedule()` 函数的进程虽然可以让出处理器，但依然在当前 CPU 的运行队列中。这使得在空闲的系统中无法进入 idle 状态，因为即便 CPU 的运行队列中只有当前一个进程，它也会陷入让出处理器之后马上又被调度运行，然后再让出处理器这样的怪圈中。无法进入 idle 状态对系统的电源管理模块来说不是件好事情，因为有些智能化的电源管理模块可以根据当前 CPU 的负载情况来决定是否改变 CPU 的运行频率，频率的改变与 CPU 供电电压的改变是息息相关的。CPU 的 idle 状态可以被电源管理模块所利用，如果后者发现 CPU 进入了 idle 状态，可以降低 CPU 的核心频率从而降低整机的能耗，这在以 ARM 为主的嵌入式平台上尤为常见。

当然，相对于一直占用 CPU 资源的忙等待，这种方法提升了 CPU 资源的利用率，使得当前进程在延迟等待期间 CPU 可以运行其他的进程。另外要提的一点是，进程在调用 `schedule()` 函数让出处理器后，并不能保证可以很快再次获得处理器，其再次获得处理器的时间间隔取决于当前处理器运行队列中进程的数量，以及这些进程与延迟等待进程的调度优先级。换言之，问题 2 中阐述的预设延迟时间段延伸的问题依然存在。

上面提到的采用 `schedule()` 函数的解决方法之所以还不是最佳的，其根本原因在于调用 `schedule()` 函数的进程依然处于 CPU 的运行队列中。为了解决这个问题，此时应该能想到内核提供的另外一种可供设备驱动程序使用的调度类的基础设施：`schedule_timeout`。所以，如果一个延迟 1 s 的函数可以用下面的这样一个简单的代码段来实现：

```
delay_1s()
{
    set_current_state(TASK_UNINTERRUPTIBLE)5;
```

⁵ 此处当然可以使用 `TASK_INTERRUPTIBLE` 等进程状态标志，不过要小心处理在延迟时间到达之前进程被信号等中断的可能。为了便于叙述，接下来的文字中统一使用 `TASK_UNINTERRUPTIBLE` 标志。

```

    schedule_timeout(jiffies + HZ);
}

```

上面这段代码之所以可以解决直接采用 `schedule()` 函数所带来的负面问题，主要在于在调用 `schedule_timeout` 之前先调用了 `set_current_state` 宏将当前进程的状态设置为 `TASK_UNINTERRUPTIBLE`，这样当随后的 `schedule_timeout` 函数被调用时，后者的内部实现中调用了 `schedule()` 函数，因为当前进程之前的状态已经被设置为 `TASK_UNINTERRUPTIBLE`，所以在 `schedule` 函数中当前进程将会被移出处理器的运行队列，因此也就解决了采用直接调用 `schedule` 函数那种方案所带来的不利影响。也许有读者会问，在早先的那个直接调用 `schedule` 函数的方案前调用一下 `set_current_state(TASK_UNINTERRUPTIBLE)` 不也可以把当前进程移出运行队列吗？比如：

```

unsigned long j = jiffies + HZ;
while(time_before(jiffies, j)){
    set_current_state(TASK_UNINTERRUPTIBLE);
    schedule();
}

```

上面的代码实现非常糟糕，已经不只是仅对性能有影响那么简单了，通过在 `schedule()` 函数前使用 `set_current_state(TASK_UNINTERRUPTIBLE)`，固然可以使当前进程从处理器的运行队列中移开，但它此后将再也没有机会被调度，因为不会再有别的代码去更改其状态使其可以再次进入运行队列，这与使用等待队列和调用 `schedule_timeout` 完全不同。如果使用了等待队列，那么我们知道等待队列中的睡眠进程有被唤醒一说，关于这点，本书前面的章节已经讨论过。如果使用 `schedule_timeout`，当前进程从运行队列移走之后将被记录到定时器的数据节点中，当定时器的时间到期后，将会唤醒该进程使其重新进入运行队列。这里不妨通过内核中 `schedule_timeout` 实现的一些关键代码，来看一看此处蕴藏的秘密：

<kernel/timer.c>

```

-----
signed long __sched schedule_timeout(signed long timeout)
{
    struct timer_list timer;
    unsigned long expire;
    ...
    expire = timeout + jiffies;

    setup_timer_on_stack(&timer, process_timeout, (unsigned long)current);
    __mod_timer(&timer, expire, false, TIMER_NOT_PINNED);
    schedule();
    del_singleshot_timer_sync(&timer);

    /* Remove the timer from the object tracker */
    destroy_timer_on_stack(&timer);
    ...
}

```

```
}
```

函数的主要职能是在调用 `schedule()` 函数前实现了一个定时器，关于定时器，本章后续的内容将很快讨论到它，此处只要记住当 `expire` 指定的时钟滴答到期时，`setup_timer_on_stack` 函数调用中的第二个参数 `process_timeout` 会被调用到，同时注意到指向当前进程的 `current` 指针作为第三个实参传给了 `setup_timer_on_stack` 函数，这样在 `process_timeout` 函数被调用时将会获得当前进程的指针，我们很容易猜出 `process_timeout` 函数要做的事情，当定时器到期时，它被调用以唤醒 `current` 进程：

```
<kernel/timer.c>
-----
static void process_timeout(unsigned long __data)
{
    wake_up_process((struct task_struct *)__data);
}
```

它是如此简单明了，所以不需要在这上面浪费过多的文字。

现在我们已经看到，使用 `schedule_timeout` 可以确保在指定的延迟时间到期时进程可以重新获得调度的机会，因为结合了对 `set_current_state(TASK_UNINTERRUPTIBLE)` 的一起使用，使得进程在指定的延时时间段内不会出现在运行队列中，这就很好地解决了单纯调用 `schedule` 函数所带来的问题。需要提醒一下的是，当定时器到期时，虽然 `process_timeout` 将进程重新放入了处理器的运行队列，但是它何时被调度依然无法给出精确的时间点（这取决于调度器）。换言之，预定的延迟时间段的延伸在这里同样是个无法回避的问题。如果在调用 `schedule_timeout` 前没有使用 `set_current_state` 来将当前进程的状态改为 `TASK_UNINTERRUPTIBLE`，那么效果等同于直接调用 `schedule` 函数，除了已经讨论过的不利因素外，使用 `schedule_timeout` 并不会带来额外的麻烦。从内核的角度，试图唤醒一个已经处于运行队列中的进程并不会造成多少困惑和工作量，好奇的读者可以看看 `wake_up_process` 的代码实现在这种情况下是如何处理的。

事实上，Linux 内核还提供了一个基于上述 `schedule_timeout` 版本的实现毫秒级睡眠的函数 `msleep`：

```
<kernel/timer.c>
-----
void msleep(unsigned int msecs)
{
    unsigned long timeout = msecs_to_jiffies(msecs) + 1;

    while (timeout)
        timeout = schedule_timeout_uninterruptible(timeout);
}
```

`schedule_timeout_uninterruptible` 的内部实现其实就使用了 `__set_current_state` 和 `schedule_timeout`：

```
signed long __sched schedule_timeout_uninterruptible(signed long timeout)
{
    __set_current_state(TASK_UNINTERRUPTIBLE);
    return schedule_timeout(timeout);
}
```

注意到__set_current_state(TASK_UNINTERRUPTIBLE)将当前进程的状态设置为不可中断的 TASK_UNINTERRUPTIBLE, 这样将可以确保进程将至少休眠用参数 msec 指定的时间。内核中还提供了 msleep 的一个变体 msleep_interruptible:

```
<kernel/timer.c>
-----
unsigned long msleep_interruptible(unsigned int msec)
{
    unsigned long timeout = msec_to_jiffies(msec) + 1;

    while (timeout && !signal_pending(current))
        timeout = schedule_timeout_interruptible(timeout);
    return jiffies_to_msec(timeout);
}
```

相对于 msleep 函数, msleep_interruptible 函数内部通过 schedule_timeout_interruptible 在当前进程睡眠之前设置其状态为 TASK_INTERRUPTIBLE, 这样睡眠的进程将处于“可中断的睡眠”这样的状态, 如果在 msec 指定的延迟时间到期之前, 进程因为接收到了信号而被唤醒, while 循环中的 signal_pending(current)将返回 true, 那么 schedule_timeout_interruptible 将返回原先指定的休眠时间 msec 的剩余时间值, 这意味着 msleep_interruptible 将无法保证进程一定在指定的延迟时间过后醒来。通常情况下 msleep_interruptible 都会返回 0, 意味着进程完整地休眠了 msec 指定的时间值。

所以在“让出处理器”实现长延时的方案中, 直接调用内核提供的 msleep 类的函数是最简单最方便的一种方式了。

现在可以简单总结一下“让出处理器”这种延时方案的实现了, 通过对直接单纯调用 schedule 方案的改进, 我们获得了一个相对比较理想的解决方案: 在等待延迟的时间段内, 进程并不会占用处理器, 因而也就不会影响处理器进入 idle 状态, 相对于忙等待而言, 这无疑是个很大的优势。说它相对比较理想, 是因为它依然存在着延时精度的问题, 不过这毕竟是系统内核的限制, 不是我们的设备驱动程序做不到。另外要强调的是, 到目前为止所有延迟的实现都是基于时钟滴答, 因为它的实现机制导致了它在度量粒度上的固有限制(所以被称为“长延时”), 所以如果实现粒度更细的延时, 比如微秒甚至纳秒级, 就需要采用其他的方法了, 这正是下一节“短延时”所要讨论的问题。

8.2.2 短延时

有时候设备驱动程序需要更短的延时, 比如微秒甚至纳秒级的延迟, 这种量级的时延有时

候被通俗地称为“短延时”，形象地说明了在延迟粒度上与长延时的区别。基于下面的两个事实，这种量级延迟的实现已经不可能也不必要像前面讨论“长延时”那样直接用时钟滴答 jiffies 来实现：

(1) 假设在通常的 HZ=1000 的系统上，1 秒钟内 jiffies 增加的数量为 HZ，也就是说 jiffies 的值每隔 1 毫秒才会增加 1，这意味着根据前后时间点的 jiffies 值来度量时间宽度的话，其分辨率最多也只能达到毫秒级。所以在这种情况下，要实现微秒级的延迟，单纯通过 jiffies 的差值已经不可能完成。

(2) 在微秒级的水平上，必须要考虑到进程切换所带来的时间开销，因为进程切换所耗费的时间大约就在几个微秒到上百个微秒之间⁶。这种情况下，如果像“长延时”中“让出处理器”那样实现延迟，很有可能在进程切换的时候延迟的时间就到了。所以“短延时”一般都是基于忙等待来实现。

Linux 内核提供了毫秒、微秒和纳秒级的延迟实现：

```
<include/linux/delay.h>
-----
void mdelay(unsigned long msecs);
void udelay(unsigned long usecs);
void ndelay(unsigned long nsecs);
```

这些延迟的实现都是基于忙等待，其中最基础的一个宏是 `udelay`，另外两个宏都是通过宏 `udelay` 来实现自身功能。`udelay` 的实现方法与体系架构相关。在讨论上述函数的实现时，一个很重要的变量是 `loops_per_jiffy`，用来表示在一个完整的 jiffies 时间段内运行一个内部循环的次数，只要知道了该值就能计算出比如 1 us 循环的次数，这样通过在一个循环中对该循环次数递减就可以在 1 us 的时间到期时退出循环，从而实现 1 us 的延时，这就是短延时函数实现的基本原理。当然如果特定的体系架构上有更精确的硬件计数器，比如 TSC，则利用它们来实现短延时要更精确、更容易。当然这样的实现方式是以牺牲跨平台的可移植性为代价的。Linux 内核有几种方法可以获得该 `loops_per_jiffy` 值，因为和驱动程序关系不大，所以此处不再详细讨论。

8.3 内核定时器

内核定时器是设备驱动程序中经常要用到的另一个重要的内核设施。如果驱动程序希望在将来某个可度量的时间点到期后，由内核安排执行某项任务（此处的任务通常是驱动程序自身定义的某个函数，接下来的叙述中称之为定时器函数），便可以使用定时器来完成。

⁶ 这里只是给出了大约的量级，精确测量 Linux 内核中发生一次进程切换的开销不是一件简单的事情。

设备驱动程序中对内核定时器的一个典型使用场景是用它来实现轮询机制，因为定时器函数自身可以重新启用它所在的定时器，所以在一段时间到期后，定时器函数被调用，在函数内部因为又重新启用了该定时器，这样便形成了一个不断循环的定时器函数被系统调用的模式。此种情形下，如果设备驱动程序需要周期性地检查设备的某种操作状态，便可以在定时器函数中来完成。

驱动程序等内核模块如果要使用定时器，首先应该定义一个定时器类型的变量。`struct timer_list` 是内核提供的一个用来表示定时器的数据结构，其定义如下（删去了一些用于调试及统计信息的成员）：

```
<include/linux/timer.h>
-----
struct timer_list {
    /*
     * All fields that change during normal runtime grouped to the
     * same cacheline
     */
    struct list_head entry;
    unsigned long expires;
    struct tvec_base *base;

    void (*function)(unsigned long);
    unsigned long data;
    int slack;
};
```

其中在驱动程序中常用的是以下三个成员：

`unsigned long expires`

指定定时器的到期时间。

`void (*function)(unsigned long)`

定时器函数。当 `expires` 中指定的时间到期时，该函数将被触发。

`unsigned long data`

定时器对象中携带的数据。通常的用途是，当定时器函数被调用时，内核将把该成员作为实际参数传递给定时器函数。之所以要这样做，是因为定时器函数将在中断上下文中执行，而非当前进程的地址空间中。

其他的一些成员将主要由内核使用，用以实现定时器的内核机制，在后面会看到这些成员的用法。

为了让读者对驱动程序使用内核定时器有个直观的印象，接下来将先给出一段示例代码，

然后再对其中一些关键函数的使用及其内核实现机制进行分节讨论。下面的代码展示了一个设备驱动程序通过使用内核定时器来轮询设备状态。

```

struct device_regs *devreg = NULL; //定义一个用于表示设备寄存器的结构体指针
struct timer_list demo_timer; //定义一个内核定时器对象

//
//定义定时器函数，当定时器对象 demo_timer 中 expires 成员指定的时间到期后，该函数将
//被调用
//
static void demo_timer_func (unsigned long data)
{
    //在定时器函数中重新启动定时器以实现轮询的目的
    demo_timer.expires = jiffies + HZ;
    add_timer(&demo_timer);

    //定时器函数将 data 参数通过类型转换获得设备寄存器的结构体指针
    struct device_regs *preg = (struct device_regs *) data;
    //定时器函数此后将会读取设备状态
    ...
}

//
//用于打开设备的函数实现
//
static int demo_dev_open(...)
{
    ...
    //分配设备寄存器结构体的指针变量，最好放在模块初始化函数中...
    devreg = kmalloc(sizeof(struct device_regs), GFP_KERNEL);
    ...
    init_timer(&demo_timer); //调用内核函数 init_timer 来初始化定时器对象 demo_timer
    demo_timer.expires = jiffies + HZ; //设定定时器到期时间点，从现在开始 1 秒钟
    demo_timer.data = (unsigned long) devreg; //将设备寄存器指针地址作为参数
    demo_timer.function = &demo_timer_func;
    add_timer(&demo_timer);
    ...
}

//
//用于关闭设备的函数实现
//
static int demo_dev_release(...)
{
    ...
    del_timer_sync(&demo_timer); //删除定时器对象
}

```

```

    ...
}

```

8.3.1 init_timer

在前面的示例代码中，`demo_dev_open` 函数在对定时器对象 `demo_timer` 的 `expires`、`data` 和 `function` 成员赋值前，调用了 `init_timer` 函数（内核源码中以宏定义的形式出现）。`init_timer` 函数内部会调用 `__init_timer`，其定义如下（去除了 `struct timer_list` 中一些调试相关成员的代码）：

```

<kernel/timer.c>
-----
static void __init_timer(struct timer_list *timer,
                        const char *name,
                        struct lock_class_key *key)
{
    timer->entry.next = NULL;
    timer->base = __raw_get_cpu_var(tvec_bases);
    timer->slack = -1;
}

```

可见 `init_timer` 函数主要初始化定时器对象中与内核实现相关的成员，所以设备驱动程序在开始使用定时器对象前，应该调用 `init_timer`，这样从内核层面出发，后续对定时器的一些操作才会被内核所支持，下面在讨论 `add_timer` 函数时会看到这一点。

8.3.2 add_timer

当程序定义了一个定时器对象，并且通过 `init_timer` 函数及相应代码对该定时器对象中的 `expires`、`data` 和 `function` 等成员初始化之后，程序需要调用 `add_timer` 将该定时器对象加入到系统中，这样定时器才会在 `expires` 表示的时间点到期后被触发。可以想见，`add_timer` 函数的内部实现将不再独立，它必然会和内核中关于定时器的基础架构发生关联。

内核自身对于定时器的管理与操作设计有一个非常完整的框架，详细讨论这些技术细节需要相当的篇幅，其中大量的内容属于内核实现的范畴。因此我们决定将后续的讨论限定在设备驱动程序员需要关注的范围之内，也即在更广的范围内我们给出定时器内核实现原理的大体架构，在更细分的范围我们重点讨论与驱动程序中对定时器的使用等密切相关的部分。这样的安排相信对于设备驱动程序员而言是合理的：在了解了基本原理的前提下，通过对内核如何组织和调用到期的定时器函数的讨论，现实中我们将知道如何更安全更有效地使用定时器，这也是写作本书的主要目的。

接下来将首先讨论内核如何管理系统中的定时器，然后会看到定时器函数如何在指定的时间到期后被调用，最后会讨论 `add_timer` 函数是如何将一个定时器对象加入到系统中的。

内核中定义了一个数据结构 `struct tvec_base` 来管理系统中添加的所有定时器，其定义如下：

```
<kernel/timer.c>
-----
struct tvec_base {
    spinlock_t lock;
    struct timer_list *running_timer;
    unsigned long timer_jiffies;
    unsigned long next_timer;
    struct tvec_root tv1;
    struct tvec tv2;
    struct tvec tv3;
    struct tvec tv4;
    struct tvec tv5;
} ____cacheline_aligned;
```

其中的 `tv1`、`tv2`、`tv3`、`tv4` 和 `tv5` 被内核用来对系统中注册的定时器进行散列式的管理，后面会看到其用法。内核为系统中的每个 CPU 都定义了一个 `struct tvec_base` 类型的变量 `tvec_bases`：

```
<kernel/timer.c>
-----
static DEFINE_PER_CPU(struct tvec_base *, tvec_bases) = &boot_tvec_bases;
```

`tvec_bases` 用来将系统中加入的每个定时器组织管理起来。用简单的单一链表结构当然也可以实现这一目标，然而系统会在每个时钟中断中去扫描该链表并要分辨出哪些定时器已经到期或者是即将到期，所以必须使得这一任务的执行效率非常高以消耗极小的 CPU 资源。因此内核采用了上述 `struct tvec_base` 结构来组织链表，读者可以简单地认为它是基于哈希表的一个实现。每当设备驱动程序通过 `add_timer` 向系统添加一个定时器对象时，系统都会对该定时器对象的到期时间 `expires` 进行分类，根据到期时间的长短将当前定时器对象放到 `struct tvec_base` 对象的成员 `tv1`、`tv2`、`tv3`、`tv4` 和 `tv5` 领衔的定时器链表中。比如，其中的 `tv1` 中定时器的到期时间范围是 0~255 个时钟周期，前面已经看到了 `struct tvec_base` 结构的定义，它的成员 `tv1` 其实也是个数组，大小是 256，分别对应 `expires` 为 0~255 个 jiffies 的定时器，如果有多个到期时间相同的定时器，则它们将会以双链表的形式链接到同一数组项中。其他的成员 `tv2`、`tv3`、`tv4` 和 `tv5` 用来存放到到期时间更久的定时器，除此之外与 `tv1` 的原理是一样的。图 8-2 为向系统中添加一个定时器对象的示意图。

至此程序只是完成了向系统添加一个定时器对象的工作，接下来讨论添加的定时器对象在指定的时间到期时如何被触发，也就是定时器对象中的定时器函数何时被调用的问题。

我们知道，Linux 内核一秒中都会发生很多次的时间中断，在每个时钟中断处理函数中，严格地说是时钟中断处理的下半部也就是 `softirq` 部分，会对 `tvec_bases` 管理的定时器队列进行扫描，以确定当前队列中有哪些定时器已经到期。

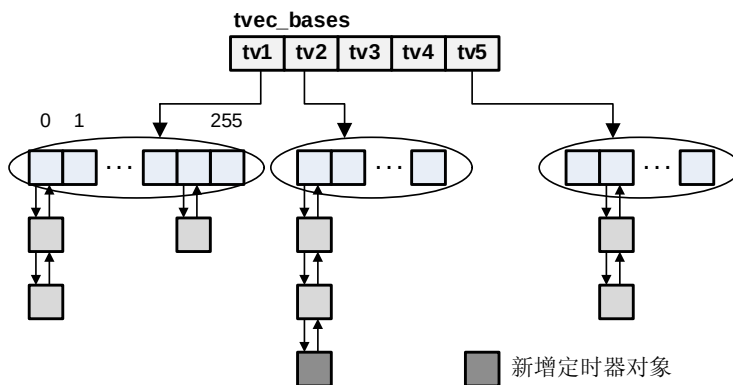


图 8-2 通过 add_timer 向系统新增一个定时器对象

Linux 内核中时钟中断的 softirq 为 TIMER_SOFTIRQ，对应的软中断处理函数的安装发生在系统初始化阶段的 init_timers 函数中：

```
<kernel/timer.c>
void __init init_timers(void)
{
    ...
    open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
}
```

所以，当时钟中断的 softirq 被调度执行时，它将运行对应的 run_timer_softirq 函数。在每个时钟中断处理的上半部分，都会调用 run_local_timers 函数，后者则通过使用 raise_softirq 函数来触发时钟中断的 softirq 部分：

```
<kernel/timer.c>
void run_local_timers(void)
{
    hrtimer_run_queues();
    raise_softirq(TIMER_SOFTIRQ);
    softlockup_tick();
}
```

所以，当时钟中断的 softirq 部分被调度执行时，run_timer_softirq 会负责扫描 tvec_bases 所在的定时器管理队列，找到已经到期的函数，然后调用到期定时器对象节点上的定时器函数：

```
<kernel/timer.c>
static void run_timer_softirq(struct softirq_action *h)
{
    struct tvec_base *base = __get_cpu_var(tvec_bases);
    ...
    if (time_after_eq(jiffies, base->timer_jiffies))
```

```

        __run_timers(base);
    }

```

`run_timer_softirq` 对于那些到期的定时器队列调用 `__run_timers` 函数进一步处理，后者的部分核心代码如下：

```

<kernel/timer.c>
static inline void __run_timers(struct tvec_base *base)
{
    struct timer_list *timer;

    spin_lock_irq(&base->lock);
    while (time_after_eq(jiffies, base->timer_jiffies)) {
        ...
        while (!list_empty(head)) {
            void (*fn)(unsigned long);
            unsigned long data;

            timer = list_first_entry(head, struct timer_list, entry);
            fn = timer->function;
            data = timer->data;
            ...
            detach_timer(timer, 1);

            spin_unlock_irq(&base->lock);
            call_timer_fn(timer, fn, data);
            spin_lock_irq(&base->lock);
        }
    }
}

```

函数的总体思想是，对 `tvec_bases` 管理的定时器队列进行扫描，如果发现有定时器到期（代码中用 `time_after_eq` 来进行判断），则调用该定时器对象的 `fn` 函数（`fn = timer->function, fn(data)`），这个过程发生在 `call_timer_fn` 函数中。读者需要注意，在调用 `call_timer_fn` 前 `__run_timers` 调用了 `detach_timer(timer, 1)`，该函数会把当前正在处理的定时器对象从 `tvec_bases` 中删除，所以当该定时器对象中的定时函数被调用时，该定时器对象已经从系统的定时器队列中删除了，所以如果要想该定时器对象在以后能继续被系统所调用，则需要再次调用 `add_timer` 或者是 `mod_timer` 来将该定时器对象重新加入到系统中去，这是设备驱动程序用定时器来实现轮询机制的基本原理。

通过上面的讨论可以知道，由于内核对系统中的定时器队列的扫描发生在时钟中断的 `softirq` 部分，鉴于 `softirq` 的实现机制⁷，在某些情况下可能会导致当一个定时器对象中的定

⁷ 关于 `softirq` 的实现原理，可以参考“中断处理”一章。

时器函数被调用时，实际的 jiffies 值已经超出了当时安装定时器时预设的 jiffies 值，换句话说，使用定时器也同样存在着实际到期时间点延伸的问题，如果使用当中对定时精度有严格的要求，那么也许要考虑在现有的通用内核上加入某些实时性的扩展。

8.3.3 del_timer 和 del_timer_sync

同 add_timer 函数相反，del_timer 类的函数负责从系统的定时器管理队列中摘除一个定时器对象。del_timer 和 del_timer_sync 的函数原型为：

```
<kernel/timer.c>
-----
int del_timer(struct timer_list *timer);
int del_timer_sync(struct timer_list *timer);
```

del_timer 与 del_timer_sync 函数只在 SMP 系统上才有所区别，在单处理器系统中，del_timer_sync 等同于 del_timer。

对于 del_timer 函数要摘除的定时器对象 timer，函数会首先判断该对象是否是一个 pending 的定时器，一个处于 pending 状态的定时器是处在处理器的定时器管理队列中正等待被调度执行的定时器对象。如果一个要被 del_timer 函数删除的 timer 对象已经被调度执行（内核源码称这种定时器状态为 inactive），函数将直接返回 0，否则函数将通过 detach_timer 将该定时器对象从队列中删除。在多处理器的 SMP 系统中，del_timer_sync 函数要完成的任务除了同 del_timer 一样从定时器队列中删除一个定时器对象外，还会确保当函数返回时系统中没有任何处理器正在执行定时器对象上的定时器函数，而如果只是调用 del_timer，那么当函数返回时，被删除的定时器对象的定时器函数可能正在其他处理器上运行。

8.4 本章小结

本章讨论了设备驱动程序中可能用到的与时间度量和定时相关的话题。这类时间管理相关的任务从总体上可以分成两大类，一类是延迟当前处理器的执行，另一类是设定一个延迟时间点，当该延迟时间点到期后执行特定的动作。

对于延迟的实现，又可以分为“忙等待”和“让出处理器”两种方式，前者是让当前的处理器进入到一个不断的循环中以实现延迟当前的执行，因为处理器在这种循环中无法进行其他任务的处理，所以这种方式会浪费 CPU 的资源。因此为了改善这种处理器浪费的现象，又有了所谓“让出处理器”的延迟方式，相对于“忙等待”，前者会在当前进程的延迟期间让出处理器，这样处理器就可以用来执行别的进程，从而提高其利用率。但如果程序需要的延迟时间非常短，比如只在微秒甚至纳秒级，这种情况如果采用“让出处理器”的方式来实现，那么由于进程切换的时间开销大约也是在微秒这个级别上，所以极有可能当前需要延迟执行进程刚被切换出处理器，延迟的时间就已经到了，此时又需要重新将该进程切

换至处理器，如此效率反而不高。

定时器在设备驱动程序中最常见也最典型的使用场景是用来实现轮询机制，内核为定时器机制设计了一套完整的机制，对于设备驱动程序而言，只需定义一个定时器对象并指定其到期时间及实现一个定时器函数，然后通过 `add_timer` 或者 `mod_timer` 将该定时器对象加到系统中即可。当一个定时器对象到期被执行时，内核会将其从系统的定时器管理队列中摘除下来，所以为了实现轮询，驱动程序需要在定时器函数中重新将该定时器对象加入到管理队列中。因为定时器的实现机制是基于系统中的硬件时钟中断，因为受硬件时钟精度以及时钟中断 `softirq` 固有的实现特性，定时器的精度并非完美，但是对于绝大多数的设备驱动而言已经足够了。



第 9 章

Linux设备驱动模型

到目前为止，所讨论的 Linux 系统下的设备驱动都是独立的，驱动与驱动之间并没有实质性的联系。随着 Linux 系统越来越成熟，与设备驱动相关的一些新的特性需要加入，而之前独立的设备驱动已经无法胜任新形势下的工作，于是 Linux 需要找出一种方式，让系统中的各种设备及其驱动程序能有效地沟通起来，如同人类社会发展那样，孤独的原始人类需要进入群居的时代，于是部落产生了。Linux 为此建立了这样的一种“部落”，这就是本章要讨论的主题：Linux 设备驱动模型。在具体剖析这个模型的每个细节的时候，希望读者脑海中始终抓住一个关键的主题：相对于以前独立的设备驱动开发模式，Linux 的这种新的设备模型到底给系统带来了哪些好处，换言之，为什么要提出这种设备模型的概念。

在本章的讨论中，不会涉及设备驱动模型的每个细节，因为按照笔者的经验，这样的叙述虽然看起来很全面，但是读者读完之后难以在心中建立一个完整的设备模型的全局印象。我们按照主线进行，那是绝大多数 Linux 设备驱动程序员在实际的工作中可能与之打交道的地方。

9.1 sysfs 文件系统

理解 Linux 设备驱动模型也许并不困难，然而如何把这中间错综复杂的关系理清楚讲清楚却并不容易，尤其是以一种通俗易懂的方式阐述其中的设计思想。

本章第一节先讨论 sysfs 文件系统。在作者看来，Linux 设备模型如同一栋规模宏大的建筑，为了构建它，除了基本的建筑材料外（这就是接下来会谈到的 kobject、kset 等基础类数据结构），尚需要一种机制，来向建筑外面的世界（用户空间的程序）展示内部的构造，并且通过文件接口的方式实现与外界的沟通与互动。sysfs 文件系统就充当了这种角色，它不但在各种基础的建筑材料之间建立彼此的互联层次关系，而且向外界提供了与建筑内设施进行互动的文件接口。这种形象的比喻反映到 Linux 系统，我们可以看到 sysfs 文件除了在内核空间所展现的合纵连横作用外，而且以文件目录层次结构的形式向用户空间提供了系统硬件设备间的一个拓扑图，这种文件形式的接口也让用户空间与内核空间的数据对象的交互成为可能。读者如果熟悉 proc 文件系统的话，应当知道 sysfs 文件系统实际上取代了 proc

文件系统的功能，当然取代 `proc` 文件系统只是 `sysfs` 文件系统一小部分的功能而已。这种数据对象间的交互的一个具体的例子就是，透过 `sysfs` 文件系统可以取代 `ioctl` 的功能：在本书前面的章节曾经讨论过 `ioctl` 的实现，在那里，如果向一个设备文件发送 `ioctl` 命令的话，需要首先打开该设备文件，然后再通过 `ioctl` 函数向设备发出命令，很显然需要一个完整（虽然代码可能很简单）的应用程序来做这件事，现在有了 `sysfs` 文件系统，一个很简单的 `shell` 命令也许就可以完成前面所说的工作。

虽然作者在这里说得有点天花乱坠，但是仅凭这些语句根本无法打消读者心中的疑虑与忧郁，的确，Linux 下的设备驱动模型是个复杂的系统，理解它包容它需要我们有足够的耐心。然而也许具体的例子才是解说 `sysfs` 文件系统在 Linux 设备驱动模型中作用的最好方法，所以在下面的讨论中我们会刻意试图做到这一点。

现在我们打算从这个文件系统的起源开始谈起，但是我们不会讨论 `sysfs` 文件系统实现的细节，因为像 `read`、`write` 和创建一个目录等等，属于内核中文件系统应该要讨论的范围，而且这些操作的原理都大同小异，在这些方面放置过多的篇幅对读者理解 Linux 设备驱动模型并无裨益。我们这里对 `sysfs` 文件的讨论，是希望读者在读完本节后对 `sysfs` 在系统中的地位有个全局性的认识，这样当我们在后续的设备模型高级阶段的讨论中，读者理解起来也许就会轻松那么一点。

`sysfs` 文件系统的初始化发生在 Linux 系统的启动阶段：

```
<fs/sysfs/mount.c>
-----
int __init sysfs_init(void)
{
    ...
    err = register_filesystem(&sysfs_fs_type);
    if (!err) {
        sysfs_mount = kern_mount(&sysfs_fs_type);
    } else
        goto out_err;
    ...
}
```

实际的 `sysfs_init` 函数源代码绝不会如此不堪，不过核心的东西都包含在这里了。

函数将向系统注册一个类型为 `sysfs_fs_type` 的文件系统，`sysfs_fs_type` 的定义为：

```
<fs/sysfs/mount.c>
-----
static struct file_system_type sysfs_fs_type = {
    .name      = "sysfs",
    .get_sb    = sysfs_get_sb,
    .kill_sb   = sysfs_kill_sb,
};
```

关于这个结构没有多少需要解释的地方,唯一可能要注意的地方是 `sysfs_fs_type` 中的 `get_sb` 成员,它指向函数 `sysfs_get_sb`,感兴趣的读者可以自己看看这个函数的源码实现,它实际上在内核空间创造了一棵独立的 VFS 树(关于 VFS 的内核机制,读者可以参考文章 <http://www.embexperts.com/viewthread.php?tid=4&extra=page%3D1>),内核创建这棵 VFS 树主要用来沟通系统中总线、设备与驱动,同时向用户空间提供接口及展示系统中各种设备的拓展视图等,事实上它并不用来作为其他实际文件系统的挂载点。

`sysfs_get_sb` 函数用来产生 `sysfs` 文件系统的超级块,其内部调用的最主要的函数是 `sysfs_fill_super`,后者再经过一系列的函数调用链进入到 `sysfs_init_inode` 函数,这里之所以重点强调这个函数,是因为在接下来谈到内核对象的属性问题时会看到用户空间和内核对象的沟通问题,这种文件接口形式的交互发生在内核空间和用户空间,所以我们需要知道这条沟通的通道是如何建立起来的。在 `sysfs_init_inode` 中,函数将为 `sysfs` 文件系统中每个文件或目录所对应的 `inode` 赋予一个新的操作对象:

<fs/sysfs/inode.c>

```
static void sysfs_init_inode(struct sysfs_dirent *sd, struct inode *inode)
{
    struct bin_attribute *bin_attr;

    inode->i_private = sysfs_get(sd);
    inode->i_mapping->a_ops = &sysfs_aops;
    inode->i_mapping->backing_dev_info = &sysfs_backing_dev_info;
    inode->i_op = &sysfs_inode_operations;

    set_default_inode_attr(inode, sd->s_mode);
    sysfs_refresh_inode(sd, inode);

    /* initialize inode according to type */
    switch (sysfs_type(sd)) {
    case SYSFS_DIR:
        inode->i_op = &sysfs_dir_inode_operations;
        inode->i_fop = &sysfs_dir_operations;
        break;
    case SYSFS_KOBJ_ATTR:
        inode->i_size = PAGE_SIZE;
        inode->i_fop = &sysfs_file_operations;
        break;
    case SYSFS_KOBJ_BIN_ATTR:
        bin_attr = sd->s_bin_attr.bin_attr;
        inode->i_size = bin_attr->size;
        inode->i_fop = &bin_fops;
        break;
    case SYSFS_KOBJ_LINK:
        inode->i_op = &sysfs_symlink_inode_operations;
        break;
```



```

        default:
            BUG();
        }

        unlock_new_inode(inode);
    }

```

读者将在本章稍后对内核对象属性的讨论中看到这个函数的用途。

sysfs 文件系统是个基于 **RAM** 实现的文件系统，如果编译内核时指定了 **CONFIG_SYSFS** 选项，那么这个文件系统就会包含到内核中。对于用户进程中的文件系统来说，**sysfs** 的标准挂载点是“**/sys**”目录，将 **sysfs** 文件系统挂载到用户进程的“**/sys**”目录的命令为：

```
mount -t sysfs sysfs /sys
```

如此，所有内核层面的对 **sysfs** 文件树的操作，都将一成不变地显示在用户空间的“**/sys**”目录下。

接下来照理应该讨论 **sysfs** 文件系统在 Linux 设备驱动模型中的具体作用，换句话说，在 Linux 设备驱动模型中，内核基于 **sysfs** 文件系统之上都会有哪些操作，但是因为目前大部分主角都还没有登场，所以不妨把这种讨论稍稍推迟。

9.2 kobject 和 kset

本节将讨论 Linux 实现设备驱动模型的底层数据结构 **kobject** 和 **kset**，在后面讨论总线、设备与驱动时，会经常看到对这些底层数据结构的操作，因此虽然本节的讨论非常抽象，但是为了清楚理解整个 Linux 设备驱动模型的实现机制，用一定量的篇幅来讲述这些抽象的概念还是值得的，不过幸运的是，作为一名设备驱动程序员，基本上不会与这些属于建筑内部框架结构设计的函数接口有照面的机会，所以好奇心不强的读者跳过此节也无妨。

如果将 Linux 设备模型比喻成一座大厦，那么 **kobject** 和 **kset** 就是构成这座大厦内部的钢筋及由若干钢筋构建的钢架结构，再由若干的它们构成了整座大厦内部的表现形式，设备驱动模型中的 **bus**、**device** 和 **driver** 已经是整座大厦向外界展示的那部分了，所以程序员们主要是和后三者打交道。

9.2.1 kobject

Linux 内核用 **kobject** 来表示一个内核对象，它在源码中的定义为：

```

<include/linux/kobject.h>
-----
struct kobject {
    const char      *name;

```

```

    struct list_head entry;
    struct kobject      *parent;
    struct kset         *kset;
    struct kobj_type    *ktype;
    struct sysfs_dirent *sd;
    struct kref         kref;
    unsigned int state_initialized:1;
    unsigned int state_in_sysfs:1;
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
    unsigned int uevent_suppress:1;
};

```

在此先简单解释一下其每位成员的作用：

```
const char      *name
```

用来表示该内核对象的名称。如果该内核对象加入系统，那么它的 `name` 将会出现在 `sysfs` 文件系统中（表现形式是一个新的目录名）。

```
struct list_head entry
```

用来将一系列的内核对象构成链表。

```
struct kobject      *parent
```

指向该内核对象的上层节点。通过引入该成员构建内核对象之间的层次化关系。

```
struct kset         *kset
```

当前内核对象所属的 `kset` 对象的指针。`kset` 对象代表一个 `subsystem`，其中容纳了一系列同类型的 `kobject` 对象。

```
struct kref         kref
```

其核心数据是一原子型变量，用来表示内核对象的引用计数。内核通过该成员追踪内核对象的生命周期。

```
struct kobj_type    *ktype
```

定义了该内核对象的一组 `sysfs` 文件系统相关的操作函数和属性。显然不同类型的内核对象会有不同的 `ktype`，用以体现 `kobject` 所代表的内核对象的特质。通过该成员，C 中的 `struct` 数据类型具备了 C++ 中 `class` 类型的某些特点，这里体现了基于 C 的面向对象设计思想。同时，内核通过 `ktype` 成员将 `kobject` 对象的 `sysfs` 文件操作与其属性文件关联起来。

```
struct sysfs_dirent *sd
```

用来表示该内核对象在 sysfs 文件系统中对应的目录项的实例。

unsigned int state_initialized

表示该 kobject 所代表的内核对象初始化的状态，1 表示对象已被初始化，0 表示尚未初始化。

unsigned int state_in_sysfs

表示该 kobject 所代表的内核对象有没有在 sysfs 文件中建立一个入口点。

unsigned int uevent_suppress

如果该 kobject 对象隶属于某一 kset，那么它的状态变化可以导致其所在的 kset 对象向用户空间发送 event 消息。成员 uevent_suppress 用来表示当该 kobject 状态发生变化时，是否让其所在的 kset 向用户空间发送 event 消息。值 1 表示不让 kset 发送这种 event 消息。

kobject 数据结构最通用的用法是嵌在表示某一对象的数据结构中，比如内核中定义的字符型设备对象 cdev 中就嵌入了 kobject 结构：

```
<include/linux/cdev.h>
-----
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```

下面介绍内核中定义的对 kobject 对象上的一些常用的操作函数。需要提醒读者的是，设备驱动程序一般不会与这些底层的函数直接打交道，这里简单介绍这些函数是希望读者能大致了解其功能，因为在讨论到设备驱动模型的高层框架时，它们将经常被提及。当然，如果的确有需要，也可以直接在设备驱动程序模块中调用这些底层的函数，本节后面会给出一些在驱动程序中调用这些底层函数的例子。

○ kobject_set_name

该函数用来设定 kobject 中的 name 成员，函数原型为：

```
int kobject_set_name(struct kobject *kobj, const char *fmt, ...)
```

○ kobject_init

该函数用来初始化一个内核对象的 kobject 结构，其核心功能代码为（去除了一些参数检查等的代码）：

```
<lib/kobject.c>
```

```
void kobject_init(struct kobject *kobj, struct kobj_type *ktype)
{
    ...
    kobject_init_internal(kobj);
    kobj->ktype = ktype;
    return;
}
```

除了为 `kobj` 指定 `ktype` 成员外，真正的初始化工作发生在 `kobject_init_internal` 中：

```
<lib/kobject.c>
```

```
static void kobject_init_internal(struct kobject *kobj)
{
    if (!kobj)
        return;
    kref_init(&kobj->kref);
    INIT_LIST_HEAD(&kobj->entry);
    kobj->state_in_sysfs = 0;
    kobj->state_add_uevent_sent = 0;
    kobj->state_remove_uevent_sent = 0;
    kobj->state_initialized = 1;
}
```

函数中的 `kref_init(&kobj->kref)` 用于将 `kobject` 的引用计数 `refcount` 初始化为 1，`state_initialized` 置为 1 表示该内核对象已被初始化，`state_in_sysfs` 置为 0 表示该内核对象尚未出现在 `sysfs` 文件树中。

○ `kobject_add`

函数原型为：

```
int kobject_add(struct kobject *kobj, struct kobject *parent, const char *fmt, ...)
```

对于 `kobject` 来讲，这是个非常重要的函数。因为内核源码中这个函数调用链比较烦琐，所以此处不再列出其源码，我们解释该函数的主要功能，然后给出作为示范性质的代码片段。

`kobject_add` 的主要功能有两个，一是建立 `kobject` 对象间的层次关系，二是在 `sysfs` 文件系统中建立一个目录。在将一个 `kobject` 对象通过 `kobject_add` 函数调用加入系统前，`kobject` 对象必须已被初始化。

关于这两个功能的实现细节，`kobject_add` 首先将参数 `parent` 赋值给 `kobj` 的 `parent` 成员 `kobj->parent = parent`，然后调用 `kobject_add_internal(kobj)` 函数。在 `kobject_add_internal` 函数内部，如果调用 `kobject_add` 时 `parent` 是一 `NULL` 指针，那么要看该 `kobj` 是否在一个 `kset` 对象中：如果是就把该 `kset` 中的 `kobject` 成员作为 `kobj` 的 `parent`；否则该 `kobj` 对象在 `sysfs`

文件树中就将处于根目录的位置。

<lib/kobject.c>

```
static int kobject_add_internal(struct kobject *kobj)
{
    ...
    parent = kobject_get(kobj->parent);
    //在 kobj 有所属的 kset 的情况下
    if (kobj->kset) {
        //如果调用 kobject_add 时，传入的 parent 参数是一 NULL 指针
        if (!parent)
            //就把 kobj 所在的 kset 中的 kobj 作为它的 parent
            parent = kobject_get(&kobj->kset->kobj);
        //将 kobj 加入到所属 kset 链表的末尾
        kobj_kset_join(kobj);
        kobj->parent = parent;
    }
    //在 kobj 没有所属的 kset 的情况下，如果调用 kobject_add 时 parent 为 NULL
    //那么 kobj->parent 也将为 NULL
    ...
}
```

kobject_add 接下来会调用 sysfs_create_dir 在 sysfs 文件树中创建目录：

<fs/sysfs/dir.c>

```
int sysfs_create_dir(struct kobject * kobj)
{
    ...
    if (kobj->parent)
        parent_sd = kobj->parent->sd;
    else
        parent_sd = &sysfs_root;
    ...
    error = create_dir(kobj, parent_sd, type, ns, kobject_name(kobj), &sd);
    if (!error)
        kobj->sd = sd;
    return error;
}
```

可以看到，如果 kobj->parent 为 NULL（刚刚在 kobject_add_internal 函数中讨论过这种情况），调用 create_dir 在 sysfs 文件树中为当前 kobj 创建目录时，parent_sd = &sysfs_root，否则 parent_sd = kobj->parent->sd。parent_sd = &sysfs_root 意味着在 sysfs 文件树的根目录下为 kobj 创建一个新的目录，否则就是在 parent_sd 对应的目录底下创建新目录。如果 kobj 在 sysfs 中成功创建了一个新目录，自然应该将 kobj->state_in_sysfs 置为 1。

在 sysfs 文件系统中，目录对应的数据结构为 struct sysfs_dirent，函数中用 sd 表示该类型的

一个实例，在将 kobj 对象加入 sysfs 文件树之后，kobj->sd = sd。

○ kobject_init_and_add

函数原型为：

```
int kobject_init_and_add(struct kobject *kobj, struct kobj_type *ktype,
                        struct kobject *parent, const char *fmt, ...)
```

该函数实际的工作是将 kobject_init 和 kobject_add 两个函数的功能合并到了一起。

○ kobject_create

该函数用来分配并初始化一个 kobject 对象：

```
<lib/kobject.c>
-----
struct kobject *kobject_create(void)
{
    struct kobject *kobj;

    kobj = kzalloc(sizeof(*kobj), GFP_KERNEL);
    if (!kobj)
        return NULL;

    kobject_init(kobj, &dynamic_kobj_ktype);
    return kobj;
}
```

如果调用 kobject_create 来产生一个 kobject 对象，那么调用者将无法为该 kobject 对象另行指定 kobj_type。kobject_create 为产生的 kobject 对象指定了一个默认的 kobj_type 对象 dynamic_kobj_ktype，这个行为将影响 kobject 对象上的 sysfs 文件操作。如果调用者需要明确指定一个自己的 kobj_type 对象给该 kobject 对象，那么还应该使用其他函数，比如调用 kobject_init_and_add 函数。

○ kobject_create_and_add

函数内部首先调用 kobject_create 来分配并初始化一个 kobject 对象，然后再调用 kobject_add 函数在 sysfs 文件系统中为新生成的 kobject 对象建立一个新的目录：

```
<lib/kobject.c>
-----
struct kobject *kobject_create_and_add(const char *name, struct kobject *parent)
{
    struct kobject *kobj;
    int retval;

    kobj = kobject_create();
```

```

    if (!kobj)
        return NULL;

    retval = kobject_add(kobj, parent, "%s", name);
    if (retval) {
        printk(KERN_WARNING "%s: kobject_add error: %d\n",
               __func__, retval);
        kobject_put(kobj);
        kobj = NULL;
    }
    return kobj;
}

```

○ kobject_del

函数的实现代码为：

```

<lib/kobject.c>
-----
void kobject_del(struct kobject *kobj)
{
    if (!kobj)
        return;

    sysfs_remove_dir(kobj);
    kobj->state_in_sysfs = 0;
    kobj_kset_leave(kobj);
    kobject_put(kobj->parent);
    kobj->parent = NULL;
}

```

函数将在 sysfs 文件树中把 kobj 对应的目录删除，另外如果 kobj 隶属于某一 kset 的话，将其从 kset 的链表中删除。

以上介绍了内核针对 kobject 对象定义的一些常见的操作函数，很枯燥也很抽象。现在我们来试着做点看起来可能比较有趣的事情，在设备驱动程序中调用 kobject_create_and_add 在 sysfs 文件树中生成一个目录：

```

//先声明一个 kobject 对象指针 parent
static struct kobject *parent = NULL;

static int __init kobj_demo_init(void)
{
    ...
    parent = kobject_create_and_add("pa_obj", NULL);
    ...
}

module_init(kobj_demo_init);

```

在把上述模块加载到系统之后，就可在/sys 目录下看到一名为“pa_obj”的新目录：

```
root@AMDLinuxFGL:/sys# ls -l
total 0
...
drwxr-xr-x  2 root root 0 Jun 18 19:03 pa_obj
drwxr-xr-x  2 root root 0 Jun 18 15:04 power
```

如果在调用 kobject_create_and_add 时指定了 parent 参数，那么新 kobject 对象所对应的目录将建立在 parent 目录之下。比如把上面的代码改成：

```
//声明两个 kobject 对象，层次结构为父子关系
static struct kobject *parent = NULL;
static struct kobject *child = NULL;

static int __init kobj_demo_init(void)
{
    ...
    parent = kobject_create_and_add("pa_obj", NULL);
    //指定 child kobj 的 parent
    child = kobject_create_and_add("cld_obj", parent);
    ...
}
```

加载上面的代码将会在 sysfs 树中生成两个新的目录，体现在用户空间的/sys 目录下便是 /sys/pa_obj 目录和/sys/pa_obj/cld_obj 目录。

如果读者想试验上面的代码，记得要在模块的退出函数中调用 kobject_del(child)和 kobject_del(parent)将 child 与 parent 所指向的对象删除，这样/sys 目录下的“cld_obj”和“pa_obj”目录才会消失，否则在删除这个新目录时会遇上麻烦，因为 sysfs 文件系统没有为用户进程提供删除目录的接口。

通过上面对于 kobject 上一些主要操作函数的讨论，可以知道将一个 kobject 对象添加进系统或者从系统中删除，主要是围绕 sysfs 文件系统展开的，对应的结果反映到/sys 目录中就是一个新目录的诞生或者是一个已存在目录的消亡。这种对 sysfs 文件树的操作的现实意义除了向用户空间展示不同 kobject 对象之间的层次关系外，还在于用户空间的程序可以通过文件系统的接口配置内核空间 kobject 对象的某些属性，这是下一节要讨论的主题。

9.2.2 kobject 的类型属性

kobject 数据结构中内嵌有一个 struct kobj_type 类型的成员*ktype，在内核中 struct kobj_type 的定义为：


```
<include/linux/kobject.h>
```

```
struct kobj_type {
    void (*release)(struct kobject *kobj);
    const struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
    const struct kobj_ns_type_operations *(*child_ns_type)(struct kobject *kobj);
    const void *(*namespace)(struct kobject *kobj);
};
```

关于内核对象 `kobject` 的命名空间 (`namespace`) 问题, 本书不作讨论。下面重点看 `kobj_type` 的前三个成员函数, `release` 显然是一个函数指针, 成员 `sysfs_ops` 是一 `struct sysfs_ops` 类型的指针, `struct sysfs_ops` 的定义为:

```
<include/linux/sysfs.h>
```

```
struct sysfs_ops {
    ssize_t (*show)(struct kobject *, struct attribute *, char *);
    ssize_t (*store)(struct kobject *, struct attribute *, const char *, size_t);
};
```

所以 `sysfs_ops` 实际上定义了一组针对 `struct attribute` 对象的操作函数的集合, `struct attribute` 数据结构则是为 `kobject` 内核对象定义的属性成员, 它在源码中的定义是:

```
<include/linux/sysfs.h>
```

```
struct attribute {
    const char *name;
    mode_t mode;
};
```

记得前面讨论 `kobject_init` 函数初始化一个内核对象 `kobject` 的时候, 会同时赋予它一个具体的 `struct kobj_type` 对象成员, 那么现在的问题是, 内核会如何使用 `kobject` 的这个成员呢?

对这个问题的探讨其实关系到内核把一个 `kobject` 对象加入到 `sysfs` 文件树中的使用意图。

为一个 `kobject` 对象创建一个属性文件使用的函数为 `sysfs_create_file`:

```
<fs/sysfs/file.c>
```

```
int sysfs_create_file(struct kobject *kobj, const struct attribute *attr)
{
    BUG_ON(!kobj || !kobj->sd || !attr);
    return sysfs_add_file(kobj->sd, attr, SYSFS_KOBJ_ATTR);
}
```

在使用这个函数时, 必须确保要添加属性文件的 `kobj` 对象之前已经加入了 `sysfs` (也即 `kobj->state_in_sysfs = 1`), `sysfs_add_file` 函数将在 `kobj->sd` 对应的目录下生成一个属性文件。如果以先前的 “`cld_obj`” 内核对象为基础, 在其下添加一个属性文件 “`cldatt`”, 可以使用

下面的代码：

```
static struct attribute cld_att = {
    .name = "cldatt",
    .mode = S_IRUGO | S_IWUSR,
};

sysfs_create_file(child, &cld_att);
```

运行上面的代码后，可在/sys/pa_obj/cld_obj 目录下看到一名为“cldatt”的属性文件，如下：

```
root@AMDLinuxFGL:/sys/pa_obj/cld_obj# ls -l
total 0
-rw-r--r-- 1 root root 4096 Jun 18 21:25 cldatt
```

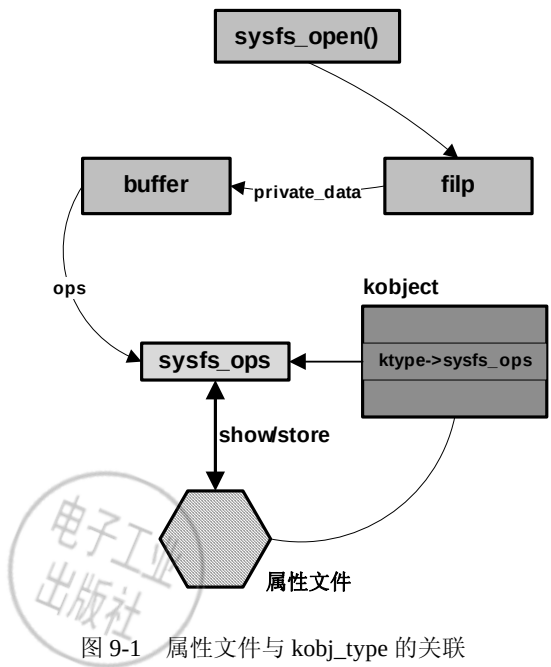
用户空间的程序在使用一个内核对象 kobject 的属性文件时，会首先 open 这个属性文件，比如 open("cldatt", O_RDONLY | O_LARGEFILE)，然后通过系统调用等一系列潜在的调用链，这个函数最终会调用到 sysfs_open_file：

```
<fs/sysfs/file.c>
static int sysfs_open_file(struct inode *inode, struct file *file)
{
    struct sysfs_dirent *attr_sd = file->f_path.dentry->d_fsdata;
    struct kobject *kobj = attr_sd->s_parent->s_dir.kobj;
    struct sysfs_buffer *buffer;
    const struct sysfs_ops *ops;
    ...
    if (kobj->ktype && kobj->ktype->sysfs_ops)
        ops = kobj->ktype->sysfs_ops;
    else {
        WARN(1, KERN_ERR "missing sysfs attribute operations for "
             "kobject: %s\n", kobject_name(kobj));
        goto err_out;
    }
    ...
    buffer = kzalloc(sizeof(struct sysfs_buffer), GFP_KERNEL);
    buffer->needs_read_fill = 1;
    buffer->ops = ops;
    file->private_data = buffer;
    ...
}
```

函数明确地使用到了 kobj 对象上的成员 ktype，将 ktype->sysfs_ops 赋值给了 ops：ops = kobj->ktype->sysfs_ops。然后通过新分配的 buffer 空间，间接地将 ktype->sysfs_ops 放到了 file 的 private_data 成员中：file->private_data = buffer。这样用户空间的程序在后续对该属性文件

的 read 操作中，将会利用 file->private_data 来获得 kobj->ktype 中定义的显示 kobj 属性值的 show 函数，如果要改变 kobj 的某一属性值，则应该使用 sysfs_ops 中的 store 函数。

内核对象 kobj 属性文件的创立与来自 kobj->kobj_type 上针对属性文件的读写操作的关系可以用图 9-1 来表达：



如果要删除一个属性文件，则应该使用如下函数：

```
void sysfs_remove_file(struct kobject * kobj, const struct attribute * attr);
```

所以，内核通过 kobject 属性文件的方式给用户空间程序提供了一种显示与更新某一内核对象 kobject 上的属性信息的接口。

9.2.3 kset

kset 可以认为是一组 kobject 的集合，是 kobject 的容器。kset 本身也是一个内核对象，所以需要内嵌一个 kobject 对象。其完整定义如下：

```
<include/linux/kobject.h>
-----
struct kset {
    struct list_head list;
    spinlock_t list_lock;
    struct kobject kobj;
    const struct kset_uevent_ops *uevent_ops;
};
```

struct list_head list

用来将其中的 kobject 对象构建成链表。

spinlock_t list_lock

对 kset 上的 list 链表进行访问操作时用来作为互斥保护使用的自旋锁。

struct kobject kobj

代表当前 kset 内核对象的 kobject 变量。

const struct kset_uevent_ops *uevent_ops

定义了一组函数指针, 当 kset 中的某些 kobject 对象发生状态变化需要通知用户空间时, 调用其中的函数来完成。struct kset_uevent_ops 类型声明如下:

```
<include/linux/kobject.h>
-----
struct kset_uevent_ops {
    int (* const filter)(struct kset *kset, struct kobject *kobj);
    const char *(* const name)(struct kset *kset, struct kobject *kobj);
    int (* const uevent)(struct kset *kset, struct kobject *kobj,
        struct kobj_uevent_env *env);
};
```

kset 上的一些主要操作有:

kset_init

用来初始化一个 kset 对象, 函数原型为:

```
void kset_init(struct kset *k)
```

kset_register

用来初始化并向系统注册一个 kset 对象, 函数的实现如下:

```
<lib/kobject.c>
-----
int kset_register(struct kset *k)
{
    int err;
    if (!k)
        return -EINVAL;
    kset_init(k);
    err = kobject_add_internal(&k->kobj);
    if (err)
        return err;
    kobject_uevent(&k->kobj, KOBJ_ADD);
}
```

```

    return 0;
}

```

其中 `kset_init` 和 `kobject_add_internal` 的功能都比较直观，分别用来初始化 `kset` 对象和向系统注册该 `kset` 对象，因为 `kset` 对象本身就是一个由 `kobject` 代表的内核对象，所以 `kobject_add_internal` 函数会为代表该 `kset` 对象的 `k->kobj` 在 `sysfs` 文件树中生成一个新目录，这个过程同前面谈到的 `kobject` 的操作是完全一样的。

`kset` 对象与单个的 `kobject` 对象不一样的地方在于，将一个 `kset` 对象向系统注册时，如果 Linux 内核编译时启用了 `CONFIG_HOTPLUG`，那么需要将这一事件通知用户空间，这个过程由 `kobject_uevent` 完成。如果一个 `kobject` 对象不属于任一 `kset`，那么这个孤立的 `kobject` 对象将无法通过 `uevent` 机制向用户空间发送 `event` 消息。

作为下文讨论的基础，下面先看看图 9-2 所示 `kobject` 与 `kset` 的层次关系：

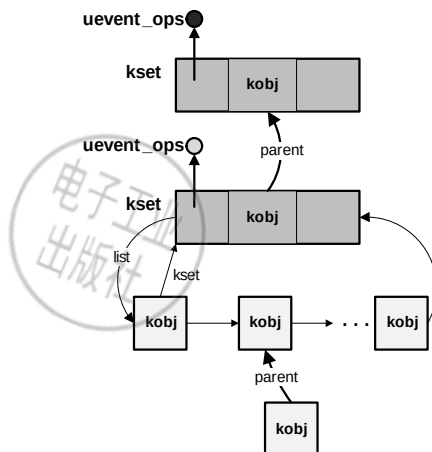


图 9-2 `kobject` 与 `kset` 的层次关系

图中，`kobj` 之间通过 `parent` 成员实现层次关系，如果某一 `kobj` 的 `parent` 为 `NULL`，那么在调用 `kobject_add` 函数将该 `kobj` 加入系统时，函数首先看 `kobj->kset` 是否为 `NULL`，如果不为 `NULL`，就会把 `kobj->kset->kobj` 作为 `kobj` 的 `parent`，否则系统中将产生一个孤立的 `kobject` 对象，该对象将无法通过 `uevent` 机制向用户空间发送 `event` 消息。`kset` 将所有隶属于它的 `kobject` 对象放到一个链表 `list` 中，同时可以看到 `kset` 的数据结构中内嵌了一个 `kobject` 成员，所以 `kset` 自身也是作为一个内核对象而存在。

`kset` 上其他的一些操作包括：

○ `kset_create_and_add`

函数原型为：

```

struct kset *kset_create_and_add(const char *name,

```

```
const struct kset_uevent_ops *uevent_ops,
struct kobject *parent_kobj)
```

主要作用是动态产生一 `kset` 对象然后将其加入到 `sysfs` 文件系统中。参数 `name` 是创建的 `kset` 对象的名称, `uevent_ops` 是新 `kset` 对象上用来处理用户空间 `event` 消息的操作集, `parent_kobj` 是 `kset` 对象的上层（父级）的内核对象指针。

○ `kset_unregister`

函数原型为：

```
void kset_unregister(struct kset *k)
```

用来将 `k` 指向的 `kset` 对象从系统中注销，完成的是 `kset_register` 的反向操作。

9.2.4 热插拔中的 `uevent` 和 `call_usermodehelper`

这里的热插拔（`hotplug`）可以简单描述为，当一个设备动态加入系统时（典型地如用户将一个 `USB` 盘插到计算机上），设备驱动程序可以检查到这种设备状态的变化（加入或者移除），然后通过某种机制使得在用户空间找到该设备对应的驱动程序模块并加载之。在 `Linux` 系统上有两种机制可以在设备状态发生变化时，通知用户空间去加载或者卸载该设备所对应的驱动程序模块：一个是 `udev`，另一个是 `/sbin/hotplug`。在 `Linux` 发展的早期阶段，用户空间支持热插拔的唯一工具是 `/sbin/hotplug`，它的幕后推手是 `call_usermodehelper` 函数，后者能够从内核空间启动一个用户空间的应用程序。随着内核的发展演进，后来又发展出了 `udev` 机制并逐渐取代了 `/sbin/hotplug`，现在 `udev` 工具包已成为大多数 `Linux` 发行版本中首选的方法。`udev` 的实现基于内核中的网络机制，它通过创建标准的 `socket` 接口来监听来自内核的网络广播包，并对接收到的包进行分析处理。

恰如刚才所提到的，两种机制都必须得到来自内核空间的支持才可以工作，因为本书主要讨论设备驱动的内核机制，所以对用户空间的这些工具不作过多描述，接下来讨论的重点是设备驱动程序如何在内核空间对这些工具给予支持。

`Linux` 设备模型中一个非常重要的功能便是对设备热插拔特性的支持，具体到底层的实现细节上，热插拔在内核中通过一个名为 `kobject_uevent` 的函数来实现。它通过发送一个 `uevent` 消息和调用 `call_usermodehelper` 来与用户空间进行沟通。`kobject_uevent` 所实现的功能和 `Linux` 系统中用以实现热插拔的特性息息相关，它是 `udev` 和 `/sbin/hotplug` 等工具赖以工作的基石。所以有足够的理由让我们用出一定的篇幅来仔细讨论一下 `kobject_uevent` 函数，该函数在内核中的实现为：

```
<lib/kobject_uevent.c>
```

```
int kobject_uevent(struct kobject *kobj, enum kobject_action action)
{
```

```

        return kobject_uevent_env(kobj, action, NULL);
    }

```

参数 `action` 是个枚举型变量，其类型定义为：

```
<include/linux/kobject.h>
```

```

enum kobject_action {
    KOBJ_ADD,
    KOBJ_REMOVE,
    KOBJ_CHANGE,
    KOBJ_MOVE,
    KOBJ_ONLINE,
    KOBJ_OFFLINE,
    KOBJ_MAX
};

```

这些枚举数值定义了 `kset` 对象的一些状态变化，此处使用的是 `KOBJ_ADD`，表明将向系统添加一个 `kset` 对象。

`kobject_uevent` 函数的主体功能是在 `kobject_uevent_env` 调用中完成的，这个函数的实现比较长，这里只列出它的部分核心功能代码：

```
<lib/kobject_uevent.c>
```

```

int kobject_uevent_env(struct kobject *kobj, enum kobject_action action,
                      char *envp_ext[])
{
    struct kobj_uevent_env *env;
    ...
    //此处的 while 循环用来查找 kobj 所隶属的最顶层 kset
    top_kobj = kobj;
    while (!top_kobj->kset && top_kobj->parent)
        top_kobj = top_kobj->parent;
    //如果当前 kobj 没有隶属的 kset，那么它将不能使用 uevent 机制
    if (!top_kobj->kset) {
        pr_debug("kobject: '%s' (%p): %s: attempted to send uevent "
                "without kset!\n", kobject_name(kobj), kobj,
                __func__);
        return -EINVAL;
    }
    //得到 kobj 所隶属的顶层 kset 的 uevent 操作集对象 uevent_ops
    kset = top_kobj->kset;
    uevent_ops = kset->uevent_ops;
    ...
    //如果 kobj->uevent_suppress=1，表明该 kobj 不希望使用 uevent 机制
    if (kobj->uevent_suppress) {
        pr_debug("kobject: '%s' (%p): %s: uevent_suppress "

```

```

        "caused the event to drop!\n",
        kobject_name(kobj), kobj, __func__);
    return 0;
}
//首先调用 filter 函数，如果函数返回 0，表明 kobj 希望发送的 event 消息被顶层 kset
//过滤掉了
if (uevent_ops && uevent_ops->filter)
    if (!uevent_ops->filter(kset, kobj)) {
        pr_debug("kobject: '%s' (%p): %s: filter function "
            "caused the event to drop!\n",
            kobject_name(kobj), kobj, __func__);
        return 0;
    }
...
//准备使用 uevent 机制向用户空间发送 event 消息，通过 add_uevent_var 添加环境变量
//信息
env = kzalloc(sizeof(struct kobj_uevent_env), GFP_KERNEL);
retval = add_uevent_var(env, "ACTION=%s", action_string);
if (retval)
    goto exit;
retval = add_uevent_var(env, "DEVPATH=%s", devpath);
if (retval)
    goto exit;
retval = add_uevent_var(env, "SUBSYSTEM=%s", subsystem);
if (retval)
    goto exit;

//此处在向用户空间发送 event 消息之前，给 kset 最后一次机会以完成一些私人事情
if (uevent_ops && uevent_ops->uevent) {
    retval = uevent_ops->uevent(kset, kobj, env);
    if (retval) {
        pr_debug("kobject: '%s' (%p): %s: uevent() returned "
            "%d\n", kobject_name(kobj), kobj,
            __func__, retval);
        goto exit;
    }
}
...
//如果配置了 CONFIG_NET 宏，表明内核打算使用 netlink 机制实现 uevent 消息的发送1
#ifdef CONFIG_NET
/* send netlink message */
mutex_lock(&uevent_sock_mutex);

```

¹ 如果没有特别的理由，CONFIG_NET 都应该被选中，即使运行 Linux 的机器不打算与外部世界进行网络连接，一些工具比如 udev 也需要有网络子系统的支持才能工作。


```

list_for_each_entry(ue_sk, &uevent_sock_list, list) {
    struct sock *uevent_sock = ue_sk->sk;
    struct sk_buff *skb;
    size_t len;

    /* allocate message with the maximum possible size */
    len = strlen(action_string) + strlen(devpath) + 2;
    skb = alloc_skb(len + env->buflen, GFP_KERNEL);
    if (skb) {
        char *scratch;

        /* add header */
        scratch = skb_put(skb, len);
        sprintf(scratch, "%s@%s", action_string, devpath);

        /* copy keys to our continuous event payload buffer */
        for (i = 0; i < env->envp_idx; i++) {
            len = strlen(env->envp[i]) + 1;
            scratch = skb_put(skb, len);
            strcpy(scratch, env->envp[i]);
        }

        NETLINK_CB(skb).dst_group = 1;
        retval = netlink_broadcast_filtered(uevent_sock, skb,
                                           0, 1, GFP_KERNEL,
                                           kobj_bcast_filter,
                                           kobj);

        /* ENOBUFS should be handled in userspace */
        if (retval == -ENOBUFS)
            retval = 0;
    } else
        retval = -ENOMEM;
}
mutex_unlock(&uevent_sock_mutex);
#endif

//使用 uevent_helper 机制实现 uevent
if (uevent_helper[0] && !kobj_usermode_filter(kobj)) {
    char *argv [3];

    argv [0] = uevent_helper;
    argv [1] = (char *) subsystem;
    argv [2] = NULL;
    retval = add_uevent_var(env, "HOME=/");
    if (retval)
        goto exit;
    retval = add_uevent_var(env,

```

```

        "PATH=/sbin:/bin:/usr/sbin:/usr/bin");
    if (retval)
        goto exit;

    retval = call_usermodehelper(argv[0], argv,
                                env->envp, UMH_WAIT_EXEC);
}
}

```

`kobject_uevent_env` 总体上可以分成三个功能部分，第一部分用到 `kset->uevent_ops`，调用其中的 `filter` 函数，以决定 `kset` 对象当前状态的改变是否要通知到用户层，如果 `uevent_ops->filter(kset, kobj)` 返回 0，将不再通知用户层。不同的 `kset` 对象拥有不同的 `uevent_ops` 对象，因此也意味着不同的 `kset` 都有自己独特的 `uevent_ops` 操作集，在后续使用到 `uevent_ops` 操作集的集体例子中将再来讨论此处的操作。总之，读者需要记住，一个 `kset` 对象状态的变化，将会首先调用隶属于该 `kset` 对象的 `uevent_ops` 操作集中的 `filter` 函数，以决定是否向用户层报告该事件。

如果 `filter` 函数通过了，换句话说，`kset` 中发生的事件需要通知用户层，那么将进入第二部分。第二部分主要是完成环境变量的设置，在一开始先通过 `env = kzalloc(sizeof(struct kobj_uevent_env), GFP_KERNEL)` 分配一个存储环境变量的空间对象 `env`，接下来把用户空间程序可能需要的环境变量通过 `add_uevent_var` 函数加入到 `env` 中。如同第一部分的 `filter` 函数一样，第二部分在处理完环境变量之后，会调用 `kset` 对象的 `uevent_ops` 操作集中的 `uevent` 函数，这是内核赋予 `kset` 通过该函数完成自己特定功能的最后一次机会。

第三部分是 `kobject_uevent_env` 函数的亮点，也是最有意思的地方，主要用来和用户空间进程进行交互（或者在内核空间启动执行一个用户空间的程序）。在 Linux 内核中，有两种方式完成这项任务，一个是代码中由 `CONFIG_NET` 宏包含的部分，这部分代码通过 `netlink` 的方式向用户空间广播当前 `kset` 对象中的 `uevent` 消息。另一种方式是在内核空间启动一个用户空间的进程，通过给该进程传递内核设定的环境变量的方式来通知用户空间 `kset` 对象中的 `uevent` 事件。虽然 `/sbin/hotplug` 方式已经逐渐被 `udev` 取代，但是因为 `/sbin/hotplug` 在内核中需要一个 `call_usermodehelper` 函数的支持，这是个比较有趣的函数，所以这里我们只讨论 `uevent_helper` 方式的实现。

`uevent_helper` 方法通过调用 `call_usermodehelper` 来达到从内核空间运行一个用户空间进程的目的，用户空间进程的二进制文件的路径由 `uevent_helper` 提供，该变量是一字符数组，在内核源码中的定义为：

```
<lib/kobject_uevent.c>
```

```
char uevent_helper[UEVENT_HELPER_PATH_LEN] = CONFIG_UEVENT_HELPER_PATH;
```

`CONFIG_UEVENT_HELPER_PATH` 是一内核编译阶段的配置宏，依赖于 `CONFIG_`

HOTPLUG，这意味着如果系统需要支持设备的热插拔等特性，则需要给出用户空间进程的文件路径信息。通常，CONFIG_UEVENT_HELPER_PATH 会指向/sbin/hotplug，后者用来处理系统中出现的热插拔事件，不过现在的 Linux 系统多半没有/sbin/hotplug 这个文件。

下面讨论 call_usermodehelper 函数的内核实现。对内核空间如何运行一个用户空间的进程感兴趣的读者，或者想深入理解内核如何支持设备的 hotplug 特性的设备驱动程序员，也许都不应该错过这里讨论的内容。call_usermodehelper 函数在 Linux 内核中的源码为：

```
<include/linux/kmod.h>
-----
static inline int
call_usermodehelper(char *path, char **argv, char **envp, enum umh_wait wait)
{
    return call_usermodehelper_fns(path, argv, envp, wait, NULL, NULL, NULL);
}
```

接下来会有很长的一段函数调用链，我们不妨略过这些不是很精彩的部分，直接看看核心的代码。这段函数调用链的核心部分在 call_usermodehelper_fns 函数中，它的代码是：

```
<include/linux/kmod.h>
-----
static inline int
call_usermodehelper_fns(char *path, char **argv, char **envp,
                        enum umh_wait wait,
                        int (*init)(struct subprocess_info *info),
                        void (*cleanup)(struct subprocess_info *), void *data)
{
    struct subprocess_info *info;
    gfp_t gfp_mask = (wait == UMH_NO_WAIT) ? GFP_ATOMIC : GFP_KERNEL;
    info = call_usermodehelper_setup(path, argv, envp, gfp_mask);
    if (info == NULL)
        return -ENOMEM;
    call_usermodehelper_setfns(info, init, cleanup, data);
    return call_usermodehelper_exec(info, wait);
}
```

call_usermodehelper_fns 函数的设计思想是采用工作队列的方式，在 call_usermodehelper_setup 函数内部会初始化一个工作队列的节点：

```
INIT_WORK(&sub_info->work, __call_usermodehelper);
```

其中 sub_info 是一 struct subprocess_info 类型的变量，工作队列节点作为它的一个内嵌对象，sub_info 其他成员用来存储运行用户态进程的一些相关信息，主要是相关的环境变量。__call_usermodehelper 是该工作节点上的延迟执行的函数。

将 call_usermodehelper_setup 中建立的工作节点提交到工作队列的行为发生在 call_usermodehelper_exec 函数中，其定义如下：

<kernel/kmod.c>

```
int call_usermodehelper_exec(struct subprocess_info *sub_info,
                           enum umh_wait wait)
{
    DECLARE_COMPLETION_ONSTACK(done);
    int retval = 0;

    helper_lock();
    if (sub_info->path[0] == '\0')
        goto out;

    if (!khelper_wq || usermodehelper_disabled) {
        retval = -EBUSY;
        goto out;
    }

    sub_info->complete = &done;
    sub_info->wait = wait;

    queue_work(khelper_wq, &sub_info->work);
    if (wait == UMH_NO_WAIT) /* task has freed sub_info */
        goto unlock;
    wait_for_completion(&done);
    retval = sub_info->retval;

out:
    call_usermodehelper_freeinfo(sub_info);
unlock:
    helper_unlock();
    return retval;
}
```

该函数的逻辑功能很直观,也许有几个细节注意一下会对理解整个 hotplug 的机制会有所帮助。首先是 `khelper_wq`,这是一个工作队列,其创建发生在 Linux 系统初始化阶段:

<kernel/kmod.c>

```
void __init usermodehelper_init(void)
{
    khelper_wq = create_singlethread_workqueue("khelper");
}
```

其次, `call_usermodehelper_exec` 函数通过引入一个 completion 变量 `done` 来实现和工作节点 `sub_info->work` 上的延迟函数 `__call_usermodehelper` 的同步:函数通过 `queue_work(khelper_wq, &sub_info->work)` 将工作节点提交到 `khelper_wq` 队列之后,将等待在 `wait_for_completion(&done)` 语句上。可以猜想当延迟函数 `__call_usermodehelper` 执行完毕,会

通过 complete 函数来唤醒睡眠的 call_usermodehelper_exec 函数。

最后，来看看__call_usermodehelper 要完成的工作：

<kernel/kmod.c>

```
static void __call_usermodehelper(struct work_struct *work)
{
    struct subprocess_info *sub_info =
        container_of(work, struct subprocess_info, work);
    enum umh_wait wait = sub_info->wait;
    pid_t pid;

    /* CLONE_VFORK: wait until the usermode helper has execve'd
     * successfully We need the data structures to stay around
     * until that is done. */
    if (wait == UMH_WAIT_PROC)
        pid = kernel_thread(wait_for_helper, sub_info,
                             CLONE_FS | CLONE_FILES | SIGCHLD);
    else
        pid = kernel_thread(__call_usermodehelper, sub_info,
                             CLONE_VFORK | SIGCHLD);

    switch (wait) {
    case UMH_NO_WAIT:
        call_usermodehelper_freeinfo(sub_info);
        break;

    case UMH_WAIT_PROC:
        if (pid > 0)
            break;
        /* FALLTHROUGH */
    case UMH_WAIT_EXEC:
        if (pid < 0)
            sub_info->retval = pid;
        complete(sub_info->complete);
    }
}
```

该函数会通过 kernel_thread 来生成一个新的进程，kernel_thread 的调用将会导致__call_usermodehelper 中出现两条执行路径，一是父进程，二是子进程，也就是新产生的进程。父进程在调用 kernel_thread 后会直接返回，而子进程则需要等到首次被调度的机会才会从 kernel_thread 返回，因此函数接下来出现了三个 case 来处理父子进程间的同步问题，不过这不是这里要重点关注的话题。

因为当初在调用 call_usermodehelper 函数时指定的 wait 参数是 UMH_WAIT_EXEC，所以

下面先按照这个路径进行讨论。`kernel_thread` 的具体实现是讲述内核之类的书籍应该关心的事情，这里我们只要知道它会产生一个新的进程，然后当该进程被调度执行时，`__call_usermodehelper` 函数会被调用，传递给它的参数是 `sub_info`，那里带有要执行的用户空间进程的路径及环境变量等信息。

注意这个函数最后的 `complete(sub_info->complete)`，它将会唤醒睡眠的 `call_usermodehelper_exec` 函数。

我们不再给出 `__call_usermodehelper` 的完整代码，在它的内部核心的调用是：

```
<kernel/kmod.c>
-----
static int __call_usermodehelper(void *data)
{
    struct subprocess_info *sub_info = data;
    ...
    kernel_execve(sub_info->path, sub_info->argv, sub_info->envp);
    ...
    sub_info->retval = retval;
    do_exit(0);
}
```

所以 `__call_usermodehelper` 执行完毕后，所在的进程将会因为 `do_exit` 的调用而从系统中消失掉。读者估计已经猜到 `kernel_execve` 函数用来在内核空间运行一个用户空间的进程，该进程的路径存放在 `sub_info->path` 中，进程运行时的环境变量等信息由 `sub_info->argv` 和 `sub_info->envp` 来提供。

`kernel_execve` 是个体系架构相关的函数，这里讨论其在 x86 架构上的实现，以满足那些好奇心强的读者：

```
<arch/x86/kernel/sys_i386_32.c>
-----
int kernel_execve(const char *filename, char *const argv[], char *const envp[])
{
    long __res;
    asm volatile ("push %%ebx ; movl %2,%%ebx ; int $0x80 ; pop %%ebx"
        : "=a" (__res)
        : "0" (__NR_execve), "ri" (filename), "c" (argv), "d" (envp) : "memory");
    return __res;
}
```

看到 `int $0x80`，知道这将导致一个系统调用。太多的时间我们看到的系统调用都是从用户空间发起，这里却是从内核空间发起，很有趣，不是吗？不过这样做不会有任何问题（如果读者对 x86 的 IDT 和 Linux 系统调用的实现机制很熟，那么一定不会对 `int $0x80` 这条指令的幕后行为感到陌生）。进入系统调用后，一个很重要的参数是系统调用号，Linux 用 `eax` 寄存器来保存系统调用号，在这段嵌入的汇编代码中，`__NR_execve` 的值将写入 `eax` 寄存

器，那就是系统调用号了。这个系统调用号对应的函数为 `sys_execve`，部分核心代码如下：

```
<arch/x86/kernel/process.c>
-----
long sys_execve(char __user *name, char __user * __user *argv,
                char __user * __user *envp, struct pt_regs *regs)
{
    long error;
    char *filename;

    filename = getname(name);
    error = PTR_ERR(filename);
    error = do_execve(filename, argv, envp, regs);
    putname(filename);
    return error;
}
```

`do_execve` 函数将执行 `filename` 所对应的进程文件，这个过程如同在 `shell` 里面执行一个可执行文件是一样的了（用户空间在运行一个二进制可执行文件时，也要通过系统调用 `sys_execve` 完成）。

至此，我们花了一定的篇幅讨论了内核空间如何运行一个用户空间的可执行文件，内核基于这个机制，在 `kset` 的状态发生变化时可以通知到用户进程，后者可以据此完成相应的工作，本章后续部分还会有具体使用这里讨论到的机制的例子。

前面的这个话题是从 `kset_register` 讨论延伸开去的，现在已经知道当向系统注册一个 `kset` 时可能会发生的事情，理解了这种底层数据结构上的操作，当我们接下来讨论设备驱动模型的高级部分，也即总线、设备与驱动时，将会加深这里的认识。

9.2.5 实例源码

经过以上对 `kobject` 与 `kset` 的详细讨论后，现在可以给出一个完整的源码来展示如何创建、初始化并向系统中添加一个 `kobject` 对象，以及如何通过 `sysfs` 文件系统接口在用户空间和内核空间进行沟通，另一个有趣的事情是它通过 `/sbin/hotplug` 机制来通知用户空间某一个 `kobject` 状态的变化。在这个例子中，我们将用自己编译的一个应用程序取代系统的 `/sbin/hotplug`²，该应用程序会打出一些环境变量，记录在 `/var/log/messages` 文件中。

首先是内核模块的源码：

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

² 不过估计现在大部分的 Linux 的发布中都没有提供现成的 `/sbin/hotplug` 工具，笔者的 Ubuntu 就没有。

```

#include <linux/sysfs.h>
#include <linux/slab.h>

static struct kobject *parent;
static struct kobject *child;
static struct kset *c_kset;

static unsigned long flag = 1;

static ssize_t att_show(struct kobject *kobj, struct attribute *attr, char *buf)
{
    size_t count = 0;
    count += sprintf(&buf[count], "%lu\n", flag);

    return count;
}

static ssize_t att_store(struct kobject *kobj, struct attribute *attr,
                        const char *buf, size_t count)
{
    flag = buf[0] - '0';
    //通过 kobject_uevent 来将内核对象 kobj 的状态变化通知用户程序
    switch(flag){
    case 0:
        kobject_uevent(kobj, KOBJ_ADD);
        break;
    case 1:
        kobject_uevent(kobj, KOBJ_REMOVE);
        break;
    case 2:
        kobject_uevent(kobj, KOBJ_CHANGE);
        break;
    case 3:
        kobject_uevent(kobj, KOBJ_MOVE);
        break;
    case 4:
        kobject_uevent(kobj, KOBJ_ONLINE);
        break;
    case 5:
        kobject_uevent(kobj, KOBJ_OFFLINE);
        break;
    }
    return count;
}

static struct attribute cld_att = {

```



```
.name = "cldatt",
.mode = S_IRUGO | S_IWUSR,
};

static const struct sysfs_ops att_ops = {
    .show = att_show,
    .store = att_store,
};

static struct kobj_type cld_ktype = {
    .sysfs_ops = &att_ops,
};

static int kobj_demo_init(void)
{
    int err;

    parent = kobject_create_and_add("pa_obj", NULL);

    child = kzalloc(sizeof(*child), GFP_KERNEL);
    if(!child)
        return PTR_ERR(child);

    //一个能够通知用户空间状态变化的 kobject 必须隶属于某一个 kset，也就是所谓的
    //subsystem，所以此处给内核对象 child 创建一个 kset 对象 c_kset
    c_kset = kset_create_and_add("c_kset", NULL, parent);
    if(!c_kset)
        return -1;
    child->kset = c_kset;

    err = kobject_init_and_add(child, &cld_ktype, parent, "cld_obj");
    if(err)
        return err;

    //为内核对象 child 创建一个属性文件
    err = sysfs_create_file(child, &cld_att);

    return err;
}

static void kobj_demo_exit(void)
{
    sysfs_remove_file(child, &cld_att);

    kset_unregister(c_kset);
    kobject_del(child);
}
```

```

    kobject_del(parent);
}

module_init(kobj_demo_init);
module_exit(kobj_demo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("dennis chen @ AMDLinuxFGL");
MODULE_DESCRIPTION("A simple kernel module to demo the kobject behavior");

```

以上代码在 2.6.39 内核版本的 Linux 系统上编译通过，对应的 Makefile 为：

```

obj-m := kobj_demo.o
KERNELDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
    rm -f *.o *.ko *.mod.*

```

将编译好的内核模块 `kobj_demo.ko` 通过 `insmod` 加入系统后，除了在 `/sys` 目录下生成 `parent` 与 `child` 内核对象所对应的入口点外，还会在 `/sys/pa_obj/cld_obj` 目录下生成 `child` 内核对象的一个属性文件 `cldatt`：

```

root@AMDLinuxFGL:/sys/pa_obj/cld_obj# ls -l
total 0
-rw-r--r-- 1 root root 4096 Jun 18 21:25 cldatt

```

可以很方便地通过 `sysfs` 文件系统接口来改变内核模块中的变量 `flag`：

```

root@AMDLinuxFGL:/sys/pa_obj/cld_obj# cat cldatt
1
root@AMDLinuxFGL:/sys/pa_obj/cld_obj# echo '8' > cldatt
root@AMDLinuxFGL:/sys/pa_obj/cld_obj# cat cldatt
8

```

接下来看看内核对象 `child` 的 `uevent` 特性如何工作，用来编译应用程序 `/sbin/hotplug` 的源码为：

```

#include <stdio.h>
#include <syslog.h>

extern char **environ;

int main(int argc, char *argv[])
{

```

```

char **var;
syslog(LOG_INFO|LOG_LOCAL0, "-----\n");
syslog(LOG_INFO|LOG_LOCAL0, "argv[1]=%s\n", argv[1]);
for(var = environ; *var != NULL; ++ var)
    syslog(LOG_INFO|LOG_LOCAL0, "env=%s\n", *var);
syslog(LOG_INFO|LOG_LOCAL0, "-----\n");

return 0;
}

```

首先清空/var/log/messages 文件，这样便于看清/sbin/hotplug 向里面记录的内容：

```
root@AMDLinuxFGL:/home/dennis/Linux/book/chap09/kobj# cat /dev/null > /var/log/messages
```

其次需要把/sbin/hotplug 机制打开，这由/proc/sys/kernel 下的 hotplug 文件来决定：

```
root@AMDLinuxFGL:/proc/sys/kernel# echo "/sbin/hotplug" > hotplug
```

然后用 insmod 加入模块 kobj_demo.ko：

```
root@AMDLinuxFGL:/home/dennis/Linux/book/chap09/kobj# insmod kobj_demo.ko
```

最后来看/var/log/messages 中的内容：

```

root@AMDLinuxFGL:/home/dennis/Linux/book/chap09/kobj# cat /var/log/messages
Jun 19 03:57:11 AMDLinuxFGL hotplug: -----
Jun 19 03:57:11 AMDLinuxFGL hotplug: argv[1]=module
Jun 19 03:57:11 AMDLinuxFGL hotplug: env=ACTION=add
Jun 19 03:57:11 AMDLinuxFGL hotplug: env=DEVPATH=/module/kobj_demo
Jun 19 03:57:11 AMDLinuxFGL hotplug: env=SUBSYSTEM=module
Jun 19 03:57:11 AMDLinuxFGL hotplug: env=SEQNUM=1463
Jun 19 03:57:11 AMDLinuxFGL hotplug: env=HOME=/
Jun 19 03:57:11 AMDLinuxFGL hotplug: env=PATH=/sbin:/bin:/usr/sbin:/usr/bin
Jun 19 03:57:11 AMDLinuxFGL hotplug: -----

```

上面的信息显然是模块加载器加载 kobj_demo 模块时发出的通知。接下来通过写内核对象 child 的属性文件 cldatt 来模拟其状态变化：

```

root@AMDLinuxFGL:/sys/pa_obj/cld_obj# echo '0' > cldatt
root@AMDLinuxFGL:/sys/pa_obj/cld_obj# echo '1' > cldatt
root@AMDLinuxFGL:/sys/pa_obj/cld_obj# echo '2' > cldatt
root@AMDLinuxFGL:/sys/pa_obj/cld_obj# echo '3' > cldatt
root@AMDLinuxFGL:/sys/pa_obj/cld_obj# echo '4' > cldatt

```

```
root@AMDLinuxFGL:/sys/pa_obj/cld_obj# echo '5' > cldatt
```

再看看/var/log/messages 中的信息有何变化:

```
root@AMDLinuxFGL:/home/dennis/Linux/book/chap09/kobj# cat /var/log/messages
Jun 19 03:57:11 AMDLinuxFGL hotplug: -----
Jun 19 03:57:11 AMDLinuxFGL hotplug: argv[1]=module
Jun 19 03:57:11 AMDLinuxFGL hotplug: env=ACTION=add
Jun 19 03:57:11 AMDLinuxFGL hotplug: env=DEVPATH=/module/kobj_demo
Jun 19 03:57:11 AMDLinuxFGL hotplug: env=SUBSYSTEM=module
Jun 19 03:57:11 AMDLinuxFGL hotplug: env=SEQNUM=1463
Jun 19 03:57:11 AMDLinuxFGL hotplug: env=HOME=/
Jun 19 03:57:11 AMDLinuxFGL hotplug: env=PATH=/sbin:/bin:/usr/sbin:/usr/bin
Jun 19 03:57:11 AMDLinuxFGL hotplug: -----
Jun 19 04:02:38 AMDLinuxFGL hotplug: -----
Jun 19 04:02:38 AMDLinuxFGL hotplug: argv[1]=c_kset
Jun 19 04:02:38 AMDLinuxFGL hotplug: env=ACTION=add
Jun 19 04:02:38 AMDLinuxFGL hotplug: env=DEVPATH=/pa_obj/cld_obj
Jun 19 04:02:38 AMDLinuxFGL hotplug: env=SUBSYSTEM=c_kset
Jun 19 04:02:38 AMDLinuxFGL hotplug: env=SEQNUM=1464
Jun 19 04:02:38 AMDLinuxFGL hotplug: env=HOME=/
Jun 19 04:02:38 AMDLinuxFGL hotplug: env=PATH=/sbin:/bin:/usr/sbin:/usr/bin
Jun 19 04:02:38 AMDLinuxFGL hotplug: -----
Jun 19 04:02:43 AMDLinuxFGL hotplug: -----
Jun 19 04:02:43 AMDLinuxFGL hotplug: argv[1]=c_kset
Jun 19 04:02:43 AMDLinuxFGL hotplug: env=ACTION=remove
Jun 19 04:02:43 AMDLinuxFGL hotplug: env=DEVPATH=/pa_obj/cld_obj
Jun 19 04:02:43 AMDLinuxFGL hotplug: env=SUBSYSTEM=c_kset
Jun 19 04:02:43 AMDLinuxFGL hotplug: env=SEQNUM=1465
Jun 19 04:02:43 AMDLinuxFGL hotplug: env=HOME=/
Jun 19 04:02:43 AMDLinuxFGL hotplug: env=PATH=/sbin:/bin:/usr/sbin:/usr/bin
Jun 19 04:02:43 AMDLinuxFGL hotplug: -----
Jun 19 04:02:47 AMDLinuxFGL hotplug: -----
Jun 19 04:02:47 AMDLinuxFGL hotplug: argv[1]=c_kset
Jun 19 04:02:47 AMDLinuxFGL hotplug: env=ACTION=change
Jun 19 04:02:47 AMDLinuxFGL hotplug: env=DEVPATH=/pa_obj/cld_obj
Jun 19 04:02:47 AMDLinuxFGL hotplug: env=SUBSYSTEM=c_kset
```

```
Jun 19 04:02:47 AMDLinuxFGL hotplug: env=SEQNUM=1466
Jun 19 04:02:47 AMDLinuxFGL hotplug: env=HOME=/
Jun 19 04:02:47 AMDLinuxFGL hotplug: env=PATH=/sbin:/bin:/usr/sbin:/usr/bin
Jun 19 04:02:47 AMDLinuxFGL hotplug: -----
Jun 19 04:02:50 AMDLinuxFGL hotplug: -----
Jun 19 04:02:50 AMDLinuxFGL hotplug: argv[1]=c_kset
Jun 19 04:02:50 AMDLinuxFGL hotplug: env=ACTION=move
Jun 19 04:02:50 AMDLinuxFGL hotplug: env=DEVPATH=/pa_obj/cld_obj
Jun 19 04:02:50 AMDLinuxFGL hotplug: env=SUBSYSTEM=c_kset
Jun 19 04:02:50 AMDLinuxFGL hotplug: env=SEQNUM=1467
Jun 19 04:02:50 AMDLinuxFGL hotplug: env=HOME=/
Jun 19 04:02:50 AMDLinuxFGL hotplug: env=PATH=/sbin:/bin:/usr/sbin:/usr/bin
Jun 19 04:02:50 AMDLinuxFGL hotplug: -----
Jun 19 04:02:54 AMDLinuxFGL hotplug: -----
Jun 19 04:02:54 AMDLinuxFGL hotplug: argv[1]=c_kset
Jun 19 04:02:54 AMDLinuxFGL hotplug: env=ACTION=online
Jun 19 04:02:54 AMDLinuxFGL hotplug: env=DEVPATH=/pa_obj/cld_obj
Jun 19 04:02:54 AMDLinuxFGL hotplug: env=SUBSYSTEM=c_kset
Jun 19 04:02:54 AMDLinuxFGL hotplug: env=SEQNUM=1468
Jun 19 04:02:54 AMDLinuxFGL hotplug: env=HOME=/
Jun 19 04:02:54 AMDLinuxFGL hotplug: env=PATH=/sbin:/bin:/usr/sbin:/usr/bin
Jun 19 04:02:54 AMDLinuxFGL hotplug: -----
Jun 19 04:02:58 AMDLinuxFGL hotplug: -----
Jun 19 04:02:58 AMDLinuxFGL hotplug: argv[1]=c_kset
Jun 19 04:02:58 AMDLinuxFGL hotplug: env=ACTION=offline
Jun 19 04:02:58 AMDLinuxFGL hotplug: env=DEVPATH=/pa_obj/cld_obj
Jun 19 04:02:58 AMDLinuxFGL hotplug: env=SUBSYSTEM=c_kset
Jun 19 04:02:58 AMDLinuxFGL hotplug: env=SEQNUM=1469
Jun 19 04:02:58 AMDLinuxFGL hotplug: env=HOME=/
Jun 19 04:02:58 AMDLinuxFGL hotplug: env=PATH=/sbin:/bin:/usr/sbin:/usr/bin
Jun 19 04:02:58 AMDLinuxFGL hotplug: -----
```

从上面的输出可以看到，模拟的 5 个状态变化一个不落地被/sbin/hotplug 捕捉到了。

本例展示了在 Linux 设备模型最底层 kobject 一级的 sysfs 文件系统的作用以及对设备热插拔机制的支持，虽然内核已经通过更高层次的 bus、device 与 driver 这些对象向程序员屏蔽了 sysfs 和 uevent 底层的这些操作，但是我们应该知道 bus、device 与 driver 这一层面的诸

如属性文件相关操作以及对设备热插拔特性的支持，其幕后的原理其实就发源于此。

这段文字临结束时，我又突发奇想，不妨插个 U 盾到机器里来看看会发生什么，于是 /var/log/messages 中就有了下面的输出：

```
Jun 19 04:11:45 AMDLinuxFGL kernel: [ 5803.856011] usb 3-1: new full speed USB device number 3
using uhci_hcd
Jun 19 04:11:46 AMDLinuxFGL hotplug: -----
Jun 19 04:11:46 AMDLinuxFGL hotplug: argv[1]=usb
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=ACTION=add
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=DEVPATH=/devices/pci0000:00/0000:00:1a.0/usb3/3-1
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=SUBSYSTEM=usb
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=MAJOR=189
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=MINOR=258
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=DEVNAME=bus/usb/003/003
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=DEVTYPE=usb_device
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=PRODUCT=8e6/1813/100
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=TYPE=0/0/0
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=BUSNUM=003
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=DEVNUM=003
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=SEQNUM=1470
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=HOME=/
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=PATH=/sbin:/bin:/usr/sbin:/usr/bin
Jun 19 04:11:46 AMDLinuxFGL hotplug: -----
Jun 19 04:11:46 AMDLinuxFGL hotplug: -----
Jun 19 04:11:46 AMDLinuxFGL hotplug: argv[1]=usb
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=ACTION=add
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=DEVPATH=/devices/pci0000:00/0000:00:1a.0/usb3/3-1/
3-1:1.0
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=SUBSYSTEM=usb
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=DEVTYPE=usb_interface
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=PRODUCT=8e6/1813/100
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=TYPE=0/0/0
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=INTERFACE=3/0/0
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=MODALIAS=usb:v08E6p1813d0100dc00dsc00dp00i
c03isc00ip00
```

```

Jun 19 04:11:46 AMDLinuxFGL hotplug: env=SEQNUM=1471
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=HOME=/
Jun 19 04:11:46 AMDLinuxFGL hotplug: env=PATH=/sbin:/bin:/usr/sbin:/usr/bin
Jun 19 04:11:46 AMDLinuxFGL hotplug: -----

```

虽然我们在这个例子中使用了自己编译的一个可执行文件，但是/sbin/hotplug 完全可以是一个脚本程序，实际上/sbin/hotplug 的用法就是通过脚本来加载卸载模块的。如果系统中有 udevd 守护进程，那么它应该一直在监听 kobject_uevent 通过 netlink 广播出去的 uevent 数据包。无论如何，内核空间通过 kobject_uevent 这个函数实现了将内核中发生的一些事件通知到了用户空间。

9.3 总线、设备与驱动

前面已经介绍了 Linux 设备驱动模型的底层数据结构及相关操作，现在开始讨论该模型的高层部分，也是 Linux 下设备驱动程序员与之打交道最多的部分。高层部分的核心分为三个组件，正如本节标题揭示的那样，分别是总线（bus）、设备（device）和驱动（driver），它们构成了 Linux 设备驱动模型这一宏大建筑的外在表现。接下来将依次讨论每个组件，看看 Linux 引入的这个新的设备模型到底给系统，给设备驱动程序员带来了哪些好处和不足。

9.3.1 总线及其注册

总线可以看成 Linux 设备驱动模型这座建筑的核心框架，系统中其他的设备与驱动将紧密团结在以总线为核心的设备模型的周围，完成各自的使命。不过设备驱动程序员在系统中创建一个新的总线的机会并不多。驱动模型中的总线，既可以是实际物理总线（比如 PCI 总线和 I2C 总线等）的抽象，也可以是出于驱动模型架构需要而产生的虚拟“平台”总线，因为一个符合 Linux 驱动模型的设备与驱动必须挂靠在一根总线上，无论它是实际存在的总线还是系统虚拟出的总线。

内核为总线对象定义的数据结构是 bus_type，其完整定义如下：

```

<include/linux/device.h>
-----
struct bus_type {
    const char      *name;
    struct bus_attribute *bus_attrs;
    struct device_attribute *dev_attrs;
    struct driver_attribute *drv_attrs;

    int (*match)(struct device *dev, struct device_driver *drv);
    int (*uevent)(struct device *dev, struct kobj_uevent_env *env);

```

```

int (*probe)(struct device *dev);
int (*remove)(struct device *dev);
void (*shutdown)(struct device *dev);

int (*suspend)(struct device *dev, pm_message_t state);
int (*resume)(struct device *dev);

const struct dev_pm_ops *pm;
struct subsys_private *p;
};

```

下面先简单介绍一下其中的一些主要成员，这里的介绍虽然有点抽象，但是当我们在本节后续看到这些成员具体的使用方式时，相信读者应该会加深此处的理解：

```
const char    *name
```

总线的名称。如同每个人都有名字一样，总线也不例外。

```
struct bus_attribute *bus_attrs
```

总线的属性，包括操作这些属性的一组函数，都包含在 `struct bus_attribute` 结构体内。

```
struct device_attribute *dev_attrs
```

挂载到该总线上的设备的属性，功能逻辑与总线属性一样。

```
struct driver_attribute *drv_attrs
```

挂载到该总线上的驱动的属性，功能逻辑与总线属性一样。

```
int (*match)(struct device *dev, struct device_driver *drv)
```

总线用来对试图挂载到其上的设备与驱动执行的匹配操作。除了这个函数，`struct bus_type` 结构中还定义了一些操作函数，这里不再一一讲述。读者很快就会在本章后续的讨论中看到内核对它们的使用。

```
const struct dev_pm_ops *pm
```

总线上一组跟电源管理相关的操作集，用来对总线上的设备进行电源管理。

```
struct subsys_private *p
```

一个用来管理其上设备与驱动的数据结构，在内核中的定义为：

```

<drivers/base/base.h>
-----
struct subsys_private {
    struct kset subsys;
    struct kset *devices_kset;
};

```



```

    struct kset *drivers_kset;
    struct klist klist_devices;
    struct klist klist_drivers;
    struct blocking_notifier_head bus_notifier;
    unsigned int drivers_autoprobe:1;
    struct bus_type *bus;

    struct list_head class_interfaces;
    struct kset glue_dirs;
    struct mutex class_mutex;
    struct class *class;
};

```

其中, `struct kset subsys` 用来表示该 `bus` 所在的子系统, 在内核中所有通过 `bus_register` 注册进系统的 `bus` 所在的 `kset` 都将指向 `bus_kset`, 换句话说 `bus_kset` 是系统中所有 `bus` 内核对象的容器, 而新注册的 `bus` 本身也是一个 `kset` 型对象。`struct kset *drivers_kset` 表示该 `bus` 上所有驱动的一个集合, `struct kset *devices_kset` 则表示该 `bus` 上所有设备的一个集合。`struct klist klist_devices` 和 `struct klist klist_drivers` 则分别表示该 `bus` 上所有设备与驱动的链表。`drivers_autoprobe` 用来表示当向系统 (确切地说是系统中某一总线) 中注册某一设备或者驱动的时候, 是否进行设备与驱动的绑定操作。`struct bus_type *bus` 指向与 `struct bus_type_private` 对象相关联的 `bus`。

为了便于接下来对总线、设备与驱动的讨论, 我们从总线的角度, 给出图 9-3 所示三者之间的互联层次关系:

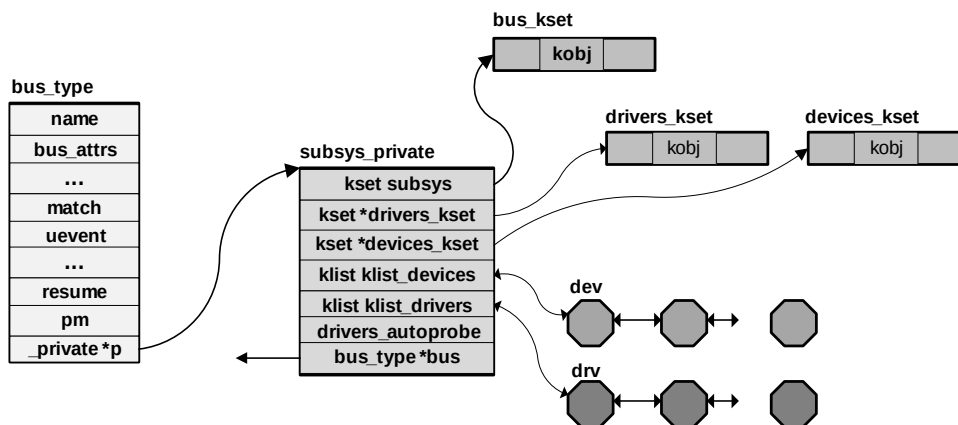


图 9-3 bus、dev 与 drv 的层次关系

图 9-3 展示了一个总线对象所衍生出来的拓扑关系, 这种拓扑关系主要通过 `bus_type` 中的 `struct subsys_private *p` 成员来体现。在这个成员中, `struct kset subsys` 标识了系统中当前总线对象与 `bus_kset` 间的隶属关系, 而 `struct kset *drivers_kset` 和 `struct kset *devices_kset` 则

是在向系统注册当前新总线时动态生成的容纳该总线上所有驱动与设备的 `kset`，与此对应，两个 `klist` 成员则以链表的形式将该总线上所有的驱动与设备链接到了一起。

Linux 内核中针对总线的一些主要操作有：

○ buses_init

`buses_init` 函数揭示了总线在系统中的起源，在系统的初始化阶段，就通过 `buses_init` 函数为系统中后续的 `bus` 操作奠定了基础，该函数的实现为：

<drivers/base/bus.c>

```
int __init buses_init(void)
{
    bus_kset = kset_create_and_add("bus", &bus_uevent_ops, NULL);
    if (!bus_kset)
        return -ENOMEM;
    return 0;
}
```

前面已经介绍过 `kset_create_and_add` 函数，此处理解 `buses_init` 函数应该没有什么问题，它将创建一个名称为“bus”的 `kset` 并将其加入到 `sysfs` 文件系统树中，注意这里的 `bus_uevent_ops` 定义了当“bus”这个 `kset` 中有状态变化时，用来通知用户空间 `uevent` 消息的操作集。前面在讨论 `kset` 时知道，当某个 `kset` 中有状态的变化时，如果需要向用户空间发送 `event` 消息，将由该 `kset` 的最顶层 `kset` 来执行，因为 `bus_kset` 是系统中所有 `bus subsystem` 最顶层的 `kset`，所以 `bus` 中的 `uevent` 调用最终会汇集到这里的 `bus_uevent_ops` 中。这个操作集只定义了一个 `filter` 操作，意味着当“bus”`kset` 中发生状态变化时，会通过 `bus_uevent_ops` 中的 `filter` 函数先行处理，以决定是否通知用户态空间。`bus_uevent_ops` 定义如下：

<drivers/base/bus.c>

```
static const struct kset_uevent_ops bus_uevent_ops = {
    .filter = bus_uevent_filter,
};
```

<drivers/base/bus.c>

```
static int bus_uevent_filter(struct kset *kset, struct kobject *kobj)
{
    struct kobj_type *ktype = get_ktype(kobj);
    if (ktype == &bus_ktype)
        return 1;
    return 0;
}
```

如果要求发送 `uevent` 消息的 `kobj` 对象类型不是总线类型 (`bus_type`)，那么函数将返回 0，意味着 `uevent` 消息将不会发送到用户空间，所以 `bus_uevent_ops` 使得 `bus_kset` 只用来发送

bus 类型的内核对象产生的 uevent 消息。

buses_init 将在 sysfs 文件系统的根目录下建立一个“bus”目录，在用户空间看来，就是 /sys/bus。buses_init 函数创建的“bus”总线将是系统中所有后续注册总线的祖先。

○ bus_register

该函数用来向系统中注册一个 bus，其部分核心代码如下：

<drivers/base/bus.c>

```
int bus_register(struct bus_type *bus)
{
    int retval;
    struct subsys_private *priv;

    priv = kzalloc(sizeof(struct subsys_private), GFP_KERNEL);
    if (!priv)
        return -ENOMEM;

    priv->bus = bus;
    bus->p = priv;

    BLOCKING_INIT_NOTIFIER_HEAD(&priv->bus_notifier);

    retval = kobject_set_name(&priv->subsys.kobj, "%s", bus->name);
    if (retval)
        goto out;

    priv->subsys.kobj.kset = bus_kset;
    priv->subsys.kobj.ktype = &bus_ktype;
    priv->drivers_autoprobe = 1;

    //在/sys/bus 目录下为当前注册的 bus 生成一个新的目录
    retval = kset_register(&priv->subsys);
    if (retval)
        goto out;

    //生成 bus 的属性文件
    retval = bus_create_file(bus, &bus_attr_uevent);
    if (retval)
        goto bus_uevent_fail;

    //为当前 bus 产生容纳设备的 kset 容器
    priv->devices_kset = kset_create_and_add("devices", NULL,
                                            &priv->subsys.kobj);
    if (!priv->devices_kset) {
```

```

        retval = -ENOMEM;
        goto bus_devices_fail;
    }

    //为当前 bus 产生容纳驱动的 kset 容器
    priv->drivers_kset = kset_create_and_add("drivers", NULL,
                                           &priv->subsys.kobj);

    if (!priv->drivers_kset) {
        retval = -ENOMEM;
        goto bus_drivers_fail;
    }

    //初始化 bus 上的设备与驱动的链表
    klist_init(&priv->klist_devices, klist_devices_get, klist_devices_put);
    klist_init(&priv->klist_drivers, NULL, NULL);

    //为当前 bus 增加 probe 相关的属性文件
    retval = add_probe_files(bus);
    if (retval)
        goto bus_probe_files_fail;

    retval = bus_add_attrs(bus);
    if (retval)
        goto bus_attrs_fail;

    pr_debug("bus: '%s': registered\n", bus->name);
    return 0;

bus_attrs_fail:
    remove_probe_files(bus);
bus_probe_files_fail:
    kset_unregister(bus->p->drivers_kset);
bus_drivers_fail:
    kset_unregister(bus->p->devices_kset);
bus_devices_fail:
    bus_remove_file(bus, &bus_attr_uevent);
bus_uevent_fail:
    kset_unregister(&bus->p->subsys);
out:
    kfree(bus->p);
    bus->p = NULL;
    return retval;
}

```

函数首先分配一个 struct subsys_private 类型的对象, 然后通过 kobject_set_name 为 bus 所在的内核对象设定名称, 该名称将显示在 sysfs 文件系统树中。前面提到, bus 作为一个 kset

类型的内核对象，其对象属性等特性体现在 struct subsys_private 对象的 subsys 成员中，这是个 kset 型变量，所以注册一个 bus，将同时赋予该 bus 特定的属性特质，这由下面两条语句完成：

```
priv->subsys.kobj.kset = bus_kset;
priv->subsys.kobj.ktype = &bus_ktype;
```

第一条语句指明了当前注册的 bus 对象所属的上层 kset 对象，就是 buses_init 中创建的名为“bus”的 kset。第二条语句指明了当前注册的 bus 的属性类型 bus_ktype，后者定义了该特定 bus 上的一些与总线属性文件相关的操作：

<drivers/base/bus.c>

```
static struct kobj_type bus_ktype = {
    .sysfs_ops = &bus_sysfs_ops,
};
```

bus_sysfs_ops 中的操作主要是用来显示（show）或者设置（store）当前注册的 bus 在 sysfs 文件系统上的属性。

函数中的 kset_register(&priv->subsys)用来将当前操作的 bus 所对应的 kset 加入到 sysfs 文件系统树中，因为 priv->subsys.kobj.parent = NULL 并且 priv->subsys.kobj.kset = bus_kset，所以当前注册的 bus 对应的 kset 的目录将建立在/sys/bus 当中。

bus_create_file(bus, &bus_attr_uevent)将为该 bus 创建一属性文件。关于 bus 的属性问题，稍后将另开一节予以讨论。

接下来可以看到有两个 kset_create_and_add 调用：

```
priv->devices_kset = kset_create_and_add("devices", NULL, &priv->subsys.kobj);
priv->drivers_kset = kset_create_and_add("drivers", NULL, &priv->subsys.kobj);
```

前面讨论过 kset_create_and_add 函数，它将生成一个 kset 对象并将其加入到 sysfs 文件系统中。注意这里在调用 kset_create_and_add 函数时，parent 参数均为&priv->subsys.kobj，这意味着将在当前正在向系统注册的新 bus 目录下产生两个 kset 目录，分别对应新 bus 的 devices 和 drivers，假设新 bus 的名称是“new_bus”，那么反应到/sys 文件目录中就是 /sys/bus/new_bus/devices 和 /sys/bus/new_bus/drivers。

图 9-4 反应了通过 bus_register 向系统注册一个新的 bus1 时所产生的组件及层次关系：

图中虚线部分是将 bus1 通过 bus_register 注册进系统时所产生的层次关系结构。首先代表 bus1 的一个 kset 对象将被产生出来并且加入到 sysfs 文件系统中，该 kset 的 parent 内核对象为 buses_init 函数中所产生的 bus_kset。其次 bus_register 通过调用 kset_create_and_add 函数产生连接到 bus1 上的 devices_kset 和 drivers_kset 两个集合，对应到 sysfs 文件系统，将会在 bus1 的目录下产生两个新的目录“devices”和“drivers”。最后为了让用户空间看

到或者重新配置 bus1 上的某些属性值, bus_register 调用 bus_create_file 函数为 bus1 产生一些属性文件, 这些属性文件也将位于 /sys/bus/bus1 目录之下, 属性文件实际上向用户空间提供了一种接口, 使得用户程序可以通过文件的方式来显示某一内核对象的属性或者重新配置这一属性。

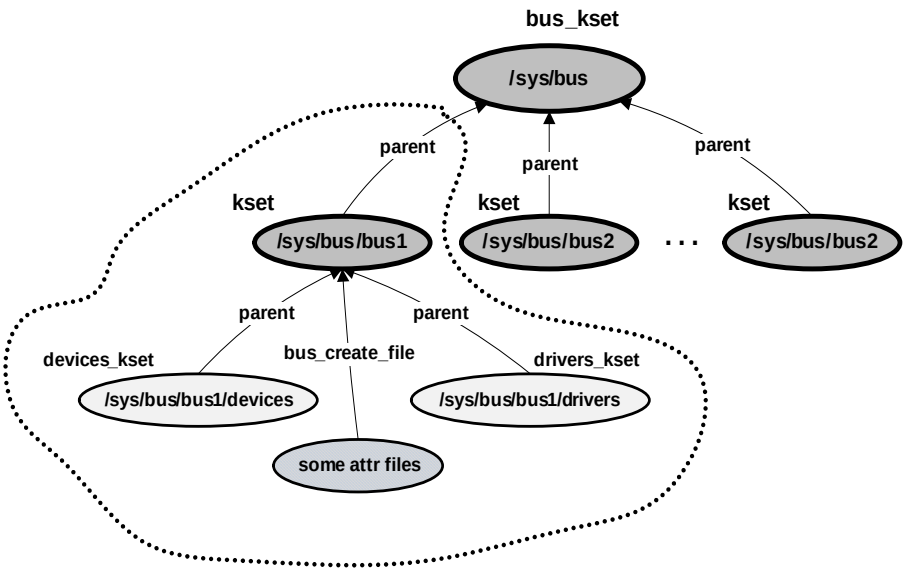


图 9-4 通过 bus_register 向系统注册一根总线 bus1

9.3.2 总线的属性

总线属性代表着该总线特有的信息与配置, 如果通过 sysfs 文件系统为总线生成属性文件, 那么用户空间的程序可以通过该文件接口的方式很容易地显示或者更改该总线的属性。根据实际需要, 可以为总线创建不止一个属性文件, 每个文件代表该总线的的一个或一组属性信息。总线属性在内核中的数据结构为:

```
<include/linux/device.h>
-----
struct bus_attribute {
    struct attribute    attr;
    ssize_t (*show)(struct bus_type *bus, char *buf);
    ssize_t (*store)(struct bus_type *bus, const char *buf, size_t count);
};
```

成员变量 attr 表示总线的属性信息, 其类型为 struct attribute:

```
<include/linux/sysfs.h>
-----
struct attribute {
    const char    *name;
    mode_t        mode;
```

```
};
```

struct bus_attribute 的另外两个成员 show 与 store 分别用来显示和更改总线的属性。内核定义有一个宏 BUS_ATTR，用来方便为总线定义一个属性对象：

```
<include/linux/device.h>
-----
#define BUS_ATTR(_name, _mode, _show, _store) \
struct bus_attribute bus_attr_##_name = __ATTR(_name, _mode, _show, _store)

<include/linux/sysfs.h>
-----
#define __ATTR(_name, _mode, _show, _store) { \
    .attr = { .name = __stringify(_name), .mode = _mode }, \
    .show = _show, \
    .store = _store, \
}
```

BUS_ATTR 宏将定义一个以 “bus_attr_” 开头的总线属性对象，而生成总线属性文件则需要使用 bus_create_file 函数：

```
<drivers/base/bus.c>
-----
int bus_create_file(struct bus_type *bus, struct bus_attribute *attr)
{
    int error;
    if (bus_get(bus)) {
        error = sysfs_create_file(&bus->p->subsys.kobj, &attr->attr);
        bus_put(bus);
    } else
        error = -EINVAL;
    return error;
}
```

sysfs_create_file 用来在 sysfs 文件树中创建一个属性文件，这里不会讨论 sysfs 实现这个函数的细节。我们关注的是，用户层的应用程序如何利用总线属性文件的接口来显示和更改总线属性。这里以 bus_register 函数中的 add_probe_files 调用为例，后者会用 BUS_ATTR 宏定义一个总线属性，然后为之生成一个属性文件（读者可以在 /sys/bus 目录中的任一总线目录下发现 drivers_autoprobe 文件）。

通过 BUS_ATTR 宏，add_probe_files 为此定义的总线属性为：

```
<drivers/base/bus.c>
-----
static BUS_ATTR(drivers_autoprobe, S_IWUSR | S_IRUGO,
    show_drivers_autoprobe, store_drivers_autoprobe);
```

上面的宏将产生一个总线属性对象 bus_attr_drivers_autoprobe，该文件的模式为 S_IWUSR | S_IRUGO，表明对 root 用户而言具有读与写的权限。

显示该总线属性的函数为 `show_drivers_autoprobe`:

```
<drivers/base/bus.c>
-----
static ssize_t show_drivers_autoprobe(struct bus_type *bus, char *buf)
{
    return sprintf(buf, "%d\n", bus->p->drivers_autoprobe);
}

static ssize_t store_drivers_autoprobe(struct bus_type *bus,
                                       const char *buf, size_t count)
{
    if (buf[0] == '0')
        bus->p->drivers_autoprobe = 0;
    else
        bus->p->drivers_autoprobe = 1;
    return count;
}
```

通过上面这个函数实现，可以发现该属性文件向用户空间提供了一个显示和更改 `bus->p->drivers_autoprobe` 成员的接口。下面看一个用户空间如何在 `shell` 里面显示和更改一个 `bus` 的 `drivers_autoprobe` 成员的例子，比如对于 `/sys/bus/pci` 而言，在 `Linux shell` 里面：

```
dennis@ubuntu:/sys/bus/pci$ cat drivers_autoprobe
1
```

输出的 1 表明当前 `PCI` 总线的 `drivers_autoprobe` 成员值为 1。要更改这个值，可以使用如下命令³：

```
root@AMDLinuxFGL:/home/dennis/book # echo 0 > drivers_autoprobe
```

命令成功执行后，再用“`cat drivers_autoprobe`”命令，就会发现输出已经是 0 了。

如果读者对用户空间进程如何调用到属性文件的过程（`show` 和 `store`）感兴趣，应该回过头来看看“内核对象 `kobject` 的属性”一节。在那里曾提到在构造 `sysfs` 文件系统的超级块时，内核会调用到 `sysfs_init_inode` 函数，这个函数为 `sysfs` 文件系统上的 `inode` 初始化了相关的操作对象 `i_op` 和 `i_fop`，这样对于在 `sysfs` 文件系统中生成总线属性文件的 `bus_create_file` 而言，它生成的属性文件被用户空间的 `shell` 命令 `cat` 操作时，将利用到 `inode` 上 `i_fop` 操作集：

```
<fs/sysfs/inode.c>
-----
static void sysfs_init_inode(struct sysfs_dirent *sd, struct inode *inode)
{
    ...
    case SYSFS_KOBJ_ATTR:
```

³ 如果出现权限不够的错误信息，通过 `sudo chmod` 命令将该文件的 `mode` 修改成别的用户也具有写权限即可。


```

        inode->i_size = PAGE_SIZE;
        inode->i_fop = &sysfs_file_operations;
        break;
    ...
}

```

sysfs_file_operations 的定义为:

```

<fs/sysfs/file.c>
-----
const struct file_operations sysfs_file_operations = {
    .read      = sysfs_read_file,
    .write     = sysfs_write_file,
    .llseek    = generic_file_llseek,
    .open      = sysfs_open_file,
    .release   = sysfs_release,
    .poll      = sysfs_poll,
};

```

所以 shell 环境下的 cat 命令最终会调用到 sysfs_read_file 函数, 在后者调用的 fill_read_buffer 中, 将调用到总线属性对象中的 show 函数:

```

<fs/sysfs/file.c>
-----
static int fill_read_buffer(struct dentry * dentry, struct sysfs_buffer * buffer)
{
    ...
    count = ops->show(kobj, attr_sd->s_attr.attr, buffer->page);
    ...
}

```

这里只是给出了用户空间与内核空间通过总线属性文件交互的通道框架, 鉴于具体的实现细节已经偏离本章的主题, 故不再具体讨论, 感兴趣的读者可以自行研究 sysfs 文件系统的内部实现。

9.3.3 设备与驱动的绑定

在继续讨论设备与驱动的话题之前, 先来看看 Linux 设备驱动模型中的一个重要概念: 设备与驱动的绑定 (binding)。这里的绑定, 简单地说就是将一个设备与能控制它的驱动程序结合到一起的行为。两个内核对象间的结合自然是靠各自背后的数据结构中的某些成员来完成。

总线在设备与驱动绑定的过程中发挥着核心作用: 总线相关的代码屏蔽了大量底层琐碎的技术细节, 为驱动程序员们提供了一组使用友好的外在接口, 从而简化了驱动程序的开发工作。在总线上发生的两类事件将导致设备与驱动绑定行为的发生: 一是通过 device_register 函数向某一 bus 上注册一设备, 这种情况下内核除了将该设备加入到 bus 上

的设备链表的尾端，同时会试图将此设备与总线上的所有驱动对象进行绑定操作（当然，操作归操作，能否成功则是另外一回事）；二是通过 `driver_register` 将某一驱动注册到其所属的 `bus` 上，内核此时除了将该驱动对象加入到 `bus` 的所有驱动对象构成的链表的尾部，也会试图将该驱动与其上的所有设备进行绑定操作。

下面从代码的角度看看设备与驱动的绑定到底意味着什么。当调用 `device_register` 向某一 `bus` 上注册一设备对象时，`device_bind_driver` 函数会被调用来将该设备与它的驱动程序绑定起来：

<drivers/base/dd.c>

```
int device_bind_driver(struct device *dev)
{
    int ret;
    ret = driver_sysfs_add(dev);
    if (!ret)
        driver_bound(dev);
    return ret;
}
```

其中 `driver_sysfs_add` 用来在 `sysfs` 文件系统中建立绑定的设备与驱动程序之间的链接符号文件。而 `driver_bound` 函数中关于绑定的最核心的代码为：

```
klist_add_tail(&dev->p->knode_driver, &dev->driver->p->klist_devices);
```

用来将设备 `private` 结构中的 `knode_driver` 节点加入到与该设备绑定的驱动 `private` 结构中的 `klist_devices` 链表中。所以所谓设备与驱动的绑定，从代码的角度看，其实是在两者之间通过某种数据结构的使用建立了一种关联的渠道。

9.3.4 设备

设备在内核中的数据结构为 `struct device`，该类型的实例是对具体设备的一个抽象：

<include/linux/device.h>

```
struct device {
    struct device      *parent;
    struct device_private *p;
    struct kobject      kobj;
    const char        *init_name;
    struct device_type *type;
    struct mutex        mutex;
    struct bus_type     *bus;
    struct device_driver *driver;
    void                *platform_data;
    struct dev_pm_info  power;
```

```

#ifdef CONFIG_NUMA
    int        numa_node;
#endif

    u64        *dma_mask;
    u64        coherent_dma_mask;
    struct device_dma_parameters *dma_parms;
    struct list_head dma_pools;

    struct dma_coherent_mem *dma_mem;
    struct dev_archdata archdata;
    dev_t        devt;
    spinlock_t    devres_lock;
    struct list_head devres_head;
    struct klist_node knode_class;
    struct class    *class;
    const struct attribute_group **groups;
    void (*release)(struct device *dev);
};

```

```
struct device    *parent
```

当前设备的父设备。

```
struct device_private *p
```

指向该设备的驱动相关的数据。

```
struct kobject    kobj
```

代表 struct device 的内核对象。

```
const char    *init_name
```

设备对象的名称。在将该设备对象加入到系统中时，内核会把 init_name 设置成 kobj 成员的名称，后者在 sysfs 中表现为一个目录。

```
struct bus_type *bus
```

设备所在的总线对象指针。

```
struct device_driver *driver
```

用以表示当前设备是否已经与它的 driver 进行了绑定，如果该值为 NULL，说明当前设备还没有找到它的 driver。

系统中的每个设备都是一个 struct device 对象，内核为容纳所有这些设备定义了一个

kset——devices_kset，作为系统中所有 struct device 类型内核对象的容器。同时，内核将系统中的设备分为两大类：block 和 char。每类对应一个内核对象，分别为 sysfs_dev_block_kobj 和 sysfs_dev_char_kobj，自然地这些内核对象也在 sysfs 文件树中占有对应的入口点，block 和 char 内核对象的上级内核对象为 dev_kobj。设备相关的这些事儿发生得比较早，在 Linux 系统初始化期间由 devices_init 来完成，有关设备的故事就从那里开始：

<drivers/base/core.c>

```
int __init devices_init(void)
{
    ...
    devices_kset = kset_create_and_add("devices", &device_uevent_ops, NULL);
    dev_kobj = kobject_create_and_add("dev", NULL);
    sysfs_dev_block_kobj = kobject_create_and_add("block", dev_kobj);
    sysfs_dev_char_kobj = kobject_create_and_add("char", dev_kobj);
    ...
    return 0;
}
```

这个函数的操作反映到/sys 文件目录下，就是生成了/sys/devices、/sys/dev、/sys/dev/block 和/sys/dev/char。

Linux 内核中针对设备的主要操作有：

○ device_initialize

用于设备的初始化，该函数的实现为：

<drivers/base/core.c>

```
void device_initialize(struct device *dev)
{
    dev->kobj.kset = devices_kset;
    kobject_init(&dev->kobj, &device_ktype);
    INIT_LIST_HEAD(&dev->dma_pools);
    mutex_init(&dev->mutex);
    lockdep_set_novalidate_class(&dev->mutex);
    spin_lock_init(&dev->devres_lock);
    INIT_LIST_HEAD(&dev->devres_head);
    device_pm_init(dev);
    set_dev_node(dev, -1);
}
```

这个函数主要用于初始化 dev 的一些成员，其中 dev->kobj.kset = devices_kset 表明了 dev 所属的 kset 对象为 devices_kset，device_pm_init 用来初始化 dev 与电源管理相关的部分。

○ device_register

用来向系统注册一个设备，在源码中的实现为：

```
<drivers/base/core.c>
-----
int device_register(struct device *dev)
{
    device_initialize(dev);
    return device_add(dev);
}
```

所以 `device_register` 内部除了调用 `device_initialize` 来初始化 `dev` 对象外，还会通过 `device_add` 的调用将设备对象 `dev` 加入到系统中。

`device_add` 是个非常重要的函数，对于理解 Linux 设备驱动模型非常有帮助。由于其源代码看起来不是很紧凑，因此这里不打算列出其实现代码，而是将按照其实现的几大逻辑功能进行讨论。该函数的原型为：

```
<include/linux/device.h>
-----
extern int __must_check device_add(struct device *dev);
```

我们把 `device_add` 函数中一些比较重要的功能分成下面几个部分来描述：

在 sysfs 文件系统中建立系统硬件拓扑关系结构图

建立代表 `dev` 的内核对象 `kobject` 的层次关系，简单地讲就是为 `dev` 找到它的上级（parent）内核对象，这个层次关系决定了 `dev` 加入到系统后在 `sysfs` 文件树中的目录层次。代码中关于这种层次关系的建立虽然不是很难理解，但是比较烦琐，而且牵涉到 `dev->class` 成员，根据 `dev->class` 与 `dev->parent` 的值分成四种情况讨论：

（1）`dev->class` 和 `dev->parent` 都为空

由于在对 `dev` 对象调用 `device_initialize` 函数时，曾指定了 `dev` 所属的 `kset` 为 `devices_kset`：`dev->kobj.kset = devices_kset`，所以这种情况下在将 `dev->kobj` 加入系统时，内核会将 `devices_kset` 所对应的 `kobj` 为 `dev->kobj` 的 `parent`，所以 `dev->kobj.parent = devices_kset->kobj`。由于 `devices_kset` 是在 `devices_init` 中建立的设备的顶层 `kset`，这种情况下 `dev` 对象将会在 `/sys/devices` 目录下产生一个新的目录 `/sys/devices/dev->init_name`。

（2）`dev->class` 为空，`dev->parent` 不为空

这种情况下对应 `dev` 对象的新目录将建立在 `dev->parent->kobj` 对应的目录之下。

（3）`dev->class` 不为空，`dev->parent` 为空

`dev->class` 不为空意味着该 `dev` 属于某一 `class`，对于这种情况系统将为 `dev->kobj.parent` 建立一个虚拟上层对象“virtual”，如此，将 `dev` 对象加入系统将会在 `/sys/devices/virtual` 中产生一个新的目录 `/sys/devices/virtual/dev->init_name`。

(4) `dev->class` 和 `dev->parent` 都不为空

这种情况下要看 `dev->parent->class` 是否为空，如果不为空，则 `dev` 的 `parent kobject` 为 `dev->parent->kobj`，即父设备的内嵌 `kobject`。

如果 `dev->parent->class` 为空，则内核需要在 `dev->class->p->class_dirs.list` 中寻找是否有满足条件的 `kobject` 对象 `k`，使得 `k->parent=&parent->kobj`，如果找到那么 `dev->kobj` 的 `parent kobj` 就是 `dev` 设备的父设备的内嵌 `kobject`，否则需要重新生成一个 `kobject` 对象作为 `dev->kobj` 的父 `kobj`。

调用 `device_create_sys_dev_entry(dev)` 建立一个新的链接，该链接的目的和源取决于 `dev->class`。链接源的产生：

```
<drivers/base/core.c>
-----
static struct kobject *device_to_dev_kobj(struct device *dev)
{
    struct kobject *kobj;
    if (dev->class)
        kobj = dev->class->dev_kobj;
    else
        kobj = sysfs_dev_char_kobj;
    return kobj;
}
```

假设 `dev` 对象的设备号 `major=251`，`minor=0`，设备名称为 `dev->init_name`，那么：

如果 `dev->class` 为空，则新链接为 `/sys/dev/char/251:0 /sys/devices/dev->init_name`；

如果 `dev->class` 不为空，那么链接文件的源头将在 `dev->class->dev_kobj` 所对应的目录下产生，目的链接则为 `/sys/devices/virtual/dev->init_name`。

调用 `bus_add_device(dev)`，在 `/sys/bus/devices` 目录下创建一个链接文件，指向 `/sys/devices/dev->init_name`。

假设设备的名称 `dev->init_name` 为“`demodev`”，主次设备号分别为 251 和 0，`dev->class` 为空，那么通过 `device_add` 向系统添加“`demodev`”设备后，`sysfs` 文件树中反映的系统硬件拓扑结构如图 9-5 所示，图中阴影部分为 `device_add` 新增的目录和链接文件：

这里的描述有点抽象，不过对这种层次关系建立细节的理解对于驱动程序员而言并不重要。

在 `sysfs` 文件树中创建与该 `dev` 对象对应的属性文件

`uevent_attr` 是 `dev` 对象的一个属性，其定义如下：

```
static struct device_attribute uevent_attr =
__ATTR(uevent, S_IRUGO | S_IWUSR, show_uevent, store_uevent);
```

```
device_create_file(dev, &uevent_attr)
```

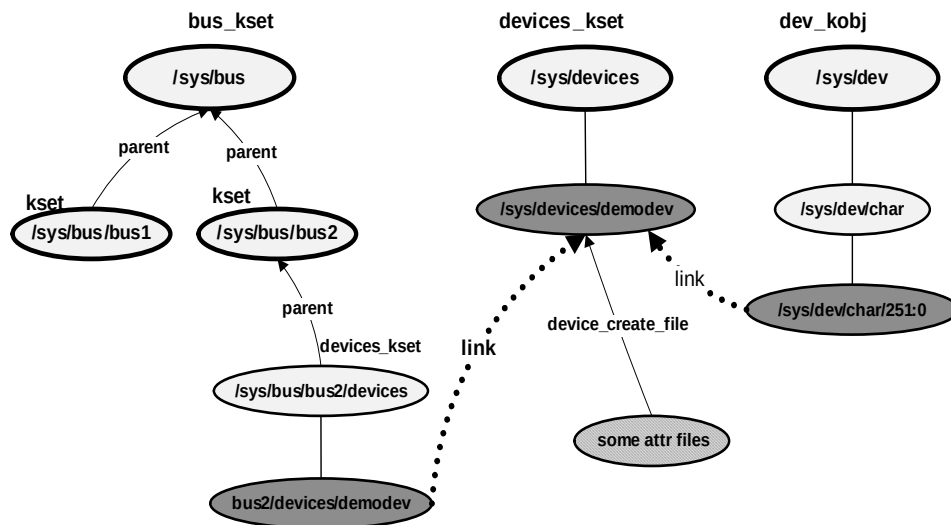


图 9-5 device_add 添加一名为“demodev”的设备后的 sysfs 拓扑图

前面说过，属性文件以文件的形式向用户空间的程序提供了一个显示和更改内核对象属性的方法。这种显示和更改内核对象属性的方法由创建该内核对象的模块提供，换句话说，如果设备驱动程序需要给用户空间的程序提供这种能力，那么由驱动程序来实现这些显示和更改属性的函数。对于 `device_register` 而言，内核已经为 `uevent_attr` 属性提供了默认的显示和修改的函数 `show_uevent` 和 `store_uevent`。

如果说前面介绍的两个功能还稍显平淡的话，下面的这个功能就比较有趣了。如果 `dev` 对象中指定的主设备号不为 0：if (`MAJOR(dev->devt)`)，那么函数除了会在 sysfs 中新增一个属性文件“dev”外，还会调用 `devtmpfs_create_node(dev)` 在 `/dev` 目录下动态生成一个设备节点。读者也许知道，在 Linux 的早期，`/dev` 目录下的设备节点需要用 `mknod` 命令手动添加，现在通过 `devtmpfs` 文件系统，就可以在 `device_register` 注册设备时自动向 `/dev` 目录添加设备节点，该节点的名字就是 `dev->init_name`。

关于 `devtmpfs` 文件系统，它是内核建立的另一棵独立的 VFS 树，最终挂载（mount）到用户空间的 `/dev` 目录之上，在内核中 `devtmpfs` 主要用来动态生成设备节点。关于这个文件系统的实现细节，本书不再详细讨论，对此感兴趣的读者可以参考 <http://www.embexperts.com/viewthread.php?tid=4&extra=page%3D1>。

关于 `class` 的相关操作，将在接下来的“class”一节中专门讨论。

一个体现设备驱动模型中总线、设备与驱动相互沟通的重要函数调用 `bus_probe_device(dev)`，该函数的实现如下：

<drivers/base/bus.c

```
void bus_probe_device(struct device *dev)
{
    struct bus_type *bus = dev->bus;
    int ret;

    if (bus && bus->probe->drivers_autoprobe) {
        ret = device_attach(dev);
        WARN_ON(ret < 0);
    }
}
```

如果满足 if 语句中的条件，将会调用 device_attach 试图将当前的设备绑定到它的驱动程序上，device_attach 进行绑定的核心代码如下：

<drivers/base/dd.c

```
int device_attach(struct device *dev)
{
    ...
    if (dev->driver) {
        device_bind_driver(dev);
    } else {
        ret = bus_for_each_drv(dev->bus, NULL, dev, __device_attach);
    }
    ...
}
```

如果 dev->driver 不为空，表明当前的设备对象 dev 已经和它的驱动程序进行了绑定，这种情况下只需调用 device_bind_driver(dev)在 sysfs 文件树中建立 dev 与其驱动程序之间的互联关系。

如果 dev->driver 为空，表明当前设备对象 dev 还没有和它的驱动程序绑定，此时需要遍历 dev 所在总线 dev->bus 上挂载的所有驱动程序对象：

```
bus_for_each_drv(dev->bus, NULL, dev, __device_attach);
```

然后对遍历过程中的每个驱动程序对象 drv，调用 __device_attach(drv, dev)进行绑定：

<drivers/base/dd.c>

```
static int __device_attach(struct device_driver *drv, void *data)
{
    struct device *dev = data;

    if (!driver_match_device(drv, dev))
        return 0;
```



```

        return driver_probe_device(drv, dev);
    }

```

函数中的 `driver_match_device(drv, dev)` 用来判断 `drv` 与 `dev` 是否匹配，其内部的实现为：

```

return drv->bus->match ? drv->bus->match(dev, drv) : 1;

```

意味着如果当前 `drv` 对象所在的总线定义了 `match` 方法，那么就调用它来进行是否匹配的判断；如果总线没有定义 `match`，那么 `driver_match_device(drv, dev)` 函数返回 1，表明匹配成功，不成功返回 0，`device_attach` 函数继续对 `dev->bus` 上的下一个驱动程序对象进行匹配操作。

如果 `driver_match_device` 匹配成功，那么将调用 `driver_probe_device(drv, dev)` 将 `drv` 和 `dev` 进行绑定，这个工作实际上是由 `really_probe` 函数来完成的：

<drivers/base/dd.c>

```

static int really_probe(struct device *dev, struct device_driver *drv)
{
    ...
    dev->driver = drv;
    if (driver_sysfs_add(dev)) {
        printk(KERN_ERR "%s: driver_sysfs_add(%s) failed\n",
               __func__, dev_name(dev));
        goto probe_failed;
    }

    if (dev->bus->probe) {
        ret = dev->bus->probe(dev);
    } else if (drv->probe) {
        ret = drv->probe(dev);
    }

    driver_bound(dev);
    ...
}

```

函数首先将当前驱动程序对象 `drv` 赋值给 `dev->driver`，然后，如果 `dev->bus->probe` 不为空，即 `dev` 所在的总线定义了 `probe` 方法，则调用之，否则如果 `drv` 对象定义了该方法，就调用 `drv->probe(dev)`，所以现在知道了我们 `driver` 中定义的 `probe` 函数什么时候会被调用到。这种设计机制给驱动程序提供了一个探测硬件的机会，即在其 `probe` 函数中作出判断：当前的设备是不是自己所支持的，以及当前设备是否处于工作状态等。驱动程序中实现的 `probe` 函数如果认为探测成功，那么应该返回 0。

最后的 `driver_bound(dev)` 用来将驱动程序的一些数据信息加入到 `dev` 对象中。

○ device_unregister

用来将一个设备从系统中注销掉，其实现为：

<drivers/base/core.c>

```
void device_unregister(struct device *dev)
{
    device_del(dev);
    put_device(dev);
}
```

函数的重点在 device_del 中：

```
void device_del(struct device *dev)
{
    struct device *parent = dev->parent;
    struct class_interface *class_intf;

    if (dev->bus)
        blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                                      BUS_NOTIFY_DEL_DEVICE, dev);
    //设备电源管理函数，关闭本设备电源同时通知其父设备（如果有的话）
    device_pm_remove(dev);
    dpm_sysfs_remove(dev);
    //有父设备，将当前设备从父设备所属链表中删除
    if (parent)
        klist_del(&dev->p->knode_parent);
    //如果 dev 设备对象的主设备号不为 0
    if (MAJOR(dev->devt)) {
        //动态删除设备节点文件
        devtmpfs_delete_node(dev);
        device_remove_sys_dev_entry(dev);
        //删除设备的属性文件
        device_remove_file(dev, &devt_attr);
    }
    if (dev->class) {
        device_remove_class_symlinks(dev);
        mutex_lock(&dev->class->p->class_mutex);
        /* notify any interfaces that the device is now gone */
        list_for_each_entry(class_intf, &dev->class->p->class_interfaces, node)
            if (class_intf->remove_dev)
                class_intf->remove_dev(dev, class_intf);
        /* remove the device from the class list */
        klist_del(&dev->knode_class);
        mutex_unlock(&dev->class->p->class_mutex);
    }
    device_remove_file(dev, &uevent_attr);
}
```

```

    device_remove_attrs(dev);
    bus_remove_device(dev);
    devres_release_all(dev);

    if (platform_notify_remove)
        platform_notify_remove(dev);
    kobject_uevent(&dev->kobj, KOBJ_REMOVE);
    cleanup_device_parent(dev);
    kobject_del(&dev->kobj);
    put_device(parent);
}

```

9.3.5 驱动

内核为驱动对象定义的数据结构是 struct device_driver:

```

<include/linux/device.h>
-----
struct device_driver {
    const char      *name;
    struct bus_type *bus;
    struct module   *owner;
    const char      *mod_name;
    bool suppress_bind_attrs;

    int (*probe) (struct device *dev);
    int (*remove) (struct device *dev);
    void (*shutdown) (struct device *dev);
    int (*suspend) (struct device *dev, pm_message_t state);
    int (*resume) (struct device *dev);

    const struct attribute_group **groups;
    const struct dev_pm_ops *pm;
    struct driver_private *p;
};

```

const char *name

驱动的名称。

struct bus_type *bus

驱动所属的总线。

struct module *owner

驱动所在的内核模块。

int (*probe) (struct device *dev)

驱动程序所定义的探测函数。当在总线 bus 中将该驱动与对应的设备进行绑定时，内核会首先调用 bus 中的 probe 函数（如果该 bus 实现了 probe 函数），如果 bus 没有实现自己的 probe 函数，那么内核会调用驱动程序中实现的 probe 函数。

int (*remove) (struct device *dev)

驱动程序所定义的卸载函数。当调用 driver_unregister 从系统中卸载一个驱动对象时，内核会首先调用 bus 中的 remove 函数（如果该 bus 实现了 remove 函数），如果 bus 没有实现自己的 remove 函数，那么内核会调用驱动程序中实现的 remove 函数。

驱动上的主要操作有：

○ driver_find

在一个 bus 的 drivers_kset 集合中查找指定的驱动，函数原型为：

```
struct device_driver *driver_find(const char *name, struct bus_type *bus)
```

参数 name 是要查找的驱动的名称，参数 bus 指明在哪个总线上进行当前的查找。如果查找成功，将返回该驱动对象的指针，否则返回 0。

○ driver_register

该函数用来向系统注册一个驱动，其核心实现代码为：

<drivers/base/driver.c>

```
int driver_register(struct device_driver *drv)
{
    int ret;
    struct device_driver *other;
    ...
    other = driver_find(drv->name, drv->bus);
    ...
    ret = bus_add_driver(drv);
    ...
}
```

函数首先调用 driver_find 在 drv->bus 上查找当前要注册的 drv，这主要是防止向系统重复注册同一个驱动，如果当前要注册的驱动没有被注册过，那么将调用 bus_add_driver(drv) 进行实际的注册操作。

<drivers/base/bus.c>

```
int bus_add_driver(struct device_driver *drv)
{
```

```

    struct driver_private *priv;
    ...
    bus = bus_get(drv->bus);
    if (!bus)
        return -EINVAL;
    priv = kzalloc(sizeof(*priv), GFP_KERNEL);
    if (!priv) {
        error = -ENOMEM;
        goto out_put_bus;
    }
    klist_init(&priv->klist_devices, NULL, NULL);
    priv->driver = drv;
    drv->p = priv;
    priv->kobj.kset = bus->p->drivers_kset;
    error = kobject_init_and_add(&priv->kobj, &driver_ktype, NULL,
                                "%s", drv->name);

    if (drv->bus->p->drivers_autoprobe) {
        error = driver_attach(drv);
        if (error)
            goto out_unregister;
    }
    klist_add_tail(&priv->knode_bus, &bus->p->klist_drivers);
    module_add_driver(drv->owner, drv);
    ...
    kobject_uevent(&priv->kobj, KOBJ_ADD);
    return 0;
}

```

函数首先为 `drv` 分配了一块类型为 `struct driver_private` 的空间对象 `priv`，然后将其与 `drv` 对象建立了关联，同时调用 `kobject_init_and_add` 把 `drv` 所对应的内核对象加入到 `sysfs` 文件树中，如此将在 `/sys/bus/drivers` 目录下新建一目录，其名称为 `drv->name`。

如果 `drv` 所在的 `bus` 对应的 `drivers_autoprobe` 属性值为 1，将调用 `driver_attach` 将当前注册的 `drv` 与该 `bus` 上所属的设备进行绑定。绑定的过程将遍历 `bus` 上的所有设备，对于其中的每个设备 `dev`，将调用 `really_probe(dev, drv)` 进行实际的绑定操作。如果此前 `bus` 上定义了 `match` 方法，则它将被首先调用以确定 `drv` 与 `dev` 是否 `match`，如果不 `match`，那么将继续遍历下一个设备，否则调用 `really_probe` 进行实际的绑定操作。`really_probe(dev, drv)` 函数如能将 `drv` 与 `dev` 成功绑定，则将在 `sysfs` 文件树中通过链接文件为 `dev` 和 `drv` 所对应的内核对象建立拓扑关系。同时，如果所在 `bus` 上定义有 `probe` 函数，将调用之，否则如果当前要注册的 `drv` 定义有 `probe` 函数，那么将调用之。

在 `bus_add_driver` 函数中，也会通过调用 `driver_create_file` 函数在新建的 `drv` 目录中生成属性文件，比如 `driver_create_file(drv, &driver_attr_uevent)` 等。驱动的属性由宏 `DRIVER_ATTR`

来定义：

```
<include/linux/device.h>
-----
#define DRIVER_ATTR(_name, _mode, _show, _store)    \
struct driver_attribute driver_attr_##_name =      \
    __ATTR(_name, _mode, _show, _store)
```

○ driver_unregister

该函数用来将某一指定的驱动从系统中注销掉。函数原型为：

```
void driver_unregister(struct device_driver *drv)
```

参数 `drv` 用于指定要注销的某一驱动对象。函数基本上是做 `driver_register` 的反向工作，其主要的工作是在 `bus_remove_driver(drv)` 函数中完成的。需要注意的是，在注销一个驱动对象的过程中，如果其所在的总线定义了 `remove` 方法，那么内核会调用它，否则要看驱动所在的驱动程序中有没有实现该方法，如果实现了的话内核会调用该函数。

9.4 class

Linux 设备驱动模型中的另一个比较重要的概念是类 `class`，相对于设备 `device`，`class` 是一种更高层次的抽象，用于对设备进行功能上的划分，有时候也被称为设备类。Linux 的设备模型引入类，是将其用来作为具有同类型功能设备的一个容器。

Linux 为类定义的数据结构是：

```
<include/linux/device.h>
-----
struct class {
    const char      *name;
    struct module    *owner;
    struct class_attribute *class_attr;
    struct device_attribute *dev_attr;
    struct kobject    *dev_kobj;

    int (*dev_uevent)(struct device *dev, struct kobj_uevent_env *env);
    char *(*devnode)(struct device *dev, mode_t *mode);
    void (*class_release)(struct class *class);
    void (*dev_release)(struct device *dev);
    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);

    const struct kobj_ns_type_operations *ns_type;
    const void *(*namespace)(struct device *dev);
```

```
const struct dev_pm_ops *pm;
struct class_private *p;
};
```

const char *name

类的名称。

struct module *owner

拥有该类的模块的指针。

struct class_attribute *class_attrs

类的属性。

struct device_attribute *dev_attrs

设备的属性。

struct kobject *dev_kobj

代表当前类中设备的内核对象。

struct class_private *p

类的私有数据区，用于处理类的子系统及其所包含的设备链表。

内核针对类对象定义的主要操作有：

○ classes_init

系统中类的起源函数，在系统初始化期间调用，主要作用是产生类对象的顶层 kset——class_kset：

<drivers/base/class.c>

```
int __init classes_init(void)
{
    class_kset = kset_create_and_add("class", NULL, NULL);
    if (!class_kset)
        return -ENOMEM;
    return 0;
}
```

函数中对 `kset_create_and_add("class", NULL, NULL)` 的调用将导致在 `/sys` 目录下新生成一个“class”目录（`/sys/class`），在以后的 class 相关的操作中，`class_kset` 将作为系统中所有 class 内核对象的顶层 kset。

○ class_create

```
<include/linux/device.h>
-----
#define class_create(owner, name) \
({ \
    static struct lock_class_key __key; \
    __class_create(owner, name, &__key); \
})
```

宏 `class_create` 用来生成一个类对象，其用途主要是将同类型的设备添加其中。该宏的核心是对函数 `__class_create` 的调用，该函数定义如下：

```
<drivers/base/class.c>
-----
struct class *__class_create(struct module *owner, const char *name,
                           struct lock_class_key *key)
{
    struct class *cls;
    int retval;

    cls = kzalloc(sizeof(*cls), GFP_KERNEL);
    ...
    cls->name = name;
    cls->owner = owner;
    cls->class_release = class_create_release;

    retval = __class_register(cls, key);
    ...
    return cls;
}
```

函数会动态生成一个 `class` 对象，经过一些初步的初始化之后，调用 `__class_register` 向系统注册该新生成的类对象。限于篇幅，此处将不再列出 `__class_register` 函数的源代码，而是直接介绍其主要功能。`__class_register` 会首先为 `__class_create` 函数中生成的新的类对象分配私有数据空间（`struct class_private *cp = kzalloc(sizeof(*cp), GFP_KERNEL)`），代表 `class` 内核对象的 `kobject` 内嵌在 `struct class_private` 数据结构的 `class_subsys` 成员中（`class_subsys.kobj`），

函数将类对象的 `name` 成员赋值给代表类的 `kobject` 对象名称（`kobject_set_name(&cp->class_subsys.kobj, "%s", cls->name)`），同时为类的 `kobj` 指定 `kset` 和 `ktype`：

```
<drivers/base/class.c>
-----
int __class_register(struct class *cls, struct lock_class_key *key)
{
    ...
    cp->class_subsys.kobj.kset = class_kset;
    cp->class_subsys.kobj.ktype = &class_ktype;
```


之前讨论过 `class_kset` 为系统中所有 `class` 对象的顶层 `kset`, 此处将当前 `class` 对象的 `kobj.kset` 指向 `class_kset`, 意味着通过 `class_create` 生成的 `class`, 在 `sysfs` 文件系统中的入口点 (目录) 将在 `/sys/class` 目录下产生。

函数接下来调用 `kset_register` 将之前产生的 `class` 加入到系统中:

```
kset_register(&cp->class_subsys);
```

这样将会在 `/sys/class` 目录下生成一个新的目录。

○ `class_destroy`

用于从系统中注销一个 `class` 对象, 函数原型为:

```
void class_destroy(struct class *cls);
```

○ `device_create`

本来这个函数应该是属于设备相关的操作范畴, 但是因为 `class` 的引入, 使得设备的创建与 `class` 产生了相关性, 所以我们把这个创建设备的函数放到 `class` 一节中讲解。

`device_create` 的源码为:

```
<drivers/base/core.c>
-----
struct device *device_create(struct class *class, struct device *parent,
                             dev_t devt, void *drvdata, const char *fmt, ...)
{
    va_list vargs;
    struct device *dev;
    va_start(vargs, fmt);
    dev = device_create_vargs(class, parent, devt, drvdata, fmt, vargs);
    va_end(vargs);
    return dev;
}
```

函数的核心在 `device_create_vargs` 调用中, 其功能基本和 `device_register` 相同, 但是 `device_create_vargs` 会为设备指定 `class`:

```
dev->class = class;
```

之前在讨论 `device_register` 函数时曾讨论过“在 `sysfs` 文件系统中建立系统硬件拓扑关系结构图”的四种情况, 如果用 `device_create` 向系统中增加设备, 显然属于这四种情况中的后两种。

○ `device_destroy`

函数原型为:

```
void device_destroy(struct class *class, dev_t devt)
```

该函数用于从系统中移除通过 `device_create` 增加的设备 `device`。

9.5 本章小结

本章讨论了 Linux 的设备驱动模型，该模型是个非常复杂的系统，从一个比较高的层次来看，主要由总线、设备和驱动构成。内核为了实现这些组件间的相关关系，定义了 `kobject` 和 `kset` 这样的基础底层数据结构，然后通过 `sysfs` 文件系统向用户空间展示发生在内核空间中的各组件间的互联层次关系，并以文件系统接口的方式为用户空间程序提供了访问内核对象属性信息的简易方法。

Linux 设备模型通过总线将系统中的设备和驱动关联起来，由于设备和驱动的分离，增加了系统设计的灵活性，伴随而来的代价就是增加了复杂度。

第 10 章

内存映射与DMA

本章将讨论设备驱动程序中如何实现内存映射和进行 DMA 操作。内存映射与第 3 章中提到的内存分配不同，它要完成的任务是将设备的地址空间映射到用户空间或者直接使用用户空间中的地址，设备程序这样做的目的显然是从提升系统性能的角度出发。如果将这种概念更具体化，内存映射部分实际上是描述如何实现设备驱动程序中 `file_operations` 中的 `mmap` 方法。

本章还将讨论设备的 DMA (Direct Memory Access) 操作，主要是在设备缓冲区和系统主内存间如何传输数据，因为这种传输不需要 CPU 的参与，所以可以极大提升系统性能，因此对于设备驱动程序而言，深入理解 Linux 内核实现的 DMA 接口的背后机制，有利于在系统设计时采用最佳的解决方案以提高系统吞吐量。DMA 操作的核心是如何为一个 DMA 传输通道建立源地址和目标地址，也就是所谓的 DMA 映射问题。

10.1 设备缓存与设备内存

在继续下面的讨论前，需要澄清或者界定两个概念：设备缓存和设备内存。

设备缓存是由驱动程序管理的位于系统主存 RAM 中的一段内存区域，而设备内存则是设备所固有的一段存储空间（比如某些设备的 FIFO，显卡设备的 Frame Buffer 等），从设备驱动程序的角度，它应该属于特定设备的硬件范畴，与设备是紧密相关的。

Linux 系统下设备缓存与设备内存的典型用法是在两者之间建立 DMA 通道，这样当设备内存中接收到的数据达到一定的阈值时，设备将启动 DMA 通道将数据从设备内存传输到位于主存中的设备缓存中，发送数据则正好相反，需要发送的数据首先被放到设备缓存中，然后在设备驱动程序的介入下启动 DMA 传输，将缓存中的数据传输到设备内存中。在本章接下来的讨论中，有可能将这两个概念混用，但在特定的上下文中会指明是哪种内存。

10.2 mmap

在“分配内存”一章中曾经讨论过 `ioremap` 函数，这个函数主要用来将内核空间的一段虚

拟地址映射到外部设备的存储区（设备的 I/O 地址空间）中。本节要讨论的 `mmap` 则用来将用户空间的一段虚拟地址映射到设备的 I/O 空间中，这样一来，用户空间进程将可以直接访问设备内存。驱动程序在此要完成的功能则是在其内部实现 `file_operations` 中的 `mmap` 方法。

10.2.1 struct vm_area_struct

先从 `file_operations` 中定义的 `mmap` 方法的原型看起：

```
<include/linux/fs.h>
-----
struct file_operations {
    ...
    int (*mmap) (struct file *, struct vm_area_struct *);
    ...
};
```

`mmap` 函数的第一个参数是用来表示当前正在操作的一个 `struct file` 对象指针，第二个参数用来表示用户进程中一段需要被映射的虚拟地址区域。结构体 `struct vm_area_struct` 中的一些关键成员定义如下：

```
<include/linux/mm_types.h>
-----
struct vm_area_struct {
    struct mm_struct * vm_mm; /* The address space we belong to. */
    unsigned long vm_start; /* Our start address within vm_mm. */
    unsigned long vm_end; /* The first byte after our end address
                           within vm_mm. */

    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next, *vm_prev;

    pgprot_t vm_page_prot; /* Access permissions of this VMA. */
    unsigned long vm_flags; /* Flags, see mm.h. */

    /* Function pointers to deal with this struct. */
    const struct vm_operations_struct *vm_ops;

    ...
};
```

```
struct mm_struct * vm_mm
```

当前 `struct vm_area_struct` 对象所表示的虚拟地址段所归属的进程虚拟地址空间。

```
unsigned long vm_start
```

当前 `struct vm_area_struct` 对象所表示的虚拟地址段的起始地址。

`unsigned long vm_end`

当前 `struct vm_area_struct` 对象所表示的虚拟地址段的结束地址。

`struct vm_area_struct *vm_next, *vm_prev`

用来将一系列的 `struct vm_area_struct` 对象构建成链表。代表进程虚拟地址空间的 `struct mm_struct` 对象中的 `struct vm_area_struct * mmap` 成员用来指向该链表。

`pgprot_t vm_page_prot`

在将当前 `struct vm_area_struct` 对象所表示的虚拟地址段映射到设备内存时的页保护属性，主要体现在页目录（表）项的映射属性当中。

`unsigned long vm_flags`

当前 `struct vm_area_struct` 对象所表示的虚拟地址段的访问属性，比如 `VM_READ`、`VM_WRITE`、`VM_EXEC` 及 `VM_SHARED` 等。

`const struct vm_operations_struct *vm_ops`

用来定义对当前 `struct vm_area_struct` 对象所表示的虚拟地址段上的一组操作集。

内核中的每个 `struct vm_area_struct` 对象都表示用户进程地址空间的一段区域，它是访问用户进程中 `MMAP` 地址空间的最小单元，内核为管理这些 `struct vm_area_struct` 对象准备了大量的代码，一个核心的数据结构是红黑树，不过对这些数据结构和算法的讨论不是本书的主题。

在继续讨论设备驱动程序如何将用户地址空间中的虚拟地址映射到它的设备 I/O 空间之前，笔者会先介绍一下用户进程的虚拟地址空间布局，然后再介绍 `mmap` 的系统调用过程，因为这将有利于读者了解到内存映射的整体框架和用户空间程序如何利用驱动程序提供的 `mmap` 机制。

10.2.2 用户空间虚拟地址布局

因为 `mmap` 用来映射用户空间的虚拟地址，所以有必要搞清楚在 Linux 系统下一个进程的用户虚拟地址空间的布局，此处的讨论按照经典的 x86 架构的 3 GB/1 GB 方式展开，也即用户空间虚拟地址大小是 3 GB，内核空间虚拟地址大小是 1 GB。此处的布局是指在 3 GB 的进程虚拟地址空间中规划出进程的代码段（`text`）、存储全局变量和动态分配变量地址的堆，以及用于保存局部变量和实现函数调用的栈等存储块的起始地址和大小。

Linux 内核中采用两种布局方式，在此先给出这两种布局示意图以期读者对后续的讨论有个直观的印象。两种布局如图 10-1 所示：

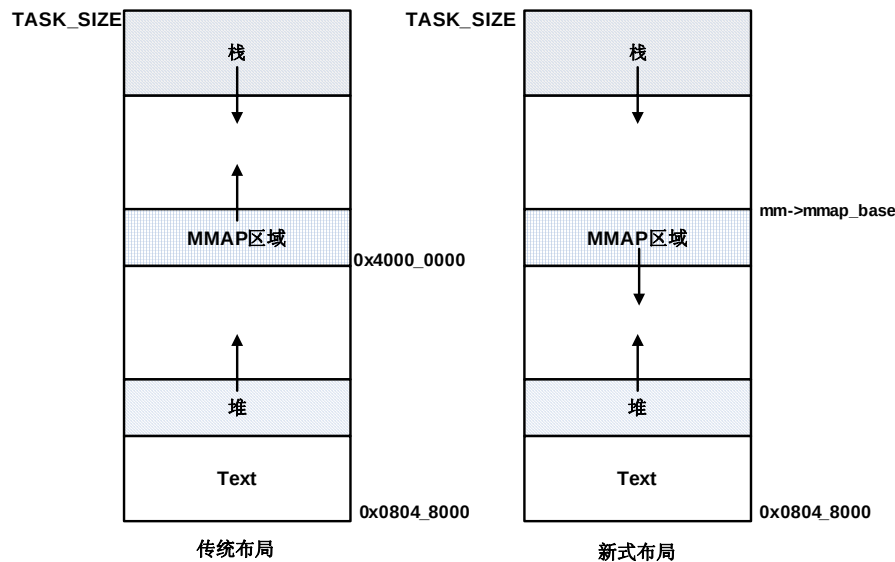


图 10-1 用户进程虚拟地址空间的两种布局方式

对用户进程虚拟地址空间布局的设计是操作系统要完成的任务之一。当 Linux 系统运行一个应用程序时，系统调用 exec 通过调用 load_elf_binary 函数来将该应用程序对应的 ELF 二进制文件加载到进程 3 GB 大小的虚拟地址空间中，布局由此产生。

这两种布局的区别并不是本书要关注的重点，这里给出这两种布局是希望读者在本章后续的讨论中先建立一个全局概念，如果读者对布局相关细节比较好奇，可以看看下文中一些概要性的介绍。马上会谈到驱动程序中 mmap 方法的实现，这也是本章前半部分的主题：内核如何配合驱动程序将用户进程虚拟地址空间中的 MMAP 区域的某段地址映射到设备内存中。

load_elf_binary 函数建立布局相关的函数调用链是：load_elf_binary()→setup_new_exec()→arch_pick_mmap_layout()。

```
<arch/x86/mm/mmap.c>
-----
void arch_pick_mmap_layout(struct mm_struct *mm)
{
    if (mmap_is_legacy()) {
        mm->mmap_base = mmap_legacy_base();
        mm->get_unmapped_area = arch_get_unmapped_area;
        mm->unmap_area = arch_unmap_area;
    } else {
        mm->mmap_base = mmap_base();
    }
}
```

```

        mm->get_unmapped_area = arch_get_unmapped_area_topdown;
        mm->unmap_area = arch_unmap_area_topdown;
    }
}

```

函数中的参数 `mm` 是一类型为 `struct mm_struct` 的对象指针，Linux 系统中每个进程都拥有一个 `struct mm_struct` 类型的对象，该对象保存了进程中与内存管理相关的信息。Linux 内核为进程的虚拟空间提供了两种布局方案，在本书中分别称之为传统（legacy）布局和新式布局。`arch_pick_mmap_layout` 函数用 `mmap_is_legacy()` 来判断是采用传统布局还是新式布局：

```

<arch/x86/mm/mmap.c>
-----
static int mmap_is_legacy(void)
{
    if (current->personality & ADDR_COMPAT_LAYOUT)
        return 1;

    if (rlimit(RLIMIT_STACK) == RLIM_INFINITY)
        return 1;

    return sysctl_legacy_va_layout;
}

```

`mmap_is_legacy` 函数决定了对一个进程的虚拟地址空间是采用传统布局还是新式布局：

- （1）如果 `current->personality` 设置了 `ADDR_COMPAT_LAYOUT` 位，那么将采用传统布局。
- （2）如果进程空间对栈的增长没有限制，那么也将采用传统布局。
- （3）如果以上两个条件都未能满足，那么布局的方式将由 `sysctl` 的 `sysctl_legacy_va_layout` 参数来控制。

从 `arch_pick_mmap_layout` 函数的代码可以看出，传统布局与新式布局的主要区别在于 `MMAP` 区域的扩展方向。在传统布局中 `mm->mmap_base = mmap_legacy_base()`，如果把这条调用链展开，基本上可以认为 `mm->mmap_base = TASK_UNMAPPED_BASE`，在 x86 平台上，`TASK_UNMAPPED_BASE=3 GB/3=1 GB`。也就是说，对于传统布局，`MMAP` 区域的起始地址从 `0x40000000` 处开始，`mm->get_unmapped_area = arch_get_unmapped_area` 表示用来获得 `MMAP` 区域尚未被映射的一段内存的函数，传统布局使用 `arch_get_unmapped_area`，后者是一体系结构相关的函数，但是内核为此提供了一个通用的函数，当系统没有定义 `HAVE_ARCH_UNMAPPED_AREA` 宏时，内核将调用这个通用的函数来在用户进程的 `MMAP` 区域分配尚未被映射的内存块。该函数将从低地址向高地址方向分配空闲的 `struct vm_area_struct` 对象，每个对象代表一段连续的虚拟地址空间。这里暗含的概念是对进程 3 GB 的虚拟地址空间中 `MMAP` 区域的地址进行分配，目的是要找到一

块尚未被映射的内存区域。

对于新式布局, `mm->mmap_base = mmap_base()`, 说明该布局下 MMAP 空间的起始地址由 `mmap_base` 函数来决定:

```
<arch/x86/mm/mmap.c>
-----
static unsigned long mmap_base(void)
{
    unsigned long gap = rlimit(RLIMIT_STACK);
    if (gap < MIN_GAP)
        gap = MIN_GAP;
    else if (gap > MAX_GAP)
        gap = MAX_GAP;
    return PAGE_ALIGN(TASK_SIZE - gap - mmap_rnd());
}
```

函数首先通过 `rlimit(RLIMIT_STACK)` 获得当前进程的栈空间的最大值, 然后通过 `PAGE_ALIGN(TASK_SIZE - gap - mmap_rnd())` 来获得新式布局下 MMAP 空间的起始地址。

`arch_pick_mmap_layout` 函数中的 `mm->get_unmapped_area = arch_get_unmapped_area_topdown` 指明了新式布局下在 MMAP 区域分配空闲 `struct vm_area_struct` 对象的方式, 此处不会详细讨论该函数的实现机制, 读者只需记住这个函数将从高地址向低地址在 MMAP 区域中分配空闲的 `vm_area_struct` 对象, 每个 `vm_area_struct` 对象代表一段连续的虚拟内存空间。

对于驱动程序程序员而言, 了解两种布局的区别并不是重点。重点是知道要映射的地址区域出自 3 GB 大小的用户空间的 MMAP 区域 (读者马上将看到此处 MMAP 区域的具体指代), 内核会很好地管理 MMAP 区域, 管理该区域的最小单位是由 `struct vm_area_struct` 数据结构表示的对象, 此处管理的语义是分配和释放一个 `vm_area_struct` 对象 (想象一下用户进程对 `mmap` 和 `munmap` 等 API 的调用)。系统中一个实际进程的 MMAP 区域看起来可能如图 10-2 所示:

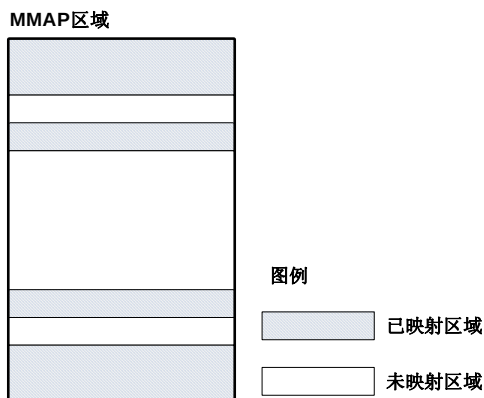


图 10-2 一个进程虚拟地址空间中 MMAP 区域的状态

从图中可以看到一个进程用户虚拟地址空间中的 MMAP 区域的空间状态，因为响应应用程序中 `mmap` 和 `munmap` 等系统调用的缘故，MMAP 区域充斥了映射的区域和尚未被映射的空闲区域。每个区域由一个 `struct vm_area_struct` 对象表示，直觉上可以知道内核必须跟踪 MMAP 区域的分配情况（这跟“分配内存”一章中讨论过的 `vmalloc` 机制极其相似），并且应该能很好地处理从 MMAP 区域分配一个待映射的 `vm_area_struct` 对象或者释放一个被映射的 `vm_area_struct` 对象。显然这不是一件轻松的事情，幸运的是内核基本上为我们打点了所有这一切的细节。

大体上，这种用户空间虚拟地址映射到设备内存的过程可以概括为：内核先在进程虚拟地址空间的 MMAP 区域分配一个空闲（即未映射）的 `struct vm_area_struct` 对象，然后通过页目录表项的方式将 `struct vm_area_struct` 对象所代表的虚拟地址空间映射到设备的存储空间中。如此，用户进程将可以直接访问设备的存储区，从而提高系统性能。页目录表项的介入也意味着每个 `vm_area_struct` 对象表示的地址空间应该是页对齐的，大小是页的整数倍。

10.2.3 mmap 系统调用过程

在用户空间，`mmap` 系统调用的函数原型为：

```
void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

其中，`start` 表示映射区的起始地址，`length` 是映射区的长度，`prot` 表示用户进程在映射区被映射时所期望的保护方式，常见的 `prot` 值有 `PROT_READ`、`PROT_WRITE` 及 `PROT_EXEC` 等，`flags` 用于指定映射区的类型，`fd` 是当前正在操作的文件的描述符，`offset` 是实际数据在映射区中的偏移值。在实际使用中，`start` 参数常常设为 `NULL`，表示让系统在 MMAP 区域找一个合适的空闲区域。如果一切正常，`mmap` 函数将返回已经被映射的 MMAP 区域中一段虚拟地址的起始地址，应用程序因此可以访问到对应的物理内存。

当用户空间程序调用 `mmap` 函数时，Linux 系统将通过系统调用 `sys_mmap_pgoff` 进入内核，由当前设备文件中实现的 `mmap` 方法来完成用户程序所要求的映射。`sys_mmap_pgoff` 的核心代码如下：

```
<mm/mmap.c>
-----
SYSCALL_DEFINE6(mmap_pgoff, unsigned long, addr, unsigned long, len,
                unsigned long, prot, unsigned long, flags,
                unsigned long, fd, unsigned long, pgoff)
{
    struct file *file = NULL;
    ...
    file = fget(fd);
```

```

        flags &= ~(MAP_EXECUTABLE | MAP_DENYWRITE);
        down_write(&current->mm->mmap_sem);
        retval = do_mmap_pgoff(file, addr, len, prot, flags, pgoff);
        up_write(&current->mm->mmap_sem);
        ...
    }

```

`sys_mmap_pgoff` 除了作一些错误检查之外，主要做两件事：一是通过 `fget` 函数由文件描述符获得对应的 `struct file` 对象指针；二是调用 `do_mmap_pgoff` 来完成后续的内存映射工作。

`do_mmap_pgoff` 的函数实现比较长，但是对比一下驱动程序中要实现的 `mmap` 方法的原型，基本上可以推测出 `do_mmap_pgoff` 函数的主体脉络应该是根据用户空间进程调用 `mmap` API 时传入的参数构造一个 `struct vm_area_struct` 对象的实例，然后调用 `file->f_op->mmap()`。

本来到此我们可以直接转到对设备驱动程序如何实现其 `mmap` 方法的讨论，但是如果略过 `do_mmap_pgoff` 的某些实现细节，对本书的读者来说，是个损失。因为 `do_mmap_pgoff` 的实现过程虽然不是那么流畅，但是其中还是有一些代码颇值得仔细玩味一番。因此，读者如果有兴趣的，我们不妨仔细看看 `do_mmap_pgoff` 函数的实现，相信应该是不虚此行的。

函数 `do_mmap_pgoff` 的代码比较长，下面将按照其主要功能逻辑分段摘录讨论。

<mm/mmap.c>

```

-----
unsigned long do_mmap_pgoff(struct file *file, unsigned long addr,
                           unsigned long len, unsigned long prot,
                           unsigned long flags, unsigned long pgoff)
{
    //一些防御性代码的常规检查
    ...
    len = PAGE_ALIGN(len);

```

此处要确保映射区的长度应该是一个 `PAGE` 大小的整数倍，因为映射发生时，最小的单位就是一个页，这是由体系结构中的 `MMU` 单元的特性决定的。

```

    if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
        return -EOVERFLOW;

```

检查参数中的 `pgoff` 是否会溢出（`OVERFLOW`），这告诉用户空间程序员在使用 `mmap` API 时需要保证 `offset` 参数的合法性。

```

    addr = get_unmapped_area(file, addr, len, pgoff, flags);
    if (addr & ~PAGE_MASK)
        return addr;

```

本函数中比较重要的一个调用，用来在进程的 3 GB 的虚拟地址空间中分配一块空闲区域。在前面“进程虚拟地址空间布局”一节中曾提到内核的两种布局方式，对任一布局而言，

都会对当前进程的 mm 对象中的 get_unmapped_area 成员进行赋值，对传统布局是 mm->get_unmapped_area = arch_get_unmapped_area，对新式布局则是 mm->get_unmapped_area = arch_get_unmapped_area_topdown。另外，我们知道 file_operations 结构体中有一个 get_unmapped_area 方法：

```
<include/linux/fs.h>
-----
struct file_operations {
    ...
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long,
                                       unsigned long);
    ...
};
```

mm->get_unmapped_area() 和 filp->get_unmapped_area() 的主要作用都是在用户进程的虚拟地址空间中分配空闲的内存区域，在 get_unmapped_area 函数的内部，则通过 mm->get_unmapped_area 或者 filp->get_unmapped_area 来实现空闲虚拟地址的分配，这种选择基于以下代码：

```
<mm/mmap.c>
-----
unsigned long
get_unmapped_area(struct file *file, unsigned long addr, unsigned long len,
                  unsigned long pgoff, unsigned long flags)
{
    unsigned long (*get_area)(struct file *, unsigned long,
                              unsigned long, unsigned long, unsigned long);
    ...
    get_area = current->mm->get_unmapped_area;
    if (file && file->f_op && file->f_op->get_unmapped_area)
        get_area = file->f_op->get_unmapped_area;
    addr = get_area(file, addr, len, pgoff, flags);
    ...
}
```

因此 get_unmapped_area 所作的选择是，如果驱动程序在其 file_operations 对象中没有定义 get_unmapped_area 方法，即 file->f_op->get_unmapped_area 为空，那么函数将利用当前进程 mm 对象中的 get_unmapped_area 函数来分配空闲的虚拟地址空间，否则将使用 file->f_op->get_unmapped_area，现实中很少有驱动程序需要在自己的 file_operations 对象中实现 get_unmapped_area 方法，所以我们的讨论按照内核提供的标准分配函数进行。

对于传统布局而言，内核提供的分配 MMAP 区域中空闲虚拟地址空间的标准函数是 arch_get_unmapped_area：

```
<mm/mmap.c>
-----
unsigned long
```

```

arch_get_unmapped_area(struct file *filp, unsigned long addr,
                      unsigned long len, unsigned long pgoff, unsigned long flags)
{
    struct mm_struct *mm = current->mm;
    struct vm_area_struct *vma;
    unsigned long start_addr;

    if (len > TASK_SIZE)
        return -ENOMEM;
    if (flags & MAP_FIXED)
        return addr;
    if (addr) {
        addr = PAGE_ALIGN(addr);
        vma = find_vma(mm, addr);
        if (TASK_SIZE - len >= addr &&
            (!vma || addr + len <= vma->vm_start))
            return addr;
    }
    if (len > mm->cached_hole_size) {
        start_addr = addr = mm->free_area_cache;
    } else {
        start_addr = addr = TASK_UNMAPPED_BASE;
        mm->cached_hole_size = 0;
    }
    full_search:
    for (vma = find_vma(mm, addr); ; vma = vma->vm_next) {
        /* At this point: (!vma || addr < vma->vm_end). */
        if (TASK_SIZE - len < addr) {
            /*
             * Start a new search - just in case we missed
             * some holes.
             */
            if (start_addr != TASK_UNMAPPED_BASE) {
                addr = TASK_UNMAPPED_BASE;
                start_addr = addr;
                mm->cached_hole_size = 0;
                goto full_search;
            }
            return -ENOMEM;
        }
        if (!vma || addr + len <= vma->vm_start) {
            /*
             * Remember the place where we stopped the search:
             */
            mm->free_area_cache = addr + len;
            return addr;
        }
    }
}

```

```

    }
    if (addr + mm->cached_hole_size < vma->vm_start)
        mm->cached_hole_size = vma->vm_start - addr;
    addr = vma->vm_end;
}
}

```

如果应用程序指定的待映射区域的长度大于 `TASK_SIZE`，函数将把一个错误码-`ENOMEM` 返回给用户进程，告知用户进程目前空闲的虚拟地址空间不足以满足本次映射需求。如果用户进程指定了 `MAP_FIXED` 标志，表明映射将从 `addr` 参数指定的起始地址处开始，因此这种情况函数将直接返回 `addr`。接下来函数检查调用者有没有指定要优先映射的虚拟地址，如果有，内核将检查 `addr` 和 `len` 所确定的待映射的虚拟地址空间是否与已经被映射的虚拟地址空间重叠，如果不重叠将直接返回 `addr`。如果调用者没有指定一个需要优先映射的地址（这种情况下应用程序在调用 `mmap` 函数时传递的参数 `start` 为 `NULL`），那么内核必须遍历用户进程中所有可用区域，设法找到一个大小合适的空闲区域，通常情况下，应用程序在调用 `mmap` 时都是将 `start` 参数设定为 `NULL`，也就是让内核自己在当前进程用户空间的 `MMAP` 区域去找一块待映射区域。

无论如何，现在通过 `do_mmap_pgoff` 中的 `get_unmapped_area` 函数调用在 `MMAP` 区域获得了一个空闲的尚未被映射的 `vm_area_struct` 对象。

前面花了一些篇幅讨论了 `get_unmapped_area` 函数，现在继续回过头来看看 `do_mmap_pgoff` 函数的后续部分。在后续部分，函数除了做一些权能，标志位的检测之外，一个关键的调用是 `mmap_region` 函数：

```

<mm/mmap.c>
-----
unsigned long do_mmap_pgoff(struct file *file, unsigned long addr,
                           unsigned long len, unsigned long prot,
                           unsigned long flags, unsigned long pgoff)
{
    ...
    return mmap_region(file, addr, len, flags, vm_flags, pgoff);
}

```

实际的映射工作在 `mmap_region` 函数中完成（显然需要设备驱动程序中实现的 `mmap` 方法的配合）。当然出于很多方面的考量，`mmap_region` 函数一如既往地继承了 Linux 代码特有的稳健但冗长的风格。为了窥探内幕的便捷，此处还是给出 `mmap_region` 函数的主体脉络：

```

<mm/mmap.c>
-----
unsigned long mmap_region(struct file *file, unsigned long addr,
                         unsigned long len, unsigned long flags,
                         unsigned int vm_flags, unsigned long pgoff)
{

```

```

...
vma = kmem_cache_zalloc(vm_area_cachep, GFP_KERNEL);
if (!vma) {
    error = -ENOMEM;
    goto unacct_error;
}
vma->vm_mm = mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = vm_flags;
vma->vm_page_prot = vm_get_page_prot(vm_flags);
vma->vm_pgoff = pgoff;
...
if (file) {
    ...
    vma->vm_file = file;
    get_file(file);
    error = file->f_op->mmap(file, vma);
    ...
} else if (vm_flags & VM_SHARED) {
    error = shmem_zero_setup(vma);
    if (error)
        goto free_vma;
}
...
}

```

当该函数被调用时，参数 `addr` 已经指向了一块空闲的待映射的 `MMAP` 区域中的起始地址，所以函数会首先利用 `kmem_cache_zalloc` 分配出一个 `struct vm_area_struct` 实例对象，然后对其初始化。

接下来的重点由 `if...else if...` 领衔，不过 `if` 里面的代码才是我们关注的重点，至于 `else if`，我们列出这个条件是想让读者知道 `mmap` 除了映射设备内存，还有一些其他用途，只不过不是本书的重点，所以就略过不提了。

`if` 语句的核心很简单，`error = file->f_op->mmap(file, vma)`。是的，调用到了驱动程序实现的 `mmap` 方法。再回顾一下 `file_operations` 中 `mmap` 方法的原型：

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

很吻合，不是吗？

本章行文至此，我们知道了内存映射所要讨论的主要内容以及 `mmap` 机制在系统中的来龙去脉，我们看到了内核为即将进行的内存映射准备了 `struct vm_area_struct` 对象，并且调用了设备驱动程序中的 `mmap` 方法，内核的任务到此要告一段落，接下来应该是讨论设备驱

动程序如何实现 mmap 的时刻了。

10.2.4 驱动程序中 mmap 方法的实现

设备驱动程序中 mmap 方法的主要功能是将内核提供的用户进程空间中来自 MMAP 区域的一段内存(内核将这段区域以 struct vm_area_struct 对象作为参数的方式告诉设备驱动程序)映射到设备内存上。正如读者所猜想的, 驱动程序需要通过配置相对应的页目录表项的方式来完成。在继续下面的讨论前, 先来看一张整个 mmap 机制的流程图(图 10-3), 以了解目前讨论的进度:

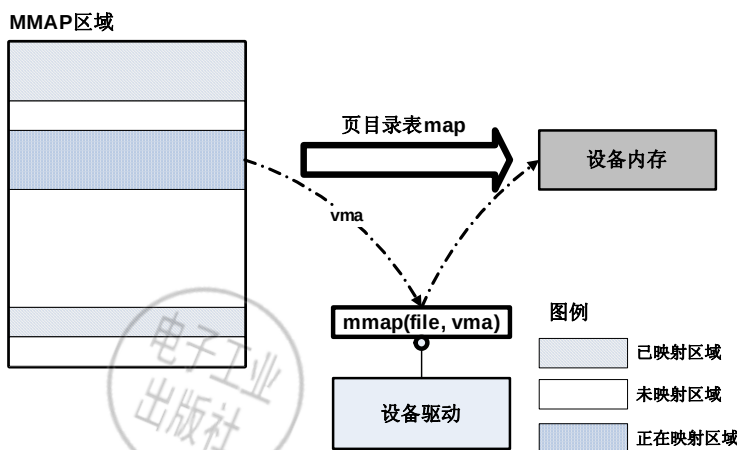


图 10-3 mmap 内存映射机制流程

到目前为止, 内核为我们在 MMAP 区域分配了一个空闲的 vma 对象, 然后调用 file 对应的设备驱动中的 mmap 方法, 驱动需要在它的 mmap 方法的实现里将 vma 对象代表的用户空间地址映射到设备内存中。

驱动程序需要在自己的 mmap 方法的实现代码中完成页目录表项的配置以便将 vma 对象表示的虚拟地址映射到对应的物理内存上, 注意这里说的是物理内存, 虽然本章内存映射的主旨是建立用户空间虚拟地址到设备内存的映射, 但是系统 RAM 也是一种物理设备, 因此物理内存的提法在更广的范围内涵盖了 mmap 机制的功能。

驱动程序当然可以直接使用页目录表项的一些操作函数来建立这种映射, 这从原理上来讲非常简单, 但事实上 Linux 系统下凡是涉及内存的操作一般都不是孤立的, 需要考虑到与其他模块之间的关联, 所以这种页目录表的操作已不再如想象中的那般单纯。幸运的是, Linux 内核为方便设备驱动程序员提供了一些接口函数供其使用。了解这些接口函数的实现机制, 有助于设备驱动正确建立所要求的内存映射关系, 另一方面, 对这些函数的理解也可以让读者熟悉如何通过页目录表项的操作函数建立内存映射。

下面开始讨论 Linux 内核中提供的操作页目录表项以建立页面映射的一些接口函数:

○ remap_pfn_range

该函数的原型为：

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long addr,
                   unsigned long pfn, unsigned long size, pgprot_t prot);
```

这个函数可以用来将参数 `addr` 起始的大小为 `size` 的虚拟地址空间映射到 `pfn` 表示的一组连续的物理页面上，`pfn` 是页框号（page frame number），在页面大小为 4 KB 的系统中，一个物理地址右移 12 位即可得到该物理地址对应的页框号。简言之，函数为 `[addr,addr+size]` 范围的虚拟地址建立页目录表项，将其映射到以 `pfn` 开始的物理页面上。

通常，将用户空间的地址通过 `remap_pfn_range` 映射到设备内存上，尤其是设备的寄存器所在的地址空间，都不希望 `cache` 机制发挥作用，驱动程序可以通过最后一个参数 `prot` 来影响页表项中属性位的建立，比如使用 `pgprot_noncached()`。

`remap_pfn_range` 函数的实现为：

<mm/memory.c>

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long addr,
                   unsigned long pfn, unsigned long size, pgprot_t prot)
{
    pgd_t *pgd;
    unsigned long next;
    unsigned long end = addr + PAGE_ALIGN(size);
    struct mm_struct *mm = vma->vm_mm;
    int err;

    if (addr == vma->vm_start && end == vma->vm_end) {
        vma->vm_pgoff = pfn;
        vma->vm_flags |= VM_PFN_AT_MMAP;
    } else if (is_cow_mapping(vma->vm_flags))
        return -EINVAL;

    vma->vm_flags |= VM_IO | VM_RESERVED | VM_PFNMAP;

    err = track_pfn_vma_new(vma, &prot, pfn, PAGE_ALIGN(size));
    if (err) {
        vma->vm_flags &= ~(VM_IO | VM_RESERVED | VM_PFNMAP);
        vma->vm_flags &= ~VM_PFN_AT_MMAP;
        return -EINVAL;
    }

    BUG_ON(addr >= end);
    pfn -= addr >> PAGE_SHIFT;
    pgd = pgd_offset(mm, addr);
```



```

flush_cache_range(vma, addr, end);
do {
    next = pgd_addr_end(addr, end);
    err = remap_pud_range(mm, pgd, addr, next,
                          pfn + (addr >> PAGE_SHIFT), prot);
    if (err)
        break;
} while (pgd++, addr = next, addr != end);

if (err)
    untrack_pfn_vma(vma, pfn, PAGE_ALIGN(size));
return err;
}

```

函数首先将 `addr+size` 调整到下一个页面的边界处，因为内存映射的最小单位是页。函数对 COW（copy-on-write）的映射处理比较谨慎，不过这种情况不是驱动程序员关注的重点。绝大多数情况下，`if (addr == vma->vm_start && end == vma->vm_end)` 条件满足，接下来的核心是操作页表建立虚拟内存到物理内存之间的映射。

`pgd_offset` 函数用来获得某一虚拟地址在页目录表中的对应单元的地址 `pgd`：

```
pgd = pgd_offset(mm, addr);
```

`flush_cache_range` 函数是体系架构相关的函数，用来将 `(addr,end)` 地址范围对应的 `cache` 内容同步到主存中。

`do while` 循环用来在页目录中建立对应的映射页表项，因为 MMU 也是一个体系相关的概念，此处的讨论限定在 32 位 x86 的两级映射，即第一级是页目录表（`pgd`），第二级是页表（`pte`）¹，页大小为 4 KB。

页目录表中的每一项 `entry` 可以映射 4 MB 的地址空间，总共有 1024 个 `entry`，所以可以映射 $1024 \times 4 \text{ MB} = 4 \text{ GB}$ 的虚拟地址空间。`next = pgd_addr_end(addr, end)` 用来获取 `addr` 对应页表项的下一个 `entry` 对应的虚拟起始地址，所以如果 `end - addr` 的值不超过 4 MB，那么 `next` 的值和 `end` 的值是相等的，如果超过 4 MB 但是小于 8 MB，那么基本上 `next=addr+4 MB`，依此类推。换句话说，如果映射一个不超过 4 MB 的虚拟地址空间，1 次 `do while` 循环后映射就完成了。

`remap_pud_range` 用来做实际的页目录表项的操作，其实现如下：

```
<mm/memory.c>
```

```

static inline int remap_pud_range(struct mm_struct *mm, pgd_t *pgd,
                                unsigned long addr, unsigned long end,

```

¹ 虽然 Linux 内核统一使用四级映射机制，对于 x86 平台传统的二级映射，内核虚拟了 `pud` 与 `pmd`。

```

        unsigned long pfn, pgprot_t prot)
{
    pud_t *pud;
    unsigned long next;

    pfn = addr >> PAGE_SHIFT;
    pud = pud_alloc(mm, pgd, addr);
    if (!pud)
        return -ENOMEM;
    do {
        next = pud_addr_end(addr, end);
        if (remap_pmd_range(mm, pud, addr, next,
            pfn + (addr >> PAGE_SHIFT), prot))
            return -ENOMEM;
    } while (pud++, addr = next, addr != end);
    return 0;
}

```

Linux 内核采用统一的四级映射 pgd、pud、pmd 和 pte。对于 32 位 x86 架构而言，只有经典的二级映射机制，因此内核通过虚拟的 pud 和 pmd 来统一 x86 的这种二级映射。简而言之，在这种硬件机制只有两级映射的情况下，pud=pmd=pgd。

图 10-4 显示了一个 32 位虚拟地址在做物理地址映射时的构成，前 10 位用做 pgd 的索引，总共能索引 $2^{10}=1024$ 个页目录项，中间的 10 位用做页表项的索引，同样可以索引 1024 个项，最后的 12 位作为映射到的物理页面中的偏移地址。

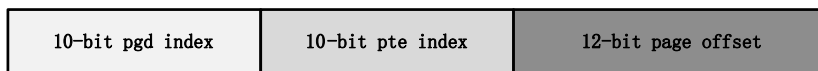


图 10-4 32 位虚拟地址构成

如果抛开内核对 pud 与 pmd 的模拟，那么在 remap_pud_range 函数的调用链中，最终的页表建立实际发生在 remap_pmd_range 中：

<mm/memory.c>

```

static inline int remap_pmd_range(struct mm_struct *mm, pud_t *pud,
    unsigned long addr, unsigned long end,
    unsigned long pfn, pgprot_t prot)
{
    pmd_t *pmd;
    unsigned long next;

    pfn = addr >> PAGE_SHIFT;
    pmd = pmd_alloc(mm, pud, addr);
    if (!pmd)
        return -ENOMEM;
}

```

```

do {
    next = pmd_addr_end(addr, end);
    if (remap_pte_range(mm, pmd, addr, next,
        pfn + (addr >> PAGE_SHIFT), prot))
        return -ENOMEM;
    } while (pmd++, addr = next, addr != end);
return 0;
}

```

函数中的 `remap_pte_range` 将建立对应的页表项，对于传统的 32 位 x86 的两级映射，在调用该函数时 `pmd=pgd`，`remap_pte_range` 函数将首先通过 `__pte_alloc` 分配一个物理页面作为第二级映射的页表：

<mm/memory.c>

```

int __pte_alloc(struct mm_struct *mm, pmd_t *pmd, unsigned long address)
{
    ...
    pgtable_t new = pte_alloc_one(mm, address);
    ...
    spin_lock(&mm->page_table_lock);
    if (!pmd_present(*pmd)) { /* Has another populated it ? */
        mm->nr_ptes++;
        pmd_populate(mm, pmd, new);
        new = NULL;
    }
    spin_unlock(&mm->page_table_lock);
    ...
}

```

在得到新页表的物理地址 `new` 之后，`__pte_alloc` 调用 `pmd_populate(mm, pmd, new)` 将新的页表物理地址写到 `pgd` 中。之后根据要映射到的物理地址 `address` 计算出它在新页表中的对应项，然后将 `address` 地址写到该项，对应的代码在：

<mm/memory.c>

```

static int remap_pte_range(struct mm_struct *mm, pmd_t *pmd,
    unsigned long addr, unsigned long end,
    unsigned long pfn, pgprot_t prot)
{
    ...
    pte = pte_alloc_map_lock(mm, pmd, addr, &ptl);
    ...
    do {
        set_pte_at(mm, addr, pte, pte_mkspecial(pfn_pte(pfn, prot)));
        pfn++;
    } while (pte++, addr += PAGE_SIZE, addr != end);
    ...
}

```

```
}

```

上面的代码在将实际映射到的物理页面地址写到对应 `pte` 中时, `pfn` 的操作是 `pfn++`, 这意味着 `remap_pfn_range` 函数将把 `vma` 对象表征的一段虚拟地址映射到从 `pfn` 开始的一段连续的物理页面中, 当然这里的前提是 `remap_pfn_range` 需要映射到的空间范围多于一个物理页。这里只是从全局的角度讨论了 `remap_pfn_range` 函数的实现机制, 更多关于实际映射过程的建立细节需要读者知晓 x86 的二级页表的相关知识, 比如页目录项和页表项的构成等。

现在回顾一下 `remap_pfn_range` 函数的总体流程。函数首先根据需要映射的虚拟地址块的首地址 (由函数的参数 `addr` 表示) 的前 10 位得到第一级映射在页目录表中的 `entry` (页目录表的每一表项映射空间的大小为 4 MB), 接着分配一块物理页面作为新的二级页表, 并将该页表的物理地址填入前面的 `entry` 中, 最后通过虚拟地址首地址的中间 10 位来确定对应的 4 KB 大小的映射在新页表中的 `entry` (二级页表的每个 `entry` 映射一个 4 KB 大小的物理页面), 找到之后将要映射的物理页的起始地址 (由 `remap_pfn_range` 函数的第三个参数 `pfn` 提供) 放到该 `entry` 中。需要深入钻研的读者也许要阅读对应处理器架构的 MMU 单元部分, 比如 x86 或者 ARM, 以了解这些映射的技术细节。鉴于本书的主题侧重在内核为设备驱动程序提供的各种内核机制的剖析上, 所以对这种映射的技术细节的讨论只会到目前的层面上, 此时读者应该已经了解了用户空间的地址如何映射到实际的物理内存。

10.2.5 mmap 使用范例

本节将用一定的篇幅讨论一个实际的设备驱动程序在其 `mmap` 方法的实现中如何使用 `remap_pfn_range` 函数来映射设备内存。这个例子源自 Linux 内核源码中的 `drivers/media/video/cpia2`, 这个驱动程序实现了自己的 `mmap` 方法来将用户空间地址映射到其设备内存 (Frame Buffer) 上:

```
<drivers/media/video/cpia2/cpia2_v4l.c>
```

```
static int cpia2_mmap(struct file *file, struct vm_area_struct *area)
{
    struct camera_data *cam = video_drvdata(file);
    int retval;
    ...
    retval = cpia2_remap_buffer(cam, area);
    ...
    return retval;
}
```

```
<drivers/media/video/cpia2/cpia2_core.c>
```

```
int cpia2_remap_buffer(struct camera_data *cam, struct vm_area_struct *vma)
{
    const char *adr = (const char *)vma->vm_start;
    unsigned long size = vma->vm_end-vma->vm_start;
```

```

unsigned long start_offset = vma->vm_pgoft << PAGE_SHIFT;
unsigned long start = (unsigned long) adr;
unsigned long page, pos;
...
pos = ((unsigned long) (cam->frame_buffer)) + start_offset;
while (size > 0) {
    page = kvirt_to_pa(pos);
    if (remap_pfn_range(vma, start, page >> PAGE_SHIFT, PAGE_SIZE, PAGE_SHARED))
        return -EAGAIN;
    start += PAGE_SIZE;
    pos += PAGE_SIZE;
    if (size > PAGE_SIZE)
        size -= PAGE_SIZE;
    else
        size = 0;
}

cam->mmapped = true;
return 0;
}

```

经过前面几节的讨论，cpia2_mmap 函数的参数我们已经非常熟悉了。设备的内存由 cam->frame_buffer 指定，它通过 cpia2_allocate_buffers 函数来为之分配，为了 cpia2_mmap 函数讨论的完整性，等下再讨论 cpia2_allocate_buffers 函数如何分配设备内存。现在我们有设备内存，它保存在 cam->frame_buffer 中，接下来的核心是在 while 循环中，它首先通过 kvirt_to_pa(pos) 得到设备内存所映射到的物理内存地址，然后通过 remap_pfn_range 将用户进程空间的虚拟地址映射到该物理内存页面上，这里每次调用 remap_pfn_range 映射一个物理页面，直到所有的设备内存页面全部被映射完毕。在 cpia2_mmap 成功执行后，用户空间将可以通过 mmap 函数返回的地址直接访问 cam 设备的 Frame Buffer。

现在看一下 allocate_frame_buf 函数，看看驱动程序如何分配设备内存：

<drivers/media/video/cpia2/cpia2_core.c>

```

int cpia2_allocate_buffers(struct camera_data *cam)
{
    int i;
    ...
    if(!cam->frame_buffer) {
        cam->frame_buffer = vmalloc(cam->frame_size*cam->num_frames);
        if (!cam->frame_buffer) {
            ERR("couldn't vmalloc frame buffer data area\n");
            kfree(cam->buffers);
            cam->buffers = NULL;
            return -ENOMEM;
        }
    }
}

```

```

    }
    ...
    return 0;
}

```

函数需要通过 `rvmalloc` 来分配 `cam` 设备的 Frame Buffer，其中 `cam->frame_size` 表示单个帧缓冲区大小，`cam->num_frames` 是 `cam` 设备拥有的 Frame Buffer 的数量。`rvmalloc` 是 `cpia2` 驱动程序自己实现的一个内存分配函数，其核心是调用 `vmalloc_32` 来分配内存：

```

<drivers/media/video/cpia2/cpia2_core.c>
-----
static void *rvmalloc(unsigned long size)
{
    void *mem;
    unsigned long adr;

    /* Round it off to PAGE_SIZE */
    size = PAGE_ALIGN(size);

    mem = vmalloc_32(size);
    if (!mem)
        return NULL;

    memset(mem, 0, size); /* Clear the ram out, no junk to the user */
    adr = (unsigned long) mem;

    while ((long)size > 0) {
        SetPageReserved(vmalloc_to_page((void *)adr));
        adr += PAGE_SIZE;
        size -= PAGE_SIZE;
    }
    return mem;
}

```

`vmalloc_32` 分配一段连续的虚拟地址，然后通过内核的伙伴系统分配相应的物理页面并提交到前面的虚拟地址上。接下来的 `while` 循环中，`vmalloc_to_page` 函数根据虚拟地址 `adr` 去查找页目录表项，得到该虚拟地址映射到的物理页面所在的 `struct page` 对象，然后用 `SetPageReserved` 将这些为设备帧缓存分配的物理页面标识为 `Reserved`，`SetPageReserved` 用来设置内核虚拟地址所对应的 `struct page` 对象的 `PG_reserved` 属性，按照内核源码注释，被 `reserved` 的都是一些特殊的页面，这些页面最显著的特性是脱离了内核 VM 组件的管理，因此内核源码说用 `SetPageReserved` 设置的页面可以确保不会被交换出去。驱动程序使用 `remap_pfn_range` 将内核虚拟地址重新映射到用户空间，本质上是将被映射的内核虚拟地址视做设备内存，比如 PCI 设备的 MM 空间和 IO 空间，这些空间在内核中都没有对应的 `struct page` 对象，`SetPageReserved` 从某种意义上试图将系统物理页面模拟成设备内存。关于这方

面的内容，在内核 VM 组件的开发演进过程中，意见其实并不统一，读者可以阅读 <http://lwn.net/Articles/161204/>，对应 SetPageReserved 的操作是 ClearPageReserved。vmalloc 返回的是内核虚拟地址空间的 vmalloc 区中的某一虚拟地址，该段虚拟地址将被 cam 设备用做 Frame Buffer。

cpia2_mmap 函数在调用 remap_pfn_range 将用户进程的地址空间映射到设备帧缓存之前，需要通过调用 kvirt_to_pa 来获得 vmalloc 区中虚拟地址所映射到的物理地址 page，这样调用 remap_pfn_range 函数的第三个参数页帧号就可以通过 `page >> PAGE_SHIFT` 获得。图 10-5 展示了这一过程：

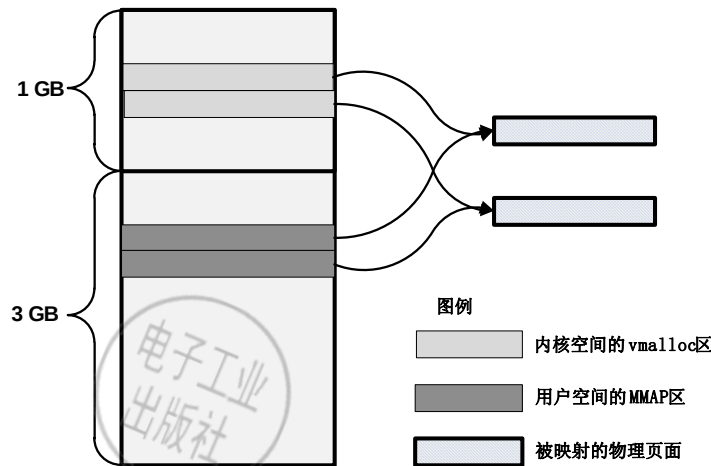


图 10-5 cpia2 的 remap_pfn_range 应用示例

图中驱动程序首先通过 vmalloc 函数在内核空间的 vmalloc 区为设备的帧缓存分配了空间，该空间将映射到两个物理页面上（它们可能连续也可能不连续），然后驱动程序调用 remap_pfn_range 将用户空间 MMAP 区的一段虚拟内存也映射到这两个物理页面上。这个例子也展示了内核空间的虚拟地址如何被用户空间地址所映射：在本例中，内核空间虚拟地址由 vmalloc 分配，然后通过 mmap 机制将 vmalloc 得到的虚拟地址重新映射回用户空间，更确切地说，通过 kvirt_to_pa 函数获得 vmalloc 虚拟地址映射到的物理页面，然后再建立用户空间对应的页表项将属于它的一段虚拟地址空间映射到同样的物理页面上。如果 cpia2 设备的 Frame Buffer 由 kmalloc 这样的函数分配²，因为这种情形下获得的物理页面是连续的，所以一个单次的 remap_pfn_range 就可以完成映射的任务。

把上述过程总结一下就是，驱动通过 vmalloc 函数分配设备内存的虚拟地址，然后将这段虚拟地址映射的物理页面（可能是不连续的）映射到用户进程的地址空间，如此应用程序将可以直接使用这些给设备帧缓存使用的物理页面，而无须经内核周转。现实中设备缓存

² 因为映射总是以页为单位，所以这种情况下最好是用 __get_free_pages 这样的页面级分配器。

的典型用法是在设备内存与设备缓存间建立一个 DMA 通道，这样设备的数据将以极高的性能传递到驱动程序所管理的设备缓存中，而后者可以被用户程序直接使用，这正是 mmap 机制要完成的功能所在。关于 DMA 通道映射将是本章下半部分的话题。

在这个例子中，应用程序通过 mmap 方法获得的虚拟地址所映射的目标地址依然是在系统内存中，然而现实中映射到设备内存的例子更为常见，比如在 Linux 显卡驱动程序中，X server 运行在用户空间，在其启动过程中会将显卡驱动的用户空间部分³以.so 的形式加载进来。在用户空间运行的这部分代码如果想要访问显卡的 Frame Buffer 空间或者是 MMIO 空间，就可以使用 mmap 这种方式。现在显卡设备多以 PCI 设备形式存在，它的 Frame Buffer 和 MMIO 空间就是典型的设备内存，显卡驱动程序通过 PCI 配置空间获得显卡设备 MMIO 的信息（起始地址及空间大小范围等），然后映射到用户空间虚拟地址区域。这其实是典型的用户空间驱动程序用以访问硬件设备资源的例子，如此 X server 将可以直接读写这些寄存器，而不必经由 ioctl 这样的方式。

下面是 X server lib 库中一个通过 mmap 方法将 PCI 设备地址空间映射到用户空间的函数，显卡驱动程序会调用该函数来将显卡寄存器所在的 I/O 空间映射到用户空间：

```
static int
pci_device_x86_map_range(struct pci_device *dev, struct pci_device_mapping *map)
{
    int memfd = open("/dev/mem", O_RDWR);
    int prot = PROT_READ;

    if (memfd == -1)
        return errno;
    if (map->flags & PCI_DEV_MAP_FLAG_WRITABLE)
        prot |= PROT_WRITE;

    map->memory = mmap(NULL, map->size, prot, MAP_SHARED, memfd, map->base);
    close(memfd);

    if (map->memory == MAP_FAILED)
        return errno;

    return 0;
}
```

pci_device_x86_map_range 函数使用 mmap 来映射 PCI 的总线地址到用户空间，map->base 就是 PCI 设备的起始总线地址，map->size 是要映射的地址范围。

因此，如果 remap_pfn_range 映射的地址不属于系统的 RAM 区间，比如 PCI 设备的 I/O 地

³ Linux 下的显卡驱动非常复杂，常常涵盖许多模块在内，既有用户空间，也有内核空间。

址空间等，因为这些被映射的物理地址不在内存子系统的管理范畴之列，它们没有对应的 `struct page` 对象，因而也就不存在映射到系统 RAM 时可能会遭遇到的问题，所以在这些情况下 `remap_pfn_range` 的使用方式就很直接。

为了加深读者的理解，这里再给出一个将内核空间某一物理页面映射到用户空间的例子，这里的物理页面其实可引申为设备的内存（比如某 PCI-E 显卡设备的 Frame Buffer 所在的总线地址 `0xd0000000`）。这个例子将展示用户空间如何通过 `mmap` 来映射某一段物理地址空间并对其进行操作。内核空间的物理页面通过 `alloc_pages` 获得，其对应的物理地址将用 `printk` 打印出来，这样用户空间才可以告诉 `mmap` 函数要映射到哪个物理页面上。为了使代码简洁，程序去除了对错误情况的处理。首先是内核模块代码 `mmap_demo.c`：

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

#include <linux/device.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/fcntl.h>
#include <linux/gfp.h>
#include <linux/string.h>
#include <linux/mm_types.h>
#include <linux/mm.h>
#include <linux/highmem.h>

#define KSTR_DEF    "Hello world from kernel virtual space"

static struct cdev *pcdev;
static dev_t ndev;
static struct page *pg;
static struct timer_list timer;

//定时器函数，打印出被映射的物理页面的内容
static void timer_func(unsigned long data)
{
    printk("timer_func:%s\n", (char *)data);
    timer.expires = jiffies + HZ*10;
    add_timer(&timer);
}

static int demo_open(struct inode *inode, struct file *filp)
{
    return 0;
}
```

```

static int demo_release(struct inode *inode, struct file *filp)
{
    return 0;
}

static int demo_mmap(struct file *filp, struct vm_area_struct *vma)
{
    int err = 0;
    unsigned long start = vma->vm_start;
    unsigned long size = vma->vm_end - vma->vm_start;
    //用 remap_pfn_range 将用户空间地址映射到内核空间的物理页面
    err = remap_pfn_range(vma, start, vma->vm_pgoff, size, vma->vm_page_prot);
    return err;
}

static struct file_operations mmap_fops =
{
    .owner = THIS_MODULE,
    .open = demo_open,
    .release = demo_release,
    .mmap = demo_mmap,
};

static int demo_map_init(void)
{
    int err = 0;
    char *kstr;

    //在高端物理内存区分配一个页面
    pg = alloc_pages(GFP_HIGHUSER, 0);
    //设置页面的 PG_reserved 属性, 防止映射到用户空间的页面被 swap out 出去
    SetPageReserved(pg);
    //因为物理页面来自高端内存, 所以在使用前需要调用 kmap 为该物理页面建立映射
    //关系
    kstr = (char *)kmap(pg);
    strcpy(kstr, KSTR_DEF);
    printk("kpa = 0x%X, kernel string = %s\n", page_to_phys(pg), kstr);

    pcdev = cdev_alloc();
    cdev_init(pcdev, &mmap_fops);
    alloc_chrdev_region(&ndev, 0, 1, "mmap_dev");
    printk("major = %d, minor = %d\n", MAJOR(ndev), MINOR(ndev));
    pcdev->owner = THIS_MODULE;
    cdev_add(pcdev, ndev, 1);
}

```

```

//创建定时器每隔 10 秒打印一次被映射的物理页面中的内容
init_timer(&timer);
timer.function = timer_func;
timer.data = (unsigned long)kstr;
timer.expires = jiffies + HZ*10;
add_timer(&timer);

return err;
}

static void demo_map_exit(void)
{
    del_timer_sync(&timer);
    cdev_del(pcdev);
    unregister_chrdev_region(ndevice, 1);
    kunmap(pg);
    ClearPageReserved(pg);
    __free_pages(pg, 0);
}

module_init(demo_map_init);
module_exit(demo_map_exit);

MODULE_AUTHOR("dennis chen @ AMDLinuxFGL");
MODULE_DESCRIPTION("A demo kernel module to remap a physical page to the user space");
MODULE_LICENSE("GPL");

```

内核模块使用了定时器每隔 10 秒钟打印被用户空间映射的物理页面的内容, 这样就可以验证该物理页面是否可以在用户空间被改写。

该内核模块在 2.6.39 内核版本的 Linux 系统上编译通过, 对应的 Makefile 为:

```

obj-m := mmap_demo.o
KERNELDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
    rm -f *.o *.ko *.mod.*

```

对应的应用程序为:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

```

```

#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

#define MAP_SIZE 4096
#define USTR_DEF    "String changed from the User Space"

int main(int argc, char *argv[])
{
    int fd;
    char *pdata;

    if(argc <= 1){
        printf("Usage: main devfile pamapped\n");
        return 0;
    }
    fd = open(argv[1], O_RDWR|O_NDELAY);
    if(fd >= 0){
        pdata = (char *)mmap(0, MAP_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED,
                             fd, strtoul(argv[2], 0, 16));
        printf("UserAddr = %p, Data from kernel:%s\n", pdata, pdata);
        printf("Writing a string to the kernel space...");
        strcpy(pdata, USTR_DEF);
        printf("Done\n");
        munmap(pdata, MAP_SIZE);
        close(fd);
    }
    return 0;
}

```

应用程序首先用 `mmap` 来映射内核空间的物理页面，然后读取其内容，紧接着进行改写。

首先将内核模块加入系统：

```
root@AMDLinuxFGL:/home/dennis/book/chap10/mmap# insmod mmap_demo.ko
```

`dmesg` 针对上述指令的输出为：

```

root@AMDLinuxFGL:/home/dennis/book/chap10/mmap# dmesg
[17194.802286] kpa = 0x3358E000, kernel string = Hello world from kernel virtual space
[17194.802290]major = 249, minor = 0

```

根据上述信息，要映射的物理页面起始地址为 `0x3358E000`，主次设备号分别为 `249` 和 `0`，可据此创建一个设备节点：

```
root@AMDLinuxFGL:/home/dennis/book/chap10/mmap# mknod /dev/demo_map c 249 0
```

设备节点创建好之后，现在可以运行应用程序了：

```
root@AMDLinuxFGL:/home/dennis/book/chap10/app#./main /dev/demo_map 0x3358E000
```

应用程序输出信息如下：

```
UserAddr = 0xb7767000, Data from kernel:Hello world from kernel virtual space
writing a string to the kernel space...Done
```

此时再用 `dmesg` 看看内核的输出：

```
root@AMDLinuxFGL:/home/dennis/book/chap10/mmap# dmesg
[17194.802286] kpa = 0x3358E000, kernel string = Hello world from kernel virtual space
[17194.802290]major = 249, minor = 0
[17204.832010]timer_func: Hello world from kernel virtual space
[17214.848005]timer_func: Hello world from kernel virtual space
[17234.880009]timer_func: String changed from the User Space
```

最后一条信息表明用户空间已经成功改写了被映射的物理页面中的内容。通过这个例子，我们看到应用程序只需通过 `mmap` 一次系统调用，就可以直接操作物理内存（或是设备内存），因此如果需要在用户空间和内核空间之间传输数据，相对于 `copy_from_user` 以及 `copy_to_user`，使用 `mmap` 的优势是不言而喻的。

○ `io_remap_pfn_range`

`io_remap_pfn_range` 是内核提供的另外一个用来映射用户空间虚拟地址的函数，从函数名称上来看，它与 `remap_pfn_range` 的区别是 `io_remap_pfn_range` 用来将用户地址映射到设备的 I/O 空间，而 `remap_pfn_range` 则是将用户地址映射到主存 RAM 中。然而这只是函数名称上的区分，在 Linux 内核的实际代码中，对于绝大多数常见的体系架构，`io_remap_pfn_range` 与 `remap_pfn_range` 是完全等价的（因为映射的核心是 MMU，而对 MMU 来说，无须区分映射的目标地址类型）。比如对于 x86 平台：

```
<arch/x86/include/asm/pgtable.h>
-----
#define io_remap_pfn_range(vma, vaddr, pfn, size, prot)    \
    remap_pfn_range(vma, vaddr, pfn, size, prot)
```

对于 ARM 平台：

```
<arch/arm/include/asm/pgtable.h>
-----
#define io_remap_pfn_range(vma,from,pfn,size,prot) \
    remap_pfn_range(vma, from, pfn, size, prot)
```

但是从代码易读性的角度，建议如果映射的目标地址是在 RAM 或者是，比如 PCI 设备的 MM 空间中，使用 `remap_pfn_range` 函数，如果映射的目的地址是在设备 I/O 空间，则应该使用 `io_remap_pfn_range` 函数。

本书第三章讨论过 `ioremap` 函数，它与 `io_remap_pfn_range` 函数的区别是，前者用来将设备的 I/O 空间映射到内核空间，而后者则是将设备的 I/O 空间映射到用户空间。

10.2.6 munmap

`munmap` 做的事情与 `mmap` 恰好相反，不过严格意义上这个话题并不属于设备驱动程序的范畴。换句话说，设备驱动程序无须为 `munmap` 行为做特定的工作，内核会根据现有的映射信息拆除在 `mmap` 中建立的页表项，反映到驱动程序的 `file_operations` 对象上，里面只有 `mmap` 方法而无对应的 `munmap`。

对于应用程序而言，这个 API 函数的原型是：

```
int munmap(void *start, size_t length);
```

参数 `start` 应是 `mmap` 函数返回的地址，表示要拆除映射的虚拟地址段的起始地址。`length` 则应与调用 `mmap` 函数时保持一致。

在内核中，它对应的系统调用为：

```
<mm/mmap.c>
-----
SYSCALL_DEFINE2(munmap, unsigned long, addr, size_t, len)
{
    int ret;
    struct mm_struct *mm = current->mm;
    profile_munmap(addr);
    down_write(&mm->mmap_sem);
    ret = do_munmap(mm, addr, len);
    up_write(&mm->mmap_sem);
    return ret;
}
```

其中的核心调用是 `do_munmap` 函数，具体的技术细节不再深入讨论。下面给出撤销映射页表项相关的调用链：`do_munmap()`→`unmap_region()`→`unmap_region()`→`free_pgtables()`→`free_pgd_range()`→`free_pud_range()`→`free_pmd_range()`→`free_pte_range()`。

总体思想是通过虚拟地址找到对应的页目录项和页表项，然后清除这些页目录表项中的相应内容，从而撤销掉 `mmap` 建立的虚拟地址到物理页面的映射关系。

10.3 DMA

直接内存访问 DMA (Direct Memory Access) 用来在设备内存与主存 RAM 之间直接进行数据交换, 因为这个过程无须 CPU 的干预, 所以对于与系统有大量数据交换的设备而言, 如果能充分利用 DMA 特性, 可以大大提高系统性能。这种情况对设备驱动程序提出了新的要求, 即必须能很好地支持设备的 DMA 操作。

本节将首先讨论 Linux 内核中的 DMA 层, 然后再讨论 DMA 操作的核心即 DMA 内存映射, 包括一致性 DMA 映射、流式 DMA 映射和分散/聚集映射。

10.3.1 内核中的 DMA 层

为了方便设备驱动程序的开发, 内核为设备驱动程序提供了统一的 DMA 接口, 这些接口屏蔽了不同平台之间的差异, 因此使用内核 DMA 层提供的接口的设备驱动程序将具有更好的可移植性。当然不同平台的 Linux 内核需要提供特定平台的 DMA 映射操作代码, 比如 x86 或者 ARM 平台, 不过这不是设备驱动程序员要做的工作。

在继续下面的讨论之前, 先给出 Linux 内核 DMA 层的一个大体框架 (图 10-6), 以帮助读者建立一个感性的认识。

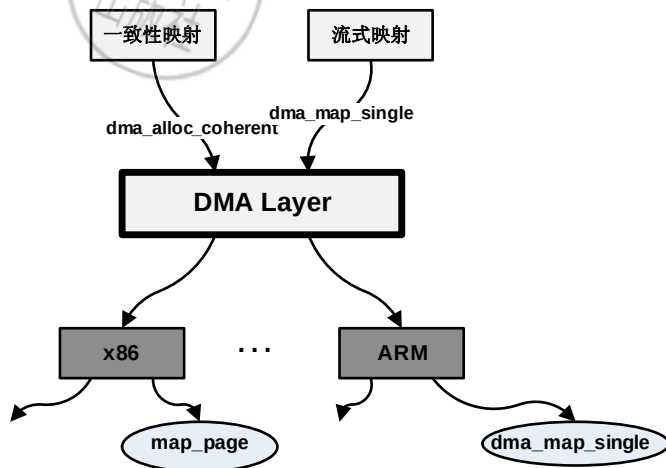


图 10-6 Linux 内核 DMA Layer 框架

从图中可以看到, Linux 内核中的 DMA 层为设备驱动程序提供标准的 DMA 映射接口, 例如一致性映射类型的 `dma_alloc_coherent` 和流式映射类型的 `dma_map_single`。在 DMA 层的下方, 不同平台的 Linux 内核代码实现平台相关的 DMA 映射操作, 如此, 通过 DMA 层 Linux 系统为设备驱动程序屏蔽了平台的差异。

按照 Linux 内核对 DMA 层的架构设计, 各平台 DMA 缓冲区映射之间的差异应该由内核定

义的一个 DMA 映射操作集 struct dma_map_ops 对象来完成，换句话说不同平台应该提供各自的 struct dma_map_ops 对象来实现相应的 DMA 映射。但不幸的是，笔者写这本书时偏偏是以 x86 和 ARM 架构为重点，在对比这两个平台的实现代码时，发现 ARM 架构上的代码对 struct dma_map_ops 对象的支持很不给力，它常常绕开 dma_map_ops 对象而自成一派，使得 Linux 内核中 DMA 层的实现代码风格不太统一。作者对此的理解是，x86 是一种标准的硬件平台，内核开发者很容易得到这种平台的测试环境，因此向内核社区提交 x86 平台的代码相对比较容易。而 ARM 则主要以一种 SoC 的形态存在，其硬件平台环境在现实中往往千差万别，鉴于这种情况，主流的 Linux 内核源码树对 ARM 的支持更多的是在处理器的核心层面，其象征意义往往要大于实际意义，这也是为什么主流的 Linux 内核必须加入特定的基于 ARM 的某一 SoC 平台的 BSP 代码才能构成完整内核的原因，鉴于本书对于 ARM 平台的代码分析都出自标准的 Linux 内核源码树（源自 www.kernel.org），因此对于在 ARM 下从事嵌入式 Linux 开发的那些读者来说，很可能会发现本书中摘录的 ARM 平台的代码与自己手边针对某一 ARM 开发板的 Linux 代码间会有些偏差，但是作者相信这细微偏差的背后所蕴含的原理一定是一样的。

最后，作为对一个 struct dma_map_ops 对象的直观感受，我们看看 Linux 为它定义的数据结构：

```
<include/linux/dma-mapping.h>
```

```
-----
struct dma_map_ops {
    void* (*alloc_coherent)(struct device *dev, size_t size,
                           dma_addr_t *dma_handle, gfp_t gfp);
    void (*free_coherent)(struct device *dev, size_t size,
                          void *vaddr, dma_addr_t dma_handle);
    dma_addr_t (*map_page)(struct device *dev, struct page *page,
                           unsigned long offset, size_t size, enum dma_data_direction dir,
                           struct dma_attrs *attrs);
    void (*unmap_page)(struct device *dev, dma_addr_t dma_handle,
                       size_t size, enum dma_data_direction dir, struct dma_attrs *attrs);
    int (*map_sg)(struct device *dev, struct scatterlist *sg, int nents,
                  enum dma_data_direction dir, struct dma_attrs *attrs);
    void (*unmap_sg)(struct device *dev, struct scatterlist *sg, int nents,
                     enum dma_data_direction dir, struct dma_attrs *attrs);
    void (*sync_single_for_cpu)(struct device *dev, dma_addr_t dma_handle, size_t size,
                                enum dma_data_direction dir);
    void (*sync_single_for_device)(struct device *dev, dma_addr_t dma_handle, size_t size,
                                   enum dma_data_direction dir);
    void (*sync_sg_for_cpu)(struct device *dev, struct scatterlist *sg, int nents,
                             enum dma_data_direction dir);
    void (*sync_sg_for_device)(struct device *dev, struct scatterlist *sg, int nents,
                                enum dma_data_direction dir);
    int (*mapping_error)(struct device *dev, dma_addr_t dma_addr);
};
```



```
int (*dma_supported)(struct device *dev, u64 mask);
int (*set_dma_mask)(struct device *dev, u64 mask);
int is_phys;
};
```

struct dma_map_ops 中定义的 DMA 操作的方法涵盖了本章所有内容，相关细节将在本章后续内容中慢慢讨论。

10.3.2 物理地址与总线地址

出于下面讨论的需要，这里需要了解一下在设备驱动程序中经常出现的总线地址的概念，作为对比在此给出另一个地址类型 CPU 的物理地址。

所谓 CPU 物理地址，即 CPU 的地址信号线上产生的地址。在有 MMU 的系统中，CPU 执行的程序指令中的地址是虚拟地址，经过 MMU 的转换，虚拟地址转变为物理地址用来驱动 CPU 的地址信号线，或者用来访问系统主存 RAM，或者用来访问设备 I/O 空间（对于 x86 架构而言，需要专门的 I/O 指令，至于访问的是系统主存还是 I/O 空间，要由特定的控制信号线决定。而对于 ARM 架构，主存和 I/O 空间统一编址，由统一的指令访问）。

而总线地址，可以简单认为是从设备角度看到的地址，不同类型的总线具有不同类型的总线地址，目前最常见的是 PCI 总线。图 10-7 是一典型的 PC 架构的数据通道示意：

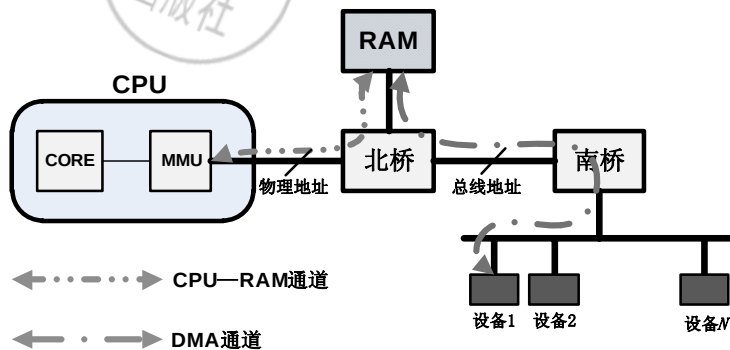


图 10-7 PC 架构的数据通道

从图中可以看到两条典型的数据交换通道：一条是从 CPU 到系统主存 RAM，CPU 核心使用的是虚拟地址，经过 MMU 转化成物理地址，用来访问 RAM；另一条是设备与主存之间的 DMA 通道，这里从设备的角度看过去，与主存进行数据传输时，使用的是总线地址。对于 x86 平台而言，从 CPU 地址总线出来的信号实际上都纳入了 PCI 总线的范畴，我们在上图中简单地将 CPU 到主存之间的地址称为物理地址，而将北桥通往南桥的地址称为总线地址。事实上南桥之下的大部分 PCI 设备都会将自己的存储空间或者映射到 CPU 的 RAM 空间或者映射到 CPU 的 I/O 空间，从这个角度，CPU 与南桥之间的地址通道就是 PCI 总线地址。

在笔者撰写本章的时候，2011 年国际消费电子展正在拉斯维加斯如火如荼地进行，在这次展会上 AMD 展示了从 2006 年收购 ATI 公司以来 fusion 理念的结晶——APU 处理器：将 GPU 和 CPU 真正地融聚到了单一芯片中。CPU 和 GPU 的融聚目前正在成为趋势，这也意味着 CPU 已经在慢慢吞噬北桥的功能，从目前的技术发展来看，个人以为也许过不了多久，南桥也将会被 CPU 所吞噬，形成 APU 或者 SoC，图 10-7 将会成为一代永恒经典。

总之，这是个跟体系架构与系统设计密切相关的概念，对于设备驱动程序而言，其实更关心的是 DMA 地址，它用来在设备与主存之间寻址，虽然它就是总线地址，但是从内核代码的角度，它被叫做 DMA 地址，与之相对应的数据结构是 `dma_addr_t`。设备驱动程序可以不必清楚物理地址与总线地址间这种形而上的区别，但是一定要明白 `dma_addr_t` 的概念，因为在实际工作中，与它打交道的机会要远远大于如何搞明白物理地址与总线地址的区别。

10.3.3 dma_set_mask

该函数用来查询设备的 DMA 寻址范围，如果设备对象 `dev` 的 DMA 操作支持参数 `mask` 指定的范围，则函数返回 0，否则返回一负的错误代码。设备对象 `dev` 在其成员 `dma_mask` 中来标识其 DMA 寻址范围。该函数在 x86 平台上的实现为：

```
<arch/x86/kernel/pci-dma.c>
```

```
int dma_set_mask(struct device *dev, u64 mask)
{
    if (!dev->dma_mask || !dma_supported(dev, mask))
        return -EIO;
    *dev->dma_mask = mask;
    return 0;
}
```

其中 `dma_supported` 的内部通过获得的设备 `dev` 上定义的 `struct dma_map_ops` 对象（该对象表示对应设备上的一个 DMA 操作集）的 `dma_supported` 成员来获得设备上 DMA 的寻址范围信息，如果设备没有实现其 `dma_supported` 方法，则内核将采用默认值：

```
<arch/x86/kernel/pci-dma.c>
```

```
int dma_supported(struct device *dev, u64 mask)
{
    ...
    struct dma_map_ops *ops = get_dma_ops(dev);
    if (ops->dma_supported)
        return ops->dma_supported(dev, mask);
    ...
}
```

因为在 Linux 设备驱动模型中，设备和驱动是各自分开进行的，所以只有设备对象 `dev` 本

身才对自己的 DMA 操作能力最为清楚，所以为了支持 `dma_set_mask` 功能，`dev` 需要提供其 `dma_ops` 成员 (`dev->archdata.dma_ops`) 上的 `dma_supported` 方法的实现。而 `dma_set_mask` 的调用者则多半是在 `dev` 的驱动 `driver` 的探测函数 `probe` 中，以下是 Linux 内核源码中网络设备驱动使用 `dma_set_mask` 来获得其驱动的设备 DMA 寻址能力的例子：

```
<drivers/net/e1000e/netdev.c>
-----
static int __devinit e1000_probe(struct pci_dev *pdev,
                                const struct pci_device_id *ent)
{
    ...
    pci_using_dac = 0;
    err = dma_set_mask(&pdev->dev, DMA_BIT_MASK(64));
    if (!err) {
        err = dma_set_coherent_mask(&pdev->dev, DMA_BIT_MASK(64));
        if (!err)
            pci_using_dac = 1;
    } else {
        err = dma_set_mask(&pdev->dev, DMA_BIT_MASK(32));
        if (err) {
            err = dma_set_coherent_mask(&pdev->dev,
                                        DMA_BIT_MASK(32));
            if (err) {
                dev_err(&pdev->dev, "No usable DMA "
                        "configuration, aborting\n");
                goto err_dma;
            }
        }
    }
    ...
}
```

10.3.4 DMA 映射

DMA 映射主要为在设备与主存之间建立 DMA 数据传输通道时，在主存中为该 DMA 通道分配内存空间的行为，该内存空间也称为 DMA 缓冲区。这个任务原本可以很简单，但是由于现代处理器 `cache` 的存在，使得事情变得有点复杂。

图 10-8 显示了在主存 RAM 和一个设备之间建立 DMA 通道时在 RAM 中建立的一个 DMA 映射。因为现代处理器为了提升系统性能在 CPU 与 RAM 之间加入了高速缓存 `cache`，所以当在 RAM 中为一个 DMA 通道建立一段缓冲区时，必须仔细考虑 RAM 与 `cache` 内容的一致性问题。比如在该图中，如果 RAM 与 Device 之间的一次数据交换改变了 RAM 中 DMA 缓冲区的内容，假设在这个案例里恰好 `cache` 中缓存了 DMA 缓冲区对应的 RAM 中一段内存块，如果没有一种机制确保 `cache` 中的内容被新的 DMA 缓冲区数据所更新（或者无效），

那么很明显 cache 和它对应的 RAM 中的一段内存块在内容上出现了不一致性。若此时 CPU 试图去读取 Device 传到 RAM 的 DMA 缓冲区中的数据，它将直接从 cache 获得数据，这些数据显然不是它所期望的，因为 cache 对应的 RAM 中的数据已经被更新了。

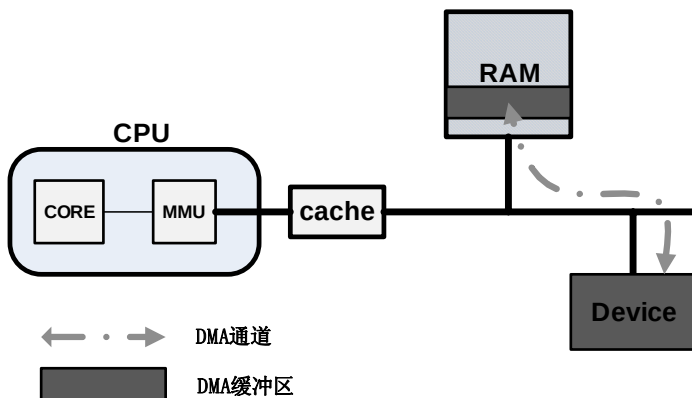


图 10-8 DMA 通道与缓冲区示意图

对于设备驱动程序而言，cache 与 DMA 缓冲区中数据的不一致的问题必须被小心予以处理，如同互斥问题导致的系统不稳定一样，这些问题在实际中的表现可能非常隐蔽，因此要求驱动程序需要透彻了解 DMA 映射的内幕机制。

单就 cache 一致性的问题，不同的体系架构有不同的策略，有些是在硬件层面予以保证（比如 x86 平台），有些则没有硬件支持而需要软件的参与（比如 ARM 平台）。Linux 内核中的通用 DMA 层尽力为设备驱动程序提供统一的接口来处理 cache 缓存一致性的问题，而将大量平台相关的代码对设备驱动程序隐藏起来。

下面根据 DMA 映射的三种情况来分别讨论这些接口。

○ 一致性 DMA 映射

Linux 内核 DMA 层为一致性 DMA 映射提供的接口函数为 `dma_alloc_coherent`，其函数原型是：

```
static inline void *
dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *dma_handle, gfp_t gfp)
```

函数分配的一致性 DMA 缓冲区的总线地址（也就是 DMA 地址）由参数 `dma_handle` 带回，函数返回的则是映射到 DMA 缓冲区的虚拟地址的起始地址。

前面提到 cache 一致性问题时讲到，不同体系架构对该问题的处理不尽相同，所以反映到该函数的具体实现上，不同平台必然会有不同的实现代码。下面以 x86 和 ARM 平台为例，分别讨论这两个平台上 `dma_alloc_coherent` 函数的实现。

对于 x86 平台，`dma_alloc_coherent` 的核心代码如下：

```
<arch/x86/include/asm/dmap-mapping.h>
```

```
static inline void *
dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *dma_handle,
                  gfp_t gfp)
{
    struct dma_map_ops *ops = get_dma_ops(dev);
    void *memory;
    gfp &= ~(__GFP_DMA | __GFP_HIGHMEM | __GFP_DMA32);
    if (dma_alloc_from_coherent(dev, size, dma_handle, &memory))
        return memory;

    ...

    memory = ops->alloc_coherent(dev, size, dma_handle,
                                dma_alloc_coherent_gfp_flags(dev, gfp));
    ...
    return memory;
}
```

函数首先试图通过 `dma_alloc_from_coherent` 在 per-device 的一致性存储区域中分配所需的 DMA 缓冲区，具体地 per-device 的一致性存储区域放在 `dev->dma_mem` 中。不过这种分配情况并不常见，所以 `dma_alloc_coherent` 实际上会通过 `ops->alloc_coherent()` 函数来为当前 DMA 传输分配缓冲区，对于 x86 平台，`ops->alloc_coherent` 指向的实际函数为 `dma_generic_alloc_coherent`。后者会根据指定的需要分配缓冲区的大小 `size` 调用 `alloc_pages_node` 来获得一组连续的物理页面，如果分配成功，将会把这种物理页面的起始物理地址放到 `dma_addr` 中返回供后续 DMA 通道传输数据时使用，同时会把该组物理页面对应的起始虚拟地址作为返回值返回：

```
<arch/x86/kernel/pci-dma.c>
```

```
void *dma_generic_alloc_coherent(struct device *dev, size_t size,
                                dma_addr_t *dma_addr, gfp_t flag)
{
    ...
    flag |= __GFP_ZERO;
again:
    page = alloc_pages_node(dev_to_node(dev), flag, get_order(size));
    ...
    addr = page_to_phys(page);
    ...
    *dma_addr = addr;
    return page_address(page);
}
```

因为 x86 平台由硬件处理 cache 一致性问题，所以此时的一致性 DMA 映射的建立只需保证能获得一组所需要大小的连续的物理页面即可。

作为对比，再来看一下 ARM 平台如何通过软件层面来保证这种 cache 的一致性：

<arch/arm/mm/dma-mapping.c>

```
void * dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *handle, gfp_t gfp)
{
    void *memory;
    if (dma_alloc_from_coherent(dev, size, handle, &memory))
        return memory;
    return __dma_alloc(dev, size, handle, gfp,
        pgprot_dmacoherent(pgprot_kernel));
}
```

函数中的 `dma_alloc_from_coherent` 与 x86 平台完全一样，不再讨论。下面重点看 `__dma_alloc`，其实现为：

<arch/arm/mm/dma-mapping.c>

```
static void *
__dma_alloc(struct device *dev, size_t size, dma_addr_t *handle, gfp_t gfp,
    pgprot_t prot)
{
    ...
    *handle = ~0;
    size = PAGE_ALIGN(size);
    page = __dma_alloc_buffer(dev, size, gfp);
    ..
    if (!arch_is_coherent())
        addr = __dma_alloc_remap(page, size, gfp, prot);
    else
        addr = page_address(page);

    if (addr)
        *handle = page_to_dma(dev, page);

    return addr;
}
```

函数首先调用 `__dma_alloc_buffer` 来分配大小为 `size` 的一段连续的物理内存页，因为 ARM 不是通过硬件来保证 cache 一致性，所以在 `__dma_alloc_buffer` 中，除了分配一段连续的物理页面外，还会对这段物理页面对应的虚拟地址调用 `dma_flush_range`，使其对应的 cache 和 write buffer 无效 (invalidated)，这使得 CPU 无论对主存进行读或者写操作，都不会因为 cache 和 write buffer 的存在而导致 DMA 操作时出现 cache 一致性问题：

<arch/arm/mm/dma-mapping.c>

```
static struct page * __dma_alloc_buffer(struct device *dev, size_t size, gfp_t gfp)
{
```

```

...
page = alloc_pages(gfp, order);
...
ptr = page_address(page);
memset(ptr, 0, size);
dmac_flush_range(ptr, ptr + size);
...
}

```

继续__dma_alloc 函数的讨论，__dma_alloc_buffer 的调用使得我们获得了一组连续的物理页面，而且对应的虚拟地址范围已经使 cache 失效，这是 ARM 平台在软件层面保证 cache 一致性的第一道工序。

__dma_alloc 函数中的 arch_is_coherent 返回 0，这表明 ARM 不是一种通过硬件保证 cache 一致性的平台，所以需要调用__dma_alloc_remap 函数来确保 cache 一致性，这个函数要做的工作其实是重新建立页表项来映射__dma_alloc_buffer 分配的一组物理页面，通过在新建的页表项关闭映射区的 cache 功能来解决 cache 一致性问题，更进一步的细节是 ARM 在虚拟地址空间的[0xFFC00000,0xFFE00000]这一 2 MB 的虚拟地址空间保留做 uncached 的 DMA 映射空间，这个区间的映射都是将 cache 功能关闭的。

所以__dma_alloc_remap 的功能是在[0xFFC00000,0xFFE00000]区间寻找一段虚拟地址段，将其重新映射到__dma_alloc_buffer 分配的一组物理页面，由于这种映射关闭了 cache 功能，所以保证了 DMA 操作时不会出现 cache 一致性的问题。

至此，我们已经讨论了一致性 DMA 映射的技术细节，通过 x86 和 ARM 两种平台实现一致性 DMA 映射的代码分析，可以看到，一致性映射最根本的操作是获得一组连续的物理页用做后续 DMA 操作的缓冲区。对于 x86 平台而言，因为硬件保证了 cache 一致性，所以 x86 平台的一致性映射最为简单；对于 ARM 平台而言，因为没有硬件参与解决 cache 一致性的问题，所以在软件层面上，通过重新映射新获得的物理地址空间，在页目录和页表项中关闭了这段映射区间上的 cache⁴功能，所以使得 cache 一致性也不再成为问题。另外，这种一致性映射所获得的 DMA 缓冲区的大小都是页面的整数倍，如果驱动程序需要更小的一致性 DMA 缓冲区，则应该使用内核提供的 DMA 池（pool）机制，稍后有一节专门讨论 DMA 缓冲池。

当驱动模块不再需要前面分配出的一致性 DMA 缓冲区时（比如对应的模块从系统中卸载时），需要使用 dma_free_coherent 函数来释放缓冲区，其原型是：

```
void dma_free_coherent(struct device *dev, size_t size, void *vaddr, dma_addr_t bus)
```

⁴ ARM 还有 write buffer 的概念，本书一概以 cache 统称了。

其中 `vaddr` 表示要释放的 DMA 缓冲区的起始虚拟地址，参数 `bus` 表示 DMA 缓冲区的总线地址。

对于一致性 DMA 映射来说，因为在分配 DMA 缓冲区时各平台相关代码已经从根本上解决了 cache 一致性问题，所以驱动程序使用这种映射来进行 DMA 操作时，将无须再关注 cache 的问题。实际的驱动程序中，一致性映射的缓冲区都是由驱动程序自身在初始化阶段分配，其生命周期可以一直延续到该驱动程序模块从系统中移除。但是在某些情况下，一致性映射也会遇到无法克服的困难，这主要是指驱动程序中使用到的 DMA 缓冲区并非由驱动程序分配，而是来自其他模块（典型的如网络设备驱动程序中用于数据包传输的 `skb->data` 所指向的缓冲区），此时需要使用另一种 DMA 映射方式：流式 DMA 映射。

○ 流式 DMA 映射

前面在总结一致性 DMA 映射时提到了这种映射的限制，因此内核同时提供了另一种 DMA 映射的方式，即流式 DMA 映射。这种 DMA 操作的特点是，DMA 传输通道所使用的缓冲区往往不是由当前驱动程序自身分配的，而且往往对每次 DMA 传输都会重新建立一个流式映射的缓冲区，此外由于无法确定外部模块传入的 DMA 缓冲区的映射情况，所以使用流式 DMA 映射时，设备驱动程序必须小心负责处理可能出现的 cache 一致性问题。在某些平台上，比如 ARM，CPU 的读/写用的是不同的 cache（读是用 cache，写则是用 write buffer），所以建立流式 DMA 映射需要指明数据在 DMA 通道中的流向，以便由内核决定是操作 cache 还是 write buffer。

Linux 内核 DMA 层为设备驱动程序提供的建立流式 DMA 映射的函数为 `dma_map_single`，严格意义上，`dma_map_single` 在内核代码中的存在形式是一个宏定义：

```
<include/asm-generic/dma-mapping-common.h>
-----
#define dma_map_single(d, a, s, r) dma_map_single_attrs(d, a, s, r, NULL)
```

为了方便理解和使用，下面给出其等价的函数原型：

```
dma_addr_t dma_map_single(struct device *dev, void *cpu_addr,
                          size_t size, enum dma_data_direction dir);
```

函数中的 `dev` 参数是一设备对象的指针，`cpu_addr` 是 CPU 的虚拟地址，也是流式映射 DMA 需要映射的区域，参数 `size` 指明了当前流式映射的空间范围，参数 `dir` 则用于表明当前的流式映射中 DMA 传输通道中的数据流向。函数返回的是 `dma_addr_t`，显然这是个 DMA 地址，被用来直接作为后续的 DMA 操作中的源地址或者目的地址，具体的数据流向在内核中由一个枚举型 `enum dma_data_direction` 表示，该数据类型定义如下：

```
<include/linux/dma-mapping.h>
-----
enum dma_data_direction {
    DMA_BIDIRECTIONAL = 0,
```



```

DMA_TO_DEVICE = 1,
DMA_FROM_DEVICE = 2,
DMA_NONE = 3,
};

```

如同建立一致性映射的 `dma_alloc_coherent` 函数一样，`dma_map_single` 内部用来完成实际的流式映射操作的代码也是体系架构相关的，内核通过 `struct dma_map_ops` 对象来屏蔽这种平台的差异，当然具体的平台需要提供其特有的 `struct dma_map_ops` 对象来供内核中的 DMA 层使用。正如前面所看到的，`dma_map_single` 宏被定义成 `dma_map_single_attrs`，后者是一个实实在在的函数：

```

<include/asm-generic/dma-mapping-common.h>
-----
static inline dma_addr_t dma_map_single_attrs(struct device *dev, void *ptr,
                                              size_t size,
                                              enum dma_data_direction dir,
                                              struct dma_attrs *attrs)
{
    struct dma_map_ops *ops = get_dma_ops(dev);
    dma_addr_t addr;
    ...
    addr = ops->map_page(dev, virt_to_page(ptr),
                        (unsigned long)ptr & ~PAGE_MASK, size,
                        dir, attrs);
    ...
    return addr;
}

```

不出所料，函数首先通过对 `get_dma_ops` 的调用来获得一个指向 `struct dma_map_ops` 对象的指针 `ops`，然后调用 `ops` 中的 `map_page` 方法，注意此处调用 `map_page` 时通过 `virt_to_page` 把要映射的 CPU 虚拟地址转化成了对应的物理页面的 `struct page` 指针，这虽然貌似是个很小的细节，但是却有几个很关键的暗示，我们不妨先看看 `virt_to_page` 在内核中的具体实现再说话：

```

<arch/x86/include/asm/page.h>
-----
#define virt_to_page(kaddr)  pfn_to_page(__pa(kaddr) >> PAGE_SHIFT)

```

貌似很简单的一个宏定义，内涵却并不单薄：`__pa(kaddr)` 中的 `__pa` 意味着可以对 `kaddr` 地址进行 `__pa` 操作，熟悉 Linux 内存管理的读者应该知道，只有内核空间中的线性映射区中的地址才可以通过 `__pa` 和 `__va` 宏来作物理地址和虚拟地址的转换，更具体地，`kmalloc` 分配的虚拟地址可以做 `__pa` 操作，`vmalloc` 则不可以。`__pa(kaddr) >> PAGE_SHIFT` 将得到的物理地址右移 12 位（以 x86，32 位，4 KB 页面大小为例）以获得该物理页的页帧号，紧接着 `pfn_to_page` 将页帧号转换成该页对应的 `struct page` 对象的指针。调用 `map_page` 时构造的第三个参数为 `(unsigned long)ptr & ~PAGE_MASK`，用来获得 `ptr` 所对应地址在物理页

面中的偏移量。

所以，如果想通过 `dma_map_single` 来对某一虚拟地址段作流式映射，则必须保证传进来的虚拟地址是通过 `kmalloc` 获得的。这里虽然是以 x86 平台为例，但其他平台的情况也大致如此。

关于 `ops->map_page` 的调用，`map_page` 的实现已经跟具体平台紧密相关了，本章依然以 x86 和 ARM 平台来探讨流式 DMA 映射的幕后机制。

对于 x86 平台而言，它为内核中 DMA 层的 `dma_map_single` 准备的 `struct dma_map_ops` 对象为 `nommu_dma_ops`⁵：

```
<arch/x86/kernel/pci-nommu.c>
-----
struct dma_map_ops nommu_dma_ops = {
    .alloc_coherent      = dma_generic_alloc_coherent,
    .free_coherent       = nommu_free_coherent,
    .map_sg              = nommu_map_sg,
    .map_page            = nommu_map_page,
    .sync_single_for_device = nommu_sync_single_for_device,
    .sync_sg_for_device  = nommu_sync_sg_for_device,
    .is_phys             = 1,
};
```

所以 `dma_map_single` 最终建立流式 DMA 映射的工作实际上发生在 `nommu_map_page` 中，该函数的实现如下：

```
<arch/x86/kernel/pci-nommu.c>
-----
static dma_addr_t nommu_map_page(struct device *dev, struct page *page,
                                unsigned long offset, size_t size,
                                enum dma_data_direction dir,
                                struct dma_attrs *attrs)
{
    dma_addr_t bus = page_to_phys(page) + offset;
    WARN_ON(size == 0);
    if (!check_addr("map_single", dev, bus, size))
        return DMA_ERROR_CODE;
    flush_write_buffers();
    return bus;
}
```

函数的主要任务在 `bus = page_to_phys(page) + offset` 代码中完成，即获得流式映射 DMA 缓

⁵ `nommu` 意味着没有 IOMMU 的支持，x86 架构下的 AMD 和 Intel 的实现现在都已包含了对 IOMMU 的支持，本书重点讨论 `nommu` 这种方式。

冲区的地址，flush_write_buffers 在 x86 平台上可以认为是个空函数。这段调用中 ops->map_page 中的 virt_to_page 和(unsigned long)ptr & ~PAGE_MASK 参数的构造与 nommu_map_page 中 bus = page_to_phys(page) + offset，给人一种冗余重复的感觉，读者或许觉得干脆在 dma_map_single_attrs 中直接返回__pa(ptr)不就完了吗，代码这样设计其实是出于可扩展性的一种需要，因为不同平台都需要有自己的 ops->map_page 的具体实现，即便是同一平台如 x86，还有 IOMMU 与 MMU 的区别，所以为了 map_page 的更大范围的可适应性，我们便看到了目前 ops->map_page 方法中参数的构造方式。

对于 ARM 平台而言，dma_map_single 函数的实现代码为：

```
<arch/arm/include/asm/dma-mapping.h>
```

```
static inline dma_addr_t dma_map_single(struct device *dev, void *cpu_addr,
                                         size_t size, enum dma_data_direction dir)
{
    ...
    __dma_single_cpu_to_dev(cpu_addr, size, dir);
    return virt_to_dma(dev, cpu_addr);
}
```

函数要完成的使命是将 cpu_addr 表示的一段虚拟地址映射到 DMA 缓冲区中，该缓冲区的起始地址将作为函数返回值返回。函数的核心是在__dma_single_cpu_to_dev 里面：

```
<arch/arm/mm/dma-mapping.c>
```

```
void __dma_single_cpu_to_dev(const void *kaddr, size_t size,
                             enum dma_data_direction dir)
{
    unsigned long paddr;
    dmac_map_area(kaddr, size, dir);
    paddr = __pa(kaddr);
    if (dir == DMA_FROM_DEVICE) {
        outer_inv_range(paddr, paddr + size);
    } else {
        outer_clean_range(paddr, paddr + size);
    }
}
```

如果 dir == DMA_FROM_DEVICE，表示 DMA 通道中数据的流向是从设备到主存，此时从 CPU 角度要完成的任务是从 RAM 中读取来自设备的数据，因为 ARM 没有硬件来解决 cache 一致性的问题，所以这种情况下需要通过软件来使得[paddr,paddr+size]范围所对应的 cache 失效，这样 CPU 将从主存中获得数据而不会读到 cache 中的数据。上面的代码中 outer_inv_range 函数用来完成这一任务，通常这是用 ARM 的汇编语言写成的代码，用来使 ARM 的 cache 失效。

如果 dir 是 DMA_FROM_DEVICE 之外的参数，主要是针对 DMA_TO_DEVICE 这种情况，

此时 DMA 传输通道中的数据流向是从主存到设备，因此要确保 CPU 往主存 RAM 中写入的数据不会只写到 cache 中，也就是要求 cache 具有 write-through 的特性，上述代码中的 `outer_clean_range` 用来完成这一任务。

通过以上对一致性 DMA 映射与流式 DMA 映射的分析，可以看到，当驱动程序主动去分配一个 DMA 缓冲区并且该缓冲区的存在周期与所在的驱动模块一样长时，就是用一致性 DMA 映射的好时机，这种映射类型的缓冲区因为在驱动程序一开始为 DMA 操作分配缓冲区时就解决了 cache 一致性问题，所以后续的 DMA 相关操作将无须再考虑这一问题。如果驱动程序需要使用从别的模块传进来的地址空间作为 DMA 缓冲区，那么就需要考虑使用流式 DMA 映射，这种映射对传进来的虚拟地址空间的要求是，它必须位于内核空间的线性映射区中，驱动程序在这种情况下处理主要是确保每次 DMA 操作前后 cache 的一致性问题，因为 x86 由硬件保证了 cache 的一致性，所以 x86 平台的流式映射只是简单地将虚拟地址对应的物理地址返回给驱动程序作为 DMA 缓冲区，而对 ARM 平台而言，则要保证虚拟地址和物理地址映射时关闭 cache 的功能。

所以建立流式 DMA 映射的关键点有两个：一是确保 CPU 侧的虚拟地址所对应的物理地址能够被设备 DMA 正确访问；二是要确保 cache 一致性的问题。

对于第一点，如果 CPU 侧的虚拟地址对应的物理地址不适合用来做 DMA 传输，那么有可能需要使用所谓回弹缓冲区（bounce buffer）的概念，稍后会讨论。

对于第二点，x86 平台上的驱动程序无须做什么事情，而 ARM 就比较特殊。出于代码可移植性的考虑，即便是为 x86 平台写驱动，也最好调用内核 DMA 层提供的接口来做 cache 一致性的操作。`struct dma_map_ops` 对象中的 `sync_single_for_cpu`、`sync_single_for_device`、`sync_sg_for_cpu` 和 `sync_sg_for_device` 方法就是用来处理 cache 一致性的问题，这些函数在 x86 平台不做什么事情，但是 ARM 平台就不一样了。为了让读者增加直观印象，这里我们对 ARM 平台上的相关操作简单介绍一下：

`sync_single_for_cpu` 方法用于数据从设备传到主存这种情况：当 DMA 完成时，设备已经将数据放到了位于主存的缓冲区中，CPU 需要读取该数据，为了避免 cache 的介入导致 CPU 读到的只是 cache 中的老数据，驱动程序需要在 CPU 读取之前调用该函数。函数在 ARM 平台上的操作是一个“invalidate”，也就是使 cache 无效，这样处理器将直接从主存获得数据。

`sync_single_for_device` 方法用于数据从主存传到设备这种情况：在启动 DMA 操作前，CPU 需要将数据放到位于主存的 DMA 缓冲区中，为了防止 write buffer 的介入，导致数据只是临时写到 write buffer 中，驱动程序需要在 CPU 往主存写数据之后启动 DMA 操作之前调用该函数。函数在 ARM 平台上的操作是一个“flush/clean”，也就是把 write buffer 中的数据冲到主存中，这样后续的 DMA 操作才会把正确的数据传给设备。

○ 分散/聚集映射（scatter/gather map）

到目前为止，对 DMA 操作时缓冲区的映射问题的讨论还仅限于单个缓冲区，本节将讨论另一种类型的 DMA 映射——分散/聚集映射。

正如其名称所暗示的那样，分散/聚集映射通过将虚拟地址上分散的 DMA 缓冲区通过一个类型为 `struct scatterlist` 的数组或者链表组织起来，然后通过一次的 DMA 传输操作在主存 RAM 与设备之间传输数据，如图 10-9 所示：

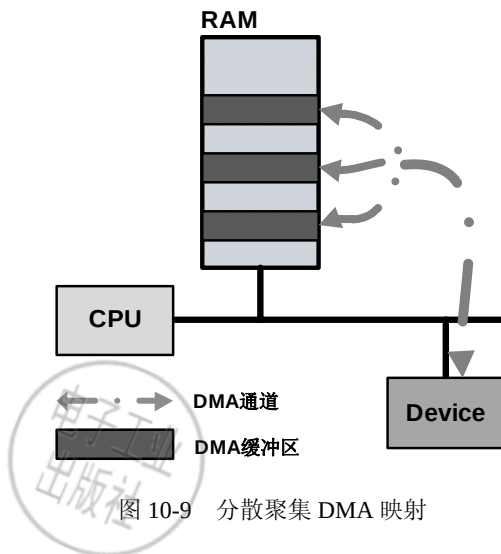


图 10-9 分散聚集 DMA 映射

图中显示了主存中三个分散的物理页面与设备之间进行的一次 DMA 传输时分散/聚集映射示意，其中单个物理页面与设备之间可以看做是一个单一的流式映射，每个这样的单一映射在内核中有数据结构 `struct scatterlist` 来表示：

```
<include/asm-generic/scatterlist.h>
-----
struct scatterlist {
    unsigned long    page_link;
    unsigned int offset;
    unsigned int length;
    dma_addr_t dma_address;
#ifdef CONFIG_NEED_SG_DMA_LENGTH
    unsigned int dma_length;
#endif
};
```

如果从 CPU 的角度看这种分散/聚集映射，它对应的需求是有三块数据（分别存放在三段分散的虚拟地址空间中）需要和设备进行交互（发送或者接收），通过建立 `struct scatterlist` 类型的数组/链表在一次 DMA 传输中完成所有的数据传递。

在 `struct scatterlist` 结构体中，`page_link` 指明了虚拟地址所对应的物理页面 `struct page` 对象

的地址（因为地址的最低两位总是 0，所以内核在 `page_link` 的后两位中安插了一些其他信息。例如：如果最低两位是 01，表示当前对象的 `page_link` 将会是指向下一个 `scatterlist` 数组的首地址，此时形成 `scatterlist` 的链式结构；如果最低两位是 10，表明当前对象是 `scatterlist` 数组中的最后一个），比如下面的代码：

```
//sg 是 struct scatterlist 类型指针
struct page *page =( struct page *) (sg->page_link & ~0x03);
```

`offset` 是数据在 DMA 缓冲区中的偏移地址，`length` 是要传输的数据块的大小，`dma_address` 则是设备 DMA 操作要使用的 DMA 地址（物理地址）。

内核中的 DMA 层为分散/聚集映射所提供的接口函数为 `dma_map_sg`，其原型为：

```
int dma_map_sg(struct device *dev, struct scatterlist *sg,
               int nents, enum dma_data_direction dir);
```

参数 `dev` 是设备对象的指针，`sg` 是 `struct scatterlist` 类型数组的首地址，`nents` 表示当前的分散/聚集映射中单一流式映射的个数，也是 `struct scatterlist` 数组/链表中的元素个数，`dir` 用于指明 DMA 传输中数据流的方向。

接下来分别以 x86 和 ARM 平台来讨论 `dma_map_sg` 建立分散/聚集映射的内核机制。

首先是 x86 平台，该平台的 `dma_map_sg` 通过 `struct dma_map_ops` 对象来调用其 `map_sg` 方法，此处依然选择以 `nommu_dma_ops` 对象为例，其 `map_sg` 方法的函数实现为：

```
<arch/x86/kernel/pci-nommu.c>
-----
static int nommu_map_sg(struct device *hwdev, struct scatterlist *sg,
                       int nents, enum dma_data_direction dir,
                       struct dma_attrs *attrs)
{
    struct scatterlist *s;
    int i;

    WARN_ON(nents == 0 || sg[0].length == 0);
    for_each_sg(sg, s, nents, i) {
        BUG_ON(!sg_page(s));
        s->dma_address = sg_phys(s);
        if (!check_addr("map_sg", hwdev, s->dma_address, s->length))
            return 0;
        s->dma_length = s->length;
    }
    flush_write_buffers();
    return nents;
}
```

函数的主体通过 `for_each_sg` 遍历 `sg` 数组/链表的所有元素，对于每个元素，都执行一个流式

映射，对于 x86 而言，如果没有 IOMMU 的介入，设备的 DMA 操作时使用的 DMA 地址就是物理地址，因此只需通过 `sg_phys` 获得当前元素所对应的物理地址即可，其代码如下：

```
<include/linux/scatterlist.h>
-----
static inline dma_addr_t sg_phys(struct scatterlist *sg)
{
    return page_to_phys(sg_page(sg)) + sg->offset;
}
```

`sg_page` 通过 scatterlist 对象 `sg` 的 `page_link` 成员取得所对应的物理页面的 `struct page` 对象地址：

```
static inline struct page *sg_page(struct scatterlist *sg)
{
    ...
    return (struct page *)((sg)->page_link & ~0x3);
}
```

`sg_phys` 再将返回的 `struct page` 指针通过 `page_to_phys` 获得页面的起始物理地址，加上实际数据块在页面中的偏移值 `sg->offset`，就获得了本次 DMA 操作的 DMA 地址。

ARM 平台的 `dma_map_sg` 的实现为：

```
<arch/arm/mm/dma-mapping.c>
-----
int dma_map_sg(struct device *dev, struct scatterlist *sg, int nents,
               enum dma_data_direction dir)
{
    ...
    for_each_sg(sg, s, nents, i) {
        s->dma_address = dma_map_page(dev, sg_page(s), s->offset,
                                       s->length, dir);
        if (dma_mapping_error(dev, s->dma_address))
            goto bad_mapping;
    }
    return nents;

bad_mapping:
    for_each_sg(sg, s, i, j)
        dma_unmap_page(dev, sg_dma_address(s), sg_dma_len(s), dir);
    return 0;
}
```

函数通过 `dma_map_page` 来映射 scatterlist 上的 `page_link`，跟 x86 平台不一样的是，ARM 架构需要通过软件来保证 cache 一致性问题，所以做完这种虚拟物理地址的转换之后，ARM 需要做的是使映射区对应的 cache 无效，以保证设备通过 DMA 将数据放到主存之后，CPU 读到的不是 cache 中的数据，或者是保证 CPU 写到 RAM 中的数据立刻反映到 RAM 中，

而不是暂时缓存到 cache 中，这样后续 DMA 在把主存中的数据传到设备中时，才能确保数据的有效性。

通过上面的讨论可以看到，分散/聚集 DMA 映射本质上是通过一次 DMA 操作把主存中分散的数据块在主存与设备之间进行传输，对于其中的每个数据块内核都会建立对应的一个流式 DMA 映射。另外，分散/聚集 DMA 映射需要设备的支持，而不完全由内核或者驱动程序决定。

10.3.5 回弹缓冲区（bounce buffer）

如果 CPU 侧虚拟地址对应的物理地址不适合设备的 DMA 操作，那么需要建立所谓的回弹缓冲区，它相当于一个中转站的作用，在把数据往设备方向传输时，驱动程序需要把 CPU 给的数据拷贝到回弹缓冲区，然后再启动 DMA 操作，反之亦然。所以回弹缓冲区必然是可以直接与设备进行 DMA 传输的，当传输结束时，再通过 CPU 的介入把回弹缓冲区中的数据搬运到最终的目标，所以除非外部传入的地址不能进行 DMA 传输，否则不应当使用它。图 10-10 展示了一个回弹缓冲区的使用：

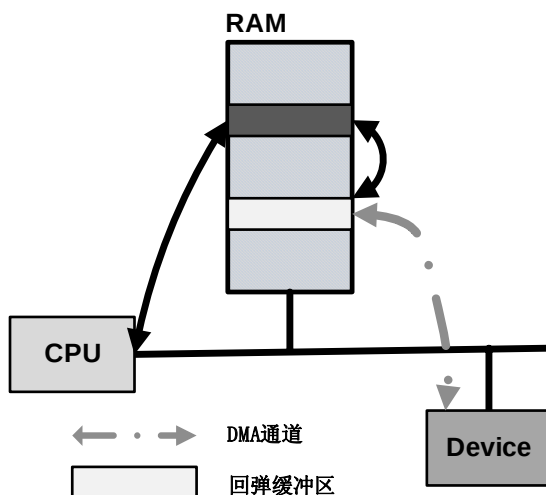


图 10-10 回弹缓冲区使用示意图

10.3.6 DMA 池

前面在讨论一致性 DMA 映射时，知道这种 DMA 映射所建立的缓冲区大小是单个页面的整数倍，如果驱动程序需要更小的一致性映射的 DMA 缓冲区，可以使用内核提供的 DMA 池机制。

DMA 池机制非常类似于 Linux 内存管理中的 slab 机制，它的实现建立在一致性 DMA 映射所获得的连续物理页面的基础之上，通过 DMA 池的接口函数在物理页面之上分配所谓块

大小的 DMA 缓冲区, 为方便叙述, 本书称这样的块为 DMA 缓冲块, 以区别于一致性 DMA 映射中页面级大小的缓冲区。显然为了管理跟踪物理页面中 DMA 缓冲块的分配和余下空闲空间的多少, 内核需要引入对应的管理数据结构, `struct dma_pool` 就是内核用来完成该任务的数据结构, 其定义如下:

<mm/dmapool.c>

```
struct dma_pool {
    struct list_head page_list;
    spinlock_t lock;
    size_t size;
    struct device *dev;
    size_t allocation;
    size_t boundary;
    char name[32];
    wait_queue_head_t waitq;
    struct list_head pools;
};
```

一些成员的作用如下:

`struct list_head page_list`

用来将一致性 DMA 映射建立的页面组织成链表。

`size_t size`

该 DMA 池用来分配一致性 DMA 映射的缓冲区的大小, 也称块大小。

`struct device *dev`

进行 DMA 操作的设备对象指针。

`char name[32]`

DMA 池的名称, 主要在调试或者诊断时使用。

`struct list_head pools`

用来将当前 DMA 池对象加到 `dev->dma_pools` 链表中。

在利用 DMA 池进行缓冲区分配之前, 首先需要创建一个 DMA 池, 这是通过函数 `dma_pool_create` 来完成的, 该函数的核心实现为:

<mm/dmapool.c>

```
struct dma_pool *dma_pool_create(const char *name, struct device *dev,
                                size_t size, size_t align, size_t boundary)
{
```

```

    struct dma_pool *retval;
    size_t allocation;
    ...
    allocation = max_t(size_t, size, PAGE_SIZE);

    if (!boundary) {
        boundary = allocation;
    } else if ((boundary < size) || (boundary & (boundary - 1))) {
        return NULL;
    }

    retval = kmalloc_node(sizeof(*retval), GFP_KERNEL, dev_to_node(dev));
    if (!retval)
        return retval;

    strncpy(retval->name, name, sizeof(retval->name));
    retval->dev = dev;

    INIT_LIST_HEAD(&retval->page_list);
    spin_lock_init(&retval->lock);
    retval->size = size;
    retval->boundary = boundary;
    retval->allocation = allocation;
    init_waitqueue_head(&retval->waitq);
    ...
    list_add(&retval->pools, &dev->dma_pools);
    ...
}

```

函数的核心工作就是分配一个 `struct dma_pool` 对象并初始化。不过有些细节还是值得探讨一下, 先看函数的参数, `name` 用于指定即将创建的 DMA 池的名称, `size` 用于指定在该 DMA 池中分配缓冲块的大小, `align` 用于指定当前 DMA 池分配操作所遵守的对齐方式。

函数首先确定当前 DMA 池分配的对齐指标, 我们对此不感兴趣。接下来 `allocation` 用来保存参数 `size` 与 `PAGE_SIZE` 之间的最大值, 所以如果 `size` 小于一个页面大小的话, `allocation` 将等于 `PAGE_SIZE`, 在后续的 DMA 池的页面分配部分, 该值用来决定需要分配的连续物理页的数量。如果函数调用时没有指定 `boundary` (`boundary` 为空), 那么 `boundary` 就是 `allocation` 的大小。 `kmalloc_node` 函数用来分配一个 `struct dma_pool` 对象, 紧接着就是对该对象进行初始化。

以上是 `dma_pool_create` 函数总体框架, 现在我们有了一个已经被初始化过的 DMA 池对象, 如果要在该对象中分配一个一致性映射的 DMA 缓冲区块, 应该使用 `dma_pool_alloc` 函数:

<mm/dmapool.c>

```
void *dma_pool_alloc(struct dma_pool *pool, gfp_t mem_flags,
```

```

        dma_addr_t *handle)
{
    unsigned long flags;
    struct dma_page *page;
    size_t offset;
    void *retval;

    spin_lock_irqsave(&pool->lock, flags);
restart:
    list_for_each_entry(page, &pool->page_list, page_list) {
        if (page->offset < pool->allocation)
            goto ready;
    }
    page = pool_alloc_page(pool, GFP_ATOMIC);
    if (!page) {
        if (mem_flags & __GFP_WAIT) {
            DECLARE_WAITQUEUE(wait, current);
            __set_current_state(TASK_INTERRUPTIBLE);
            __add_wait_queue(&pool->waitq, &wait);
            spin_unlock_irqrestore(&pool->lock, flags);
            schedule_timeout(PPOOL_TIMEOUT_JIFFIES);
            spin_lock_irqsave(&pool->lock, flags);
            __remove_wait_queue(&pool->waitq, &wait);
            goto restart;
        }
        retval = NULL;
        goto done;
    }

ready:
    page->in_use++;
    offset = page->offset;
    page->offset = *(int *) (page->vaddr + offset);
    retval = offset + page->vaddr;
    *handle = offset + page->dma;
done:
    spin_unlock_irqrestore(&pool->lock, flags);
    return retval;
}

```

这个函数的主线框架是，如果当前 DMA 池中有页面满足接下来的缓冲块分配需求，那么就在该页面上分配，否则通过调用 `pool_alloc_page` 来重新分配一段连续物理页。DMA 池中每段这样的页面都用一个 `struct dma_page` 类型的对象来表示：

<mm/dmapool.c>

```
struct dma_page {
```

```

    struct list_head page_list;
    void *vaddr;
    dma_addr_t dma;
    unsigned int in_use;
    unsigned int offset;
};

```

`dma_pool_alloc` 函数返回 DMA 池中某一段物理页面中空闲块的虚拟地址，其对应的 DMA 地址由参数 `handle` 返回。如果调用 `dma_pool_alloc` 函数时在 `mem_flags` 中指定了 `__GFP_WAIT` 标志，那么在系统中暂时没有一段连续的物理页面满足分配需求时，函数会进入睡眠等待状态，等 `POOL_TIMEOUT_JIFFIES` 指定的时间到期，或者前面的分配请求可以被满足（比如有模块调用了 `dma_pool_free` 来释放当前 DMA 池中的某一 DMA 缓冲块），该函数才会从睡眠等待状态中醒来。

与 `dma_pool_alloc` 相对应，如果要从一个 DMA 池中释放某一 DMA 缓冲块，则应该调用 `dma_pool_free` 函数，该函数原型为：

```

<include/linux/dmapool.h>
-----
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t addr);

```

参数 `pool` 用于指明要释放的 DMA 缓冲块隶属的 DMA 池，`vaddr` 是要释放的 DMA 缓冲块的虚拟地址，`addr` 则是其 DMA 地址。

如果一个 DMA 池不再使用，应该调用函数 `dma_pool_destroy` 销毁之，该函数的原型为：

```

<include/linux/dmapool.h>
-----
void dma_pool_destroy(struct dma_pool *pool);

```

参数 `pool` 是指向要销毁的 DMA 池对象的指针。函数调用者需要保证调用该函数时 DMA 池中已经没有 DMA 缓冲块还在使用，而且一旦 DMA 池对象被销毁，后续将没有模块试图再去使用它。

10.4 本章小结

本章主要讨论了两个话题，一个是如何将用户空间的地址映射到设备内存中，将用户空间的地址直接映射到设备地址上可以使得应用程序直接使用设备内存，因为绕过了内核部分的介入，使得程序的性能得以提高。

另一个话题与 DMA 操作相关，主要集中在如何为一个 DMA 传输建立 DMA 缓冲区，所谓缓冲区的建立，主要是在系统的主存中为 DMA 操作分配一段内存区域，因为 cache 的存在使得原本单纯的任务变得有些不那么坦荡，而且还应该注意并不是主存中所有的区域都适合 DMA 传输。内核为方便设备驱动程序的使用，提供了一个通用的 DMA 层来统一 DMA

操作缓冲区建立等操作。内核中关于 DMA 缓冲区的建立主要有三种方式。一是一致性 DMA 映射，这种映射的主要操作是在主存中分配一段连续的物理页面作为后续 DMA 操作的缓冲区，对于那些没有硬件保证 cache 一致性的平台，必须由软件来保证 cache 一致性，主要的原理是将新分配的 DMA 缓冲区所对应范围内的 cache 等特性关闭。因为一致性 DMA 映射在建立之初就解决了 cache 一致性问题，所以后续的 DMA 操作就无须再关心这个问题了。二是流式 DMA 映射，基本上这种映射的缓冲区都不是驱动程序自身所分配，因此驱动程序对于这种映射要完成的任务主要是确保外部传入的缓冲区的虚拟地址映射的物理地址范围可以进行 DMA 操作，然后将这些虚拟地址转化成物理地址作为后续 DMA 操作的缓冲区。因为驱动程序在此只是简单地做虚拟地址到 DMA 地址的转换工作，所以后续的每次 DMA 操作时都需要小心采取措施解决 cache 一致性的问题（如果那种平台没有在硬件上保证 cache 一致性的话）。

如果有更小的 DMA 一致性缓冲区分配需求（比如小于一个物理页面），可以使用内核提供的 DMA 池机制。



第 11 章

块设备驱动程序

在 Linux 的设备驱动架构中，块设备是与字符型设备不同类型的另一种设备，因此内核在支持块设备驱动程序时所使用的的数据结构和 I/O 模型的设计等方面都与字符型设备驱动程序有所不同。相对于字符型设备，Linux 内核对块设备的支持要复杂得多，基本上要牵涉到内核组件的很多方面。再者因为主要是内核的文件系统在与块设备驱动程序打交道，所以在对块设备所提供接口的调用关系上，块设备相对于字符设备驱动程序而言，也要晦涩很多。不过幸运的是，对设备驱动程序而言，我们不需要了解这其中的每个技术细节。在本章的讨论中，我们致力于勾勒出块设备相关的整体框架，这样读者将会对 Linux 下与块（Block）相关的部分有个总体的认识，在此基础上我们会将更多的笔墨放在与设备驱动息息相关的那些部分。

本章的结构总体上可以分成三部分：第一部分讨论块设备与系统的交互，即块设备如何被注册进系统，以及块设备在系统中的存在形式等；第二部分将从块设备驱动程序的角度出发，探讨块设备驱动程序的整体框架，包括各种外部接口函数的讨论等；第三部分将讨论块设备如何完成其真正的功能——让其所控制的设备完成上层的 I/O 请求，这部分的重点是块设备的请求队列，在此基础上将讨论一个块设备如何在现有的请求队列上实现自己的请求函数，通过这些讨论，读者将会明白块设备驱动程序中与请求队列相关的函数被调用时的上下文背景与幕后细节。在所有这些技术细节展开之前，我们会通过一个 RAM DISK 的实例来展示块设备驱动程序编写的若干要素，读者可以亲自在自己的机器上编译、运行与操作这个基于系统 RAM 模拟出来的磁盘，我希望通过这个具体的例子让读者建立起探索块设备驱动程序背后神秘内幕的好奇心。

在讨论开始前，我先给出一张 Linux 系统下块相关组件的框架图（图 11-1），以便读者能在进入后续的具体讨论前先建立个全局性的印象。图中，最上层是 Linux 内核的文件系统组件，主要是磁盘文件系统，同时也包括块设备文件等。接下来是一个通用的块层，用来完成块设备的相关核心功能，在通用的块层之下是 I/O 调度器组件，主要用来对块设备请求队列中的请求进行调度，以最大程度优化硬件操作的性能（比如 I/O 调度器可能会对请求队列中的某些请求进行合并或者调整各请求间的顺序，以尽可能减少磁盘磁头移动的距离）。I/O 调度器之下是本章要讨论的主角——块设备驱动程序，它控制对应的硬件设备以完成来自上层的 I/O 请求等操作。为了叙述方便，笔者将 Linux 内核中所有与块相关的组

件统称为块子系统。

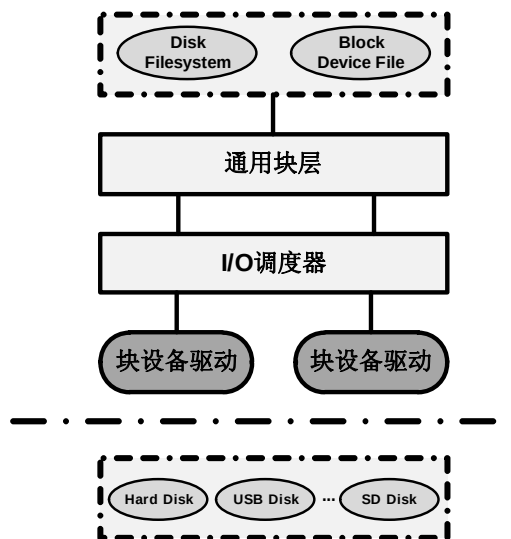


图 11-1 Linux 块设备相关组件框架图

11.1 块子系统初始化

该子系统的初始化发生在 `genhd_device_init` 函数中，该函数的实现代码为：

<block/genhd.c>

```
static int __init genhd_device_init(void)
{
    int error;

    block_class.dev_kobj = sysfs_dev_block_kobj;
    error = class_register(&block_class);
    if (unlikely(error))
        return error;
    bdev_map = kobj_map_init(base_probe, &block_class_lock);
    blk_dev_init();
    register_blkdev(BLOCK_EXT_MAJOR, "blkext");

    return 0;
}
```

这个函数在系统启动的时候被调用，函数前面的 `__init` 标志也证实了这一点。函数首先为内核对象 `block_class` 指定其 `kobject` 类型成员 `dev_kobj` 的所属，`block_class` 是个全局型的 `struct class` 对象。这里为 `block_class` 中的 `dev_kobj` 所指定的 `sysfs_dev_block_kobj` 是个 `kobject` 类型的指针，该指针的生成发生在 `devices_init`：

```
<drivers/base/core.c>
```

```
int __init devices_init(void)
{
    ...
    dev_kobj = kobject_create_and_add("dev", NULL);
    if (!dev_kobj)
        goto dev_kobj_err;
    sysfs_dev_block_kobj = kobject_create_and_add("block", dev_kobj);
    ...
}
```

所以如果单从 sysfs 文件系统的角度出发, `block_class.dev_kobj = sysfs_dev_block_kobj` 将使 `block_class` 指向 `/dev/block` 这个目录, 此处读者留意下这个细节, 在后续相关的讨论中或许会用的着。 `class_register` 将 `block_class` 这个类对象注册进系统, 相关细节已在“Linux 设备驱动模型”一章讨论过。

`bdev_map` 是一个类型为 `struct kobj_map` 的全局指针型变量, `kobj_map_init` 函数用来给它动态分配一个 `struct kobj_map` 类型对象, 这里的操作原理以及系统对 `bdev_map` 如何使用, 读者不妨再回头看看 2.5 节“字符设备的注册”, 块设备与字符设备在对 `struct kobj_map` 变量的使用上是相同的。

接下来的 `blk_dev_init` 主要是创建一个名为“kblockd”的工作队列和两个 `kmem_cache` 缓冲池, 具体代码为:

```
<block/blk-core.c>
```

```
int __init blk_dev_init(void)
{
    kblockd_workqueue = create_workqueue("kblockd");
    if (!kblockd_workqueue)
        panic("Failed to create kblockd\n");

    request_cachep = kmem_cache_create("blkdev_requests",
                                       sizeof(struct request), 0, SLAB_PANIC, NULL);

    blk_requestq_cachep = kmem_cache_create("blkdev_queue",
                                             sizeof(struct request_queue), 0, SLAB_PANIC, NULL);

    return 0;
}
```

这里用到了“分配内存”和“延迟操作”两章中讨论过的内容, 此处不应该有什么难以理解的地方。在 Linux 系统下可以通过 `ps` 命令看到这里创建的 `kblockd` 工作队列, 大致如下:

```
[root@AMDLinuxFGL ~]# ps aux | grep kblockd
```


root	25	0.0	0.0	0	0 ?	S<	17:48	0:00 [kblockd/0]
root	26	0.0	0.0	0	0 ?	S<	17:48	0:00 [kblockd/1]
root	27	0.0	0.0	0	0 ?	S<	17:48	0:00 [kblockd/2]
root	28	0.0	0.0	0	0 ?	S<	17:48	0:00 [kblockd/3]

`create_workqueue` 函数在创建工作队列时，会在系统的每个 CPU 上都创建一个工作进程，因为系统中有 4 个 CPU，所以上面 `ps` 的输出信息显示创建了 4 个 `kblockd` 进程。紧接着两个名为“`blkdev_requests`”与“`blkdev_queue`”的 `kmem_cache` 被创建出来，因为内核中块设备的请求队列（`queue`）及请求（`request`）对象的分配与释放非常频繁，所以内核采用了 `kmem_cache` 方式来进行。

`genhd_device_init` 函数为 Linux 内核中块设备驱动程序的整体框架进行了必要的初始化，在本章后续的内容中将看到该函数此处所做工作的作用。这段内容看起来的确很抽象，而且貌似与设备驱动程序的关系也不大，但这里不得不把它先交代一下，因为一旦正剧开始后，我们需要引用到这里面的东西。

11.2 ramdisk 源码实例

下面这个 `ramdisk` 的完整源码，读者可以在 www.embexperts.com 网站上下载。因为篇幅的原因，此处我将这些源码进行了精简，删减了诸如错误处理，模块参数等方面的内容，但是块设备驱动程序的关键元素都在，通过这里例子，读者一来可以直观感受一下块设备驱动程序，二来也可以把它做为练习块设备驱动程序编写的起点，我会在后续的内容当中揭示这些关键元素的内核机制。

`ramdisk` 是用系统中的 RAM 来模拟一个块设备，在我们的这个例子中，会产生两个 `disk` 设备，每个设备使用 8 MB 虚拟地址空间，因为空间来自于 `vmalloc`，所以无法保证这段空间在物理内存页面上的连续性，不过这并不会影响该 `disk` 的行为。图 11-2 显示了其中的一个块设备与所对应 RAM 之间的关系：

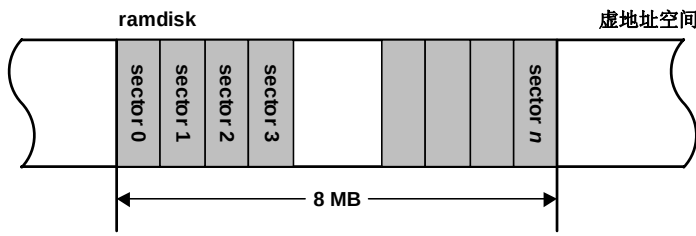


图 11-2 ramdisk 使用的虚拟地址空间

在我们的源码中，`ramdisk` 所在虚拟地址空间的起始地址保存在 `RAMHD_DEV` 对象的 `data`

成员中，驱动程序在处理来自内核块子系统中的读/写请求时，会给出扇区 sector 的信息，程序据此对 data 区进行寻址，然后用 memcpy 函数在块设备与块子系统之间传输数据。当一个 ramdisk 设备被创建出来之后，可以用 fdisk 给它分区，可以用 mkfs.ext3 等工具来在分区上创建文件系统，然后再把它挂载到一个目录上。所有这些，从用户的角度，除了系统掉电之后 ramdisk 上的内容会丢失外，跟真实的硬盘几乎没有区别。当然在驱动程序的实现方面，它要比实际的硬盘容易多了，因为在底层的 I/O 方面，ramdisk 只需使用 memcpy 这样的函数。实际的硬盘驱动，比如 SATA，会涵盖相当复杂的硬件层面操作逻辑，在 x86 平台上，这由南桥的开发者手册或者是 datasheet 来提供。不过我们用 ramdisk 已经足以揭示 Linux 内核中关于块设备驱动程序框架设计的潜在秘密。

这个例子还有一个非常重要的潜在用途，那就是通过 ramdisk 来研究 Linux 的文件系统，比如 ext3 等，因为 mkfs.ext3 工具会将 ext3 文件系统做到这个 ramdisk 中，这意味着 ext 文件系统家族的超级块、组描述符、数据位图、inode 位图和 inode 表等一系列的重量级数据结构会被记录到 ramdisk 中，我们可以通过另外的方式去读/写这段 RAM 空间来获得现场的数据，这对理解 Linux 中 ext 文件系统家族的源代码是非常有帮助的。而记得大约在 8 年以前，为了获得这些数据，我是直接通过操作南桥寄存器的方式来对硬盘数据进行读/写，这种原生态的方式虽然有趣，但显然比较低效。

这个例子有两个版本的实现，一个是通过 make_request 方式，另一个是通过 request 方式，关于两者之间的区别，后续的讨论中会予以说明。

11.2.1 make_request 版本的 RAM DISK 源码

```
<ramhd_mkreq.c>
-----
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

#include <linux/fs.h>
#include <linux/types.h>
#include <linux/fcntl.h>
#include <linux/vmalloc.h>
#include <linux/blkdev.h>
#include <linux/hdreg.h>

#define RAMHD_NAME          "ramhd"
#define RAMHD_MAX_DEVICE    2
#define RAMHD_MAX_PARTITIONS 4

#define RAMHD_SECTOR_SIZE    512
#define RAMHD_SECTORS        16
```

```
#define RAMHD_HEADS          4
#define RAMHD_CYLINDERS      256

#define RAMHD_SECTOR_TOTAL (RAMHD_SECTORS * RAMHD_HEADS * RAMHD_CYLINDERS)
#define RAMHD_SIZE          (RAMHD_SECTOR_SIZE * RAMHD_SECTOR_TOTAL) //8MB

typedef struct{
    unsigned char *data;
    struct request_queue *queue;
    struct gendisk *gd;
}RAMHD_DEV;

static char *sdisk[RAMHD_MAX_DEVICE] = {NULL,};
static RAMHD_DEV *rdev[RAMHD_MAX_DEVICE] = {NULL,};

static dev_t ramhd_major;

static int ramhd_space_init(void)
{
    int i;
    int err = 0;
    for(i = 0; i < RAMHD_MAX_DEVICE; i++){
        sdisk[i] = vmalloc(RAMHD_SIZE);
        if(!sdisk[i]){
            err = -ENOMEM;
            return err;
        }
        memset(sdisk[i], 0, RAMHD_SIZE);
    }

    return err;
}

static void ramhd_space_clean(void)
{
    int i;
    for(i = 0; i < RAMHD_MAX_DEVICE; i++){
        vfree(sdisk[i]);
    }
}

static int ramhd_open(struct block_device *bdev, fmode_t mode)
{
    return 0;
}
```

```

static int ramhd_release(struct gendisk *gd, fmode_t mode)
{
    return 0;
}

static int ramhd_ioctl(struct block_device *bdev, fmode_t mode, unsigned int cmd, unsigned long arg)
{
    int err;
    struct hd_geometry geo;

    switch(cmd)
    {
        case HDIO_GETGEO:
            err = !access_ok(VERIFY_WRITE, arg, sizeof(geo));
            if(err) return -EFAULT;

            geo.cylinders = RAMHD_CYLINDERS;
            geo.heads = RAMHD_HEADS;
            geo.sectors = RAMHD_SECTORS;
            geo.start = get_start_sect(bdev);
            if(copy_to_user((void *)arg, &geo, sizeof(geo)))
                return -EFAULT;
            return 0;
    }

    return -ENOTTY;
}

static struct block_device_operations ramhd_fops =
{
    .owner = THIS_MODULE,
    .open = ramhd_open,
    .release = ramhd_release,
    .ioctl = ramhd_ioctl,
};

static int ramhd_make_request(struct request_queue *q, struct bio *bio)
{
    char *pRHdata;
    char *pBuffer;
    struct bio_vec *bvec;
    int i;
    int err = 0;

    struct block_device *bdev = bio->bi_bdev;
    RAMHD_DEV *pdev = bdev->bd_disk->private_data;

```

```
if(((bio->bi_sector * RAMHD_SECTOR_SIZE) + bio->bi_size) > RAMHD_SIZE){
    err = -EIO;
    goto out;
}

pRHdata = pdev->data + (bio->bi_sector * RAMHD_SECTOR_SIZE);

bio_for_each_segment(bvec, bio, i) {
    pBuffer = kmap(bvec->bv_page) + bvec->bv_offset;
    switch(bio_data_dir(bio))
    {
        case READ:
            memcpy(pBuffer, pRHdata, bvec->bv_len);
            flush_dcache_page(bvec->bv_page);
            break;
        case WRITE:
            flush_dcache_page(bvec->bv_page);
            memcpy(pRHdata, pBuffer, bvec->bv_len);
            break;
        default:
            kunmap(bvec->bv_page);
            goto out;
    }
    kunmap(bvec->bv_page);
    pRHdata += bvec->bv_len;
}

out:
    bio_endio(bio, err);
    return 0;
}

static int alloc_ramdev(void)
{
    int i;
    for(i = 0; i < RAMHD_MAX_DEVICE; i++){
        rdev[i] = kzalloc(sizeof(RAMHD_DEV), GFP_KERNEL);
        if(!rdev[i])
            return -ENOMEM;
    }
    return 0;
}

static void clean_ramdev(void)
{
    int i;
```

```

        for(i = 0; i < RAMHD_MAX_DEVICE; i++){
            if(rdev[i])
                kfree(rdev[i]);
        }
    }

static int __init ramhd_init(void)
{
    int i;

    ramhd_space_init();
    alloc_ramdev();

    ramhd_major = register_blkdev(0, RAMHD_NAME);

    for(i = 0; i < RAMHD_MAX_DEVICE; i++)
    {
        rdev[i]->data = sdisk[i];
        rdev[i]->queue = blk_alloc_queue(GFP_KERNEL);
        blk_queue_make_request(rdev[i]->queue, ramhd_make_request);
        rdev[i]->gd = alloc_disk(RAMHD_MAX_PARTITIONS);
        rdev[i]->gd->major = ramhd_major;
        rdev[i]->gd->first_minor = i * RAMHD_MAX_PARTITIONS;
        rdev[i]->gd->fops = &ramhd_fops;
        rdev[i]->gd->queue = rdev[i]->queue;
        rdev[i]->gd->private_data = rdev[i];
        sprintf(rdev[i]->gd->disk_name, "ramhd%c", 'a'+i);
        rdev[i]->gd->flags |= GENHD_FL_SUPPRESS_PARTITION_INFO;
        set_capacity(rdev[i]->gd, RAMHD_SECTOR_TOTAL);
        add_disk(rdev[i]->gd);
    }
    return 0;
}

static void __exit ramhd_exit(void)
{
    int i;

    for(i = 0; i < RAMHD_MAX_DEVICE; i++)
    {
        del_gendisk(rdev[i]->gd);
        put_disk(rdev[i]->gd);
        blk_cleanup_queue(rdev[i]->queue);
    }
    clean_ramdev();
    ramhd_space_clean();
}

```

```

    unregister_blkdev(ramhd_major, RAMHD_NAME);
}

module_init(ramhd_init);
module_exit(ramhd_exit);

MODULE_AUTHOR("dennis chen @ AMDLinuxFGL");
MODULE_DESCRIPTION("The ramdisk implementation with request function");
MODULE_LICENSE("GPL");

```

11.2.2 request 版本的 RAM DISK 源码

```

<ramhd_req.c>
-----
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

#include <linux/fs.h>
#include <linux/types.h>
#include <linux/fcntl.h>
#include <linux/vmalloc.h>
#include <linux/blkdev.h>
#include <linux/hdreg.h>

#define RAMHD_NAME "ramsd"
#define RAMHD_MAX_DEVICE 2
#define RAMHD_MAX_PARTITIONS 4

#define RAMHD_SECTOR_SIZE 512
#define RAMHD_SECTORS 16
#define RAMHD_HEADS 4
#define RAMHD_CYLINDERS 256

#define RAMHD_SECTOR_TOTAL (RAMHD_SECTORS * RAMHD_HEADS * RAMHD_CYLINDERS)
#define RAMHD_SIZE (RAMHD_SECTOR_SIZE * RAMHD_SECTOR_TOTAL) //8MB

typedef struct{
    unsigned char *data;
    struct request_queue *queue;
    spinlock_t lock;
    struct gendisk *gd;
}RAMHD_DEV;

static char *sdisk[RAMHD_MAX_DEVICE];
static RAMHD_DEV *rdev[RAMHD_MAX_DEVICE];

```

```
static dev_t ramhd_major;

static int ramhd_space_init(void)
{
    int i;
    int err = 0;
    for(i = 0; i < RAMHD_MAX_DEVICE; i++){
        sdisk[i] = vmalloc(RAMHD_SIZE);
        if(!sdisk[i]){
            err = -ENOMEM;
            return err;
        }
        memset(sdisk[i], 0, RAMHD_SIZE);
    }

    return err;
}

static void ramhd_space_clean(void)
{
    int i;
    for(i = 0; i < RAMHD_MAX_DEVICE; i++){
        vfree(sdisk[i]);
    }
}

static int alloc_ramdev(void)
{
    int i;
    for(i = 0; i < RAMHD_MAX_DEVICE; i++){
        rdev[i] = kzalloc(sizeof(RAMHD_DEV), GFP_KERNEL);
        if(!rdev[i])
            return -ENOMEM;
    }
    return 0;
}

static void clean_ramdev(void)
{
    int i;
    for(i = 0; i < RAMHD_MAX_DEVICE; i++){
        if(rdev[i])
            kfree(rdev[i]);
    }
}
```



```
int ramhd_open(struct block_device *bdev, fmode_t mode)
{
    return 0;
}

int ramhd_release(struct gendisk *gd, fmode_t mode)
{
    return 0;
}

static int ramhd_ioctl(struct block_device *bdev, fmode_t mode, unsigned int cmd, unsigned long arg)
{
    int err;
    struct hd_geometry geo;

    switch(cmd)
    {
        case HDIO_GETGEO:
            err = !access_ok(VERIFY_WRITE, arg, sizeof(geo));
            if(err) return -EFAULT;

            geo.cylinders = RAMHD_CYLINDERS;
            geo.heads = RAMHD_HEADS;
            geo.sectors = RAMHD_SECTORS;
            geo.start = get_start_sect(bdev);
            if(copy_to_user((void *)arg, &geo, sizeof(geo)))
                return -EFAULT;
            return 0;
    }

    return -ENOTTY;
}

static struct block_device_operations ramhd_fops =
{
    .owner = THIS_MODULE,
    .open = ramhd_open,
    .release = ramhd_release,
    .ioctl = ramhd_ioctl,
};

void ramhd_req_func (struct request_queue *q)
{
    struct request *req;
    RAMHD_DEV *pdev;
```

```

char *pData;
unsigned long addr, size, start;
req = blk_fetch_request(q);
while (req) {
    start = blk_rq_pos(req); // The sector cursor of the current request
    pdev = (RAMHD_DEV *)req->rq_disk->private_data;
    pData = pdev->data;
    addr = (unsigned long)pData + start * RAMHD_SECTOR_SIZE;
    size = blk_rq_cur_bytes(req);
    if (rq_data_dir(req) == READ)
        memcpy(req->buffer, (char *)addr, size);
    else
        memcpy((char *)addr, req->buffer, size);

    if (!__blk_end_request_cur(req, 0))
        req = blk_fetch_request(q);
}
}

int ramhd_init(void)
{
    int i;
    ramhd_space_init();
    alloc_ramdev();

    ramhd_major = register_blkdev(0, RAMHD_NAME);

    for(i = 0; i < RAMHD_MAX_DEVICE; i++)
    {
        rdev[i]->data = sdisk[i];
        rdev[i]->gd = alloc_disk(RAMHD_MAX_PARTITIONS);
        spin_lock_init(&rdev[i]->lock);
        rdev[i]->queue = blk_init_queue(ramhd_req_func, &rdev[i]->lock);
        rdev[i]->gd->major = ramhd_major;
        rdev[i]->gd->first_minor = i * RAMHD_MAX_PARTITIONS;
        rdev[i]->gd->fops = &ramhd_fops;
        rdev[i]->gd->queue = rdev[i]->queue;
        rdev[i]->gd->private_data = rdev[i];
        sprintf(rdev[i]->gd->disk_name, "ramsd%c", 'a'+i);
        set_capacity(rdev[i]->gd, RAMHD_SECTOR_TOTAL);
        add_disk(rdev[i]->gd);
    }

    return 0;
}

```

```

void ramhd_exit(void)
{
    int i;
    for(i = 0; i < RAMHD_MAX_DEVICE; i++)
    {
        del_gendisk(rdev[i]->gd);
        put_disk(rdev[i]->gd);
        blk_cleanup_queue(rdev[i]->queue);
    }
    unregister_blkdev(ramhd_major, RAMHD_NAME);
    clean_ramdev();
    ramhd_space_clean();
}

module_init(ramhd_init);
module_exit(ramhd_exit);

MODULE_AUTHOR("dennis chen @ AMDLinuxFGL");
MODULE_DESCRIPTION("The ramdisk implementation with request function");
MODULE_LICENSE("GPL");

```

为了编译上面的 ramdisk 源码，这里给出一个简单的 Makefile，该 Makefile 针对的是 make_request 版本，不过读者可以轻易将其修正为针对 request 的版本：

```

obj-m := ramhd_mkreq.o
KERNELDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
    rm -f *.o *.ko *.mod.*

```

11.2.3 ramdisk 的使用

我已经在 2.6.39 版本的 Linux 系统上对上述两个版本的 ramdisk 进行了编译，如果读者那边也一切正常，那么现在你的手头上应该有了两个 ramdisk 的内核模块：ramhd_mkreq.ko 和 ramhd_req.ko。ramhd_mkreq.ko 与 ramhd_req.ko 在使用方式上完全一样，这里以 ramhd_mkreq.ko 为例进行描述。

首先将 ramhd_mkreq.ko 加入内核：

```
root@AMDLinuxFGL:/home/dennis/book/chap11# insmod ramhd_mkreq.ko
```

insmod 执行后没有输出任何信息就再次回到 shell 命令接收状态，表明模块已经成功加入系

统。表象上看来似乎风平浪静，其实内核之中针对本次的 `insmod` 已经完成了相当多的操作。最简单的，我们尝试用 `dmesg` 看一下内核对此有何输出：

```
root@AMDLinuxFGL:~# dmesg
[152.762357] ramhda: detected capacity change from 0 to 8388608
[152.763343] ramhda:unknown partition table
[152.763750] ramhdb: detected capacity change from 0 to 8388608
[152.764859] ramhdb:unknown partition table
```

至于此处的“unknown partition table”云云，我们暂时也不关注，后面看了内核的机制之后自然就顿悟。然后再看看 `/dev` 目录下有没有添加什么：

```
root@AMDLinuxFGL:~# ls -l /dev/ram*
brw-rw---- 1 root disk 251, 0 May 29 11:36 /dev/ramhda
brw-rw---- 1 root disk 251, 4 May 29 11:36 /dev/ramhdb
```

显然有两个块设备 `ramhda` 和 `ramhdb` 被加入了系统中，主设备号是 251，次设备号分别是 0 和 4。除了这些以外，把一个块设备添加进系统当然还会导致有其他方面的变化，不过我打算把它留到讨论块设备的内核机制时再把它们放出来，彼时作为直接的现场数据，也许效果会更好。

现在对于块设备 `ramhda` 和 `ramhdb`，可以用 `fdisk` 给它分区，也可以直接在上面做个文件系统。我们打算在 `ramhda` 上做出两个主分区，此处略过 `fdisk` 的具体操作，当分区完成后，可以在 `/dev` 目录下发现两个新的块设备文件 `ramhda1` 和 `ramhda2`，次设备号分别是 1 和 2：

```
root@AMDLinuxFGL:~# ls -l /dev/ram*
brw-rw---- 1 root disk 251, 0 May 29 19:40 /dev/ramhda
brw-rw---- 1 root disk 251, 1 May 29 19:40 /dev/ramhda1
brw-rw---- 1 root disk 251, 2 May 29 19:40 /dev/ramhda2
brw-rw---- 1 root disk 251, 4 May 29 19:39 /dev/ramhdb
```

接下来用 `mkfs` 工具在 `/dev/ramhda1` 上做个 `ext3` 文件系统出来：

```
root@AMDLinuxFGL:~# mkfs.ext3 /dev/ramhda1
mke2fs 1.41.11 (14-Mar-2010)
Filesystem Label=
OS type:Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
```

```

976 inodes, 3896 blocks
194 blocks (4.98%) reserved for the super user
First data block=1
Maximum filesystem blocks=4194304
1 block group
8192 blocks per group, 8192 fragments per group
976 inodes per group

Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 29 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.

```

文件系统做好，就可以把/dev/ramhda1 设备 mount 到一个目录上，比如：

```

root@AMDLinuxFGL:~# mount /dev/ramhda1 /mnt
root@AMDLinuxFGL:~# ls -l /mnt
total 12
drwx----- 2 root root 12288 May 29 20:01 lost+found

```

现在读者可以在/mnt 下创建目录和建立新的文件，跟建立在实际硬盘设备上的文件系统没有任何不同。本章接下来的内容将围绕 ramdisk 程序来探讨块设备驱动程序的内核机制。

11.3 块设备号的注册与管理

对于块设备驱动程序而言，设备号的注册与管理由 register_blkdev 函数来完成，该函数的实现为：

```

<block/genhd.c>
-----
int register_blkdev(unsigned int major, const char *name)
{
    struct blk_major_name **n, *p;
    int index, ret = 0;

    mutex_lock(&block_class_lock);

    /* temporary */
    if (major == 0) {

```

```

        for (index = ARRAY_SIZE(major_names)-1; index > 0; index--) {
            if (major_names[index] == NULL)
                break;
        }
        if (index == 0) {
            printk("register_blkdev: failed to get major for %s\n",
                   name);
            ret = -EBUSY;
            goto out;
        }
        major = index;
        ret = major;
    }

    p = kmalloc(sizeof(struct blk_major_name), GFP_KERNEL);
    if (p == NULL) {
        ret = -ENOMEM;
        goto out;
    }

    p->major = major;
    strcpy(p->name, name, sizeof(p->name));
    p->next = NULL;
    index = major_to_index(major);

    for (n = &major_names[index]; *n; n = &(*n)->next) {
        if ((*n)->major == major)
            break;
    }
    if (!*n)
        *n = p;
    else
        ret = -EBUSY;

    if (ret < 0) {
        printk("register_blkdev: cannot get major %d for %s\n",
               major, name);
        kfree(p);
    }
}

out:
    mutex_unlock(&block_class_lock);
    return ret;
}

```

`register_blkdev` 函数的功能及实现方式和字符设备的 `register_chrdev_region` 函数非常类似，只不过它使用 `major_names` 数组来管理系统中的块设备号：

```
<block/genhd.c>
```

```
static struct blk_major_name {
    struct blk_major_name *next;
    int major;
    char name[16];
} *major_names[BLKDEV_MAJOR_HASH_SIZE];
```

宏 `BLKDEV_MAJOR_HASH_SIZE` 的值是 255。虽然 `register_blkdev` 函数的名称看起来像是往系统中添加一个块设备，但事实上它主要用来跟踪系统中块设备号的使用情况，以防止出现系统中多个块设备使用到同一个设备号的情况。而 `major_names` 数组在当前的 Linux 版本中也只是被 `blkdev_show` 函数所使用，后者在定义了 `CONFIG_PROC_FS` 的情况下用来在 `/proc/devices` 目录下显示所加入的块设备的名称。对应的块设备号注销函数则是 `unregister_blkdev`，所做的事情和 `register_blkdev` 恰好相反。所以单从功能上讲，块设备驱动程序可以通过调用 `register_blkdev` 函数来动态获得一个块设备的设备号，这么做的时候，传递给 `register_blkdev` 函数的第一个参数应该为 0（正如在前面 `ramdisk` 程序中所做的那样），如果函数成功分配了一个尚未使用的设备号，将通过函数返回值返回，若失败则返回一个错误码，动态分配的主设备号范围在 1~254 之间。如果块设备驱动程序事先知道要使用的主设备号，那么对 `register_blkdev` 函数的调用只是让系统能够跟踪到设备号的使用情况。

与 `register_blkdev` 相反，当驱动程序决定不再使用一个设备号时（这通常发生在驱动程序所在模块即将从系统中卸载时），应该调用 `unregister_blkdev` 函数将所占用的设备号释放掉，这样后续的设备才可以重新使用它。`unregister_blkdev` 的函数原型为：

```
<block/genhd.c>
```

```
void unregister_blkdev(unsigned int major, const char *name)
```

如果一个设备驱动程序不调用 `register_blkdev` 函数就直接使用设备号，那么它就变成了一个不遵守系统规则的破坏者，系统因无法跟踪设备号的使用情况，将导致潜在的设备号使用冲突的问题。无论如何，一个设计良好且安分守己的设备驱动程序没有理由不用 `register_blkdev` 函数来告之系统，它即将使用哪一个设备号。

11.4 block_device

内核用 `struct block_device` 来表示一个逻辑块设备对象，可以想象，这个数据结构的组成不会很简单。其定义如下：

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

```
struct block_device {
    dev_t          bd_dev; /* not a kdev_t - it's a search key */
    struct inode *  bd_inode; /* will die */
    struct super_block * bd_super;
```

```

int                bd_openers;
struct mutex       bd_mutex; /* open/close mutex */
struct list_head   bd_inodes;
void *             bd_claiming;
void *             bd_holder;
int                bd_holders;

#ifdef CONFIG_SYFS
    struct list_head bd_holder_list;
#endif

struct block_device * bd_contains;
unsigned             bd_block_size;
struct hd_struct *   bd_part;
/* number of times partitions within this device have been opened. */
unsigned            bd_part_count;
int                 bd_invalidated;
struct gendisk *     bd_disk;
struct list_head     bd_list;
unsigned long        bd_private;

/* The counter of freeze processes */
int                 bd_fsfreeze_count;
/* Mutex for freeze */
struct mutex         bd_fsfreeze_mutex;
};

```

上面的结构看起来貌似还比较直白，但其实里面嵌入了不少复杂的重量级的数据成员，比如 `bd_part` 和 `bd_disk`，前者的类型是 `struct hd_struct` 指针，后者则是个 `struct gendisk` 类型指针。

内核用这个数据结构既可以表示一个完整的逻辑块设备，也可以表示逻辑块设备中的某一个分区。当 `struct block_device` 表示一个完整的块设备时，其中的成员变量 `bd_part` 将指向该块设备的分区结构信息；当 `struct block_device` 表示块设备中的某一分区时，成员变量 `bd_contains` 指向该分区所在的块设备。当块设备（包括分区所对应的设备）所对应的设备文件被打开时，内核会创建一个 `block_device` 对象，这个过程将在“块设备文件节点”一节中具体讨论。`block_device` 在 Linux 内核中主要用来沟通文件系统组件与实际的块设备驱动程序，这种类似粘合剂的作用使得块设备驱动程序很少有与之直接打交道的机会。内核常常将 `block_device` 和一个所谓的“bdev”VFS 文件系统一起使用，后者只在内核空间使用，不会暴露到用户空间，关于这个文件系统的简单介绍，我将把它延后到本章的“块设备文件的打开”一节中。

11.5 struct gendisk

在内核空间，数据结构 `struct gendisk` 用来表示一个实际磁盘设备的抽象，这使得它与 `struct`

block_device 有了区分, gendisk 将直接被块设备驱动程序分配与操控, 它在内核中的定义是:

```
<include/linux/genhd.h>
-----
struct gendisk {
    /* major, first_minor and minors are input parameters only,
     * don't use directly. Use disk_devt() and disk_max_parts().
     */
    int major;           /* major number of driver */
    int first_minor;
    int minors;          /* maximum number of minors, =1 for
                        * disks that can't be partitioned. */

    char disk_name[DISK_NAME_LEN]; /* name of major driver */
    char *(*devnode)(struct gendisk *gd, mode_t *mode);

    unsigned int events; /* supported events */
    unsigned int async_events; /* async events, subset of all */

    /* Array of pointers to partitions indexed by partno.
     * Protected with matching bdev lock but stat and other
     * non-critical accesses use RCU. Always access through
     * helpers.
     */
    struct disk_part_tbl __rcu *part_tbl;
    struct hd_struct part0;

    const struct block_device_operations *fops;
    struct request_queue *queue;
    void *private_data;

    int flags;
    struct device *driverfs_dev; // FIXME: remove
    struct kobject *slave_dir;

    struct timer_rand_state *random;
    atomic_t sync_io; /* RAID */
    struct disk_events *ev;
#ifdef CONFIG_BLK_DEV_INTEGRITY
    struct blk_integrity *integrity;
#endif
    int node_id;
};
```

虽然尚没有讨论到实际的使用场景, 但是至少有两条线索可以暂时让我们猜测一下 struct block_device 与 struct gendisk 区别: 其一是两者定义的头文件不一样, struct block_device 定义在 include/linux/fs.h 中, 而 struct gendisk 定义在 include/linux/genhd.h, 所以 block_device

应该与文件系统部分关系更密切,而 `gendisk` 则被内核用来表示现实中一个通用磁盘设备的抽象,应该是块设备驱动程序的主要操作对象。另一条线索来自两者各自的成员组成,可以看到 `struct gendisk` 中有如下一些比较重要的成员:

```
int major
int first_minor
int minors
```

`major` 表示块设备的主设备号,用于指定当前设备对应的驱动程序。`first_minor` 和 `minors` 用于表示从设备号的范围,在对前面 `ramdisk` 设备进行分区时已经看到,从设备代表一个分区。`minors` 指定了当前 `gendisk` 对象所能包含的最大从设备的个数,如果该值为 1,意味着磁盘无法进行分区¹。

```
char disk_name[DISK_NAME_LEN]
```

当前块设备对象名称,作为块设备所对表示的内核对象名称而存在,显示于 `sysfs` 文件系统中。

```
struct disk_part_tbl *part_tbl
```

表示 `gendisk` 磁盘对象的分区表信息。在其成员中, `part` 是一个容纳 `struct hd_struct` 指针,而每一个 `struct hd_struct` 对象则代表当前磁盘上的一个分区。

```
const struct block_device_operations *fops
```

表示针对当前 `gendisk` 对象上的一组操作集合,在后续部分会看到这个结构的定义。

```
struct request_queue *queue
```

当前的 `gendisk` 对象所代表的块设备上的 I/O 请求队列。

```
struct hd_struct part0
```

表示当前块设备的第一分区,如果设备没有分区则指代整个设备。

```
void *private_data
```

一个指向驱动程序的私有数据的指针,内核的块子系统不会修改它。

块设备驱动程序需要负责产生 `gendisk` 对象,并初始化其中相关成员,关于这一过程,将在

¹ `minors=1` 常常导致在用 `fdisk` 分区时出现类似 “WARNING: Re-reading the partition table failed with error 22: Invalid argument. The kernel still uses the old table. The new table will be used at the next reboot...” 这样的警告信息。如果块设备驱动程序在 `alloc_disk` 时用 1 作为参数,将导致 `minors=1`,对应的块设备理论上无法分区。

后面继续讨论。gendisk 可以表示一个已经分区的磁盘，也可以表示一个未分区的磁盘。当驱动程序调用 add_disk 将一个 gendisk 对象加入系统时，内核将视具体情况决定是否扫描该对象上的分区信息，关于这方面的更多细节将在本章的 add_disk 部分予以讨论。

11.6 struct hd_struct

内核使用该结构来表示块设备上的某一分区信息，其定义如下：

```
<include/linux/genhd.h>
-----
struct hd_struct {
    sector_t start_sect;
    sector_t nr_sects;
    sector_t alignment_offset;
    unsigned int discard_alignment;
    struct device __dev;
    struct kobject *holder_dir;
    int policy, partno;
    struct partition_meta_info *info;
#ifdef CONFIG_FAIL_MAKE_REQUEST
    int make_it_fail;
#endif
    unsigned long stamp;
    atomic_t in_flight[2];
#ifdef CONFIG_SMP
    struct disk_stats __percpu *dkstats;
#else
    struct disk_stats dkstats;
#endif
    atomic_t ref;
    struct rcu_head rcu_head;
};
```

其中有 start_sect、nr_sects 和 partno 成员，分别表示当前分区的起始扇区、分区的大小即扇区数量，以及分区编号。还包含一个 struct device __dev 成员，这意味着在内核中磁盘上的一个分区也将被视为一个设备。

11.7 用 alloc_disk 分配 gendisk 对象

当设备驱动程序需要产生一个 gendisk 对象时，应该调用 alloc_disk 函数，该函数除了完成必要的动态分配一个 gendisk 对象外，还对其进行一些初始化，其代码如下：

```
<block/genhd.c>
```

```
struct gendisk *alloc_disk(int minors)
{
    return alloc_disk_node(minors, -1);
}

struct gendisk *alloc_disk_node(int minors, int node_id)
{
    struct gendisk *disk;

    disk = kmalloc_node(sizeof(struct gendisk),
                        GFP_KERNEL | __GFP_ZERO, node_id);
    if (disk) {
        if (!init_part_stats(&disk->part0)) {
            kfree(disk);
            return NULL;
        }
        disk->node_id = node_id;
        if (disk_expand_part_tbl(disk, 0)) {
            free_part_stats(&disk->part0);
            kfree(disk);
            return NULL;
        }
        disk->part_tbl->part[0] = &disk->part0;

        hd_ref_init(&disk->part0);

        disk->minors = minors;
        rand_initialize_disk(disk);
        disk_to_dev(disk)->class = &block_class;
        disk_to_dev(disk)->type = &disk_type;
        device_initialize(disk_to_dev(disk));
    }
    return disk;
}
```

`alloc_disk` 的实际工作是在 `alloc_disk_node` 中完成的, 后者先通过 `kmalloc` 分配一个 `gendisk` 对象, 在 `disk_expand_part_tbl` 函数中, 将为 `gendisk` 对象分配由其成员 `part_tbl` 指向的空间, 这个空间用来容纳当前 `gendisk` 对象的分区信息, 因为块设备的分区可以动态增加或者删除, 因此 `part_tbl` 所指向的空间大小也会相应变化。随后 `disk_part_tbl` 中的 `part[0]` 将指向 `gendisk` 对象中 `part0` 成员所在空间, `part0` 是一个 `struct hd_struct` 类型, 用来标识一个分区, 内核同时也用它标识整块设备。`alloc_disk` 所分配的 `gendisk` 对象和 `disk_part_tbl` 对象间的关系如图 11-3 所示:

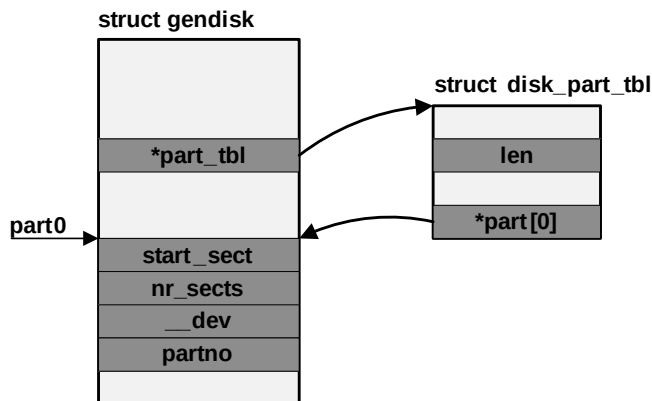


图 11-3 alloc_disk 分配的 gendisk 对象和分区表数据空间

之后，调用 `alloc_disk` 时传入的参数 `minors` 将被赋予 `disk->minors`，表示当前 `gendisk` 对象所允许存在的最大分区数。

接下来的代码是完成 Linux 设备驱动模型所要求的工作，其中我们看到内核将 `part0.__dev` 用来代表当前 `gendisk` 所属的设备，这体现在 `disk_to_dev(disk)` 代码中，后者用来获得 `disk` 对象内嵌的设备对象，这个宏的定义为：

```
<include/linux/genhd.h>
-----
#define disk_to_dev(disk) (&(disk)->part0.__dev)
```

所以，事实上内核将 `disk->part0` 视为 `disk` 所属设备的真正代表。

与 `alloc_disk` 所做的工作相反，当设备驱动程序不再需要 `alloc_disk` 分配出来的 `gendisk` 对象时，应该调用 `del_gendisk` 予以注销。`del_gendisk` 的函数原型为：

```
void del_gendisk(struct gendisk *disk);
```

11.8 向系统添加一个块设备 add_disk

与字符设备一样，内核用设备号来标识一个块设备，通常主设备号对应一个驱动程序，次设备号对应该驱动程序所管理块设备上的一个分区。在 Linux 系统下，磁盘上的一个独立分区被看做一个设备，对应 `/dev` 目录下的一个设备节点。Linux 下设备号的数据类型为 `dev_t`，无论字符设备还是块设备，这个类型都是适用的。

与字符设备驱动程序调用 `cdev_add` 向系统注册设备一样，块设备驱动程序需要调用 `add_disk` 函数来向系统注册一个磁盘设备。但是相比于 `cdev_add`，`add_disk` 要复杂得多。为了更好地表述这一过程，这里先做个限定：在随后的讨论中，将用 `add_disk` 来把一个尚未分区的设备加到系统中，当驱动程序这么做时，内核将试图读取该磁盘设备上的分区信

息，对每个有效分区形成一个驱动模型中设备 `device` 的对象，并通过 `device_add` 加到系统中，但此时这些分区并不会产生对应的 `block_device` 对象，直到分区设备被打开。不过由于要加入的设备尚未产生有效分区，所以在 `add_disk` 时系统将无法获得分区信息，这也正是前面在加载 `ramhd_mkreq.ko` 模块后 `dmesg` 命令显示如下信息的原因：

```
[152.763343] ramhda:unknown partition table
```

现在来仔细考察 `add_disk`，其核心代码如下：

<block/genhd.c>

```
void add_disk(struct gendisk *disk)
{
    struct backing_dev_info *bdi;
    dev_t devt;
    int retval;

    /* minors == 0 indicates to use ext devt from part0 and should
     * be accompanied with EXT_DEVT flag. Make sure all
     * parameters make sense.
     */
    WARN_ON(disk->minors && !(disk->major || disk->first_minor));
    WARN_ON(!disk->minors && !(disk->flags & GENHD_FL_EXT_DEVT));

    disk->flags |= GENHD_FL_UP;

    retval = blk_alloc_devt(&disk->part0, &devt);
    if (retval) {
        WARN_ON(1);
        return;
    }
    disk_to_dev(disk)->devt = devt;

    /* ->major and ->first_minor aren't supposed to be
     * dereferenced from here on, but set them just in case.
     */
    disk->major = MAJOR(devt);
    disk->first_minor = MINOR(devt);

    /* Register BDI before referencing it from bdev */
    bdi = &disk->queue->backing_dev_info;
    bdi_register_dev(bdi, disk_devt(disk));

    blk_register_region(disk_devt(disk), disk->minors, NULL,
                        exact_match, exact_lock, disk);
    register_disk(disk);
    blk_register_queue(disk);
}
```

```

        retval = sysfs_create_link(&disk_to_dev(disk)->kobj, &bdi->dev->kobj,
                                   "bdi");
        WARN_ON(retval);

        disk_add_events(disk);
    }

```

先看该函数的参数，这是一个 `struct gendisk` 类型的指针，设备驱动程序通过调用 `alloc_disk` 生成该对象，在进行必要的初始化后，将该对象的地址作为参数调用 `add_disk` 将一个块设备添加到系统，一旦一个块设备对象 `gendisk` 被加入系统，也就向系统宣示了它的存在，内核中的块子系统将可以操作它，所以这必是在驱动程序最后的阶段当所有关于 `gendisk` 的初始化工作都完成之后才可以进行。

`add_disk` 按照不同情况有几条执行路径，这里我打算以之前 `ramdisk` 例子所展示的情形（在例子中，调用 `add_disk` 加入一个未经分区的 `disk`，`disk->minors=4`）讨论其中一条路径，它也是最常见的一条路径。在此基础上，为了使描述更为简单明了，将 `add_disk` 按照逻辑上的功能分解成如下几个部分进行讨论：

○ `blk_alloc_devt` 函数

函数中的 `blk_alloc_devt` 用来生成当前块设备的设备号，它其实只是使用宏 `MKDEV` 来组合一个 `dev_t` 类型的数值，因此设备号的主次设备号应该在调用 `add_disk` 之前就应该被分配好，或者使用静态指定的方法，或者使用动态分配的方法（通过调用 `register_blkdev` 函数）。换句话说，`gendisk` 对象中的 `major` 和 `first_minor` 成员应该在 `add_disk` 之前就已经赋过值了。组合后的设备号由 `devt` 带回。

○ `blk_register_region` 函数

接下来的 `blk_register_region` 是个很重要的调用，经过这个函数处理后，当前的块设备才真正进入系统的视野，意味着系统已经可以发现这个新加入的设备，更具体地，这其中的媒介是一个类型为 `struct kobj_map` 的全局变量 `bdev_map`。`blk_register_region` 函数要完成的功能对应字符型设备驱动程序中的 `cdev_add` 函数，它会把当前 `gendisk` 对象加入到 `bdev_map` 中，如图 11-4 所示。

在“字符设备驱动程序”一章中已经详细讨论过 `cdev_add` 函数的实现原理，此处不再重复。

○ `register_disk` 函数

紧跟在 `blk_register_region` 函数之后的是 `register_disk` 函数，它要完成的功能也非比寻常，其核心代码如下：

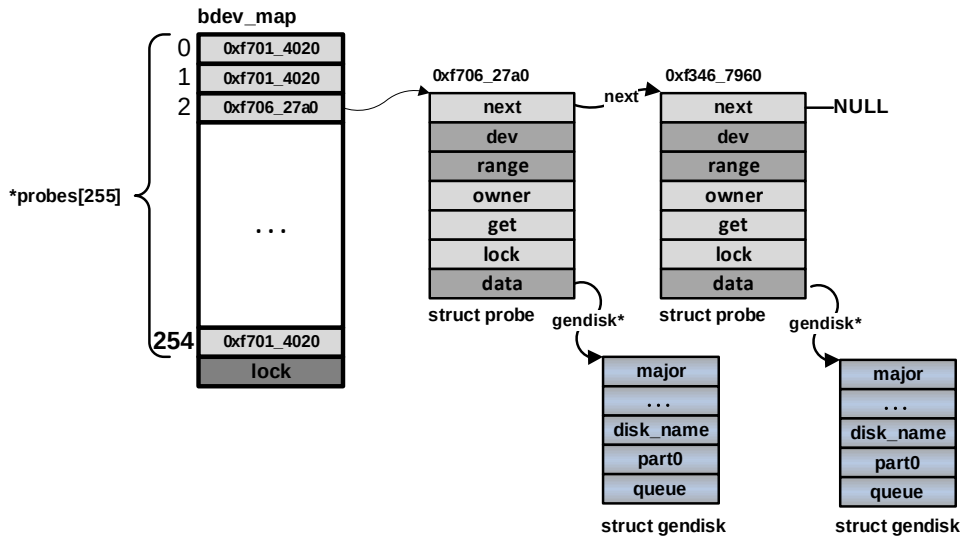


图 11-4 将 gendisk 对象加到 bdev_map 中

```
<fs/partitions/check.c>
```

```
void register_disk(struct gendisk *disk)
{
    struct device *ddev = disk_to_dev(disk);
    struct block_device *bdev;
    struct disk_part_iter piter;
    struct hd_struct *part;
    int err;

    ddev->parent = disk->driverfs_dev;

    dev_set_name(ddev, disk->disk_name);

    /* delay uevents, until we scanned partition table */
    dev_set_uevent_suppress(ddev, 1);

    if (device_add(ddev))
        return;
#ifdef CONFIG_SYSFS_DEPRECATED
    err = sysfs_create_link(block_depr, &ddev->kobj,
                           kobject_name(&ddev->kobj));
    if (err) {
        device_del(ddev);
        return;
    }
#endif
    disk->part0.holder_dir = kobject_create_and_add("holders", &ddev->kobj);
    disk->slave_dir = kobject_create_and_add("slaves", &ddev->kobj);
}
```



```

    /* No minors to use for partitions */
    if (!disk_partitionable(disk))
        goto exit;

    /* No such device (e.g., media were just removed) */
    if (!get_capacity(disk))
        goto exit;

    bdev = bdget_disk(disk, 0);
    if (!bdev)
        goto exit;

    bdev->bd_invalidated = 1;
    err = blkdev_get(bdev, FMODE_READ);
    if (err < 0)
        goto exit;
    blkdev_put(bdev, FMODE_READ);

exit:
    /* announce disk after possible partitions are created */
    dev_set_uevent_suppress(ddev, 0);
    kobject_uevent(&ddev->kobj, KOBJ_ADD);

    /* announce possible partitions */
    disk_part_iter_init(&piter, disk, 0);
    while ((part = disk_part_iter_next(&piter)))
        kobject_uevent(&part_to_dev(part)->kobj, KOBJ_ADD);
    disk_part_iter_exit(&piter);
}

```

函数前半段实现的是 Linux 设备驱动模型中设备对象的相关操作,核心是对 `device_add(ddev)` 的调用,这将导致当前的块设备在 `/dev` 目录下生成一个新的设备节点文件,比如在前面的 `ramdisk` 例子中 `insmod ramhd_mkreq.ko` 之后出现的 `/dev/ramhda` 与 `/dev/ramhdb`,就是在这里产生的。其中大多数函数的细节都曾在“Linux 设备驱动模型”一章中讨论过。

在实现了设备驱动模型所要求的功能之后,接下来的内容是驱动程序员比较感兴趣的,因为它们跟设备驱动程序关系更为密切。其中 `disk_partitionable` 用来检测当前块设备是否有分区,检测的依据是当前设备的 `disk->minors` 是否大于 1,如是则继续往下进行,否则表明这是个不分区 的磁盘设备,代码将直接进入退出路径。`get_capacity` 则是返回 `disk->part0.nr_sects`, `ramdisk` 例子在 `add_disk` 之前有对 `set_capacity()` 的调用,所以这两个函数都不会返回 0,于是 `bdget_disk` 函数将被调用,产生出一个新的 `block_device` 对象,这个过程发生在 `bdget_disk()`→`bdget()`→`iget5_locked()`调用链中。

先看 bdget 在源码中的实现：

```
<fs/block_dev.c>
-----
struct block_device *bdget(dev_t dev)
{
    struct block_device *bdev;
    struct inode *inode;

    inode = iget5_locked(blockdev_superblock, hash(dev),
                        bdev_test, bdev_set, &dev);

    if (!inode)
        return NULL;

    bdev = &BDEV_I(inode)->bdev;
    if (inode->i_state & I_NEW) {
        bdev->bd_contains = NULL;
        bdev->bd_inode = inode;
        bdev->bd_block_size = (1 << inode->i_blkbits);
        bdev->bd_part_count = 0;
        bdev->bd_invalidated = 0;
        inode->i_mode = S_IFBLK;
        inode->i_rdev = dev;
        inode->i_bdev = bdev;
        inode->i_data.a_ops = &def_blk_aops;
        mapping_set_gfp_mask(&inode->i_data, GFP_USER);
        inode->i_data.backing_dev_info = &default_backing_dev_info;
        spin_lock(&bdev_lock);
        list_add(&bdev->bd_list, &all_bdevs);
        spin_unlock(&bdev_lock);
        unlock_new_inode(inode);
    }
    return bdev;
}
```

在 add_disk 的调用上下文中，由于在 iget5_locked 中 “bdev” 文件系统的 inode 节点尚未产生，所以实际上 iget5_locked 将通过 bdev_alloc_inode 函数分配一个 struct bdev_inode 类型的对象，调用链是 iget5_locked()→get_new_inode()→alloc_inode()，alloc_inode 最终会调用到 bdev_alloc_inode：

```
<fs/block_dev.c>
-----
static struct inode *bdev_alloc_inode(struct super_block *sb)
{
    struct bdev_inode *ei = kmem_cache_alloc(bdev_cachep, GFP_KERNEL);
    if (!ei)
        return NULL;
}
```

```

    return &ei->vfs_inode;
}

```

这里 struct bdev_inode 的定义为：

```

<fs/block_dev.c>
struct bdev_inode {
    struct block_device bdev;
    struct inode vfs_inode;
};

```

所以通过 struct bdev_inode 结构，bdev_alloc_inode 函数在分配了一个 struct inode 对象的同时，也分配了一个 struct block_device 对象，虽然返回的是 inode，但是通过 container_of 宏很容易得到 struct bdev_inode 对象的指针。vfs_inode 这个 inode 的分配因为来自“bdev”文件系统的超级块，所以应该隶属于“bdev”文件系统（关于这个文件系统，将在本章后续的“块设备文件的打开”一节中予以讨论）。vfs_inode 在刚从 iget5_locked 中分配出来的时候，i_state 成员上是带有 I_NEW 标识的，所以 bdget 函数中的 if (inode->i_state & I_NEW) 条件是满足的，在这个条件块里，直到 unlock_new_inode(inode) 语句执行完，i_state 上的 I_NEW 标志才会被清除掉。

bdget 函数虽然返回的是 struct block_device 对象，但是 iget5_locked 中分配的 vfs_inode 会被记录到 struct block_device 对象的 bd_inode 成员中。图 11-5 示意了 iget5_locked 函数产生的 struct bdev_inode 对象中 bdev 与 vfs_inode 之间的关联：

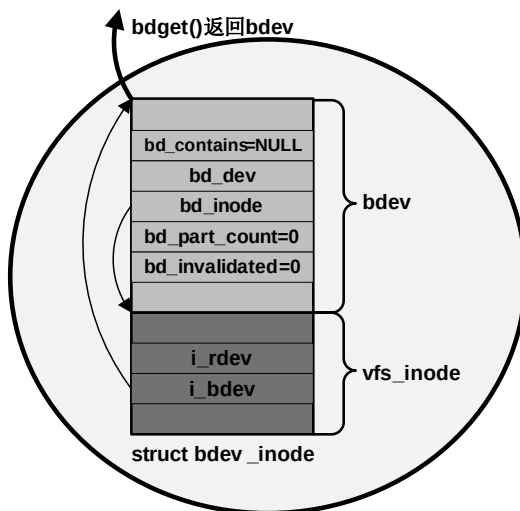


图 11-5 bdget_disk 产生的 bdev_inode 对象

另外一个需要注意的地方是，iget5_locked() 函数在执行时，通过传入的 bdev_set 函数将块设备文件节点中的设备号放在了新产生的 block_device 对象的 bd_dev 成员中：

```
<fs/block_dev.c>
```

```
static int bdev_set(struct inode *inode, void *data)
{
    BDEV_I(inode)->bdev.bd_dev = *(dev_t *)data;
    return 0;
}
```

当 `bdget_disk` 函数将新生成的 `block_device` 对象指针 `bdev` 返回后, `bdev->bd_invalidated` 被重置为 1, 这使得内核将有机会对新加入的 `gendisk` 设备进行分区扫描。

`register_disk` 函数中接下来还有一个很重要的调用 `blkdev_get(bdev, FMODE_READ)`, 后者实际上调用的是 `__blkdev_get(bdev, FMODE_READ, 0)`。这个函数很长, 下面根据其在 `ramdisk` 例子的上下文中执行的逻辑顺序解释一下该函数的主要功能:

函数首先调用 `disk = get_gendisk(bdev->bd_dev, &partno)` 来获得 `gendisk` 对象, 第一个参数 `bdev->bd_dev` 是 `block_device` 对象中记录下的设备号, `get_gendisk` 利用该设备号调用 `kobj_lookup` 函数在 `bdev_map` 中查找 `gendisk` 对象, 读者可以参考图 11-4。函数将把在 `bdev_map` 中找到的 `gendisk` 对象指针 `disk` 返回, 同时 `partno=0`。

在当前由 `add_disk` 发起的这个调用链中, `bdev->bd_openers` 显然是 0, 表明 `bdev` 所在的逻辑设备尚未被 `open` 过:

```
if (!bdev->bd_openers) {
    bdev->bd_disk = disk;
    bdev->bd_contains = bdev;
```

`bdev->bd_disk` 此时指向了正被 `add_disk` 加入系统的 `gendisk` 对象, `bdev->bd_contains` 指向自身。

在这条路径中, `if (!partno)` 显然是满足的, `bdev->bd_part = disk_get_part(disk, partno)` 的执行将导致 `bdev->bd_part` 指向 `disk->part0`。之后, 如果设备驱动程序的 `block_device_operations` 对象中实现了 `open` 函数, 那么内核将调用它:

```
if (disk->fops->open) {
    ret = disk->fops->open(bdev, mode);
```

在我们的 `ramdisk` 例子中, `add_disk` 的调用将使得 `ramhd_open` 函数被调用, `ramhd_open` 直接返回 0。随后有个重要的调用是 `rescan_partitions`, 它将扫描当前 `gendisk` 设备上的分区信息, 调用以下面的方式发起:

```
if (bdev->bd_invalidated && (!ret || ret == -ENOMEDIUM))
    rescan_partitions(disk, bdev);
```

`rescan_partitions` 函数扫描 `disk` 设备上分区的原理是, 将识别不同分区的函数指针放在一个 `check_part` 数组中:

```
<fs/partitions/check.c>
-----
static int (*check_part[])(struct parsed_partitions *) = {
    ...
    #ifdef CONFIG_MSDOS_PARTITION
        msdos_partition,
    #endif
    ...
    #ifdef CONFIG_IBM_PARTITION
        ibm_partition,
    #endif
    ...
    NULL
};
```

rescan_partitions 函数在一个 while 循环中依次调用 check_part 中的函数去判断当前的 gendisk 对象上是否存在有效分区，在我们的 ramdisk 例子中，因为 vmalloc 分配出来的空间中不存在任何有效分区，所以 rescan_partitions 将在无法发现分区的情况下打印出“unknown partition table”这样的信息。

如果当前要通过 add_disk 加入系统的块设备上存在有效的分区，会发生什么情况呢？rescan_partitions 函数会调用 disk_expand_part_tbl 来扩展 gendisk->part_tbl 所指向的空间，然后通过 add_partition() 向系统增加该分区设备，所以 add_partition 会首先分配一个 struct hd_struct 类型的空间 p 来容纳该分区的信息（分区的起始扇区、大小、编号等），然后把 p 记录到 gendisk 的 part_tbl 所指向的空间内，假设扫描到的分区编号为 partno，那么就是 gendisk->part_tbl->part[partno] = p。因为内核把分区视为独立的设备，所以分区有属于自己的 device 结构，rescan_partitions 函数最终会调用 device_add 将分区设备加入系统，导致在 /dev 目录下生成类似 /dev/ramhda1 和 /dev/ramhda2 这样的设备文件。在 add_disk 发起的调用链中，因为通过 rescan_partitions 发现的分区设备只是会被 device_add 加入系统，所以对新发现的分区，并没有对应的新的 block_device 对象产生。

○ blk_register_queue 函数

add_disk 中在 register_disk 调用之后的另外一个调用是 blk_register_queue，用来对当前的 disk 对象请求队列进行必要的初始化。这个函数虽然表面上看与块设备的重点请求队列关系密切，但是其代码要完成的功能还只限于 Linux 设备模型中与 sysfs 相关的操作，对于设备驱动程序而言，并没有特别要注意的地方。

所以到现在我们看到 add_disk 的主要功能是把一个通用磁盘对象 gendisk 加到 bdev_map 中，同时在 /dev 下动态生成一个设备节点（这由 device_add 完成），然后还动态生成一个 block_device 对象和一个隶属于“bdev”文件系统的 inode，这两个动态对象由于是绑定在同一个数据结构 struct bdev_inode 中，所以通过“bdev”文件系统的 inode 可以很容易地获得 block_device 对象。这样在打开一个块设备文件节点时，通过查找这个“bdev”文件系

统的 `inode` 就可以获得对应的 `block_device` 对象。另外 `add_disk` 在向系统中加入一个磁盘对象时，如果该磁盘对象拥有分区，那么 `add_disk` 还会扫描分区信息，对于每一个识别出的分区，都会产生一个新的 `device` 出来，同时会通过调用 `device_add` 把分区对应的设备加入系统。内核在通过 `add_partition` 增加分区时，并不会伴随有新的 `block_device` 对象产生，这与打开一个分区设备不同。

稍作总结，`add_disk` 函数在把 `gendisk` 对象加入系统时，会产生一个新的 `block_device` 对象与之对应，如果用图来表示上述 `add_disk` 执行期间所建立的 `gendisk`、`block_device` 和 `hd_struct` 之间的关系，大体上如图 11-6 所示，`block_device` 对象的 `bd_part` 指向 `gendisk` 的 `part0` 成员所在空间，`block_device` 的 `bd_disk` 指向 `gendisk` 对象：

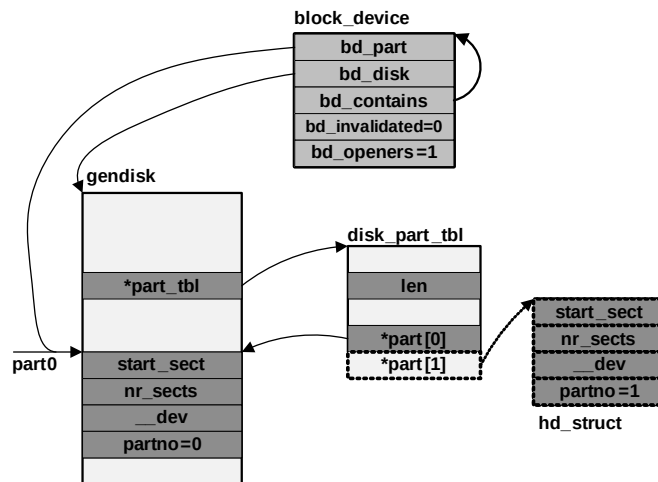


图 11-6 `add_disk` 产生的 `block_device` 与 `gendisk` 等的关系

图中 `disk_part_tbl` 空间中的虚线框表示，如果 `rescan_partitions` 函数在 `gendisk` 所表示的块设备上发现有效分区，那么新的分区将会导致一个新的 `hd_struct` 对象产生，同时 `add_disk` 会将 `gendisk->part_tbl->part[partno]` 指向这个新产生的 `hd_struct` 对象，新分区会作为一个独立的 `device` 加到 `/dev` 目录下。在新的分区设备被打开前，不会有对应的 `block_device` 对象产生。将很快在后续的“块设备文件的打开”一节中看到打开一个块设备时内核的行为。

11.9 block_device_operations

对于字符设备而言，内核为其定义的一整套操作集包含在数据结构 `file_operations` 中。而对于块设备而言，对应的数据结构则是 `block_device_operations`：

```
<include/linux/blkdev.h>
-----
struct block_device_operations {
    int (*open) (struct block_device *, fmode_t);
```

```

int (*release) (struct gendisk *, fmode_t);
int (*locked_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
int (*direct_access) (struct block_device *, sector_t, void **, unsigned long *);
int (*media_changed) (struct gendisk *);
void (*unlock_native_capacity) (struct gendisk *);
int (*revalidate_disk) (struct gendisk *);
int (*getgeo)(struct block_device *, struct hd_geometry *);
/* this callback is with swap_lock and sometimes page table lock held */
void (*swap_slot_free_notify) (struct block_device *, unsigned long);
struct module *owner;
};

```

相对于字符设备的 `file_operations` 结构，对于块设备驱动程序而言，`block_device_operations` 的作用已经明显被弱化了，在前面 `ramdisk` 的实现当中，我们很少去实现该结构中所定义的函数指针。细心的读者也许已经发现和字符设备的 `file_operations` 结构的一个很大的区别是 `block_device_operations` 结构中没有类似的 `read` 和 `write` 函数，块设备当然不可能没有对应的读/写函数，否则也就没有多少存在的价值了，只不过块设备的读/写操作由另一个重要的组件读/写请求队列来完成，这种设计上的差异也体现在了块设备与字符设备的使用模式上，相对于需要和系统进行大量数据传输的块设备，字符设备和系统的数据交互往往很少，因此块设备需要有重新的设计来优化数据读/写这部分的性能。另一方面，应用程序在使用字符设备驱动程序时，通过 `file_operations` 作中转，这个调用过程相当清晰而直白，但是对于块设备而言，它主要被系统中的文件系统组件所使用，一般的用户程序很少会像使用字符设备那样使用块设备。

11.10 块设备文件的打开

现在我们打算从块设备文件节点的生成过程作为切入点来讨论打开一个块设备文件时内核所做的事情，同时我们也将看到此过程中 `block_device` 对象在系统中的用途。虽然在设备驱动程序中，块设备的打开函数功能已经被弱化，但是通过探讨块设备文件的打开过程，可以加深对块设备驱动程序在内核中的相关技术细节的了解。这期间有些代码和 `add_disk` 是重复的，但是两者的执行路径有时会有不同，为方便读者阅读，笔者会把前面出现过的代码尽量精简地摘录下来。

先来看块设备文件节点的生成。在支持动态设备节点生成的系统中（现在内核通过支持“`devtmpfs`”设备文件系统²在 `/dev` 目录下动态生成设备节点，本章以这种情况为讨论主线），

² 这是个基于 RAM 的文件系统，在内核中作为一棵独立的 VFS 树而存在。内核在初始化期间，如果检测到一些设备，比如 PCI 的扫描过程，会在该 VFS 上增加新的设备节点。“`devtmpfs`”文件系统最终会 `mount` 到根文件系统的 `/dev` 目录上，使得用户空间得以访问到它。

`device_add` 将在 `/dev` 目录下生成一个块设备文件节点，正如 2.6 节“设备文件节点的生成”中介绍的那样，一个新加入的块设备，比如 `ramdisk` 例子中的“`ramhda`”，将会在 `/dev` 下生成 `/dev/ramhda` 设备节点，这个节点的 `inode` 由“`devtmpfs`”文件系统负责生成，在生成该 `inode` 时 `init_special_inode` 函数会被调用，正如在第 2 章中看到的那样，`init_special_inode` 主要给 `inode->i_fop` 和 `inode->i_rdev` 赋值：

<fs/inode.c>

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {
        inode->i_fop = &def_chr_fops;
        inode->i_rdev = rdev;
    } else if (S_ISBLK(mode)) {
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = rdev;
    }
    ...
}
```

所以，如果是块设备，那么 `inode->i_fop = &def_blk_fops`，`def_blk_fops` 的定义如下：

<fs/block_dev.c>

```
const struct file_operations def_blk_fops = {
    .open      = blkdev_open,
    .release   = blkdev_close,
    .llseek    = block_llseek,
    .read      = do_sync_read,
    .write     = do_sync_write,
    .aio_read  = generic_file_aio_read,
    .aio_write = blkdev_aio_write,
    .mmap      = generic_file_mmap,
    .fsync     = blkdev_fsync,
    .unlocked_ioctl = block_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl = compat_blkdev_ioctl,
#endif
    .splice_read = generic_file_splice_read,
    .splice_write = generic_file_splice_write,
};
```

理论上 `def_blk_fops` 结构中所定义的成员函数都应该能在 `block_device_operations` 对象中找到对应的项，但是稍稍比较一下就会发现，两者之间还有较大的差异，原因是内核中的块子系统截留了 `def_blk_fops` 中的一些函数调用而没有将其下传到驱动程序中，比如 `read/write` 和 `aio_read/aio_write` 等。

言归正传，当用户程序打开一个块设备文件时，比如 `/dev/ramhda`，`blkdev_open` 将被调用，该函数的原型是：

```
int blkdev_open(struct inode * inode, struct file * filp);
```

实际传入的参数 `inode` 由 “`devtmpfs`” 设备文件系统产生³。`blkdev_open` 的源码相对比较冗长，这里不再摘录，对这个函数我们感兴趣的地方有以下几点：

(1) 函数会获得一个 `struct block_device` 对象（该对象是在 `add_disk` 函数调用时动态生成的），这个过程发生在 `blkdev_open()`→`bd_acquire()` 函数调用链中：

<fs/block_dev.c>

```
static int blkdev_open(struct inode *inode, struct file *filp)
{
    struct block_device *bdev;
    ...
    bdev = bd_acquire(inode);
    ...
}
```

`bd_acquire` 函数的具体实现细节并不如想象中的那样直白，要讲清楚这个过程还是有些吃力，因为其中牵涉到另外一个基于 RAM 的文件系统 “`bdev`”，这个文件系统的性质是 VFS，并没有挂载到用户空间的根文件系统上，所以用户空间无法看到这棵树，内核源码称之为 “`pseudo-fs`”。下面是内核给它的定义：

<fs/block_dev.c>

```
static struct file_system_type bd_type = {
    .name      = "bdev",
    .get_sb    = bd_get_sb,
    .kill_sb   = kill_anon_super,
};
```

因为 `bd_acquire` 函数中会用到该文件系统的功能，所以在继续 `bd_acquire` 函数之前读者需要一些 “`bdev`” 文件系统的相关背景，该文件系统的初始化发生在 `bdev_cache_init` 函数中：

<fs/block_dev.c>

```
void __init bdev_cache_init(void)
{
    int err;
    struct vfsmount *bd_mnt;
```

³ 这一节中关于 `inode` 部分的讨论比较容易让人困惑，因为有两个类型的 `inode` 会频繁出现：一个是对应 `/dev` 下设备文件节点的 `inode`，该 `inode` 属于 “`devtmpfs`” 文件系统范畴；另一个会和 `block_device` 一起产生，该 `inode` 属于 “`bdev`” 伪文件系统。读者可能要在这些地方花点心思。

```

bdev_cachep = kmem_cache_create("bdev_cache", sizeof(struct bdev_inode),
                                0, (SLAB_HWCACHE_ALIGN|SLAB_RECLAIM_ACCOUNT|
                                    SLAB_MEM_SPREAD|SLAB_PANIC), init_once);
err = register_filesystem(&bd_type);
if (err)
    panic("Cannot register bdev pseudo-fs");
bd_mnt = kern_mount(&bd_type);
if (IS_ERR(bd_mnt))
    panic("Cannot create bdev pseudo-fs");
...
blockdev_superblock = bd_mnt->mnt_sb; /* For writeback */
}

```

bdev_cachep 是一个 kmem_cache 分配池，struct block_device 对象的分配就出自这里。bd_mnt=kern_mount(&bd_type)用来产生“bdev”文件系统 mnt 结构，意味着它在内核中是一棵独立的 VFS 文件树。blockdev_superblock 保存着该文件系统的超级块，blockdev_superblock = bd_mnt->mnt_sb，bd_mnt->mnt_sb 所指向的超级块显然应该由 bd_type 对象的成员 bd_get_sb 来分配，这一过程发生在 kern_mount 函数中，后者先分配 mnt 对象，然后分配文件系统超级块，接下来依次是分配“bdev”文件系统根节点所对应的 inode 和 dentry，这其中顺序很重要，超级块先于 inode 被分配出来，这样分配 inode 就可以通过先期分配出的超级块对象中的 sb->s_op->alloc_inode 函数来完成。

对于“bdev”文件系统，sb->s_op 操作集的实例化来自于内核定义的一个 struct super_operations 对象：

<fs/block_dev.c>

```

static const struct super_operations bdev_sops = {
    .statfs = simple_statfs,
    .alloc_inode = bdev_alloc_inode,
    .destroy_inode = bdev_destroy_inode,
    .drop_inode = generic_delete_inode,
    .clear_inode = bdev_clear_inode,
};

```

对于该对象，我们目前只关注.alloc_inode = bdev_alloc_inode，它将用来分配“bdev”文件系统的 inode 节点。

再回到 bd_acquire 函数，在打开/dev/ramhda 的上下文中，我们知道它要获得 add_disk 产生的 struct block_device 对象，具体代码发生在 bd_acquire() bdget() iget5_locked()调用链中，当链中的 bdget 函数被调用时，实际传入的参数是一个设备号 dev_t dev，这个参数来自/dev/ramhda 在“devtmpfs”文件系统下对应的 inode->i_rdev。注意此时讨论的内容是在打开 ramdisk 例子所产生的/dev/ramhda 的上下文中，这意味着对应/dev/ramhda 的属于“bdev”

文件系统的 inode 已经产生（由前面讨论的 `add_disk` 调用产生），所以 `iget5_locked` 函数的代码在执行时，通过内部的 `find_inode` 调用将得到一个隶属于“bdev”文件系统的 inode，这个 inode 对象实际上是 `struct bdev_inode` 结构中的 `vfs_inode`。

但是现在假如打开的是隶属于 `ramhda` 的一个分区设备 `ramhda1`，从前面 `add_disk` 的讨论可知，此时 `ramhda1` 尚未在“bdev”文件系统中产生 `vfs_inode` 及 `block_device` 对象，所以 `bdev_alloc_inode` 函数在这种情况下将被调用，为当前的 `ramhda1` 分区设备节点产生一个新的 `block_device` 和 `vfs_inode` 对象。

因为 `block_device` 实例与 `vfs_inode` 实例绑定在同一个结构体对象中，所以由 `vfs_inode` 可以得到 `block_device` 对象，内核通过宏 `BDEV_I` 来完成。以上这些过程都反映在 `bdget` 函数的如下代码中（读者应该记得前面讨论 `add_disk` 函数时，这个函数也曾被调用到）：

```
<fs/block_dev.c>
-----
struct block_device *bdget(dev_t dev)
{
    struct block_device *bdev;
    struct inode *inode;

    inode = iget5_locked(blockdev_superblock, hash(dev),
                        bdev_test, bdev_set, &dev);
    if (!inode)
        return NULL;
    bdev = &BDEV_I(inode)->bdev;
    ...
    return bdev;
}
```

看过这段代码的读者应该知道，它里面还有一个 `if (inode->i_state & I_NEW)` 语句块，不过在打开块设备文件 `/dev/ramhda` 的这个流程中，该 `if` 条件语句不满足，所以不会执行。`bdget` 函数最后返回一个 `block_device` 对象。

`bd_acquire` 中还有一个任务是将新获得的 `struct block_device` 对象记录到“devtmpfs”文件系统下对应 `/dev/ramhda` 目录的 inode 的 `i_bdev` 中：

```
<fs/block_dev.c>
-----
static struct block_device *bd_acquire(struct inode *inode)
{
    struct block_device *bdev;
    ...
    bdev = bget(inode->i_rdev);
    if(bdev){
        inode->i_bdev = bdev;
        ...
    }
}
```

```

    }
}

```

对比字符设备驱动程序可以看到，字符设备打开时设备节点所对应的 inode 的 `i_cdev` 指向 `cdev` 对象，字符设备驱动程序直接操作 `cdev` 对象，而块设备则不同，块设备文件节点的 inode 的 `i_bdev` 指向 `block_device` 对象，但是块设备驱动程序中直接打交道的不是它，而是 `gendisk` 对象，从块设备节点到 `gendisk` 对象，这中间隔了一个 `block_device` 对象。

总之，不管是打开已经被 `add_disk` 加入系统的 `/dev/ramhda` 还是打开 `ramhda` 设备上的一个分区 `/dev/ramhda1`，`bd_acquire()` 函数都会返回一个 `block_device` 对象，它或者是在 `add_disk` 中已经分配好的，或者是在打开一个分区设备时新分配的。

(2) 如果不考虑对块设备文件的排他性打开 (`FMODE_EXCL`)，`blkdev_open` 接下来将调用 `blkdev_get` 函数：

```

<fs/block_dev.c>
-----
static int blkdev_open(struct inode *inode, struct file *filp)
{
    struct block_device *bdev;
    int res;
    ...
    bdev = bd_acquire(inode);
    ...
    res = blkdev_get(bdev, filp->f_mode);
}

```

`blkdev_get` 函数实际上直接调用了 `__blkdev_get` 函数，后者是个非常冗长的函数。在前面对 `add_disk` 函数的讨论中已经看到过这个函数，那时讨论的执行路径是整块 `ramhda` 设备 (`partno=0`)，下面通过打开 `ramhda` 的一个分区设备 `ramhda1` (`partno=1`) 来讨论 `__blkdev_get` 函数的另一条执行路径，以使读者对 `block_device`、`gendisk` 和 `hd_struct` 之间的关系有更加清楚的认识。

`__blkdev_get` 首先有一个核心的调用是 `get_gendisk`，它将获得一个 `gendisk` 对象。读者此处应该记住，对一个块设备及其可能的若干分区设备来说，`gendisk` 对象只有一个，它是在 `add_disk` 函数调用链中被加入到 `bdev_map` 哈希表中的，这个调用发生在：

```

<fs/block_dev.c>
-----
static int __blkdev_get(struct block_device *bdev, fmode_t mode, int for_part)
{
    struct gendisk *disk;
    int partno;
    ...
    disk = get_gendisk(bdev->bd_dev, &partno);
    ...
}

```

```
}
```

注意__blkdev_get 函数的第一个实参,我们现在正处在打开分区设备/dev/ramhda1 的上下文中,所以它是对应 ramhda1 设备的 block_device 对象,因此在接下来对 get_gendisk 调用时,其第一个参数 bdev->bd_dev 应该是分区设备 ramhda1 的设备号,如果 ramhda 的设备号是 MKDEV(251, 0),那么此时的 bdev->bd_dev 就应该是 MKDEV(251, 1)。此处 bdev->bd_dev 的来历是在 blkdev_open()→bd_acquire()→bdget()→iget5_locked():

```
<fs/inode.c>
```

```
struct inode *iget5_locked(struct super_block *sb, unsigned long hashval,
                           int (*test)(struct inode *, void *),
                           int (*set)(struct inode *, void *), void *data)
{
    ...
    old = find_inode(sb, hash, test, data);
    if (!old) {
        if (set(inode, data))
            goto set_failed;
        ...
    }
}
```

以上代码中的 set(inode, data)用来将/dev/ramhda1 对应“devtmpfs”文件系统的 inode 的 i_rdev 赋值给新生成的 block_device 对象的 bd_dev 成员。再回过头来看__blkdev_get 中的 get_gendisk 调用,此时是用分区设备的设备号到 bdev_map 中查找 gendisk,读者可以去阅读这段 kobj_lookup 函数,它一定会找到当初在 add_disk 调用链中以 MKDEV(251, 0)作为设备号加入的 gendisk 对象。这里没有理由忽略对 get_gendisk 函数的调用,因为通过它很快就会将 ramhda 这一块设备及其分区设备的诸多数据结构串联起来。

继续讨论__blkdev_get 函数,在获得了驱动程序用 add_disk 添加的一个 gendisk 对象之后,它将进入如下执行路径:

```
if (!bdev->bd_openers) {
    bdev->bd_disk = disk;
    bdev->bd_contains = bdev;
    if (!partno) {
        ...
    }else{
        struct block_device *whole;
        whole = bdget_disk(disk, 0);
        ...
        ret = __blkdev_get(whole, mode, 1);
        ...
        bdev->bd_contains = whole;
        ...
    }
}
```

```

bdev->bd_part = disk_get_part(disk, partno);
...
bd_set_size(bdev, (loff_t)bdev->bd_part->nr_sects << 9);
}

```

上述 else 语句中, whole 将指向 gendisk 所对应的 block_device 对象, 而 /dev/ramhda1 所对应的 block_device 对象 bdev 将会和 whole 以及分区对象建立关联。而对 __blkdev_get 的调用也有机会使得驱动程序中的 block_device_operations 对象的 open 函数被调用。下面在图 11-6 的基础上再给出一张体现块设备及其上分区设备各数据结构之间关系图 (图 11-7):

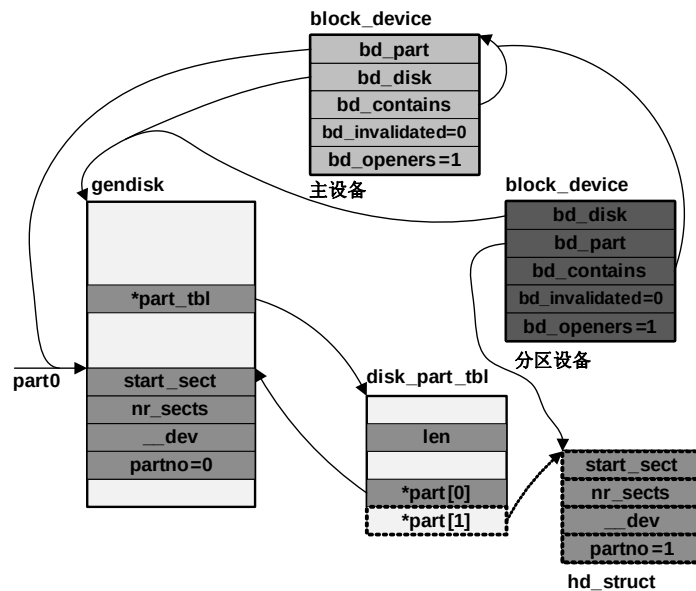


图 11-7 打开一个分区设备产生的关联

可以看到主设备总是对应一个 block_device 对象, 这个对象在 add_disk 中产生; 而对于分区设备, 内核在检测到主设备上的一个有效分区时, 只是通过 add_device 把它加到 /dev 中, 并不会产生 block_device 对象。当打开一个分区设备时, 将产生 block_device 对象, 该对象会和主设备对应的 block_device 对象建立关联。无论如何, block_device 对象中的 bd_disk 成员总是指向 gendisk 对象, 后者在主设备与可能的若干分区设备中只有一个实例。

读者可以看到, 块设备的内核框架实际上要远比字符设备复杂, 对内核中完整的块子系统的讨论不是一本书的一个章节可以胜任的任务, 好在从驱动程序员的角度来说, 更多的关注点是在与设备驱动程序相关的部分上。如果没有对内核代码强烈好奇心和足够的耐心, 无论用如何通俗的语言, 块设备子系统对我们而言依然如雾里看花。写作本书的一个目的是希望能尽可能帮助读者更容易读懂内核, 此处我们对块设备文件的打开似乎依然还比较迷茫, 我将再次把前面对块设备打开的讨论用一张简图来表示, 它大约看起来就像图 11-8 那样:

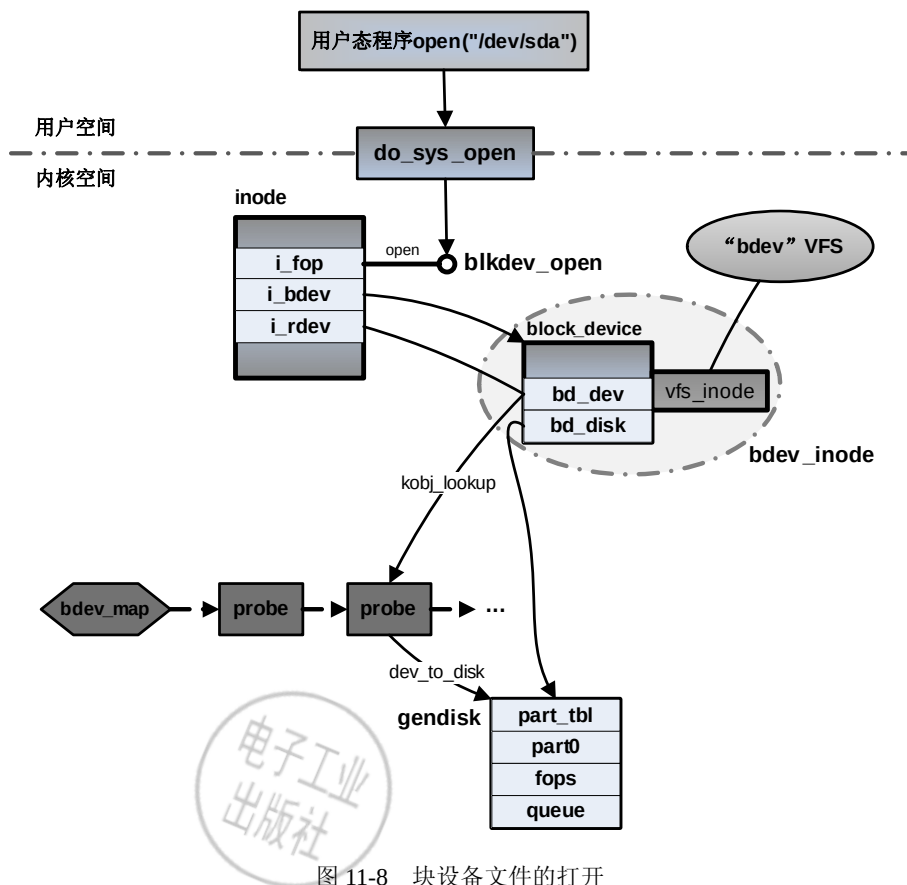


图 11-8 块设备文件的打开

虽然该图描述打开一个块设备的大体流程，但是实际上我们也能从中发现 `block_device` 与 `gendisk` 的区分与关联，设备驱动程序基本上不需要与 `block_device` 以及“bdev”文件系统打交道，它们只是作为内核中块设备驱动框架上的设计元素，驱动程序更多的是要与 `gendisk` 打交道。

上面重点讨论了块主设备和分区设备打开的执行流程，在此基础上，本应该再讨论一下打开操作的相反过程：如何关闭一个块设备。不过鉴于目前章节的篇幅，我决定还是把 `blkdev_close` 这个函数的代码分析留给读者。

11.11 blk_init_queue

来自文件系统的对块设备的数据传输等操作以请求（request）的方式发送给块设备，为了防止请求的丢失，块设备需要备有一个容纳请求的队列。因此在块设备驱动程序中，需要在内核的帮助下来为当前块设备申请一个请求队列，同时需要提供一个能够处理队列中每个请求的设备特定的处理函数（下文简称请求处理函数）。具体到块设备驱动程序，根据处

理方式的不同，目前有两种方式可以完成当前的任务，为了叙述的方便，不妨将其简称为 request 和 make_request 方式。

本节讨论 request 方式，下节再讨论 make_request 方式的实现。采用 request 方式时，块设备驱动程序需要调用 blk_init_queue 来为当前块设备分配一个请求队列，同时安装驱动程序实现的请求处理函数。在调用该函数时，驱动程序内部需要实现一个请求处理函数并将其作为参数传递给 blk_init_queue，驱动程序实现的这个请求处理函数会被 Linux 内核的块通用层代码所调用，显然这是个与具体设备密切相关的函数，在本章稍后部分会继续讨论。

blk_init_queue 函数源码为：

```
<block/blk-core.c>
```

```
struct request_queue *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)
{
    return blk_init_queue_node(rfn, lock, -1);
}
```

其中第一个参数的类型为 request_fn_proc，代表请求处理函数，其原型定义为：

```
<include/linux/blkdev.h>
```

```
typedef void (request_fn_proc) (struct request_queue *q);
```

这个请求处理函数只有一个参数 struct request_queue *q。在继续下面的讨论前有必要先介绍一下该参数的类型 struct request_queue，内核用这个结构来表示一个请求队列：

```
<include/linux/blkdev.h>
```

```
struct request_queue
{
    /*
     * Together with queue_head for cacheline sharing
     */
    struct list_head    queue_head;
    struct request      *last_merge;
    struct elevator_queue *elevator;

    /*
     * the queue request freelist, one for reads and one for writes
     */
    struct request_list rq;

    request_fn_proc      *request_fn;
    make_request_fn      *make_request_fn;
    prep_rq_fn           *prep_rq_fn;
    unprep_rq_fn         *unprep_rq_fn;
    merge_bvec_fn        *merge_bvec_fn;
    softirq_done_fn      *softirq_done_fn;
```



```
rq_timed_out_fn      *rq_timed_out_fn;
dma_drain_needed_fn  *dma_drain_needed;
lld_busy_fn          *lld_busy_fn;

/*
 * Dispatch queue sorting
 */
sector_t             end_sector;
struct request        *boundary_rq;

/*
 * Delayed queue handling
 */
struct delayed_work   delay_work;

struct backing_dev_info backing_dev_info;

/*
 * The queue owner gets to use this for whatever they like.
 * ll_rw_blk doesn't touch it.
 */
void                 *queuedata;

/*
 * queue needs bounce pages for pages above this limit
 */
gfp_t                bounce_gfp;

/*
 * various queue flags, see QUEUE_* below
 */
unsigned long         queue_flags;

/*
 * protects queue structures from reentrancy. -> __queue_lock should
 * _never_ be used directly, it is queue private. always use
 * -> queue_lock.
 */
spinlock_t           __queue_lock;
spinlock_t           *queue_lock;

/*
 * queue kobject
 */
struct kobject kobj;
```

```

/*
 * queue settings
 */
unsigned long    nr_requests; /* Max # of requests */
unsigned int     nr_congestion_on;
unsigned int     nr_congestion_off;
unsigned int     nr_batching;

void             *dma_drain_buffer;
unsigned int     dma_drain_size;
unsigned int     dma_pad_mask;
unsigned int     dma_alignment;

struct blk_queue_tag    *queue_tags;
struct list_head    tag_busy_list;

unsigned int     nr_sorted;
unsigned int     in_flight[2];

unsigned int     rq_timeout;
struct timer_list    timeout;
struct list_head    timeout_list;

struct queue_limits    limits;

/*
 * sg stuff
 */
unsigned int     sg_timeout;
unsigned int     sg_reserved_size;
int             node;
#ifdef CONFIG_BLK_DEV_IO_TRACE
    struct blk_trace    *blk_trace;
#endif
/*
 * for flush operations
 */
unsigned int     flush_flags;
unsigned int     flush_pending_idx:1;
unsigned int     flush_running_idx:1;
unsigned long    flush_pending_since;
struct list_head    flush_queue[2];
struct list_head    flush_data_in_flight;
struct request     flush_rq;

struct mutex     sysfs_lock;

```

```

#ifdef(CONFIG_BLK_DEV_BSG)
    struct bsg_class_device bsg_dev;
#endif

#ifdef CONFIG_BLK_DEV_THROTTLING
    /* Throttle data */
    struct throtl_data *td;
#endif
};

```

很复杂的一个数据结构，内部还嵌入了不少重量级的数据结构，不过目前只需了解几个常用的成员即可：

`struct list_head queue_head`

双向链表元素，在请求队列中作为一个表头，将所有加入队列的 I/O 请求组建成一个双向链表。链表中的每个元素都是一个 `request` 类型，表示一个 I/O 请求对象。内核为了对块设备的 I/O 获得最好的性能，会对请求队列中的 I/O 请求对象进行重排或者是合并，以最大程度减少硬件磁头的移动距离。

`request_fn_proc` `*request_fn`

指向块设备驱动程序需要实现的请求处理函数。当内核中的其他组件比如文件系统需从底层块设备读取或者写入数据时，如果设备驱动程序采用的是 `request` 方式的实现，内核会调用该例程。因此该例程需要针对特定的设备实现底层的 I/O 操作。

`make_request_fn` `*make_request_fn`

如果驱动程序调用 `blk_init_queue` 来处理请求，那么在其调用链中，内核会为当前请求队列的 `make_request_fn` 提供一个标准的实现 `__make_request`。但如果驱动程序采用所谓的 `make_request` 方式实现，则驱动程序需要调用 `blk_queue_make_request` 为此处的 `make_request_fn` 提供一个实现，由于 `blk_queue_make_request` 函数的内部不负责创建设备的请求队列，所以 `make_request` 方式需要驱动程序在调用 `blk_queue_make_request` 前显式地为设备创建一个请求队列。

`unsigned long` `queue_flags`

用来表示当前请求队列的状态，状态标志定义在 `include/linux/blkdev.h` 中，比如 `QUEUE_FLAG_STOPPED`、`QUEUE_FLAG_PLUGGED` 和 `QUEUE_FLAG_QUEUED` 等。

请求队列中还提供了针对队列操作的其他一些操作函数，但是对于设备驱动程序而言，重点在于了解上面的 `request_fn` 及 `make_request_fn` 如何被内核通用块层使用，队列的其他操作在内核中都有一套标准的函数来处理。

由于请求队列主要用来存放针对块设备的 I/O 请求，每个请求都是一个 struct request 类型的对象，所以此处也顺势列出这个结构的定义以使读者对 I/O 请求建立直观印象：

```
<include/linux/blkdev.h>
```

```
struct request {
    struct list_head queuelist;
    struct call_single_data csd;

    struct request_queue *q;

    unsigned int cmd_flags;
    enum rq_cmd_type_bits cmd_type;
    unsigned long atomic_flags;

    int cpu;

    /* the following two fields are internal, NEVER access directly */
    unsigned int __data_len; /* total data len */
    sector_t __sector;      /* sector cursor */

    struct bio *bio;
    struct bio *biotail;

    struct hlist_node hash; /* merge hash */
    /*
     * The rb_node is only used inside the io scheduler, requests
     * are pruned when moved to the dispatch queue. So let the
     * completion_data share space with the rb_node.
     */
    union {
        struct rb_node rb_node; /* sort/lookup */
        void *completion_data;
    };

    /*
     * Three pointers are available for the IO schedulers, if they need
     * more they have to dynamically allocate it.
     */
    void *elevator_private;
    void *elevator_private2;
    void *elevator_private3;

    struct gendisk *rq_disk;
    unsigned long start_time;
#ifdef CONFIG_BLK_CGROUP
    unsigned long long start_time_ns;
```

```

    unsigned long long io_start_time_ns;    /* when passed to hardware */
#endif
    /* Number of scatter-gather DMA addr+len pairs after
     * physical address coalescing is performed.
     */
    unsigned short nr_phys_segments;

    unsigned short ioprio;

    int ref_count;

    void *special;    /* opaque pointer available for LLD use */
    char *buffer;    /* kaddr of the current segment if available */

    int tag;
    int errors;

    /*
     * when request is used as a packet command carrier
     */
    unsigned char __cmd[BLK_MAX_CDB];
    unsigned char *cmd;
    unsigned short cmd_len;

    unsigned int extra_len; /* length of alignment and padding */
    unsigned int sense_len;
    unsigned int resid_len; /* residual count */
    void *sense;

    unsigned long deadline;
    struct list_head timeout_list;
    unsigned int timeout;
    int retries;

    /*
     * completion callback.
     */
    rq_end_io_fn *end_io;
    void *end_io_data;

    /* for bidi */
    struct request *next_rq;
};

```

上述 struct request 结构描述了发送给块设备的请求对象，其中常用的成员有：

```
struct list_head queue_list
```

请求对象中的链表元素，用来把当前请求加到 struct blk_plug 所表示的链表中。

```
struct request_queue *q
```

指向用于存放当前请求的请求队列。

```
unsigned int cmd_flags
```

```
enum rq_cmd_type_bits cmd_type
```

请求除了表示 I/O 操作外，还可以是一些控制之类的命令。上述成员描述了命令的类型和特征。

```
unsigned int __data_len
```

```
sector_t __sector
```

前者用于表示当前请求要求数据传输的总的数据量，后者表示当前请求要求数据传输的块设备的起始扇区。内核源码注释说“NEVER access directly”，驱动程序可以使用内核提供的函数来访问这两个变量：blk_rq_bytes 返回 __data_len 成员，blk_rq_pos 返回 __sector 成员。不允许直接访问，可能是将来存在变更的可能性，那么内核应该保证驱动程序使用的 blk_rq_bytes 和 blk_rq_pos 在这种情况下发生时也能正常工作。

```
struct bio *bio
```

```
struct bio *biotail
```

内核中块子系统在将一个 bio 对象所携带的信息转储到当前请求对象中，或者是一个新的 bio 对象合并到当前请求对象时使用它们，请求对象通过它们将这些 bio 以链表的形式组织起来。如果设备驱动程序使用内核提供的标准 __make_request 函数，那么在驱动程序实现的请求处理函数中将可以访问到这些 bio，如果设备可以使用分散/聚集的 I/O 方式，那么可以使用 rq_for_each_segment 来遍历这个 bio 链表，典型的使用是通过 blk_rq_map_sg 函数来为 DMA 方式建立分散/聚集映射。

介绍完请求队列 request_queue 和请求 request 数据结构，下面继续讨论 blk_init_queue 函数，其内部调用了 blk_init_queue_node：

```
<block/blk-core.c>
```

```
struct request_queue *
blk_init_queue_node(request_fn_proc *rfn, spinlock_t *lock, int node_id)
{
    struct request_queue *uninit_q, *q;

    uninit_q = blk_alloc_queue_node(GFP_KERNEL, node_id);
    if (!uninit_q)
```

```

        return NULL;

    q = blk_init_allocated_queue_node(uninit_q, rfn, lock, node_id);
    if (!q)
        blk_cleanup_queue(uninit_q);

    return q;
}

```

函数 `blk_init_queue` 将返回一个类型为 `struct request_queue` 的请求队列的指针，函数的第一个参数 `rfn` 是个函数指针，设备驱动程序将负责实现该函数并在调用 `blk_init_queue` 时将其传递给 `rfn`，由 `rfn` 来处理块设备队列中的请求。

`blk_alloc_queue_node` 的主要功能是在前面提到的 `blk_requestq_cachep` 缓冲池中分配一个 `struct request_queue` 对象 `uninit_q` 并做一些初始化工作。这个 `uninit_q` 还需要通过 `blk_init_allocated_queue_node` 函数的进一步处理才能返回给驱动程序使用。关于等待队列更实质性的工作则放在 `blk_init_allocated_queue_node` 函数中完成，其源码为：

<block/blk-core.c>

```

struct request_queue *
blk_init_allocated_queue_node(struct request_queue *q, request_fn_proc *rfn,
                             spinlock_t *lock, int node_id)
{
    if (!q)
        return NULL;

    q->node = node_id;
    if (blk_init_free_list(q))
        return NULL;

    q->request_fn      = rfn;
    q->prep_rq_fn      = NULL;
    q->unplug_fn       = generic_unplug_device;
    q->queue_flags     = QUEUE_FLAG_DEFAULT;
    q->queue_lock      = lock;

    /*
     * This also sets hw/phys segments, boundary and size
     */
    blk_queue_make_request(q, __make_request);

    q->sg_reserved_size = INT_MAX;

    /*
     * all done

```

```

        */
        if (!elevator_init(q, NULL)) {
            blk_queue_congestion_threshold(q);
            return q;
        }

        return NULL;
    }
}

```

函数继续对传进来的请求队列 `uninit_q` 初始化, 包括在 `blk_queue_make_request` 函数中, 其中我们看到设备驱动程序中实现的 `rfn` 被赋值给了请求队列的 `request_fn` 成员, 另外我们要注意 `blk_queue_make_request(q, __make_request)` 的调用, 在这个调用中, 队列对象 `q` 的 `make_request_fn` 成员被赋值为 `__make_request`, 后者是内核实现的一个标准函数。这里关注这个调用, 是因为 `__make_request` 将用来为队列 `q` 产生新的请求, 而且最终会调用到驱动程序中实现的 `request_fn` 函数, 在后边的讨论中将会看到这一点。

与请求队列更直接的工作则发生在 `elevator_init` 函数中, 我们很快就会知道, 内核中的块子系统对请求队列使用的调度算法是所谓的“电梯算法”。`elevator_init` 主要用来为前面产生的请求队列 `q` 选择相应的调度算法, 当前 Linux 内核针对请求队列共提供了三种调度算法, 我们并不关心这些算法的实现细节, 不过了解一下内核在 `elevator_init` 中如何为请求队列选择调度算法还是有些必要的:

<block/elevator.c>

```

int elevator_init(struct request_queue *q, char *name)
{
    struct elevator_type *e = NULL;
    struct elevator_queue *eq;
    void *data;

    if (unlikely(q->elevator))
        return 0;

    INIT_LIST_HEAD(&q->queue_head);
    q->last_merge = NULL;
    q->end_sector = 0;
    q->boundary_rq = NULL;

    if (name) {
        e = elevator_get(name);
        if (!e)
            return -EINVAL;
    }

    if (!e && *chosen_elevator) {

```



```

        e = elevator_get(chosen_elevator);
        if (!e)
            printk(KERN_ERR "I/O scheduler %s not found\n",
                    chosen_elevator);
    }

    if (!e) {
        e = elevator_get(CONFIG_DEFAULT_IOSCHED);
        if (!e) {
            printk(KERN_ERR
                    "Default I/O scheduler not found. " \
                    "Using noop.\n");
            e = elevator_get("noop");
        }
    }

    eq = elevator_alloc(q, e);
    if (!eq)
        return -ENOMEM;

    data = elevator_init_queue(q, eq);
    if (!data) {
        kobject_put(&eq->kobj);
        return -ENOMEM;
    }

    elevator_attach(q, eq, data);
    return 0;
}

```

`elevator_init` 函数主要用来选择 I/O 调度器同时对选定的调度器进行初始化，系统中的每个调度器对象都用一个 `struct elevator_type` 数据结构来表示：

```

<include/linux/elevator.h>
-----
struct elevator_type
{
    struct list_head list;
    struct elevator_ops ops;
    struct elv_fs_entry *elevator_attr;
    char elevator_name[ELV_NAME_MAX];
    struct module *elevator_owner;
}

```

`struct elevator_type` 中定义了针对当前调度器对象的一组操作集（`struct elevator_ops ops`），成员 `list` 用来将当前调度器对象加到一个链表中，`elevator_name` 则是调度器的名称。

函数的第二个参数用来表示调度器的名称，如果函数调用者指定了名称，那么将通过 `elevator_get` 来得到调度器的类型。内核将所有支持的调度器放在一个全局的 `elv_list` 链表中，这样 `elevator_get` 只需遍历该链表就可以找到指定名称的调度器。在系统初始化阶段，所有 I/O 调度器通过调用 `elv_register` 将自己加到 `elv_list` 链表中。

如果调用 `elevator_init` 时没有指定名称，那么函数会检查是否在内核启动命令行中提供有调度器信息，命令行中指定调度器的格式为“`elevator=xxx`”。如果命令行中也没有指定调度器，那么内核将从 `CONFIG_DEFAULT_IOSCHED` 中获得 I/O 调度器信息，块设备默认的调度器可以在 `shell` 底下通过命令看到，比如对于块 `ramhda`，通过下面的命令可以看到如下结果：

```
root@AMDLinuxFGL:~# cat /sys/block/ramhda/queue/scheduler
noop deadline [cfq]
```

从设备驱动程序角度出发，我们对调度器的技术细节不再做更多的讨论。至此，我们通过对 `blk_init_queue` 函数的调用为当前的块设备分配了一个请求队列，并且为该请求队列指定了一个调度器，调度器的主要任务是合并一些起始扇区相邻的请求，目的是使磁盘在寻址数据时所经过的路径最短。所有的这一切使得块设备进一步工作的环境准备就绪，现在我们开始讨论内核如何通过请求队列向设备发送请求以完成系统与外设之间的数据传输。

11.12 blk_queue_make_request

如果块设备驱动程序中调用了 `blk_queue_make_request`，实际上就是修改了内核为请求队列 `q` 提供的一个默认的 `make_request_fn` 函数 `__make_request`，换言之，`blk_queue_make_request` 将把驱动程序自己实现的请求处理函数 `hook` 到 `q->make_request_fn` 上，这就成了前面提到的块设备驱动程序对请求处理的两种方式之一的 `make_request` 方式。驱动程序如果采用这种方式，需要在调用 `blk_queue_make_request` 函数前显式地调用比如 `blk_alloc_queue` 来为自己产生一个请求队列对象。

`blk_queue_make_request` 函数为块设备驱动程序的请求队列提供了另外一个处理请求的函数，其核心代码片段为：

```
<block/blk-settings.c>
-----
void blk_queue_make_request(struct request_queue *q, make_request_fn *mfn)
{
    /*
     * set defaults
     */
    q->nr_requests = BLKDEV_MAX_RQ;
```

```

    q->make_request_fn = mfn;
    ...
    blk_queue_congestion_threshold(q);
    q->nr_batching = BLK_BATCH_REQ;
    ...
}

```

主核心的操作是为块设备请求队列 `q` 安装一个请求处理函数 `q->make_request_fn = mfn`，这个函数的其他代码设定了请求队列的一些属性。

在设备驱动所在的模块被从系统中卸载时需要把请求队列所占的资源返还给系统，这个任务由 `blk_cleanup_queue` 函数来完成，其原型如下：

```
void blk_cleanup_queue(struct request_queue *q);
```

不论驱动程序使用 `request` 方式还是 `make_request` 方式来处理请求，`blk_cleanup_queue` 都应在模块卸载时被调用以清除所占用的资源。

11.13 向队列提交请求

当内核文件子系统需要与块设备进行数据传输或者对块设备发送控制命令时，内核需要向对应块设备所属的请求队列发送请求对象。这个任务由函数 `submit_bio` 来完成，其原型为：

```
<block/blk-core.c>
```

```
void submit_bio(int rw, struct bio *bio);
```

内核在需要和块设备进行数据传输时（比如文件系统操作）就可以发起对 `submit_bio` 的调用。通过 `submit_bio` 的原型可以看到，该函数接受两个参数：第一个参数 `rw` 用来标识当前请求是读还是写；第二个参数是一个 `struct bio` 类型的指针，具体指明了对块设备发送的请求信息的细节，驱动程序负责解读这个结构并控制实际硬件完成对应的请求。关于 `struct bio` 结构的细节，稍后有独立的一节予以讨论。`submit_bio` 总是会在需要传输数据的设备被打开之后才可能被调用，而通过前面的讨论，一个块设备（无论是主设备还是分区设备）被打开时，总是会伴随着一个 `block_device` 对象 `bdev`，`bio` 对象中的 `bi_bdev` 对象指向 `bdev`，因此不用担心由 `submit_bio` 所发出的请求不会到达目标设备的请求队列中。

当内核其他组件调用 `submit_bio` 时，必然已经产生了一个 `struct bio` 的对象，此处我们并不关心这个对象是如何产生的，我们关心的是作为当前块设备的驱动程序，如何让所管理的设备完成包含在请求中的数据传输。`submit_bio` 函数的核心代码片段为：

```
<block/blk-core.c>
```

```
void submit_bio(int rw, struct bio *bio)
{
```

```

    int count = bio_sectors(bio);
    bio->bi_rw |= rw;
    ...
    generic_make_request(bio);
}

```

函数有两个参数 `rw` 和 `bio`，调用 `submit_bio` 的当前进程用 `rw` 来表示所请求数据传输的方向，数据传输的其他信息则放在 `bio` 中。`submit_bio` 在做完一些统计等操作之后，调用 `generic_make_request`，传入的参数为 `bio` 对象。`generic_make_request` 函数的实现是：

<block/blk-core.c>

```

void generic_make_request(struct bio *bio)
{
    struct bio_list bio_list_on_stack;

    if (current->bio_list) {
        /* make_request is active */
        bio_list_add(current->bio_list, bio);
        return;
    }
    BUG_ON(bio->bi_next);
    bio_list_init(&bio_list_on_stack);
    current->bio_list = &bio_list_on_stack;
    do {
        __generic_make_request(bio);
        bio = bio_list_pop(current->bio_list);
    } while (bio);
    current->bio_list = NULL; /* deactivate */
}

```

函数中的 `if (current->bio_list)` 语句用来判断当前进程是否有请求正在被处理，如果有，则只是把当前的 `bio` 对象加到 `current->bio_list` 链表尾部就返回。否则调用 `__generic_make_request` 来处理这个 `bio`，其中 `do...while` 循环将遍历 `current->bio_list` 中的所有 `bio`，对于每一个提取出的 `bio` 都会调用 `__generic_make_request` 来处理。一个要注意的细节是，`bio_list_pop` 再从 `current->bio_list` 链表的头部 `pop` 出一个 `bio` 对象后，会将 `bio->bi_next` 置为 `NULL`，然后交予 `__generic_make_request`。

`__generic_make_request` 中一些核心的函数调用为：

<block/blk-core.c>

```

static inline void __generic_make_request(struct bio *bio)
{
    struct request_queue *q;
    int ret, nr_sectors = bio_sectors(bio);
    ...
}

```

```

    if (bio_check_eod(bio, nr_sectors))
        goto end_io;
    do {
        q = bdev_get_queue(bio->bi_bdev);
        ...
        blk_partition_remap(bio);
        ...
        ret = q->make_request_fn(q, bio);
    } while (ret);
end_io:
    bio_endio(bio, err);
}

```

函数首先作一些安全检查，`bio_sectors(bio)` 获得本次请求需要传输的数据量总的扇区数，因为块设备的扇区数总是有限大小，如果要求传输的扇区数大于目标设备拥有的扇区数，`bio_check_eod` 将会结束这次请求。之后，函数调用 `bdev_get_queue(bio->bi_bdev)` 来获得当前请求目标设备的请求队列，因为 `bio->bi_bdev` 总是指向目标设备的 `block_device` 对象，所以 `bio->bi_bdev->bd_disk` 将指向 `gendisk` 对象，因此也就找了设备的请求队列 `q = gendisk->queue`。我们知道，不论是主设备还是由此衍生出来的分区设备，它们都共享同一个 `gendisk` 对象，因此也就共享同一个请求队列。随后的 `blk_partition_remap(bio)` 用来将针对分区设备的 `bio` 调整为针对整个块设备，原因是当打开一个分区设备比如 `ramhda1` 并从其第 0 扇区开始进行数据传输时，`bio->bi_sector` 为 0，刚刚提到分区设备和主设备都适用同一个请求队列，所以需要将其调整到整块设备范围上的寻址扇区，否则将读取主设备 `ramhda` 的第 0 个扇区，而不是 `ramhda1` 的第 0 个扇区。明白了这些，理解 `blk_partition_remap` 函数的实现原理就很简单了，它利用目标设备的 `bdev` 对象中的 `bd_part` 成员来获取当前分区设备在整块设备中的起始扇区：

```

    struct hd_struct *p = bdev->bd_part;
    bio->bi_sector += p->start_sect;
    bio->bi_bdev = bdev->bd_contains;

```

最后一行代码将当前 `bio` 对象所指向的分区设备的 `bdev` 变成了主设备的 `bdev`，读者可参考图 11-7。

`__generic_make_request` 最后调用请求队列的 `make_request_fn` 函数：

```
ret = q->make_request_fn(q, bio);
```

如果驱动程序使用 `blk_init_queue` 来分配并初始化请求队列，那么由 `blk_init_queue` 发起的调用链中将为请求队列 `q` 的成员 `make_request_fn` 赋值 `__make_request`，后者是由内核提供的一个函数，其核心操作代码片段为：

```
<block/blk-core.c>
```

```
static int __make_request(struct request_queue *q, struct bio *bio)
```

```

{
    struct request *req;
    struct blk_plug *plug;
    ...
    if (attempt_plug_merge(current, q, bio))
        goto out;
    ...
    el_ret = elv_merge(q, &req, bio);
    if (el_ret == ELEVATOR_BACK_MERGE) {
        if (bio_attempt_back_merge(q, req, bio)) {
            if (!attempt_back_merge(q, req))
                elv_merged_request(q, req, el_ret);
            goto out_unlock;
        }
    } else if (el_ret == ELEVATOR_FRONT_MERGE) {
        if (bio_attempt_front_merge(q, req, bio)) {
            if (!attempt_front_merge(q, req))
                elv_merged_request(q, req, el_ret);
            goto out_unlock;
        }
    }
}
get_rq:
...
init_request_from_bio(req, bio);
...
plug = current->plug;
if (plug) {
    ...
} else {
    ...
    __blk_run_queue(q);
    ...
}
out_unlock:
    spin_unlock_irq(q->queue_lock);
out:
    return 0;
}

```

函数的核心思想是，当一个新的 bio 对象（代表外部组件对块设备的一个请求）进来时，I/O 调度器要进行判断并采取行动：如果该 bio 对象无法与当前请求队列 q 中其他的某些请求 req 合并，那么将为该 bio 对象产生一个新的请求对象并加入队列，代码进入 get_rq 标识的区域中。如果该 bio 对象可以与队列中现有的某些请求对象 req 进行合并，比如说，当前队列 q 中有个 req 对象，希望从块设备的第 0 个扇区起连续读 8 个扇区，而新进来的 bio 中的数据表明它希望从块设备的第 8 个扇区起连续读 10 个扇区，那么这就显然可以将队列中的

这两个请求予以合并而无须产生一个新的请求对象，这种情况下只需 `req->__data_len += bio->bi_size` 即可。当然 Linux 内核中请求队列的实现细节不是那么简单，因为有太多的因素需要考虑，但是因为关于这部分的大量内容都应该归结到描述 Linux 内核机制的书里，所以对于设备驱动程序而言，我们只需关注与设备驱动程序关系密切的那部分就可以了。

具体说来，其中 `elv_merge` 试图通过块子系统的 I/O 调度器对请求进行重新排列，以最大程度减少磁盘磁头移动的距离。`init_request_from_bio` 则将新进入的 `bio` 对象的一些信息转储到请求对象 `req` 中，所以我们在前面 `ramdisk` 的 `request` 版本看到，驱动程序中实现的 `ramhd_req_func` 请求处理函数只是操作 `req` 对象中记录的读/写信息（比如读/写的起始扇区、读/写的数据量的大小等），这是因为内核为请求队列对象 `q` 中的 `make_request_fn` 函数指针成员提供的 `__make_request` 在最终调到设备驱动程序中的请求处理函数时，已经将 `bio` 的信息转储到了 `req` 中。但如果设备驱动程序中调用 `blk_queue_make_request` 函数来为请求队列中的 `make_request_fn` 函数指针成员提供一个新的 `make_request` 函数，那么在驱动程序自己的请求处理函数中就必须自己处理每个 `bio` 对象，正如我们在 `ramdisk` 的 `make_request` 版本中看到的那样。

在 `__make_request` 的最后对 `__blk_run_queue` 函数的调用⁴中，我们看到了 `request_fn_proc` 被调用到，也就是设备驱动程序中提供的请求处理函数：

```
<block/blk-core.c>
-----
void __blk_run_queue(struct request_queue *q)
{
    if (unlikely(blk_queue_stopped(q)))
        return;

    q->request_fn(q);
}
```

函数中的 `blk_queue_stopped` 用来检查请求队列对象 `q` 的 `queue_flags` 标志，以判断当前请求队列的状态。内核中定义了一些宏来检查请求队列对象 `q` 中的 `queue_flags` 标志，比如：

```
<include/linux/blkdev.h>
-----
#define blk_queue_tagged(q)      test_bit(Queue_FLAG_QUEUEED, &(q->queue_flags))
#define blk_queue_stopped(q)    test_bit(Queue_FLAG_STOPPED, &(q->queue_flags))
#define blk_queue_nomerges(q)   test_bit(Queue_FLAG_NOMERGES, &(q->queue_flags))
```

经过前面的讨论，我们看到当其他内核组件（典型的如文件系统）通过 `submit_bio` 来向块设备提交数据传输请求时，根据驱动程序对请求处理函数设计的不同行为（也即通常说的

⁴ 为了使块子系统的 I/O 调度机制有机会对请求队列中的请求进行合并与重排，并非每次 `submit_bio` 都会使得驱动程序中的请求处理函数得以调用，内核有另外的机制来确保请求队列可以积累一定数量的请求后再予以处理。

request 和 make_request 方式), 当这个请求最终达到驱动程序请求处理函数时, 读/写请求参数出现的形式会有变化, 因此设备驱动程序必须有针对性地予以处理, 图 11-9 显示了这一过程:

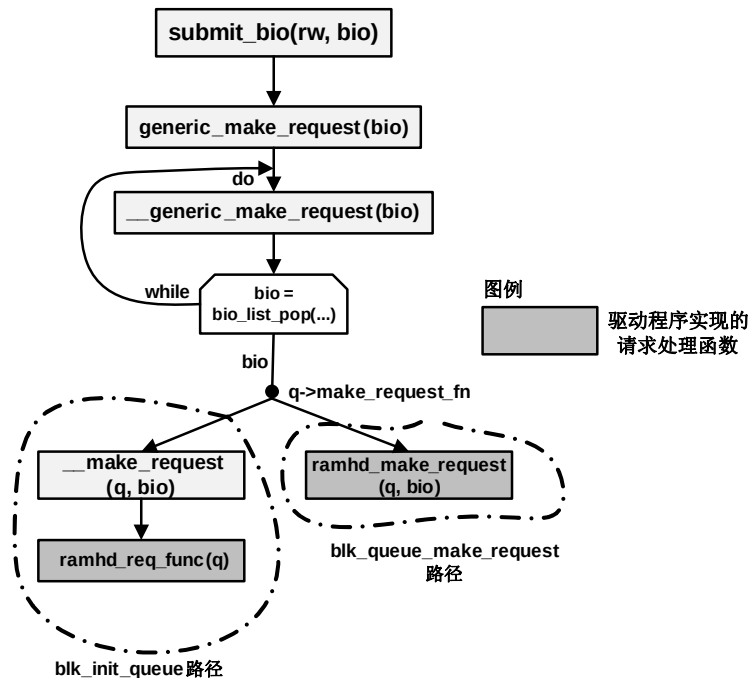


图 11-9 块设备驱动程序实现请求处理函数的两种方式

经过前面的讨论, 我们现在已经可以分清驱动程序用 request 和 make_request 两种方式实现请求处理的区别了: 当驱动程序采用 request 方式时, 外部组件发送给它的 bio 对象将首先被内核提供的函数 __make_request 拦截并处理, __make_request 使用了复杂的逻辑来试图优化目标设备请求队列中的各个请求, 以最大程度提升系统性能。__make_request 在最终调用驱动程序请求处理函数前, 会将 bio 对象中的相关数据转储到请求对象 req 中, 然后把它作为参数传递给请求处理函数, 所以后者利用传入的 req 对象便可知道当前请求的所有信息。当驱动程序使用 make_request 方式时, 实际上是用驱动程序自己实现的一个函数(比如 ramdisk 例子中的 ramhd_make_request)取代了系统的 __make_request 函数, 这种情形下除缺少了块子系统 I/O 调度器的参与之外(当然驱动程序可以自己加入对它们的支持, 但是这种情况极其罕见), 驱动程序实现的 make_request_fn 函数将直接面对 bio, 而不是请求对象 req, 因此需要显式地操作 bio 对象。

因此, 在内核中, 绝大多数块设备的驱动程序使用 request 方式, 而只有少量的驱动程序使用 make_request 方式, 比如内核源码中提供的 ramdisk(源码位于 drivers/block/brd.c)和 loop 等设备。但是对两种实现方式的真正决策依据应该来自于要驱动的设备本身, 而不是内核代码的统计量: 比如 ramdisk 这种设备, 如果使用 request 方式, 意味着将启用块设备子系

统请求重排及合并机制，这对 ramdisk 而言显得有些浪费，所以还是直接接受请求比较好。

11.14 块设备的请求处理函数

系统中驱动程序是了解底层设备的唯一组件，因此高层产生的请求最终必然转发到设备驱动程序中予以处理，为此块设备驱动程序需要实现一个处理函数，正如前面讨论过的，驱动程序有两种方式可以达成该目的。此处仅讲述一下 request 方式，至于 make_request 方式的处理函数，读者可自行参考本章开始部分给出的 ramdisk 代码。

request 处理函数的原型前面已经提过，这里再重复一下：

```
typedef void (request_fn_proc) (struct request_queue *q);
```

函数唯一的参数是隶属于当前设备的请求队列对象 q，前面的 ramdisk 实例展示了块设备驱动程序中一个典型的请求处理函数：

```
void ramhd_req_func (struct request_queue *q)
{
    struct request *req;
    ...
    req = blk_fetch_request(q);
    while (req) {
        start = blk_rq_pos(req); // The sector cursor of the current request
        pdev = (RAMHD_DEV *)req->rq_disk->private_data;
        pData = pdev->data;
        addr = (unsigned long)pData + start * RAMHD_SECTOR_SIZE;
        size = blk_rq_cur_bytes(req);
        if (rq_data_dir(req) == READ)
            memcpy(req->buffer, (char *)addr, size);
        else
            memcpy((char *)addr, req->buffer, size);

        if(!__blk_end_request_cur(req, 0))
            req = blk_fetch_request(q);
    }
}
```

该函数的主体用一个 while 循环来对请求队列 q 中的请求进行处理，blk_fetch_request 用来取得 q 中的第一个请求对象 req。接下来的 rq_data_dir(req)用来判断当前请求 req 上的数据传输方向。在驱动程序把当前请求处理完毕后，调用__blk_end_request_cur 来结束当前请求。

正如前面所讲，块设备的请求队列有两种状态，用 Linux 内核代码中的术语分别是 plug 状态和空闲状态。当一个请求队列处于空闲状态时，其中的请求将会被处理；而当一个队列

处于 `plug` 状态时，队列中的请求并不会被处理，但是允许新的请求进入队列，这就使得内核有机会合并多个请求为一个大的请求以提高系统性能。

从设备驱动程序的角度出发，我们应该熟悉内核提供的那些对请求队列的操作函数。

11.15 bio 结构

通过上面的讨论我们知道，块设备驱动程序的核心是对来自外部的数据传输请求进行处理，而这个过程最核心的数据对象则是 `bio`，它在外部组件与块子系统（包括块设备驱动程序）之间来回流动，将要读/写的数据带来带去。如同公司的班车，在张江集电港与广兰路地铁站之间来回穿梭，一批目光呆滞的工程师被送走了，一批蓬头垢面的工程师被带来了，如此往复，构成了现实世界的一部分。所以没有任何理由不花点篇幅去介绍一下 `struct bio` 结构，`bio` 对象包括了块设备执行一个请求所需要的所有信息，即便是 `req` 对象，我们在前面已看到了，它所携带的关于数据传输的信息也是由 `bio` 转储而来。而且通过 `bio` 对象，块设备驱动程序在执行 I/O 的过程中也无须与创建这个 `bio` 对象的用户进程相关联。另外，`bio` 对象只表示从某扇区开始的若干连续扇区的数据空间，不会表示分散的扇区数据块。该结构的定义为：

```
<include/linux/bio.h>
```

```
struct bio {
    sector_t        bi_sector;        /* device address in 512 byte sectors */
    struct bio       *bi_next;        /* request queue link */
    struct block_device *bi_bdev;
    unsigned long    bi_flags;        /* status, command, etc */
    unsigned long    bi_rw;          /* bottom bits READ/WRITE, top bits priority*/
    unsigned short   bi_vcnt;        /* how many bio_vec's */
    unsigned short   bi_idx;        /* current index into bvl_vec */

    /* Number of segments in this BIO after
     * physical address coalescing is performed.
     */
    unsigned int     bi_phys_segments;
    unsigned int     bi_size;        /* residual I/O count */

    /*
     * To keep track of the max segment size, we account for the
     * sizes of the first and last mergeable segments in this bio.
     */
    unsigned int     bi_seg_front_size;
    unsigned int     bi_seg_back_size;
    unsigned int     bi_max_vecs;    /* max bvl_vecs we can hold */
    unsigned int     bi_comp_cpu;    /* completion CPU */
    atomic_t         bi_cnt;        /* pin count */
}
```

```

    struct bio_vec          *bi_io_vec; /* the actual vec list */
    bio_end_io_t            *bi_end_io;
    void                    *bi_private;
    bio_destructor_t        *bi_destructor; /* destructor */

    /*
     * We can inline a number of vecs at the end of the bio, to avoid
     * double allocations for a small number of bio_vecs. This member
     * MUST obviously be kept at the very end of the bio.
     */
    struct bio_vec          bi_inline_vecs[0];
};

```

其中一些重要的成员如下：

`sector_t` `bi_sector`

指定了本次传输的起始扇区号。

`struct bio` `*bi_next`

指向当前 `bio` 的下一个对象。

`struct block_device` `*bi_bdev`

与请求相关联的块设备对象指针，该成员将引导 `submit_bio` 将请求发往那个设备的请求队列。

`unsigned short` `bi_vcnt`

`bi_io_vec` 数组中元素的个数。

`unsigned short` `bi_idx`

当前处理的 `bi_io_vec` 数组元素索引。

`unsigned int` `bi_size`

本次请求需要传输的数据总量，单位为字节（扇区大小的整数倍）。

`struct bio_vec` `*bi_io_vec`

指向一个 I/O 向量的数组，数组中的每个元素对应一个物理内存页帧的 `page` 对象。`struct bio_vec` 的定义如下：

```

<include/linux/bio.h>
-----
struct bio_vec {

```

```

struct page *bv_page;
unsigned int bv_len;
unsigned int bv_offset;
};

```

其中, `bv_page` 指向用于数据传输的页面所对应的 `struct page` 对象, `bv_len` 表示当前要传输的数据大小, `bv_offset` 表示数据在页面内的偏移量(在非整页传输的情况下)。通过 `bi_io_vec` 数组, 我们看到一个特定的请求所需要传输的数据可能分布在内存的不同页面中, 换句话说驱动程序处理的一个块设备的 I/O 请求可能会使用不止一个数据缓冲区, 这些缓冲区散布在整个内存中。

实际上, `bio` 所形成的链表常常存在于请求 `request` 的对象之中, 这个链表是由块子系统的 I/O 调度器对 `bio` 进行合并的结果。图 11-10 显示了 `bio` 数据结构的构成以及与 `request` 的关系:

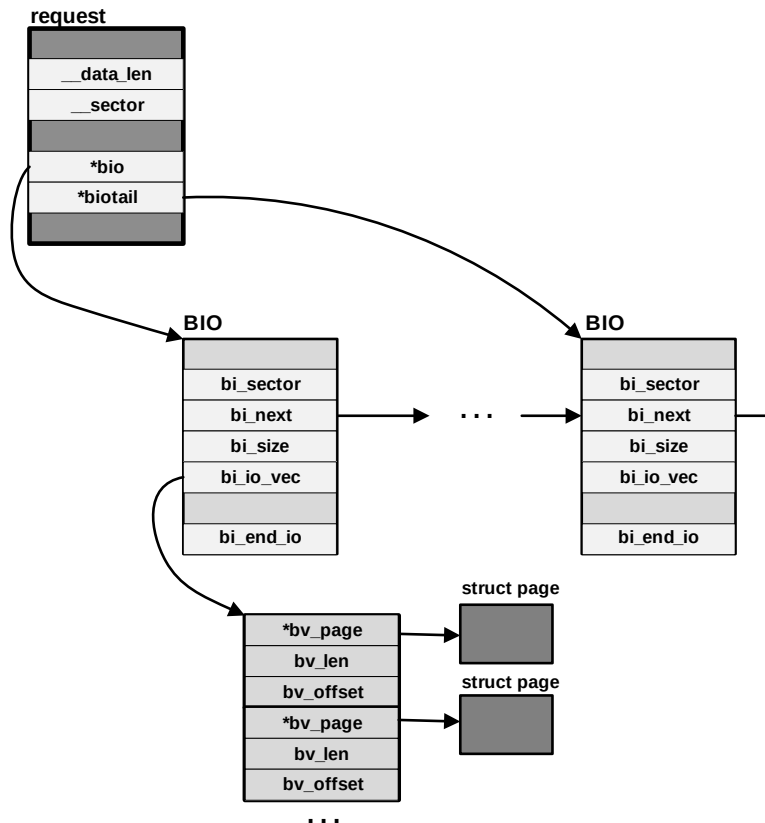


图 11-10 struct bio

正如读者所猜想的那样, 内核为驱动程序操作这些 `BIO` 的成员提供了一组接口函数, 为了代码的易维护性与一致性起见, 设备驱动程序员应该在程序中使用这些函数, 其中驱动程序中常用的一些宏及函数如下:

○ bio_for_each_segment 宏

该宏用来遍历 BIO 成员中的 bi_io_vec 数组中的各个 bio_vec 元素，其定义的形式为：

```
<include/linux/bio.h>
-----
#define bio_for_each_segment(bvl, bio, i) \
    __bio_for_each_segment(bvl, bio, i, (bio)->bi_idx)
```

在这个宏的三个参数中，bvl 指向遍历过程中当前的 struct bio_vec 对象，bio 是整个请求 bio 对象的指针，i 对应当前的 struct bio_vec 对象的序号。

下面是 ramdisk 例子使用 bio_for_each_segment 的一段示例代码：

```
bio_for_each_segment(bvec, bio, i) {
    pBuffer = kmap(bvec->bv_page) + bvec->bv_offset;
    switch(bio_data_dir(bio))
    {
        case READ:
            memcpy(pBuffer, pRHdata, bvec->bv_len);
            flush_dcache_page(bvec->bv_page);
            break;
        case WRITE:
            flush_dcache_page(bvec->bv_page);
            memcpy(pRHdata, pBuffer, bvec->bv_len);
            break;
        default:
            kunmap(bvec->bv_page);
            goto out;
    }
    kunmap(bvec->bv_page);
    pRHdata += bvec->bv_len;
}
```

对于每个 struct page 对象，代码用 kmap 来获得其所对应的虚拟地址。

○ bio_iovec_idx(bio, idx)

```
<include/linux/bio.h>
-----
#define bio_iovec_idx(bio, idx)    (&((bio)->bi_io_vec[(idx)]))
```

该宏用于得到 bi_io_vec 数组中第 idx 个元素对象的指针。

○ bio_iovec(bio)

```
<include/linux/bio.h>
-----
#define bio_iovec(bio)             bio_iovec_idx((bio), (bio)->bi_idx)
```

该宏用于得到当前正在处理的（由 bi_idx 做索引值）bi_io_vec 数组元素的指针，也即当前缓冲区对象。


```

{
    return bvec_kmap_irq(bio_iovec_idx(bio, idx), flags);
}
#define __bio_kunmap_irq(buf, flags)    bvec_kunmap_irq(buf, flags)

#define bio_kmap_irq(bio, flags) \
    __bio_kmap_irq((bio), (bio)->bi_idx, (flags))
#define bio_kunmap_irq(buf, flags) __bio_kunmap_irq(buf, flags)

```

无论 `bio` 中携带的页面地址来自低端内存还是高端内存，该函数都可以得到其内核虚拟地址，但是 `bvec_kmap_irq` 在内部实现时，对于高端地址（`HIGHMEM`），由于在通过 `kmap_atomic` 做映射前使用了 `local_irq_save`，所以在 `bio_kmap_irq` 与 `bio_kunmap_irq` 之间的代码不应该重新启用中断。

11.16 本章小结

在 Linux 下共有三种类型的设备，分别是字符型设备、块设备与网络设备。本章讨论了其中的块设备驱动程序及其相关的外部接口，其中包括块设备相关的数据结构以及与文件系统的交互等。对于块设备驱动程序而言，其核心功能是要控制块设备进行数据传输以实现系统与块设备的数据传输。

与字符型设备驱动程序不同，内核为块设备驱动程序做了更多幕后的工作，上层对块设备的数据传输指令以请求 `request` 的方式放入块设备所拥有的请求队列中，驱动程序需要遍历该请求队列以完成实际的数据传输指令。内核可以在需要的地方调用 `submit_bio` 来向某一块设备提交一个请求，请求将被放入与该块设备对应的一个请求队列中，设备驱动程序负责生成设备的请求队列。内核为了提升块设备 I/O 性能，提供了几个 I/O 调度器用来对请求队列中的请求进行调度，内核在初始化阶段会根据实际情况选择其中一个调度器，通过合并或者重排请求队列中的请求，块设备可以最大程度减少磁头移动的距离并试图一次硬件操作能同时完成多个请求。

块设备驱动程序可以充分利用内核中块子系统所提供的这些 I/O 调度机制，驱动程序这么做的时候，需要提供一个请求处理函数用于处理请求队列中的每个请求。由于高层的 I/O 调度的存在，内核可以将多个 `bio` 合并到一个 `request` 对象中，所以请求处理函数可以根据实际需要灵活处理这些 `bio`，实际的块设备驱动程序中常常使用 DMA 在 `bio` 提供的缓冲区和目标块设备之间传输数据。

当然块设备驱动程序也可以绕开 I/O 调度器，此时需要通过 `blk_queue_make_request` 函数将请求队列的 `make_request_fn` 指针指向自己实现的请求处理函数，驱动程序这样做的时候，实际是用自己实现的请求处理函数取代了内核提供的 `__make_request` 函数，因为 I/O 调度器的实现机制都体现在 `__make_request` 函数中，所以这种情形下相当于驱动程序实现的请求处理函数直接接收来自 `submit_bio` 提交的 `bio` 对象。

第 12 章

网络设备驱动程序

前面依次讨论了字符设备和块设备，本章将讨论 Linux 下的另一类设备：网络设备。网络设备是 Linux 下三大标准设备类型之一，现实中又通常被称为“网卡”，用来完成高层网络协议（如 TCP/UDP 等）的底层数据传输及设备控制等功能。

就数据传输而言，网络设备类似于前面讨论过的块设备，通常情况下两者都被用来与系统进行大量的数据交互，根据上层模块的需求进行数据的发送和接收，但是网络设备如下的一些特性使得它与块设备区别开来。

首先网络设备在 `/dev` 目录下没有入口点，换句话说，网络设备在系统中并不像块设备那样以一个设备文件的形式存在，在应用层，用户通过套接口 API 的 `socket` 函数来使用网络设备。

`socket` API 的原型函数为：

```
<sys/socket.h>
```

```
int socket(int family, int type, int protocol);
```

其中，参数 `family` 用来表示套接口所使用的协议族，包括 `AF_INET`（IPv4 协议族）和 `AF_INET6`（IPv6 协议族）等。参数 `type` 用来表示套接口的类型，有 `SOCK_STREAM`（字节流套接口）、`SOCK_DGRAM`（数据报套接口）和 `SOCK_RAW`（原始套接口）。一般来说，函数的第三个参数在实际使用中常设置为 0，除非是用在原始套接口上。

其次，网络设备除了响应来自内核的请求外，还需要异步地处理来自外部世界的数据包，而对于块设备而言，只需响应来自内核的请求，这使得网络设备驱动程序的设计模式无法等同于块设备驱动程序。除处理数据外，网络设备驱动程序还需要完成诸如地址设置、配置网络传输参数及流量统计等一些管理类的任务。

说到这里，不得不提一下经典的网络协议分层模型。虽然我们不打算在本书过多涉及这方面的内容，因为这不是本书的主题，但是作为互联网世界的幕后推手，无论如何还是有必要简单介绍一下。互联网发展史上曾出现过两种协议分层模型，分别是国际标准化组织 ISO（International Organization for Standardization）的开放系统互联 OSI（Open Systems Interconnection）模型和 TCP/IP 参考模型。前者将组成网络的协议分为七层，分别是应用层、表示层、会话层、传输层、网络层、数据链路层和物理层，后者则将网络分为四层，

分别是应用层、传输层、网际互联层和网络访问层。由于 ISO 的 OSI 模型只是一个理论上的模型，并没有成熟的产品，所以当今互联网事实上的国际标准其实是基于 TCP/IP 模型的，图 12-1 显示了两种模型之间的区别：

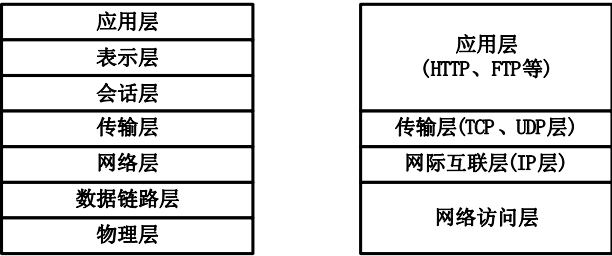


图 12-1 ISO/OSI 模型与 TCP/IP 参考模型

在这两个参考模型中，各层只能与相邻的层进行通信。按照 TCP/IP 参考模型，网络设备及其驱动程序实际上完成的是最底层的网络访问层，该层直接面向实际承担数据传输任务的物理媒体，为数据通信的介质提供规范和定义，主要关心的是在通信线路上传输比特流的问题（信号与接口等）。

由于协议数据的封装性，通过网卡与外部交换数据时最终是用网卡的硬件地址来标识各主机，对于著名的以太网卡而言也称为 MAC 地址，由 48 位二进制数组成。MAC 地址作为一种网络世界的 ID，必须具有全球唯一性，网卡生产商通常把分配到的 MAC 地址烧写进硬件中。

在 Linux 网络部分相关的源代码中，经常会出现“octet”这样的字眼，这是一个在网络世界中使用的术语，指代一个 8 位的数据位，跟字节是一个意思，它是网络设备和协议所能理解的最小单位。本书并不会刻意强调这种提法，因为从网络设备驱动程序员的角度，字节的叫法要更通用。

在实际使用的 TCP/IP 参考模型下，对于网络系统的数据传输而言，也相应地使用了数据封装的方式，它是分层模型的具体体现，图 12-2 显示了网络数据包在各协议层之间传递时的数据封装情况：

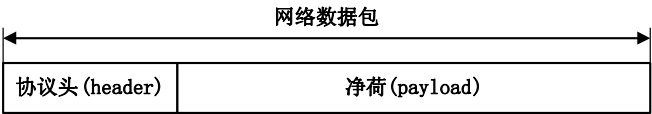


图 12-2 网络数据包的封装格式

当一个网络数据包从上层往下层传递时，下层协议会将上层传下来的数据包视为一个净荷，然后加上本层的协议头以完成该层协议所实现的功能控制信息，这个过程也就是平常所谓的数据包的封装，俗称打包。而当数据包从下层往上传递时，过程正好相反，俗称解包。

广义地讲，网络设备种类繁多，比如网卡 NIC（Network Interface Card）、中继器（repeater）、网桥（bridge）及路由设备（router）等，然而在实际的工作中，读者接触最多的网络设备是以太网设备 NIC，所以本书将以该设备为讨论的主体对象，在后续的描述中，有时会将 NIC、网络设备和网卡或以以太网设备等称谓混用，它们均指代同一个意思。这里给出一个经以太网设备传递的数据包的具体协议封装形式，以方便读者在本章后续部分的解读中建立直观的印象，图 12-3 显示了某一以太网协议数据包的封装形式：

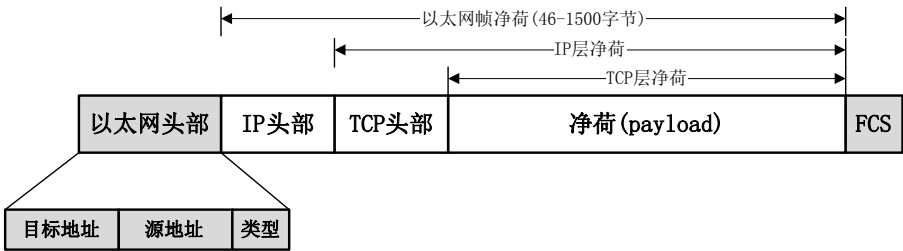


图 12-3 以太网帧封装格式

当某一网络数据包经高层一层一层往下传递，最终到达以太网卡时，将被冠以以太网头部，图中的目标地址和源地址即通常所说的 MAC 地址（6 字节），是网络世界实体的身份号码，更多关于以太网协议的细节可以参考网络协议类书籍。下面将开始解读 Linux 内核源码中关于网络设备驱动程序的幕后细节。

12.1 net_device

在 Linux 内核中，网络设备由数据结构 `net_device` 来表示，它存储着特定网络设备的所有信息，这是一个极其庞大的数据结构¹，也是网络设备驱动程序开发者要面对的第一个核心数据，鉴于本章后续部分经常会引用到这个数据结构，所以经过一些简单编辑与简化后，我们将其在内核中的定义摘录如下。需要注意的是，网络设备驱动程序并不会使用到该结构体中的所有成员，有些数据成员仅是提供给内核中的网络子系统所使用，在该结构体定义的后面，我们会将网络设备驱动程序可能用到的一些常见的成员变量作简单介绍。

```
<include/linux/netdevice.h>
-----
struct net_device {

    /*
     * This is the first field of the "visible" part of this structure
```

¹ 源码中的注释写道，定义这么庞大的一个结构是个很大的错误，因为结构中各成员间的逻辑关系显得比较混乱。
“Actually, this whole structure is a big mistake. It mixes I/O data with strictly "high-level" data, and it has to know about almost every data structure used in the INET module.”

```
    * (i.e. as seen by users in the "Space.c" file).  It is the name
    * the interface.
    */
    char                name[IFNAMSIZ];

    struct pm_qos_request_list *pm_qos_req;

    /* device name hash chain */
    struct hlist_node    name_hlist;
    /* snmp alias */
    char                *ifalias;

    /*
     *   I/O specific fields
     *   FIXME: Merge these and struct ifmap into one
     */
    unsigned long        mem_end; /* shared mem end */
    unsigned long        mem_start; /* shared mem start */
    unsigned long        base_addr; /* device I/O address */
    unsigned int         irq; /* device IRQ number */

    /*
     *   Some hardware also needs these fields, but they are not
     *   part of the usual set specified in Space.c.
     */

    unsigned char        if_port; /* Selectable AUI, TP,.. */
    unsigned char        dma; /* DMA channel */

    unsigned long        state;

    struct list_head      dev_list;
    struct list_head      napi_list;
    struct list_head      unreg_list;

    /* currently active device features */
    u32                   features;
    /* user-changeable features */
    u32                   hw_features;
    /* user-requested features */
    u32                   wanted_features;
    /* VLAN feature mask */
    u32                   vlan_features;

    /* Interface index. Unique device identifier */
    int                   ifindex;
```

```

int            iflink;

struct net_device_stats stats;
atomic_long_t  rx_dropped; /* dropped packets by core network
                           * Do not use this in drivers.
                           */

/* Management operations */
const struct net_device_ops *netdev_ops;
const struct ethtool_ops *ethtool_ops;

/* Hardware header description */
const struct header_ops *header_ops;

unsigned int    flags; /* interface flags (a la BSD) */
unsigned short  gflags;
unsigned short  priv_flags; /* Like 'flags' but invisible to userspace. */
unsigned short  padded; /* How much padding added by alloc_netdev() */

unsigned char    operstate; /* RFC2863 operstate */
unsigned char    link_mode; /* mapping policy to operstate */

unsigned int     mtu; /* interface MTU value */
unsigned short   type; /* interface hardware type */
unsigned short   hard_header_len; /* hardware hdr length */

/* extra head- and tailroom the hardware may need, but not in all cases
 * can this be guaranteed, especially tailroom. Some cases also use
 * LL_MAX_HEADER instead to allocate the skb.
 */
unsigned short    needed_headroom;
unsigned short    needed_tailroom;

/* Interface address info. */
unsigned char      perm_addr[MAX_ADDR_LEN]; /* permanent hw address */
unsigned char      addr_assign_type; /* hw address assignment type */
unsigned char      addr_len; /* hardware address length */
unsigned short     dev_id; /* for shared network cards */

spinlock_t        addr_list_lock;
struct netdev_hw_addr_list uc; /* Unicast mac addresses */
struct netdev_hw_addr_list mc; /* Multicast mac addresses */
int                uc_promisc;
unsigned int        promiscuity;
unsigned int        allmulti;

```

```

void          *atalk_ptr; /* AppleTalk link */
struct in_device __rcu *ip_ptr; /* IPv4 specific data */
struct dn_dev __rcu *dn_ptr; /* DECnet specific data */
struct inet6_dev __rcu *ip6_ptr; /* IPv6 specific data */
void          *ec_ptr; /* Econet specific data */
void          *ax25_ptr; /* AX.25 specific data */
struct wireless_dev *ieee80211_ptr; /* IEEE 802.11 specific data,
                                     assign before registering */

/*
 * Cache line mostly used on receive path (including eth_type_trans())
 */
unsigned long last_rx; /* Time of last Rx
                        * This should not be set in
                        * drivers, unless really needed,
                        * because network stack (bonding)
                        * use it if/when necessary, to
                        * avoid dirtying this cache line.
                        */

struct net_device *master; /* Pointer to master device of a group,
                             * which this device is member of.
                             */

/* Interface address info used in eth_type_trans() */
unsigned char *dev_addr; /* hw address, (before bcast
                           because most packets are
                           unicast) */

struct netdev_hw_addr_list dev_addrs; /* list of device
                                       hw addresses */

unsigned char broadcast[MAX_ADDR_LEN]; /* hw bcast add */

rx_handler_func_t __rcu *rx_handler;
void __rcu *rx_handler_data;

struct netdev_queue __rcu *ingress_queue;

struct netdev_queue *tx ____cacheline_aligned_in_smp;

/* Number of TX queues allocated at alloc_netdev_mq() time */
unsigned int num_tx_queues;

/* Number of TX queues currently active in device */
unsigned int real_num_tx_queues;

```

```

/* root qdisc from userspace point of view */
struct Qdisc      *qdisc;

unsigned long      tx_queue_len;    /* Max frames per queue allowed */
spinlock_t        tx_global_lock;

/* These may be needed for future network-power-down code. */
/*
 * trans_start here is expensive for high speed devices on SMP,
 * please use netdev_queue->trans_start instead.
 */
unsigned long      trans_start; /* Time (in jiffies) of last Tx */

int                watchdog_timeo; /* used by dev_watchdog() */
struct timer_list  watchdog_timer;

/* Number of references to this device */
int __percpu       *pcpu_refcnt;

/* delayed register/unregister */
struct list_head   todo_list;
/* device index hash chain */
struct hlist_node  index_hlist;

struct list_head   link_watch_list;

/* register/unregister state machine */
enum { NETREG_UNINITIALIZED=0,
        NETREG_REGISTERED,    /* completed register_netdevice */
        NETREG_UNREGISTERING, /* called unregister_netdevice */
        NETREG_UNREGISTERED, /* completed unregister todo */
        NETREG_RELEASED,      /* called free_netdev */
        NETREG_DUMMY,         /* dummy device for NAPI poll */
} reg_state:16;

enum {
        RTNL_LINK_INITIALIZED,
        RTNL_LINK_INITIALIZING,
} rtnl_link_state:16;

/* Called from unregister, can be used to call free_netdev */
void (*destructor)(struct net_device *dev);

#ifdef CONFIG_NETPOLL
struct netpoll_info *npinfo;

```

```

#endif

#ifdef CONFIG_NET_NS
    /* Network namespace this network device is inside */
    struct net          *nd_net;
#endif

    /* mid-layer private */
    union {
        void                *ml_priv;
        struct pcpu_lstats __percpu *lstats; /* loopback stats */
        struct pcpu_tstats __percpu *tstats; /* tunnel stats */
        struct pcpu_dstats __percpu *dstats; /* dummy stats */
    };
    /* GARP */
    struct garp_port __rcu *garp_port;

    /* class/net/name entry */
    struct device      dev;
    /* space for optional device, statistics, and wireless sysfs groups */
    const struct attribute_group *sysfs_groups[4];

    /* rnetlink link ops */
    const struct rtnl_link_ops *rtnl_link_ops;

    /* for setting kernel sock attribute on TCP connection setup */
#define GSO_MAX_SIZE      65536
    unsigned int          gso_max_size;

    u8 num_tc;
    struct netdev_tc_txq tc_to_txq[TC_MAX_QUEUE];
    u8 prio_tc_map[TC_BITMASK + 1];

    /* n-tuple filter list attached to this device */
    struct ethtool_rx_ntuple_list ethtool_ntuple_list;

    /* phy device may attach itself for hardware timestamping */
    struct phy_device *phydev;

    /* group the device belongs to */
    int group;
};

char          name[IFNAMSIZ]

```

网络设备的名称，当前内核为其指定的长度 IFNAMSIZ 为 16。在 Linux 内核中，设备

名称字符串末尾的数字用来表示同一网络设备类型的多个适配器（比如系统中有两块以太网卡），表 12-1 所列为常见的一些网络设备的命名规则。

表 12-1 网络设备命名规则

名 称	设备类型
ethX	以太网设备
pppX	调制解调器等 PPP 连接类型的设备
isdnX	ISDN 卡
lo	环回（loopback）设备，用于本地计算机通信

```
struct hlist_node    name_hlist
struct list_head     dev_list
struct hlist_node    index_hlist
```

用于网络设备的列表管理，成功注册进系统的网络设备都将被加到这三个链表中，其中 name_hlist 和 index_hlist 分别用于网络设备名称与接口索引的散列表，dev_list 则用于将当前设备加到所属命名空间（name space）中的 dev_base_head 所管理的全局链表。

```
struct list_head napi_list
```

用于支持 NAPI 特性的网络设备，它将 struct napi_struct 对象的 dev_list 加到 napi_list 所对应的链表中，后续的 NAPI 相关内容会有涉及。

```
const struct net_device_ops *netdev_ops
```

网络设备方法操作集。该数据结构定义了针对当前设备的一组操作集合，比如 ndo_open、ndo_stop 和 ndo_start_xmit 等。稍早一些的内核将这些设备方法直接定义在 net_device 结构下面，当前内核则将这些设备方法放到了一个内嵌在 net_device 中的 struct net_device_ops 数据结构中。

```
const struct header_ops *header_ops
```

针对网络访问层数据帧的一组操作集。对于以太网设备，这个操作集定义在全局变量 eth_header_ops 中。

```
int                ifindex
```

当前网络设备所在的命名空间的接口索引，用来唯一标示设备所提供的接口。

```
unsigned int      mtu
```

网络访问层的最大传输单元 MTU（maximum transfer unit），是针对上一层的净荷。对于以太网设备，该值为 1500。

unsigned short hard_header_len

当前网络设备所处理的网络访问层的硬件协议头的长度。对于以太网设备，为 14 个字节。

unsigned char addr_len

网络访问层硬件地址长度。对于以太网设备，为 6 个字节。

struct netdev_hw_addr_list uc

网络设备的单播（unicast）MAC 地址列表。

struct netdev_hw_addr_list mc

网络设备的多播（multicast）MAC 地址列表。

unsigned char *dev_addr

指向网络设备硬件地址的指针。

struct netdev_hw_addr_list dev_addrs

网络设备硬件地址链表。struct netdev_hw_addr_list 类型定义为：

```
<include/linux/netdevice.h>
-----
struct netdev_hw_addr_list {
    struct list_head      list;
    int                   count;
};
```

net_device 对象通过 list 成员来将隶属于当前设备的硬件地址加到一个链表中，成员 count 表示链表中元素的个数。

unsigned char broadcast[MAX_ADDR_LEN]

网络访问层硬件广播地址。

struct netdev_queue __rcu *ingress_queue

网络设备的接收队列。

struct netdev_queue *_tx

网络设备的发送队列。

unsigned int num_tx_queues

由 alloc_netdev_mq 函数分配的隶属于当前网络设备的发送队列的数量。

```
unsigned int    real_num_tx_queues
```

网络设备中当前活动的发送队列的数量。

```
enum            reg_state
```

当前设备在系统中的注册状态，由六种枚举型变量表示。

```
struct net      *nd_net
```

网络设备所在的命名空间。当前内核已经开始对网络设备引入了命名空间的概念，相对于之前的单一的全区命名空间，`struct net` 的引入应该算是一个新的机制，内核通过引入命名空间的机制，可以在系统中建立多个独立的虚拟视图，各个视图之间彼此分隔。不过这种机制更多地被内核使用，对于设备驱动程序而言，并不会直接和这些概念打交道。

```
int             watchdog_timeo
```

用于设定网络设备在传输数据包时传输超时的到期时间。

```
struct timer_list watchdog_timer
```

发送分组超时定时器。当网络子系统发送队列中某一数据包在指定的时间段之后依然没有成功发送出去，那么该定时器将到期，最终导致设备驱动程序的传输超时函数被调用。

```
struct device    dev
```

内嵌的内核对象所在的数据结构，该成员将会把网络设备纳入到设备驱动模型的体系当中。

对于网络设备驱动程序而言，首先要分配一个 `net_device` 类型的对象来代表所管理的网卡设备，内核为此提供了一个分配 `net_device` 对象的宏 `alloc_netdev`，所以设备驱动程序无须调用 `kmalloc` 函数来分配 `net_device` 对象，因为 `alloc_netdev` 除了分配内存还执行必要的初始化任务：

```
<include/linux/netdevice.h>
-----
#define alloc_netdev(sizeof_priv, name, setup) \
    alloc_netdev_mqs(sizeof_priv, name, setup, 1, 1)
```

所以 `alloc_netdev` 宏实际上调用的是 `alloc_netdev_mqs` 函数，该函数的实现²为：

```
<net/core/dev.c>
-----
struct net_device *alloc_netdev_mqs(int sizeof_priv, const char *name,
    void (*setup)(struct net_device *),
```

² 此处实现中删去了 `CONFIG_RPS` 部分，该选项在 `SMP` 系统中用来实现 `RPS`（Receive Packet Steering）。

```

        unsigned int txqs, unsigned int rxqs)
{
    struct net_device *dev;
    size_t alloc_size;
    struct net_device *p;
    ...
    alloc_size = sizeof(struct net_device);
    if (sizeof_priv) {
        /* ensure 32-byte alignment of private area */
        alloc_size = ALIGN(alloc_size, NETDEV_ALIGN);
        alloc_size += sizeof_priv;
    }
    /* ensure 32-byte alignment of whole construct */
    alloc_size += NETDEV_ALIGN - 1;

    p = kzalloc(alloc_size, GFP_KERNEL);
    if (!p) {
        printk(KERN_ERR "alloc_netdev: Unable to allocate device.\n");
        return NULL;
    }

    dev = PTR_ALIGN(p, NETDEV_ALIGN);
    dev->padded = (char *)dev - (char *)p;

    dev->pcpu_refcnt = alloc_percpu(int);
    if (!dev->pcpu_refcnt)
        goto free_p;

    if (dev_addr_init(dev))
        goto free_pcpu;

    dev_mc_init(dev);
    dev_uc_init(dev);
    dev_net_set(dev, &init_net);
    dev->gso_max_size = GSO_MAX_SIZE;

    INIT_LIST_HEAD(&dev->ethtool_ntuple_list.list);
    dev->ethtool_ntuple_list.count = 0;
    INIT_LIST_HEAD(&dev->napi_list);
    INIT_LIST_HEAD(&dev->unreg_list);
    INIT_LIST_HEAD(&dev->link_watch_list);
    dev->priv_flags = IFF_XMIT_DST_RELEASE;
    setup(dev);

    dev->num_tx_queues = txqs;
    dev->real_num_tx_queues = txqs;

```

```

    if (netif_alloc_netdev_queues(dev))
        goto free_all;

    strcpy(dev->name, name);
    dev->group = INIT_NETDEV_GROUP;
    return dev;

free_all:
    free_netdev(dev);
    return NULL;

free_pcpu:
    free_percpu(dev->pcpu_refcnt);
    kfree(dev->_tx);

free_p:
    kfree(p);
    return NULL;
}

```

先看 `alloc_netdev` 宏的参数。第一个参数是个整型的 `sizeof_priv`，用来表示驱动程序的“私有数据”区的大小，等一下将看到该“私有数据”区实际上是通过 `alloc_netdev_mqs` 函数来分配的，驱动程序可以通过 `netdev_priv` 来得到该“私有数据”区的指针。第二个参数是字符型的 `name`，用来表示该网络接口的名称，其在用户空间是可见的。第三个参数是个函数指针，类型为 `void (*setup)(struct net_device *)`，程序员需要在其设备驱动程序中负责实现一个该类型的函数，在调用 `alloc_netdev` 时作为实参传入，`alloc_netdev_mqs` 将调用该函数来初始化 `net_device` 对象中余下的一部分成员变量。

下面考察一下 `alloc_netdev_mqs` 函数的实现细节，整体上该函数包含了两大部分，分别是内存分配和初始化。首先分配的是 `net_device` 对象的空间，如果设备驱动程序需要拥有自己的“私有数据”区，那么将在 `sizeof(struct net_device)` 的基础上对齐到 32 字节的整数倍，然后加上“私有数据”区的大小，之后用 `kzalloc` 函数分配出 `net_device` 对象和“私有数据”区所在的空间。函数中另一个要分配的重要数据结构是 `struct netdev_queue`，该结构的实例用来表示设备所拥有的队列。关于这个结构的用法，将在本章后续部分予以介绍，当前仍将注意力放在 `alloc_netdev_mqs` 函数上。函数调用 `kcalloc` 来分配 `netdev_queue` 的对象，内核中 `kcalloc` 用来为一个数组分配内存空间，在当前的上下文中，`kcalloc` 将分配只有一个元素的队列，该元素的大小是 `sizeof(struct netdev_queue)`，`kcalloc` 会将分配出的内存空间全部初始化为 0。

接下来是初始化环节，`dev_addr_init(dev)` 用来初始化 `dev` 对象中的硬件地址链表 `dev_addrs`，函数将在该链表中加入一个地址值全为 0 的硬件地址对象，然后将 `dev` 的 `dev_addr` 成员指向该硬件地址对象。`dev_mc_init` 和 `dev_uc_init` 函数分别用来初始化当前设备对象的组播和

单播 MAC 地址列表。接下来函数初始化发送与接收队列的相关成员。

函数的最后在设定设备名称前调用了传入的 `setup` 函数指针,这给了网络设备驱动程序一个机会以初始化一些在当前函数中尚未初始化的 `net_device` 对象成员。

现实中遇到的网络设备绝大多数属于以太网卡设备,所以设备驱动程序直接使用 `alloc_netdev` 函数的机会并不多。为了方便这种情况下的设备驱动程序的编写,内核特别针对以太网设备提供了一个 `alloc_etherdev` 宏,它其实是对刚讨论过的 `alloc_netdev_mqs` 函数的一个封装,专门用来分配并初始化一个以太网设备。`alloc_etherdev` 宏的定义如下:

```
<include/linux/etherdevice.h>
-----
#define alloc_etherdev(sizeof_priv) alloc_etherdev_mq(sizeof_priv, 1)
#define alloc_etherdev_mq(sizeof_priv, count) alloc_etherdev_mqs(sizeof_priv, count, count)
```

可以看到 `alloc_etherdev` 最终调用了 `alloc_etherdev_mqs`, 后者的定义如下:

```
<net/ethernet/eth.c>
-----
struct net_device *alloc_etherdev_mqs(int sizeof_priv, unsigned int txqs,
                                     unsigned int rxqs)
{
    return alloc_netdev_mqs(sizeof_priv, "eth%d", ether_setup, txqs, rxqs);
}
```

要注意的是调用时使用的参数,以太网设备名称被冠以"eth%d",用以初始化设备的 `setup` 函数则变成了 `ether_setup`,后者用来初始化 `net_device` 对象为以太网设备。`ether_setup` 函数的具体实现是:

```
<net/ethernet/eth.c>
-----
void ether_setup(struct net_device *dev)
{
    dev->header_ops      = &eth_header_ops;
    dev->type             = ARPHRD_ETHER;
    dev->hard_header_len  = ETH_HLEN;
    dev->mtu              = ETH_DATA_LEN;
    dev->addr_len         = ETH_ALEN;
    dev->tx_queue_len     = 1000;    /* Ethernet wants good queues */
    dev->flags             = IFF_BROADCAST|IFF_MULTICAST;
    memset(dev->broadcast, 0xFF, ETH_ALEN);
}
```

所以,对于以太网设备的驱动程序而言,应该使用 `alloc_etherdev` 函数来分配 `net_device` 对象,其内部会调用 `ether_setup` 将新分配出的 `net_device` 对象初始化为一个以太网设备。

如读者所猜想的那样,如果手边的设备不幸不是一个以太网设备,那么内核同样为此封装了 `alloc_netdev` 函数,只不过传入的用做设备初始化的参数 `setup` 的实参会根据具体的设备

而有所不同，以下是常见的一些网络设备在 Linux 内核中对应的 `alloc_xxx` 函数：

- 光纤分布式数据接口 FDDI (Fiber Distributed Data Interface)： `alloc_fddidev(int sizeof_priv)`。
- 令牌环网络设备 TR (Token Ring)： `alloc_trdev(int sizeof_priv)`。
- Apple LocalTalk 设备： `alloc_ltalkdev(int sizeof_priv)`。
- 高性能并行接口 HIPPI (High-Performance Parallel Interface)： `alloc_hippi_dev(int sizeof_priv)`。
- 光纤通道设备 FC (Fiber Channel)： `alloc_fcdev(int sizeof_priv)`。

作为通用的规则，当驱动所在的模块从系统中移除时，设备驱动程序应该负责释放 `alloc_netdev` 所分配的系统资源，内核为此提供了对应的函数 `free_netdev`，其代码如下：

`<net/core/dev.c>`

```
void free_netdev(struct net_device *dev)
{
    struct napi_struct *p, *n;
    release_net(dev_net(dev));
    kfree(dev->_tx);
    kfree(rcu_dereference_raw(dev->ingress_queue));

    /* Flush device addresses */
    dev_addr_flush(dev);

    /* Clear ethtool n-tuple list */
    ethtool_ntuple_flush(dev);

    list_for_each_entry_safe(p, n, &dev->napi_list, dev_list)
        netif_napi_del(p);

    free_percpu(dev->pcpu_refcnt);
    dev->pcpu_refcnt = NULL;

    /* Compatibility with error handling in drivers */
    if (dev->reg_state == NETREG_UNINITIALIZED) {
        kfree((char *)dev - dev->padded);
        return;
    }

    BUG_ON(dev->reg_state != NETREG_UNREGISTERED);
    dev->reg_state = NETREG_RELEASED;
```

```

    /* will free via device release */
    put_device(&dev->dev);
}

```

函数的主要功能在于释放 alloc_netdev 分配的资源，同时更新对应的设备状态 reg_state，使当前设备所对应的内核对象 device 的引用计数减 1，如果当前函数的调用是对该设备对象的最后一个引用，那么设备所在的空间将最终被释放。

12.2 网络设备的注册

前面讨论了如何分配并初始化一个新的 net_device 对象，它是当前网络设备驱动程序所控制的网络设备的一个软件抽象，如果没有进一步的动作，系统并不会意识到有这样一个网络设备存在，更具体地，当 Linux 内核的网络子系统需要发送或者接收一个网络数据包时，它不会调用到该 net_device 对象提供的函数。所以理所当然地，在驱动程序分配出一个新的 net_device 对象并将其初始化之后，接下来就需要把它注册到系统中。这一任务由内核提供的 register_netdev 函数来完成，其在内核源码中的定义是：

```

<net/core/dev.c>
-----
int register_netdev(struct net_device *dev)
{
    ...
    if (strchr(dev->name, '%')) {
        err = dev_alloc_name(dev, dev->name);
        ...
    }
    err = register_netdevice(dev);
    ...
}

```

register_netdev 函数主体由两部分构成：一是调用 dev_alloc_name 来为设备分配一个接口的名称；二是调用 register_netdevice 完成设备的注册工作。此处的重点是网络设备的注册，所以要重点讨论 register_netdevice 函数，后者在 Linux 内核源码中的实现主体结构为：

```

<net/core/dev.c>
-----
int register_netdevice(struct net_device *dev)
{
    int ret;
    struct net *net = dev_net(dev);
    ...
    spin_lock_init(&dev->addr_list_lock);
    netdev_set_addr_lockdep_class(dev);
}

```

```

dev->iflink = -1;

/* Init, if this function is available */
if (dev->netdev_ops->ndo_init) {
    ret = dev->netdev_ops->ndo_init(dev);
    if (ret) {
        if (ret > 0)
            ret = -EIO;
        goto out;
    }
}

ret = dev_get_valid_name(dev, dev->name, 0);
if (ret)
    goto err_uninit;

dev->ifindex = dev_new_index(net);
if (dev->iflink == -1)
    dev->iflink = dev->ifindex;

/* Transfer changeable features to wanted_features and enable
 * software offloads (GSO and GRO).
 */
dev->hw_features |= NETIF_F_SOFT_FEATURES;
dev->features |= NETIF_F_SOFT_FEATURES;
dev->wanted_features = dev->features & dev->hw_features;

/* Enable GRO and NETIF_F_HIGHDMA for vlans by default,
 * vlan_dev_init() will do the dev->features check, so these features
 * are enabled only if supported by underlying device.
 */
dev->vlan_features |= (NETIF_F_GRO | NETIF_F_HIGHDMA);

ret = call_netdevice_notifiers(NETDEV_POST_INIT, dev);
ret = notifier_to_errno(ret);
if (ret)
    goto err_uninit;

ret = netdev_register_kobject(dev);
if (ret)
    goto err_uninit;
dev->reg_state = NETREG_REGISTERED;

netdev_update_features(dev);

/*

```



```

    *   Default initial state at registry is that the
    *   device is present.
    */

    set_bit(__LINK_STATE_PRESENT, &dev->state);

    dev_init_scheduler(dev);
    dev_hold(dev);
    list_netdevice(dev);

    /* Notify protocols, that a new device appeared. */
    ret = call_netdevice_notifiers(NETDEV_REGISTER, dev);
    ret = notifier_to_errno(ret);
    if (ret) {
        rollback_registered(dev);
        dev->reg_state = NETREG_UNREGISTERED;
    }
    /*
     *   Prevent userspace races by waiting until the network
     *   device is fully setup before sending notifications.
     */
    if (!dev->rtnl_link_ops ||
        dev->rtnl_link_state == RTNL_LINK_INITIALIZED)
        rtmsg_ifinfo(RTM_NEWLINK, dev, ~0U);

out:
    return ret;

err_uninit:
    if (dev->netdev_ops->ndo_uninit)
        dev->netdev_ops->ndo_uninit(dev);
    goto out;
}

```

该函数被调用前，当前的网络设备对象 `dev` 在系统中注册的状态应该是 `NETREG_UNINITIALIZED`，之后如果设备对象定义有特定于当前设备的初始化函数，那么就调用这个初始化函数。`dev_new_index` 函数用来在当前设备所在的命名空间为设备所提供的接口寻找一个唯一的接口索引值。在 `register_netdevice` 这个函数中我们真正关注的重点函数应该是 `netdev_register_kobject`，它通过 Linux 设备驱动模型来向系统中添加当前的设备。`netdev_register_kobject` 函数首先获取当前网络设备对象的 `dev` 成员，通过前面对网络设备数据结构 `net_device` 的解读，`dev` 成员是一个 `struct device` 类型的变量，内核通过它将网络设备变成一个内核对象（`kobject`），继而通过 Linux 的设备模型来操控当前的网络设备，所以在 `netdev_register_kobject` 函数中可看到如下代码：

<net/core/net-sysfs.c>

```
int netdev_register_kobject(struct net_device *net)
{
    struct device *dev = &(net->dev);

    ...
    device_initialize(dev);
    dev->class = &net_class;
    ...
    device_add(dev);
}
```

本书“Linux 设备驱动模型”一章中已经详细讨论了有关内核对象 `device` 的操作，熟悉设备驱动模型的读者想必对这里的概念不会陌生，总之，网络设备内嵌的 `dev` 成员作为一个内核对象被加到了系统中，并通过 `sysfs` 文件系统向用户空间披露了它的存在，这正是 `netdev_register_kobject` 函数所要完成的核心功能。

现在继续回过头来看网络设备的注册过程，`register_netdevice` 函数在通过 `netdev_register_kobject` 将当前设备添加进系统之后，它将设备对象的 `reg_state` 成员设置成 `NETREG_REGISTERED` 状态，这标志着网络设备的注册过程已经结束。此处另一个值得关注的函数是 `list_netdevice`，其源码实现为：

<net/core/dev.c>

```
static int list_netdevice(struct net_device *dev)
{
    struct net *net = dev_net(dev);

    ...
    write_lock_bh(&dev_base_lock);
    list_add_tail_rcu(&dev->dev_list, &net->dev_base_head);
    hlist_add_head_rcu(&dev->name_hlist, dev_name_hash(net, dev->name));
    hlist_add_head_rcu(&dev->index_hlist,
                      dev_index_hash(net, dev->ifindex));
    write_unlock_bh(&dev_base_lock);
    return 0;
}
```

从函数的上述实现可以看到，它将当前设备对象加入到特定命名空间的几个散列链表中，这样当网络子系统高层代码需要发送数据包时，它将能通过这些散列链表找到特定的网络设备，在本章后续讨论驱动程序中接收和发送数据包的实现部分时将会再次看到该散列表的用途。当 `list_netdevice` 调用完毕时，`register_netdev` 的核心操作基本上都已经结束，所以函数接下来要做的只是一些辅助性质的善后工作，诸如通知高层协议模块一个新的网络设备加到了系统中。所以尽管 `register_netdevice` 函数的代码量不少，但是核心的功能其实只在 `device_add` 和 `list_netdevice` 这两个函数身上。一旦当前的设备被成功注册进系统，就意味着设备所提供的功能已经可由驱动模块所暴露的接口为外部其他模块所调用，因此

合理的逻辑顺序应该是只当设备所要完成的功能接口函数全部就绪后，设备模块才最终向系统注册该设备。在 `register_netdevice` 函数实现的最后部分，还有一个对 `dev_init_scheduler` 函数的调用，关于这个函数在此处的作用我们将在本章稍后的“传输超时”一节中再予以讨论。至于内核的网络子系统高层在发送一个数据包时，如何确定由哪一个 NIC 设备（比如当前系统中拥有不止一个激活的 NIC 设备）来发送，那其实是“路由”相关的话题。

在 Linux 环境下通过 `route` 命令可以查看到当前系统的路由信息，例如：

```
dennis@AMDLinuxFGL:/$ route
Kernel IP routing table
Destination      Gateway          Genmask         Flags         Metric        Ref  Use  Iface
10.237.74.0      *               255.255.254.0   U             1             0    0    eth0
link-local       *               255.255.0.0     U            1000          0    0    eth0
default          rtp002704rts.am 0.0.0.0         UG            0             0    0    eth0
```

当网络子系统高层要发送一个数据包时，通过上述路由表得到接口信息，比如上面的 `eth0`，然后再到设备列表中查找对应的设备。所以当 一个 NIC 设备向系统成功注册后，它将被纳入到系统的网络设备列表管理体系中，这意味着从那以后它将“暴露”在网络子系统的高层代码之中，因此其携带的设备方法随时可能被高层代码所“征用”。

与设备注册过程相反，当驱动所在的模块要从系统中移除时，需要调用相应的设备注销函数 `unregister_netdev`，其实现代码为：

```
<net/core/dev.c>
-----
void unregister_netdev(struct net_device *dev)
{
    rtnl_lock();
    unregister_netdevice(dev);
    rtnl_unlock();
}
```

在 `unregister_netdevice` 函数发起的调用链中，`rollback_registered_many` 函数承载着设备注销的实质性任务，基本上 `unregister_netdev` 完成与 `register_netdev` 相反的功能。一个网络设备在从系统中注销后，将不会再被网络子系统的高层所使用。

12.3 设备方法

前面讨论了内核中如何分配及向系统注册一个网络设备对象，这只是网络设备驱动程序框架的一个基本步骤，因为归根结底，系统中的网络设备并不只单单向系统注册一下就可以大功告成，需知它最核心的功能是用来收发网络数据包，除此之外还需要提供一些配置与


```

        void                (*ndo_vlan_rx_kill_vid)(struct net_device *dev,
                                                    unsigned short vid);
#ifdef CONFIG_NET_POLL_CONTROLLER
        void                (*ndo_poll_controller)(struct net_device *dev);
        void                (*ndo_netpoll_cleanup)(struct net_device *dev);
#endif
        int                 (*ndo_set_vf_mac)(struct net_device *dev,
                                              int queue, u8 *mac);
        int                 (*ndo_set_vf_vlan)(struct net_device *dev,
                                              int queue, u16 vlan, u8 qos);
        int                 (*ndo_set_vf_tx_rate)(struct net_device *dev,
                                              int vf, int rate);
        int                 (*ndo_get_vf_config)(struct net_device *dev,
                                              int vf,
                                              struct ifla_vf_info *ivf);
        int                 (*ndo_set_vf_port)(struct net_device *dev,
                                              int vf,
                                              struct nlattr *port[]);
        int                 (*ndo_get_vf_port)(struct net_device *dev,
                                              int vf, struct sk_buff *skb);
#ifdef CONFIG_FCOE || defined(CONFIG_FCOE_MODULE)
        int                 (*ndo_fcoe_enable)(struct net_device *dev);
        int                 (*ndo_fcoe_disable)(struct net_device *dev);
        int                 (*ndo_fcoe_ddp_setup)(struct net_device *dev,
                                              u16 xid,
                                              struct scatterlist *sgl,
                                              unsigned int sgc);
        int                 (*ndo_fcoe_ddp_done)(struct net_device *dev,
                                              u16 xid);

#define NETDEV_FCOE_WWNN 0
#define NETDEV_FCOE_WWPN 1
        int                 (*ndo_fcoe_get_wwn)(struct net_device *dev,
                                              u64 *wwn, int type);
#endif
};

```

这是个颇为全面的针对网络设备的操作集，限于篇幅原因这里并不打算详细介绍其中每个成员函数的实现策略，在此我们会选择现实中使用最多最核心的一些成员，来阐述其在设备驱动程序中的实现机制。需要说明的是，对于设备驱动程序而言，一个网络设备对象操作集中的函数是可选的，这意味着驱动程序需要根据自己所管理设备的实际功能决定实现哪些函数，如果某一功能在驱动程序中没有实现，那么对应的函数指针应该是个空指针。

12.3.1 设备初始化

```
int (*ndo_init)(struct net_device *dev)
```

这个函数我们在前面讨论网络设备的注册时，在 `register_netdev` 函数中已经看到过它的身影，正如名称所提示的那样，该函数用来对当前正在向系统注册的网络设备对象做一些晚期阶段的初始化工作。这里之所以说晚期阶段的初始化工作，是因为在前面讨论用于分配一个 `net_device` 对象的 `alloc_netdev` 函数时，已经看到过 `alloc_netdev` 会对分配出的 `net_device` 对象的部分成员进行初始化的工作，所以当设备驱动程序调用 `register_netdev` 函数向系统注册一个网络设备对象时，如果该设备对象的 `net_device_ops` 操作集中定义了 `ndo_init` 函数，它将有机会被 `register_netdev` 函数所调用。因为 `alloc_netdev` 中的初始化更多是从内核的角度产生的一个通用的过程，而如果设备需要一些特定的与众不同的初始化工作，则不应该忽略掉此处的 `ndo_init` 函数。`ndo_init` 函数的执行过程如果失败，应该返回一个错误码，后者将由 `register_netdev` 函数返回，这也意味着设备注册过程的失败。

在 `net_device_ops` 操作集中与 `ndo_init` 函数对应的是 `ndo_uninit`，很显然这是个 `ndo_init` 的逆向过程，所以它在驱动程序中主要在一些收拾残局的场景中被使用，比如在 `register_netdev` 函数中，如果在 `ndo_init` 函数调用之后的一些流程中出现非正常的情况，`ndo_uninit` 将有机会被调用以恢复 `ndo_init` 所做的一些工作，又或者比如当设备所在的模块从系统中移除时所调用的 `unregister_netdev` 函数，`ndo_uninit` 也有机会被执行到。

12.3.2 设备接口的打开与停止

当设备所在的网络接口被激活时，比如使用 `ifconfig` 激活某一网络接口时，接口将被打开。此时如果对应的驱动程序提供了 `ndo_open` 函数，那么它将被调用。不同设备的 `ndo_open` 函数所完成的工作也是不一样的，这里并没有一个通用的准则，一些常见的操作包括分配接收/发送网络数据包所需要的资源，初始化设备的硬件中断并向系统注册中断，启动 `watchdog` 定时器，通知上层网络子系统当前接口已就绪等，不一而论。与之相反的过程则发生在 `ndo_stop` 函数中，它在当前的网络设备接口被关闭（`shutdown`、`deactive`...）时被调用，通常完成的工作与 `ndo_open` 相反。

12.3.3 数据包的发送

这是本章要重点讨论的内容之一，因为网络设备的核心功能就是发送和接收数据包。数据包的发送函数实现在 `net_device_ops` 的 `ndo_start_xmit` 成员中，其原型是：

```
netdev_tx_t (*ndo_start_xmit)(struct sk_buff *skb, struct net_device *dev);
```

参数 `skb` 的类型 `struct sk_buff` 是网络设备驱动程序中另一个重要的数据结构，通常叫做套接字缓冲区，本章接下来会有专门一节讨论该数据结构及内核提供的操作该结构对象的相关接口函数，目前只要知道待发送的数据包的数据就包含在 `skb` 参数中就够了，如果再具体一些，`skb->data` 指向要发送的数据包在内存中的位置，而 `skb->len` 则是以字节为单位的该数据包的长度。第二个参数 `dev` 自然就是本次用来发送网络数据包的设备对象了。

不同于网络数据包的接收，发送过程算得上是个同步的过程，当 Linux 内核中的网络子系统上层部分有数据包需要发送时，它将通过网络设备的 `ndo_start_xmit` 函数来发送该数据包，网络设备驱动程序的核心任务之一便是实现该函数。很显然，这是个跟具体的网络设备硬件相关的函数，作为一般的规则，驱动程序通常需要使用 DMA 的方式将套接字缓冲区中的数据传输到网络设备的存储空间中，然后由网络设备的硬件逻辑负责把设备存储空间中刚接收到的数据发送出去，在数据成功发送后，设备会产生一个硬件中断以通知驱动程序进行相应的处理，比如释放上层传下来的套接字缓冲区，进行数据统计等。这里为了叙述方便，抽象出图 12-4 所示模型来描述设备驱动程序实现发送函数的一般性原理：

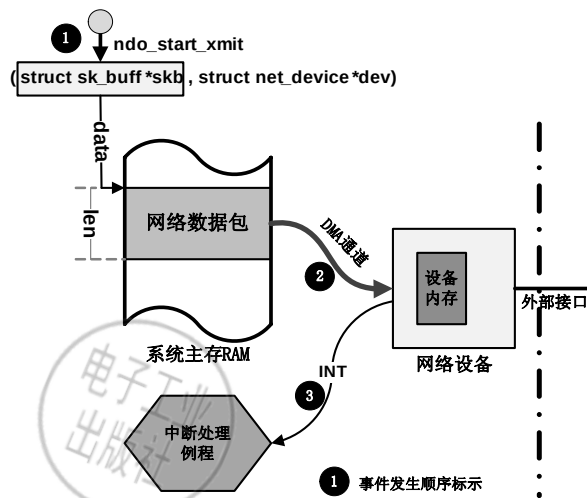


图 12-4 网络设备驱动程序数据包发送模型

图中，当网络子系统上层有数据包要发送时，通过调用网络设备驱动例程中实现的 `ndo_start_xmit` 函数，将要发送的数据包封装在套接字缓冲区 `skb` 参数中。在驱动程序的发送数据包函数的具体实现中，它将首先在 `skb` 数据包所在主存中的数据块和网络设备内部的设备内存间建立一个 DMA 通道，然后启动该 DMA 通道将数据包由主存传输到设备内存中，之后便是由网络设备的硬件逻辑的电气特性来完成通过诸如 RJ45 等接口向外部世界发送设备内存中新接收的数据。网络设备的数据成功发送完毕后，将向处理器发出一个硬件中断，如此驱动程序的中断处理例程便会参与进来做一些数据包发送后的善后处理工作。

在上面的过程中，我们看到由于 DMA 通道的源端数据所在的缓冲区 `skb->data` 来自于内核的网络系统上层，换言之它不属于驱动程序所能控制的范围之内，所以现实中为了建立对应的 DMA 映射，一般多采用流式 DMA 映射，当然原理上采用一致性映射也是可行的，不过因为需要在 `skb->data` 与一致性缓冲区之间进行拷贝操作，因而可能会付出性能上的代价。

好奇心强的读者也许会希望了解从 `ndo_start_xmit` 往上去的代码执行路径的相关细节，这其实已经超出了网络设备驱动程序的范畴，因为它将进入 Linux 内核的网络子系统部分，这是个极其庞大复杂的模块，详细地讨论它也许需要一两本书的容量。不过在此我们可以简

单阐述一下协议高层的代码如何与底层的 `ndo_start_xmit` 关联起来, 直接调用 `ndo_start_xmit` 的是一个叫做 `dev_hard_start_xmit` 的函数, 它定义在 `net/core/dev.c` 文件中, 后者又被定义在同一文件中的 `dev_queue_xmit` 函数所调用, 它的原型是:

```
int dev_queue_xmit(struct sk_buff *skb)
```

在它的实现中, 我们会看到用来发送当前 `skb` 的设备已经包含在了 `skb` 的 `dev` 成员中:

```
struct net_device *dev = skb->dev;
```

这意味着更高层的代码在调用到 `dev_queue_xmit` 时, 其实已经获得了用来发送本次数据包 of 的底层网络设备, 如果由此上溯, 将可以看到从 `socket API` 系统调用到网络设备驱动程序最终发送出一个数据包的奇妙旅程。

还是回到当前的主题, 理想情况下, 在新建的 `DMA` 通道成功地将套接字缓冲区 `skb` 中的数据传输到设备内存后, 网络设备的硬件发送逻辑会圆满完成本次发送任务。然而事实并非总是如想象中的那样美好与理想化, 软件层面的高速性与实际硬件的发送速度几乎总是会存在矛盾, 此时作为网络设备驱动程序员, 必须提供相应的处理机制以尽可能确保在数据包的传输过程中不会出现丢包的现象, 这个话题也是本章后续小节要讨论的内容。

为使以上的讨论具体化, 这里给出一个网络设备驱动程序中数据包发送函数的具体实现代码³。以下代码片段来自某一 `ARM` 平台:

```
static int
fec_enet_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    ...
    struct fec_enet_private *fep;
    volatile fec_t      *fecp;
    volatile cbd_t      *bdp;
    unsigned short      status;
    unsigned long flags;

    fep = netdev_priv(dev);
    fecp = (volatile fec_t*)dev->base_addr;

    if (!fep->link) {
        /* Link is down or autonegotiation is in progress. */
        return 1;
    }

    spin_lock_irqsave(&fep->hw_lock, flags);
```

³ 该网络驱动程序基于早期的 `Linux` 版本, 此处主要用来展示一个具体的实例。


```
/* Fill in a Tx ring entry */
bdp = fep->cur_tx;

status = bdp->cbd_sc;
/* Clear all of the status flags. */
status &= ~BD_ENET_TX_STATS;

/* Set buffer length and buffer pointer. */
bdp->cbd_bufaddr = __pa(skb->data);
bdp->cbd_datlen = skb->len;

/*
 * On some FEC implementations data must be aligned on
 * 4-byte boundaries. Use bounce buffers to copy data
 * and get it aligned. Ugh.
 */
if ((bdp->cbd_bufaddr & FEC_ALIGNMENT) {
    unsigned int index;
    index = bdp - fep->tx_bd_base;
    memcpy(fep->tx_bounce[index], (void *) skb->data, skb->len);
    bdp->cbd_bufaddr = __pa(fep->tx_bounce[index]);
}

/* Save skb pointer. */
fep->tx_skbuff[fep->skb_cur] = skb;

dev->stats.tx_bytes += skb->len;
fep->skb_cur = (fep->skb_cur+1) & TX_RING_MOD_MASK;

/* Push the data cache so the CPM does not get stale memory
 * data.
 */
fec_dcache_flush_range(__va(bdp->cbd_bufaddr), __va(bdp->cbd_bufaddr) +
    bdp->cbd_datlen);

/* Send it on its way. Tell FEC it's ready, interrupt when done,
 * it's the last BD of the frame, and to put the CRC on the end.
 */

status |= (BD_ENET_TX_READY | BD_ENET_TX_INTR
    | BD_ENET_TX_LAST | BD_ENET_TX_TC);
bdp->cbd_sc = status;

dev->trans_start = jiffies;

/* Trigger transmission start */
```

```

fecp->fec_x_des_active = 0x01000000;

/* If this was the last BD in the ring, start at the beginning again*/
if (status & BD_ENET_TX_WRAP) {
    bdp = fecp->tx_bd_base;
} else {
    bdp++;
}
/*flow control, tell the uplayer don't send the package now*/
if (bdp == fecp->dirty_tx) {
    fecp->tx_full = 1;
    netif_stop_queue(dev);
}

fecp->cur_tx = (cbd_t *)bdp;
spin_unlock_irqrestore(&fecp->hw_lock, flags);

return 0;
}

```

上述代码是一个基于 FEC（Fast Ethernet Controller）的发送数据包的函数，硬件设备会接收一个被称为缓冲区描述符（Buffer Descriptor）的数据对象，它非常类似于常见的 DMA 描述符，里面包含有缓冲区的地址以及待操作数据包的长度，控制器也会根据数据包的实际发送与接收的操作结果更新该描述符中的状态字段，因为与实际硬件关联性非常大，所以此处不会详细讲解该 `fec_enet_start_xmit`。可以看到这个函数的主体框架采用了典型的流式 DMA 映射来建立 DMA 通道以传输 `skb` 中的网络数据包，其中 `fec_dcache_flush_range` 函数的调用目的是在进行 DMA 传输前将当前 DMA 源地址对应的 cache 中的数据 flush 到主存中，以防止 DMA 因 cache 存在的缘故将陈旧的数据传输给网络设备。在这之后，函数用 `fecp->fec_x_des_active = 0x01000000` 来操作硬件设备的寄存器以触发 DMA 传输操作，后者就完全是网卡硬件逻辑要完成的事情了：将系统主存中的数据包通过内置的 DMA 控制器传输到其内部的缓冲区（设备内存、FIFO 等）中，然后借助网线等物理传输介质将数据发送出去。当一个数据包被成功发送出去，或者发送过程中出现了错误状况，网卡设备将以中断的方式通知其驱动程序。

需要注意的是，网络子系统高层传下来的套接字缓冲区需要由设备驱动程序在完成一次 DMA 传输后负责释放，设备驱动程序一般在中断处理例程中完成这个任务⁴，所以在上面的 `fec_enet_start_xmit` 函数中没有看到类似 `dev_kfree_skb(skb)` 这样的调用。关于释放 `skb`

⁴ 严格意义上，成功发送出去的数据包所在的 `skb` 真正的释放工作在函数 `net_tx_action` 中完成，它是 `NET_TX_SOFTIRQ` 所对应的 `softirq` 处理例程。内核认为释放 `skb` 缓冲区是一项比较耗时的操作，而驱动程序中的中断处理例程只应完成最关键的操作。

的话题，将在后续的“中断处理”和“套接字缓冲区”小节中予以讨论。

如果对数据包的发送过程作个简单小结，那就是：一个数据包的发送过程逻辑上可以分成两个独立的部分，按照时间顺序，分别是网络子系统部分和设备驱动程序部分。网络子系统部分在整个 Linux 网络部分源码中是独立于底层的网络硬件的，出于性能及可靠性等因素的考虑，网络子系统部分实现有一个传输队列，系统中每个 CPU 都拥有自己的传输队列，每个要发送的数据包都会先放到传输队列中。真正的发送过程发生在网络设备驱动程序所实现的 `ndo_start_xmit` 函数中，后者的实现依赖于具体的硬件设备，通常硬件在当前帧传输结束时会以中断的方式通知驱动程序。

12.3.4 网络数据包发送过程中的流控机制

理想情况下网络数据包的发送也许很简单，在软件的控制下由建立好的 DMA 通道去传输数据就可以了，但是现实情况往往比较复杂，比如当 `ndo_start_xmit` 函数返回时，并不意味着实际的硬件设备（网卡）已将设备内存中刚刚获得的数据包全部成功发送了出去。换句话说，软件层面的 `ndo_start_xmit` 调用过程与网络设备的实际数据发送行为之间是异步的，其各自的行为是独立的。

由此带来的问题是，当内核的网络子系统有新的数据包需要发送时，它可能会再次调用 `ndo_start_xmit` 函数，当后者被调用时，前次的调用所传递到网络设备内存的数据包也许还没有发送完。问题的本质在于网络子系统的高层代码可以快速“发送”大量的数据包，但是底层网络设备的实际发送速度无法与之匹配，内核因此需要维护一个发送队列，显然这对提升网络系统的性能是有帮助的。如果将这个问题稍稍扩大：内核子系统上层组件在很短的时间里调用了大量的 `ndo_start_xmit` 函数，CPU 执行这个过程总是很快，但是网络设备将其内在存储区中的数据包发送出去就没有那么神速了，于是此种情况导致的一个显而易见的问题是，网络设备的设备内存空间会很快被消耗光，而再也没有能力去接收网络子系统高层所发送来的数据包。

针对这种情况，驱动程序需要一种机制，当发现网络设备的内部存储空间暂时无法使用时，可以通知网络子系统的高层暂停数据包的发送，显然这是一种软件层面的流控机制，可以避免对 CPU 资源的无谓浪费：如果内核提前得知下层的网络设备不可能将一个数据包成功发送出去，就没有必要再去调用驱动程序中实现的发送函数。一个更为智能的内核行为也许可以通过对发送队列的仔细观察来洞察底层硬件的发送结果，从而决定是否调用驱动程序的 `ndo_start_xmit` 函数来发送当前的分组，但无论是逻辑上还是实现的复杂度上，让驱动程序主动告知内核要更加自然，因为没有谁比设备驱动程序自身更了解其所控制的硬件行为。驱动程序所要完成的这种流控机制显然需要来自内核中网络子系统代码的支持，内核为此专门给设备驱动程序提供了这样一个函数 `netif_stop_queue`，该函数的主要作用是让设备驱动程序告诉内核的网络子系统高层：当前底层的网络设备硬件无法继续传输数据包，

高层代码需要停止数据包的发送。这个函数的实现非常简单，我们把它稍作改写，其代码如下：

```
<include/linux/netdevice.h>
-----
static inline void netif_stop_queue(struct net_device *dev)
{
    struct netdev_queue *dev_queue = &dev->_tx[0];
    set_bit(__QUEUE_STATE_XOFF, &dev_queue->state);
}
```

netif_stop_queue 其实就是将 net_device 对象 dev 中的发送队列_tx[0]的状态 state 的 __QUEUE_STATE_XOFF 位置 1，后者是一 netdev_queue_state_t 类型的枚举变量，用来表示 net_device 对象中队列的状态。netdev_queue_state_t 在内核中的定义为：

```
<include/linux/netdevice.h>
-----
enum netdev_queue_state_t {
    __QUEUE_STATE_XOFF,
    __QUEUE_STATE_FROZEN,
};
```

至于 netif_stop_queue 为什么可以让高层网络代码停止调用 ndo_start_xmit 函数继续发送网络包，只需沿着 netif_stop_queue 在内核中的调用链向上追溯很快便会获得答案。在内核的 dev_hard_start_xmit 函数中，可以看到如下代码：

```
<net/core/dev.c>
-----
int dev_hard_start_xmit(struct sk_buff *skb, struct net_device *dev,
                       struct netdev_queue *txq)
{
    ...
    do {
        struct sk_buff *nskb = skb->next;
        skb->next = nskb->next;
        nskb->next = NULL;
        ...
        rc = ops->ndo_start_xmit(nskb, dev);
        ...
        txq_trans_update(txq);
        if (unlikely(netif_tx_queue_stopped(txq) && skb->next))
            return NETDEV_TX_BUSY;
    } while (skb->next);
    ...
}
```

在上述函数中，与数据包发送流控相关的代码出现在最后部分的 netif_tx_queue_stopped 代码中，如果网络设备驱动程序调用了 netif_stop_queue 函数，那么当前网络设备对象 dev 中

的发送队列状态中的 `__QUEUE_STATE_XOFF` 将被置位，此时 `netif_tx_queue_stopped(txq)` 将返回真，如果 `skb->next` 不为空，意味着有下一个数据包需要发送，`dev_hard_start_xmit` 函数将把一个错误码 `NETDEV_TX_BUSY` 传递到网络系统的高层，最顶层的 API 调用极可能获得一个类似“device busy”的错误信息，因此它将知道当前的发送没有成功。

与 `netif_stop_queue` 对应的另一个流控函数是 `netif_start_queue`，当网络设备驱动程序发现设备可以继续进行数据包的传输时，应该调用 `netif_start_queue` 函数通知上层可以继续发送数据包。可以很容易想象出 `netif_start_queue` 函数的实现应该只是清除掉被 `netif_stop_queue` 置 1 的 `__QUEUE_STATE_XOFF` 位，下面是 `netif_start_queue` 在内核中的实现：

```
<include/linux/netdevice.h>
```

```
static inline void netif_start_queue(struct net_device *dev)
{
    struct netdev_queue *dev_queue = &dev->_tx[0];
    clear_bit(__QUEUE_STATE_XOFF, &dev_queue->state);
}
```

现实中的网络设备驱动程序需要根据实际情况决定如何调用 `netif_stop_queue` 和 `netif_start_queue` 函数。一个典型的情形是，当打开一个网络设备的接口时（此时设备驱动程序中的 `ndo_open` 函数被调用），驱动程序需要调用 `netif_start_queue` 以告诉内核，网络子系统高层代码可以调用 `dev_hard_start_xmit` 进行数据包的发送；与此相反，当一个网络设备接口被关闭时（此时设备驱动程序中的 `ndo_close` 函数被调用），对应的驱动程序应该调用 `netif_stop_queue` 以通知上层代码。

流控中的另一个重要的函数是 `netif_wake_queue`，相对于 `netif_start_queue` 而言，`netif_wake_queue` 不仅需要考察当前设备发送队列的 `__QUEUE_STATE_XOFF` 状态位，还需要考察当前设备发送队列的 `qdisc` 成员所对应的状态 `__QDISC_STATE_SCHED`。`netif_wake_queue` 对这两个状态的操作逻辑是：如果当前设备发送队列的 `__QUEUE_STATE_XOFF` 位被置 1，则清除之，紧接着考察当前设备发送队列 `qdisc` 成员的 `__QDISC_STATE_SCHED` 位有没有被置 1，如果是表明当前设备的发送队列尚未加入 CPU 的发送队列（由一个 per-CPU 型的 `struct softnet_data` 变量来管理）中，将其加到 CPU 发送队列尾部同时调用 `raise_softirq_irqoff(NET_TX_SOFTIRQ)` 来触发发送中断处理流程的下半段（softirq）。图 12-5 揭示了 `netif_wake_queue` 的处理流程。

所以 `netif_start_queue` 只是简单地清除发送队列的 `__QUEUE_STATE_XOFF` 比特位，并不触发数据包的发送流程，而 `netif_wake_queue` 在清除 `__QUEUE_STATE_XOFF` 之后，会有机会重新触发网络子系统数据包的传输流程。

设备驱动程序在以下两种典型的情况下使用 `netif_wake_queue`：

- 看门狗定时器（watchdog timer）超时，此种情形下驱动程序的 `ndo_tx_timeout` 函数会

被内核调用以重新配置 NIC，使得因某种原因挂起的 NIC 可以重新开始工作。在网络设备挂起的时间段内，在当前的设备上可能存在其他的传输尝试，因此当看门狗定时器超时重置 NIC 之后，驱动程序需要调用 `netif_wake_queue` 先开启队列，然后为设备调度以重新发送在设备挂起期间进入到发送队列的数据包。此种情况可以引申到因网络设备的硬件错误所导致的中断处理例程中，在那里驱动程序同样需要调用 `netif_wake_queue` 来将设备重新纳入网络子系统的高层调度体系中。

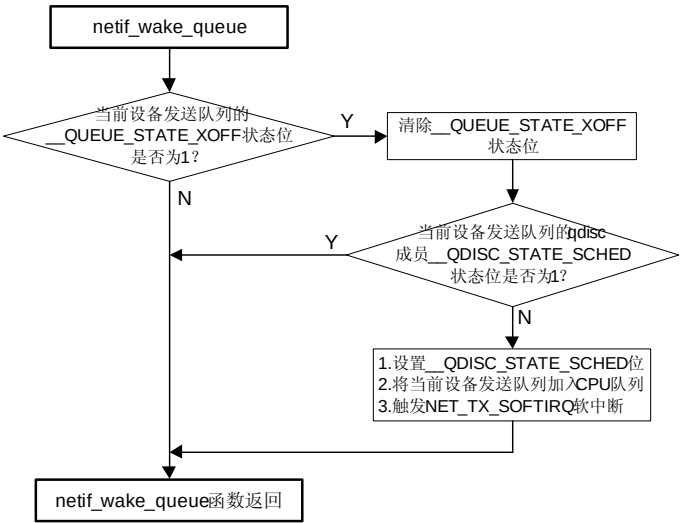


图 12-5 netif_wake_queue 处理流程

- 当设备通过中断通知驱动程序可以进行数据包的传输时（在此之前，驱动程序通常的做法是：因设备无法进行数据包的传输而通知高层关闭传输队列），在中断处理例程中，设备应该被唤醒，因为同样存在着设备队列被关闭期间发生传输的可能性，所以此种情况下设备驱动程序需要调用 `netif_wake_queue` 来为当前设备的传输进行重新调度。与此类似的一个情形是，NIC 在一次 DMA 传输之后中断处理器，告知数据包发送完毕，中断处理例程在通过 `netif_queue_stopped` 得知当前设备传输队列被关闭的情形下应该调用 `netif_wake_queue` 重启队列。

与网络数据包发送路径相对应，在接收数据包的路径上同样存在类似的流控机制，关于这个话题将延后到“数据包的接收”一节中予以讨论。

12.3.5 传输超时 (watchdog timeout)

对于网络子系统高层传下来的数据包，如果网络设备在指定的时间内因某种原因而没有发送出去，则会产生所谓传输超时的问题。显然，这里需要某种机制来发现问题，实际的 Linux 系统使用定时器来处理这一问题。幸运的是，Linux 内核中网络子系统模块已设计有相应的框架来应对此种状况，因此对网络设备驱动程序而言，针对传输超时问题所采取

的措施相对比较简单。基本上有两个步骤需要在驱动程序中完成：一个是在 `net_device` 实例所代表的网络设备对象 `dev` 的成员 `watchdog_timeo` 上设定超时定时器的到期时间；另一个则是实现 `net_device_ops` 中的 `ndo_tx_timeout` 函数，当 `watchdog_timeo` 设定的时间到期时，该函数将被网络子系统的代码所调用，设备驱动程序可以在自己的 `ndo_tx_timeout` 函数实现中完成对传输超时问题的处理。

在进一步讨论 `ndo_tx_timeout` 函数可能需要完成的任务前，我们打算用一定的篇幅讨论内核的网络子系统是如何与设备驱动程序中的 `ndo_tx_timeout` 函数合作来共同完成对可能出现的传输超时问题的处理的。事情的源起也许要回溯到前面已经讨论过的 `register_netdev` 函数中，在它的调用链中有一个名为 `dev_init_scheduler` 的函数，其源码实现为：

<net/sched/sch_generic.c>

```
void dev_init_scheduler(struct net_device *dev)
{
    dev->qdisc = &noop_qdisc;
    netdev_for_each_tx_queue(dev, dev_init_scheduler_queue, &noop_qdisc);
    dev_init_scheduler_queue(dev, &dev->rx_queue, &noop_qdisc);
    setup_timer(&dev->watchdog_timer, dev_watchdog, (unsigned long)dev);
}
```

该函数的实现中与此处讨论的主题相对应的是最后一行函数调用 `setup_timer`，后者为当前网络设备的 `watchdog_timer` 定时器设定了一个到期函数 `dev_watchdog`，很显然当 `watchdog_timer` 对象中的 `expires` 成员所指定的时间到期后，`dev_watchdog` 函数将会被调用。现在的问题是，Linux 系统中谁来负责设定网络设备对象的 `watchdog_timer` 成员中的到期时间 `expires`？另外，网络设备对象中的 `watchdog_timeo` 成员又是如何与 `watchdog_timer` 建立起联系的？换言之，当 `watchdog_timeo` 所指定的时间到期后，`ndo_tx_timeout` 是如何被调用的？

答案是，当一个网络设备接口被打开时，一个名为 `__netdev_watchdog_up` 的函数最终将被调用，以下是其完整实现：

<net/sched/sch_generic.c>

```
void __netdev_watchdog_up(struct net_device *dev)
{
    if (dev->netdev_ops->ndo_tx_timeout) {
        if (dev->watchdog_timeo <= 0)
            dev->watchdog_timeo = 5*HZ;
        if (!mod_timer(&dev->watchdog_timer,
            round_jiffies(jiffies + dev->watchdog_timeo)))
            dev_hold(dev);
    }
}
```

在 `mod_timer` 函数中, `__netdev_watchdog_up` 使用了 `round_jiffies(jiffies + dev->watchdog_timeo)` 来设定 `dev->watchdog_timer` 中的 `expires`。

如此, 当 `dev->watchdog_timeo` 所设定的时间到期时, `dev->watchdog_timer` 定时器中的到期函数将会被调用。之前看到在向系统注册一个网络设备时由 `dev_init_scheduler` 指定了到期函数为 `dev_watchdog`, 其核心实现是:

<net/sched/sch_generic.c>

```
static void dev_watchdog(unsigned long arg)
{
    struct net_device *dev = (struct net_device *)arg;
    ...
    for (i = 0; i < dev->num_tx_queues; i++) {
        struct netdev_queue *txq;
        txq = netdev_get_tx_queue(dev, i);
        /*
         * old device drivers set dev->trans_start
         */
        trans_start = txq->trans_start ? : dev->trans_start;
        if (netif_tx_queue_stopped(txq) &&
            time_after(jiffies, (trans_start + dev->watchdog_timeo))) {
            some_queue_timedout = 1;
            break;
        }
    }

    if (some_queue_timedout) {
        ...
        dev->netdev_ops->ndo_tx_timeout(dev);
    }
    if (!mod_timer(&dev->watchdog_timer, round_jiffies(jiffies + dev->watchdog_timeo)))
        dev_hold(dev);
    ...
}
```

函数的总体思想是, 如果发现某一传输队列处于停止状态并且当前已经过了指定的超时时间, 将导致对当前网络设备对象的 `ndo_tx_timeout` 函数的调用, 因此设备驱动程序有机会对该问题进行处理。

关于驱动程序中 `ndo_tx_timeout` 函数要完成的任务, 此处并没有一个通行的规则, 从逻辑上看它应该跟驱动程序中传输部分的代码联系比较紧密, 常见的操作包括在接口的统计信息中记录本次的传输错误, 调用 `netif_wake_queue` 函数以重启传输队列, 有时甚至需要重新 `reset` 当前网络设备等, 总之跟手边的硬件以及要完成的功能息息相关。

12.3.6 数据包的接收

相对于网络数据包的发送来说，接收过程要稍微复杂些，因为对驱动程序而言，数据包的到达是随机的，类似于一个异步的过程，通常当网络设备成功接收到一个数据包时，需要通过中断的方式引起驱动程序的干预。不同的网络设备在硬件逻辑设计上并不相同，在嵌入式领域，一些网络设备只需要驱动程序提供好 DMA 的描述符，在其他功能寄存器都配置好的情况下，当网络设备成功地接收到一个数据包时，该数据包往往已经由硬件设备自动地启动 DMA 传输到了描述符所指向的系统主存空间中，同时硬件设备也会自动更新描述符中的某些成员，比如本次接收的数据包的长度等。

如同网络数据包的发送一样，驱动程序中接收数据包的实现方法依然依赖于具体的硬件设备，但是通常驱动程序需要负责分配一个套接字缓冲区 skb 来容纳收到的数据包，然后将 skb 传递到网络子系统的上层代码中，后者负责释放该 skb 所占用的内存。DMA 的操作在这个过程中常常作为一个关键的步骤而存在，它将网络设备接收到的外部数据包从设备内存传输到系统内存中，现在所见到的几乎所有网卡设备都支持 DMA 操作，能够自发地将接收到的数据包传输到系统主存中。作为一种通用的抽象，图 12-6 展示了底层网络设备接收到一个数据包时设备驱动程序中的处理流程模型：

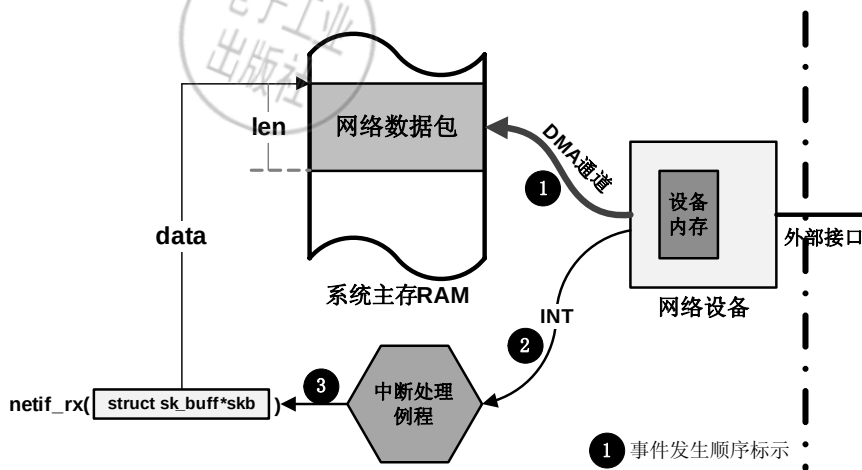


图 12-6 网络设备驱动程序数据包接收模型

与图 12-4 中的发送模型不同，在图 12-6 中，驱动程序中的控制流程由网络设备的中断所发起。通常网络设备驱动程序在其初始化过程中都会针对所控制的网络设备注册一个中断处理例程，网络设备中的很多情况都会引起一个中断从而将代码的执行路径引入到该中断处理例程中。网卡设备接收到一个数据包便是其中极为典型的一种，当该中断发生时，网卡设备接收到的数据包可能还在其自身的设备内存中，当然也可能已经通过网卡自身的 DMA 硬件逻辑传输到了系统主存中，这依赖于手头硬件的具体功能。但不管怎样，在这种情况下网络设备驱动程序都需要分配一个套接字缓冲区 skb，然后调用 netif_rx(skb) 将数据

包传递到网络子系统的高层代码中。分配一个套接字缓冲区 `skb` 并确保数据包从设备内存传输到了 `skb` 所指向的系统内存, 然后再调用 `netif_rx(skb)` 是网络设备驱动程序中数据包接收的经典处理流程。

本来任务可以到此结束, 但是还有一些细节值得挖掘一下, 细心的读者应该已注意到 `netif_rx` 函数的调用是在中断处理例程中发生的, 换句话说 `netif_rx` 要运行在中断上下文中, 所以需要尽可能快地返回以使 CPU 可以接收下一个中断。不过基于网络子系统的多层协议的复杂性, 如果要通过 `netif_rx` 发动的调用链将接收到的数据包最终传递到最后的接收者那里, 时间上的开销一定不会小而且也不现实。内核对此的解决方法是大家都熟悉的所谓软中断 `softirq`, 好奇的读者可以沿着 `netif_rx` 函数往上追溯一下, 它其实是将接收到的数据包放到一个接收队列上⁵, 然后触发 `NET_RX_SOFTIRQ` 软中断。在 `netif_rx` 调用链中, 触发该软中断的任务由 `napi_schedule` 函数完成, 在内核中的源码为:

```
<net/core/dev.c>
-----
static inline void ____napi_schedule(struct softnet_data *sd,
                                     struct napi_struct *napi)
{
    list_add_tail(&napi->poll_list, &sd->poll_list);
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

而对应 `NET_RX_SOFTIRQ` 的软中断处理例程则早在 Linux 系统启动的初始化阶段便由 `net_dev_init` 函数完成了:

```
<net/core/dev.c>
-----
static int __init net_dev_init(void)
{
    ...
    open_softirq(NET_TX_SOFTIRQ, net_tx_action);
    open_softirq(NET_RX_SOFTIRQ, net_rx_action);
    ...
}
```

对 `net_rx_action` 等函数的讨论已经超出了本书的范围, 那是内核中网络子系统要完成的任务, 大体上它们需要将接收队列上的网络数据包向高层传递。对于驱动程序而言, 我们现在知道 `netif_rx` 要做的事情其实相当明确: 将接收到的数据包加入一个队列, 触发一个软中断标志位。正因如此, `netif_rx` 在中断上下文中可以快速返回, 返回时基本上就意味着接收数据包的中断处理任务已经完成, 可以退出中断上下文了。对驱动程序员而言, 看看 `netif_rx` 函数的原型也许更为重要:

⁵ 为了在多处理器系统上获得最佳的性能, 内核为系统中每个 CPU 都分配这样一个接收队列, 内核为此定义了一个数据结构 `struct softnet_data`, 它作为 per-CPU 变量被使用。

```
<include/linux/netdevice.h>
```

```
extern int      netif_rx(struct sk_buff *skb);
```

函数有两个返回值 `NET_RX_SUCCESS` 和 `NET_RX_DROP`，如果 `netif_rx` 返回 `NET_RX_DROP`，意味着网络子系统接收队列已经用满，理论上在这种情形下设备驱动程序不应再调用 `netif_rx`，但是大多数的网络设备驱动程序并不关注这里的返回值，因为既然无法阻止数据包的到来（这是个异步的过程），那么与其关闭网卡的接收功能，倒不如直接让上层去处理更安全，所以也就很少有驱动程序去关心 `netif_rx` 的返回值了。

最后，在结束本节的讨论前，我们也给出一个网络设备驱动程序中接收数据包的示例代码，以使得读者建立一个具体的印象，下面的代码片段依然来自于某 ARM 平台，对它的调用出现在一个中断处理例程中：

```
static void
fec_enet_rx(struct net_device *dev)
{
    struct fec_enet_private *fep;
    volatile fec_t      *fecp;
    volatile cbd_t *bdp;
    unsigned short status;
    struct sk_buff      *skb;
    ushort      pkt_len;
    __u8 *data;
    int      rx_index ;

    ...

    fep = netdev_priv(dev);
    fecp = (volatile fec_t*)dev->base_addr;

    spin_lock_irq(&fep->hw_lock);

    /* First, grab all of the stats for the incoming packet.
     * These get messed up if we get called due to a busy condition.
     */
    bdp = fep->cur_rx;

    while (!((status = bdp->cbd_sc) & BD_ENET_RX_EMPTY)) {
        rx_index = bdp - fep->rx_bd_base;
        if (!fep->opened)
            goto rx_processing_done;

        /* Check for errors. */
        if (status & (BD_ENET_RX_LG | BD_ENET_RX_SH | BD_ENET_RX_NO |
                     BD_ENET_RX_CR | BD_ENET_RX_OV)) {
            dev->stats.rx_errors++;
        }
    }
}
```

```

        if (status & (BD_ENET_RX_LG | BD_ENET_RX_SH)) {
            /* Frame too long or too short. */
            dev->stats.rx_length_errors++;
        }
        if (status & BD_ENET_RX_NO) /* Frame alignment */
            dev->stats.rx_frame_errors++;
        if (status & BD_ENET_RX_CR) /* CRC Error */
            dev->stats.rx_crc_errors++;
        if (status & BD_ENET_RX_OV) /* FIFO overrun */
            dev->stats.rx_fifo_errors++;
    }

    /* Report late collisions as a frame error.
     * On this error, the BD is closed, but we don't know what we
     * have in the buffer. So, just drop this frame on the floor.
     */
    if (status & BD_ENET_RX_CL) {
        dev->stats.rx_errors++;
        dev->stats.rx_frame_errors++;
        goto rx_processing_done;
    }

    /* Process the incoming frame.
     */
    dev->stats.rx_packets++;
    pkt_len = bdp->cbd_datlen;
    dev->stats.rx_bytes += pkt_len;
    data = (__u8*)__va(bdp->cbd_bufaddr);
    fec_dcache_inv_range(data, data+pkt_len-4);

    skb = dev_alloc_skb(FEC_ENET_RX_FRSIZE);

    struct sk_buff *pskb = fep->rx_skbuff[rx_index];
    fep->rx_skbuff[rx_index] = skb;
    skb->data = FEC_ADDR_ALIGNMENT(skb->data);
    bdp->cbd_bufaddr = __pa(skb->data);
    skb_put(pskb, pkt_len-4); /* Make room */
    skb = pskb;
    skb->protocol = eth_type_trans(skb, dev);
    netif_rx(skb);
rx_processing_done:

    status &= ~BD_ENET_RX_STATS;
    status |= BD_ENET_RX_EMPTY;
    bdp->cbd_sc = status;

```

```

        /* Update BD pointer to next entry.
        */
        if (status & BD_ENET_RX_WRAP)
            bdp = fep->rx_bd_base;
        else
            bdp++;
        ...
    } /* while (!((status = bdp->cbd_sc) & BD_ENET_RX_EMPTY)) */

    fep->cur_rx = (cbd_t *)bdp;
    spin_unlock_irq(&fep->hw_lock);
}

```

出于篇幅的原因，对该函数的代码进行了一定的删减，保留了最能体现接收函数特点的部分。接收数据包依然使用了 DMA 传输的方式，也是典型的流式 DMA 映射，同时为了确保 CPU 能获得正确的接收数据包，在建立流式 DMA 缓冲区前调用了 `fec_dcache_inv_range` 以确保 DMA 通道的目标地址所对应的 cache 无效，这样当 DMA 传输完成，CPU 从主存中读取接收到的数据包时将不会只读取到 cache 中的数据。另外，我们注意到网络设备驱动程序的接收函数需要负责为本次接收到的数据包分配套接字缓冲区 `skb`，这是通过 `dev_alloc_skb` 函数来完成的，当接收的数据包通过 DMA 通道成功地由设备内存传输到主存中时，函数调用 `netif_rx(skb)` 将该数据包传递给网络子系统的高层代码。

12.4 套接字缓冲区

对于套接字缓冲区 `struct sk_buff *skb` 读者现在也许并不陌生，前面已经看到了对它的使用，只不过到目前为止还没有对它详细讨论而已。从软件层面的角度，`skb` 在网络协议各层之间流动，起着沟通各层间互动的类似桥梁作用。我们在此处讨论它，则更多是从设备驱动程序的角度出发，了解其一些重要成员的作用，以及内核为操作该数据对象所提供的一些接口函数，如此读者才能在实际的网络设备驱动程序的编写中娴熟地使用对应的各种函数来操作 `skb`。另外如果读者对 Linux 内核中网络子系统的高层代码感兴趣，也需要此处讨论的内容，因为对 `skb` 的使用会频繁出现在网络组件的大部分关键场合。

下面是经过适当精简后的 `struct sk_buff` 数据结构在内核源码中的定义：

```

<include/linux/skbuff.h>
-----
struct sk_buff {
    /* These two members must be first. */
    struct sk_buff    *next;
    struct sk_buff    *prev;

    ktime_t            tstamp;

```

```

struct sock      *sk;
struct net_device *dev;

/*
 * This is the control buffer. It is free to use for every
 * layer. Please put your private variables there. If you
 * want to keep them across layers you have to do a skb_clone()
 * first. This is owned by whoever has the skb queued ATM.
 */
char              cb[48] __aligned(8);
unsigned long      _skb_refdst;
unsigned int       len, data_len;
__u16              mac_len, hdr_len;
union {
    __wsum          csum;
    struct {
        __u16       csum_start;
        __u16       csum_offset;
    };
};
__u32              priority;
kmemcheck_bitfield_begin(flags1);
__u8               local_df:1,
                  cloned:1,
                  ip_summed:2,
                  nohdr:1,
                  nfctinfo:3;
__u8               pkt_type:3,
                  fclone:2,
                  ipvs_property:1,
                  peeked:1,
                  nf_trace:1;
kmemcheck_bitfield_end(flags1);
__be16             protocol;

void              (*destructor)(struct sk_buff *skb);
int               skb_iif;
__u32             rxhash;

kmemcheck_bitfield_begin(flags2);
__u16             queue_mapping:16;
kmemcheck_bitfield_end(flags2);

/* 0/14 bit hole */
union {

```

```

        __u32                mark;
        __u32                dropcount;
    };

    __u16                    vlan_tci;

    sk_buff_data_t           transport_header;
    sk_buff_data_t           network_header;
    sk_buff_data_t           mac_header;
    /* These elements must be at the end, see alloc_skb() for details. */
    sk_buff_data_t           tail;
    sk_buff_data_t           end;
    unsigned char             *head, *data;
    unsigned int              truesize;
    atomic_t                  users;
};

```

网络设备驱动程序中经常要使用到的一些成员如下：

`struct net_device *dev`

当前用于发送和接收该套接字缓冲区的网络设备对象。

`sk_buff_data_t transport_header`

对应网络传输层协议头部数据的地址。

`sk_buff_data_t network_header`

对应网络层协议头部数据的地址。

`sk_buff_data_t mac_header`

对应网络 MAC 层协议头部数据的地址。

`sk_buff_data_t tail`

`sk_buff_data_t end`

`unsigned char *head, *data`

指向套接字缓冲区中数据的指针。其中，`head` 指向一个已分配空间的头部，`end` 指向该空间的尾部；`data` 指向这部分空间中有效数据的头部，`tail` 指向该有效数据的尾部。当一个套接字缓冲区在网络各协议层间交互流动时，`head` 和 `end` 这两个值是不变的，而 `data` 和 `tail` 则会在各层中由相应的模块根据需要进行修改，以容纳或者剥离对应的有效数据。因此，上述四个值其实是指向同一内存块的不同位置，后者所在的内存区域由 `__alloc_skb` 函数负责分配。通过改变指针位置而不是数据拷贝或者移动的方式来管理套接字缓冲区中的数据，

可以提升操作效率。

unsigned int len, data_len

len 是该套接字缓冲区中全部数据的长度, 包括上述 data 指向的数据和 end 后面分片数据的总长, 而 data_len 只是分片数据段的长度。

unsigned int truesize

该成员变量表示 sk_buff 所在空间加数据区的大小, 可以简单认为 truesize = sizeof(sk_buff) + len。

atomic_t users

缓冲区当前的引用计数, 用以决定是否释放该缓冲区。如果该值不为 1, 表明尚有模块在使用这片缓冲区, 出于安全方面的考虑, 系统此时将不会释放掉它。

以上简单介绍了 sk_buff 中一些常见成员变量的作用, 接下来讨论内核提供的一些操作 sk_buff 的函数:

○ alloc_skb

用来分配一个套接字缓冲区 skb 及其所对应的数据区 data, 函数原型为:

```
<include/linux/skbuff.h>
-----
struct sk_buff *alloc_skb(unsigned int size, gfp_t priority)
```

参数 size 表示当前要分配的套接字缓冲区所对应的数据区的大小。函数的内部通过调用 __alloc_skb 来做实际的内存分配, __alloc_skb 函数的核心实现为:

```
<net/core/skbuff.c>
-----
struct sk_buff *__alloc_skb(unsigned int size, gfp_t gfp_mask,
                           int fclone, int node)
{
    struct kmem_cache *cache;
    struct skb_shared_info *shinfo;
    struct sk_buff *skb;
    u8 *data;

    cache = fclone ? skbuff_fclone_cache : skbuff_head_cache;

    /* Get the HEAD */
    skb = kmem_cache_alloc_node(cache, gfp_mask & ~__GFP_DMA, node);
    ...
    size = SKB_DATA_ALIGN(size);
    data = kmalloc_node_track_caller(size + sizeof(struct skb_shared_info),
```



```

        gfp_mask, node);
    ...

    /*
     * Only clear those fields we need to clear, not those that we will
     * actually initialise below. Hence, don't put any more fields after
     * the tail pointer in struct sk_buff!
     */
    memset(skb, 0, offsetof(struct sk_buff, tail));
    skb->truesize = size + sizeof(struct sk_buff);
    atomic_set(&skb->users, 1);
    skb->head = data;
    skb->data = data;
    skb_reset_tail_pointer(skb);
    skb->end = skb->tail + size;
    ...
    return skb;
}

```

因为 `sk_buff` 在 Linux 网络子系统中分配和释放的频率非常高，所以内核采用 `kmem_cache` 的内存分配方式来分配 `sk_buff` 的空间，了解这种内存分配方式的读者一定会猜到在系统初始化期间会有对 `kmem_cache_create` 的调用来产生这里的 `skbuff_fclone_cache` 和 `skbuff_head_cache` 两个全局变量。事实的确如此，在 Linux 系统初始化期间，通过调用链 `sock_init()→skb_init()` 来产生这两个变量：

<net/core/skbuff.c>

```

void __init skb_init(void)
{
    skbuff_head_cache = kmem_cache_create("skbuff_head_cache",
                                          sizeof(struct sk_buff),
                                          0,
                                          SLAB_HWCACHE_ALIGN|SLAB_PANIC,
                                          NULL);

    skbuff_fclone_cache = kmem_cache_create("skbuff_fclone_cache",
                                             (2*sizeof(struct sk_buff)) +
                                             sizeof(atomic_t),
                                             0,
                                             SLAB_HWCACHE_ALIGN|SLAB_PANIC,
                                             NULL);
}

```

`__alloc_skb` 函数中接下来的一个重要步骤是调用 `kmalloc_node_track_caller` 来分配 `sk_buff` 中的数据空间 `data`，`kmalloc_node_track_caller` 函数最终调用 `__kmalloc` 来为套接字缓冲区 `sk_buff` 的数据区分配空间，所以它们在物理地址空间是连续的，了解这点对网络设备驱动程序中正确使用 DMA 操作很重要。函数的最后是对分配后的 `sk_buff` 对象进行必要的初始

化，此处为了便于读者理解，我们将 alloc_skb 分配出来的 skb 和 data 空间的关联用图 12-7 来表示⁶：

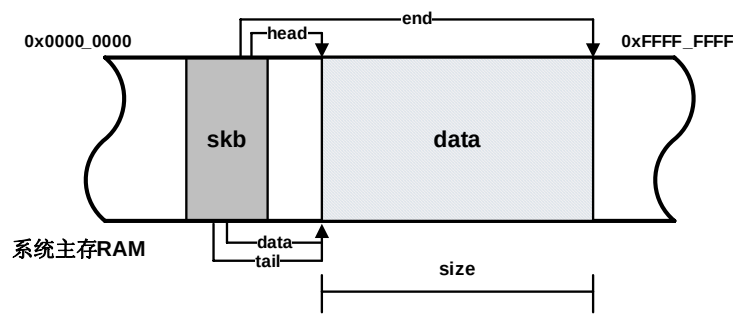


图 12-7 alloc_skb 分配出的 skb 与 data 空间示意图

当一个套接字缓冲区对象 skb 在网络子系统的各协议层之间流动时，各层通过改变 skb->data 和 skb->tail 的值来获得当前层对应的协议数据首地址，而无须显式地进行内存复制等操作，图 12-8 展示了一个 skb 所对应的数据包从 TCP 层传递到 MAC 层时各协议层所对应的 skb->data 值的变化，如果是接收数据包，则这个过程正好相反：

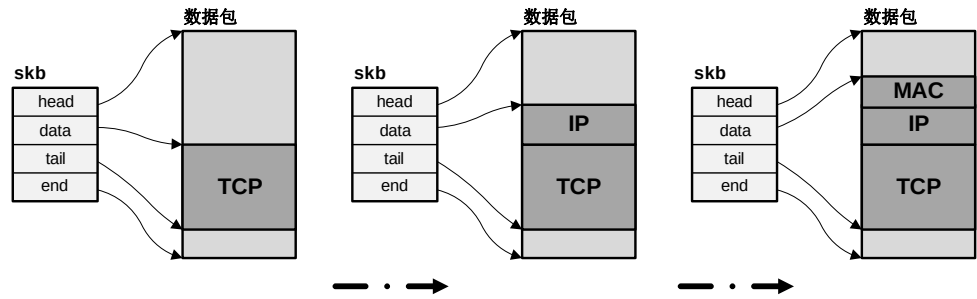


图 12-8 skb 所在的数据包依次通过 TCP、IP 和 MAC 层时 data 指针的变化

为了便于各协议层操作，在 skb 中的 transport_header、network_header 和 mac_header 成员分别等同于网络数据包在 TCP 层、IP 层和 MAC 层时的 skb->data 值。

○ dev_alloc_skb

函数原型如下：

```
struct sk_buff *dev_alloc_skb(unsigned int length)
```

该函数也用来分配一个套接字缓冲区和数据区，其最终的分配工作是通过调用 alloc_skb 来完成的，但与 alloc_skb 不同的是，dev_alloc_skb 在分配内存时会使用 GFP_ATOMIC 标志，

⁶ alloc_skb 实际分配出的 data 空间比图中的 size 要大一些，因为在 data 底部会有一部分空间用来容纳 struct skb_shared_info 对象，但对驱动程序而言，不必理会这一部分额外空间。

同时会在分配出的数据区头部保留一段大小为 NET_SKB_PAD 的空间供网络层优化使用，驱动程序不会使用到该保留区域。因此，如果驱动程序需要在中断上下文中分配一个套接字缓冲区，应该使用 dev_alloc_skb。

○ kfree_skb、dev_kfree_skb、dev_kfree_skb_irq 与 dev_kfree_skb_any

这四个函数都用来释放一个套接字缓冲区 skb 及其所对应的数据空间，其函数原型实质上是一样的：

```
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
void dev_kfree_skb_irq(struct sk_buff *skb);
void dev_kfree_skb_any(struct sk_buff *skb);
```

其中 kfree_skb 与 dev_kfree_skb 在本质上是等价的，都是通过 __kfree_skb 来释放 skb 及其对应的数据区所占有的内存空间。__kfree_skb 的主要操作分为两部分：一是调用 kfree 函数释放 skb 所对应的数据空间；二是通过 kmem_cache_free 来释放 skb 对象所占据的空间。

后两个释放函数可以看做是 dev_kfree_skb 的变体，其中 dev_kfree_skb_irq 用在中断上下文中或者硬件中断关闭的情况下，因为作为一个通用的规则，在硬件中断上下文中，代码的执行时间应尽可能短，而如果硬件中断关闭，为了防止可能的中断丢失也应尽快完成当前的操作。我们可以看看 dev_kfree_skb_irq 在内核中的源码来理解为什么在硬件中断上下文中环境下释放一个 skb 应该使用 dev_kfree_skb_irq：

<net/core/dev.c>

```
void dev_kfree_skb_irq(struct sk_buff *skb)
{
    if (atomic_dec_and_test(&skb->users)) {
        struct softnet_data *sd;
        unsigned long flags;
        local_irq_save(flags);
        sd = &__get_cpu_var(softnet_data);
        skb->next = sd->completion_queue;
        sd->completion_queue = skb;
        raise_softirq_irqoff(NET_TX_SOFTIRQ);
        local_irq_restore(flags);
    }
}
```

可见，与 dev_kfree_skb 函数不同，dev_kfree_skb_irq 并不直接在其内部释放 skb，而是在确定当前要释放的 skb 已无其他引用者时，内核才将 skb 放到一个完成队列，然后触发软中断 NET_TX_SOFTIRQ，让该 softirq 去执行 skb 真正的释放操作。这显然要比 dev_kfree_skb 中直接调用 __kfree_skb 函数要快得多。至于 NET_TX_SOFTIRQ 软中断例程如何释放此处的 skb，有兴趣的读者可以去阅读 net_tx_action 函数中的代码。

最后一个函数 `dev_kfree_skb_any` 则更加智能，它可以自动判断当前的执行路径是不是处在硬件中断上下文或者硬件中断关闭的情况下，根据判断的结果决定是调用 `dev_kfree_skb_irq` 还是 `dev_kfree_skb`：

```
<net/core/dev.c>
void dev_kfree_skb_any(struct sk_buff *skb)
{
    if (in_irq() || irqs_disabled())
        dev_kfree_skb_irq(skb);
    else
        dev_kfree_skb(skb);
}
```

○ `skb_put`

函数原型如下：

```
unsigned char *skb_put(struct sk_buff *skb, unsigned int len)
```

该函数通过向后移动 `skb->tail` 从而在原来的 `tail` 和新的 `tail` 之间开辟出一个新的空间。

图 12-9 展示了调用 `skb_put(skb, 64)` 前后 `tail` 指针的变化，可以看到 `skb_put(skb, 64)` 在原来数据块的尾部拓展了一个大小为 64 字节的空间：

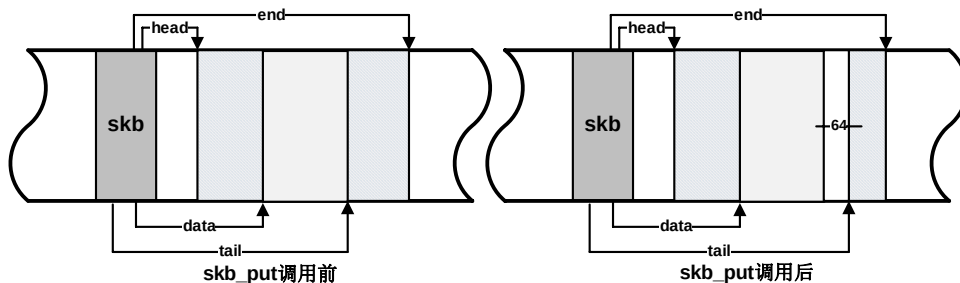


图 12-9 `skb_put(skb, 64)` 调用前后对比

在将 `skb->tail` 移到新的位置后，老的 `tail` 值将作为返回值返回。

○ `skb_push`

函数原型如下：

```
unsigned char *skb_push(struct sk_buff *skb, unsigned int len)
```

与 `skb_put` 相反，该函数将向前移动 `skb->data`，`skb->data -= len`，这样将在原数据块的头部拓展出一个大小为 `len` 的空间，然后将移动后的 `skb->data` 值返回。

○ `skb_headroom`

函数原型如下：

```
unsigned int skb_headroom(const struct sk_buff *skb)
```

返回 `skb->data` 与 `skb->head` 之间的空闲空间大小，也即 `skb->data - skb->head`。

○ `skb_tailroom`

函数原型如下：

```
int skb_tailroom(const struct sk_buff *skb)
```

返回 `skb->tail` 与 `skb->end` 之间的空闲空间大小，也即 `skb->end - skb->tail`。

○ `skb_reserve`

函数原型如下：

```
void skb_reserve(struct sk_buff *skb, int len)
```

该函数同时将 `skb->data` 和 `skb->tail` 增加参数 `len` 指定的字节数，这样将为 `skb` 的 `head` 空间扩展 `len` 个字节，该增加的空间从 `tail` 空间里补充，相当于减少了 `tail` 空间的大小。

当然，内核提供的操作 `skb` 的函数远不止上面列出的这些，其他一些函数因为在网络设备驱动程序中用到的几率并不高，所以本书不再一一列举。

12.5 中断处理

现在几乎所有网卡都支持中断操作模式，通过中断的方式，网卡可以在一个网络分组成功发送出去、成功接收到一个网络分组或者是硬件内部出现错误状态时通知 CPU 进行处理。但是网络设备驱动程序相对于其他一般设备有其自身的特殊性，比如在高负载的情况下，网络设备可能需要接收大量外部进来的数据包，这种密集型的数据包接收对于网络设备驱动程序中的中断处理程序而言是个极大的考验。设备驱动程序实现的中断处理例程必须完成最关键的操作而迅速返回以为下一次的中断到来作好准备，它应该将一些耗时的工作延迟到 `softirq` 中来完成。内核为此定义了两个 `softirq`，分别用于应对发送和接收触发的软中断处理，它们是 `NET_TX_SOFTIRQ` 和 `NET_RX_SOFTIRQ`。不过此处无须深入讨论这两个软件中断的内部机制，因为它们跟驱动程序并没有直接的关联。

网络设备驱动程序员也许更想知道在中断处理函数中一般的处理原则是什么，下面的代码是来自某 ARM 平台上的一个典型的中断处理函数：

```
static irqreturn_t
fec_enet_interrupt(int irq, void * dev_id)
{
```

```

struct net_device *dev = dev_id;
volatile fec_t      *fecp;
uint  int_events;
irqreturn_t ret = IRQ_NONE;

fecp = (volatile fec_t*)dev->base_addr;
/* Get the interrupt events that caused us to be here.*/
do {
    int_events = fecp->fec_ievent;
    fecp->fec_ievent = int_events;

    /* Handle receive event in its own function.*/
    if (int_events & (FEC_ENET_RXF | FEC_ENET_RXB)) {
        ret = IRQ_HANDLED;
        fec_enet_rx(dev);
    }

    /* Transmit OK, or non-fatal error. Update the buffer
       descriptors. FEC handles all errors, we just discover
       them as part of the transmit process.
    */
    if (int_events & (FEC_ENET_TXF | FEC_ENET_TXB)) {
        ret = IRQ_HANDLED;
        fec_enet_tx(dev);
    }

    if (int_events & FEC_ENET_MII) {
        ret = IRQ_HANDLED;
        fec_enet_mii(dev);
    }

} while (int_events);

return ret;
}

```

函数的主体框架是，当中断发生中断处理函数 `fec_enet_interrupt` 被调用时，首先获得硬件设备的状态寄存器的信息，然后根据该信息判断当前中断的类型：是之前的一个数据包被成功发送出去产生的中断，还是硬件接收到了一个数据包，再或者是因为硬件的其他状况（比如内部出现错误）而产生的中断，函数会根据不同的中断类型进行相应的处理。虽然不同的中断处理和手边的硬件密切相关，但是一般而言还是会有一些广泛的通用原则，比如对于接收中断的处理，驱动程序可能需要分配一个 `skb` 缓冲区，然后把接收的网络数据包放到该缓冲区中，之后调用 `netif_rx(skb)` 来触发 `NET_RX_SOFTIRQ` 软中断，而对于发送成功的中断处理则相对比较简单，驱动程序一般只需要更新一些统计量同时负责释放上层代

码传递下来的 `skb` 缓冲区。

读者也许注意到上述 `fec_enet_interrupt` 函数的主体框架建立在一个 `do...while` 循环结构中，这样在一次硬件中断中就可以处理若干个外部进来的数据包，否则每个进入的数据包都会引发一次中断，由此带来的系统开销相当可观。对于一些更高速的设备，为了防止频繁的中断所带来的高额的系统开销，Linux 内核引入了一种所谓的 NAPI 机制。

12.6 NAPI

很明显，内核为网络设备的中断请求设计了精密的处理框架，然而即便如此，对于高速网络设备而言，单纯采用中断驱动的方式，CPU 仍可能在短时间内接收到大量密集的网络数据包，如果每一个进入的数据包都向 CPU 产生一次中断请求，对这些请求单独处理无疑会造成 CPU 资源的浪费甚至导致系统瘫痪。于是在这种情况下，一种被称为 NAPI (New API) 的处理模式被引入到了内核中。

NAPI 的设计思想其实是结合了中断与轮询的各自优势，虽然在设备驱动程序中，轮询的名声不大好，但并非一无是处，比如在 NAPI 的机制中。NAPI 简单地说，就是当有数据包到达时将会触发硬件中断，在中断处理中关闭中断，系统对硬件的掌控将进入轮询模式，直到所有的数据包接收完毕，再重新开启中断，进入下一个中断轮询周期。显然在系统对硬件进行轮询期间，硬件可能会接收到大量进入的数据包，但是它们不会产生中断。

与在一次硬件中断中处理多个数据包不同，内核中提供了对 NAPI 支持的框架，因此如果设备驱动程序需要利用 NAPI 带来的益处，则需要根据内核中实现的 NAPI 机制提供对应的支持。我们不打算详细讨论内核为实现 NAPI 机制的技术细节，此处仅从设备驱动程序的角度讨论如何实现对 NAPI 的支持。

虽然内核提供了对 NAPI 的支持，但需要注意的是，并不是每个网络设备都支持 NAPI 操作，一个支持 NAPI 操作的设备至少应该满足：

- 能让驱动程序关闭分组接收中断而不影响其他的中断，因为 NAPI 主要针对接收分组的操作路径，在内核轮询处理设备接收到的分组时不应该影响其他的中断进入处理器，比如硬件自身的错误等。
- 设备应该能同时保留多个接收到的分组，否则轮询将失去意义，因为当在中断处理中处理一个接收分组时，后续进入的分组将被直接丢弃从而失去轮询的必要。

内核为支持 NAPI 机制定义了一个数据结构 `struct napi_struct`：

```
<include/linux/Netdevice.h>
-----
struct napi_struct {
```

```

    struct list_head poll_list;

    unsigned long state;
    int weight;
    int (*poll)(struct napi_struct *, int);
    ...
    unsigned int gro_count;
    struct net_device *dev;
    struct list_head dev_list;
    struct sk_buff *gro_list;
    struct sk_buff *skb;
};

```

以上数据结构中最重要的成员是 `poll_list`、`weight` 和 `poll`。其中 `poll_list` 用来将当前设备放置到内核维护的一个轮询列表中；`weight` 表明了当前设备的权重（如果某一设备上长时间连续接收到分组，不至于使系统中其他设备失去被轮训的机会），当内核在多个设备之间轮询时，该值用来赋予对一个设备进行轮询处理的时间宽度；`poll` 是一个函数指针，驱动程序需要实现这个函数，它将在内核对当前设备轮询时被调用。

设备驱动程序中需要分配一个 `struct napi_struct` 对象，该对象一般放置在设备驱动程序自己的私有数据区，然后驱动程序需要调用 `netif_napi_add` 来对其初始化，该函数在内核中的实现为：

<net/core/dev.c>

```

void netif_napi_add(struct net_device *dev, struct napi_struct *napi,
                    int (*poll)(struct napi_struct *, int), int weight)
{
    INIT_LIST_HEAD(&napi->poll_list);
    napi->gro_count = 0;
    napi->gro_list = NULL;
    napi->skb = NULL;
    napi->poll = poll;
    napi->weight = weight;
    list_add(&napi->dev_list, &dev->napi_list);
    napi->dev = dev;
    set_bit(NAPI_STATE_SCHED, &napi->state);
}

```

函数主要用于初始化 `struct napi_struct` 的对象指针 `napi`，参数 `poll` 是设备驱动程序中需要实现的内核用来轮询当前设备的函数。

设备驱动程序在 `poll` 中主要负责接收后续到达的网络数据包，当发现所处理的数据包数量低于 `poll` 函数的第二个参数指定的配额时，将调用 `napi_complete` 来退出轮询模式，同时打开设备的接收中断。

设备驱动程序在完成上述任务后就完成了对 NAPI 的支持，假设一个数据包的到来触发了驱动程序的中断处理例程，在那里 `netif_rx` 被调用用来通知网络子系统的高层数据包到达，由此进入的 `netif_rx enqueue_to_backlog` 调用链将把当前设备加到 `struct softnet_data` 对象的 `poll_list` 链表中，之后在接收分组中断的下半部 `NET_RX_SOFTIRQ` 对应的 `net_rx_action` 中将会操作 `struct softnet_data` 对象的 `poll_list` 链表，其中的每个元素代表一个需要轮询的设备，内核将调用该元素的 `poll` 函数，它正是设备驱动程序之前实现的以轮询方式处理当前设备接收到的分组的函数。

12.7 本章小结

本章讨论了 Linux 设备驱动世界的第三类设备——网络设备，因为内核代码中的网络子系统是个极其复杂的结构，所以这些技术细节不是本书要讨论的主题，但是作为讲述网络设备驱动程序内核机制的书籍，在讨论网络设备驱动程序相关主题时将不可避免地要进入网络子系统的某些细节内幕。正是因为网络系统涉及的范围极其宽广，所以本章主要从网络设备的数据结构抽象，网络设备注册以及网络设备实现的方法入手，试图让读者对网络设备驱动程序的内幕细节有个比较深入的理解。

本章中，大量的篇幅集中在网络数据包的发送和接收上，因为这是一个 NIC 设备驱动程序要实现的核心功能。就驱动程序本身而言，发送与接收数据包的功能实现与硬件紧密相关，但一般的流程是，对于发送路径，高层网络代码负责分配 `skb` 以存储网络数据包，然后将该 `skb` 传到驱动程序的发送函数中。驱动程序的发送函数一般会使用 DMA 来将位于主存中的 `skb` 传输到网络设备内存中，再由硬件逻辑发送出去。硬件在成功发送完一个数据帧后，通常会以中断的方式通知处理器，接下来相应驱动程序的中断处理函数被调用来处理这种情况。对于成功发送分组产生的中断，在其中断处理函数中会释放该分组所在的 `skb`。对于接收路径而言，底层网络设备在成功将一个分组传输到主存中后，也会用中断的方式通知驱动程序，后者负责分配一个新的 `skb` 来容纳该新入的分组，然后通过调用 `netif_rx` 函数通知网络子系统高层新数据包到达。

本章也讨论了分组发送过程中的流控机制，其目的是尽量在初期对一个可能不会成功发送出去的分组不使用驱动程序中的发送函数，这样可以避免系统资源的浪费，内核为此提供了 `netif_stop_queue`、`netif_start_queue` 等流控函数供驱动程序使用。

本章最后还讨论了对高速设备引入的 NAPI 机制，这种机制混合了中断与轮询操作模式的优点，可避免对于密集而至的每个分组都产生硬件中断的问题，因为这可能会导致 CPU 资源的大量消耗甚至系统崩溃。但如果驱动程序要使用这种机制，则需要在其内部按照内核中 NAPI 机制的要求实现对应的函数。

深入Linux设备驱动程序 内核机制

本书涵盖编写设备驱动程序所需要的几乎所有的内核设施，比如内核模块、中断处理、互斥与同步、内存分配、延迟操作、时间管理，以及新设备驱动模型等内容。

ISBN: 978-7-121-15052-4

书价: 98.00